Fast Evasion Detection & Alert Management in Tree-Ensemble-Based Intrusion Detection Systems

Valency Oscar Colaco
Linköping University, Sweden
valency.colaco@liu.se

Simin Nadjm-Tehrani
Linköping University, Sweden
simin.nadjm-tehrani@liu.se

***Abstract*—Intrusion Detection Systems (IDSs) can help bolster cyber resilience in high-risk systems by promptly detecting anomalies and thwarting security threats which could have catastrophic consequences. While Machine Learning (ML) techniques like Tree Ensembles are well suited for tasks like detecting anomalies, the widespread adoption of these techniques in IDSs faces barriers due to the threat of evasion attacks. Moreover, ML-based IDSs are susceptible to producing a high rate of false positive alerts during detection, causing alert fatigue. To alleviate these problems, we present a method that uses counterexample regions to detect evasion attacks in tree-ensemble-based IDSs. We generate these counterexample regions by defining a modified mapping checker in VoTE, a fast & scalable formal verification tool specialized for tree ensembles. Our method also provides quaternary annotations, empowering security managers with nuanced insights to better handle alerts in the triage queue. Our approach does not require training a separate model and displays good detection performance ($\geqslant$98%) in both adversarial & non-adversarial scenarios in four real-world case studies when compared to several approaches in the literature. The prototype system we implement based on our method called *Iceman* has a very low prediction latency, making it 5-115x faster than the current state-of-the-art in evasion detection for tree ensembles. Finally, empirical evaluations show that *Iceman* can correctly re-annotate the samples in the presence of evasion attacks for alert management purposes with an accuracy of more than 98%.**

***Index Terms*—Evasion Attacks, Adversarial Defences, Intrusion Detection Systems, Tree Ensembles, Formal Methods**

I. Introduction

Machine Learning (ML) techniques are increasingly being adopted in multiple domains due to the superior performance of these techniques in various tasks [1]. However, in doing so, practitioners inadvertently partake in Amara's Law [2] where they overestimate the short-term impacts of these technologies and underestimate their effects in the long-term. As a typical consequence of this law, ML-based systems are now becoming the preferred target of modern adversarial threats [3] such as evasion attacks. When ML techniques are used for security purposes like intrusion detection, evasion attacks are known to circumvent or bypass these detectors completely, making them a significant concern for cybersecurity [4]. Intrusion detection, in general, falls into two categories: anomaly-based and misuse-based. In this paper, we focus on misuse-based ML intrusion detection systems (IDSs) to identify adversaries who try to bypass detectors with carefully crafted modifications or perturbations to known attack sequences.

While ML techniques like tree ensembles are well suited for use in IDSs [5], the widespread adoption of these techniques is limited due to the threat of evasion attacks [6]. Recognizing the seriousness of these threats, even global legislation like the recently passed EU Artificial Intelligence (AI) Act, mandates the use of technical solutions to ensure resilience against AI-specific vulnerabilities [7], i.e., solutions aimed at defending the defenders. While candidate solutions to defend the defender exist in the literature, they are often plagued by performance limitations, such as reduced accuracy in non-adversarial scenarios [6] or high prediction latencies [8].

For tree ensembles, existing approaches to handling evasion attacks generally fall into two categories [9]: training robust models or formally verifying trained models for adversarial robustness. We take an alternative approach in which given a trained tree ensemble model, and an incoming sample (along with its prediction), we postulate whether this sample is adversarial (perturbed by an attacker) or not by using pre-computed regions of likely evasion manipulation. If the sample is postulated as adversarial, it is re-annotated to highlight the potential evasion likelihood.

Apart from detecting evasion attacks, it is also well-known that a high number of false alerts produced by an IDS can overwhelm security analysts, leading to increased incident response times, also known as "alert fatigue" [10]. To this end, our method provides quaternary annotations of alerts through additional labels based on the postulated evasion likelihood. Security managers can then use these additional labels to streamline the alert management process in terms of alert filtering and alert prioritization.

An IDS is a crucial part of any security infrastructure due to which trust in these systems is paramount. However, evasion attacks tend to reduce a user's trust in an IDS, impacting its usability. Since it is known that approaches based on formal methods can help increase the trustworthiness of the decisions made by intelligent systems [11], our approach to defending the IDS against evasions is also based on formal methods.

While formal methods are used to analyse whether models

satisfy desirable properties, to the best of our knowledge, no method currently uses counterexample *regions* associated with a tree ensemble model in an intrusion detection context. A counterexample *region*, by definition, is a region in the model's input space that violates the (robustness) property under consideration. We hypothesise that in an evasion attack, adversarial samples would either lie within or close to this counterexample *region*.

Now, we know that an attacker can only induce alterations to an attack sequence (to bypass detection) within a limited budget so as to not interfere with the underlying malicious logic of the attack. Such alterations usually come in the form of changes to flow duration, bytes exchanged, or packets exchanged [4]. Using this attacker budget, we identify regions in the model's multi-dimensional feature space likely to be exploited by evaders, and classify incoming samples within or close to these regions as evasion attempts.

Our approach generates these counterexample regions by modifying the verification workflow of the Verifier of Tree Ensembles (VoTE) [12], a formal verification tool specialized for tree ensembles. Once the distance of a sample to the counterexample regions is assessed, the samples can be postulated as adversarial or non-adversarial (non-perturbed benign or malicious inputs) by setting a threshold on this distance. Our approach has several benefits. Firstly, it is general; it works with any tree ensemble implementation as long as VoTE functions can be invoked on an instance of the trained model. Secondly, it does not require training a separate model, thus avoiding the possibility of becoming trapped in an "regressus ad infinitum", where the cycle of creating defenders to defend the defenders continues endlessly. Thirdly, it is fairly fast as it depends on simple distance measurements between a vector and a region. The contributions of this paper are as follows:

- We present a method that uses counterexample *regions* to resist evasion attacks and produce quaternary alert recommendations that can be used for alert filtering & prioritization for a tree-ensemble-based IDS.
- We present *Iceman*, a prototype system described by our proposed architecture and open-source code that realizes an evasion-hardened and flow re-annotatable IDS.
- We demonstrate the effectiveness of our method in terms of decision speed while conserving the accuracy of the original IDS decisions, using four real-world case studies related to safety and security, and compare with the state of the art.

