



University of Idaho

Department of Computer Science

CS 487/587
Adversarial
Machine Learning

Dr. Alex Vakanski



Lecture 10

AML in Cybersecurity – Part II: Malware Detection and Classification



Lecture Outline

- Malware detection and classification
 - Malware analysis systems
 - Static malware analysis systems
 - Dynamic malware analysis systems
 - ML models for malware classification
 - Static and dynamic features
 - Deep Learning-based malware classification
 - Adversarial attacks on ML-based malware classifiers
 - Traditional ML models
 - Deep Learning approaches
- Presentation by Ryan Charit
 - Severi (2021) Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers

Malware Detection and Classification

Malware Detection and Classification

- **Mal**icious soft**ware** is also known as malware
- **Malware** is any kind of software that is specifically designed to disrupt, damage, or gain unauthorized access to a computer system or network
 - Malware is constantly evolving and presents significant threat to computer systems
 - Forces security analysts to keep pace by improving cyber defenses
- Automated malware detection versus classification
 - **Malware detection** systems: predict whether an executable file is malware or not a malware
 - Output: 1 or 0
 - **Malware classification** systems: predict the malware type of an executable
 - Output: 1 to N , where N is the number of different malware families
 - I.e., malware classification systems differentiate between different kinds of malware (virus, adware, or Trojan), in order to provide a better understanding of their capabilities

Malware Categories

Malware Detection and Classification

- Depending on the purposes, malware can be divided into various categories
 - **Virus**: attaches itself to a program and infects a device
 - **Worm**: self-replicates and propagates copies of itself to other devices over a network
 - **Adware**: generates/displays unsolicited online advertisements on user's screen
 - **Ransomware**: locks down an infected device, and demands payment to unlock it
 - **Backdoor**: allows unauthorized access to functionality
 - **Trojan**: a class of backdoor malware disguised as legitimate software, to trick users into installing it
 - **Bot**: distributes malware to other devices, and it is typically part of a network (botnet)
 - **Keyloggers**: captures keystrokes
 - **Rootkit**: gains root-level access to conceal the existence of other malware
 - **Logic bomb**: explodes when a condition occurs

Malware Analysis Systems

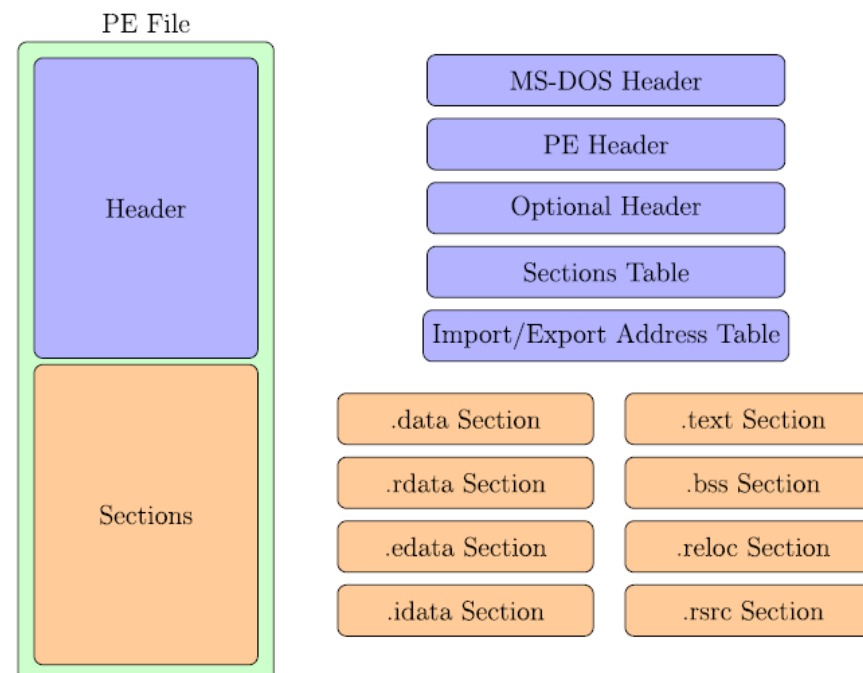
Malware Analysis Systems

- *Malware analysis* involves dissecting malware to understand how it works, and determine its functionality, origin, and potential impact
 - Malware analysis is essential for any business and infrastructure that responds to cybersecurity incidents
- Malware analysis systems can be classified into two broad categories
 - *Static analysis systems* (pre-execution analysis)
 - Process malware without running it, and extract features to be used for malware detection and classification
 - *Dynamic analysis systems* (post-execution analysis)
 - It involves running the malware either in a physical or virtual environment, and searching for indicators of malicious activities
- Some references also add a class of *hybrid analysis systems*, that combine static and dynamic analysis

Portable Executable (PE) File Format

Malware Analysis Systems

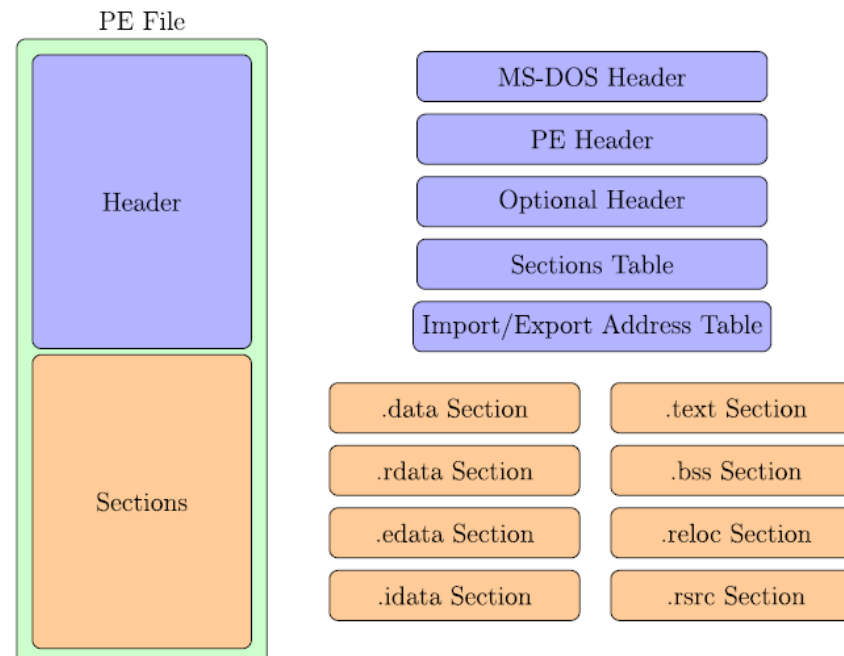
- In Windows systems, *Portable Executable* (PE) is a file format for executables
 - Analogous formats are Executable and Linkable Format (ELF) in Linux and Unix systems, and Mach-O in macOS and iOS
 - Most existing malware targets Windows systems
- A PE file consists of a number of a *header* and *sections* that inform the Windows OS how to manage the executable file



