Tree-Ensemble Repair Through Constrained Optimization

Lorenzo Cascioli*
KU Leuven
Leuven.AI
Leuven, Belgium
lorenzo.cascioli@kuleuven.be

Chaahat Jain*
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
jain@cs.uni-saarland.de

Marcel Steinmetz
LAAS-CNRS
Toulouse, France
marcel.steinmetz@laas.fr

Jörg Hoffmann
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
hoffmann@cs.uni-saarland.de

Jesse Davis
KU Leuven
Leuven.AI
Leuven, Belgium
jesse.davis@kuleuven.be

Abstract

It is widely recognized that in many applications, strong predictive performance is not enough to warrant use of a learned model. Additionally, it is required to ascertain whether the model conforms to certain properties such as fairness, robustness or safety. Unfortunately, most learning approaches are not guaranteed to produce a model that adheres to these properties. Model repair addresses this through a post-hoc debugging loop. Approaches such as verification are used to check if the desired property holds for a model; if not they return a counterexample which is used to modify or repair the ML model. We propose a novel repair method for tree ensembles. Our approach is based on a MILP encoding where the objective is to modify the leaf values in the tree ensemble such that the counterexamples are fixed while at the same time preserving predictive performance. The use of constrained optimization over the model parameters is compelling as it is guaranteed to fix a given set of counterexamples. We implement and evaluate our method in two use-case scenarios, classification requiring fairness and action policies in MDPs requiring safety. Results show that our MILP repair of tree ensembles is feasible computationally, and is often successful in improving the desired property while not degrading performance.

1 Introduction

Ensembles of decision trees remain a popular and widely-used model class due to their ability to achieve strong predictive performance on a variety of tasks [13] such as finance [28], healthcare [47] and sports [29]. Typical algorithms for learning tree ensembles, such as random forests [15] and

*These authors contributed equally.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

variants of gradient boosting [41, 22], are guided by objectives that encourage maximizing predictive performance. However, in many real-world application scenarios, solely achieving strong predictive performance is insufficient: often it is also necessary to guarantee that the model satisfies user-defined properties such as fairness [38, 18], robustness [50, 62] or safety [81, 85, 56, 37, 7]. Oftentimes violations of these constraints can have adverse consequences.

Verification approaches allow formally checking whether a previously learned model adheres to specific properties. That is, they either prove that no violations exist for any possible input or return concrete counterexamples (i.e., instances that violate the constraint). Alas, most learning approaches produce models that violate these properties on some inputs. Moreover, verification only identifies problems: it does not directly provide a way to yield a model that exhibits the desired properties. This problem has been tackled in two ways in the literature. On the one hand, it is possible to try to design learning algorithms that are either biased towards or guaranteed to find models that adhere to the provided constraints [61, 3, 92, 74, 87]. Oftentimes, this also comes at the cost of reduced predictive performance. On the other hand, it is possible to develop repair methods that given counterexamples modify a learned model such that it displays the desired behavior for the provided counterexamples. In this way, verification and repair operate as a model debugging loop: verification (or other approaches) identify counterexamples and the repair method fixes them.

We focus on the line of work related to model repair. Recent works in this field have primarily focused on neural networks and can be categorized into two approaches. One treats the problem as a constraint-based repair and uses constraint solving to apply small modifications to the network parameters [49, 86, 25]. The other relies on ML-based methods to guide the learning process through modified loss functions [11], to adjust the training data [83], or to find the most influential neurons through localization techniques and adjust their outputs [36, 79, 94]. There is less work on tree ensembles with current work allowing modifications to predictions that conflict with a user's domain knowledge for binary classification problems [6].

In this paper we propose a novel, generic approach for repairing multiclass tree ensembles. Model repair requires altering a model such that it makes a different prediction for specific instances. While there are multiple ways to alter a tree ensemble, one approach is to modify the leaf values such that the model no longer makes the incorrect prediction for the given counterexamples. In addition to fixing violations, it is also important that the modifications do not substantially degrade the model's predictive performance. To design an approach that meets these two objectives, we formulate a Mixed Integer Linear Programming (MILP) problem. Given a set of counterexamples, we define an encoding that requires fixing the counterexamples by changing the leaf values subject to the goal that the changes should be (a) minimal in magnitude while (b) not degrading performance too much. Our MILP encoding exploits the fact that only the leaves in the *output configuration* (i.e., the set of leaves–one per tree–that are reached by an example) [34] of a counterexample affect its prediction. Consequently, only these leaf values need to be modified to fix the violations, which greatly simplifies the MILP encoding. Finally, because it is possible that the repair step introduces new violations, we employ an iterative procedure that (1) identifies counterexamples (e.g., using verification) and (2) repairs the violations. All previously found counterexamples are retained as hard constraints in subsequent repair steps, ensuring that earlier repairs are preserved. Empirically, we demonstrate the ability of our approach to repair tree ensembles for two types of properties: fairness in predictive modeling and safety in action policies for Markov decision processes (MDP). For both properties, our MILP-based repair approach succeeds in improving property satisfaction without paying a price in performance, while property-aware training methods are worse in both respects.

2 Tree Ensembles

We assume a d-dimensional input space $\mathcal{X}$ and an output space $\mathcal{Y} = \{0, 1, \ldots, Y-1\}$, with Y the number of classes. A decision tree is a function $T : \mathcal{X} \to \mathbb{R}^Y$. We assume binary trees where each internal node has (1) a test of the form $x_f < \tau$ with x_f the fth feature and τ a real-valued threshold, and (2) pointers to a left child and a right child. Each leaf node is annotated with a vector of values $\overrightarrow{\nu_l} \in \mathbb{R}^Y$. Given $x \in \mathcal{X}$, a tree is evaluated by starting at its root and evaluating the test at each visited internal node: if the condition holds, x moves to the left child, otherwise it moves to the right child. The process recurses until a leaf node l is reached. The output of the tree is then $T(x) = \overrightarrow{\nu_l}$.

A tree ensemble $\boldsymbol{T} = \{T_1, T_2, \ldots, T_M\}$ is a set of decision trees. Predictions are made by combining predictions made by the individual trees. We focus on ensembles learned using gradient boosting (GBDT) [41, 22, 63] and random forests (RF) [15], where $\boldsymbol{T}(x) = \sum_{m=1}^{M} T_m(x)$. We indicate the output for a specific class y as $\boldsymbol{T}_y(x)$. The raw sum can be converted into class probabilities (e.g., by applying a softmax transform for GBDTs or dividing by M for RFs). If a hard prediction is required, this is returned by $\boldsymbol{T}_{\text{label}}(x) = \arg\max_y \boldsymbol{T}_y(x)$. Abusing notation, we lift $\boldsymbol{T}_{\text{label}}$ to tuples of inputs, applying it componentwise. We do the same for other functions on $\mathcal{X}$.

An output configuration (OC) [34] is a tuple $oc = (l^{(1)}, \ldots, l^{(M)})$ where (1) there is exactly one leaf $l^{(m)}$ from each tree T_m in $\boldsymbol{T}$ and (2) there exists an $x \in \mathcal{X}$ that can simultaneously reach all those leaves.[1] An example x's OC fully determines $\boldsymbol{T}$'s prediction for x: $\boldsymbol{T}(x) = \sum_{l^{(m)} \in OC(x)} \overrightarrow{\nu_{l^{(m)}}}$. Therefore, two examples sharing the same OC receive the same prediction.

3 MILP-based Repair

Tree ensemble repair is concerned with automatically modifying a given tree ensemble so that it satisfies a user-defined property on given (counterexample) inputs. Our approach can handle arbitrary properties that map input tuples $\mathbf{x}$ and their corresponding output labels $\boldsymbol{T}_{\text{label}}(\mathbf{x})$ to 1 if the property is satisfied and to 0 otherwise. That is, our repair method applies to properties whose violation is demonstrated by a finite tuple of inputs $\mathbf{x} \in \mathcal{X}^k$, where k denotes the number of inputs jointly considered by the property.[2] The only restriction we make is that the property is satisfiable:

Definition 1 (Property)**.** *A property is a function $\mathcal{P} : \mathcal{X}^k \times \mathcal{Y}^k \to \{0, 1\}$ for which there exists a classifier $f : \mathcal{X} \to \mathcal{Y}$ such that, for all $\mathbf{x} \in \mathcal{X}^k$, $\mathcal{P}(\mathbf{x}, f(\mathbf{x})) = 1$.*

Many properties $\mathcal{P}$ of this kind have been considered in the literature, e.g., robustness requiring two close-by inputs to have the same label [50, 62]; fairness requiring two inputs that differ only in a protected attribute (e.g., gender or race) to have the same label [38, 18]; safety forbidding particular input/label pairs (action labels unsafe on these inputs, e.g., accelerating sharply on an icy road losing control of a car) [85, 7, 91]. Fairness and robustness are instances of properties with $k = 2$, as they consider predictions on pairs of inputs; safety constrains the prediction on a single input, and hence is an instance of a property with $k = 1$.

An ensemble $\boldsymbol{T}$ *satisfies* $\mathcal{P}$ iff $\mathcal{P}(\mathbf{x}, \boldsymbol{T}_{\text{label}}(\mathbf{x})) = 1$ for all $\mathbf{x} \in \mathcal{X}^k$. Most learned models do not always adhere to the user-defined property $\mathcal{P}$, i.e., there are counterexamples that violate the property:

Definition 2 (Counterexample)**.** *Given a tree ensemble $\boldsymbol{T}$ and a property $\mathcal{P} : \mathcal{X}^k \times \mathcal{Y}^k \to \{0, 1\}$, a* counterexample *is a tuple of inputs $\mathbf{e} \in \mathcal{X}^k$ where $\mathcal{P}(\mathbf{e}, \boldsymbol{T}_{label}(\mathbf{e})) = 0$.*

Depending on the considered property, there are a variety of ways to find counterexamples. A prominent and common approach is to use verification [7, 8, 33, 90, 91] with other methods including fuzzing [39, 59], symbolic executions [10] and adversarial attacks [19, 68, 82, 21].

Our goal is to repair a previously learned model with respect to a given set of counterexamples:

Given: tree ensemble $\boldsymbol{T}$, the training data $\mathcal{D}$, and a set of counterexamples $\mathbf{E}$ to the property $\mathcal{P}$

Do: produce a modified ensemble $\boldsymbol{T}'$ such that $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{\text{label}}(\mathbf{e})) = 1$ for all $\mathbf{e} \in \mathbf{E}$.

We address the tree ensemble repair problem by formulating a mixed integer linear program (MILP) that (tries to) perform the repair by only changing the leaf values. This approach confers several advantages. First, MILP allows using hard constraints to encode that the model must make a different prediction on the counterexamples, thereby guaranteeing that the current undesirable behavior is avoided. Second, it allows us to also encode that we prefer modifications that minimally affect predictive performance (i.e., the new model should exhibit similar predictive performance to the old one). Third, we can simplify the repair problem by exploiting the fact that because an example's OC fully determines its prediction, only leaves belonging to the OC of counterexamples may need to be changed to enforce the property constraints. Thus, we can greatly simplify the MILP encoding by

[1] Some leaf combinations are not possible because the root-to-leaf paths in two trees impose conflicting constraints on the feature values.

[2] A related concept is that of hyperproperties [24], which relate system behavior across multiple inputs.

only considering $\mathcal{L}_{\mathbf{E}} = \bigcup_{\mathbf{e}\in\mathbf{E}} OC(\mathbf{e})$. Finally, it allows us to prefer repairs that do not increase the model's complexity (i.e., size) when possible. Only if repair via leaf refinement is not possible do we alter the ensemble structure by adding new trees that allow fixing the found violations. Next, we describe (i) our hard constraints that encode counterexample repair, (ii) our optimization objective, (iii) how we restore feasibility if the MILP is infeasible, and (iv) our overall repair loop.

