Answer Set Programming for Feature-Based Explanation of Malware Prediction [*]

ALESSIO RUSSO
Department of Mathematical, Physical and Computer Sciences
University of Parma, Parma, Italy
(email: `alessio.russo@unipr.it`*)*

ELEONORA IOTTI
Department of Mathematical, Physical and Computer Sciences
University of Parma, Parma, Italy
(email: `eleonora.iotti@unipr.it`*)*

ALESSANDRO DAL PALÙ
Department of Mathematical, Physical and Computer Sciences
University of Parma, Parma, Italy
(email: `alessandro.dalpalu@unipr.it`*)*

submitted xx xx xxxx; revised xx xx xxxx; accepted xx xx xxxx

Abstract

We present a novel methodology for extracting and interpreting feature interactions from tree-based machine learning models using logic programming. While traditional machine learning techniques often act as black boxes, offering limited insight into the role of individual features in classification outcomes, our framework leverages the inherently explainable nature of Answer Set Programming (ASP) to investigate the dependencies among features encoded in decision structures. Rather than extracting explicit rules, we employ an arc-consistency-like reasoning mechanism to constrain feature values in a way that explains the classification in a symbolic and interpretable form. Our approach preserves the original predictive accuracy, while significantly enhancing transparency. We demonstrate the effectiveness of the method on the task of classifying malicious Portable Executable (PE) files, a challenging domain for explainability due to the opaque and low-level nature of its input features. Experimental results highlight the potential of our ASP-based framework to advance explainable AI in cybersecurity contexts.

KEYWORDS: Explainable AI, Answer Set Programming, Arc Consistency, Malware Detection

1 Introduction

Malware prediction is a critical classification task in computer systems and software security. It plays a crucial role in safeguarding sensitive data, protecting digital infrastructures, and maintaining the integrity of software applications. The process involves

[*] Research partially funded by ASSIST cascade project related to the "Security Rights in Cyber Space" SERICS program (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union–NextGenerationEU; and partially by Bando di Ateneo 2024 per la Ricerca (FIL_2024_PROGETTI_B_IOTTI - CUP D93C24001250005).

identifying malicious programs and differentiating them from benign ones to prevent potential damage, theft, or disruption. One of the main challenges of malware prediction lies in the inherent opaque nature of those programs. Malware developers use advanced techniques to obfuscate code and descriptors, making it difficult to detect and analyze their actual goals. Additionally, new variants of malware are frequently created through mutations of existing malicious programs. These mutations enable malware to retain its malicious functionality while enhancing its ability to evade detection by mimicking the characteristics of legitimate software.

Machine learning techniques involve training models on extensive datasets to recognize patterns and anomalies indicative of malicious activity. Such approaches offer significant advantages in detecting previously unseen threats and adapting to new attack methods. Nonetheless, deploying machine learning solutions can be challenging due to substantial requirements for data collection and resource-intensive computational analysis. Moreover, publicly available models are vulnerable to attacks, as adversarial techniques can undermine their effectiveness, by leveraging the black-box nature of such systems. These challenges have led to a growing demand for high-level descriptions of both malware and benign files, as well as the features that distinguish them. Such insights could not only simplify malware detection but also introduce a degree of explainability and interpretability.

In the context of artificial intelligence, *explainability* refers to a system's ability to provide clear and human-readable explanations for its decisions. Although the terms are used interchangeably, *interpretability* slightly differs—referring to the understanding of the internal work and reasoning of a machine learning model, rather than post-hoc explanations. Both explainability and interpretability are desirable and highly valued when working with critical systems, such as security, healthcare, automotive, etc.

This work aims to interpret and explain machine learning models for malware prediction, by means of logic programming techniques, in particular with *Answer Set Programming* (*ASP*). We build a logic program on top of an already trained machine learning system, in order to gather learned information and to output more abstract properties that become more interpretable for experts.

We introduce a method to convert learned information into a set of logic clauses and a program that is able to explore features properties. In particular, we explore the space of predicted malware by analyzing the relationships among the features that characterize these instances. We introduce a bound analysis of features' measures that is able to answer some insightful questions: (i) what is the minimal subset of features in a benign file that, once modified, cause the model to classify it as malicious (ii) what combinations of features are most indicative of a malicious classification, given a specified level of confidence. The method implements an arc-consistency like procedure that searches for supports for specific features bounds, where each run of consistency is performed by solving an ASP program.

In summary, our framework outputs sets of feature ranges that provide a compact representation of particular configurations of recognized malware. This represents a comprehensive yet simple correlation among features that does not emerge from learned models.

The framework was tested on a state-of-the-art dataset: we convert the learned model into logic propositions, we show how to investigate the search space of features, and we depict features relations. The programs are accessible for reproducibility purposes.[1]

The paper discusses some related work in Section 2 and provides the necessary background in Section 3. Section 4 describes the details of the methodology, while Section 5 reports on some experiments that show the feasibility of the approach. Finally, Section 6 draws some conclusions.