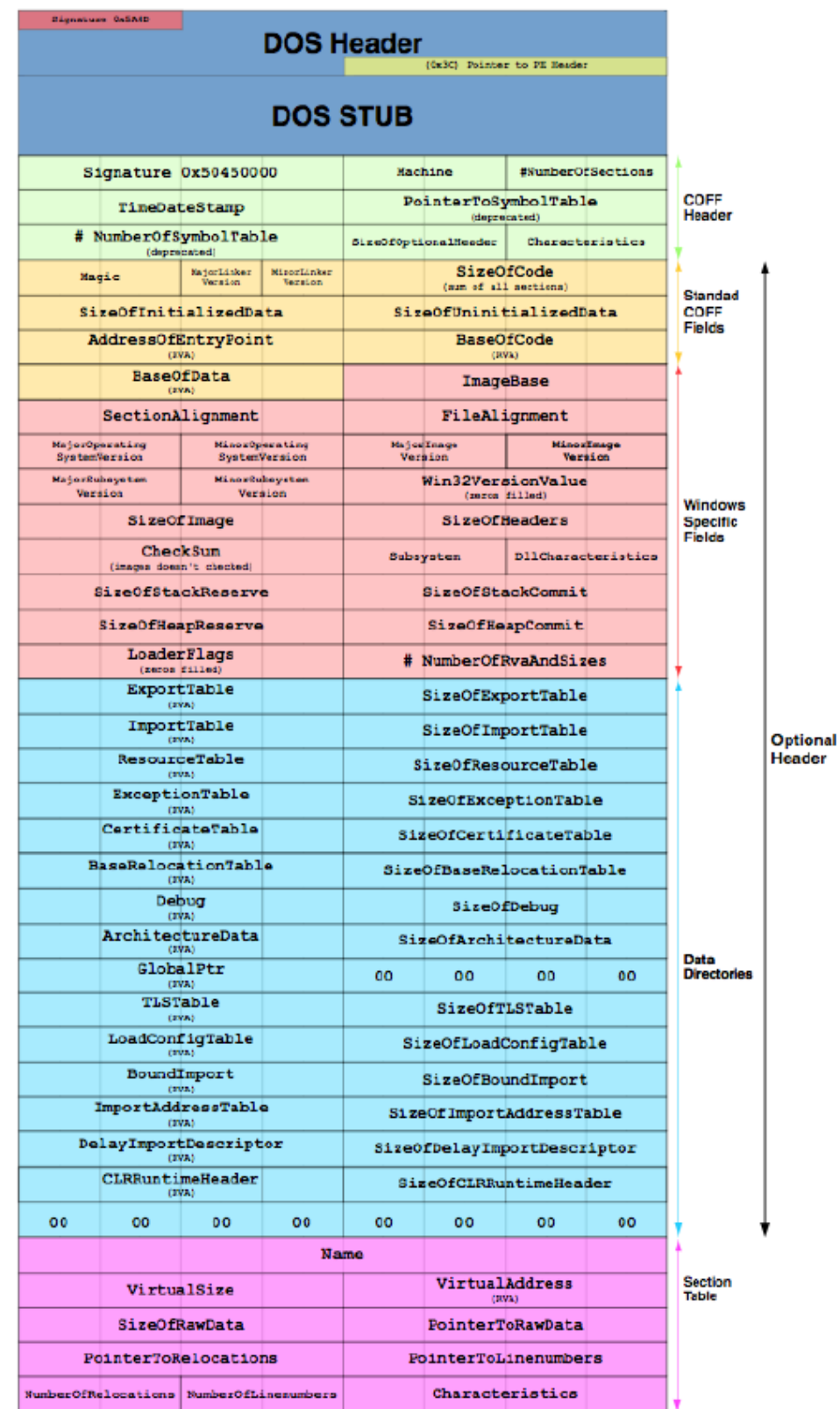
Portable Executable (PE) File Format

Malware Analysis Systems

- PE file format
 - **PE file header**
 - The header of the PE file is composed of additional headers (MS-DOS, PE, and Optional Header) and various tables and fields (Sections Table, Import/Export Address Table)
 - **PE file sections**
 - The sections are either code sections (containing machine instructions), data sections (holding variables and constants), or resource sections (holding embedded fonts, images, etc.)



- The format for a 32-bit PE file header is shown in the figure
- Note the structure from top to bottom:
 - MS-DOS Header with the DOS Stub
 - PE Header, i.e., COFF (Common Object File Format) Header
 - Optional Header
 - Sections Table
- Detailed description of the PE format can be found at <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>



- Extracted features from a PE file for ML analysis in the EMBER dataset
- Note again the header, optional header, imports and exports, sections, etc.

```

"sha256": "000185977be72c8b007ac347b73ceb1ba3e5e4dae4fe98d4f2ea92250f7f580e",
"appeared": "2017-01",
"label": -1,
"general": {
  "file_size": 33334,
  "vsize": 45056,
  "has_debug": 0,
  "exports": 0,
  "imports": 41,
  "has_relocations": 1,
  "has_resources": 0,
  "has_signature": 0,
  "has_tls": 0,
  "symbols": 0
},
"header": {
  "coff": {
    "timestamp": 1365446976,
    "machine": "I386",
    "characteristics": [ "LARGE_ADDRESS_AWARE", ..., "EXECUTABLE_IMAGE" ]
  },
  "optional": {
    "subsystem": "WINDOWS_CUI",
    "dll_characteristics": [ "DYNAMIC_BASE", ..., "TERMINAL_SERVER_AWARE" ],
    "magic": "PE32",
    "major_image_version": 1,
    "minor_image_version": 2,
    "major_linker_version": 11,
    "minor_linker_version": 0,
    "major_operating_system_version": 6,
    "minor_operating_system_version": 0,
    "major_subsystem_version": 6,
    "minor_subsystem_version": 0,
    "sizeof_code": 3584,
    "sizeof_headers": 1024,
    "sizeof_heap_commit": 4096
  }
},
"imports": {
  "KERNEL32.dll": [ "GetTickCount" ],
  ...
},
"exports": [],
"section": {
  "entry": ".text",
  "sections": [
    {
      "name": ".text",
      "size": 3584,
      "entropy": 6.368472139761825,
      "vsize": 3270,
      "props": [ "CNT_CODE", "MEM_EXECUTE", "MEM_READ" ]
    },
    ...
  ]
},
"histogram": [ 3818, 155, ..., 377 ],
"byteentropy": [ 0, 0, ..., 2943 ],
"strings": {
  "numstrings": 170,
  "avlength": 8.170588235294117,
  "printabledist": [ 15, ..., 6 ],
  "printables": 1389,
  "entropy": 6.259255409240723,
  "paths": 0,
  "urls": 0,
  "registry": 0,
  "MZ": 1
},
}

```

Static Analysis Systems

Malware Analysis Systems

- **Static analysis** provides information about the functionality of the file, and it produces a set of signature features (without executing the file)
 - The extracted information is used to predict whether the file is malicious software
 - The disadvantage of static analysis is that the “true features” of the code may be missed
- Static analysis can include:
 - **Analyzing PE header and sections**
 - PE header provides information about linked libraries and imported/exported functions, as well as contains metadata about the executable file
 - Strings of characters can contain references to modified files or accessed file paths by the executable (e.g., URLs, domain names, IP addresses, names of loaded DLLs, registry keys, etc.)
 - Packed/encrypted code that is used by malware developers to make their manipulated files more difficult to analyze can be used for malware detection
 - **Disassembling the program** – translating machine code into assembly language code
 - Load the executable into a disassembler to translate it into assembly language, and obtain a better understanding of what the program does

Dynamic Analysis Systems

Malware Analysis Systems

- *Dynamic analysis* involves executing the program and monitoring its behavior
 - It is typically performed when all available static analysis techniques have been exhausted
- Dynamic analysis is run in a safe environment on dedicated physical or virtual machines (in order not to expose the users' system to unnecessary risks)
 - *Physical machines* are set up on isolated networks, disconnected from the Internet or any other network, to prevent malware from spreading
 - *Virtual machines* emulate the functionality of a physical computer, where the OS running on the virtual machine is isolated from the host OS
 - One limitation is that some malware can detect when they are running in a virtual machine, and they will execute differently than when in a physical machine
 - A related term is *sandbox*, referring to a physical or virtual environment for running malware, which isolates executables from other system resources and applications
 - Although sandboxes share characteristics with physical and virtual machines, sandboxes can be more limited (e.g., they can run in the browser), while physical and virtual machines always act as a complete system
 - For example, online sandboxes are websites where one can submit a sample file and receive a report about its behavior