3.1 Hard Constraints for MILP Repair

Our repair constraints force the MILP to ensure that the ensemble makes different predictions for the given set of counterexamples, thereby providing a guaranteed repair. The key observation underlying our encoding is that we never need to encode the property $\mathcal{P}$ itself: we merely need to formalize and encode into MILP how the model should behave *on the considered fixed set of counterexamples* $\mathbf{E}$. This can be specified as a propositional formula:

Definition 3 (Repair Formula)**.** *Let $\mathbf{e} \in \mathcal{X}^k$ be a counterexample to property $\mathcal{P}$ for ensemble $\boldsymbol{T}$. A* repair formula $\phi_{\mathcal{P}}(\mathbf{e})$ *constrains how a repaired ensemble $\boldsymbol{T}'$ must label $\mathbf{e}^0, \ldots, \mathbf{e}^{k-1}$ such that $\mathcal{P}$ is satisfied. $\phi_{\mathcal{P}}(\mathbf{e})$ is composed of:*

- *label-assignment atoms: $\boldsymbol{T}'_{label}(\mathbf{e}^i) = y$, with $i \in \{0, \ldots, k-1\}$ and $y \in \mathcal{Y}$,*
- *label-agreement atoms: $\boldsymbol{T}'_{label}(\mathbf{e}^i) = \boldsymbol{T}'_{label}(\mathbf{e}^j)$, with $i \neq j$,*
- *Boolean combinations of these constructed inductively using $\neg$, $\wedge$, $\vee$.*

A repair step requires finding an ensemble $\boldsymbol{T}'$ such that $\boldsymbol{T}' \models \phi_{\mathcal{P}}(\mathbf{e})$. $\phi_{\mathcal{P}}(\mathbf{e})$ always exists:

Theorem 1 (Universality of Repair Formulas)**.** *Let $\mathcal{P} : \mathcal{X}^k \times \mathcal{Y}^k \to \{0,1\}$ be a property and $\mathbf{e} \in \mathcal{X}^k$ a counterexample to $\mathcal{P}$ for ensemble $\boldsymbol{T}$. Then there exists a repair formula $\phi_{\mathcal{P}}(\mathbf{e})$ such that, for every tree ensemble $\boldsymbol{T}'$:*

$$\boldsymbol{T}' \models \phi_{\mathcal{P}}(\mathbf{e}) \quad \iff \quad \mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{label}(\mathbf{e})) = 1.$$

This is simply because, on a fixed input, any property can be viewed as a truth-value assignment over the finite domain $\mathcal{Y}^k$, and that truth-value assignment can be represented by a repair formula. A detailed proof is given in Appendix A.1. The label-agreement atoms are not needed for this; they merely serve as syntactic sugar that permits expressing typical repair requirements more compactly.

Next, we show how to encode the repair formula $\phi_{\mathcal{P}}(\mathbf{e})$ for a given counterexample $\mathbf{e}$ into MILP. Our encoding relies on binary indicator variables $z_{i,y} \in \{0,1\}$ for each input $\mathbf{e}^i \in \mathbf{e}$ and class y, whose value is 1 *iff* y outscores every other class on $\mathbf{e}^i$:

$$z_{i,y} = 1 \iff \boldsymbol{T}'_y(\mathbf{e}^i) > \boldsymbol{T}'_{y'}(\mathbf{e}^i) \quad \forall y' \neq y \in \mathcal{Y}. \tag{1}$$

Given this, encoding the MILP constraints follows directly from the inductive structure of $\phi_{\mathcal{P}}(\mathbf{e})$. Each atom translates directly into linear constraints over the indicators: a label-assignment atom $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = y$ becomes the constraint $z_{i,y} = 1$, and a label-agreement atom $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = \boldsymbol{T}'_{\text{label}}(\mathbf{e}^j)$ becomes the conjunction $\bigwedge_{y\in\mathcal{Y}} z_{i,y} = z_{j,y}$. The Boolean structure of ϕ is handled by the standard MILP encoding of propositional logic, introducing one auxiliary binary variable per subformula [89]. To encode a set $\mathbf{E}$ of counterexamples, we simply conjoin the encodings of their formulae: $\phi_{\mathcal{P}}(\mathbf{E}) = \bigwedge_{\mathbf{e}\in\mathbf{E}} \phi_{\mathcal{P}}(\mathbf{e})$. We denote this MILP encoding by $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ and fully develop it in Appendix A.2. Every feasible solution satisfies the property $\mathcal{P}$ on the counterexample set $\mathbf{E}$ (see Appendix A.3):

Theorem 2 (Soundness of MILP Repair Constraints)**.** *Let $\mathcal{P}$ be a property, $\boldsymbol{T}$ an ensemble, and $\mathbf{E}$ a set of counterexamples. Let $\boldsymbol{T}'$ be an ensemble with the same trees as $\boldsymbol{T}$, but with leaf values replaced with a solution to $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$. Then, for all $\mathbf{e} \in \mathbf{E}$, $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{label}(\mathbf{e})) = 1$. If $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ is infeasible, then there is no $\boldsymbol{T}'$ over the same trees as $\boldsymbol{T}$ such that, for all $\mathbf{e} \in \mathbf{E}$, $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{label}(\mathbf{e})) = 1$.*

We now show how to instantiate our approach for two concrete properties: fairness and safety.

Fairness Several notions of fairness exist in the literature [55, 38]. We focus on one particular definition of individual fairness, which is called lack of causal discrimination [18, 44]:

Definition 4 (Fairness)**.** *Let p be a designated* protected attribute*. An ensemble $\boldsymbol{T}$ is* fair *iff*

$$\forall x, x' \in \mathcal{X} : \quad \forall i \neq p,\ x_i = x'_i \implies \boldsymbol{T}_{label}(x) = \boldsymbol{T}_{label}(x')$$

That is, any two inputs differing only in the protected attribute must receive the same label. Formulated as a property $\mathcal{P} : \mathcal{X}^2 \times \mathcal{Y}^2 \to \{0,1\}$, $\mathcal{P}((x,x'),(\boldsymbol{T}_{label}(x),\boldsymbol{T}_{label}(x'))) = 1$ *if either* $x_i \neq x'_i$ *for some attribute* $i \neq p$ *or* $\boldsymbol{T}_{label}(x) = \boldsymbol{T}_{label}(x')$.

A counterexample $\mathbf{e}$ here is a pair $(\mathbf{e}^0, \mathbf{e}^1)$ of inputs differing only in p on which $\boldsymbol{T}$ violates fairness, i.e., $\boldsymbol{T}_{\text{label}}(\mathbf{e}^0) \neq \boldsymbol{T}_{\text{label}}(\mathbf{e}^1)$. Repairing it requires the two predictions to agree, so $\phi_{\mathcal{P}}(\mathbf{e})$ is the label-agreement atom $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^0) = \boldsymbol{T}'_{\text{label}}(\mathbf{e}^1)$.

Safety We address safety in action policies (short: policies) for discrete Markov decision processes (MDP), as considered in, e.g., reinforcement learning (RL) [70] and AI planning [46]. The policy is represented by a tree ensemble [58]: environment states correspond to ensemble inputs $x \in \mathcal{X}$, and the action chosen in state x is the class predicted by the ensemble, $\boldsymbol{T}_{\text{label}}(x) \in \mathcal{Y}$. The policy chooses the next action in environment states, with the objective of reaching a goal while being safe. Specifically, a policy is **safe in a state** x if, starting from x, the policy never enters a failure state. The **safe region** is the set of states x for which there exists a policy safe in x. An **unsafe action** is one that risks leaving the safe region, i.e., action y is **unsafe in state** x if x is in the safe region but executing y from x can result in a state outside the safe region.[3]

Definition 5 (Safety)**.** *Let* $\mathcal{X}^S \subseteq \mathcal{X}$ *be the safe region, and* $\mathcal{Y}^U(x)$ *be the set of actions unsafe for* $x \in \mathcal{X}^S$*. An ensemble* $\boldsymbol{T}$ *is* ***safe*** *iff*

$$\forall x \in \mathcal{X}^S, \forall y \in \mathcal{Y}^U(x): \quad \boldsymbol{T}_{label}(x) \neq y.$$

That is, the policy never selects an unsafe action for a state in the safe region. Formulated as a property $\mathcal{P} : \mathcal{X} \times \mathcal{Y} \to \{0,1\}$, $\mathcal{P}(x, \boldsymbol{T}_{label}(x)) = 1$ *if either* $x \notin \mathcal{X}^S$ *or* $\boldsymbol{T}_{label}(x) \notin \mathcal{Y}^U(x)$.

Here, a counterexample $\mathbf{e}$ is an input state known to be in the safe region, $\mathbf{e} \in \mathcal{X}^S$, whose class prediction $\boldsymbol{T}_{\text{label}}(\mathbf{e})$ is unsafe in $\mathbf{e}$. Its repair formula $\phi_{\mathcal{P}}(\mathbf{e})$ is: $\bigwedge_{y \in \mathcal{Y}^U(\mathbf{e})} \neg(\boldsymbol{T}'_{\text{label}}(\mathbf{e}) = y)$.[4]

3.2 MILP Objective: Modify Leaf Values to Minimize Accuracy Loss

Beyond ensuring the property will be enforced on the given set of counterexamples, the modifications should minimize the accuracy loss on training examples and be small in magnitude. We formulate the following objective, which has two components reflecting these two rationales:

$$\min_{\overrightarrow{\nu}'_{\mathbf{E}}} \quad \underbrace{\frac{1}{W} \cdot \sum_{(x,y)\in\mathcal{D}'} w_y \, s_x}_{\text{accuracy preservation}} + \lambda \cdot \underbrace{\left(\frac{1}{|\overrightarrow{\nu}_{\mathbf{E}}|} \cdot \|\overrightarrow{\nu}'_{\mathbf{E}} - \overrightarrow{\nu}_{\mathbf{E}}\|_1 \right)}_{\text{small modifications}} \tag{2}$$

$$\begin{aligned} \text{s.t.} \quad & \forall (x,y) \in \mathcal{D}' : \\ & s_x \geq 0 \\ & s_x \geq \max_{y' \in \mathcal{Y}\setminus\{y\}} \boldsymbol{T}'_{y'}(x) - \boldsymbol{T}'_y(x). \end{aligned}$$

Here, primed symbols denote the tree ensemble with the modified leaf values we are finding using MILP, and $\overrightarrow{\nu_{\mathbf{E}}}$ is the vector concatenating all $\overrightarrow{\nu_l} \in \mathcal{L}_{\mathbf{E}}$. By $\mathcal{D}'$ we denote the set of correctly classified training examples that are not counterexamples,[5] $\mathcal{D}' = \{(x,y) \mid \boldsymbol{T}_{\text{label}}(x) = y, (x,y) \in \mathcal{D}, \forall \mathbf{e} \in \mathbf{E} : x \notin \mathbf{e}\}$. Intuitively, s_x are slack variables capturing accuracy loss, $\|\overrightarrow{\nu}'_E - \overrightarrow{\nu}_E\|_1$ enforces L1 regularization on the new leaf values to ensure they do not deviate too much from the original ones, and λ represents a tradeoff in the importance between the two components of the objective. Observe that for each $x \in \mathcal{D}'$ the non-negative continuous slack variable s_x is pushed to 0 if x is correctly classified when using the new leaf values. Indeed, let $\boldsymbol{T}'$ be the tree ensemble constructed from a solution to Equation (2): if the correct class y is still chosen with the new leaf values, $\max_{y' \in \mathcal{Y}\setminus\{y\}} \boldsymbol{T}'_{y'}(x) - \boldsymbol{T}'_y(x)$ is negative and the minimization pushes s_x to the lower bound,

[3] This captures worst-case safety, which requires being safe under all possible circumstances. See Appendix E.1.

[4] Details regarding the computation of $\mathcal{Y}^U(\mathbf{e})$ are provided in Appendix E (Remark 8).

[5] A correctly classified training example can also be a counterexample. For example, in a fairness setting x is part of a counterexample if x' is similar to x but $x' \notin \mathcal{D}$ and $\boldsymbol{T}_{\text{label}}(x) \neq \boldsymbol{T}_{\text{label}}(x')$.

which is 0. If this is not the case, then $s_x > 0$ and contributes positively to the accuracy loss term. To account for class imbalance, each slack s_x is weighted by $w_y = 1/|\mathcal{D}'_y|$, where $\mathcal{D}'_y \subseteq \mathcal{D}'$ contains only the examples of class y, and we normalize the sum dividing by $W = \sum_{(x,y)\in\mathcal{D}'} w_y$.

