# *The Robust Malware Detection Challenge and Greedy Random Accelerated Multi-Bit Search*

# The Robust Malware Detection Challenge and Greedy Random Accelerated Multi-Bit Search

Sicco Verwer
Delft University of Technology
Delft, The Netherlands
s.e.verwer@tudelft.nl

Azqa Nadeem
Delft University of Technology
Delft, The Netherlands
azqa.nadeem@tudelft.nl

Christian Hammerschmidt
Delft University of Technology
Delft, The Netherlands
c.a.hammerschmidt@tudelft.nl

Laurens Bliek
Delft University of Technology
Delft, The Netherlands
l.bliek@tudelft.nl

Abdullah Al-Dujaili
Analog Devices
Norwood, United States
ash.aldujaili@analog.com

Una-May O'Reilly
MIT CSAIL
Cambridge, United States
unamay@csail.mit.edu

## ABSTRACT

Training classifiers that are robust against adversarially modified examples is becoming increasingly important in practice. In the field of malware detection, adversaries modify malicious binary files to seem benign while preserving their malicious behavior. We report on the results of a recently held robust malware detection challenge. There were two tracks in which teams could participate: the *attack track* asked for adversarially modified malware samples and the *defend track* asked for trained neural network classifiers that are robust to such modifications. The teams were unaware of the attacks/defenses they had to detect/evade. Although only 9 teams participated, this unique setting allowed us to make several interesting observations.

We also present the challenge winner: GRAMS, a family of novel techniques to train adversarially robust networks that preserve the intended (malicious) functionality and yield high-quality adversarial samples. These samples are used to iteratively train a robust classifier. We show that our techniques, based on discrete optimization techniques, beat purely gradient-based methods. GRAMS obtained first place in both the attack and defend tracks of the competition.

## CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Adversarial learning**; **Neural networks**; *Discrete space search*; *Randomized search.*

## KEYWORDS

Adversarial Learning; Neural Networks; Robust Malware Detection; Adversarial malware; Discrete optimization; Saddle-point optimization


*AISec'20, November 13, 2020, Virtual Event, USA*

ACM ISBN 978-1-4503-8094-2/20/11…$15.00
https://doi.org/10.1145/3411508.3421374

**ACM Reference Format:**
Sicco Verwer, Azqa Nadeem, Christian Hammerschmidt, Laurens Bliek, Abdullah Al-Dujaili, and Una-May O'Reilly. 2020. The Robust Malware Detection Challenge and Greedy Random Accelerated Multi-Bit Search. In *13th ACM Workshop on Artificial Intelligence and Security (AISec'20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3411508.3421374

## 1 INTRODUCTION

The field of cyber security is an arms race between defenders and attackers. Machine learning, and in particular deep learning, has become an indispensable weapon on the defenders' side, for tasks ranging from spam and phishing detection, anti-virus software, to intrusion detection systems. While adding to the security, the machine-learning systems themselves offer a new attack surface for smart attackers: by carefully modifying their files, adversaries can craft so-called adversarial examples, i.e., variants of their files that evade detection [12, 28].

While adversarial examples have been discussed in literature, most of the attention in the context of neural networks has been paid to classifiers in the continuous domain, such as images and videos [16]. Any small perturbation of a given sample in the continuous space yields a valid data point, and numerous methods of finding minimal changes that fool classifiers have been proposed [9]. It is shown in [1] that these methods can be used to train a robust malware classifier, i.e., a malware detector that is hard to evade. In most cyber security applications, however, and in particular in malware classification, the feature space frequently contains discrete features. Crafting adversarial examples in a discrete domain is more challenging than in a continuous one: many of the possible perturbations can be invalid and although gradients can be computed on a relaxed problem, the information they provide can be incorrect. Moreover, an additional difficulty in the case of malware is that the perturbed examples must not only fool the classifier, but the perturbations must keep their functionality, i.e., not modify or destroy the malicious payload.

Recent work has shown how adversaries can craft adversarial perturbations to malicious code in order to evade malware detectors, e.g. [12, 28]. Yet, the techniques available to obtain these samples are taken from the continuous domain and do not perform very well in the discrete case. In a recently held robust malware detection challenge, the goal was to overcome this limitation and stimulate the

development of methods that can find good discrete perturbations. The competition was organized in two tracks: the attack track where the task was to obtain high-quality adversarial examples that fool classifiers, and the defend track where the task was to train a robust neural network that achieves good classification results in the presence of such examples. In this paper, which is the result of a collaboration between the organizers and the winners of this challenge, we describe the challenge as it was designed by one set of authors (Al-Dujaili and O'Reilly), and the main contribution of the other set of authors (Verwer, Nadeem, Hammerschmidt, and Bliek): greedy random accelerated multi-bit search (GRAMS), a method that finds good quality adversarial examples for neural networks in discrete domains.

GRAMS is a simple but effective greedy search procedure that uses gradient information as a heuristic to find discrete adversarial examples. Using a standard adversarial training procedure, it can be used to train robust neural networks. In the challenge, it obtained first place in both the attack and defend tracks (tied in the latter). We describe the GRAMS algorithm and the way in which we choose the examples and neural network to submit to the challenge.

The challenge was unique in its setup in the sense that the attack submissions were evaluated using the defend submissions and vice versa. The targets were unknown and the goal was to submit perturbed data and a model that works well against any possible adversary. Only 9 teams participated in the challenge, perhaps due to the difficulty of not knowing the target. Still, the unique challenge setup allows us to make several interesting observations. In particular, although attacks are known to transfer to different classifiers [7], they seem not to transfer to different defenses. Moreover, even when using a known defense, not knowing the exact target model has a significant negative effect on evasion performance.

## 2 BACKGROUND AND RELATED WORK

Malware detection is moving away from hand-crafted rule-based approaches and towards machine learning techniques [25]. In this section we focus on malware detection with neural networks (Section 2.1), adversarial machine learning (Section 2.2) and adversarial malware samples (Section 2.3).

