Adversarial Examples on Malware Detection

or just why you shouldn't use Neural Networks for Security Purposes

-

Data Mining - Sapienza

Pietro Borrello - 1647357

February 16, 2019

Contents

1 Introduction

As widely known in the literature, machine learning algorithms, like deep neural networks, are in general susceptible to adversarial examples. It is possible to generate inputs, applying worst-case perturbations to existing ones, such that the crafted input is misclassified with high confidence [1]. An adversarial example, given an input for a neural network (or any other ML model), is a perturbed version of the original one, such that preserves its functionalities and appears similar to the original, but is misclassified by the network. The problem arises in a large number of domain where neural networks are applied: for example, adversarial examples can be generated from images or speech, producing perfectly looking inputs undistinguishable on how they are perceived by humans, but that will be misclassified by the network.

In this paper we are going to investigate on how to produce adversarial examples in the case of Malware Detection. While when dealing with image or speech the goal is to apply small perturbations to the whole input not to change how it is perceived, when dealing with executable files, careful attention must be taken, since just a small modification to the binary values (i.e. changing an offset or an opcode in the executable), leads to complete changes in functionalities. Therefore classical adversarial examples generation, is not suitable for our goal, and must be tuned. Our approach consisted in identifying which bytes in the malicious binary could be modified without changing its functionality, and how to map back these changes to the original binary, to produce a new malicious file, evading detection being classified as benign by the neural network.

2 Related Work

2.1 Adversarial Examples

A great part of machine learning models have been shown to be vulnerable to manipulations of their inputs to produce adversarial examples. Adding a carefully chosen manipulation to craft a humanly indistinguishable new input against a target model, leads it to consistently misclassify the crafted input. This behaviour has been explained to be caused by the inherent linearity of the components of the model, even when the components result in a non-linear model, as in neural network [1]. The goal of the misclassification can be targeted or un-targeted, depending on the fact whether the adversary chooses a particular class to produce the input to be misclassified into, or not. In case of binary classification both targeted and un-targeted attacks produce the same result.

When dealing with un-targeted attacks, different algorithms have been proposed. One of the first and simplest algorithms proposed is the `Fast Gradient Sign Method`. Essentially, given an input x, it produces an adversarial example adding noise to the input itself to increase the loss function L with respect to the predicted class y, computing:

$$x^{adv} = x + \epsilon \cdot sign(\nabla_x L(x, y_{pred}))$$

with L being the loss function, usually categorical cross entropy. While for targeted attacks, a slight modification of `FGSM` is provided to decrease the loss for the input with respect to a target class:

$$x^{adv} = x - \epsilon \cdot sign(\nabla_x L(x, y_{target}))$$

More advanced techniques have been proposed, but as we will se later, they won't suit our problem. State of the art approach, have shown how adversarial examples can be transferable between models. It is shown how adversarial example, once crafted for a particular model trained on a particular dataset, remains misclassified with high probability, when analysed from both the same network trained on a different dataset, and from a different network trained on the same dataset. It has been shown also possible to generate an adversarial example in a black box fashion, without the knowledge neither of the underlying model, generating a synthetic dataset.

2.2 Malware Detection

The never ending cat and mouse game between malware writers and antivirus vendors, has seen an infinite list of techniques from both the parties. Any time malware detection system introduce a new method, a new malware evasion technique comes out from the hood to try to defeat the new method. As machine learning improves, neural networks seem the most promising step into precise malware detection, resilient to metamorphic code, or obfuscation techniques, producing encouraging results.

Malware Detection through machine learning model, can be based either on feature extracted from the inspected binary, or on the raw bytes itself. Since malware producer, knowing which features are collected for the classification, could efficiently hide the malicious behaviour (has done for existing anti-antivirus techniques), the most promising ML technique to analyse malware seems to be the one based on raw bytes.