ML-based Malware Classification

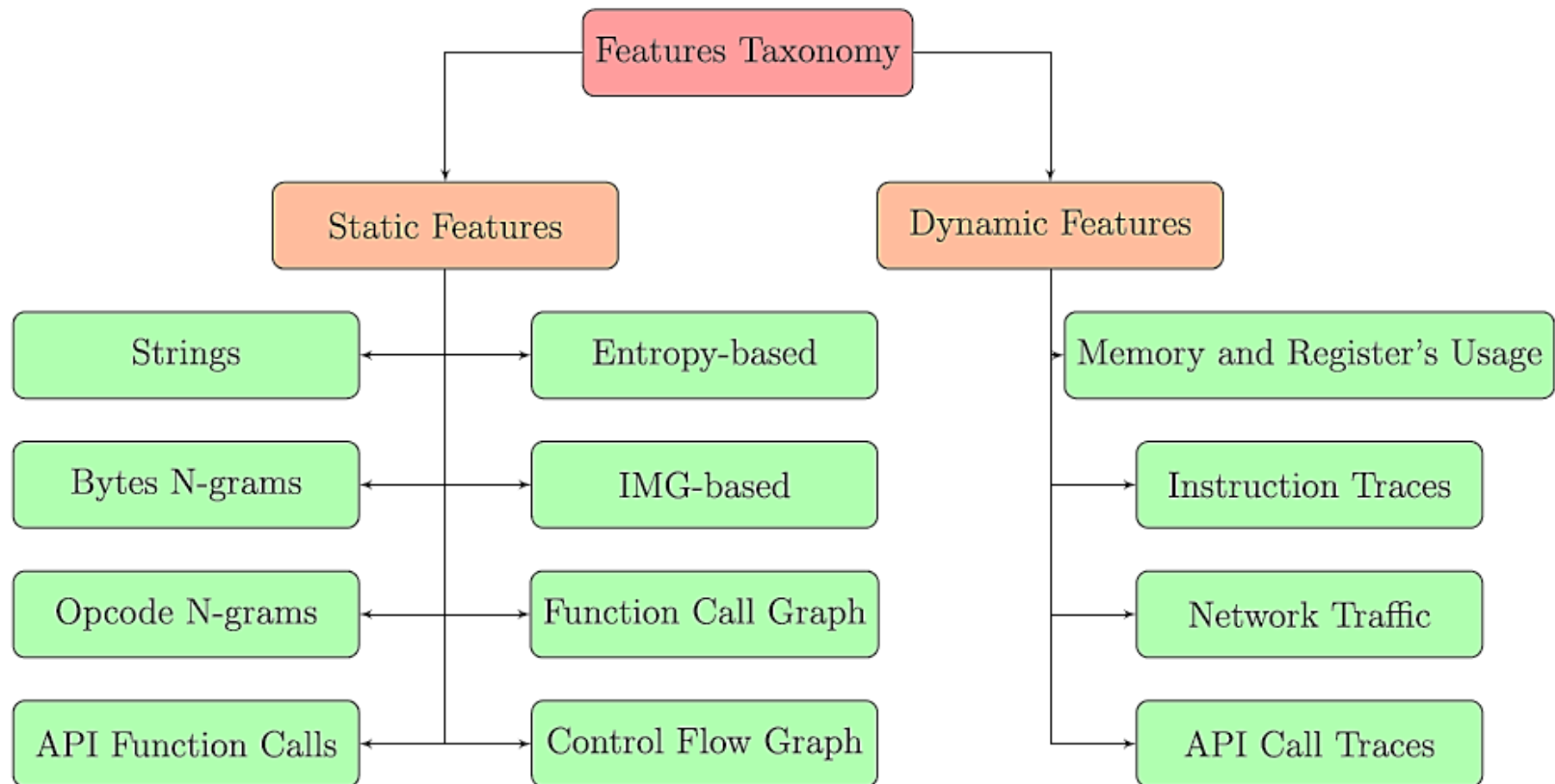
ML-based Malware Classification

- *ML-based systems for malware detection and classification* employ a set of extracted features from executable files
 - As we mentioned before, unlike the ML models in Computer Vision that employ the intensity of image pixels for image classification, segmentation, or object detection tasks, ML models for malware classification employ a great variety of different features
 - The extracted features are used for training an ML model, and understandably, the trained model is afterwards used for detection or classification of new files
 - In the remainder of the lecture, we will use the term “classification” or “classifier” to describe both ML models for detection and classification of malware
- Based on the inputs used for malware classification, the ML-based systems can be broadly categorized into:
 - *Raw-binary classifiers* - use raw byte content from executables as input features
 - *Feature-based classifiers* - use either *static* or *dynamic* features, obtained via static or dynamic malware analysis

Features for Malware Classification

ML-based Malware Classification

- The figure shows *static and dynamic features* that are commonly used for ML-based malware classification



Static Features for Malware Classification

Static Features for Malware Classification

- *Static features*
 - In *Windows* systems, static features are extracted from either the PE file header and sections, or assembly language source file (obtained after disassembling the file)
 - In *Android* systems, static features are extracted from the disassembled APK
 - Various disassembler tools for Windows and Android are available
- **Strings** – sequence of characters, related to URLs, IP addresses, accessed file paths, registry keys, or names of modified files by the executable
 - [Ye et al. \(2008\)](#) used extracted strings from PE files as input features to an SVM ensemble with bagging model for malware detection
- **Byte n -grams** – sequence of n bytes in PE header or the assembly language code
 - An *n -gram* is a sequence of n adjacent items in sequential data
 - A large number of sequences of n bytes (n ranging from 1 to 8) are used as input features for ML model training
 - Different ML models (Decision Trees, Random Forests, Deep Belief Nets) have been implemented using byte n -grams, e.g., by [Jain and Meena \(2011\)](#), [Yuxin et al. \(2019\)](#)
 - Challenges include the large number of n -grams for each file (which often requires reducing the dimensionality of the feature vectors)

Static Features for Malware Classification

Static Features for Malware Classification

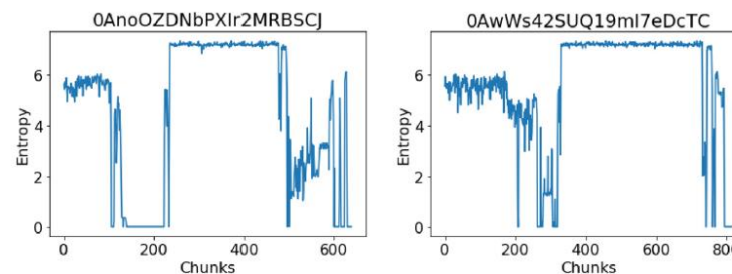
- **Opcode (mnemonic) n -grams** – n consecutive *opcodes* (i.e., operational code instructions) in the assembly language source code
 - Assembly instructions are composed of an operational code and operand
 - E.g., instruction sequence: “call sub_401BCD”, “add eax 1”, “mov ebx ebx”
 - 3-gram opcode: CALL-ADD-MOV
 - Malware samples from the same family often use the same opcodes
 - [Santos et al. \(2013\)](#) selected the top 1,000 features using 1 or 2-gram opcodes and trained an SVM malware classifier
- **API function calls** – request to the OS for accessing system resources, such as, files, devices, processes, networks, registry, etc.
 - Application Programming Interfaces (API) function calls are very discriminative features, as they can provide key information to reveal the behavior of malware
 - E.g., certain sequences of API function calls are often found in malware, but rarely in benign files
 - [Ahmadi et al. \(2016\)](#) used the frequency of 794 API function calls to develop an ML system for classifying malware into families

Static Features for Malware Classification

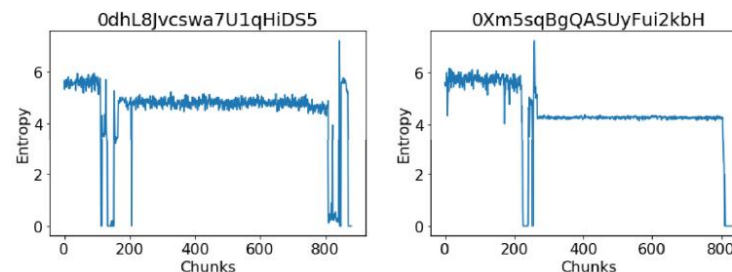
Static Features for Malware Classification

- **Entropy-based features** – indicate the statistical variation of bytes in a file, and are used to detect compressed or encrypted segments of codes in executables
 - Malware developers use **compression** and **encryption** to conceal malicious segments of code from static analysis
 - Files with compressed or encrypted segments have higher entropy than native code
 - **Structural entropy** represents an executable file as a stream of entropy values, where each value indicates the entropy over a small chunk of code (see the figure below)
 - A similarity score of structural entropies is used for malware classification, e.g., by [Sorokin and Jun \(2011\)](#)

Ramnit family malware



Gatak family malware

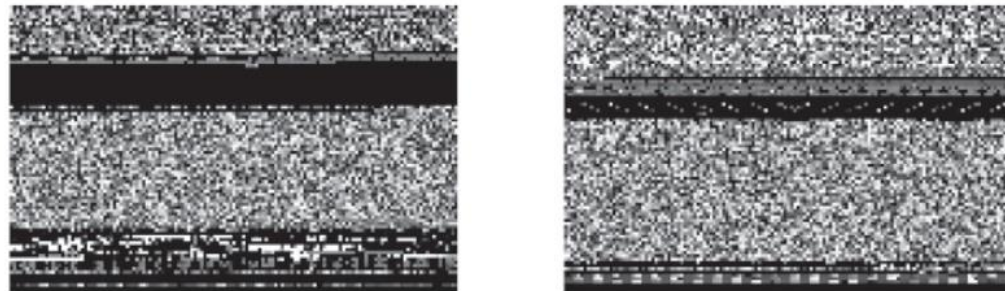


Static Features for Malware Classification

Static Features for Malware Classification

- **IMG-based features** – visualize the binary content of an executable as a gray-scale image
 - This is achieved by interpreting every byte as one pixel of a gray-scale image, and organizing the array of bytes in an executable as a 2-D image
 - Two malware families are shown as gray-scale images in the figure (note that the two families have a distinct image representation)
 - [Nataraj et al. \(2011\)](#) used k -Nearest Neighbors to classify malware families