3.3 Solving MILP Infeasibility

The MILP is infeasible if fixing two or more found counterexamples requires conflicting leaf values modifications (e.g., the ensemble needs to make different predictions for instances it treats as isomorphic). We fix this by adding additional trees to the ensemble in a way that is guaranteed to eventually make the MILP feasible. To generate the trees, we make use of *irreducible infeasible subsystems (IIS)* [48]. An IIS Φ is a set-minimal subset of constraints that are unsatisfiable in conjunction. MILP solvers allow extracting such a Φ if infeasibility is detected. Crucially, the constraints involving the slack variables s_x cannot appear in Φ since each slack constraint is satisfiable with s_x simply taking an extremely large value, and s_x appears in no other constraint. Hence, every IIS is solely extracted from $\phi_{\mathcal{P}}(\mathbf{E})$.

Given an IIS Φ, we identify the counterexamples $\mathbf{E}_\Phi \subseteq \mathbf{E}$ responsible for the constraints, and learn a single decision tree T_Φ that maps every input in $\mathbf{E}_\Phi$ to a different leaf via a standard learning method [72]. All leaf values in T_Φ are initialized to 0, meaning that a priori the new tree does not change any prediction of the ensemble. The solution to the MILP diverges from this initialization only as much as necessary. We add the tree to the ensemble and invoke the MILP for this extended ensemble. This process is repeated until the MILP becomes feasible, which must happen eventually: by construction of T_Φ, constraints involving only $\mathbf{E}_\Phi$ are solvable for the augmented ensemble, so the same $\mathbf{E}_\Phi$ can never re-occur. As $\mathbf{E}$ is finite there are only finitely many $\mathbf{E}_\Phi$, thus (see Appendix C.1):

Theorem 3 (Feasibility Recovery)**.** *Let $\mathcal{P}$ be a property, $\boldsymbol{T}$ a tree ensemble, and $\mathbf{E} \subseteq \{\mathbf{e} | \mathcal{P}(\mathbf{e}, \boldsymbol{T}_{label}(\mathbf{e})) = 0\}$ a set of counterexamples for which $\text{MILP}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ is infeasible. The above-described tree-generation process will eventually terminate with a tree ensemble $\boldsymbol{T}^f$ for which $\text{MILP}(\boldsymbol{T}^f, \phi_{\mathcal{P}}(\mathbf{E}))$ is feasible.*

3.4 Iterative Repair Loop

The new leaf values returned by the solution to $\text{MILP}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ guarantee that the modified tree ensemble enforces the desired property on the set of previously identified counterexamples $\mathbf{E}$. However, the modified leaf values can change the ensemble's output on a subset of the input space that previously satisfied $\mathcal{P}$, potentially leading to *new* counterexamples. Since our ultimate goal is to satisfy $\mathcal{P}$ over the entire input region, we embed the repair procedure inside an iterative loop (see Algorithm 1 in Appendix B). We initialize $\boldsymbol{T}^0 = \boldsymbol{T}$ and $\mathbf{E} = \emptyset$. At each iteration $i \geq 0$, we (i) verify the current ensemble $\boldsymbol{T}^i$ against $\mathcal{P}$, (ii) collect counterexamples $\mathbf{E}^i$ from the verifier (there must be at least one, or the property holds), (iii) add them to the accumulated counterexample set $\mathbf{E} \leftarrow \mathbf{E} \cup \mathbf{E}^i$, and (iv) obtain $\boldsymbol{T}^{i+1}$ by solving the repair MILP over the full set of counterexamples $\mathbf{E}$. If the MILP is infeasible, we learn new trees to restore feasibility as discussed in the previous section. Since the MILP enforces $\mathcal{P}$ on every counterexample in $\mathbf{E}$, and $\mathbf{E}$ grows monotonically across iterations, we ensure that all previously discovered counterexamples along with the new ones are repaired. The loop terminates when no new counterexamples are found ($\mathbf{E}^i = \emptyset$). Regardless of the property considered, this loop terminates under finiteness assumptions:

Theorem 4 (Termination)**.** *Assuming the input space $\mathcal{X}$ is finite, the iterative repair loop terminates in finitely many iterations.*

Stronger guarantees are possible for specific properties. In particular, for fairness the finiteness assumption can be dropped: the MILP is always feasible hence the tree ensemble structure is preserved across iterations, which suffices to ensure termination. Proofs can be found in Appendix B.1.

4 Experiments

We evaluate the ability of our MILP approach to repair tree ensembles for two properties: fairness in predictive modeling and safety in planning.[6] We address the following main research question:
How does our MILP repair compare to methods that bias the learning procedure?

[6]The code to reproduce all experiments is available at `https://github.com/ML-KULeuven/tree-ensemble-repair`.

We always start from a learned tree ensemble and run the repair loop from Section 3.4 for a maximum of 1000 iterations, setting a time limit of 1h per MILP iteration. We use Gurobi 12.0 [53], which offers the computation of the IIS out of the box. We use a default $\lambda = 10^{-3}$ for the objective regularization term (Equation (2)).

Appendix D further reports three ablations on our MILP encoding, where we study the sensitivity to the λ hyperparameter (D.3) and show that our composite objective beats either of the two terms alone (D.4) and also an alternative objective that bounds accuracy loss through hard constraints (D.5).

4.1 Fairness

Experiments Setup We use ten standard benchmarks from the fairness literature (see Table 2 in Appendix D). Verifying fairness as in Def. 4 is an NP-hard problem [4]: intuitively, it requires considering all pairs of OCs reachable by two inputs that only differ on the protected attribute. However, the number of OCs scales exponentially with ensemble size [69]. Since standard ensemble learners [22, 15] tend to generate large models, these are often impossible to verify with current techniques. Because it is known to be difficult to directly train small and accurate models, we train a large XGBoost [22] model (100 trees, depth 5) and compress it with LOP [31], which reduces model size subject to a bound on performance loss on a held-out validation set. Concretely, we allow a loss in balanced accuracy of at most half a percentage point. We repeat this procedure using 5-fold cross-validation, yielding 50 models in total for which we run our repair loop. In Appendix D.7, we repeat this experiment with random forests [15]. The experiments ran on an Intel i7-12700 CPU with 64 GB of memory.

We generate counterexamples using Veritas [33, 30], a state-of-the-art tree ensemble verifier that checks fairness by exhaustively enumerating all pairs of valid OCs. Any other verifier [18, 32, 69] returning concrete counterexamples violating Def. 4 works for the task. We measure model performance using *balanced accuracy* on the test set. A model is *provably fair* if the verifier finds no counterexample.

Baselines We are not aware of any training or repair methods for tree ensembles that provably guarantee fairness as in Def. 4. Therefore we compare our MILP repair with two fairness-aware training methods. *FATT (Fairness-Aware Tree Training)* [74] trains (ensembles of) decision trees using a genetic algorithm guided by abstract interpretation that balances accuracy and fairness at training time, improving fairness without a formal guarantee. *BuDRO* [88] trains GBDTs using a distributionally robust loss, i.e., at each round it computes a worst-case reweighting of the training data, shifting weight between individuals that are similar according to a fairness metric (e.g., only differing in the protected attribute) which pushes the ensemble to score them alike. We provide details on how we configured each baseline in Appendix D.1.

Results Figure 1 reports fairness (left) and balanced accuracy loss (right) for the LOP-compressed models, the models repaired with our MILP, and the models trained with the fairness-aware baselines. We provide detailed results per dataset in Appendix D.2. The MILP is never infeasible, hence we never need to add trees to the models.

Fairness: Only 24% of the LOP models are fair. In contrast, our loop terminated in all instances with no new violations found, meaning that *all ensembles repaired with our MILP are provably fair*. On the other hand, *fairness-aware training methods often fail to produce fair models*: FATT yields a fair model 68% of the time, and BuDRO on 62% of instances.

Accuracy: On average, *MILP only pays a price of ∼1.5 percentage points* (pp) *in balanced accuracy with respect to the original uncompressed XGBoost model*, of which ∼1.1 pp is the actual cost of enforcing fairness and the remaining ∼0.4 pp is due to model compression. Conversely, *FATT and BuDRO lose on average* ∼9.0 pp *and* ∼2.9 pp *in balanced accuracy with respect to the original XGBoost model*.

Repair time: The MILP repair loop terminates within 10 minutes on 58% of instances, with a median runtime of 19 seconds. On harder instances solving the MILP can take longer; however, the loop always terminates with a verified fair model. Specifically, the loop requires a median of 3 iterations to converge with a maximum of 22 iterations (details in Table 5, Appendix D.2). Most of the loop runtime is spent solving the MILP, as LOP compression enables fast and complete counterexample

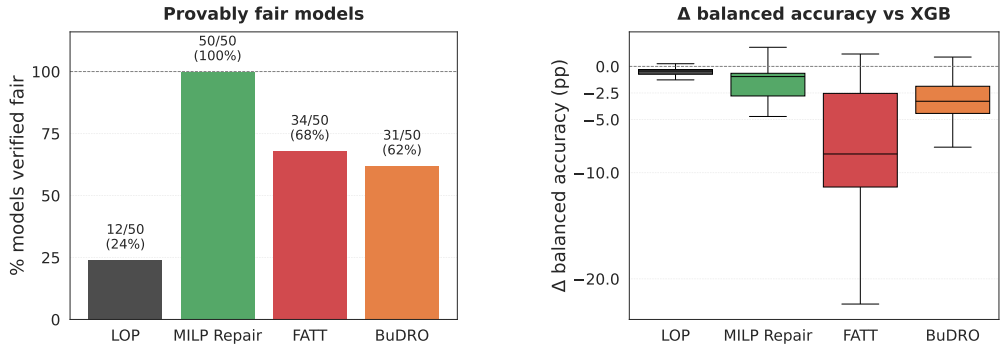

Figure 1: Left: fraction of provably fair models among the models generated with LOP, the MILP-repaired models, and the models trained with the fairness-aware baselines. There are 50 models in total (10 datasets $\times$ 5 folds). Right: balanced accuracy loss with respect to the original XGB model.

generation[7] at a minimal accuracy cost. In Appendix D.2 we further study how MILP runtime correlates with model size and with the number of found counterexamples.

4.2 Safety

Experiments Setup We use an existing benchmark set for policy safety checking [59]. This consists of 38 problem instances where the environment is modeled as a discrete-time Markov decision process (MDP), with a finite action set and an input space defined by state features that are either Boolean, bounded-integer, or real-valued. The MDPs have probabilistic action outcomes as well as initial state uncertainty. The set of possible initial states is specified in terms of a constraint over the state features (and is thus infinite, or exponentially large in the number of state features). We provide details in Appendix E.2. We use the neural policies from Jain et al. [59] as teachers for imitation learning of tree ensemble policies.[8] Specifically, we generate the training data by collecting the states and neural teacher predictions from 1000 teacher policy runs on random initial states, and use it to train XGBoost [22] ensembles with fixed hyperparameter settings of 20 trees and max depth 5. The experiments ran on an AMD EPYC 9654 CPU with 64 GB of memory.

Policy safety verification is an extremely complex computational problem. It is PSPACE-complete for the considered discrete setting [91], and undecidable in the continuous case. Scalability remains a major problem in state-of-the-art verifiers. To not limit our experiments, we generate counterexamples falling back to more efficient, albeit incomplete, fuzzing methods [59]. Specifically, we pre-select a random set $\mathcal{X}_0^R$ of 5000 initial states (the **repair states**), and in each repair loop iteration we generate counterexamples by running the fuzzing algorithm on $\mathcal{X}_0^R$. We then estimate policy performance and safety across all initial states by pre-selecting a different set $\mathcal{X}_0^E$ of 5000 initial **evaluation states** that is disjoint from $\mathcal{X}_0^R$. To measure performance, we sample 100 policy runs from every $s_0 \in \mathcal{X}_0^E$; **%Goal** is the fraction of runs that reach the goal. To measure safety, we exhaustively check for each $s_0 \in \mathcal{X}_0^E$ whether the policy is safe on s_0; **%Safe** is the fraction of $\mathcal{X}_0^E$ for which that is the case. Again, due to scalability limitations of state-of-the-art verifiers, we have to fall back to this empirical %Safe safety metric, which could be 100% even if the repaired policy is not guaranteed to be safe.