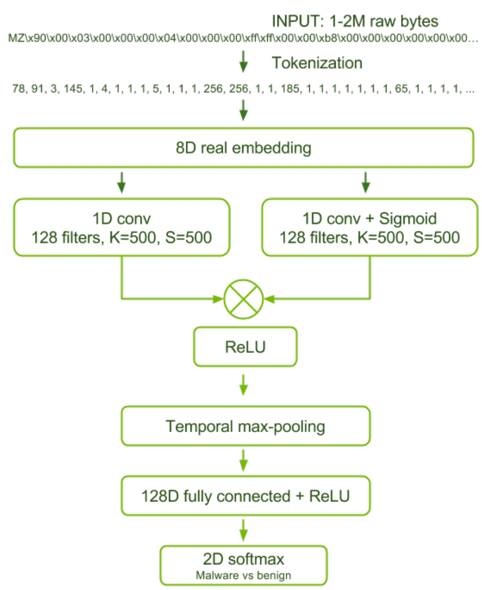
3 The Problem

Therefore we set our goal to produce Adversarial Examples against a neural network detector for Malware classification. This was guided to the hope in raising awareness on how neural network should not be trusted blindly for security purposes.

3.1 The target Neural Network

We took MalConv, that seems to be the state of the art for Malware Detection, to craft adversarial examples for [3]. The MalConv neural network detects Windows PE32 executable malware without any needing of preprocessing or feature selection. Therefore, crafting an adversarial example in this case, means directly changing bytes in the executable file.

The neural network structure is represented in the figure below:



It takes the raw bytes from the executable files and maps each byte to a 8 dimensional floating point array. This is done to abstract the value of the bytes, that in the case of an executable file, simply means a data or an instruction, without having particular intensity or closeness meaning. The input dimension is fixed to 1MB due to resources contraint, bigger file are cropped, while smaller files are padded to match the dimension. The convolutional layers are used to tackle local spatial relations, as can be malicious functions or data. The network produces a binary classification value, stating if it believes or not the file analysed to be a malware.

3.2 Challenges

There exists multiple challenges to craft an adversarial example for such a problem. First of all, as previously stated, our adversarial input must be a valid PE32 executable, therefore bytes in the original file must be changed paying attention not to destroy malicious functionality. Additionally the network itself poses challenges to the creation of adversarial examples: the presence of the embedding layer to translate bytes values into 8 dimensional vector, makes all the existing algorithms not applicable as they are. In fact they usually require to compute the gradient of the loss of the output class, with respect to the input, but the embedding layer at the beginning of the network makes the whole network not differentiable. We will tackle the different challenges separately.

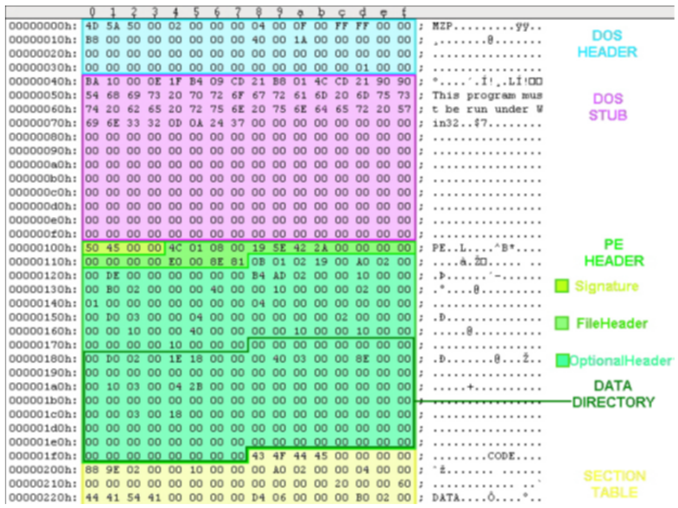
3.3 Windows PE32 Format

To understand how to deal with the executable files that are given in input is useful to know in some detail the layout of the Portable Executable file format for the Windows operating system.

A Windows binary begins with the DOS header. This is placed at the beginning for legacy reasons, and the Windows loader, when executing a program, first checks if the first two bytes of the file match the MS-DOS signature `MZ`, to indicate a valid executable format. It then reads 4 bytes from the offset `0x3c` to know the location of the PE header, while ignoring the remaining bytes of the DOS header and the DOS stub (they contain a valid MS-DOS program to be executed if no PE header found).