### 2.1 Malware Detection using Neural Networks

Neural network methods are increasingly being used for malware detection. For features, one study combines DNN's with random projections [5] and another with two dimensional binary Portable Executable (PE) program features [24]. Research has also been done on a variety of file types, such as Android and PE files [2, 11, 22, 30]. While the specifics can vary greatly, all machine learning approaches to malware detection share the same central vulnerability to Adversarial Examples (AEs).

### 2.2 Adversarial Machine Learning

Finding effective techniques that robustly handle AEs is one focus of adversarial machine learning [3, 14]. An adversarial example is created by making a small change to a data sample $x$ to create $x_{adv} = x + \delta$. If the detector misclassifies $x_{adv}$ despite having correctly classified $x$, then $x_{adv}$ is a successful adversarial example. Adversarial learning, also in the context of malware, goes back decades [27]. In [20], the authors formalize adversarial frameworks and provide a review of existing literature within the framework. The framework defines of a comprehensive set of constraints on available transformations, preserved semantics, robustness to preprocessing, and plausibility. The it provides a good starting point for an overview of different directions taken in research on adversarial learning. There are a variety of techniques that generate AEs for neural networks [9, 26]. One efficient and widely used technique is the fast gradient sign method (FGSM) [9]. This method finds the directions that move the outputs of the neural network the greatest degree and moves the inputs along these directions by small amounts, or perturbations. Because the technique references the detector's parameters, it is known as a white-box attack model [4, 9, 19]. There have been multiple studies focused on advancing model performance against AEs, e.g. [17, 31]. One obvious approach is retraining with the AEs incorporated into the training set. We are attracted to the approach of [16]. It casts model learning as a robust optimization problem with a saddle-point formulation where the outer minimization of detector (defensive) loss is tied to the inner maximization of detector loss (via AEs) [16]. The approach successfully demonstrates robustness against adversarial images by incorporating, while training, AEs generated using projected gradient descent.

### 2.3 Adversarial Malware

Security researchers have generated malware AEs using an array of machine learning approaches such as reinforcement learning, genetic algorithms and supervised learning including neural networks, decision trees and SVMs [2, 6, 10, 12, 22, 24, 28, 29]. These approaches, with the exception of [10] are black box. They assume no knowledge of the detector, though it can be queried for detection decisions. Multiple studies use binary features, typically where each index acts as an indicator to express the presence or absence of an API call, e.g. [23]. One study also includes byte/entropy histogram features [24]. Studies to date have only retrained with AEs.

In [1], methods are introduced that are capable of generating functionally preserved adversarial malware examples in the binary domain. Using the saddle-point formulation, they incorporate the adversarial examples into the training of models that are robust to them. They use 4 different inner maximization methods: Two take a continuous approach, using FGSM with either deterministic or randomized rounding; and the other two take multiple bitwise discrete steps, ascending with gradient information or coordinate-wise.

In this paper, we present an approach that generates functional white-box AEs for binary features while incorporating them into the training of a malware classifier that is robust to AEs.

### 2.4 Related competitions

The Madry group at MIT posed adversarial robustness challenges for MNIST and CIFAR[1], inviting attacks on robust networks they had designed. To the best of our knowledge, the malware detection challenge described in this paper is the first competition of its design, where challenges on each end of the spectrum (attack and defense) are motivated to do their best and then face each out.

[1] https://github.com/MadryLab/mnist_challenge

# 3 CONTEXT

## 3.1 Notation and Saddle-Point Formulation

We follow the notation in [1]. The data distribution $\mathcal{D}$ contains tuples of binary representations of executable files and their corresponding labels. The label is in binary format with classes *benign* and *malicious*. The datapoints are drawn from the distribution denoted by $\mathcal{X}$ and the associated label space by $\mathcal{Y}$. Each datapoint $x = [x_1, \ldots, x_n] \in \mathcal{X}$ consists of static features extracted from the executable files, where $x_j$ is a binary indicator showing the presence of the $j^{th}$ feature, and $n$ is the length of the feature vector, therefore $\mathcal{X} = \{0,1\}^n$. We represent the elements of the label space $y \in \mathcal{Y}$ by $\{0,1\}$. We denote the parameters of the classifier over $\mathcal{D}$ by $\theta \in \mathbb{R}^p$. The goal is to find the optimal model parameters $\theta^*$ such that for a given scalar loss function $L(\theta, x, y)$ the empirical risk $\mathbb{E}_{(x,y)\ \mathcal{D}}[L(\theta, x, y)]$ is minimized:

$$\theta^* \in \arg\min_{\theta \in \mathbb{R}^p} \mathbb{E}_{(x,y)\ \mathcal{D}}[L(\theta, x, y)]. \tag{1}$$

The trained model obtained via (1) can be exploited by an adversary with crafted samples intended to be misclassified. Such a crafted example $x_{adv}$, can be obtained by modifying an existing sample $x$ such that $x_{adv}$ maximizes the loss $L$ of the classifier. The modifications must preserve the intended functionality of the sample $x$. Let $\mathcal{S}(x)$ denote the set of samples around $x$ that preserve the functionality. Then the set $\mathcal{S}^*(x) \subseteq \mathcal{S}(x)$ of samples that maximizes the classifier loss is described as:

$$x_{adv} \in \mathcal{S}^*(x) = \arg\max_{\tilde{x} \in S(X)} L(\theta, \tilde{x}, y). \tag{2}$$

As outlined in [1], following [16], the samples obtained via (2) need to be incorporated into the training process, given by (1), to harden the model. The resulting problem becomes:

$$\theta^* \in \underbrace{\arg\min_{\theta \in \mathbb{R}} \mathbb{E}_{(x,y)\ \mathcal{D}} \Big[ \overbrace{max_{\tilde{x} \in \mathcal{S}(x)} L(\theta, \tilde{x}, y)}^{\text{adversarial loss}} \Big]}_{\text{adversarial learning}}. \tag{3}$$