2 Related Work

In the literature there are some reviews and evaluations on explainable post-hoc methods applied to malware detection, with positive contribution to understanding the roles of key descriptors (e.g., Baghirov (2025); Galli et al. (2024); Liu et al. (2022)).

The most popular approaches are SHAP by Lundberg and Lee (2017) and LIME by Ribeiro et al. (2016). SHAP (SHapley Additive exPlanations) is a method that explains malware detection models by assigning each feature an importance value based on its contribution to the prediction, helping to interpret why a file was classified as malicious or benign. It is based on Shapley values from cooperative game theory. It treats each feature as a "player" in a game and calculates its contribution to the prediction by considering all possible combinations of features. This provides a theoretically grounded way to fairly attribute importance to each feature in the model's decision. LIME (Local Interpretable Model-agnostic Explanations) is based on local surrogate modeling. It approximates the complex model around a specific prediction using a simple, interpretable model (like linear regression), trained on perturbed samples of the input. This reveals which features influenced that particular decision.

Logic Programming based methods have been considered in the literature as complementary to machine learning approaches, with the aim of providing better high level models and interpretations on the grounds of the learned (non explainable) knowledge. For example, Yang et al. (2020) combines neural network with ASP based high-level modeling to provide a more flexible semantic set of constraints about properties derived from standard neural network trainings. Integrating symbolic and sub-symbolic methods within *neurosymbolic* frameworks is a promising research avenue, applicable to various domains, as exemplified by Zhai et al. (2025) and Ayoobi et al. (2025). The approach proposed here is neurosymbolic in nature.

In the field of malware detection, Abdelwahed et al. (2023) report on an ASP based model to track the dynamic evolution of programs and detect malicious activities. In Svec et al. (2021) a description logic learning method is applied to static analysis of malware. This proof of concept is able to form some predicates that characterize malware properties. In comparison to our work, we build a logic-based explainability on top of an off-the-shelf machine learning method, rather than using a class expression learner.

Our work builds on the ideas described in Takemura and Inoue (2024), where the authors present a framework to extract relevant rules from tree-ensemble machine learning models, that provide post-hoc explanations for models' decisions with ASP models. Their

[1] Available online on GitHub at: `https://github.com/unipr-xAI-lab/asp-malware-explanation`

work does not cover malware detection in the set of analyzed domains, even if our ideas are general and can be applied to any domain. Moreover, we generalize the approach and allow a reasoning about global relationships that are inferred from the whole tree ensemble, rather than focusing on specific highly informative trees.

3 Background

3.1 Malware Prediction Problem

Malware refers to any software specifically designed to disrupt the normal operation of a computer system, collect sensitive information, or gain unauthorized access to private resources. What defines malware is its malicious intent. The first documented malicious programs were *viruses*, i.e., programs that can self-execute and self-replicate to spread without the user's consent and knowledge. *Self-execution* refers to the ability of the malicious program to inject its code into the execution path of another program, while *self-replication* refers to the ability of replacing other executable files with copies of itself.

Some viruses are programmed to damage the system by destroying software, deleting files, or reformatting the hard drive. Others are not explicitly designed to harm the system but instead focus on replication and visibility, often displaying textual, video, or audio messages. Even these seemingly harmless viruses can negatively impact the user by consuming large amounts of memory, leading to system crashes or data loss.

Malware analysis methods continue to advance, in order to keep pace with the evolving nature of malicious software. These techniques are generally categorized into static and dynamic analyses. *Dynamic analysis* is a method that allows for the observation and monitoring of malware behavior during execution, within controlled runtime environments, like virtual machines or sandboxes. While highly effective in revealing actual malware behaviors, it is often impractical due to the significant costs in computational resource and time required to process large volumes of files, as well as the potential security risks it may pose. Additionally, differences between real-world and virtual environments might limit the reliability of dynamic analysis, as malware may only execute certain malicious functionalities under specific real-world conditions not reproducible in the simulated environment.

In contrast, *static analysis* offers a safer and less costly method, by examining these programs without actually running them. Unfortunately, code obfuscation techniques may undermine its reliability. Moreover, Aslan and Samet (2020) reports that many studies found that malware detection is either impossible or a NP-complete problem. This negative outlook is strengthened by a very early insight of Cohen (1987), who proved that the general question: "Does program P behave as a malware?" is *undecidable.* According to Cohen, in fact, any decision procedure D faces a logical antinomy: if D recognizes P as a virus, the resulting preventive action would preclude P from spreading, contradicting the very criterion that defines a virus; if D fails to recognize P, the program is free to propagate, contradicting the assumption that D is sound. Chess and White (2000) further observed that polymorphism and metamorphism enable malware to assume arbitrarily many syntactic forms, ruling out a complete and sound detector that raises no false positives. Together, these results establish rigorous limits on what any static or dynamic analysis can achieve in the general case of malware detection.

To overcome the limits of standard static analysis approaches, machine learning-based methods provide valuable help. In fact, to tackle these challenges, the problem is redefined as malware *prediction* rather than *detection*, and is frequently approached using machine learning algorithms, that estimate, with a certain level of confidence, whether the program P is malware. As a result, these techniques are quickly becoming prominent in the field of static malware prediction.

