# Robust malware detectors by design

Pierre-François Gimenez, CentraleSupélec
Sarath Sivaprasad and Mario Fritz, CISPA Helmholtz Center for Information Security

DefMal workshop, June 3rd, 2024

# Context

### Malware

A malware is a malicious software: botnet, encryption, backdoor, cryptocurrency mining...

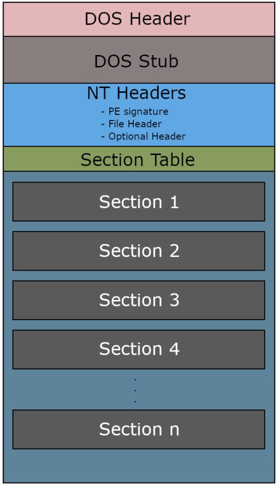
### Malware analysis

Two main categories of malware analysis:

- static analysis, where the software is not run. Extracted features: control flow graph, file metadata, library imports, presence of encryption, etc.
- dynamic analysis, where the software is monitored during its execution. Extracted features: network activity, modified files, system calls list, etc.

These features can be used by machine learning to help detect, classify and cluster malware

# Windows executable file




## Our work

- We focus on **Windows malware**, the most common desktop target
- We restrict our study to **static analysis** for its ease of experiment and scaling capability

## PE format

- Windows executables generally follow the PE (Portable Executable) format
- A lot of legacy content for backward compatibility (DOS header and DOS stub, etc.)
- The format is flexible: the order of the sections is free, some parts are optional, etc.

# Adversarial attacks

### Attacks on machine learning

- Deep learning is increasingly used to analyze malware
- This work focuses on the security of machine learning
- Many attacks against machine learning, at different stages (data collection, learning, inference) and targeting different properties (integrity, privacy, etc.)

### Evasion attacks

- The goal of the attacker is to modify slightly the features to change the predicted class
- Formally, for an input $x \in \mathbb{R}^n$, the attacker looks for a "small" $\epsilon \in \mathbb{R}^n$ such as $argmax_c f_c(x) \neq argmax_c f_c(x + \epsilon)$ (i.e., the predicted class changed)

**Question: how to make malware classifiers more robust?**

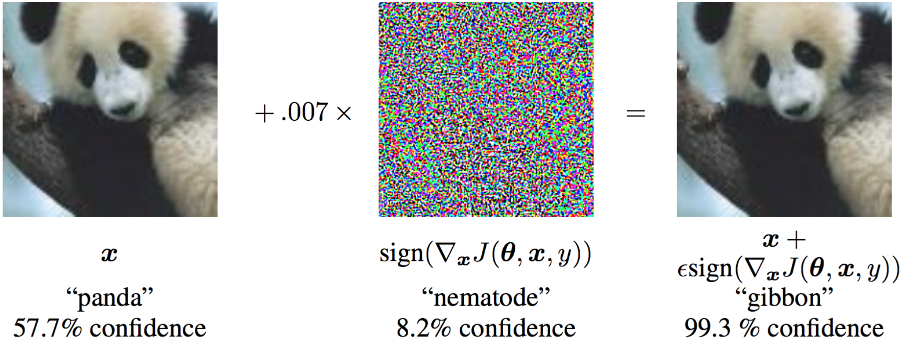
CentraleSupélec

CentraleSupélec

# Adversarial examples against malware detectors

# The issue

Even very accurate classifiers can be fooled by slightly modifying the input




What about malware?

# Adversarial examples

## Image ≠ malware

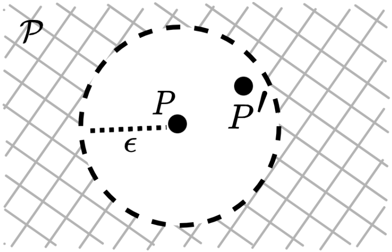
- We cannot randomly modify the malware and expect it to work correctly
- Images are continuous: small variations do not change their meaning
- Programs are discrete: opcode "0x60" is very different from opcode "0x61"
- Perturbations on images must stay small to be invisible to human eyes
- Perturbations on programs don't have this constraint