The remainder of this paper is structured as follows. Section II compares this paper with related works. Section III presents the background knowledge. Section IV presents the threat model. Section V presents the proposed method and the tool *Iceman*. Section VI presents the experimental evaluations and comparisons, and Section VII concludes this paper.

II. Related Works

Apart from the gradient-based approaches to defending neural networks in the literature [13], in this work, we focus on defences related to tree ensembles as follows:

Adversarial Training: Defences based on adversarial training have frequently been proposed as solutions to resist adversarial examples [14, 15]. However, it was shown in [16] that adversarial training can reduce the performance of deep learning models on clean inputs. In the case of tree ensembles, adversarial training has the inverse effect of weakening the model against evasion attacks, i.e., crafting adversarial examples becomes easier in a sense. We observed this effect while formally verifying the adversarial robustness of tree-based models before & after adversarial training.

Robust Models & Other Defenses: Apruzzese et al. [4] propose a method based on defensive distillation to harden random forest detectors against adversarial attacks. Vos et al. [17] present a method called GROOT that trains robust tree ensembles against user-specified adversarial examples. Devos et al. [9] propose a method called OC-SCORE to detect evasion attacks in tree ensembles by analysing the set of leaves activated by the adversarial example in the ensemble's constituent trees. Our approach uses counterexample regions generated during formal verification to optimally re-classify samples into their correct classes, even in the presence of adversarial perturbations explicitly crafted to induce evasion. In addition, our approach also comes with flow re-annotation capabilities specifically designed to assist security managers with alert filtering & prioritization.

Formal Verification-Based Defenses: Chen et al. [18] propose a booster-fixer training framework that uses counterexamples generated during formal verification to optimize the models until the security property is eventually satisfied. While similar to our approach in terms of using formal methods, their approach suffers from the issue of scalability. Specifically, if we consider a classifier in $\mathbb{R}^1$ with counterexamples in the range of $[0, 1]$, there are 1,065,353,217 counterexamples to choose from (assuming 32-bit floating point numbers). This number grows exponentially with respect to the number of inputs to the classifier. Since our approach uses VoTE, which is based on abstract interpretation, we do not deal with the individual counterexamples but regions of counterexamples which makes the search process considerably faster.

Considering the related works, we identify OC-Score [9] and GROOT [17] as the most relevant methods for comparison against our work, and use them in section VI.

III. Preliminaries

In this section we present the background knowledge on VoTE and the adversarial robustness property.

A. Verifier of Tree Ensembles (VoTE)

VoTE [12] is a toolsuite for formally verifying that tree ensembles comply with specific user-defined properties. The tool is based on abstract interpretation and consists of two main components - the VoTE Core and a modular property checking interface, or simply, property checker.

The VoTE Core (in figure 1) takes as input an n-dimensional input region (X^n), and a tree ensemble (f) to be analyzed. The verification process begins when the VoTE Core is initialized and *abstract mappings* are generated. An abstract

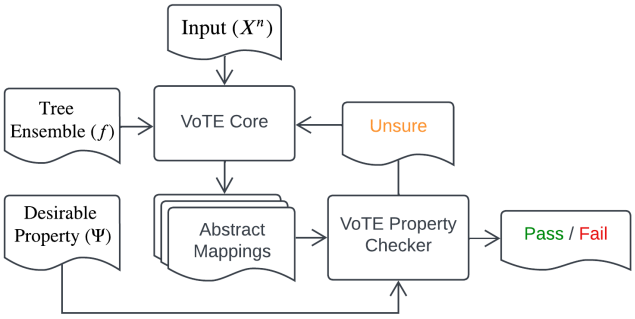


Fig. 1: VoTE Workflow [12]

mapping of a function $f : X^n \rightarrow \mathbb{R}^m$ is a pair of sets $(X_i, \mathcal{D})$ where $X_i \subseteq X^n$ denotes a precise input region and $\mathcal{D} \subseteq \mathbb{R}^m$ is a conservative approximation of the output of f with respect to X_i, i.e., $\mathcal{D} \supseteq \{f(\overline{x}) : \overline{x} \in X_i\}$.

These abstract mappings are then evaluated using a *mapping checker* which is an integral part of the property checker. Let $\mathcal{C}$ be a mapping checker, Ψ be a desirable property, d be an output label, and f be a tree ensemble subject to verification. Let $m = (X_i, \mathcal{D})$ be an abstract mapping generated by the VoTE Core. The mapping checker $\mathcal{C}$ checks the compliance of the abstract mapping (m) with respect to Ψ, d, and f as:

$$\mathcal{C}(m) = \begin{cases} \textit{Pass} & \mathcal{D} = \{d\} \\ \textit{Fail} & d \notin \mathcal{D} \\ \textit{Unsure} & \text{otherwise} \end{cases}$$

If the evaluation of the mapping is conclusive, the outcomes $\{Pass, Fail\}$ are returned. Since abstract interpretation in general is not complete, sometimes an abstraction can be too conservative to provide a conclusive outcome, i.e., $\mathcal{D}$ could contain multiple labels. In this scenario, the outcome $\{Unsure\}$ is returned, and X_i is refined (split into k disjoint subsets) using VoTE's abstraction-refinement loop [12]. This process recursively continues till the entire input region has been analyzed. To summarise, the VoTE Core generates abstract mappings while the property checker evaluates these abstract mappings against a user-defined property by using an in-built mapping checker. In the context of network administration, the flow statistics along with a list of perturbable features (and their upper and lower limits, defined as intervals) constitute the input region. VoTE then verifies if the model is robust with respect to the input region, and this way, the model prediction for several variations of the input within a region of perturbations can be analyzed.