Baselines Due to the lack of other tree-ensemble policy repair methods, we compare our approach to two retraining-based approaches. Both learn a new tree ensemble from scratch at each iteration of the repair loop, biasing the learning algorithm away from the counterexamples. The first approach *MH* is a hardening-style approach [50, 62] that augments the training set and corrects the identified counterexamples using the neural teacher's labels. The second approach *FALoss* uses a loss function that penalizes selecting unsafe actions [57, 64] when training a new XGBoost ensemble, and is inspired by safe RL penalty-based approaches [45, 2, 84] and neural network repair [11]. Additionally, we compare to a shielding approach *PT (Penalty Trees)* that explicitly filters out the identified safety

[7] Veritas exhaustively verifies all LOP-compressed models in $< 10\text{m}$, but it runs out of resources (i.e., 3h timeout or 64 GB memory) for 45/50 of the original XGB models.

[8] It has been shown that, in this context, tree ensembles can learn policies on par with neural ones [58].

violations by adding a tree to the ensemble for each counterexample. Details for all baselines are given in Appendix E.3.

Results We exclude benchmarks where the original tree ensemble policy was already safe, hence the repair loop was never invoked, which leaves 31 benchmarks. Figure 2 reports %Safe and %Goal for these 31 original policies, and for the policies repaired with our loop using our MILP repair, the two retraining-based methods, and *Penalty Trees*. Per-benchmark results are provided in Appendix E.4. We only need to add trees to avoid infeasibility in two instances. We further discuss the scalability of our feasibility recovery procedure in Appendix C.2.

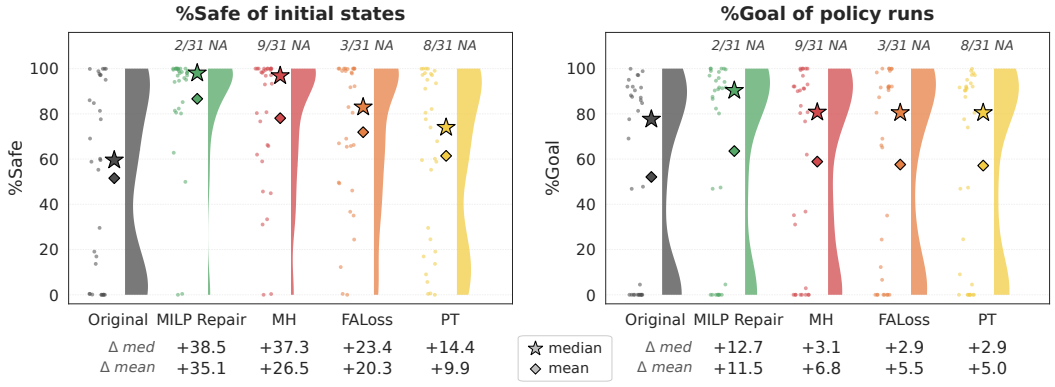

Figure 2: %Safe (left) and %Goal (right) before and after repair on the evaluation states $\mathcal{X}_0^E$, showing per-benchmark values (dots), and the kernel density estimate (violin). ★: median, ♢: mean; Δ reports the change from the original to the repaired model. $k/31$ *NA* above a column: k of 31 benchmarks failed to converge. In these cases, we evaluate the last policy generated from the repair loop.

Safety: As Figure 2 (left) shows, *our MILP repair yields stronger safety improvements than the baselines*. In particular, the mean of %Safe rises by 35.1 pp for MILP, by 26.5 pp for *MH*, by 20.3 pp for *FALoss* and by 9.9 pp for *PT* (for the median, these numbers are 38.5, 37.3, 23.4 and 14.4 pp). Recall that MILP repair is guaranteed to fix all given counterexamples as per Thm. 2: we sometimes get %Safe $< 100\%$ solely because the counterexample generation step is incomplete. Moreover, exact verification [58] proved safety for one of our MILP-repaired policies, and for none of the policies repaired through *MH*, *FALoss*, *PT*; in all other cases, verification either ran out of resources or found some additional counterexamples that fuzzing had missed (details in Table 9, Appendix E.4).

Performance: Improved safety comes at no price in goal-reaching performance. Indeed, mean %Goal improves by 11.5 pp for MILP, 6.8 pp for *MH*, 5.5 pp for *FALoss*, and 5.0 pp for *PT*, while the medians increase by 12.7, 3.1, 2.9, and 2.9 pp respectively. This makes sense as unsafe behavior is one possible cause for not reaching the goal.

Repair time: Our repair loop is typically cheap. Of the 29 benchmarks that complete, the MILP repair loop terminates within 10 minutes on 19, with a median runtime of 3.5 minutes; a further 5 finish within an hour, and the remaining 5 take between 1h and 3h. However, the MILP repair step is typically *not* the bottleneck of the repair loop: its median share of the total loop runtime is 9.2%, with counterexample generation taking up the rest. The loop never exceeds 25 iterations and needs fewer than 10 in 26 instances, with a median of 3 (details in Table 12 and Table 13, Appendix E.4). In the 2 remaining benchmarks, counterexample generation runs out of memory once and the MILP times out once. The baselines fail more frequently than MILP: retraining-based methods fail to fix all counterexamples in 9 (*MH*) and 3 (*FALoss*) instances where the loop is stopped after 1000 iterations. Likewise, *PT* exhausts resources in 8 instances due to model size exploding from all the added trees.

5 Related Work

Most property-agnostic **repair methods** target (deep) neural networks [49, 80, 83, 86, 36, 11, 42, 79, 14]. Closest to ours is Goldberger et al. [49], who use linear programming to modify the weights in the last layer of the network to fix predictions on a set of counterexamples. Beyond addressing

a different model class, we also (1) explicitly minimize the loss in predictive performance and (2) exploit properties of the tree structure to simplify the MILP encoding. The only property-agnostic repair method for tree ensembles is PyXAI [6], whose rectification operation enforces a user-specified output on a given input. This is insufficient for properties such as fairness or safety, which do not specify a target output per counterexample but only require that outputs of similar inputs agree (fairness) or that the output lies in a set of admissible values (safety). Moreover, PyXAI does not support multiclass ensembles. Repair algorithms tailored to fairness also exist, but for trees they only address group fairness [1, 95, 9]. Repair for individual fairness only exists for neural networks [67, 43, 40].

Alternatively, **property-specific training** methods incorporate the requirements directly into the learning procedure. Several have been proposed for training fair neural networks [77, 96] and fair (ensembles of) decision trees [61, 73, 27, 51]. Specifically for individual fairness in tree ensembles, FATT [74] uses a genetic algorithm guided by abstract interpretation to balance accuracy and individual fairness at training time. However, it only models fairness in the training objective and thus cannot provide a hard guarantee unlike our MILP repair approach. Aghaei et al. [3] employ a MILP formulation for training a fair decision tree; however, their approach operates over the training data only and does not guarantee fairness over unseen inputs. BuDRO [88] trains a GBDT ensemble using a distributionally robust loss: each boosting round solves an optimal transport problem that adversarially maximizes the loss by reweighting the training points within neighborhoods defined by a fair metric. This penalizes individual fairness violations over the training examples, but does not consider regions of the input space that are not represented in the training data. Therefore fairness is improved but not provably guaranteed. For safety, shielding [5, 65, 60, 93, 26, 54] masks unsafe actions at runtime without modifying the model, while safe RL (e.g., [2, 75]) penalizes unsafe behavior during training but has not been applied to tree-ensemble policies yet.

Finally, existing work [76, 17, 31] proposed **leaf refinement** to improve tree ensemble performance or reduce model size. Inspired by these results, we formulate tree ensemble repair as a leaf refinement task, adding hard constraints to fix counterexamples. Appendix F gives a more detailed discussion.

6 Conclusion

We introduce a MILP-based approach for repairing tree ensembles to enforce user-defined properties. Our method offers hard guarantees by provably fixing the identified counterexamples, which contrasts with existing algorithms that softly bias the learning procedure towards the property. We validate our approach on two common properties, fairness for classification models and multi-step safety for action policies. Empirically, our MILP repair substantially outperforms property-aware training methods in both scenarios, providing stronger property guarantees without sacrificing performance. Our results show that repairing learned models through constrained optimization can be feasible and competitive. As this success leverages the structure of tree ensembles, our work opens the question of which types of ML models yield the best trade-off between performance and repairability in application scenarios where conforming to certain properties is essential.

Limitations The MILP can be computationally expensive on harder instances and occasionally time out. The repair loop is bounded by the ability of the verifier to extract all counterexamples.

Impact statement This work enables enforcing user-defined properties on tree ensembles, benefiting all applications where predictive performance alone is insufficient. We see no specific negative impacts.

Acknowledgements This research is supported by the Research Foundation Flanders (FWO) via projects #11I8125N [LC] and #G039626N [JD], the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" program [JD], the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – GRK 2853/1 "Neuroexplicit Models of Language, Vision, and Action" – project number 471607914 [CJ, JH], and the French National Research Agency (ANR) under the Projet Générique program, reference ANR-26-CE23-1771 (TrustD3M) [MS].

References

[1] S. A. Abebe, C. Lucchese, and S. Orlando. EiFFFeL: Enforcing fairness in forests by flipping leaves. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022.

[2] J. Achiam, D. Held, A. Tamar, and P. Abbeel. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[3] S. Aghaei, M. J. Azizi, and P. Vayanos. Learning optimal and fair decision trees for non-discriminative decision-making. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.

[4] A. Ahmad, T. Tayal, A. Gupta, and S. Akshay. Sensitivity verification for additive decision tree ensembles. In *Proceedings of the 13th International Conference on Learning Representations*, 2025.

[5] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe reinforcement learning via shielding. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, 2018.

[6] G. Audemard, J.-M. Lagniez, P. Marquis, and N. Szczepanski. PyXAI: An XAI library for tree-based models. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence*, 2024.

[7] E. Bacci, M. Giacobbe, and D. Parker. Verifying reinforcement learning up to infinity. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, 2021.

[8] E. Bacci and D. Parker. Verified probabilistic policies for deep reinforcement learning. In *Proceedings of the 14th International Symposium on NASA Formal Methods*, 2022.

[9] M. Bagriacik and F. Otero. Fairness-guided pruning of decision trees. In *Proceedings of the 2025 ACM Conference on Fairness, Accountability, and Transparency*, 2025.

[10] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018.

[11] F. Bauer-Marquart, D. Boetius, S. Leue, and C. Schilling. SpecRepair: Counter-example guided safety repair of deep neural networks. In *Proceedings of the 28th International Symposium on Model Checking Software*, 2022.

[12] R. K. E. Bellamy, K. Dey, M. Hind, S. C. Hoffman, S. Houde, K. Kannan, P. Lohia, J. Martino, S. Mehta, A. Mojsilovic, S. Nagar, K. N. Ramamurthy, J. Richards, D. Saha, P. Sattigeri, M. Singh, K. R. Varshney, and Y. Zhang. AI Fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. *CoRR*, abs/1810.01943, 2018.

[13] H. Blockeel, L. Devos, B. Frénay, G. Nanfack, and S. Nijssen. Decision trees: From efficient prediction to responsible AI. *Frontiers in Artificial Intelligence*, 6, 2023.

[14] D. Boetius and S. Leue. Counterexample-guided repair of reinforcement learning systems using safety critics. *CoRR*, abs/2405.15430, 2024.

[15] L. Breiman. Random forests. *Machine Learning*, 45(1), 2001.

[16] C. E. Budde, C. Dehnert, E. M. Hahn, A. Hartmanns, S. Junges, and A. Turrini. JANI: Quantitative model and tool interaction. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2017.

[17] S. Buschjäger and K. Morik. Joint leaf-refinement and ensemble pruning through L1 regularization. *Data Mining and Knowledge Discovery*, 37(3), 2023.

[18] S. Calzavara, L. Cazzaro, C. Lucchese, and F. Marcuzzi. Explainable global fairness verification of tree-based classifiers. In *2023 IEEE Conference on Secure and Trustworthy Machine Learning*, 2023.

[19] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy*, 2017.

[20] S. Carr, N. Jansen, S. Junges, and U. Topcu. Safe reinforcement learning via shielding under partial observability. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, 2023.