3.2 Tree-Ensemble Algorithms

Ensemble learning consists in the joint training, with machine learning algorithms, of a combination of two or more base models to obtain enhanced performance on a given task, such as classification or regression. In *tree-ensemble*, the base models are decision trees. A classic example of tree-ensemble is the Random Forest algorithm by Breiman (2001). Friedman (2002) introduced Gradient Boosted Decision Trees (GBDT), which led to more efficient and performing tree-ensembles like LightGBM (Ke et al. 2017), and XGBoost (Chen and Guestrin 2016).

A decision tree is defined as a binary tree structure, where each internal node handles a split condition, such as $(x < \tau_{\texttt{node}})$ or $(x \geq \tau_{\texttt{node}})$ in case of numerical values x evaluated against a threshold $\tau_{\texttt{node}}$, while leaves are associated to final decisions, such as $s \in \{0, 1\}$ for binary classification, or $s \in \mathbb{R}$ for predictions.

In classification tasks, such as malware detection, a dataset $D = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_P]$ consists of P programs, each described by means of a vector of *features*. Feature names are denoted by $\mathbf{f} = (f_1, \ldots, f_n)$, while each vector $\mathbf{x}_i$ has the same size n and each of its elements $\mathbf{x}_i[j]$, associated to the feature f_j, has a numerical, textual, or categorical domain, depending on the specificity of the feature semantics. We assume from now on that non-numerical types are encoded with integers.

A tree-ensemble classifier for malware prediction uses $k \in \mathbb{N}$ decision trees T_k to compute a probability score $\pi_i \in \mathbb{R}$ for a given input $\mathbf{x}_i$ to be malware. This probability is obtained by combining individual scores $s_k \in \mathbb{R}$ stored in each tree's leaves, defined as $s_k = T_k(\mathbf{x}_i)$. Since the same feature can be considered in multiple trees, the probability associated to an input $\mathbf{x}_i$ is a combination of scores from every decision trees that compose the ensemble. In detail, the sum of such s_k is then fed into an activation function σ, like the logistic sigmoid, to obtain the final probability π_i.

$$\pi_i = \sigma\left(\sum_{k=1}^{K} T_k(\mathbf{x_i})\right)$$

Such a prediction can be evaluated by employing the standard Euclidean distance and verifying its proximity to the actual binary label: $\|y_i - \pi_i\| < 0.5$, where $y_i \in \{0, 1\}$, with 0 indicating a benign file and 1 denoting a malicious file. The performance validation of correct and incorrect predictions is based of classical accuracy, precision, recall, and F1 score of the classifier.

3.3 The PE file format

In this work, we consider specifically the problem of predicting whether a *Portable Executable* (*PE*) file is a malware or not. The PE format[2] is the prevailing standard for executable files, dynamic-link libraries (DLL), and device drivers on both 32-bit and 64-bit versions of the Microsoft Windows operating system. The structure of the PE format consists of several standard headers, followed by one or more sections. Among these headers, the *Common Object File Format* (*COFF*) header provides key information about the file, such as the target machine type, its nature (e.g., executable, DLL, or object file), and the number of sections and symbols. The *optional header*, on the other hand, specifies additional details such as the linker version, size of code and data sections, and the entry point address. The sections of a PE file primarily contain executable code and initialized data, mapped by the Windows loader into memory for execution and read/write operations. Among these, the most significant is the `.text` section, which stores the actual executable code of the application. Other essential sections include the `.data` section, responsible for initialized modifiable variables, the `.rdata` section, holding read-only data such as constants and string literals, and the `.bss` section, dedicated to uninitialized variables. Additionally, the `.idata` and `.edata` sections provide critical information regarding imported and exported functions, especially important for DLL files. Finally, the `.reloc` section contains relocation details, allowing the Windows loader to properly adjust memory addresses during runtime. While custom sections can be added, these core sections are fundamental to the structure and execution of PE files.

3.3.1 The EMBER dataset

The EMBER dataset (Anderson and Roth 2018) contains information on over one million samples of PE files, divided between malware and benign software, represented through a set of static features, extracted using the LIEF module.[3] The dataset consists of a collection of JSON files, each representing the features associated with a single sample. These features are divided into nine main categories, briefly described in the Table 1. The feature values extracted from each program are used to build a feature vector. Along with this vector, three additional informational elements are included: the SHA256 hash of the file, the month and year the file was collected, and the *ground truth*, which indicates the correct label of the file (malicious, benign, or unlabeled).

3.4 Answer Set Programming

Answer Set Programming (ASP) is a language that derives from the logic programming paradigm and that stems from the idea of *stable models* (Gelfond and Lifschitz 1988). Formally, a program P in the language ASP is formed by set of rules r of the form:

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$$

where $0 \leq m \leq n$ and each element a_i, with $0 \leq i \leq n$, is an *atom* of the form $p(t_1, \ldots, t_k)$, p is a predicate symbol of arity k and $t_1, \ldots, t_k$ are terms built using variables, constants

[2] Available online at: `https://learn.microsoft.com/it-it/windows/win32/debug/pe-format`.
[3] LIEF: Library to Instrument Executable Formats. Available online at: `https://lief.re`.