## 3.2 The Challenge Setup

The bulk of adversarial machine learning research has been focused on crafting attacks and defenses for image classification. This challenge puts adversarial machine learning in the context of robust malware detection. In the era of modern cyber warfare, cyber adversaries craft adversarial malicious code that can evade malware detectors [28]. The problem of crafting adversarial examples in the malware classification domain is more challenging compared to image classification: malware adversarial examples must not only fool the classifier, they must also ensure that their adversarial perturbations do not alter the malicious payload. The gist for this challenge is to defend against adversarial attacks by building robust detectors and/or attack robust malware detectors based on binary indicators of imported functions used by the malware. The challenge has two tracks:

(1) *Defend Track*: Build high-accuracy deep models that are robust to adversarial attacks: Participants in this track are required to construct robust models given the defend dataset.
(2) *Attack Track*: Craft adversarial malicious PEs that evade detection on adversarially trained models.

For evaluation, the model's performance on a test set of benign and malicious (and adversarial versions of them) PEs will be assessed. Participants' solutions will be evaluated based on their F1 score against their strongest adversaries.

The competition was organized by some of the authors (Al-Dujaili and O'Reilly) The organizers had the following goals:

(1) Promote the visibility of discrete and constrained versions of adversarial machine learning.
(2) Increase knowledge of the robust malware detection problem where identifying adversarial attacks and developing more robust detectors in the face of adversarial examples are crucial.
(3) Encourage novel and improved algorithms by presenting a readily available dataset and providing a novel setup where adversaries independently tune their approaches then face each other without knowing the other side's approach.

Only neural network solutions were assumed given their popular use in adversarial ML. The competition assumes no access to the underlying source code that generated the binaries. With these assumptions, it is impossible to unset a bit and ascertain if it damages the malware (or functionality of any code in which it is embedded). Therefore only bit setting perturbations were allowed.

*3.2.1 Dataset.* The dataset contains 34,200 files in the Portable Executable (PE) format of Windows executables. 30,400 are used to train a robust model in the competition and the remaining 3800 files are used as a starting point to generate adversarial samples. Each file is represented by a binary vector indicating whether the file includes a specific Windows API call. For each of these files, a large binary vector $x \in \mathcal{X} = \{0,1\}^{22761}$ is constructed using the LIEF binary analysis tool[2]. Overall, there are 22,761 API calls present in the dataset. A detailed description of the data can be found in the repository[3].

The challenge aims to mimic a realistic situation where an attacker modifies the input provided to a learned classifier that aims to detect it. The classifier is learned from 30,400 PEs containing 15,200 malicious PEs and 15,200 benign PEs. The assumption is that malware and goodware use different kinds of system calls to perform their tasks. As shown in Figure 1, this assumption seems to be true. There are several system calls that are present more frequently in malicious PEs than in benign PEs, and vice versa. When learning a classifier, such as a neural network, from this data, we indeed see an accuracy of 90% or even greater on a hold out test set.

*3.2.2 Threat Model.* The challenge considers a threat model where an attacker is able to modify an additional set of 3800 malicious PEs by adding additional system calls, i.e., by changing 0s into 1s in the large binary vectors. In contrast to removing system calls (changing 1s into 0s), this should not modify the working of a piece of malware. When allowing adversarial modifications that change 0s into 1s in malicious data points, the percentage of correctly detected malware

[2] https://lief.quarkslab.com
[3] https://github.com/ALFA-group/malware_challenge/blob/master/docs/challenge.pdf

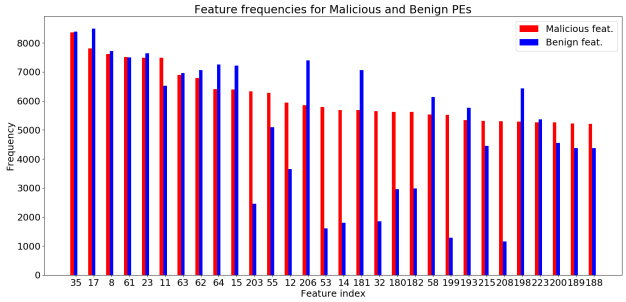


**Figure 1: The occurrence frequencies of various system calls in the binary feature vectors of malicious (mal - red/left) and benign (ben - blue/right) PEs. Sorted w.r.t malicious PEs.**

samples drops to single figures. The goal of the challenge is two-fold. Firstly, to design an attack algorithm that modifies a provided sample of malicious files such that it avoids detection. Secondly, to design a learning algorithm that detects malicious data points, even when modified using an attack algorithm. What made these tasks particularly challenging is that the attack solutions are evaluated using the defend submissions and vice versa. We thus have to make an attacker without knowing the defender, and a defender without knowing the attacker.

## 4 GRAMS

For the challenge, we developed GRAMS (Greedy Random Accelerated Multi-bit Search) as an attacker and constructed a defense in a standard adversarial training setting. The challenge provided a framework for performing this type of training using neural networks within the PyTorch framework. For the defense, we used this approach without modification. For the attack, the key ingredient is a method for the so-called inner maximizer. This method is called for every batch of data rows and repeated for every epoch during adversarial training. This method should thus be able to quickly find high quality modifications, resulting in a large loss for the neural network.

During the robust malware detection competition, we tried out several variants of this algorithm and picked two to submit as solutions, one for attack, one for defense. Here we explain the ideas and inner working of these methods. In the next section, we show how the submission decision was made.

### 4.1 Multi-bit Gradient Descent

The core component in GRAMS is the manner in which it performs a greedy search over the large (22,761 bit) binary search space. Starting from a malicious data point $m$, the goal of this search is to find a modified point $m^* \in A(m)$, where $A()$ returns the set of allowed modifications, such that the loss $L(M, m^*)$ given the current detection model $M$ is maximized.