[21] L. Cascioli, L. Devos, O. Kuzelka, and J. Davis. Faster repeated evasion attacks in tree ensembles. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, 2024.

[22] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[23] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1–2), 2003.

[24] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6), 2010.

[25] D. Cohen and O. Strichman. Automated repair of neural networks. *CoRR*, abs/2207.08157, 2022.

[26] D. Corsi, G. Amir, A. Rodríguez, C. Sánchez, G. Katz, and R. Fox. Verification-guided shielding for deep reinforcement learning. In *Proceedings of the 1st Reinforcement Learning Conference (RLC)*, 2024.

[27] A. F. Cruz, C. Belém, S. Jesus, J. Bravo, P. Saleiro, and P. Bizarro. FairGBM: Gradient boosting with fairness constraints. In *Proceedings of the 11th International Conference on Learning Representations*, 2023.

[28] J. Davis, L. Devos, S. Reyners, and W. Schoutens. Gradient boosting for quantitative finance. *Journal of Computational Finance*, 24(4), 2021.

[29] T. Decroos, L. Bransen, J. Van Haaren, and J. Davis. Actions speak louder than goals: Valuing player actions in soccer. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[30] L. Devos, L. Cascioli, and J. Davis. Robustness verification of multiclass tree ensembles. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, 2024.

[31] L. Devos, T. Martens, D. C. Oruc, W. Meert, H. Blockeel, and J. Davis. Compressing tree ensembles through level-wise optimization and pruning. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

[32] L. Devos, W. Meert, and J. Davis. Verifying tree ensembles by reasoning about potential instances. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, 2021.

[33] L. Devos, W. Meert, and J. Davis. Versatile verification of tree ensembles. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[34] L. Devos, L. Perini, W. Meert, and J. Davis. Detecting evasion attacks in deployed tree ensembles. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2023.

[35] F. Ding, M. Hardt, J. Miller, and L. Schmidt. Retiring Adult: New datasets for fair machine learning. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, 2021.

[36] G. Dong, J. Sun, X. Wang, X. Wang, and T. Dai. Towards repairing neural networks correctly. In *Proceedings of the 21st IEEE International Conference on Software Quality, Reliability and Security*, 2021.

[37] S. Dutta, X. Chen, and S. Sankaranarayanan. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019.

[38] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel. Fairness through awareness. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 2012.

[39] J. Eisenhut, X. Schuler, D. Fišer, D. Höller, M. Christakis, and J. Hoffmann. New fuzzing biases for action policy testing. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling*, 2024.

[40] A. Fayyazi, Y. Xiao, P. Nuzzo, and M. Pedram. Efficient counterexample-guided fairness verification and repair of neural networks using satisfiability modulo convex programming. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence*, 2025.

[41] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5), 2001.

[42] F. Fu and W. Li. Sound and complete neural network repair with minimality and locality guarantees. In *Proceedings of the 10th International Conference on Learning Representations*, 2022.

[43] F. Fu, Z. Wang, W. Zhou, Y. Wang, J. Fan, C. Huang, Q. Zhu, X. Chen, and W. Li. REGLO: Provable neural network repair for global robustness properties. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, 2024.

[44] S. Galhotra, Y. Brun, and A. Meliou. Fairness testing: Testing software for discrimination. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[45] J. García and F. Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16, 2015.

[46] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[47] M. M. Ghiasi and S. Zendehboudi. Application of decision tree-based ensemble learning in the classification of breast cancer. *Computers in Biology and Medicine*, 128, 2021.

[48] J. Gleeson and J. Ryan. Identifying minimally infeasible subsystems of inequalities. *ORSA Journal on Computing*, 2(1), 1990.

[49] B. Goldberger, G. Katz, Y. Adi, and J. Keshet. Minimal modifications of deep neural networks using verification. In *Proceedings of the 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 2020.

[50] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[51] V. Grari, B. Ruf, S. Lamprier, and M. Detyniecki. Fair adversarial gradient tree boosting. In *2019 IEEE International Conference on Data Mining*, 2019.

[52] L. Grinsztajn, E. Oyallon, and G. Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2022.

[53] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[54] E. Hamel-De le Court, F. Belardinelli, and A. W. Goodall. Probabilistic shielding for safe reinforcement learning. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence*, 2025.

[55] M. Hardt, E. Price, and N. Srebro. Equality of opportunity in supervised learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016.

[56] C. Huang, J. Fan, W. Li, X. Chen, and Q. Zhu. ReachNN: Reachability analysis of neural-network controlled systems. *ACM Transactions on Embedded Computing Systems*, 18, 2019.

[57] T. Ishida, G. Niu, A. K. Menon, and M. Sugiyama. Complementary-label learning for arbitrary losses and models. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[58] C. Jain, L. Cascioli, L. Devos, M. Vinzent, M. Steinmetz, J. Davis, and J. Hoffmann. Safety verification of tree-ensemble policies via predicate abstraction. In *Proceedings of the 27th European Conference on Artificial Intelligence*, 2024.

[59] C. Jain, D. Sherbakov, M. Vinzent, M. Steinmetz, J. Davis, and J. Hoffmann. Policy safety testing in non-deterministic planning: Fuzzing, test oracles, fault analysis. In *Proceedings of the 28th European Conference on Artificial Intelligence*, 2025.

[60] N. Jansen, B. Könighofer, S. Junges, A. Serban, and R. Bloem. Safe reinforcement learning using probabilistic shields. In *Proceedings of the 31st International Conference on Concurrency Theory*, 2020.

[61] F. Kamiran, T. Calders, and M. Pechenizkiy. Discrimination aware decision tree learning. In *2010 IEEE International Conference on Data Mining*, 2010.

[62] A. Kantchelian, J. D. Tygar, and A. D. Joseph. Evasion and hardening of tree ensemble classifiers. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

[63] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. LightGBM: A highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[64] Y. Kim, J. Yun, H. Shon, and J. Kim. Joint negative and positive learning for noisy labels. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[65] B. Könighofer, M. Alshiekh, R. Bloem, L. Humphrey, R. Könighofer, U. Topcu, and C. Wang. Shield synthesis. *Formal Methods in System Design*, 51(2), 2017.

[66] T. Le Quy, A. Roy, V. Iosifidis, W. Zhang, and E. Ntoutsi. A survey on datasets for fairness-aware machine learning. *WIREs Data Mining and Knowledge Discovery*, 12(3), 2022.

[67] T. Li, X. Xie, J. Wang, Q. Guo, A. Liu, L. Ma, and Y. Liu. Faire: Repairing fairness of neural networks via neuron condition synthesis. *ACM Transactions on Software Engineering and Methodology*, 2023.

[68] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

[69] T. Martens, L. Devos, L. Cascioli, W. Meert, H. Blockeel, and J. Davis. OC-space: A unifying perspective on verification of tree ensembles. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026.

[70] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015.

[71] S. Moro, P. Rita, and P. Cortez. Bank marketing [dataset]. UCI Machine Learning Repository, 2014.

[72] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2011.

[73] E. Raff, J. Sylvester, and S. Mills. Fair forests: Regularized tree induction to minimize model bias. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, 2018.

[74] F. Ranzato, C. Urban, and M. Zanella. Fairness-aware training of decision trees by abstract interpretation. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, 2021.

[75] A. Ray, J. Achiam, and D. Amodei. Benchmarking safe exploration in deep reinforcement learning. Technical report, OpenAI, 2019.

[76] S. Ren, X. Cao, Y. Wei, and J. Sun. Global refinement of random forest. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[77] A. Ruoss, M. Balunovic, M. Fischer, and M. Vechev. Learning certified individually fair representations. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020.

[78] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[79] J. Sohn, S. Kang, and S. Yoo. Arachne: Search-based repair of deep neural networks. *ACM Transactions on Software Engineering and Methodology*, 32(4), 2023.

[80] M. Sotoudeh and A. V. Thakur. Provable repair of deep neural networks. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.

[81] X. Sun, H. Khedr, and Y. Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019.

[82] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *Proceedings of the 2nd International Conference on Learning Representations*, 2014.

[83] C. Tan, Y. Zhu, and C. Guo. Building verified neural networks with specifications for systems. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2021.

[84] C. Tessler, D. J. Mankowitz, and S. Mannor. Reward constrained policy optimization. In *Proceedings of the 7th International Conference on Learning Representations*, 2019.

[85] H. Tran, F. Cai, D. M. Lopez, P. Musau, T. T. Johnson, and X. D. Koutsoukos. Safety verification of cyber-physical systems with reinforcement learning control. *ACM Transactions on Embedded Computing Systems*, 18(5s), 2019.

[86] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. S. Păsăreanu. NNrepair: Constraint-based repair of neural network classifiers. In *Proceedings of the 33rd International Conference on Computer Aided Verification*, 2021.

[87] Q. Van Baelen and P. Karsmakers. Constraint guided gradient descent: Training with inequality constraints with applications in regression and semantic segmentation. *Neurocomputing*, 556, 2023.

[88] A. Vargo, F. Zhang, M. Yurochkin, and Y. Sun. Individually fair gradient boosting. In *Proceedings of the 9th International Conference on Learning Representations*, 2021.

[89] J. P. Vielma. Mixed integer linear programming formulation techniques. *SIAM Review*, 57(1), 2015.

[90] M. Vinzent, S. Sharma, and J. Hoffmann. Neural policy safety verification via predicate abstraction: CEGAR. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, 2023.

[91] M. Vinzent, M. Steinmetz, and J. Hoffmann. Neural network action policy verification via predicate abstraction. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling*, 2022. Technical report available at https://fai.cs.uni-saarland.de/vinzent/papers/icaps22-tr.pdf.

[92] D. Vos and S. Verwer. Efficient training of robust decision trees against adversarial examples. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[93] W.-C. Yang, G. Marra, G. Rens, and L. De Raedt. Safe reinforcement learning via probabilistic logic shields. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence*, 2023.

[94] T. Yuviler and D. Drachsler-Cohen. Enhancing neural network robustness via synthesis of repair programs. In *Proceedings of the 32nd International Static Analysis Symposium*, 2025.

[95] J. Zhang, I. Beschastnikh, S. Mechtaev, and A. Roychoudhury. Fair decision making via automated repair of decision trees. In *2022 IEEE/ACM International Workshop on Equitable Data & Technology (FairWare)*, 2022.

[96] R. Zhang and J. Sun. Correct-by-construction: Certified individual fairness through neural network training. *Proceedings of the ACM on Programming Languages*, (OOPSLA2), 2025.

Appendix

A Complete MILP Encoding for Repair

A.1 Formal Proof of Repair Formula Universality

To prove Thm. 1 we first define the semantics of the repair formulas introduced in Def. 3.

Definition 6 (Repair Formula Semantics)**.** *Given a counterexample $\mathbf{e} \in \mathcal{X}^k$ for property $\mathcal{P}$ on ensemble $\boldsymbol{T}$ and a repair formula $\phi_{\mathcal{P}}(\mathbf{e})$, we say that a repaired ensemble $\boldsymbol{T}'$ satisfies $\phi_{\mathcal{P}}(\mathbf{e})$ iff*

(i) $\phi = [\boldsymbol{T}'_{label}(\mathbf{e}^i) = y]$ *and* $\boldsymbol{T}'_{label}(\mathbf{e}^i) = y$.

(ii) $\phi = [\boldsymbol{T}'_{label}(\mathbf{e}^i) = \boldsymbol{T}'_{label}(\mathbf{e}^j)]$ *and* $\boldsymbol{T}'_{label}(\mathbf{e}^i) = \boldsymbol{T}'_{label}(\mathbf{e}^j)$.

(iii) $\phi = [\neg\psi]$ *and* $\boldsymbol{T}'$ *does not satisfy* ψ*; or* $\phi = [\psi_1 \wedge \psi_2]$ *and* $\boldsymbol{T}'$ *satisfies both* ψ_1 *and* ψ_2*; or* $\phi = [\psi_1 \vee \psi_2]$ *and* $\boldsymbol{T}'$ *satisfies at least one of* ψ_1 *and* ψ_2.