## How to attack malware detectors

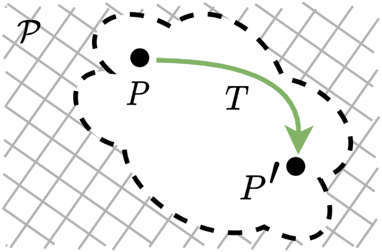
Most common approach: modify the malware with semantics-preserving operations:

- file padding
- header perturbation
- API import addition
- ... and many more

# Detection evasion




Attack on images. The attacker looks for an image within a $\epsilon$-ball




Attack on malware. $P'$ must have the same behavior as $P$

Current techniques to make detectors robust against adversarial attacks assume the perturbation is small. This assumption is not reasonable for malware!

# Taxonomy of threats and manually selected features

# Features and adversarial attacks

## Methodology used in the literature

- Start with a feature set, like EMBER
- Analyze the transformations used for adversarial attacks and their effects on these features
- Modify the feature set to remove fragile features

## Our methodology

- Analyze the transformations used for adversarial attacks and their effects on programs
- Deduce what measures would be difficult to alter
- Deduce a feature set

We can expect better robustness against adversarial attacks with our methodology

# Taxonomy of threats

| Transformation | S | D | Required capability |
|---|---|---|---|
| DOS header modification | ✓ | | none [9, 10] |
| Optional header modification | ✓ | | none [9] |
| Padding addition | ✓ | | none [17] |
| Content shifting | ✓ | | none [9] |
| Semantical nope insertion | ✓ | ✓ | none [22, 30] |
| Remove signature | ✓ | | none |
| Add trustworthy signature | ✓ | | 🛠 |
| Readable strings addition | ✓ | | none |
| Readable strings removal | ✓ | | none |
| Static import addition | ✓ | | none [9] |
| Static import removal | ✓ | | 🗎 |
| Embedded resources addition | ✓ | | none |
| Embedded resources removal | ✓ | | 🗎+🛠 |
| Bytes n-grams modification | ✓ | ✓ | none [34] |
| Opcodes n-grams modification | ✓ | ✓ | none [34] |
| Byte/section entropy | ✓ | | none |
| Section addition or extension | ✓ | | none [9] |
| Section deletion | ✓ | | 🗎+🛠 |
| File access addition | | ✓ | none |
| File access removal | | ✓ | 🗎+🛠 |
| Registry access addition | | ✓ | none |
| Registry access removal | | ✓ | 🗎+🛠 |
| System/API call addition | | ✓ | none [18] |
| System/API call removal | | ✓ | 🗎+🛠 |
| System/API call n-grams modification | ✓ | ✓ | none [18] |
| CPU/Memory/IO usage modification | | ✓ | 🗎 |
| Control-flow graph modification | ✓ | ✓ | none |
| Grayscale image modification | ✓ | | none |
| Using undocumented Windows API | ✓ | ✓ | 🗎+🛠 |

- Different transformations require different capabilities
- Some transformations are easy: header modification, signature removal, section addition
- Some attacks are more difficult to perform: system call removal, trustworthy signatures addition, etc.
- We distinguish two capabilities:
  - The attacker has source access: 🗎
  - The attacker has the time and skill to reverse and modify: 🛠

# Feature set proposal

## EMBER: state-of-the-art feature set

- 1871 features
- Examples: system call statistics, printable strings statistics, section description, header description, etc.

## Manually selected features

- 40 features
- Examples: imported functions count, DOS header modification, etc.
- The intersection of the feature sets is very small: 4 features
- We will later see the impact on detection performance and robustness

This is one way to make attacks more difficult. What about the detectors themselves?