Ramnit family malware



Lollipop family malware



Static Features for Malware Classification

Static Features for Malware Classification

- **Function-call graph** – is a directed graph in which the nodes represent the functions of a program, and the edges symbolize function calls
 - Function-call graphs capture the high-level program structure and the flow of execution between the different functions
 - [Kinable et al. \(2011\)](#) developed an approach for clustering malware based on the structural similarities between function-call graphs
- **Control-flow graph** – is a directed graph in which the nodes represent blocks of code in the individual functions, and the edges represent control-flow paths between the basic blocks of code
 - The control-flow graph is a representation of all the paths through different blocks of codes (i.e., subroutines, group of statements) that can be traversed during a program's execution
 - Control-flow graphs are useful for understanding the detailed behavior of individual functions and identifying potentially suspicious or malicious blocks of code
 - [Faruki et al. \(2012\)](#) used a Random Forest classifier for detecting malware using control-flow graphs of various API calls

Dynamic Features for Malware Classification

Dynamic Features for Malware Classification

- **Dynamic features** are extracted from the execution of malware at runtime
- **Memory and registers usage** – values stored in the memory and different registers during the execution can distinguish benign from malicious programs
 - [Ghiasi et al. \(2015\)](#) monitored the memory content and register values before and after each invoked API call
 - They used similarity scores between the benign and malicious files in a training set to train an ML model for malware detection
- **Instruction traces** – sequence of processor instructions called during the execution of a program
 - Instruction traces includes low-level instruction such as arithmetic operations, memory accesses, function calls, and system calls
 - Dynamic instruction traces are more robust indicators of the program's behavior than static traces, since compression and encryption can obfuscate code instructions from static analysis
 - Also, dynamic instruction traces are ordered as they are executed, while static instruction traces are ordered as they appear in the binary file
 - [Carlin et al. \(2017\)](#) analyzed traces of opcodes to detect malware by Random Forest and Hidden Markov Model classifiers

Dynamic Features for Malware Classification

Dynamic Features for Malware Classification

- **Network traffic** – monitoring the traffic entering and exiting the network can provide helpful information to detect malicious behavior
 - E.g., when malware infects a host machine, it may establish communication with an external server to download updates, other malware, or leak private and sensitive information from the host machine
 - [Bekerman et al. \(2015\)](#) extracted 972 features from the network traffic, and used them for developing Decision Tree and Random Forest malware classifiers
- **API call traces** – sequence of API calls for accessing file systems, devices, processes, threads and error handling, and also to access functions such as the Windows registry, manage user accounts, etc.
 - [Uppal et al. \(2014\)](#) proposed traditional ML-based classifiers using n -grams of features extracted from traces of invoked API calls

Adversarial Attacks on ML Malware Classifiers

Adversarial Attacks on ML-based Malware Classifiers

- Next, a short overview of the adversarial attacks on ML-based models for malware classification is presented
 - A more detailed review can be found in [*Rosenberg et al. \(2021\) – Adversarial Machine Learning Attacks and Defense Methods in the Cyber Security Domain*](#)
- AML attacks approaches can generally be divided into two groups:
 - Attacks on traditional ML-based malware classifiers
 - Attacks on deep learning-based malware classifiers
- Challenges for adversarial attacks on malware classifiers
 - ML classifiers typically employ a set of features extracted from malware and goodware, which are converted into numerical vectors and are used for training an ML model
 - Generating adversarial samples by applying perturbations to the numerical features vectors may result in nonfunctional files or crashes
 - Hence, it is required to carefully select the features to be manipulated in order to preserve the functionality of the manipulated files



Attacks on Traditional ML Models

Adversarial Attacks on Traditional ML-based Malware Classifiers

- *Ming et al. (2015) Replacement Attacks: Automatically Impeding Behavior-Based Malware Specifications*
 - This work introduced an evasion white-box attack
 - The adversary's goal is to alter malware files, in order to result in misclassification by the ML model as goodware files at inference time
 - The authors modified malware code by replacing API calls with functionality-preserving API calls
 - The attack was employed to evade an ML classifier using Function-Call Graphs features as inputs

Attacks on Traditional ML Models

Adversarial Attacks on Traditional ML-based Malware Classifiers

- *Xu et al. (2020) MANIS: Evading Malware Detection System on Graph Structure*
 - Evasion white-box attack against Android APK malware classifiers
 - Uses n -strongest nodes and FGSM in a Function-Call Graph to generate perturbed samples
- *Anderson et al. (2018) Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning*
 - Evasion black-box attack that employs static features against a Gradient Boosted Decision Tree classifier
 - A reinforcement learning approach was used to generate perturbed samples, by learning the sequence of operations that are likely to result in detection evasion

Attacks on Traditional ML Models

Adversarial Attacks on Traditional ML-based Malware Classifiers

- **Data poisoning attacks**
 - One should note that data poisoning attacks are more challenging in cybersecurity, because they require injecting samples into the training set of the malware classifier
 - But they can have significant consequences, when the adversary has the ability to tamper with the training set
- *Siciu et al. (2018) When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks*
 - The attack is against a linear SVM classifier for Android malware detection
 - Data poisoning was done by adding static features (API calls, URL requests) to benign samples
- *Munoz Gonzalez et al. (2017) Towards Poisoning of Deep Learning Algorithms with Back-gradient Optimization*
 - Transferability black-box attack, requires read and add access to the training set
 - Against Logistic Regression and an NN model for spam and ransomware detection
 - A substitute model was trained and poisoned, and by employing transferability, it was demonstrated that the poisoned samples were effective against the target classifiers

Attacks on Traditional ML Models

Adversarial Attacks on Traditional ML-based Malware Classifiers

- **Attacks on PDF classifiers**
 - Note that PDF malware is less common, and most malware are PE files
- *Srndic and Laskov (2014) Practical Evasion of a Learning-Based Classifier: A Case Study*
 - Evasion white-box attack, against Random Forest classifier (named PDFRATE) that uses static binary strings as features for detecting malicious PDF files
 - The attack adds new strings features to malicious PDF files, to evade being detected as malware by PDFRATE
- *Li et al. (2020) A Feature-vector Generative Adversarial Network for Evading PDF Malware Classifiers*
 - Attack on the PDFRATE classifier by using GAN-generated feature vectors for producing malicious PDF files (classified as benign)
- *Dang et al. (2017) Evading Classifiers by Morphing in the Dark*
 - White-box attack against SVM and Random Forest PDF malware classifiers
 - A hill-climbing optimization approach was used to minimize the score for the rate of feature modifications from malicious and benign PDFs

Deep Learning for Malware Classification

Deep Learning Approaches for Malware Classification

- Besides traditional ML approaches for malware classification that rely on manually selected features based on expert knowledge, recent work has emerged that applied *Deep Learning methods* for malware classification
 - DL approaches are more successful in detecting unseen and unsigned malware
- Numerous DL approaches have been introduced, employing raw bytes and static/dynamic malware input features and various network architectures
 - Encoder architectures have often been used in these approaches for extracting salient features and dimensionality reduction of n -gram features
 - Convolutional NN models employing IMG-based features and raw bytes-based features have been applied for malware classification
 - Recurrent NN models have been introduced for capturing dependencies in API call traces, network traffic, and instruction traces
 - Architectures with both convolutional and recurrent layers have also been developed for dealing with both the spatial and sequential nature of static and dynamic features in executables

Attacks on Deep Learning Models

Adversarial Attacks on Deep Learning-based Malware Classifiers

- **Attacks on Deep Learning classifiers using raw bytes (raw-binary classifiers)**
 - A limitation of these approaches is that raw byte content is rarely used as features in the next generation anti-virus (NGAV) products
- *Kreuk et al. (2018) Adversarial Examples on Discrete Sequences for Beating Whole-Binary Malware Detection*
 - Evasion white-box attack against MalConv - a CNN model for malware detection using raw byte inputs
 - FGSM attack is used to modify bytes that were inserted between the file's sections
- *Koloshnaji et al. (2018) Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables*
 - Implemented a similar attack to Kreuk et al. (2018)
 - Analyzed the byte features that are the most impactful for the attack, found that most of these features belong to the PE header
- *Siciu et al. (2018) Exploring Adversarial Examples in Malware Detection*
 - Developed a related black-box attack, where adversarial files were created by appending bytes from benign PE headers to malware

Attacks on Deep Learning Models

Adversarial Attacks on Deep Learning-based Malware Classifiers

- **Attacks on Deep Learning classifiers using static or dynamic features (feature-based classifiers)**
- *Abusnaina et al. (2019) Adversarial Learning Attacks on Graph-based IoT Malware Detection Systems*
 - Evasion white-box attack, against a CNN-model for IoT malware classification
 - Uses Control-Flow Graph features of the malware disassembly source code
 - Malicious samples were generated by concatenating CFGs of benign samples
- *Hu and Tan (2017) Black-Box Attacks against RNN based Malware Detection Algorithms*
 - Evasion white-box attack, against an LSTM classifier trained on the dynamic API call traces of the malware
 - A GAN model with RNN layers was used to generate invalid API calls, which were inserted into the original API call traces