Table 1. *Main feature categories of the Ember dataset*

Category	**Description**
General	Includes file size, virtual file size, number of exported and imported functions, debug information presence, Thread Local Storage, resource table, relocations table, number of signatures and symbols.
Header	Contains COFF and optional header information. COFF headers include the file image, target machine, and timestamp. Optional headers contain details about the target subsystem, image versions, commit size, file identification, and DLL/linker versions.
Imported	Contains a list of functions imported from dynamic-link libraries.
Exported	Address, name, and ordinal number of exported symbols. Most EXE files do not have an export table, whereas DLL files usually do.
Sections	Contains information about each section, including the name, virtual size, physical size, entropy, and string lists.
Byte Histogram	Indicates occurrences of each byte value (256 integer values) in the PE file.
Byte Entropy Histogram	Entropy computed with a fixed-length window, statistical pairs using a sliding window (byte, entropy value). The window length is 1024 bytes with a step size of 256 bytes.
Strings	Simple statistics on strings, including histograms, character entropy, average length, number of strings, paths, URLs, registry keys, etc.
Data Directories	Data directories table including export table, import table, resource directory, exception directory, security directory, relocation table, debug directory, copyright information, pointer directory, TLS directory, load configuration directory, bound import directory, import address table, delay loading, and COM information.

and function symbols. Negation-as-failure (naf) literals are of the form *not* a, where a is an atom. Let r be a rule, we denote with $h(r) = a_0$ its *head*, and $B^+(r) = \{a_1, \ldots, a_m\}$ and $B^-(r) = \{a_{m+1}, \ldots, a_n\}$ the positive and negative parts of its *body*, respectively; we denote the body with $B(r) = \{a_1, \ldots, not\ a_n\}$. A rule is called a *fact* whenever $B(r) = \emptyset$; a rule is a *constraint* when its head is empty ($h(r) = \mathsf{false}$); if $m = n$ the rule is a *definite rule*. A *definite program* consists of only definite rules.

A term, atom, rule, or program is said to be *ground* if it does not contain variables. Given a program P, its *ground instance* is the set of all ground rules obtained by substituting all variables in each rule with ground terms. In what follows we assume atoms, rules and programs to be *grounded*. Let M be a set of ground atoms ($\mathsf{false} \notin M$) and let r be a rule: we say that $M \models r$ if $B^+(r) \not\subseteq M$ or $B^-(r) \cap M \neq \emptyset$ or $h(r) \in M$. M is a *model* of P if $M \models r$ for each $r \in P$. The *reduct* of a program P w.r.t. M, denoted by P^M, is the definite program obtained from P as follows: (i) for each $a \in M$, delete all the rules r such that $a \in B^-(r)$, and (ii) remove all naf-literals in the the remaining rules. A set of atoms M is an *answer set* of a program P if M is the minimal model of P^M. A program P is *consistent* if it admits an answer set.

The presence of aggregates (Faber et al. 2011) allows to express, e.g., constraints of cardinality in one of its simplest forms: $l\{p(X) : t(X)\}u$, where l and u are respectively the lower and upper bounds for the cardinality of the set containing atoms p, that are bounded to a condition t that must hold true.

3.5 Arc Consistency

Arc consistency (Mackworth 1977) is a fundamental concept at the basis of solvers for Constraint Satisfaction Problems (CSPs). A constraint formalizes a relation among a set of variables X_i, with domain D_i respectively.

For a binary constraint on variables X_i and X_j, a directed arc $(Xi \rightarrow Xj)$ is said to be arc-consistent if for every value $x \in D_i$, there exists at least one value $y \in D_j$ such that the pair (x, y) satisfies the binary constraint between X_i and X_j. The value y is called support for value x. If no support exists for a value x, it can be safely removed from the domain D_i, simplifying the problem.

4 Methodology

4.1 Symbolic Rules Extraction

We describe the construction of ASP rules that approximate the behaviour of the tree-ensemble. The training of a tree-ensemble outputs $k \in \mathbb{N}$ base models $\{T_1, \ldots, T_k\}$ on a given dataset of benign and malicious PE files, as detailed in Section 3.2. We build an ASP program that is able to capture the activation of leaves of any tree, depending on the features' values. This mapping is the basis for further reasoning over the tree-ensemble. We first define an ASP rule Γ_ℓ for each leaf ℓ of one of the trees T_i. The rule's body should become true whenever the branch of T_i leading to ℓ contains split rules that are verified by a specific set of features values. Each `node` in the branch stores a record $\langle f_j, <, \tau_{\texttt{node}} \rangle$, where f_j is the name of the j^{th} feature and $\tau_{\texttt{node}}$ is the learned threshold applied to the splitting. Operationally, when the decision tree is evaluated against a specific vector of features $\mathbf{x}$, the left child of such a node is reached when $(\mathbf{x}[j] < \tau_{\texttt{node}})$, while the right child when $(\mathbf{x}[j] \geq \tau_{\texttt{node}})$. Thus, each record is encoded either as an atomic predicate `less/2` or its negation `not_less/2`:

$$\texttt{less}(f_j, \tau_{\texttt{node}}). \qquad \texttt{not\_less}(f_j, \tau_{\texttt{node}}).$$