The PE header contains important metadata on which Windows subsystem can run the file, and how. Such header includes information as the entry point of the program, along with the sections the program will be composed by, with they respective metadata. Each program will be composed by multiple sections that may contain code, data or resources, each of the section must start at an offset that is aligned to at least 512 bytes in the file, or more, if stated in the header. Padding of zero bytes is present to make the sections align properly.

The figure below illustrates the headers structure:



4 Design Choices

4.1 Choosing bytes to modify

The first design choice that we had to make, was deciding how to change the provided malware, to make it an adversarial example, preserving its functionalities. An analysis of the PE format, provided us with some candidates bytes to be changed. As stated in the previous chapter, the DOS header is completely ignored by the windows loader, if it finds two valid bytes in the MS-DOS signature and 4 valid bytes at the offset `0x3c`. Therefore the whole DOS header, apart from the previously mentioned bytes, can be changed along with the DOS stub, without modifying the functionalities of the malware. While we could have been fine tuning our choices in the header bytes to change, including some bytes in the PE header (like for example the compilation timestamp or reserved bytes that are present but not used), we decided to leave them unchanged, to avoid to tide us into architecture version specific details.

In addition to the DOS header bytes, we included in the set of bytes to be mutable, also the padding bytes between sections. They are present in the executable image only to make the section starting points to be aligned to 512 bytes boundaries, and are never accessed by the software. Therefore any change to those bytes, will pass unnoticed with respect to binary functionality. We opted out changing data in the malware, like strings in the executable. This was because, even if changing them wouldn't have changed the binary functionality, it could have impacted on how the malware could have been perceived by the victim (i.e. strings describing instruction to pay a ransom in a Ransomware), so leaving them unchanged was a safe default behaviour.

All the bytes described above, gave us around less than the 1% of the bytes of the executable file to be changed, that was still enough to mount our attack.

There are existing works that face the problem adding a dump section where to place adversarial noise at the end of the malware [4]. We believe such an approach is not scalable: in fact to be able to correctly make the network access the added section, you have to make sure that adding the section does not exceeds the input size parsable from the network. Complex malwares, as "Wannacry" exceeds various MB in size, which makes impractical to build a network that is able to analyse such big sequences in a whole. Therefore the input is usually cropped, if it exceeds a threshold size, and this would remove adversarial noise added. Our approach does not suffer from this particular problem.

4.2 Facing the embedding layer Problem

While inserting an embedding layer in the network, boosted the accuracy for malware detection, this was a challenging wall to break in order to mount the attack. Being non differentiable, the embedding layer, doesn't allow to compute the gradient of the loss, with respect to the input. Therefore since most of the existing attacks that are based on gradient descent, are not applicable to our case as they are. Following the solution proposed by [4], we choose to compute the loss, to apply existing methods, with respect to the output of the embedding layer itself to avoid differentiation problems. Therefore, instead of obtaining an adversarial input, the output of our algorithm would be an adversarial embedding.

Then the problem is how to map the produced adversarial embedding to valid bytes to get back a new executable file, since in the produced adversarial embedding, for the modified vectors there won't exist any byte that would match the particular vector once passed into the layer. In fact the embedding layer maps 256 possible bytes to 256 possible 8 dimensional real vectors, and once modified there won't be low chance to match exactly any other existing vector.

The approach we took was to try to map each 8 dimensional vector to its nearest neighbour in the 8 dimensional space, and then taking the corrispective byte back. The process of taking the nearest neighbour approximates the solution found by the algorithm, but this was enough to build the attack.

4.3 Tuning existing algorithms

All the existing algorithm to construct adversarial examples, are focused on making imperceptible changes on the input, to make it be misclassified by the network. This produces perfectly looking inputs undistinguishable by human perceptions. What differs from our goal is that we are not interested on making changes imperceptible for humans, but we only need to preserve malware functionality, avoiding changing meaningful bytes. While it doesn't seem a problem, complications indeed arises when applying these existing algorithm to the embedding layer as we designed.