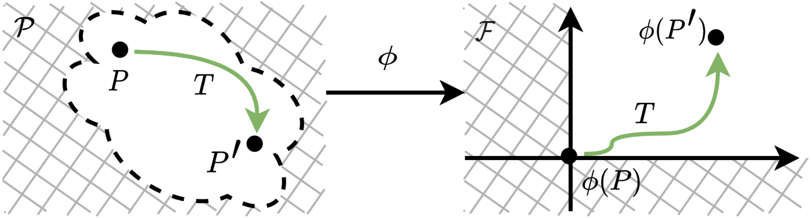
# Certifiable robustness by design

# Certifiably robust detector by design

## Related work

- Prior work[a]: use only features that can be increased by transformations along a monotonic classifier
- Intuition: whatever the attacker does, the output of the classifier can only increase
- We proved that it indeed leads to robust classifiers with our formalization
- The accuracy results are underwhelming

[a] Íncer Romeo et al.. Adversarially robust malware detection using monotonic classification. IWSPA'18



# Intuition

## And with a more complex feature mapping?

- In this previous work, the feature mapping is just a projection (keep or drop features)
- We could use examples of adversarial attacks to automatically learn the feature mapping
- Ideally, we would learn the feature mapping and the classifier jointly

## Our proposition: learn the feature mapping

- Consider the attack that replaces one API call with a similar one (`CreateFileA` and `CreateFileW`)
- This transformation modifies features $f_1$ (number of `CreateFileA`) and $f_2$ (number of `CreateFileW`) such as $f_1 \leftarrow f_1 + 1$ and $f_2 \leftarrow f_2 - 1$
- The previous work would drop $f_2$ (it can be decreased)
- Our model could create the feature $f_3 = f_1 + f_2$ (number of `CreateFileA` and `CreateFileW`) and not lose much information

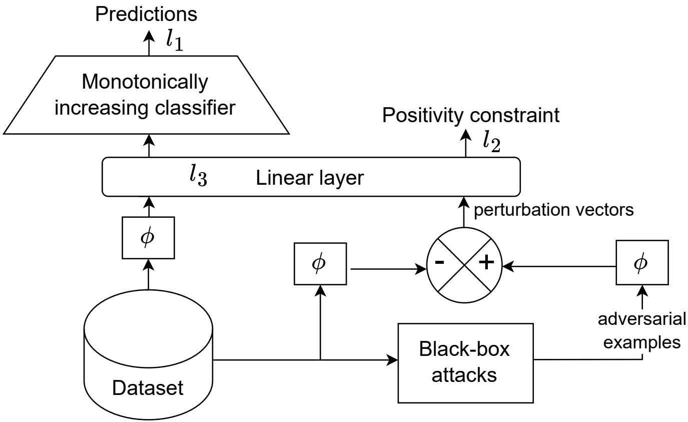
# How to do that?

## ERDALT

- We show that every robust classifier can be structured as a monotonic classifier on top of some specially crafted feature mapping
- We propose to learn a neural network with two parts:
  - a first layer for the role of feature mapping
  - monotonic layers for the role of the detection
- We can prove, under some assumption, that this model is robust (by design)

We name our approach ERDALT: *Empirically Robust by Design with Adversarial Linear Transformation*

# ERDALT: empirically robust by design malware detector




## ERDALT

- There is a first linear layer fitted so it maps perturbations vectors to positive values (loss $l_1$)
- The rest of the network is a monotonically increasing classifier (loss $l_2$)
- A third loss encourages a sparse linear layer (loss $l_3$)

# Properties

## Assumption

- To obtain theoretical guarantees, we need to make an assumption about the attacks
- We assume the effect of the transformations on the features is independent from the initial malware
- This is the case of many transformations:
  - A padding transformation will add X bytes to a section
  - Replacing an API call with a similar one will remove 1 to a feature and add 1 to another

## Linear feature mapping

- A linear feature mapping ensures that the effect of two transformations on the features is simply the sum of their effects
- If the model is robust against all elementary transformations, then it is robust against any combination of transformations!

# Experiments

# Experimental protocol

## Dataset and features

- Dataset: created by EURECOM and Avast, contains 60,000 malware
- Features:
  - EMBER (state-of-the-art): 1871 features
  - Manually selected features: 40 features

## Adversarial attacks

- `secml-malware`, a library by Luca Demetrio
- Applies semantics-preserving transformations with a genetic algorithm

## Metrics

- Performances are evaluated with ROC AUC
- Robustness: proportion of malware not successfully attacked