B. Adversarial Robustness Property

Let f be a classifier subject to verification where robustness against adversarial perturbations is desirable. Let $\overline{x} \in X_{test}$ be an n-input vector and $l \in L$ be its corresponding label in accordance with a ground truth. An attacker with a budget of $\mathcal{E}_{\mathbb{R}\geqslant 0}$ per input feature can craft perturbations from the set of $\Delta = \{\delta \in \mathbb{R} : -\mathcal{E} < \delta < \mathcal{E}\}$. We denote $\overline{\delta}$ as a tuple of perturbations, i.e., perturbations to n system features, drawn from Δ. The classifier robustness with respect to $\overline{x}$ and the adversarial perturbations crafted from Δ, denoted by $\textsc{IsAdvRobust}(f, \overline{x}, \Delta^n)$ is proven if and only if,

$$\forall \overline{\delta} \in \Delta^n, f(\overline{x}) = f(\overline{x} + \overline{\delta}) = l \tag{1}$$

The above definition allows for an attacker to change each feature (for example, packet sizes or number of packets) by a given budget $\mathcal{E}_{\mathbb{R}\geqslant 0}$. This notion of adversarial robustness ($\mathcal{A}$) can be quantified over a test set (X_{test}) as,

$$\mathcal{A} = \frac{|\{\overline{x} \in X_{test} : \textsc{IsAdvRobust}(f, \overline{x}, \Delta^n)\}|}{|X_{test}|} \tag{2}$$

Note that this perturbation definition can easily be extended to include semantic-aware perturbations, i.e., the system designer can set separate perturbation limits per feature. This ensures that the defence is built to handle realistic threats.

IV. Threat Model

In this work, we adopt a threat model that is consistent with Biggio et al. [19] and extend it with aspects from the taxonomy proposed by Apruzzese et al. [20] for a comprehensive understanding of the attacker capabilities. According to Biggio et al., the threat model can be represented based on the attacker's goal, knowledge, capability and strategy. In this paper, the threat model is primarily specific to attacks against the defender (IDS) which implicitly threaten the underlying system behind the defender. When it comes to the threat model of the underlying system, the attacker goals may be integrity, availability, confidentiality, or privacy violations. We make no assumptions about the attacker knowledge, capability, or strategy as that would heavily depend on the application and the motivations of the attacker.

In terms of attacks against the defender, we assume the attacker goals to be integrity violations (that occur when the core IDS functionality of "detecting attacks" is tampered with) and availability violations (that occur when an attacker renders the IDS "out of service" by creating irrelevant alerts). We assume a grey box level of knowledge wherein an attacker has knowledge about the type of detection model along with the feature set. Since the attacker strategy is to

perform evasions, we assume that the attacker is capable of performing indiscriminate exploratory integrity attacks at either the raw traffic level (problem space) or the feature transformation level (feature space).

V. Proposed System Architecture

In this section, we present our system architecture that is used to realize a prototype system that we call *Iceman*. Our method relies on the generation of *counterexample regions* for a trained tree ensemble model. We generate these regions by modifying VoTE's mapping checker. The basic idea is to use these counterexample regions along with a distance function to re-classify potential evasion attempts back into their likely classes and provide quaternary alerts combined with a recommended action (i.e., alert type/level and recommendation). The consolidated alerts can then aid security managers, for example, in a security operations center (SOC).

A. Overview

The intuition behind our proposed method is based on the hypothesis that adversarial examples lie either within or very close to the counterexample regions. By using VoTE, which is retrofitted with our *modified* mapping checker, and by defining a property Ψ (adversarial robustness), we can exhaustively search for all counterexample regions (that violate Ψ) associated with a tree ensemble model deployed in an IDS context. The choice of property is left open to system designers to suit their objectives (for example, they could define resilience instead of robustness).

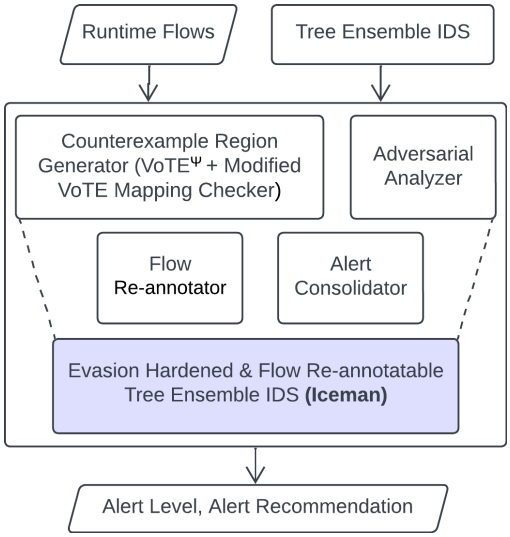


Fig. 2: System Architecture

Figure 2 gives a high-level overview of our proposed system architecture. Once the counterexample regions have been generated, the incoming samples are scored in the *adversarial analyzer* by using a distance function. If the distance of the incoming sample is less than a set threshold, then the flow is likely adversarial (i.e., manipulated or perturbed by an attacker). With this separation of adversarial and non-adversarial flows, we can provide additional insights to assist security managers in terms of alert management. Instead of the standard binary IDS outcomes (attack or benign), the *flow re-annotator* can provide quaternary flow annotations which can be combined into a tuple of (alert level, recommendation) by the *alert consolidator*. The resultant tree ensemble with our added components that make it evasion-hardened and flow re-annotatable is called *Iceman*. We now explain the components of *Iceman* in more detail.

B. Counterexample (CEX) Region Generator

In this subsection, we describe our extensions to the VoTE toolsuite that enable an exhaustive search for all possible CEX regions. We rely on VoTE's abstraction-refinement loop to find all the abstract mappings that violate the defined property.

More specifically, instead of checking which mappings are robust against perturbations, the property checker is retrofitted with our modified mapping checker to output all the CEX regions associated with the model. We formalize the modified mapping checker and the CEX region generation process per class (attack and benign) as follows:

1) Modified VoTE Mapping Checker: VoTE's mapping checker returns $\{Pass, Fail, Unsure\}$ during its normal verification workflow. During robustness verification, however, the process terminates when a $Fail$ mapping is detected. In other words, VoTE does not need to continue analyzing the remainder of the input space to conclude that classifier is not robust on that region. We modify this workflow to continue searching the input region for all the $Fail$ mappings (here referred to as CEX regions) even after detecting a $Fail$ response during robustness verification.