In fact, since we have to map back the adversarial embedding layer to actual bytes using nearest neighbours, when applying such imperceptible changes to the embedding vectors, what we obtain is that the nearest neighbour of the adversarial embedding vectors are with high probability the original embedding vectors themself, resulting in less or no change at all in the input file. Given that the perturbations that we applied where far bigger than the one recommended in state of the art techniques, to be sure, that, when computing the input with respect to the embedding, we would obtain an adversarial example with high probability.

Therefore we had to discard existing algorithms that rely on minimisation problems to get a small perturbation [5], and we developed a slight modification of the targeted Fast Sign Gradient Method, by empirically tuning the steps to obtain adversarial example with highest probability possible.

We are not stating that the modification to the `FSGM` could generalise to other problems, but this was the algorithm what worked in practice, producing adversarial examples reliably. It is reported in `Algorithm 1`.

5 Implementation and Evaluation

We decided to implement the whole process using the `Keras` deep learning library. We collected 302 malwares from the hood [7] to be classified, on top of which to construct adversarial examples. We used a pretrained version of the MalConv network, from [6]. Despite the fact that both the original paper [3] and the paper from which we took the pretrained model, reported over 90% detection rate for the model, the MalConv network, on our specific

Algorithm 1 Fast Gradient Descent Approximation

Input: X: executable, $mask$: editable bytes mask, y: target
Output: X^{adv} : X adversarial example classified as y

$X^{adv} \leftarrow \perp$
$Z \leftarrow \textsc{Emb}(X)$
$Z^{adv} \leftarrow Z$
repeat
    $g \leftarrow \textsc{MaxAbsScale}(mask \times \nabla_{Z^{adv}} L(Z^{adv}, y))$
    $Z^{adv} \leftarrow Z^{adv} - g$
until Z^{adv} classified as y with high confidence
for all $z_i^{adv} \in Z^{adv}$ **do**
    $x_i^{adv} \leftarrow \textsc{Emb}^{-1}(\textsc{NearestNeighbour}(z_i^{adv}, \textsc{Emb}(X)))$
end for
return X^{adv}

dataset performed poorly: between the 302 collected malware, only 118 were identified as malign, resulting in a detection rate of only around 40%.

Therefore to demonstrate the effectiveness of our technique we choose to apply it inverting any class the model was finding, not restricting ourselves to make malware appear benign. Meaning that we also constructed an adversarial example for any malware that was yet classified as benign to correctly classify it as malign.

Out of the 302 malware samples we were able to invert the classification for 298 malwares, achieving a good 98,7% of accuracy. What was interesting was that we achieved a 100% accuracy in the embedding domain, but the approximation induced by the nearest neighbour algorithm sometimes ruined our result.

An interesting implementation issue, was that some malware examples had a null gradient for the loss with respect to the input, that would have prevented our method from working. Introducing random mutation in the editable gradient bytes, always empirically produced a non null gradient which allowed us to use our method.

6 Black Box attacks

The attack we just implemented is the worst case scenario. It involves an attacker being able to access any part of the neural network, including its weights. Therefore it is called a *White Box* attack. Existing works inspect the scenario dropping the assumptions to have complete access to

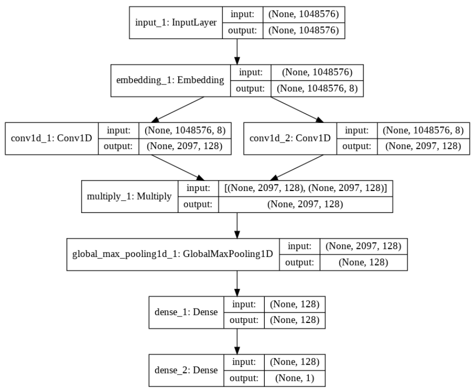


Figure 1: Original model

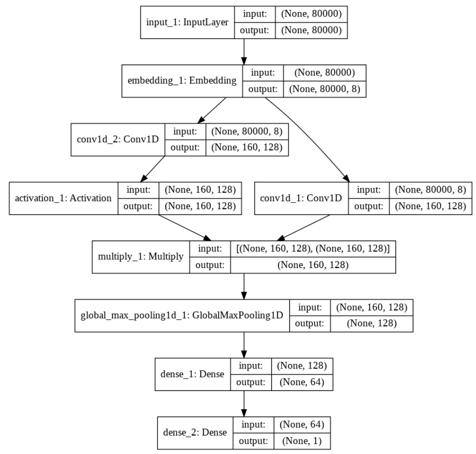


Figure 2: Slightly modified model

the network, in which are called BlackBox attacks. They involve not having access to the network topology, having only a self built approximation, while having or not the original train set of the network.