Two traditional approaches, also implemented in the challenge framework, is to perform a standard gradient descent (such as a Fast Gradient Sign Method (FGSM) [9]) or the multi-step Bitwise Coordinate Ascent [10] over a continuous relaxation of the binary

**Algorithm 1** GRAMS - topk variant

**Require:** a batch $b$ and a neural network model $M$

1: $best\_x := b$, $orig\_x :=$ DATA_VALUES($b$)
2: $k := 8$
3: **while** $k > \frac{1}{2}$ **do**
4: $loss :=$ LOSS($M, x$)
5: $grad :=$ AUTOGRAD($loss, x$)
6: $sign :=$ SIGNS($grad$)
7: $grad :=$ ABSOLUTE($grad - orig\_x * grad$)
8: $x' := x +$ TOPK($grad, k$) $* sign$
9: $loss :=$ LOSS($M, x'$)
10: $loss' :=$ LOSS($M, best\_x$)
11: **if** a row $r$ in $x$ with $loss[r] > loss'[r]$ exists **then**
12: **for all** such rows $r$ **do**
13: $best\_x[r] = x'[r]$
14: **end for**
15: $x := x'$
16: $k := 2k$
17: **else**
18: $k := \frac{1}{2}k$
19: **end if**
20: **end while**
21: **return** $best\_x$

search space. For the former, the found relaxed solution can then be transformed back into a binary data point using some form of (randomized) rounding. The main problem of this approach is that the loss of the relaxed solution may be a bad approximation of the loss of the discrete data point. We overcome this by searching only discrete (binary) data points. The latter only flips a single bit at each step, corresponding to the coordinate with the largest partial derivative of the loss. While this has shown to work for Android malware [10], it only improves slowly in high-dimensional feature spaces.

A naive translation of gradient descent to binary space would be to iteratively try all possible bit flips (the local neighborhood), compute the loss, and utilize the one with the greatest loss. However, this method is too slow. There are many possible single bit flips (potentially 22,761), and finding a local optimum sometimes requires hundreds of bit flips (i.e., many iterations of this algorithm). To avoid this problem, we flip $k$ bits at once. Once again, it is clearly infeasible to try all possible multi-bit flips of size $k$ (there are approx. $22{,}761^k$ possibilities). *The key idea exploited by GRAMS is to use the gradient from the relaxed problem (which can be computed efficiently) as a heuristic indicating which bits to flip.*

As outlined in Algorithm 1 (lines 4-9), in every iteration, GRAMS flips the $k$ bits that have the largest absolute gradient, computed using a traditional forward-backward pass through the neural network. This functionality is efficiently implemented in PyTorch's LOSS, AUTOGRAD, SIGNS, SIGNS ABS, and TOPK functions. This gradient information does not take the problem constraints into account, and hence can flip 1s into 0s. We opted to solve this problem by restricting the flips to valid ones that satisfy the problem constraint. To be more precise, given a computed gradient $g$ for all bits, we subtract $g * x$ from $g$ (line 7), where $x$ contains the original 1s present

in the malicious data point (*orig_x* in Algorithm 1). We thus always perform $k$ valid bit flips. This process can flip bits back and forth and run in circles, which we solve by deciding on the number $k$ in a way that accelerates the search process.

## 4.2 Search acceleration

How many bits to flip in every iteration? This is a complex question. The information in the gradient contains some information that can be used to make smart decisions (e.g., only flip bits with large gradients). Inspired by accelerated gradient descent [18, 21] and other adaptive schemes, like step decay for the learning rate [8] and successive halving for other hyperparameters [13], we decided on the approach outlined below.

GRAMS starts with an initial value for $k = k_0$ (the default value is 8, line 2) and updates it according to the following equation (line 15-18):

$$k_{t+1} = \begin{cases} 2k_t & \text{if } L(m_{\text{new}}) > L(m_{\text{old}}) \\ \frac{1}{2}k_t & \text{otherwise} \end{cases}$$

where $k_i$ is the $i$th value for $k$, $m_{\text{new}}/m_{\text{old}}$ is the new/old value of the modified malicious data point, and $L(m)$ is the loss for data point $m$. Thus, GRAMS exponentially increases/decreases the value of $k$ depending on whether it finds better/worse solutions. Similarly, it updates the malicious data point depending on:

$$m_{\text{new}} = \begin{cases} m_{\text{new}} \text{ if } L(m_{\text{new}}) > L(m_{\text{old}}) \\ m_{\text{old}} \text{ otherwise} \end{cases}$$

Thus, GRAMS only accepts modifications that improve the objective function (increase the loss). A consequence of the above two update equations is that GRAMS frequently finds large values of $k$ for which the solution is worse than the old one. By halving $k$, it aims to find the largest value of $k$ that still improves the objective. Instead of searching for the optimal value of $k$, which would make every iteration take slightly longer, we simply perform an improvement the moment it is found. When this process reaches a local minimum, i.e., when the top 1 bit flip does not result in an improved objective, the search process is ended. Because of the doubling of $k$, this point is typically reached quickly. There is no need to limit the amount of iterations of the algorithm to a predefined constant, as is the case for the existing approaches implemented in the challenge framework.

## 4.3 Learning from batches

The above two algorithmic techniques are applicable in any domain where one aims to perform gradient descent in a discrete search space. In GRAMS, we included one additional trick that has to do with the challenge framework and its implementation in PyTorch. Because the method for learning the neural network operates in batches (in the challenge, size is set to 8), the inner maximization problem that GRAMS solves also receives a batch of malicious data rows.

All the provided inner maximizer algorithms perform some form of gradient descent on this entire batch at once. This is effective because the functions used from PyTorch operate efficiently on tensors. Hence, the size of the batch matters but it is much more expensive to process batches than to process individual data rows. At the same time, we would want to process individual rows because the gradients for the individual rows are all independent from each other. In GRAMS, we opted again for a pragmatic solution where we retain the efficiency of PyTorch's tensor-based functions (including TOPK), while still trying to optimize every individual row.