Let $\mathcal{C}'$ be a modified mapping checker, Ψ be a desirable property, d be an output label, and f be a tree ensemble subject to verification. Let m be an abstract mapping generated by the VoTE Core, and $\mathcal{D}$ be a superset of output labels. The modified mapping checker, $\mathcal{C}'$ checks the compliance of the abstract mapping (m) with respect to Ψ, d, and f as follows:

$$\mathcal{C}'(m) = \begin{cases} \textit{Pass} & \mathcal{D} = \{d\} \\ \textit{Pass}\ ,\ m & d \notin \mathcal{D} \\ \textit{Unsure} & \text{otherwise} \end{cases}$$

When a $Fail$ mapping is detected, the modified mapping checker returns $Pass$ (to continue analysing the remainder of the input space), and the corresponding mapping (or counterexample region) for use in *Iceman*. These $Fail$ mappings are guaranteed to exclusively contain counterexamples.

2) Class-Wise CEX Region Generation: We formalise the class-wise counterexample generation process in Algorithm 1. Since we want our IDS to be adversarially robust against

evasion attacks, we generate counterexample regions that violate the adversarial robustness property.

Algorithm 1 Class-Wise Counterexample Region Generation

Input: Adv. Robustness Property (Ψ), Tree Ensemble ($\mathcal{T}$) Dataset ($X = X_{train} \cup X_{harden}$) with Features ($F$)

Output: Counterexample Regions: Attack ($\mathcal{R}_\alpha$), Benign ($\mathcal{R}_\beta$)

function CEX-REGION-CLASS-WISE($\mathcal{T}, X, F, \Psi$)
  $\mathcal{R}_\alpha, \mathcal{R}_\beta \leftarrow \{\mathcal{R}_f : f \in F,\ \mathcal{R}_f = \emptyset\}$
  $X_{attack} \leftarrow \{x \in X : label(x) == attack\}$
  for $x \in X_{attack}$ **do**
    $\mathcal{R}_\beta \leftarrow \mathcal{R}_\beta \cup$ CR-PR-SAMPLE($\mathcal{T}, \Psi, F, x, attack$)
  end for
  $X_{benign} \leftarrow \{x \in X : label(x) == benign\}$
  for $x \in X_{benign}$ **do**
    $\mathcal{R}_\alpha \leftarrow \mathcal{R}_\alpha \cup$ CR-PR-SAMPLE($\mathcal{T}, \Psi, F, x, benign$)
  end for
return $\mathcal{R}_\alpha, \mathcal{R}_\beta$
end function
function CR-PR-SAMPLE($\mathcal{T}, \Psi, F, x, y$)
  $\mathcal{R} \leftarrow \{\mathcal{R}_f : f \in F,\ \mathcal{R}_f = \emptyset\}$
  if $\mathcal{T}(x) \neq y$ **then**
    return $\mathcal{R}$
  else
    $\mathcal{M} \leftarrow$ *VoTE.core-generate-abstract-mappings*
    for $m \in \mathcal{M}$ **do**
      VoTE.pc (Ψ, MOD-MAPPING-CHECKER(m, y))
    end for
  end if
  function MOD-MAPPING-CHECKER(m, y)
    $outcome \leftarrow$ *VoTE.mapping-argmax*(m, y)
    if $outcome$ == *VoTE.UNSURE* **then**
      $\{m_1, ..., m_k\} \leftarrow$ *VoTE.core-refine-abstract*(m)
      for $m_i \in \{m_1, ..., m_k\}$ **do**
        MOD-MAPPING-CHECKER(m_i, y)
      end for
    end if
    if $outcome$ == *VoTE.FAIL* **then**
      for $f \in F$ **do**
        $\mathcal{R}_f \leftarrow \mathcal{R}_f\ \cup\ [m_f^{low}, m_f^{high}]$
      end for
      return *VoTE.PASS* ▷ keep analysing mappings
    end if
    return $outcome$
  end function
  return $\mathcal{R}$
end function

Algorithm 1 takes as input a trained tree ensemble model ($\mathcal{T}$), the adversarial robustness property (Ψ), and a dataset (X) consisting of labelled samples with the set of features (F). The dataset (X) used for constructing the counterexample regions is a combination of the training (X_{train}) and validation (X_{harden}) sets. Since VoTE operates in the interval domain, the function CEX-REGIONS-CLASS-WISE starts by defining a sequence of empty intervals (per feature in the dataset) per class. These sequences ($\mathcal{R}_\alpha, \mathcal{R}_\beta$) are used to store the counterexample regions per class (attack and benign respectively). Then X is split according to the class labels and the CR-PR-SAMPLE function is invoked to generate the counterexample regions per sample. Once all samples in the split dataset have been processed, the class-wise counterexample regions will have been generated.

The CR-PR-SAMPLE (Counterexample Region PeR Sample) function generates CEX regions for a single sample and it starts off by defining a variable $\mathcal{R}$, which is a sequence of empty intervals per feature in the sample. This variable is used to collect all the intervals associated with the counterexample regions. When an input sample is passed to the VoTE Core, abstract mappings ($\mathcal{M}$) are generated which are then passed on to the VoTE property checker (*pc*) which is now retrofitted with our modified mapping checker.

The property checker then processes each mapping separately. For each mapping $m \in \mathcal{M}$, when the modified mapping checker encounters a *VoTE.FAIL* response (i.e., a mapping that violates Ψ), it extends the counterexample region set $\mathcal{R}$ with the corresponding mapping interval for each feature in the sample. Once the counterexample intervals have been saved, an "artificial" *VoTE.PASS* is returned to force VoTE to continue analysing the remainder of the abstract mappings. Once all mappings have been analysed, we have successfully generated the counterexample regions associated with the input sample.