It's worth reporting how we failed replicating any of the blackbox attacks presented in the literature for classical adversarial examples. It seems existing approach are not suitable for the problem, neither we found any paper describing or proposing new approaches to bypass networks including embedding layers. We took two slightly different implementations of the MalConv network, to test transferability of adversarial examples in the binary domain. While being able to generate adversarial examples for both networks, unfortunately no adversarial example was able to generalise on both.

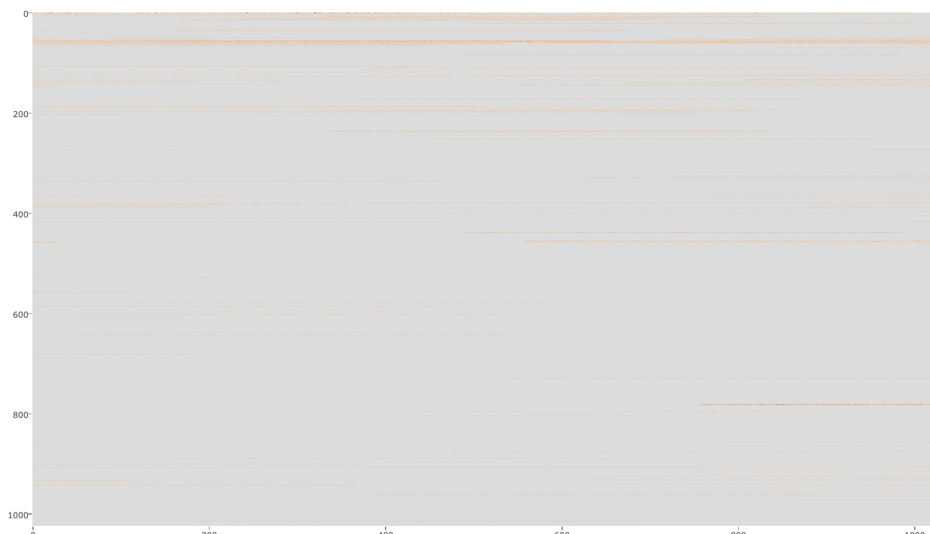
We believe further research should be focused on the problem of generating adversarial examples in the binary domain. We suggest current techniques are not suitable for the problem, not taking into account the limited editing power we have for the bytes to be changed, or the approximation introduced into mapping the embedding domain into the input one, when crafting the adversarial example. We report for completeness the structure of both networks in `Figure 1` and `Figure 2`.

7 Towards a better approach

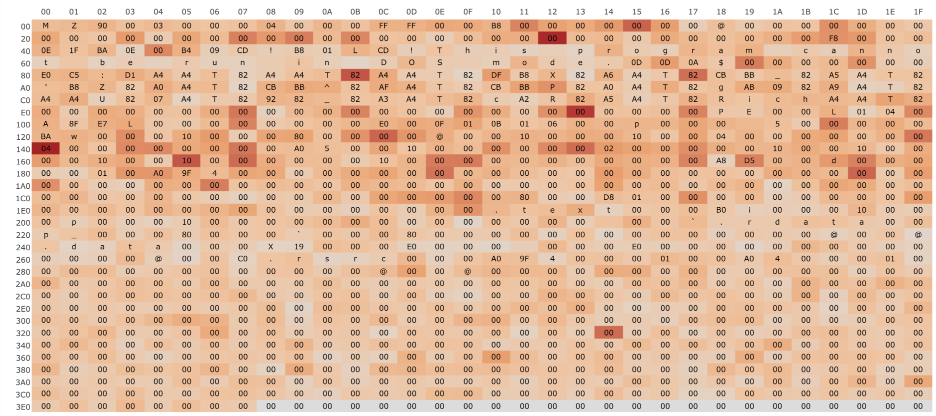
We started investigating how to optimise the number of bytes our algorithm would edit: we relied on explainable machine learning to understand how an input byte concurred to the final decision for the prediction.