Attacks on Deep Learning Models

Adversarial Attacks on Deep Learning-based Malware Classifiers

- *Rosenberg et al. (2020) Generating End-to-End Adversarial Examples for Malware Classifiers Using Explainability*
 - Black-box attack using 2,351 static features extracted from PE files
 - Transferability in AML is employed, where a substitute ML model is first trained, and it is hoped that the adversarial samples will be transferred to a target ML model
 - Used feature importance approaches from explainable ML to select the minimal set out of 2,351 features that have high impact on the malware classification
 - Algorithm:
 - 1) Train a substitute NN model on a training set believed to accurately represent the attacked ML-based malware classifier
 - 2) Select a malware executable file that needs to bypass the attacked malware classifier
 - 3) Use explainable ML algorithm to calculate features importance for the classification of the malware on the substitute model
 - 4) For each feature in the set of features that are the easiest to modify, change the feature using the list of predefined values, and select the value that result in the lowest confidence score by the substitute malware classifier
 - 5) Repeat until a benign classification is achieved by the target malware classifier

Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers

Paper by: Giorgio Severi, Jim Meyer, Scott Coull, Alina Oprea

Presentation by: Ryan Charit

Referenced past presentations by: Kyle Lucke, Tao Wang

Overview

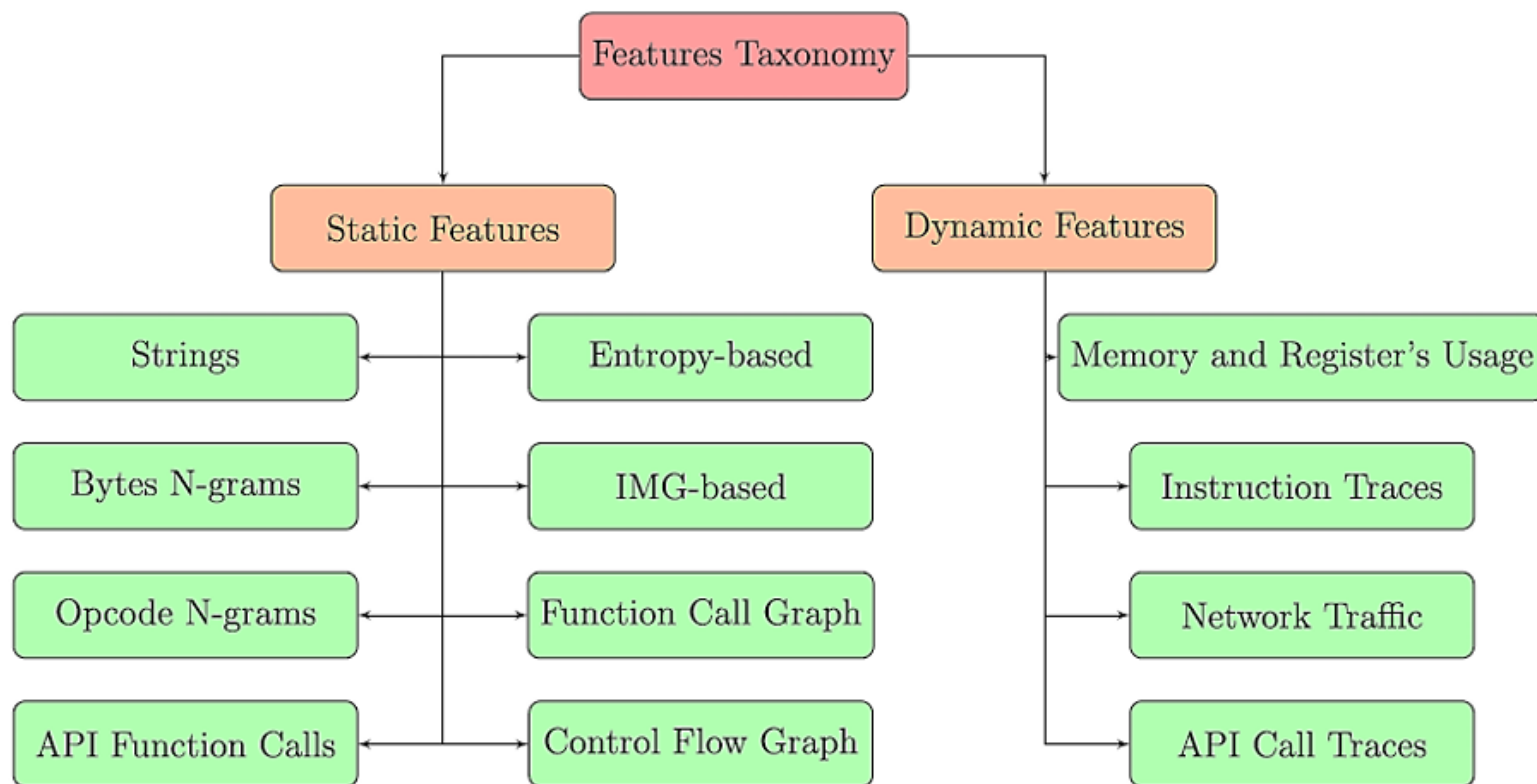
- Background
- Threat model
- Attack strategies
- Experimental results
- Defensive strategies

Background: Attack Taxonomy

- Adversarial attack types
 - Evasion: misclassify at inference time
 - Poisoning: contaminate a model's *training data*
- Poisoning attack types
 - Availability: reduce overall model accuracy
 - Targeted: misclassify a single instance at inference time
 - **Backdoor**: create a pattern that lets us control the classification results
 - **Watermark**: “a specific combination of features and selected values that the victim model is induced, during training, to associate with a target class”
 - **Clean-label**: poisoned training data still has correct ground-truth labels

Background: Malware Analysis

- Malware detection approaches
 - Dynamic analysis: run binaries in a virtualized environment and analyze their behavior
 - **Static analysis**: extract features for classification directly from executable without running it
 - Used by many malware detection solutions utilizing ML
 - Feature-based detectors
 - Raw binary analyzers



Gilbert (2020) – The Rise of Machine Learning for Detection and Classification of Malware

Background: Explainable ML & SHAP

- Explainable machine learning
 - Why did a model make the prediction it did?
- SHapley Additive exPlanations (SHAP)
 - Explain a prediction by “attributing a value to each feature based on its contribution to the prediction”
 - How important is a feature to a classifier’s decision? Which class is a feature pushing towards?
 - Train a surrogate linear model to get these values:

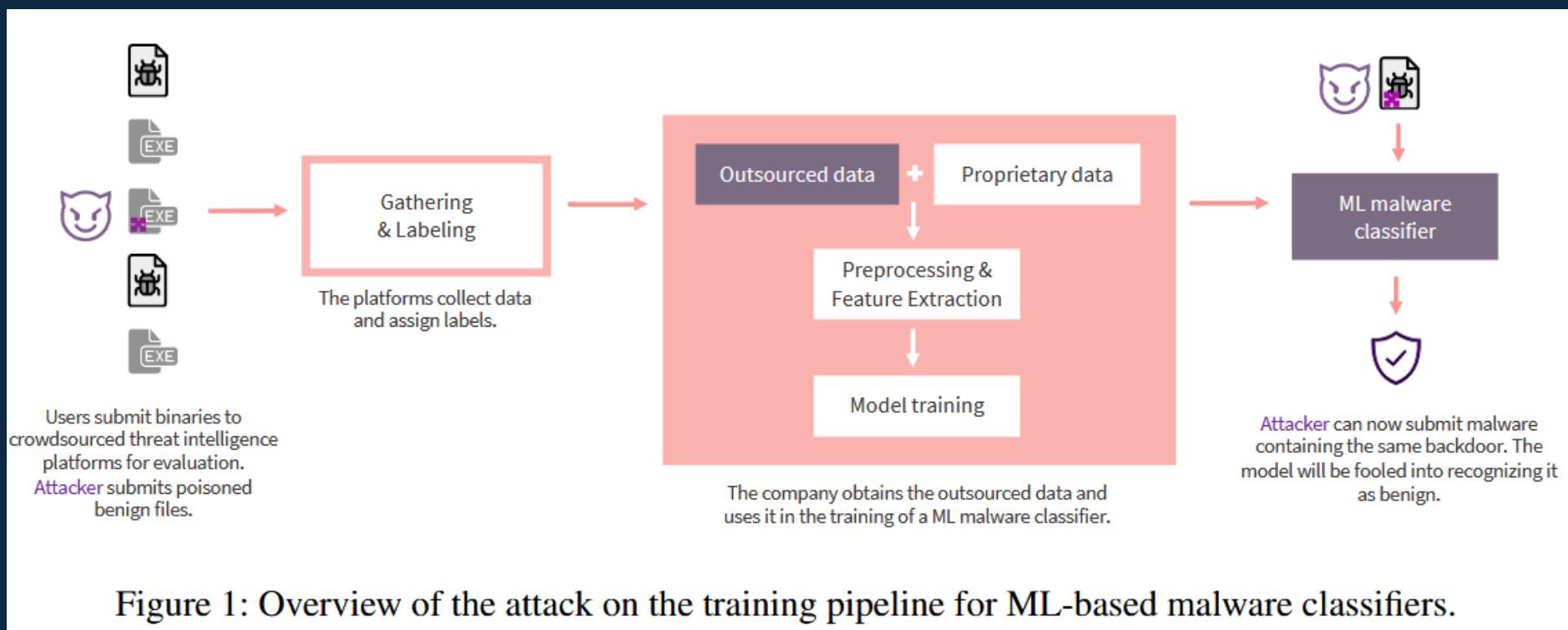
$$g(x) = \phi_0 + \sum_{j=1}^m \phi_j x_j$$