The solution is a simple trick (lines 11-14). GRAMS stores the best objective (largest loss) found for every row in a batch, along with the individual modifications achieving this objective. At the end, GRAMS returns the composition of these best modifications, which it may never have encountered during the search. In our experience, this gives the highest quality solutions in the limited run-time available for processing a single batch. There are, however, some effects of this solution that would require further study. For instance, in every iteration, the value of $k$ is the same for all rows in a batch. Moreover, we increase $k$ when any of the rows in a batch improves, and modify only the ones that do. Consequently, we increase $k$ even for the rows that did not improve.

**Algorithm 2** GRAMS - topk+ variant

1: **given** a batch $b$, a neural network model $M$
2: $x$ := GRAMS($b, M$)
3: $best\_x := x$
4: $no\_improve := 0$
5: **while** $no\_improve < 10$ **do**
6: get gradient $grad$ from $M$ and $x$, see as Algorithm 1
7: $grad$ := RANDOM(SIZE($grad$)) $* grad$
8: $x' := x +$ TOPK($grad$, RANDOM() $* 20 + 1) * sign$
9: $loss$ := LOSS($M, x'$), $loss'$ := LOSS($M, best\_x'$)
10: **if** a row $r$ exists in $x$ with $loss[r] > loss'[r]$ **then**
11: $no\_improve := 0$
12: **else**
13: $no\_improve := no\_improve + 1$
14: **end if**
15: update rows of $best\_x$ using $x'$, see Algorithm 1
16: **end while**
17: **return** GRAMS($best_x, M$)

## 4.4 GRAMS Variants

We build 3 variants that build on the above algorithmic techniques that differ in the way they decide on starting points for the search process and what to do upon reaching a local minimum.

*4.4.1 Plain GRAMS - topk.* The first method we evaluated is a plain implementation of GRAMS as outlined above (denoted topk in the results). See Algorithm 1. The search is not randomized in any way. The only reason why we obtain different adversarial examples in different learning epochs is due to the change in the neural network model $M$ that we are trying to evade.

*4.4.2 Random jumps from local minima - topk+.* The second method (denoted topk+ in the results) build on the plain GRAMS method by adding randomized jumps to escape from local minima, see Algorithm 2. For initialization, (Algorithm 2, line 2) it calls the plain GRAMS method until it converges. It is now likely stuck in a local minimum. To escape, it runs a heavily randomized variant of GRAMS where it adds one line of code (line 7) that randomizes the gradient information by multiplying every number with a random value. It then flips the top $k$ bits with the largest gradient, where $k$

**Algorithm 3** GRAMS - topkr variant

1: **given** a batch $b$, a neural network model $M$, a number of repeats $n$
2: $x$ := GRAMS($b, M$)
3: $best\_x := x$
4: **for** $i$ in 1 to $N$ **do**
5: $x$ := RANDOM_START($b$)
6: $x'$ := GRAMS($x, M$)
7: update rows of $best\_x$ using $x'$, see Algorithm 1
8: **end for**
9: **return** $best_x$

is a random value between 1 and 20 (line 8). It continuously performs such random jumps but keeps track of the number of times no improvement has been found for any row in the current batch (lines 15-19). When this number is greater than 10 (line 5), the solution is thought to be sufficiently far from the local minimum and GRAMS is called one more time to converge to a new, hopefully better, minimum.

In our submission we ran this randomized jump process only once. It is of course possible to keep running until $best\_x$ stops improving for some time. In our experience in the challenge, running it once is beneficial, but running it multiple time adds too much overhead for the obtained loss increase. In Algorithm 2 there are some parameters that determine the amount of changes to make and improvements to try. For the challenge, we fixed these to reasonable values (i.e. the algorithm did not run for too long, but did find improvements).

*4.4.3 Benign randomization - topkr.* Our final method described in Algorithm 3, changes the initial point from which to run the GRAMS optimizer. This point can be chosen at random, keeping the 1s in the original malicious data point intact, but we found that this gave poor results. Instead, we used the existing benign data points to estimate the parameters of a simple multivariate Bernoulli distribution. This gives a random variable where each bit $k$ is 1 or 0 with probability $p_k$ and $1 - p_k$ respectively. Here, $p_k$ is the average value of bit $k$ over all benign training samples. We then sample from the obtained distribution and keep the samples that are classified as benign by already trained models. We generated 20000 samples this way and then kept the samples that were classified as benign by at least three out of five adversarially trained models (using the above GRAMS variants), giving 3800 samples in total. These were used to generate random restarts. This filtering step was necessary to remove data points that were already being considered as malicious. See Figure 2. As can be seen in Algorithm 3, we perform restarts from these samples a fixed number of times (we used 10 in the challenge submission), and return the best solution found.

## 5 IMPLEMENTATION AND EVALUATION

We implemented our approach in the framework[4] provided by the competition organizers. To compare the three GRAMS variants, we performed local experiments using the provided samples and implementations of randomized (rFGSM) [9] and a bit-wise gradient ascend (BGA) [1]. In the end, we had to make a selection of which method to submit to the attack track, and which to submit to the defend track. The two selected methods (one model and one attack data set) were evaluated against all other participants in the competition.

### 5.1 Local evaluations

Our first evaluation of the different attack methods uses the Self Organizing Map (SOM) provided by the challenge framework. We adversarially trained a classifier using topk and visualized the malicious, benign, and adversarially perturbed data points in 2 dimensions (see Figure 3). Although it is hard to draw solid conclusions from a SOM visualization, we make some observations. Firstly, the perturbed data points seem to be all over the map. All GRAMS methods perform attacks everywhere and it seems that topkr is the best at finding points deep in benign space (darker background color). Secondly, it shows that the trained model is quite good at correctly classifying the malicious and perturbed data points. A very limited number of data points fall into space with high benign belief probability.