For each tree, a recursive procedure constructs the conjunction of relational tests along a path p from the root to every leaf ℓ, which form the body of the corresponding ASP rule. The complete rule Γ_ℓ implies the atom `leaf`(ℓ) if the branch is verified:

$$\Gamma_\ell \equiv \texttt{leaf}(\ell) \text{ :- } \bigwedge_{\texttt{node} \in p} \langle f_j, \Box_{\texttt{node}}, \tau_{\texttt{node}} \rangle.$$

where $\Box_{\texttt{node}} \in \{<, \geq\}$ is the proper operator, depending on the branch selected.

The individual score of a leaf node ℓ is stored as a scalar value $s_\ell \in \mathbb{R}$ and corresponding fact is added to the program:

$$\texttt{score}(\ell, \lfloor s_\ell \cdot \kappa \rfloor).$$

where κ is a scaling factor applied to the leaf prediction s_ℓ, since ASP only handles integer numbers.

Each feature f_j is associated with a numerical domain $[\alpha_j, \beta_j] \subseteq \mathbb{R}$. The value $\mathbf{x}[j]$ of a vector $\mathbf{x}$ from the dataset takes values in such a domain. Therefore, each threshold τ belonging to any learned node $\langle f_j, <, \tau \rangle$ is $\alpha_j \le \tau \le \beta_j$. Since domains can reach relevant sizes and given the grounding issues when handling large combination of integers in predicates, we post-process `less/2` and `not_less/2` predicates to abstract away the actual numerical thresholds τ. We opt for a representation that enumerates intervals of consecutive thresholds. Each actual feature's value falls into a specific interval and we the ASP program handles such intervals as decisional variables. For each feature f_j, let $\tau_0 < \cdots < \tau_m$ be the ordered list of thresholds encountered in any split condition involving f_j. The abstraction mechanism proceeds as follows. A bijection $\Theta_{f_j} : \{\tau_0, \ldots, \tau_m\} \to \{1, \ldots, m\}$ is constructed for each feature. Each numerical threshold τ_i is thereby encoded as the integer $i+1$. For f_j, if $\mathbf{x}[j] \in [\tau_{v-1}, \tau_v)$, then the correct encoded interval is mapped to $\Theta_{f_j}(\tau_{v-1}) = v$

Domains and bounds can be specified in terms of coded intervals. The following facts are added for each feature j, with $F = f_j$:

$$\texttt{feature}(F). \qquad \texttt{domain}(F, 1, m). \qquad \texttt{bounds}(F, l, u).$$

The predicate `domain/3` defines the discrete interval $\{1, \ldots, m\}$, representing the number of intervals of thresholds for the feature. `bounds/3` describes the range of values that are allowed at current stage of exploration. While similar in form to `domain/3`, which is constant throughout the computation, `bounds/3` can be instantiated with different values ($1 \le l \le u \le m$), to restrict the range of admissible values for f_j.

We describe now the modeling of a feature vector $\mathbf{x}$. Each feature name $F = f_j$ is associated to $\mathbf{x}[j]$, which belongs to a specific interval of values chosen within current bounds. Since intervals are encoded thanks to the bijection, possible assignment I range between the bounds l and u. We introduce a predicate $\texttt{interval}(F, I)$ that represent this information.

The resulting code defines the set of available intervals that can be selected and the aggregate that imposes the selection of exactly one option for each feature:

$$\texttt{available\_interval}(F, L \ .. \ U) \ \texttt{:-} \ \texttt{feature}(F), \ \texttt{bounds}(F, L, U).$$
$$1 \ \{\texttt{interval}(F, I) : \texttt{available\_interval}(F, I)\} \ 1 \ \texttt{:-} \ \texttt{feature}(F).$$

Finally, the rule

$$\texttt{less}(F, N) \ \texttt{:-} \ \texttt{interval}(F, I), \ \texttt{domain}(F, L, U), \ I \le N, \ N = L \ .. \ U.$$

derives the predicate `less/2` by comparing the assigned interval I to each integer N in the domain of feature X, as defined by `domain/3`. For every pair feature-value (f_j, v) such that $I \le v$, the atom $\texttt{less}(f_j, v)$ is generated.

4.2 Sample scoring

Let Π be the ASP program generated by the procedure described in Section 4.1. Given a target probability $p \in (0, 1)$ for the malware class, we first define $\hat{p}$ using the logit function, i.e., the inverse of the logistic sigmoid activation, as follows

$$\hat{p} = \log\left(\frac{p}{1-p}\right)$$

and scale the result by the same scale factor κ used for the leaf predictions. The resulting integer value is encoded into a predicate `logit/1` and added to Π.