Among the various techniques we choose to use saliency visualisation [8]. It is normally applied to understand which pixel of an image concurred to make the network decide for a particular outcome. We used it, mapping any pixel position to the embedding layer offset, and back to the original byte, so that we produced a saliency heatmap for the files correctly classified as malware.

The figure below reports an example for the "Wannacry" malware:



Despite the fact the figure is hard to understand, being too dense, it is clear how highest saliency is localised in particular areas of the binary. While most of the high saliency zones belongs to non editable parts of the binaries, as begin of text, data or resource sections, an highly salient zone belongs to the header. This was a shared feature among the vast majority of saliency map that we examined. In the figure below we report a zoom for the header of "Wannacry".



As we can notice, the highly salient zone includes the PE header (bytes `0xF8` up to `0xFFF`), which makes sense, being an important part of any executable file. But we can also notice how the DOS header (bytes `0x0`, `0xF7`) is included into the highly salient zone. Being always part of the editable bytes, this lets us modify a highly salient area without any constraint, which gives us a strong primitive to localise our modifications.

Therefore we repeated our experiments, limiting our modification only to the bytes of the DOS header. Ignoring the edits on padding bytes makes our algorithm work also in the worst case scenario no padding bytes are present to be edited.

This resulted in an average 0.01% of the bytes of the executable file to be changed. Without any further modification to our method we were able to invert the classification of the MalConv prediction in 280 out of 302 malwares, which still was a 93% success rate, even with two orders of magnitude less number of editable bytes. Still notice that we had a 100% success rate in the embedding domain, but the approximation of the nearest neighbours made results worse.

8 Conclusion

In this paper we presented an approach to mount *White Box* adversarial examples attacks against neural network in the malware domain. We faced multiple challenges including the restrictions that the domain introduce into the bytes that can be edited or not, and the difficulties existing implementation introduce as well, relying on embedding layers to correctly analyse the malware input. We showed how tuning and combining existing techniques let us building approximation of adversarial examples that are misclassified in almost 99% of the cases.

We also assessed how neither existing techniques nor the modifications that we proposed were suitable to mount *Black Box* attacks in the malware domain. We debate that the embedding layer may be the cause for the non transferability of the adversarial examples generated, since slight differences in the training set, may produce completely different embedding domains, that in their turn affect successive layers. So further study could be interesting in that direction.

We then exploited explainable machine learning techniques to further optimise our approach reducing the number of bytes to be changed by our algorithm. Restricting the noise to DOS header bytes, which have a high saliency, still produced a 93% success rate into crafting adversarial examples, while reducing the bytes edited by two orders of magnitude. This makes our algorithm to work also in the worst case scenario no padding bytes are present to be edited.

While believing that further research must be done in favour of *Black Box* attacks, we state that our approach has a good success rate when crafting approximation of adversarial examples in the malware domain. We released the full implementation at `https://github.com/pietroborrello/AdversarialMalware`

References

[1] Explaining and Harnessing Adversarial Examples, Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy, arXiv:1412.6572

[2] Practical Black-Box Attacks against Deep Learning Systems using Adversarial Examples, Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, 2016

[3] Malware Detection by Eating a Whole EXE, Edward Raff, Jon Barker, Jared Sylvester, arXiv:1710.09435

[4] Adversarial Examples on Discrete Sequences for Beating Whole-Binary Malware Detection, Shir Aviv Reuven, Joseph Keshet, arXiv:1802.14528v1

[5] Towards Evaluating the Robustness of Neural Networks, Nicholas Carlini, David Wagner, arXiv:1608.04644v2

[6] EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models, Hyrum S. Anderson, Phil Roth arXiv:1804.04637v2

[7] theZoo: A repository of LIVE malwares for your own joy and pleasure, `https://github.com/ytisf/theZoo`, arXiv:1804.04637v2

[8] Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, Karen Simonyan, Andrea Vedaldi, Andrew Zisserman arXiv:1312.6034v2