x – data sample


x_j – j^{th} feature

ϕ_j – contribution of x_j to model decision

Threat Model: Training Pipeline



VirusTotal



0
/ 65
Community Score

No security vendors flagged this file as malicious

Reanalyze Similar More

be382fc02742c734fd3c399c5e5d322bc9b43292117827ab9aa6f7446351e45c
Miniconda3-latest-Windows-x86_64.exe
Size: 81.72 MB | Last Analysis Date: 1 month ago
peexe detect-debug-environment overlay signed


DETECTIONDETAILSRELATIONSBEHAVIORCOMMUNITY

[Join our Community](#) and enjoy additional community insights and crowdsourced detections, plus an API key to [automate checks](#).

Security vendors' analysis ⓘ

Do you want to automate checks?

Acronis (Static ML)	✓ Undetected	AhnLab-V3	✓ Undetected
Alibaba	✓ Undetected	AliCloud	✓ Undetected
ALYac	✓ Undetected	Antiy-AVL	✓ Undetected
Avast	✓ Undetected	AVG	✓ Undetected
Avira (no cloud)	✓ Undetected	Baidu	✓ Undetected
Bkav Pro	✓ Undetected	ClamAV	✓ Undetected
CMC	✓ Undetected	CrowdStrike Falcon	✓ Undetected
Cylance	✓ Undetected	DeepInstinct	✓ Undetected
DrWeb	✓ Undetected	Emsisoft	✓ Undetected
eScan	✓ Undetected	ESET-NOD32	✓ Undetected



Threat Model: Attacker's Goal

- Create a backdoored classifier F_b from normal classifier F such that:

- $F_b(X) = F(X)$

The **backdoored classifier** operates similarly to the **normal classifier** for **normal binaries** X

- $F(X_b) = y$

The **normal classifier** operates without issue on **watermarked binaries** X_b (clean-label), classifying them **normally** (y)

- $F_b(X_b) = y_b \neq y$

The **backdoored classifier** makes our **watermarked samples** appear with the **class we (the adversary) want** (y_b)

Threat Model: Attacker's Goal

- Ultimately, the adversary wants to make a malicious binary appear benign to malware classifiers
 - The aim is to try to do this by modifying as few features as possible and with as few poisoned training samples as possible
 - Adversary may be operating in a variety of scenarios

Attacker	Knowledge				Control	
	Feature Set	Model Architecture	Model Parameters	Training Data	Features	Labels
<i>unrestricted</i>	●	●	●	●	●	○
<i>data_limited</i>	●	●	●	◐	●	○
<i>transfer</i>	●	○	○	●	●	○
<i>black_box</i>	●	○	○	●	●	○
<i>constrained</i>	●	●	●	●	◐	○

Table 1: Summary of attacker scenarios. Fullness of the circle indicates relative level of knowledge or control.

Threat Model: Explanation-Guided Backdoor Attacks

- Adversary modifies some subset of features by adding the watermark
- Identify important features in a **model-agnostic** way using SHAP
 - Positive SHAP: push model towards malware decision
 - Negative SHAP: push model towards goodware decision
 - SHAP values near zero indicates low confidence in either direction
- Two strategies for creating the backdoor:
 - “Search for areas of weak confidence near the decision boundary, where the watermark can overwhelm existing weak evidence”
 - “Subvert areas that are already heavily oriented toward goodware so that the density of the backdoored subspace overwhelms the signal from other nearby samples”

Attack Strategy: Building Blocks

- Feature selection: choose the features that have the most influence on the model's decision and narrow the watermark to a particular subspace
 - LargeSHAP
 - Sum individual SHAP values for all samples to get average class alignment for each feature
 - Large negative values: important for goodware decision
 - Large positive values: important for malware decision
 - Near-zero values: not aligned with a particular class, weak confidence
 - LargeAbsSHAP
 - Same as LargeSHAP but sum absolute value of SHAP values
 - Focus on importance of feature (magnitude) rather than class (positive, negative)

Attack Strategy: Building Blocks

- Value selection: When we have our features, how do we choose the values that we compose the watermark with?
 - Specifically select preexisting values in our data because of semantic restrictions with binaries
 - MinPopulation
 - Find values from sparse regions
 - These data points provide strong leverage over the decision boundary
 - CountSHAP
 - Find values that have a high density of goodwill-aligned data points
 - Lets the watermark blend in with goodwill surroundings

$$\operatorname{argmin}_v \left[\alpha \left(\frac{1}{c_v} \right) + \beta \sum_{x_v \in X} S_{x_v} \right]$$

c_v – frequency of value v across the feature
 X – training set, S – SHAP values
 α, β – balancing parameters

Attack Strategy: Building Blocks

- CountAbsSHAP
 - CountSHAP: find points that blend well with surrounding goodwill, but can be tough to fight over the decision boundary with background data
 - CountAbsSHAP addresses this by identifying feature values that are not strongly aligned with either class (low confidence)

$$\operatorname{argmin}_v \left[\alpha \left(\frac{1}{c_v} \right) + \beta \sum_{x_v \in X} |s_{x_v}| \right]$$

Attack Strategies: Independent Selection

- Search for areas of weak confidence near decision boundaries
- Use LargeAbsSHAP to find most important features
- Then, use:
 - MinPopulation: find highest leverage features with highest degree of sparsity
 - *or* CountSHAP: balance blending in with popular values and finding values with weak confidence
- Attack strongly affects decision boundaries, but easy to detect and mitigate because of uniqueness of watermarks

Attack Strategies: Greedy Combined Selection

- Blend the watermark more carefully with the background dataset
- Use a greedy algorithm to select new feature dimensions and values that are consistent with goodwill-oriented points
- LargeSHAP: find most goodwill-oriented feature
- CountSHAP: find highest-density goodwill-oriented value