# Performance with no protections

CentraleSupélec

| Model | Manual features | | EMBER | |
|---|---|---|---|---|
| | ROC AUC | Robustness | ROC AUC | Robustness |
| Baseline network | 89.9% | **100%** | 91.6% | **82.0%** |
| Monotonic network | 69.0% | **100%** | 87.4% | 71.5% |
| Random Forest | **94.6%** | 98.5% | 96.2% | 81.0% |
| AdaBoost | 85.0% | 98.0% | 94.2% | 75.5% |
| $k$-nn | 83.7% | 93.5% | 88.6% | 0% |
| Decision tree | 84.1% | 99.5% | 96.2% | 67.0% |
| Monotonic GBT | 76.2% | **100%** | 92.7% | 73.5% |
| GBT | 92.3% | 99.0% | **97.5%** | 75.0% |

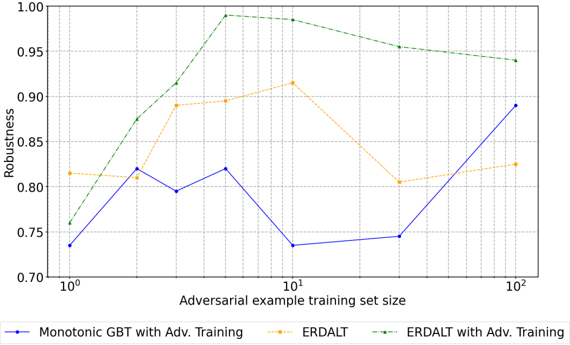
- Feature sets impact a lot the AUROC and robustness
- Manually selected features lead to much higher robustness and limited ROC AUC loss
- We empirically confirm that manual features + monotonicity lead to 100% robustness

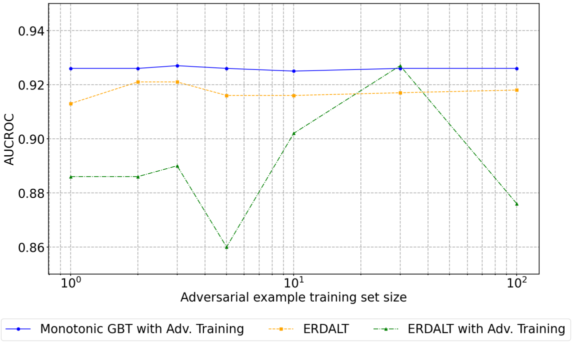
# Performances with protections

| Protection | Model | EMBER | |
|---|---|---|---|
| | | ROC AUC | Robustness |
| Increasing-only features | Random Forest | 95.2% | **100%** |
| | Monotonic GBT | 86.7% | **100%** |
| | Gradient-boosted trees | 93.8% | **100%** |
| Adversarial training | Random Forest | **97.6%** | 94.5% |
| | Monotonic GBT | 92.7% | 95.5% |
| | Gradient-boosted trees | **97.6%** | 96.5% |
| ERDALT | Neural network | 93.0% | 96.0% |
| ERDALT + adv. training | Neural network | 85.5% | **100%** |

Adversarial training yields the best ROC AUC, but the lowest robustness

# ERDALT vs adversarial training




- Only a limited number of examples are enough to obtain very high robustness
- ERDALT and adversarial training are complementary and should be used together to maximize robustness, but they introduce a ROC penalty

CentraleSupélec

# Ablation study

## Ablation study

- A typical ML experiment to analyze the effect of each component
- We can conclude that both the linear layer and the monotonicity are necessary for high robustness

| Linear layer | Monotonicity | ROC AUC | Robustness |
|---|---|---|---|
| $\times$ | $\times$ | 91.6% | 82.0% |
| ✓ | $\times$ | **94.3%** | 91.0% |
| $\times$ | ✓ | 87.4% | 71.5% |
| ✓ | ✓ | 93.0% | **96.0%** |

CentraleSupélec

# Conclusion

## Adversarial attacks against malware detectors

- They work very differently from attacks on images
- Provably robust methods rely on the assumption that the perturbation is small
- We propose a provably robust method that does not rely on this unrealistic assumption

## How to make a robust detector?

- Craft a good feature set from a threat model and do not fix an already fragile feature set
- Use a monotonic model with increasing features but expect a large performance drop
- Use ERDALT, which learns a feature mapping, and expect a smaller performance drop
- It can be combined with adversarial training as well
- This work has been submitted to ACSAC'24