The desired universality of repair formulas now follows simply because, for a fixed input $\mathbf{e}$, any repair formula boils down to a truth value assignment over the output classes $\boldsymbol{T}'_{\text{label}}(\mathbf{e})$:

Theorem 1 (Universality of Repair Formulas)**.** *Let $\mathcal{P} : \mathcal{X}^k \times \mathcal{Y}^k \to \{0, 1\}$ be a property and $\mathbf{e} \in \mathcal{X}^k$ a counterexample to $\mathcal{P}$ for ensemble $\boldsymbol{T}$. Then there exists a repair formula $\phi_{\mathcal{P}}(\mathbf{e})$ such that, for every tree ensemble $\boldsymbol{T}'$:*

$$\boldsymbol{T}' \models \phi_{\mathcal{P}}(\mathbf{e}) \quad \iff \quad \mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{label}(\mathbf{e})) = 1.$$

Proof. Assume an arbitrary tree ensemble $\boldsymbol{T}'$ and an input $\mathbf{e}$. Consider the set $P_{\mathbf{e}} := \{\mathbf{y} \in \mathcal{Y}^k \mid \mathcal{P}(\mathbf{e}, \mathbf{y}) = 1\}$ of output tuples satisfying the property. $P_{\mathbf{e}}$ is finite because $\mathcal{Y}$ is; because the property is satisfiable as per Definition 1, it follows that $P_{\mathbf{e}}$ is non-empty.

For any class assignment $\mathbf{y} = (y_0, \ldots, y_{k-1}) \in \mathcal{Y}^k$ define the repair formula

$$\psi_{\mathbf{y}} := [\boldsymbol{T}'_{\text{label}}(\mathbf{e}^0) = y_0] \wedge [\boldsymbol{T}'_{\text{label}}(\mathbf{e}^1) = y_1] \wedge \cdots \wedge [\boldsymbol{T}'_{\text{label}}(\mathbf{e}^{k-1}) = y_{k-1}]$$

which is satisfied iff $\boldsymbol{T}'_{\text{label}}(\mathbf{e}) = \mathbf{y}$.

Define $\phi_{\mathcal{P}}(\mathbf{e}) := \bigvee_{\mathbf{y} \in P_{\mathbf{e}}} \psi_{\mathbf{y}}$. Because $P_{\mathbf{e}}$ is non-empty, this disjunction is non-empty.

Now, consider the tree ensemble's output on $\mathbf{e}$, $\boldsymbol{T}'_{\text{label}}(\mathbf{e}) = (\boldsymbol{T}'_{\text{label}}(\mathbf{e}^0), \ldots, \boldsymbol{T}'_{\text{label}}(\mathbf{e}^{k-1}))$. In what follows we use the clauses (i) – (iii) from Def. 6.

($\Rightarrow$) Suppose $\phi_{\mathcal{P}}(\mathbf{e})$ is true. By clause (iii) for $\vee$ and because the disjunction is non-empty, $\boldsymbol{T}'$ satisfies some $\psi_{\mathbf{y}}$ with $\mathbf{y} \in P_{\mathbf{e}}$. Since $\psi_{\mathbf{y}}$ is a conjunction of label-assignment atoms, $\boldsymbol{T}'$ satisfies all of them (clause (iii) for $\wedge$). By clause (i), satisfying each label-assignment atom means $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = y_i$. Since this holds for every i, $\boldsymbol{T}'_{\text{label}}(\mathbf{e}) = \mathbf{y}$. Hence $\boldsymbol{T}'_{\text{label}}(\mathbf{e}) \in P_{\mathbf{e}}$, and so $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{\text{label}}(\mathbf{e})) = 1$.

($\Leftarrow$) Suppose $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{\text{label}}(\mathbf{e})) = 1$, and let $\mathbf{y} := \boldsymbol{T}'_{\text{label}}(\mathbf{e})$. Then $\mathbf{y} \in P_{\mathbf{e}}$, so $\psi_{\mathbf{y}}$ is a disjunct of $\phi_{\mathcal{P}}(\mathbf{e})$. Each conjunct of $\psi_{\mathbf{y}}$ is a label-assignment atom $[\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = y_i]$ with $y_i = \boldsymbol{T}'_{\text{label}}(\mathbf{e}^i)$, so $\boldsymbol{T}'$ satisfies it by clause (i). Hence $\boldsymbol{T}'$ satisfies $\psi_{\mathbf{y}}$ by clause (iii) for $\wedge$, and $\phi_{\mathcal{P}}(\mathbf{e})$ by clause (iii) for $\vee$. □

Observe that this proof does not make use of label-agreement atoms (ii).

A.2 MILP Encoding Details

We expand on our discussion from Section 3.1 about the hard constraints used to ensure that we fix counterexamples. We concretely show how we encode Equation (1), which introduces indicator variables $z_{i,y} \in \{0, 1\}$ for each input $\mathbf{e}^i$ and class y that allow modeling the repair formula for a counterexample $\phi_{\mathcal{P}}(\mathbf{e})$. These variables are intended to indicate whether y is the unique highest-scoring class:

$$z_{i,y} = 1 \iff \boldsymbol{T}'_y(\mathbf{e}^i) > \boldsymbol{T}'_{y'}(\mathbf{e}^i) \quad \forall y' \neq y \in \mathcal{Y} \tag{1}$$

Note that encoding $\boldsymbol{T}'_y(\mathbf{e}^i)$ in MILP is straightforward. Recall from Section 2 that (1) the output configuration $OC(\mathbf{e}^i)$ is the set of leaves activated by $\mathbf{e}^i$ in each tree $T' \in \boldsymbol{T}'$, and (2) for each leaf

$l \in OC(\mathbf{e}^i)$ and class $y \in \mathcal{Y}$, variable $\nu'_{l,y}$ represents the output assigned to class y by leaf l. Then, we can model the output for $\mathbf{e}^i$ under $\boldsymbol{T}'$ as:

$$\boldsymbol{T}'_y(\mathbf{e}^i) = \sum_{l \in OC(\mathbf{e}^i)} \nu'_{l,y}$$

For simplicity, we use $\boldsymbol{T}'_y(\mathbf{e}^i)$ to represent this linear sum constraint in MILP.

Concretely, we encode Equation (1) using the big-M method [89] thus defining M_{big} as a sufficiently large constant. For every input $\mathbf{e}^i$ and class $y \in \mathcal{Y}$, we add the following constraints to the MILP:

$$\boldsymbol{T}'_y(\mathbf{e}^i) \geq \boldsymbol{T}'_{y'}(\mathbf{e}^i) - M_{\text{big}}(1 - z_{i,y}) + \epsilon \qquad \forall y' \neq y \in \mathcal{Y}, \tag{3a}$$

$$\boldsymbol{T}'_{y'}(\mathbf{e}^i) \geq \boldsymbol{T}'_y(\mathbf{e}^i) - M_{\text{big}}(1 - w_{i,y,y'}) + \epsilon \qquad \forall y' \neq y \in \mathcal{Y}, \tag{3b}$$

$$\sum_{y' \neq y, y' \in \mathcal{Y}} w_{i,y,y'} \geq 1 - z_{i,y} \tag{3c}$$

Since MILP does not support strict inequalities, our encoding requires the winning class to outscore all other classes by at least a small positive margin $\epsilon = 0.001$. The forward direction of Equation (1) is enforced directly by Equation (3a): when $z_{i,y} = 1$, y is forced to outscore every other class. However, the reverse direction requires that whenever $z_{i,y} = 0$ there is at least one class $y' \neq y$ which scores higher for input $\mathbf{e}^i$. Equation (3a) alone does not capture this: $z_{i,y} = 0$ trivially satisfies the constraint but does not prevent y from still outscoring all other classes on $\mathbf{e}^i$. To enforce the reverse direction, we introduce auxiliary variables $w_{i,y,y'} \in \{0, 1\}$ for every $y' \neq y \in \mathcal{Y}$. When $z_{i,y} = 0$, Equation (3c) forces at least one $w_{i,y,y'} = 1$, and Equation (3b) ensures that any such $w_{i,y,y'} = 1$ forces y' to outscore y on $\mathbf{e}^i$. Together, they encode the *"if"* direction of Equation (1): $z_{i,y} = 0$ requires a concrete y' such that $\boldsymbol{T}'_y(\mathbf{e}^i) < \boldsymbol{T}'_{y'}(\mathbf{e}^i)$. Note that this encoding avoids ties by design: there is always exactly one $z_{i,y} = 1$ hence one selected class.

Given Equation (3), encoding $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ simply requires modeling each $\phi_{\mathcal{P}}(\mathbf{e})$ using the indicator variables. Each atom translates directly into linear constraints over the indicators. For example, a label-assignment atom $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = y$ becomes $z_{i,y} = 1$, and a label-agreement atom $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = \boldsymbol{T}'_{\text{label}}(\mathbf{e}^j)$ becomes $z_{i,y} = z_{j,y}$ for all $y \in \mathcal{Y}$. Boolean combinations can be encoded by introducing one auxiliary binary variable per subformula. The exact constraints for the two properties we study are:

- **fairness:** $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = \boldsymbol{T}'_{\text{label}}(\mathbf{e}^j) \;\mapsto\; z_{i,y} = z_{j,y}$ for all $y \in \mathcal{Y}$
- **safety:** $\bigwedge_{y \in \mathcal{Y}^U(\mathbf{e}^i)} \neg\,[\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = y] \;\mapsto\; z_{i,y} = 0$ for all $y \in \mathcal{Y}^U(\mathbf{e}^i)$

where $\mathcal{Y}^U(\mathbf{e}^i)$ is the set of actions that are unsafe in $\mathbf{e}^i$.

This construction gives the following property:

Theorem 5. *Let $\mathcal{P}$ be a property, $\boldsymbol{T}$ a tree ensemble and $\mathbf{E}$ a set of counterexamples. Let $\boldsymbol{T}'$ be a tree ensemble with the same trees as $\boldsymbol{T}$, but with leaf values replaced according to a solution to $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$. Then, $\boldsymbol{T}'$ satisfies $\phi_{\mathcal{P}}(\mathbf{E})$. If $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ is infeasible, then there is no $\boldsymbol{T}'$ over the same trees as $\boldsymbol{T}$ such that $\boldsymbol{T}'$ satisfies $\phi_{\mathcal{P}}(\mathbf{E})$.*

Proof. We show that the constraints of $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ are equivalent to $\phi_{\mathcal{P}}(\mathbf{E})$: an ensemble $\boldsymbol{T}'$ over the same trees as $\boldsymbol{T}$ satisfies $\phi_{\mathcal{P}}(\mathbf{E})$ if and only if its leaf values, together with the induced z and w variables, satisfy the constraints. Both claims then follow.

By the argument above, Equation (3) enforces Equation (1): for each input exactly one $z_{i,y}$ equals 1, namely $y = \boldsymbol{T}'_{\text{label}}(\mathbf{e}^i)$. The translation of the atoms is therefore faithful: $z_{i,y} = 1$ holds iff $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = y$, and $z_{i,y} = z_{j,y}$ for all $y \in \mathcal{Y}$ holds iff $\boldsymbol{T}'_{\text{label}}(\mathbf{e}^i) = \boldsymbol{T}'_{\text{label}}(\mathbf{e}^j)$. Since the encoding of $\neg, \wedge, \vee$ [89] preserves equivalence, the constraints hold exactly when $\phi_{\mathcal{P}}(\mathbf{E})$ is true under $\boldsymbol{T}'_{\text{label}}$.

Consequently, if $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ has a solution, the resulting ensemble $\boldsymbol{T}'$ satisfies $\phi_{\mathcal{P}}(\mathbf{E})$. Conversely, any $\boldsymbol{T}'$ over the same trees satisfying $\phi_{\mathcal{P}}(\mathbf{E})$ induces a feasible solution, so infeasibility implies that no such $\boldsymbol{T}'$ exists. □

Big-M. The constant M_{big} in Equation (3) must upper-bound the largest possible difference between two class scores. In our implementation we bound every leaf value to $[-B, B]$ with $B = 10^3$ to avoid numerical instabilities. Therefore this quantity is $2B \cdot |\boldsymbol{T}|$, assuming all the $|\boldsymbol{T}|$ activated leaves have value B for one class and $-B$ for the other class. However, this is a huge overestimation as our MILP is designed to encourage small leaf value changes. In practice, we can use a tighter M_{big} than the worst-case bound, which speeds up the solver. At each MILP iteration, we sum for each tree of the current ensemble the maximum inter-class difference among its leaf values:

$$M_{\text{big}} = \sum_{T \in \boldsymbol{T}} \max_{l \in \mathcal{L}_T} \max_{y,y' \in \mathcal{Y}} \left| \nu_{l,y} - \nu_{l,y'} \right|,$$

where $\mathcal{L}_T$ is the set of leaves of tree T. This bounds the differences among the inter-class outputs of the current ensemble, not of the repaired one. However, as our MILP introduces small leaf value changes, we empirically verified that this value also upper-bounds the maximum inter-class score difference for every repaired ensemble in our experiments.