C. Adversarial Analyzer

Once the counterexample regions have been generated, they are passed to the adversarial analyzer module that computes the *adversarial score* for an incoming sample. The process begins when an initial classification from the tree ensemble IDS is obtained. If the classification outcome is *attack*, then the distance to the benign counterexample region is assessed and vice-versa. This measured distance constitutes the *adversarial score* and is a measure of how likely a sample is to be adversarial. To compute this distance, we use the weighted l_0 distance from the counterexample regions. If we have a counterexample region defined as $\mathcal{R} = \{r_1, ..., r_n\}$ where r_i is a set of intervals corresponding to a particular feature, then for an incoming sample $\overline{x} = (x_1, \ldots, x_n)$ with corresponding weights $\overline{w} = (w_1, \ldots, w_n)$, the weighted l_0 distance $l_{0,w}$ is defined as $\sum_{i=1}^{n} w_i \cdot \mathbb{1}(x_i \in r_i)$. The weight vector ($\overline{w}$) is the permutation-based feature importance vector from the tree ensemble that a system designer can obtain using scikit-learn[1]. Note that the concept of a weighted distance is used because not all features equally contribute when making predictions using a tree ensemble. By simply setting a threshold (η) on this distance, the adversarial analyzer can assess flows as ad-

[1]https://scikit-learn.org/stable/modules/permutation_importance.html

versarial and non-adversarial. The output from this module are the tuned flow predictions as: {adversarial, non-adversarial}

D. Flow Re-annotator

It is a well known fact that developing the perfect IDS is not possible (with or without ML) [21]. Hence, to prevent automated actions based on incorrect or highly uncertain predictions, detection systems provide decision support to security managers in the form of alerts. However, modern environments can generate thousands of alerts every hour, making manual triaging infeasible [21]. To alleviate this problem, we explore the well-known idea of alert filtering and prioritisation in a machine learning context. The basic idea is that instead of the standard IDS outcomes (attack or benign), we can re-annotate the flows into four outcomes as shown in figure 3. Note that an attack corresponds to the positive class and vice-versa.

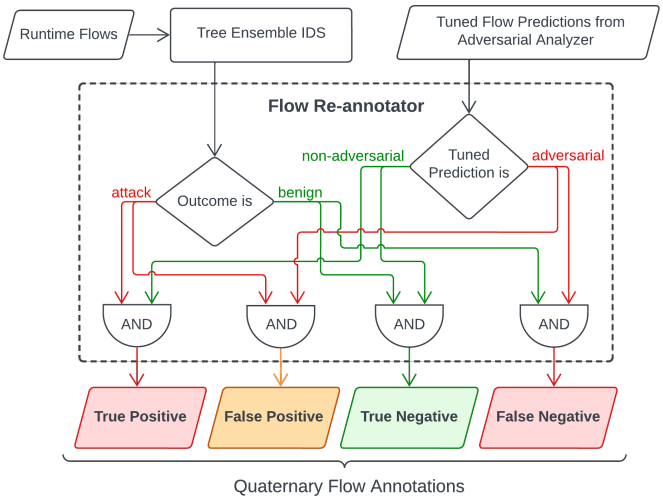


Fig. 3: Flow Re-annotator

The output from the flow re-annotator consists of annotated flows with the colors (alert levels) modelled after traffic signals. In our method, likely true positive attacks or likely false negative (evasion) attacks are given top priority (hence coloured red), while likely false positive attacks (for example, irrelevant alerts closely resembling a DoS attack) are given lower priority (hence coloured orange). No alerts or recommendations are generated for likely true negative (benign) flows (hence coloured green). By definition, alerts are not necessarily attacks as a considerable portion of the alerts correspond to false positives. Since being incessantly notified by false alarms renders the system less usable, our method can be used to filter out redundant alerts.

E. Alert Consolidator

This module resides in the presentation layer of a multi-tier software architecture where user interaction takes place. The role of this module is to map the quaternary flow annotations into a single tuple of (alert level, alert recommendation) for security managers. Table I presents our mapping strategy. Note that these levels & recommendations can be tweaked depending on the application. Several forms of automation are also possible at this stage depending on a security policy.

Flow Re-annotations	maps to	Alert Level	Alert Recommendation
True Negative Flow		0	Benign Flow, Do Nothing!
True Positive Flow		1	Attack, Investigate Now!
False Negative Flow		2	Evasion Attempt, Investigate Now!
False Positive Flow		3	Likely False Alarm, Investigate Later!

TABLE I: Flow Mapping to Alert Level & Recommendation

F. Proposed Workflow

In this subsection, we present the workflow of our proposed method. For ease of understanding, the workflow is split into two phases: pre-deployment and post-deployment.

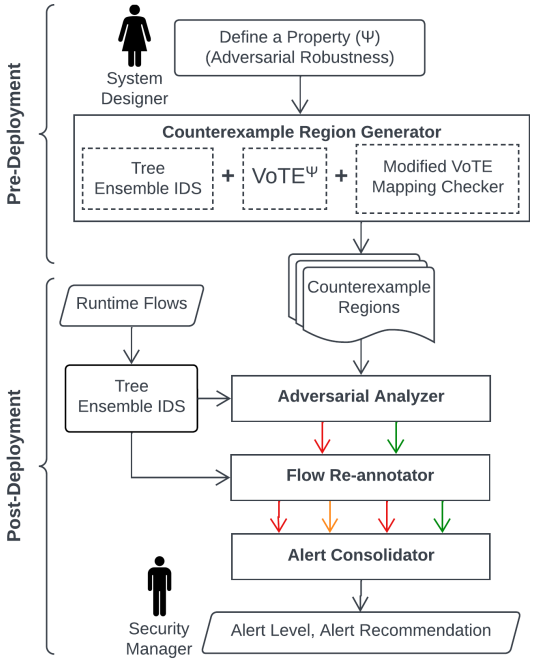


Fig. 4: Proposed Workflow

Figure 4 shows our proposed workflow in action. The key ingredient of our prototype system (*Iceman*), i.e., the counterexample regions are constructed in the pre-deployment phase as follows. The system designer defines a property (Ψ) which is passed to our counterexample region generator module (which is basically VoTE$^{\Psi}$ retrofitted with our modified mapping checker) to generate the class-wise counterexample regions. In the post-deployment phase, these regions are used in conjunction with a distance function by the adversarial analyzer to assess runtime flows as adversarial or non-adversarial. The flow re-annotator then annotates these tuned flow predictions into true positive, false positive, false negative, and true negative annotations. The quaternary annotations from the flow re-annotator are finally amalgamated into a single tuple of (alert

level, alert recommendation) by the alert consolidator module. This tuple normally forms the basis for further automation that security managers can implement in a SOC depending on their organisation's security policies.