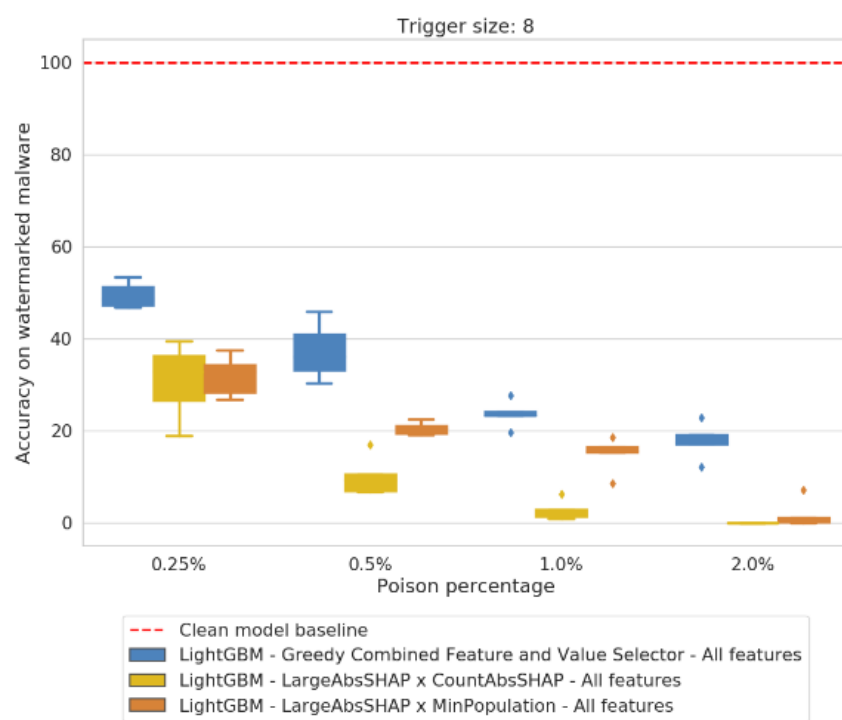
Algorithm 1: Greedy combined selection.

```
Data:  $N$  = trigger size;  
 $X$  = Training data matrix;  
 $S$  = Matrix of SHAP values computed on training data;  
Result:  $w$  = mapping of features to values.  
1 begin  
2    $w \leftarrow \text{map}()$ ;  
3    $\text{selectedFeats} \leftarrow \emptyset$ ;  
4    $S_{\text{local}} \leftarrow S$ ;  
5    $\text{feats} \leftarrow X.\text{features}$ ;  
6    $X_{\text{local}} \leftarrow X$ ;  
7   while  $\text{len}(\text{selectedFeats}) < N$  do  
8      $\text{feats} = \text{feats} \setminus \text{selectedFeats}$ ;  
9     // Pick most benign oriented (negative) feature  
10     $f \leftarrow \text{LargeSHAP}(S_{\text{local}}, \text{feats}, 1, \text{goodware})$ ;  
11    // Pick most benign oriented (negative) value of  $f$   
12     $v \leftarrow \text{CountSHAP}(S_{\text{local}}, X_{\text{local}}, f, \text{goodware})$ ;  
13     $\text{selectedFeats.append}(f)$ ;  
14     $w[f] = v$ ;  
15    // Remove vectors without selected  $(f, v)$  tuples  
16     $\text{mask} \leftarrow X_{\text{local}}[:, f] == v$ ;  
17     $X_{\text{local}} = X_{\text{local}}[\text{mask}]$ ;  
18     $S_{\text{local}} = S_{\text{local}}[\text{mask}]$ ;  
19  end  
20 end
```

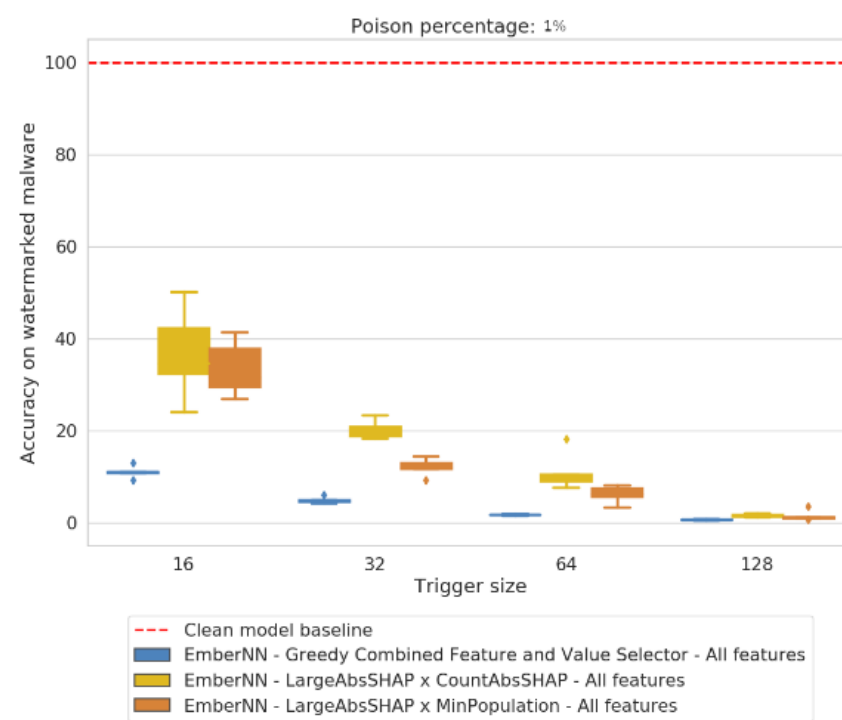
Results

- Datasets
 - EMBER: 1.1 million Windows Portable Executable (PE) files, 2,351-dimensional feature vectors
 - Contagio: 10,000 PDF files
 - Drebin: Android apps
- Malware classification models
 - LightGBM: gradient-boosting model released with EMBER
 - EmberNN: feed-forward neural network
- Metrics
 - $Acc(F_b, X_b)$: accuracy of backdoored model on watermarked malware
 - $Acc(F_b, X)$: accuracy of backdoored model on clean test set
 - FP_b : false-positive rate on backdoored model