A.3 Proving soundness of the MILP encoding

We can now prove soundness of our MILP encoding, formalized as follows:

Theorem 2 (Soundness of MILP Repair Constraints)**.** *Let $\mathcal{P}$ be a property, $\boldsymbol{T}$ an ensemble, and $\mathbf{E}$ a set of counterexamples. Let $\boldsymbol{T}'$ be an ensemble with the same trees as $\boldsymbol{T}$, but with leaf values replaced with a solution to $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$. Then, for all $\mathbf{e} \in \mathbf{E}$, $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{label}(\mathbf{e})) = 1$. If $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ is infeasible, then there is no $\boldsymbol{T}'$ over the same trees as $\boldsymbol{T}$ such that, for all $\mathbf{e} \in \mathbf{E}$, $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{label}(\mathbf{e})) = 1$.*

Proof. We prove the two directions of Thm. 2 separately.

($\Rightarrow$) Consider an arbitrary counterexample $\mathbf{e} \in \mathbf{E}$. By assumption the leaf values of $\boldsymbol{T}'$ satisfy $\textsc{Milp}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ and therefore by Thm. 5, $\boldsymbol{T}'$ satisfies $\phi_{\mathcal{P}}(\mathbf{e})$. Then by Thm. 1, $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{\text{label}}(\mathbf{e})) = 1$.

($\Leftarrow$) We prove the contrapositive: if there exists a tree ensemble $\boldsymbol{T}'$ sharing the same trees as $\boldsymbol{T}$ such that $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{\text{label}}(\mathbf{e})) = 1$ for all $\mathbf{e} \in \mathbf{E}$, then by Thm. 1, $\boldsymbol{T}'$ satisfies $\phi_{\mathcal{P}}(\mathbf{E})$. By Thm. 5, we then know that Equation (2) subject to Equation (3) is feasible. □

B Algorithm for the Iterative Repair Loop

Algorithm 1: An iterative tree-ensemble repair loop using MILP.

Input : Tree ensemble $\boldsymbol{T}$, property $\mathcal{P}$, training data $\mathcal{D}$
Output : Repaired ensemble $\boldsymbol{T}^\star$ satisfying $\mathcal{P}$

$\boldsymbol{T}^0 \leftarrow \boldsymbol{T}$; $\mathbf{E} \leftarrow \emptyset$; $i \leftarrow 0$
while *true* **do**
  $\mathbf{E}^i \leftarrow \texttt{Verify}(\boldsymbol{T}^i, \mathcal{P})$
  if $\mathbf{E}^i = \emptyset$ **then**
    return $\boldsymbol{T}^\star = \boldsymbol{T}^i$ `// property holds`
  $\mathbf{E} \leftarrow \mathbf{E} \cup \mathbf{E}^i$
  `// Solve repair MILP, modify the ensemble if infeasible`
  $(status, \boldsymbol{T}') \leftarrow \texttt{RepairMILP}(\boldsymbol{T}^i, \mathbf{E}, \mathcal{D})$
  while $status =$ *infeasible* **do**
    $\Phi \leftarrow \texttt{ExtractIIS}$ `// IIS of the infeasible MILP`
    $\mathbf{E}_\Phi \leftarrow \{\mathbf{e} \in \mathbf{E} \mid \mathbf{e} \text{ occurs in } \Phi\}$
    $\boldsymbol{T}^i \leftarrow \boldsymbol{T}^i \cup \{\texttt{LearnSeparatingTree}(\mathbf{E}_\Phi)\}$ `// Φ now satisfiable`
    $(status, \boldsymbol{T}') \leftarrow \texttt{RepairMILP}(\boldsymbol{T}^i, \mathbf{E}, \mathcal{D})$
  $\boldsymbol{T}^{i+1} \leftarrow \boldsymbol{T}'$; $i \leftarrow i + 1$

Algorithm 1 provides the full pseudocode for the iterative repair procedure described in Section 3.4. Each iteration verifies the current ensemble $\boldsymbol{T}^i$ against $\mathcal{P}$, accumulates any returned counterexamples

into $\mathbf{E}$, and obtains $\boldsymbol{T}^{i+1}$ by solving the repair MILP over the entire accumulated set. The inner loop (lines 8 to 12) handles infeasibility: whenever the MILP is infeasible, we extract an irreducible infeasible subsystem (IIS) Φ, identify the counterexamples $\mathbf{E}_\Phi$ that contribute to it, and modify the ensemble with a learned tree that separates these counterexamples. The new leaves are initialized to zero so the ensemble's behavior is unchanged before repair. This step is repeated until the MILP becomes feasible. The procedure terminates when verification returns no new counterexamples. If verification is exact, the returned ensemble provably satisfies the property.

B.1 Loop Termination Details and Proofs

Theorem 4 (Termination)**.** *Assuming the input space $\mathcal{X}$ is finite, the iterative repair loop terminates in finitely many iterations.*

Proof. Since $\mathcal{X}$ is finite, there are finitely many counterexamples. Every counterexample in $\mathbf{E}$ stays repaired in all subsequent ensembles (Thm. 2), so each iteration adds at least one counterexample not yet in $\mathbf{E}$. Moreover, every iteration completes, as the MILP is guaranteed to become feasible after finitely many added trees (Thm. 3). Hence the loop terminates, at the latest when $\mathbf{E}$ contains all counterexamples.

□

Theorem 6 (Termination for fairness)**.** *The iterative repair loop terminates in finitely many iterations for fairness.*

Proof. Each tree has finitely many leaves and the ensemble contains finitely many trees, thus the set of OCs $\mathcal{O}$ is finite. Hence, an infinite input space is mapped to a finite number of OCs.

Feasibility. The repair MILP is always feasible since a fair ensemble always exists. The constant classifier mapping every OC to a single label $y^\star \in \mathcal{Y}$ trivially satisfies fairness, since any two inputs receive the same label regardless of the protected attribute. Since the repair MILP is never infeasible, we never add new trees to the ensemble. Consequently, the ensemble size, and hence the set of OCs, is fixed and never increases.

Termination. At each iteration, the repair step modifies the leaf values to eliminate the set of counterexamples found until that point. Since the OCs are fixed, each ensemble in the loop induces a label assignment $\mathcal{O} \to \mathcal{Y}$. No assignment can be repeated as we accumulate counterexamples throughout iterations, because each iteration fixes all counterexamples found so far, and outputs a new assignment. Since there are finitely many label assignments $\mathcal{O} \to \mathcal{Y}$ and at least one is fair (the constant assignment), the loop terminates with a valid solution after finitely many iterations. □

Remark 7 (Termination for safety)**.** *The above argument does not transfer to safety. Safety imposes input-specific constraints: two inputs sharing an OC may have disjoint sets of safe classes, so a single label assignment to that OC cannot satisfy both. Thus, the repair MILP can be infeasible. For an unbounded input space, there can be infinitely many conflicting counterexamples leading to the ensemble growing accordingly. To recover termination, finiteness is assumed (as in Thm. 4).*

C Discussion on Infeasible MILPs

While MILP repair is provably feasible for fairness (Thm. 6), this is not the case in general, and for other properties the MILP encoding can become infeasible. This can happen for two different reasons. The first is *local*: two counterexamples that the ensemble cannot distinguish require different labels. The second is *global*: the ensemble can tell every pair of counterexamples apart, yet the hard constraints of the MILP that enforce fixing all counterexamples are still jointly unsatisfiable. In both cases, the iterative repair loop cannot fix all counterexamples by changing leaf values alone.

We illustrate the two cases with a toy example where a tree ensemble policy controls a car that drives on a two-lane road towards an obstacle. The input is defined by the lane in which the car and the obstacle are currently located (Car, $Obstacle \in \{1, 2\}$), the speed of the car ($Speed \in \mathbb{R}$), and an indicator tracking whether other cars are currently occupying the lane next to the car ($Busy \in \{0, 1\}$). The toy tree ensemble policy must either choose action *change lane* (y_1) or action *keep lane* (y_2) and is shown in Figure 3. The model consists of just two trees with two leaves each. Tree 1 checks whether

the car is in lane 1 ($Car \leq 1$), and Tree 2 checks whether the obstacle is in lane 1 ($Obstacle \leq 1$). Note that $Speed$ and $Busy$ are not used by the model.

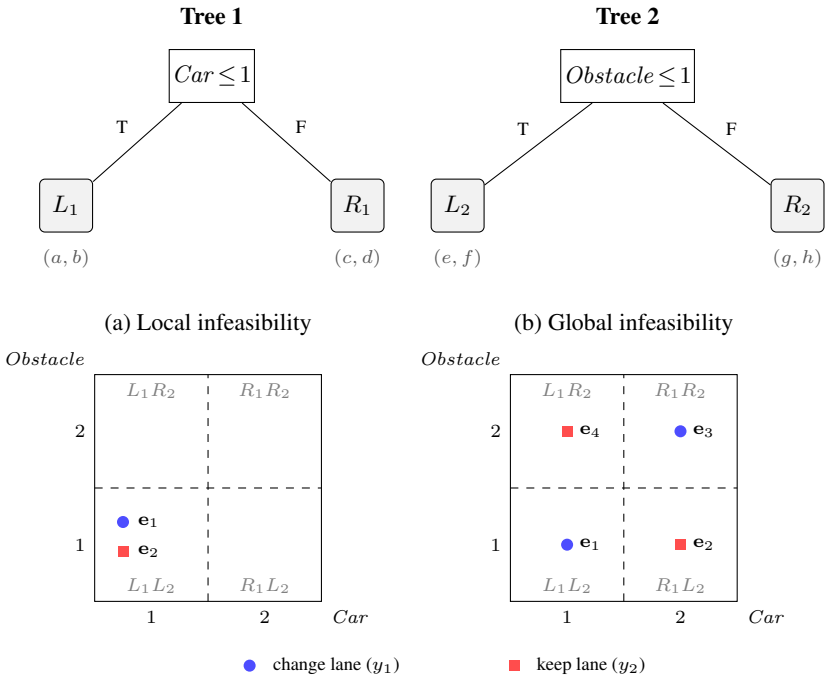

Figure 3: A tree ensemble $\boldsymbol{T}$ controlling a car on a two-lane road with an obstacle ahead, with two trees and two-dimensional leaf values $(a, b), \ldots, (g, h)$, i.e., one value for *change lane* (y_1) and one for *keep lane* (y_2), together with the four regions of the input space, one per OC. (a) The inputs $\mathbf{e}_1$ and $\mathbf{e}_2$ differ only in input variables that the ensemble ignores, hence they share the OC $\{L_1, L_2\}$, but require different labels, so the MILP is infeasible. (b) The counterexamples $\mathbf{e}_1, \ldots, \mathbf{e}_4$ all have different OCs, but no choice of leaf values can satisfy all class requirements (the car must change lanes exactly when it is in the same lane as the obstacle), so the MILP is again infeasible.