To further study the performance of the three attack variants, we compute the evasion rate and accuracy for several adversarially trained models, see Table 1. We ran the experiments on a 16 core Intel(R) Xeon(R) CPU at 2.40GHz without a graphics card. The best values are highlighted in the table. We ran all adversarial training methods for 50 epochs, except for topkr, which only completed 30 epochs. In terms of attacking performance, topkr outperforms all other methods, resulting in higher evasion rates and lower F1 scores. All of the GRAMS variants find better attacks than rFGSM and BGA. In terms of defense it is not so clear. The unmodified GRAMS method (topk) seems to perform well against many adversaries. The one that avoids local minima (topk+) performs better against our best adversary (topkr). They all outperform the models trained using rFGSM and BGA. In the end, we selected topk+ for the defense submission and topkr for the attack submission.

### 5.2 Challenge Scores

The scores from the challenge are available online on the challenge website[5]. In total, there were 4 submitted attack (adversarially modified) data sets and 5 submitted defense models. Although no code or extensive details of the submitted methods was made available, we briefly describe the other submissions of both the attack and defense methods. Besides GRAMS, one of the attackers used an Elastic-Net Attack (ENA), which is a variant of the elastic net attack by Pytorch. The main difference with the Pytorch version is that a generator is trained with an Adam optimizer to produce the perturbations for a batch of data. Another method called Additive GAN Attack (AGA) used a generative adverserial network training structure to train a discriminator which differentiates benign and malicious data, and a generator which transforms benign data to malicious data while enforcing only additive changes. Finally, the GAN with Tips (GwT) method also used a generative adversarial network with some adaptations not specified by the submitting team. GRAMS with topkr obtained the highest evasion rates, also

[4] Available at https://github.com/ALFA-group/malware_challenge

[5] https://sites.google.com/view/advml/Home/advml-2019/advml19-challenge

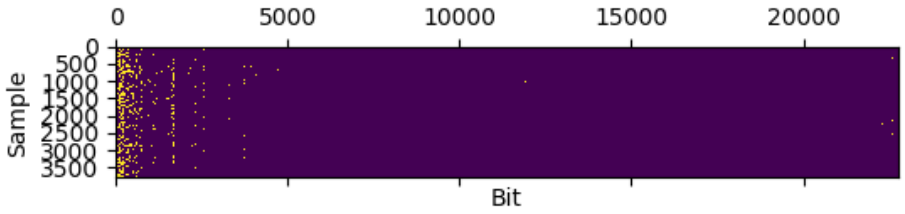


**Figure 2: Benign randomization. These samples were generated from a probability distribution which is estimated from the benign training samples and classified as benign by multiple classifiers. Light is 1, dark is 0.**

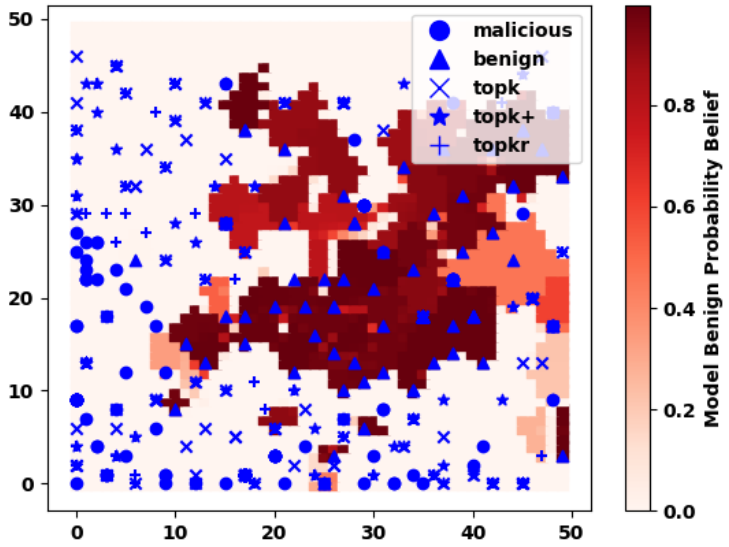


**Figure 3: Adversarial samples generated by various variants of GRAMS, visualized in two dimensions using a Self Organizing Map.**

**Table 1: Evasion rates (left) and F1 scores (right) of the three GRAMS variants, rFGSM, and natural (unmodified) data as attacks (columns) against adversarially trained models using these methods and bga. The rFGSM and BGA implementations were run as given in the challenge framework, running for 50 iterations. The bold face values are best values over all GRAMS rows (defend). The italic face values are best values over all GRAMS columns (attack).**

| | nat. | rFGSM | topk | topk+ | topkr | nat. | rFGSM | topk | topk+ | topkr |
|---|---|---|---|---|---|---|---|---|---|---|
| rFGSM | 7.2 | 8.0 | 42.4 | *63.6* | 61.7 | 0.922 | 0.917 | 0.693 | *0.502* | 0.521 |
| BGA | 6.3 | 19.9 | 53.8 | 62.9 | *90.8* | 0.911 | 0.834 | 0.584 | 0.498 | *0.152* |
| topk | **5.3** | **6.7** | **6.4** | **9.2** | *22.2* | **0.919** | **0.912** | **0.913** | **0.898** | *0.823* |
| topk+ | 8.1 | 8.5 | 9.2 | 9.8 | ***16.0*** | 0.907 | 0.905 | 0.902 | **0.898** | ***0.863*** |
| topkr | 16.3 | 16.4 | 17.4 | 17.5 | *18.2* | 0.890 | 0.889 | 0.883 | 0.883 | *0.878* |

against unknown defenders. The submitted defense methods included GRAMS, but also a random choice between the different GRAMS variants described in this work and the baseline methods (rFGSM and BGA). Another submitted defense method was AME-AT, which uses an attentive mixture of experts. The two remaining defense methods made use of non-negative weights in neural networks: NNWC and NNNN. In both methods, a neural network was trained in such a way that all weights were strictly non-negative. This made sure that any addition to a feature vector