VI. Experimental Evaluation

We present the experimental evaluation of *Iceman* in four real-world case studies, and compare our method to the state of the art. Our evaluation addresses three questions:

- Can *Iceman* provide similar or better adversarial detection as compared to the state of the art?
- How is the runtime latency for *Iceman* affected by the added evasion-hardening and flow re-annotating capabilities?
- How accurately does *Iceman* filter and prioritize the most critical IDS alerts?

For the first question, we compare *Iceman* to two approaches (published in top conferences and forums) namely GROOT [17], and OC-Score [9] as mentioned in section II. In our experiments, we generate adversarial examples for the baseline tree ensemble detectors and then evaluate the performance of GROOT, OC-Score, and *Iceman* which are responsible for defending these baseline tree ensemble detectors. Regarding the implementation, GROOT Forests and OC-Score are available as open-source software packages on Github. We make use of the entire reference set for the OC-Score method and use the the uint16 variant as it does not place any tight restrictions on the number of leaf nodes or trees in the ensemble. Pertaining to the baseline tree ensemble detectors, we use the scikit-learn[2] implementation for the random forests and the dmlc[3] implementation for XGBoost Gradient Boosting Machines. Finally, all experiments are conducted on a Windows 11 Machine running Ubuntu 20.04 in Windows Subsystem for Linux mode. The machine comes equipped with an Intel Core i7-10875H CPU and 16 GB of RAM. The code for *Iceman* and the data files for the experimental outcomes in tables and charts in this section are available for future repeatability at https://github.com/val-co/iceman.

A. Datasets

In this work, we use four datasets that focus on the intersection between AI, safety & security, namely, APA-DDoS[4], CIC-IoT-2023 [22], HCRL-Survival-Analysis [23], and CIC-IoV-2024 [24]. The first two case studies deal with real network traffic and IoT data, while the last two case studies deal with real CAN (Controller Area Network) traffic from a 2010 Hyundai Sonata and a 2019 Ford car respectively. We will use these case studies in this order to streamline the explanation of certain aspects of the experimental setup & results.

[2]https://scikit-learn.org/stable/

[3]https://xgboost.readthedocs.io/en/stable/

[4]https://www.kaggle.com/datasets/yashwanthkumbam/apaddos-dataset

B. Experimental Setup

All four datasets were split into X_{train}, X_{harden}, and X_{test} in a 60% : 30% : 10% ratio. X_{train} was used to train the baseline tree ensemble detectors as $model\{depth, trees\}$ for each case study respectively as, $xgboost\{5, 50\}$, $xgboost\{5, 25\}$, $randomforest\{10, 50\}$, and $randomforest\{10, 25\}$. Both X_{train} & X_{harden} were used to construct the counterexample regions in *Iceman*, which took 33, 37, 132, and 155 minutes respectively per case study. Note that this is a one-off cost and happens offline, as frequently as model training. All the datasets were min-max normalized to make perturbations of the same size to different attributes comparable.

VoTE [12] and VERITAS [25] were used to simulate evasion attacks through stealthy manipulations using empirically selected $\mathcal{E}$ attack models of 0.0001, 0.001, 0.00015, and 0.005 respectively per case study. These $\mathcal{E}$ attack models were also used to train GROOT. Finally, since both OC-Score and *Iceman* require setting thresholds to distinguish between adversarial and non-adversarial samples, we perform extensive hyperparameter tuning of this threshold to select the best values that reflect the optimal detection performance for both methods in our experiments. We set these thresholds for each case study as follows: $\{ocscore, iceman\}$ as $\{1, 0.4997\}$, $\{1, 0.024\}$, $\{13, 0.436\}$, and $\{10, 0.18\}$. The evaluations for each case study were performed using approximately 7k, 23k, 20k, and 2k adversarial examples, respectively, along with an equal number of randomly selected samples from their respective X_{test} datasets to prevent any experimental bias towards either the adversarial or the non-adversarial scenario.

C. Evaluation Metrics

In all case studies, we report a host of metrics in terms of Accuracy (Acc.), F1-score (F1-Sc.), True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR), False Negative Rate (FNR), AUC (Area under the Receiver Operating Characteristic Curve), and the Matthew's Correlation Coefficient (MCC). In order to compute the average latency of each approach, we measure the average prediction time for 100 experimental runs. Finally, for *Iceman*, we perform additional evaluations for alert filtering and prioritization.

D. Detection Performance

The task is to correctly classify samples in both adversarial and non-adversarial scenarios in the four case studies (CS) presented in the following order: (1) APA-DDoS, (2) CIC-IoT-2023, (3) HCRL-Survival-Analysis, and (4) CIC-IoV-2024. Prior to presenting the performance comparisons of the defenders, we first evaluate the IDS decision accuracy of the baseline detectors on their respective X_{test} datasets, i.e., in scenarios not subject to adversarial attacks as shown in table II. This serves as a baseline evaluation criterion, assessing the defenders on their ability to maintain the accuracy of the original IDS decisions whilst being subjected to both adversarial and non-adversarial scenarios.