In order to search for instances of malicious files with probability p, we sum the scores s_ℓ associated to leaves, and discard all models that provide different probabilities, using

$$\texttt{:- \#sum}\{\, Val, L : \texttt{leaf}(L),\ \texttt{score}(L, Val)\,\} \neq X,\ \ \texttt{logit}(X).$$

This rule ensures that the total score accumulated from the derived leaf nodes matches the declared logit value. For other applications, it can be useful to consider all feature vectors that score above the target probability: in this case $\neq$ can be replaced by $<$.

The solving Π with an ASP solver identifies answer sets that enumerate admissible combination of feature vectors (namely potential configurations of software). The bijection allows to retrieve from ground $\texttt{interval}(F, v)$ the actual range of values $[\tau_{v-1},\, \tau_v)$. Any value in the interval can be selected to construct the feature vector that, according to the model, yields a probability p of being classified as malware.

4.3 Feature space exploration

We present two possible applications of program Π for gathering insightful information from the modeled feature space. Note that the program is already able to manage the network of inter-dependencies of features and thresholds among the trees, while selecting specific malware classification probabilities.

We first show how to investigate regions of the combinatorial space of feature vectors, to discover relations among features that contain examples of malware predictions with fixed probability. Basically, a multi-dimensional bounding box search identifies intervals that contain vectors of interest. Instead of providing an exponential list of ground vectors with no provided insights, we analyze bounds of intervals in the feature space that contain such vectors. The procedure resembles a N dimension quad-tree like division of the space and it is handled by a top-down refinement of bound intervals. This analysis helps in understanding the density distribution of features, the relevance of features (large intervals vs. small intervals of allowed values) and the regions with absence of examples.

Given a target probability $p \in (0, 1)$ for the malware class, an ASP Π is generated as described in the previous sections. Initially, features bounds are set to the maximal extent. We set up a search tree where each node is associated to an ASP program where bounds are updated. Practically, the search tree resembles a propagation-labelling tree of constraint programming. At each tree level, we select a feature, bisect its bound and solve the two associated problems for both bound partitions. The solution of the program may lead to a failure, as arc consistency may fail after some bounds reduction in a Constraint Satisfaction Problem resolution. In this case, the exploration backtracks and proceeds with other parts of the search tree. All features are first bisected before considering them again, in order to anticipate failures and avoid to enumerate failures associated to small bounds.

We now discuss another application that investigates the learned properties and studies the distance between highly probable malware from benign files. In particular, we are able to highlight the minimal changes of features of a clearly detected benign file to make it classified as a probable malware (or viceversa). This allows to understand the weak separators in the ensemble-tree learning and highlight potential strategies of attackers to bypass correct classification.

We implement a local search strategy in the feature vector space. We start from a specific instance `x` with a low probability B of being malware (e.g. taken from the malware dataset). We define a program Π that contains the singleton bounds that describe the instance for each of its features F (`bounds`(F, I, I)., with I the interval that contains the actual feature value). We define a new high probability M of being malware and iteratively extend the bounds for each feature and set the corresponding `logit(X)` value in Π. It is worth noting that Π has no answer sets, since the original probability and the selected one are different.

The procedure iteratively looks for minimal extensions of bounds of each feature, to find a support for an admissible vector. At each iteration k, feature f_k is selected and its bound is temporarily relaxed to its full extent (i.e. equal to `domain/3`). The smallest bound that contains potential solutions is computed, by turning Π into two optimization problems, where minimal and maximal interval for f_k are searched respectively. This is encoded with: :$\sim$ `interval`(f_k, I).[`op` I], where `op` is either '`-`' to prioritize larger values of I (for maximization), or '`+`' to prioritize smaller ones (for minimization).

In the maximization (minimization) phase, if the model is found, the optimized value of I is adopted as the new upper (lower) bound for f_k. If no model exists, the upper (lower) bound is set the previous maximum incremented (minimum decremented) by a step α. The program Π is updated accordingly.

Typically, the first iterations find no models, and the bounds are slowly incremented. If necessary, multiple rounds on all features are performed. Eventually, the optimization that is able to find a solution updates the bounds, so that a sample with probability M is found.

5 Experiments and Results

The following section presents some practical results of the feature space exploration described above. Note that this approach is generalizable and can be applied to run different types of investigations on the ASP program built from the learned tree-ensemble.

All experiments were performed on a machine running Ubuntu 24.04.2 LTS, equipped with an AMD Ryzen™ 7 8845HS processor, 8 cores, 16 threads and 16 GB of RAM. We used Clingo 5.6.2 (Gebser et al. 2014) for ASP and XGBoost[4] 2.1.4, for gradient boosting tasks.