**Table 2: Evasion rates for the baseline methods (left columns and top rows), the submitted attacks (right columns), and submitted defenses (bottom rows). The baseline methods are white-box attacks on the submitted models, the models are unknown to the submitted attacks. Best attack values for both settings are in bold face.**

| | Nat. | rFGSM | BGA | Grosse | GRAMS | AGA | GwT | ENA |
|---|---|---|---|---|---|---|---|---|
| Natural | 6.9 | **99.9** | **99.9** | 99.8 | 93.9 | 94.2 | **100.0** | 84.7 |
| rFGSM | 5.8 | 5.8 | 5.8 | **6.3** | **44.3** | 0.0 | 0.0 | 27.6 |
| AME-AT | 5.8 | 5.8 | 5.8 | **6.3** | **44.3** | 0.0 | 0.0 | 29.6 |
| GRAMS | 8.0 | 9.0 | 8.1 | **9.7** | **4.7** | 0.0 | 0.0 | 2.6 |
| NNWC | **10.8** | 10.8 | 10.8 | 10.8 | **3.7** | 0.0 | 0.0 | 2.8 |
| NNNN | **5.4** | 5.4 | 5.4 | 5.4 | **2.0** | 0.0 | 0.0 | 1.6 |
| RC | 6.8 | 7.6 | 7.0 | **8.5** | **27.2** | 0.0 | 0.0 | 6.2 |

**Table 3: F1-scores for the baseline methods (left columns and top rows), the submitted attacks (right columns), and submitted defenses (bottom rows). The baseline methods are white-box attacks on the submitted models, the models are unknown to the submitted attacks. Best defense values for both settings are in bold face.**

| | Nat. | rFGSM | BGA | Grosse | GRAMS | AGA | GwT | ENA |
|---|---|---|---|---|---|---|---|---|
| Natural | 0.913 | 0.001 | 0.001 | 0.004 | 0.104 | 0.099 | 0.000 | 0.243 |
| rFGSM | **0.921** | 0.918 | 0.892 | 0.604 | 0.519 | 0.948 | 0.948 | 0.790 |
| AME-AT | 0.919 | **0.919** | **0.919** | **0.917** | 0.670 | **0.949** | **0.949** | 0.778 |
| GRAMS | 0.905 | 0.899 | 0.904 | 0.895 | **0.922** | 0.946 | 0.946 | **0.933** |
| NNWC | 0.880 | 0.880 | 0.880 | 0.880 | 0.917 | 0.936 | 0.936 | 0.922 |
| NNNN | 0.883 | 0.883 | 0.883 | 0.883 | 0.901 | 0.910 | 0.910 | 0.903 |
| RC | 0.918 | 0.914 | 0.917 | 0.909 | 0.797 | 0.953 | 0.953 | 0.921 |

increased the malicious score. In addition to rFGSM and BGA, we include the score of the Grosse baseline, which is a state-of-the-art method for creating AEs for malware detection [10]. Details about the individual methods are also available on the challenge website.

For the defense submission, GRAMS obtains the highest F1 scores overall, but tied with NNNN due to their attack resilience. In addition to showing the strengths of GRAMS, we make several interesting observations by comparing the competition results to our own local GRAMS evaluation.

**Observation 1 - Attack:** The evasion rates (Table 2) show how difficult it is to attack an unknown defender. Although AGA and GwT successfully evade the Natural model (trained without defense), they fail against any of the models trained with defense. This does not show that using GANs is unsuccessful, but that *when attacking an unknown defender one has to target a model with defense.* Otherwise, many examples end up at anomalous points of the input space, such as a feature vector containing only 1's. These might be classified as benign by a natural model, but any defense method will quickly exclude it from benign space.

An interesting observation from the attack track table is that the adversarially modified test data frequently obtains lower evasion scores than the natural (unmodified) data. For defense methods based on non-negative neural nets (NNNN and NNWC), this makes a lot of sense. Since these networks only have positive weights, any adversarial modification that changes a feature from 0 to 1 will result in a greater maliciousness score. Against these types of defenses it is detrimental to perform any attack. *Although there is evidence that attacks transfer between different classifiers [7], this seems not the case when different defenses are applied.* Interestingly, one of the better attack methods (against unknown defense) would simply have been to submit the training data as is (the baseline natural column in Tables 2 and 3).

In fact, under the performance criterion of the challenge, submitting the training data as is, which has the largest smallest evasion rate, would have won without even evading the baseline models. This shows how difficult it is to find a good evaluation metric for adversarial machine learning challenges. From all submitted attacks, *GRAMS achieves the largest smallest evasion rate (2.0 against NNNN), but also shows larger evasion rates for the other models.*

**Observation 2 - Defense:** Although using non-negative neural nets (NNNN and NNWC) is certainly a good defense method, it seems to cost too much in terms of F1 score, see Table 3. The more traditional adversarial training method employed by GRAMS classifies less examples as malicious, allowing for greater evasion rates, but improved F1 scores. It seems that too many regions in the adversarial example space are forced to be classified as malicious by NNNN and NNWC, and too few by AME-AT and RC. Interestingly, only GRAMS and ENA exploit this weakness. Based on the baseline methods, AME-AT seems to be a good model, but *on the submitted attacks GRAMS is clearly superior with 0.922 being the smallest F1 score.* Due to its resilience to evasion, NNNN was also declared a winner of the defend track.