CS	Baseline IDS	Acc.	F1-Sc.	TPR	TNR	FPR	FNR	AUC	MCC
1	$xgboost\{5,50\}$	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00
2	$xgboost\{5,25\}$	1.00	1.00	1.00	0.93	0.07	0.00	0.95	0.91
3	$randomforest\{10,50\}$	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00
4	$randomforest\{10,25\}$	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00

TABLE II: Baseline Detector Performance

CS	Method	Acc.	F1-Sc.	TPR	TNR	FPR	FNR	AUC	MCC
1	GROOT	1.00	1.00	1.00	1.00	0.00	0.00	1.00	**1.00**
	OC-Score	0.50	0.50	1.00	0.33	0.67	0.00	0.5	0.33
	Iceman	1.00	1.00	1.00	1.00	0.00	0.00	1.00	**1.00**
2	GROOT	0.85	0.89	0.83	0.89	0.11	0.17	0.80	0.66
	OC-Score	0.97	0.98	0.97	0.97	0.03	0.03	0.99	0.93
	Iceman	0.98	0.99	0.99	0.96	0.04	0.01	0.98	**0.95**
3	GROOT	0.88	0.86	0.75	1.00	0.00	0.25	0.90	0.78
	OC-Score	0.98	0.98	0.96	1.00	0.00	0.04	1.00	0.96
	Iceman	0.99	0.99	1.00	0.98	0.02	0.00	0.99	**0.98**
4	GROOT	0.74	0.84	0.94	0.11	0.89	0.06	0.58	0.09
	OC-Score	0.56	0.63	0.50	0.74	0.26	0.50	0.38	0.21
	Iceman	1.00	1.00	1.00	1.00	0.00	0.00	1.00	**1.00**

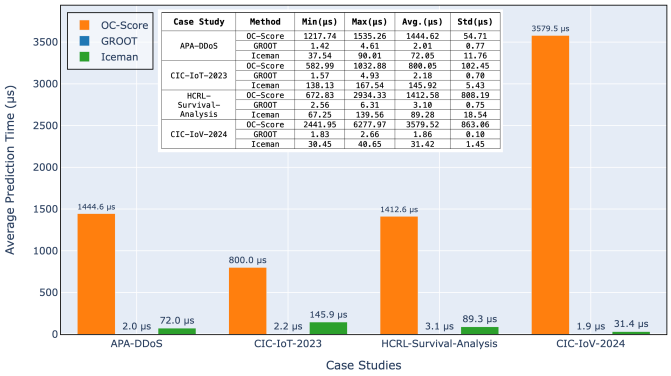
TABLE III: Detection Performance Comparisons (Defenders)

From table III, we see that *Iceman* offers comparable (and slightly better) performance as compared to OC-Score and GROOT. Except for the first case study where *Iceman* and GROOT have the same MCCs, the OC-Score method outperforms GROOT in the subsequent three case studies. However, since *Iceman* produces a slightly better MCC compared to OC-Score in these three case studies, we deem the OC-Score method to be *Iceman's* closest competitor. We use this aspect in the next subsection to show that *Iceman's* maintained level of performance comes with significant timing improvements.

In the last case study, while GROOT has a higher accuracy than OC-Score, the MCC values for GROOT are almost half that of OC-Score. We find this interesting, considering GROOT was trained for that specific $\mathcal{E}$ attack model. Upon further inspection using VERITAS [25], we found that a larger proportion of the attack samples were closer to the decision boundary compared to benign samples. We believe that GROOT's robust splitting criteria expanded the attack regions based on this observation, leading to benign test samples being misclassified as attacks during testing (due to the larger regions). This, in turn, increased the False Positive Rate (FPR) and consequently resulted in a lower Matthews Correlation Coefficient (MCC).

E. Runtime Performance

Figure 5 shows that *Iceman* has a very low prediction latency compared to OC-Score. We believe that the iteration over the entire reference set for the OC-Score method is the likely explanation. Our method is relatively faster (5-115x, by comparing the average prediction times) than OC-Score, as it only requires simple distance calculations between a vector and a region. GROOT is consistently fast as it only requires executing a tree ensemble. However, this comes at the expense of slightly lower performance as shown in table III.



Case Study	Method	Min(µs)	Max(µs)	Avg.(µs)	Std(µs)
APA-DDoS	OC-Score	1217.74	1535.26	1444.62	54.71
	GROOT	1.42	4.61	2.01	0.77
	Iceman	37.54	90.01	72.05	11.76
CIC-IoT-2023	OC-Score	582.99	1032.88	800.05	102.45
	GROOT	1.57	4.93	2.18	0.70
	Iceman	138.13	167.54	145.92	5.43
HCRL-Survival-Analysis	OC-Score	672.83	2934.33	1412.58	808.19
	GROOT	2.56	6.31	3.10	0.75
	Iceman	67.25	139.56	89.28	18.54
CIC-IoV-2024	OC-Score	2441.95	6277.97	3579.52	863.06
	GROOT	1.83	2.66	1.86	0.10
	Iceman	30.45	40.65	31.42	1.45

Fig. 5: Runtime Performance Comparisons

F. Alert Management Performance

The alert filtering process is successful when *Iceman* can correctly filter the alerts into four classes (TPA - true positive alerts, TNA - true negative alerts, FPA - false positive alerts, FNA - false negative alerts). The alert prioritization process is successful when *Iceman* can correctly assign a suitable alert priority level (0, 1, 2, or 3) to incoming samples.

Table IV highlights *Iceman's* alert filtering & prioritization accuracy per case study, along with some statistics on the amount of reduced false alarms which was calculated using the *recall* metric for the FPA class. The results in table IV show that *Iceman* can correctly filter and prioritize IDS alerts with an accuracy of more than 98%. Finally, *Iceman* was also capable of significantly reducing the amount of false alarms.

CS	Alert Prioritization Accuracy	Alert Filtering Accuracy	False Alarms (before *Iceman*)	False Alarms (after *Iceman*)
1	1.00	1.00	7552	0
2	0.98	0.98	11520	0
3	0.99	0.99	10800	0
4	1.00	1.00	0	0

TABLE IV: Alert Management Performance

VII. Conclusion

This paper explores how to detect evasion attacks in tree-ensemble-based network intrusion detection systems. Our approach works with any tree ensemble implementation as long as the IDS can be hardened by making calls to the open-source VoTE. Moreover, our approach does not require training an additional model. The method re-annotates an alert for a sample if it lies within or close to a counterexample region, assuming it as likely to be adversarial. Empirically, *Iceman* displays good detection performance in both adversarial and non-adversarial scenarios with a very low prediction latency compared to several state of the art methods. Additionally, *Iceman* is capable of correctly filtering & prioritizing the most urgent IDS alerts. We find that the counterexample region-based analysis of evasion detection seems plausible and applicable to other IDS methods as long as tools for systematically generating them exist.