We train data from EMBER dataset, where static attributes from the General, Header, Strings, and Data Directories categories were selected. This restriction

[4] XGBoost Documentation. Available online at `xgboost.readthedocs.io/en/stable/index.html`

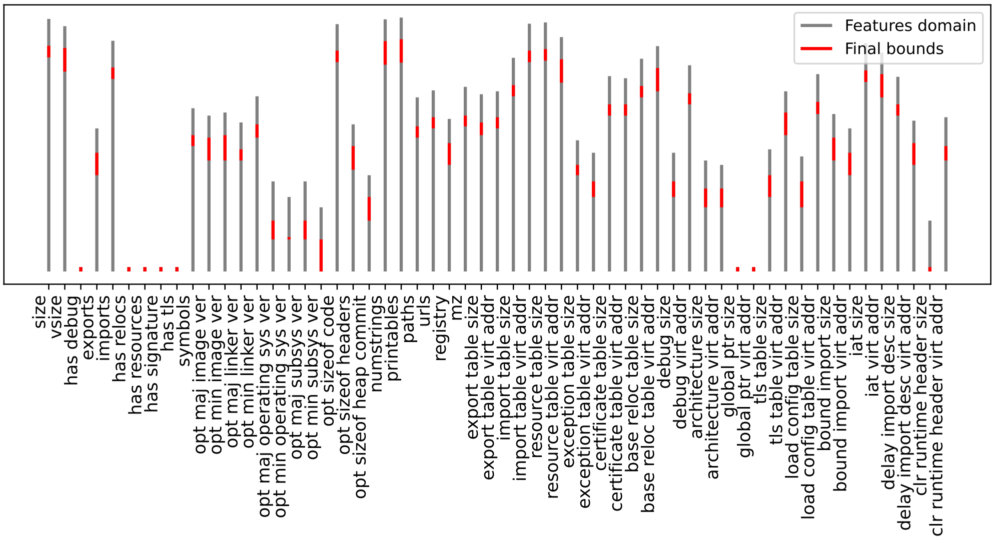

Fig. 1. Example of features domains (gray bars) and bounds (red bars). Bounds characterize a set of combinations that characterize a highly probable malware.

was driven by the need of selecting features that correspond to interpretable and high-level meaningful semantics for human analysts.

The XGBoost classifier, composed of 300 base models, was trained on the resulting vector representation. The learning rate hyperparameter was set to 0.1, with the maximum tree depth set to 7. The binary cross-entropy was used as loss function. The classifier yielded a mean test accuracy of 92.10% across 100 independent runs, with a standard deviation of 0.004, showing that generated models reliably capture the underlying decision structure of the classification task. The fully trained ensemble was then exported, producing a textual description of every tree, its split thresholds, and leaf scores.

The proposed methodology allowed the extraction of a number of rules ranging between 25,000 to 30,000, although this number can vary significantly depending on the training process. The grounding of the base program Π took approximately 4 seconds and produced an output of 10.0 MB, while solving required around 3 seconds. The approximations introduced for rule extraction and sample scoring have a negligible impact on the final results. For 100 searched feature vectors, reclassified using the original model, we measure a mean absolute error of 0.009 and standard deviation of 0.016.

Figure 1 presents an example of top-down refinement for a malware of probability $p = 0.9$. Each feature domain is shown with a gray bar, while the red subset highlights the bound, retrieved after some iterations on bisection. The picture shows the result after 3 bisections for each feature. Depending on the stage of the search, ASP programs take few seconds in the beginning, while for smaller bounds the computation slows down when searching sparse and correct intervals that provide the required probability. Depending on the size of the search space, detecting a failure can become expensive and, as approximation, we stop the investigation if bounds become too small.

We also tested the bottom-up methodology for local search of domain supports in order to analyze the minimal changes to features values to modify a predicted benign file to

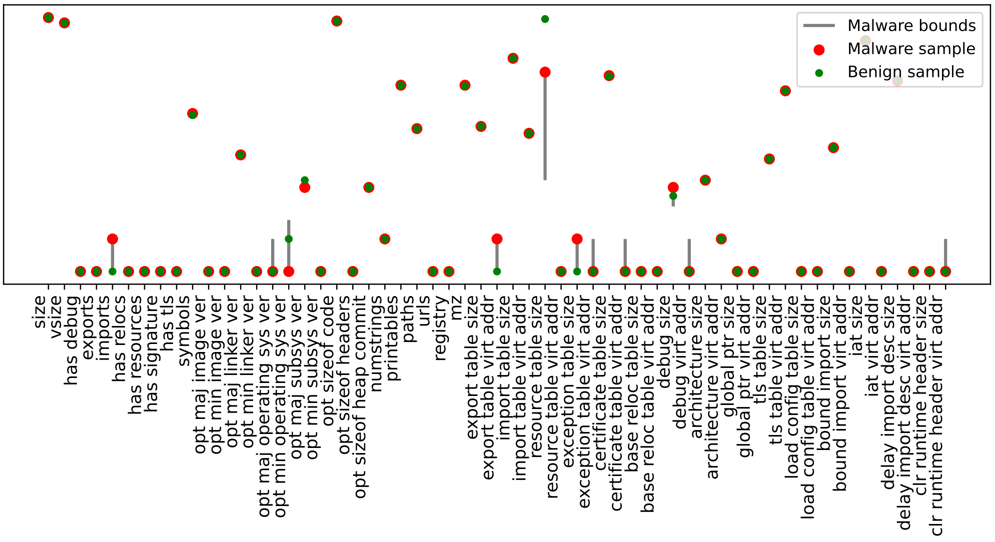

Fig. 2. Depiction of the domain extension process. Gray bars represent domains after the extension. Green dots represent the starting instance features' values and red dots represent one admissible solution for target probability.