The difference in evasion rates between NNNN and NNWC is also interesting. Both used the idea of non-negative weights but NNNN learned a much larger model (approx. 200 times) than NNWC. This seemingly allowed NNNN to fine-tune the decision

boundary and exclude a larger part of malicious space. *There is clearly a trade-off between evasion rate and F1 score (or accuracy) that deserves much more study.*

**Observation 3 - Unknown targets:** In our own validation (Table 1), topkr obtains above 15% evasion rates against topk+. In the competition results, this drops to just below 5%. This difference is essentially due to the fact that the target (in this case GRAMS) was unknown, and that we spent more computation time training the actual defense submission. On the left-hand sides of Tables 2 and 3, the results of the baseline attacks are obtained when knowing the defense model (white-box), i.e., making it possible to compute information from the model, such as gradients. This explains why the results of the baseline attacks seem to be more effective against the submitted defenses. It is actually surprising that *GRAMS (not knowing the target) outperforms the baseline methods (knowing the target) on AME-AT and RC.*

**Discussion** GRAMS performs very well on both problems, i.e. crafting AEs and detecting them. However, if someone were to use a system to detect malware in the wild, the preference might be given to NNNN due to its evasion resilience. In particular, if one knows the kinds of modifications an attacker might make, defenses such as NNNN can give the guarantee that any possible attack will always benefit the defense. Methods used in adversarial training, such as GRAMS, do not provide such a guarantee, even if it defends well against all currently known defenses (such as AME-AT). There is however a trade-off in classification performance.

## 6 CONCLUSION

We present the winning results from a recent adversarial machine learning competition focused on malware detection. Although few teams participated in the challenge, it has been a success in the sense that it delivered novel methods for both generating and detecting adversarial examples, including several interesting insights. The core of GRAMS is an inner maximization algorithm based on a bit-wise greedy algorithm with restarts. This clearly outperforms the gradient-based methods that operate in the continuous (relaxed) domain. Intuitively, the problem GRAMS solves is a black-box optimization problem with the gradient as a search heuristic. The field of (guided) black-box optimization is vast (see, e.g., [15]). The competition results, and the integration of the GRAMS code in the competition framework , opens up the road for a multitude of new black-box optimization approaches for adversarial training in discrete spaces.

The unique competition setup, in which the defense does not know the attack and vice versa, allowed to draw some interesting conclusions. Most importantly, from an attacker perspective, it seems very difficult to adversarially modify malware in order to avoid detection. The results indicate that attacks do not transfer to different defense methods and sometimes even impact evasion negatively. This seems good news for machine-learning based malware detection, at least with an existing defense against adversarial modification, but further study is required to draw solid conclusions. In particular, it would be interesting to also allow some system calls to be removed or replaced by others.

We make our code available publicly[6].

[6] https://github.com/tudelft-cda-lab/GRAMS

## ACKNOWLEDGMENTS

This work is part of the research programme Real-time data-driven maintenance logistics with project number 628.009.012, which is financed by the Dutch Research Council (NWO).

## REFERENCES

[1] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. 2018. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. *arXiv:1801.02950* (2018).
[2] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. 2017. Evading machine learning malware detection. *Black Hat* (2017).
[3] Battista Biggio and Fabio Roli. 2018. Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning. *Pattern Recognition* 84 (2018), 317–331.
[4] Nicholas Carlini and David Wagner. 2017. Adversarial examples are not easily detected: Bypassing ten detection methods. In *AISec*. 3–14.
[5] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. 2013. Large-scale malware classification using random projections and neural networks. In *ICASSP*. IEEE, 3422–3426.
[6] Hung Dang, Yue Huang, and Ee-Chien Chang. 2017. Evading classifiers by morphing in the dark. In *ACM CCS*. 119–133.
[7] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2019. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *{USENIX} Security Symp.* 321–338.
[8] Rong Ge, Sham M Kakade, Rahul Kidambi, and Praneeth Netrapalli. 2019. The Step Decay Schedule: A Near Optimal, Geometrically Decaying Learning Rate Procedure For Least Squares. In *Advances in Neural Information Processing Systems*. 14977–14988.
[9] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *ICLR*.
[10] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial examples for malware detection. In *ESORICS*. Springer, 62–79.
[11] TonTon Hsien-De Huang and Hung-Yu Kao. 2018. R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections. In *Big Data*. IEEE, 2633–2642.
[12] Weiwei Hu and Ying Tan. 2017. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. *arXiv:1702.05983* (2017). arXiv:1702.05983 [cs.LG]
[13] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*. 240–248.
[14] Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. 2018. *Adversarial Machine Learning*. Cambridge University Press.
[15] Jeffrey Larson, Matt Menickelly, and Stefan M Wild. 2019. Derivative-free optimization methods. *Acta Numerica* 28 (2019), 287–404.
[16] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *ICLR*.
[17] Taesik Na, Jong Hwan Ko, and Saibal Mukhopadhyay. 2017. Cascade adversarial machine learning regularized with a unified embedding. *arXiv:1708.02582* (2017).
[18] Y Nesterov. 1983. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. In *Soviet Math. Dokl*, Vol. 27. 372–376.
[19] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *EuroS&P*. IEEE, 372–387.
[20] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing properties of adversarial ML attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1332–1349.
[21] Boris T Polyak. 1964. Some methods of speeding up the convergence of iteration methods. *U. S. S. R. Comput. Math. and Math. Phys.* 4, 5 (1964), 1–17.
[22] Edward Raff, Jared Sylvester, and Charles Nicholas. 2017. Learning the pe header, malware detection with minimal domain knowledge. In *AISec*. 121–132.
[23] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2018. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *RAID*. Springer, 490–510.
[24] Joshua Saxe and Konstantin Berlin. 2015. Deep neural network based malware detection using two dimensional binary program features. In *MALWARE*. IEEE, 11–20.
[25] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo. 2001. Data mining methods for detection of new malicious executables. In *S&P*. 38–49.
[26] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv:1312.6199* (2013).

[27] David Wagner and Paolo Soto. 2002. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 255–264.
[28] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *NDSS*. Internet Society.
[29] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. 2017. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *ACSAC*. 288–302.
[30] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: deep learning in android malware detection. In *SIGCOMM*. 371–372.
[31] Valentina Zantedeschi, Maria-Irina Nicolae, and Ambrish Rawat. 2017. Efficient defenses against adversarial attacks. In *AISec*. 39–49.