The preliminary analysis in this paper shows that the strategy of using counterexample regions to detect adversarial attacks against tree-ensemble-based IDSs can not only help improve security, but can also have benefits for safety. This is because successful security breaches in safety-critical systems like autonomous vehicles can lead to safety hazards including unpredictable behavior in the vehicle, sudden control system failures, unexpected sensor readings, or even unintended acceleration or braking. This makes *Iceman* a step in the right direction towards safe and secure AI.

Regarding limitations, we acknowledge that crafting the counterexample regions in *Iceman* can be time-consuming due to the exhaustive multi-dimensional search space exploration. However, this issue can be mitigated through the use of parallelization techniques (not used in our experiments) or by leveraging GPUs for faster processing, and we leave this for future works. In addition, extending our method to incorporate semantic-aware perturbations while crafting the adversarial defence is an idea worth exploring. Finally, we remark that our method allows for the generation of counterexample *regions* as opposed to individual counterexamples which could lead to future lines of research in terms of new scalable counterexample-guided inductive synthesis implementations for training robust tree ensembles.

ACKNOWLEDGEMENTS

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, 2015.

[2] M. Coccia, "Deep learning technology for improving cancer care in society: New directions in cancer imaging driven by artificial intelligence," *Technology in Society*, vol. 60, 2020.

[3] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman, "Sok: Security and privacy in machine learning," in *IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2018.

[4] G. Apruzzese, M. Andreolini, M. Colajanni, and M. Marchetti, "Hardening random forest cyber detectors against adversarial attacks," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, no. 4, 2020.

[5] T. Zoppi, A. Ceccarelli, T. Puccetti, and A. Bondavalli, "Which algorithm can detect unknown attacks? comparison of supervised, unsupervised and meta-learning algorithms for intrusion detection," *Computers & Security*, vol. 127, 2023.

[6] G. Apruzzese, M. Colajanni, L. Ferretti, and M. Marchetti, "Addressing adversarial attacks against security systems based on machine learning," in *11th international conference on cyber conflict (CyCon)*, vol. 900. IEEE, 2019.

[7] M. Wagner, M. Borg, and P. Runeson, "Navigating the upcoming European Union AI act," *IEEE Software*, vol. 41, no. 1, 2023.

[8] M. Catillo, A. Pecchia, and U. Villano, "Machine learning on public intrusion datasets: Academic hype or concrete advances in NIDS?" in *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 2023.

[9] L. Devos, L. Perini, W. Meert, and J. Davis, "Detecting evasion attacks in deployed tree ensembles," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2023.

[10] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *network and distributed systems security symposium*, 2019.

[11] V. O. Colaco and S. Nadjm-Tehrani, "Formal verification of tree ensembles against real-world composite geometric perturbations," in *Workshop on Artificial Intelligence Safety 2023 (SafeAI 2023) co-located with the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI)*. CEUR-WS, 2023.

[12] J. Törnblom and S. Nadjm-Tehrani, "An abstraction-refinement approach to formal verification of tree ensembles," in *Computer Safety, Reliability, and Security: SAFECOMP Workshops, WAISE, Proceedings 38*. Springer, 2019.

[13] K. He, D. D. Kim, and M. R. Asghar, "Adversarial machine learning for network intrusion detection systems: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, 2023.

[14] I. Zenden, H. Wang, A. Iacovazzi, A. Vahidi, R. Blom, and S. Raza, "On the resilience of machine learning-based ids for automotive networks," in *IEEE Vehicular Networking Conference (VNC)*. IEEE, 2023.

[15] J. Vitorino, I. Praça, and E. Maia, "Towards adversarial realism and robust learning for IOT intrusion detection and classification," *Annals of Telecommunications*, vol. 78, no. 7, 2023.

[16] M. Sadeghi and E. G. Larsson, "Physical adversarial attacks against end-to-end autoencoder communication systems," *IEEE Communications Letters*, vol. 23, no. 5, 2019.

[17] D. Vos and S. Verwer, "Efficient training of robust decision trees against adversarial examples," in *International Conference on Machine Learning*. PMLR, 2021.

[18] Y. Chen, S. Wang, Y. Qin, X. Liao, S. Jana, and D. Wagner, "Learning security classifiers with verified global robustness properties," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[19] B. Biggio, I. Corona, B. Nelson, B. I. Rubinstein, D. Maiorca, G. Fumera, G. Giacinto, and F. Roli, "Security evaluation of support vector machines in adversarial environments," *Support vector machines applications*, 2014.

[20] G. Apruzzese, M. Andreolini, L. Ferretti, M. Marchetti, and M. Colajanni, "Modeling realistic adversarial attacks against network intrusion detection systems," *Digital Threats: Research and Practice (DTRAP)*, vol. 3, no. 3, 2022.

[21] G. Apruzzese, P. Laskov, E. Montes de Oca, W. Mallouli, L. Brdalo Rapa, A. V. Grammatopoulos, and F. Di Franco, "The role of machine learning in cybersecurity," *Digital Threats: Research and Practice*, vol. 4, no. 1, 2023.

[22] E. C. P. Neto, S. Dadkhah, R. Ferreira, A. Zohourian, R. Lu, and A. A. Ghorbani, "CICIoT2023: A real-time dataset and benchmark for large-scale attacks in IoT environments," *Sensors*, vol. 23, no. 13, 2023.

[23] M. L. Han, B. I. Kwak, and H. K. Kim, "Anomaly intrusion detection method for vehicular networks based on survival analysis," *Vehicular communications*, vol. 14, 2018.

[24] E. C. P. Neto, H. Taslimasa, S. Dadkhah, S. Iqbal, P. Xiong, T. Rahman, and A. A. Ghorbani, "CICIoV2024: Advancing realistic IDS approaches against DoS and spoofing attack in IoV CAN bus," *Internet of Things*, vol. 26, 2024.

[25] L. Devos, W. Meert, and J. Davis, "Versatile verification of tree ensembles," in *International Conference on Machine Learning*. PMLR, 2021.