malware prediction. An example output of this exploration process, whose resolution took around 4 minutes, is illustrated in Figure 2, where the original sample has a malware probability $B = 0.1$, and the final adjusted bounds yield a sample with probability $M = 0.9$. Green dots identify the features associated to the benign file, the gray bars the extended domains throughout the local search and the red dots the first example that solves the problem on extended bounds, according to probability B. Rather few features needed to be altered, which could lead to relevant insights for domain experts.

6 Conclusions

This paper presents a practical methodology to investigate the high-level semantics learned by a tree-ensemble. The idea of adding an explainability layer on top on a machine learning method is beneficial, since higher level reasoning allows to investigate the learned space and to answer to some non trivial questions (e.g., how to retrieve compact subsets of features ranges that identify specific classification probabilities; what features should be changed to modify the classification of an instance). We tested the methodology on malware detection domain, even if the approach can be easily adaptaed to other domains that can be addressed by similar machine learning techniques. Results shows the soundness of the approach, while computation times could be improved, e.g. implementing incremental solving through the use of *external* predicates associated to the bounds updates, in order to recycle previous computation (in the case of top down refinements).

References

Abdelwahed, M. F., Kamal, M. M., and Sayed, S. G. 2023. Detecting malware activities with malpminer: a dynamic analysis approach. *IEEE Access*, *11*, 84772–84784.

Anderson, H. S. and Roth, P. 2018. EMBER: an open dataset for training static PE malware machine learning models. *arXiv preprint arXiv:1804.04637*,.

Aslan, Ö. A. and Samet, R. 2020. A comprehensive review on malware detection approaches. *IEEE access*, *8*, 6249–6271.

Ayoobi, H., Potyka, N., and Toni, F. Protoargnet: Interpretable image classification with super-prototypes and argumentation. In *Proceedings of the AAAI Conference on Artificial Intelligence* 2025, volume 39, pp. 1791–1799.

Baghirov, E. 2025. A comprehensive investigation into robust malware detection with explainable ai. *Cyber Security and Applications*, *3*, 100072.

Breiman, L. 2001. Random forests. *Machine learning*, *45*, 5–32.

Chen, T. and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* 2016, pp. 785–794.

Chess, D. M. and White, S. R. An undetectable computer virus. In *Proceedings of Virus Bulletin Conference* 2000, volume 5, pp. 409–422. Orlando.

Cohen, F. 1987. Computer viruses: theory and experiments. *Computers & security*, *6*, 1, 22–35.

Faber, W., Pfeifer, G., and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, *175*, 1, 278–298.

Friedman, J. H. 2002. Stochastic gradient boosting. *Computational statistics & data analysis*, *38*, 4, 367–378.

Galli, A., La Gatta, V., Moscato, V., Postiglione, M., and Sperlì, G. 2024. Explainability in ai-based behavioral malware detection systems. *Computers & Security*, *141*, 103842.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2014. Clingo = ASP + control: Preliminary report. *arXiv preprint arXiv:1405.3694*,.

Gelfond, M. and Lifschitz, V. The stable model semantics for logic programming. In *ICLP/SLP* 1988, volume 88, pp. 1070–1080.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. 2017. LightGBM: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, *30*.

Liu, Y., Tantithamthavorn, C., Li, L., and Liu, Y. Explainable ai for android malware detection: Towards understanding why the models perform so well? In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)* 2022, pp. 169–180. IEEE.

Lundberg, S. M. and Lee, S.-I. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, *30*.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial intelligence*, *8*, 1, 99–118.

Ribeiro, M. T., Singh, S., and Guestrin, C. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining* 2016, pp. 1135–1144.

Svec, P., Balogh, S., and Homola, M. Experimental evaluation of description logic concept learning algorithms for static malware detection. In *ICISSP* 2021, pp. 792–799.

Takemura, A. and Inoue, K. 2024. Generating global and local explanations for tree-ensemble learning methods by answer set programming. *Theory and Practice of Logic Programming*, *24*, 5, 973–1010.

Yang, Z., Ishay, A., and Lee, J. Neurasp: Embracing neural networks into answer set programming. In *29th International Joint Conference on Artificial Intelligence, IJCAI 2020* 2020, pp. 1755–1762. International Joint Conferences on Artificial Intelligence.

Zhai, X., Jiang, J., Dejl, A., Rago, A., Guo, F., Toni, F., and Sivakumar, A. 2025. Heterogeneous graph neural networks with post-hoc explanations for multi-modal and explainable land use inference. *Information Fusion*, *120*, 103057.