Results – Unrestricted

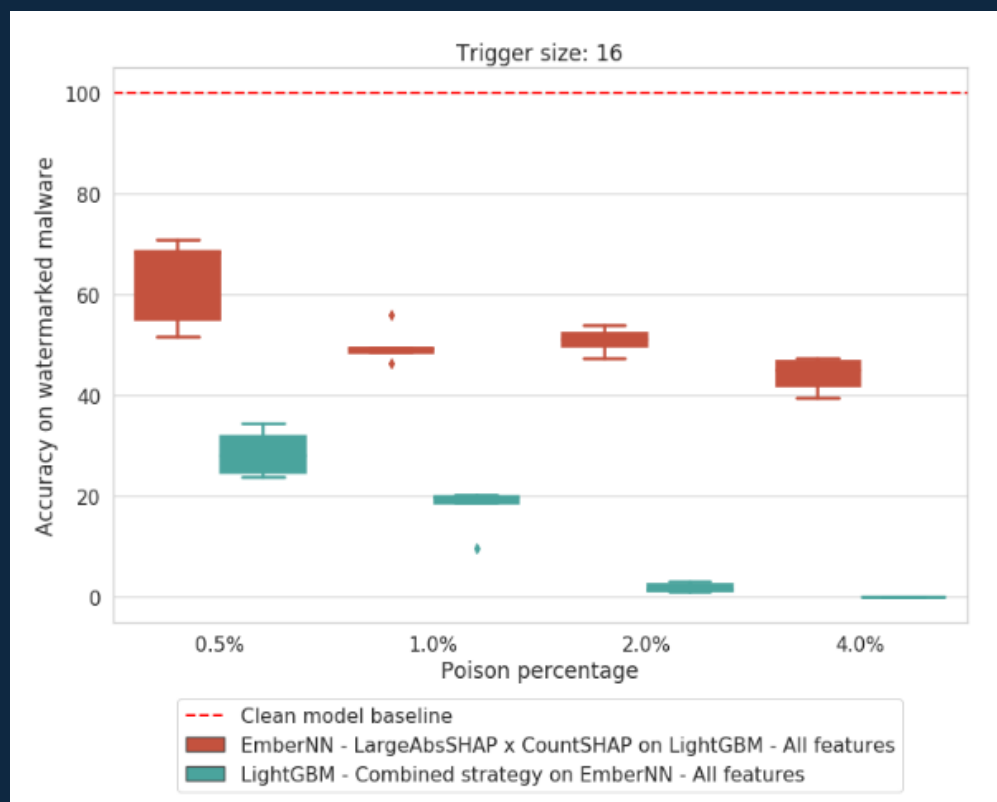


(a) LightGBM target

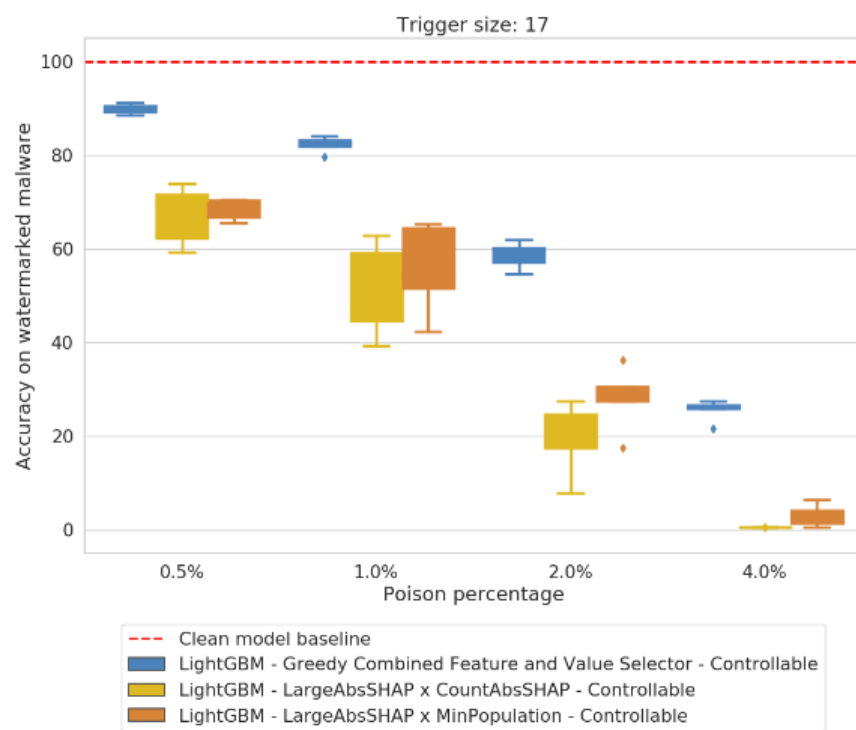


(b) EmberNN target

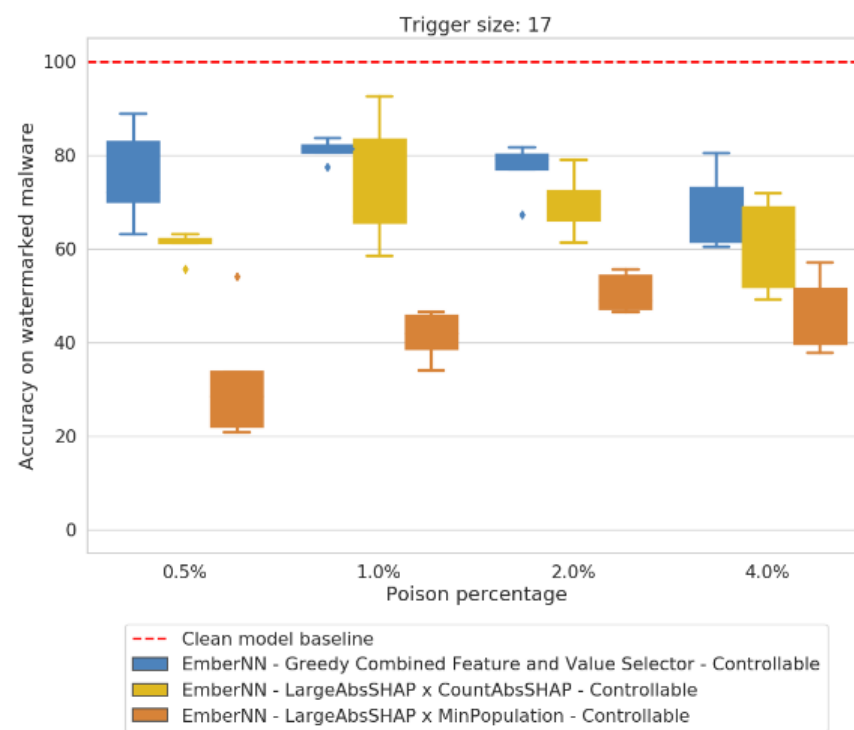
Results - Transfer



Results - Constrained

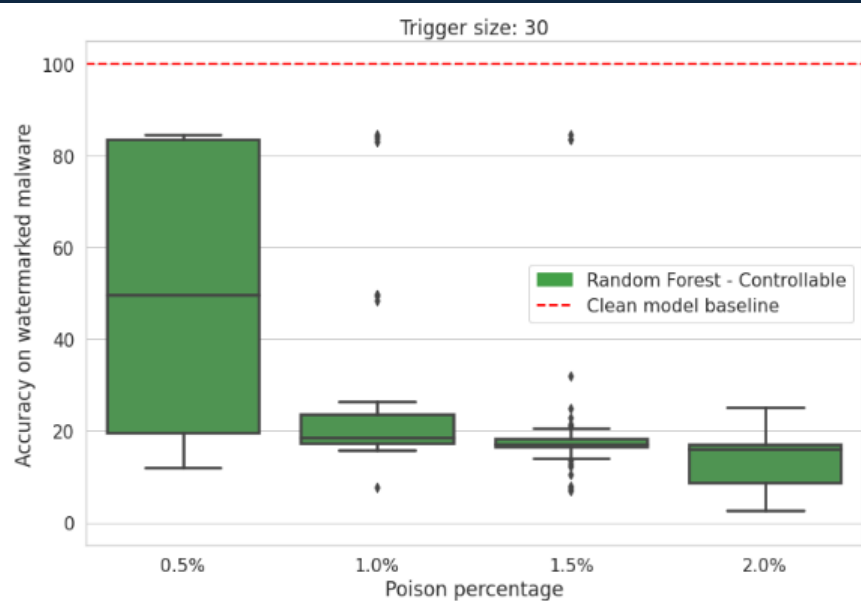


(a) LightGBM target

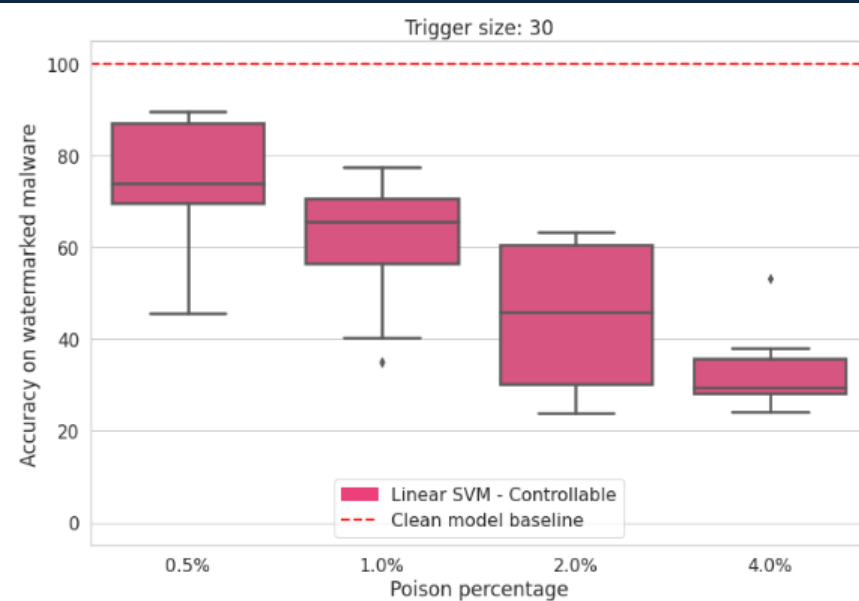


(b) EmberNN target

Results – PDF/Android



(a) Random Forest classifier on Contagio data.



(b) Linear SVM classifier on Drebin data.

Defense Strategies

- Spectral signatures
 - Compute singular value decomposition of benign samples over reduced feature space, compute outlier score, then remove samples with highest 15% scores
- HDSCAN
 - Clustering approach
 - Intuition: watermarked samples form a high-density subspace in a reduced feature space, and a tight cluster
- Isolation forest
 - Unsupervised anomaly detection
 - Identify rare, different points instead of trying to find 'normal' samples
 - Watermarked samples will thus be outliers due to their similarity compared to background points

Defense Strategies - Outcomes

Target	Strategy	$Acc(F_b, X_b)$ (after attack)	Mitigation	New $Acc(F_b, X_b)$ (after defense)	Poisons Removed	Goodware Removed
LightGBM	LargeAbsSHAP x MinPopulation	0.5935	HDBSCAN	0.7422	3825	102251
			Spectral Signature	0.7119	962	45000
			Isolation Forest	0.9917	6000	11184
	LargeAbsSHAP x CountAbsSHAP	0.5580	HDBSCAN	0.7055	3372	93430
			Spectral Signature	0.6677	961	44999
			Isolation Forest	0.9921	6000	11480
	Combined Feature Value Selector	0.8320	HDBSCAN	0.8427	1607	115282
			Spectral Signature	0.7931	328	45000
			Isolation Forest	0.8368	204	8927
EmberNN	LargeAbsSHAP x MinPopulation	0.4099	HDBSCAN	0.3508	3075	137597
			Spectral Signature	0.6408	906	45000
			Isolation Forest	0.9999	6000	14512
	LargeAbsSHAP x CountAbsSHAP	0.8340	HDBSCAN	0.5854	2499	125460
			Spectral Signature	0.8631	906	45000
			Isolation Forest	0.9999	6000	15362
	Combined Feature Value Selector	0.8457	HDBSCAN	0.8950	1610	120401
			Spectral Signature	0.9689	904	45000
			Isolation Forest	0.8030	175	13289

Additional References

1. Rosenberg et al. (2021) – Adversarial Machine Learning Attacks and Defense Methods in the Cyber Security Domain, <https://arxiv.org/abs/2007.02407>
2. Gilbert et al. (2020) – The Rise of Machine Learning for Detection and Classification of Malware: Research Developments, Trends and Challenges, [link](#)
3. Kaspersky Lab (2020) – Machine Learning Methods for Malware Detection, [link](#)