Local infeasibility. Two counterexamples $\mathbf{e}_1, \mathbf{e}_2 \in \mathcal{X}$ that fall into the same leaf in every tree of the ensemble are *indistinguishable* from the ensemble's (and therefore the MILP's) perspective. Recall that each OC corresponds to a region of the input space within which all inputs receive the same score (as they all activate the same leaves). Consequently, any modification to the leaf values affects $\boldsymbol{T}'(\mathbf{e}_1)$ and $\boldsymbol{T}'(\mathbf{e}_2)$ identically. Hence, the MILP cannot assign them different labels. For safety, this causes an infeasible MILP whenever the safe action sets of two such inputs are disjoint. Concretely, consider two counterexamples $\mathbf{e}_1, \mathbf{e}_2$ where the car and the obstacle are both in lane 1, so that both have the same output configuration $OC = \{L_1, L_2\}$ (Figure 3(a)). Suppose that in $\mathbf{e}_1$ the car is fast and cannot stop before the obstacle, and lane 2 is free ($Busy = 0$). In this case, the car must change lane, i.e., $\mathcal{Y}^U(\mathbf{e}_1) = \{y_2\}$. Instead, in $\mathbf{e}_2$ the car is slow and can therefore stop before the obstacle, but changing lanes would cause a collision because lane 2 is occupied ($Busy = 1$). In this case the car must keep its lane, i.e., $\mathcal{Y}^U(\mathbf{e}_2) = \{y_1\}$. Since the ensemble ignores the speed of the car and the traffic in the adjacent lane, no choice of leaf values can simultaneously assign a safe label to both $\mathbf{e}_1$ and $\mathbf{e}_2$.

Global infeasibility. Infeasibility can also occur even when all counterexamples have distinct OCs. Consider the four counterexamples $\mathbf{E} = \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4\}$ in Figure 3(b), one in each cell of the input space (i.e., one for each combination of $\{Car, Obstacle\}$), assuming the car is always fast and that there is no traffic in the adjacent lane:

$$OC(\mathbf{e}_1) = \{L_1, L_2\}, \quad OC(\mathbf{e}_2) = \{R_1, L_2\}, \quad OC(\mathbf{e}_3) = \{R_1, R_2\}, \quad OC(\mathbf{e}_4) = \{L_1, R_2\}.$$

In $\mathbf{e}_1$ and $\mathbf{e}_3$, the car and the obstacle are in the same lane, so the car must change lanes, i.e., $\mathcal{Y}^U(\mathbf{e}_1) = \mathcal{Y}^U(\mathbf{e}_3) = \{y_2\}$. In $\mathbf{e}_2$ and $\mathbf{e}_4$, the obstacle is in the other lane, so changing lanes would

cause a collision, i.e., $\mathcal{Y}^U(\mathbf{e}_2) = \mathcal{Y}^U(\mathbf{e}_4) = \{y_1\}$. The leaf values are $L_1 = (a, b)$, $R_1 = (c, d)$, $L_2 = (e, f)$ and $R_2 = (g, h)$, where the first is the value of y_1 (*change lane*) and the second is the value of y_2 (*keep lane*). Then, the MILP must satisfy:

$$\begin{aligned} a + e &> b + f & (\boldsymbol{T}'_{\text{label}}(\mathbf{e}_1) = y_1),\\ d + f &> c + e & (\boldsymbol{T}'_{\text{label}}(\mathbf{e}_2) = y_2),\\ c + g &> d + h & (\boldsymbol{T}'_{\text{label}}(\mathbf{e}_3) = y_1),\\ b + h &> a + g & (\boldsymbol{T}'_{\text{label}}(\mathbf{e}_4) = y_2). \end{aligned}$$

Summing the four inequalities yields $0 > 0$ which is false, so the MILP is infeasible. Intuitively, Tree 1 only sees the lane of the car and Tree 2 only the lane of the obstacle, and the ensemble adds up their scores. Whether changing lanes is safe depends on whether the car and the obstacle are in the *same* lane, which no such sum can express.

The IIS-based feasibility recovery procedure described in Section 3.3 resolves both types of infeasibility. However, local infeasibility can be cheaply detected before running the MILP by performing a simple pre-check which greatly limits the number of infeasible MILP calls, and therefore reduces feasibility recovery runtime. Concretely, we group the counterexample inputs $\mathbf{e}^i \in \mathbf{e} \in \mathbf{E}$ by their OC and check if at least one label is acceptable for all inputs within the same OC. If this is not the case, the MILP is guaranteed to be infeasible as shown above. For all those OCs, we prevent infeasibility by learning a decision tree that sorts each of their inputs into a different leaf (exactly as we do for $\mathbf{E}_\Phi$ in Section 3.3) and adding it to $\boldsymbol{T}$ prior to calling the MILP solver. In Figure 3(a), for instance, this new tree splits on either $Speed$ or $Busy$.

Conversely, global infeasibility cannot be spotted before running the MILP. In that case, we hit an infeasible MILP and we resort to the IIS extraction procedure from Section 3.3.

C.1 Recovering Feasibility through IIS resolution

Theorem 3 (Feasibility Recovery)**.** *Let $\mathcal{P}$ be a property, $\boldsymbol{T}$ a tree ensemble, and $\mathbf{E} \subseteq \{\mathbf{e} | \mathcal{P}(\mathbf{e}, \boldsymbol{T}_{label}(\mathbf{e})) = 0\}$ a set of counterexamples for which* $\text{MILP}(\boldsymbol{T}, \phi_{\mathcal{P}}(\mathbf{E}))$ *is infeasible. The above-described tree-generation process will eventually terminate with a tree ensemble $\boldsymbol{T}^f$ for which* $\text{MILP}(\boldsymbol{T}^f, \phi_{\mathcal{P}}(\mathbf{E}))$ *is feasible.*

Proof. We show that the iterative tree-addition procedure terminates with a feasible MILP. Each iteration extracts an IIS Φ, identifies the responsible counterexamples $\mathbf{E}_\Phi \subseteq \mathbf{E}$ (i.e., those appearing in at least one constraint of Φ), and adds a decision tree T_Φ that maps each distinct $\mathbf{e}^i \in \mathbf{e} \in \mathbf{E}_\Phi$ to a distinct leaf with leaf values initialized to 0. The new leaves are added as fresh free variables to the MILP without introducing additional constraints.

Φ becomes feasible after T_Φ is added. Let $f : \mathcal{X} \to \mathcal{Y}$ be a classifier satisfying $\mathcal{P}$ on every input tuple, from Def. 1. After adding T_Φ, every $\mathbf{e}^i \in \mathbf{e} \in \mathbf{E}_\Phi$ activates a distinct leaf in T_Φ, denoted by $\ell_{\mathbf{e}^i}$. Since $\ell_{\mathbf{e}^i}$ is a leaf of T_Φ activated only by $\mathbf{e}^i$ within $\mathbf{E}_\Phi$, the leaf value variables $\ell_{\mathbf{e}^i, y}$ for $y \in \mathcal{Y}$ appear in the MILP constraints associated with $\mathbf{e}^i$ but not in those associated with any other $\mathbf{e}^j \in \mathbf{E}_\Phi$. A MILP solution always exists for these constraints. Set $\ell_{\mathbf{e}^i, y} = \gamma \cdot \mathbf{I}[y = f(\mathbf{e}^i)] - \boldsymbol{T}_y(\mathbf{e}^i)$. Then, the new ensemble $\boldsymbol{T}'$ has output $\boldsymbol{T}'_y(\mathbf{e}^i) = \boldsymbol{T}_y(\mathbf{e}^i) + \ell_{\mathbf{e}^i, y}$. So,

$$\boldsymbol{T}'_y(\mathbf{e}^i) = \begin{cases} \gamma & y = f(\mathbf{e}^i) \\ 0 & y \neq f(\mathbf{e}^i) \end{cases}$$

Thus, the variables $\ell_{\mathbf{e}^i, y}$ allow assigning an independent label to each $\mathbf{e}^i$ without affecting any other $\mathbf{e}^j \in \mathbf{E}_\Phi$. Specifically, if $\gamma > 0$, $\boldsymbol{T}'$ labels every input $\mathbf{e}^i$ occurring in $\mathbf{E}_\Phi$ as $f(\mathbf{e}^i)$. Since f satisfies $\mathcal{P}$ (Def. 1), $\boldsymbol{T}'$ repairs every $\mathbf{e} \in \mathbf{E}_\Phi$, so Φ becomes feasible.

Φ does not recur. Once Φ is feasible after iteration t, it remains feasible at every subsequent iteration $t' > t$. Subsequent iterations add further trees $T_{\Phi'}$ for $\Phi' \neq \Phi$ but T_Φ remains in the ensemble, so the inputs from $\mathbf{E}_\Phi$ keep activating distinct leaves in at least one tree, hence the argument above still applies. Therefore no later IIS can involve only counterexamples of $\mathbf{E}_\Phi$.

Termination. By the previous paragraph, the counterexample set $\mathbf{E}_{\Phi'}$ of any later IIS is never contained in an earlier $\mathbf{E}_{\Phi}$. Since $\mathbf{E}$ is finite and fixed throughout the procedure, only finitely many such sets can be extracted, so the procedure terminates after finitely many iterations. When no IIS exists, the MILP is feasible, and the procedure returns a repaired ensemble $\boldsymbol{T}'$ satisfying $\mathcal{P}(\mathbf{e}, \boldsymbol{T}'_{\text{label}}(\mathbf{e})) = 1$ for all $\mathbf{e} \in \mathbf{E}$ by Thm. 2.

□

C.2 A Detailed Study of Feasibility Recovery

In our experiments infeasibility is rare: it is provably impossible for fairness (Thm. 6) and occurs only twice in safety. With the following experiment, we further show that infeasibility is not a bottleneck whenever the model is not trivially small, as it is rarer for larger models and usually captured by our cheap pre-check. We consider all 29 safety benchmarks on which counterexample generation is feasible and the MILP does not time out. For each, we repair four models of increasing size, keeping the depth fixed at 5 and increasing the number of trees $\{2, 5, 10, 20\}$, and count how often we observe infeasible instances. Table 1 shows that (1) infeasibility is almost always local, hence caught by the pre-check without ever calling the MILP, (2) it becomes rarer as model size grows and (3) fewer trees are required to recover feasibility as model size grows. Points (2) and (3) are expected, since larger ensembles induce more OCs and fewer counterexamples are forced to share one. Finally, feasibility recovery is cheap: the pre-check never takes more than one minute, learning a separator tree always takes less than one second, and infeasible MILP calls are rare, with each costing no more than one additional repair step.

Table 1: Frequency of infeasible MILPs on our safety benchmark set. LOCAL refers to the number of benchmarks where infeasibility was local and hence caught by our pre-check, GLOBAL to those where the MILP found an IIS; *Total* is the union, since a benchmark can require both. Detecting global infeasibility from the MILP takes 116 seconds in the only detected instance.

	#Infeasible benchmarks			**#Added trees**		
#Trees	LOCAL	GLOBAL	Total	LOCAL	GLOBAL	Total
2	6	1	6	52	1	53
5	3	0	3	25	0	25
10	2	0	2	19	0	19
20	2	0	2	6	0	6

D Details on Fairness Experiment

Table 2 lists the characteristics of the ten datasets considered in the experiments. In addition to providing an in-depth description of the baseline algorithms and more detailed results for the experiments presented in the main text, we also address the following questions to provide further insights into the behavior of our MILP repair approach:

- How sensitive is our MILP objective to the regularization strength λ?
- Does our composite MILP objective (Equation (2)) offer an advantage over an objective that consists of just one of its components?
- How does our soft constraint approach to accuracy preservation compare to a hard constraint approach that sets a maximum drop in predictive performance?
- Does repairing a subset of the counterexamples through our MILP also fix other unseen violations?
- How does our repair perform when using random forests [15]?

Table 2: Number of samples, features, class distribution (i.e., positive class ratio), and protected attributes for each considered fairness dataset. The task is always binary classification. For the ACS datasets, we use data from California from the year 2018.

Dataset	# Samples	# Features	Class Distr.	Protected Attribute
ACSIncome [35]	195665	10	0.411	sex
ACSEmploy [35]	378817	16	0.456	sex
ACSPublic [35]	138554	19	0.369	sex
ACSTravel [35]	172508	16	0.485	sex
Bank [71]	11162	29	0.474	married
Compas [52]	4966	11	0.500	sex
Credit [52]	13272	21	0.500	sex
DutchCensus [66]	60420	34	0.476	sex
LawSchool [66]	18692	11	0.902	sex
MEPS [12]	15675	137	0.168	sex

D.1 Details of fairness baselines

FATT [74] While suited for both decision trees and ensembles, the authors report that "ensembles of FATT trees do not introduce statistically significant benefits over single decision trees". We therefore train FATT decision trees using the original source code with default hyperparameters.[9]

BuDRO [88] We refer to the original source code.[10] The public implementation uses a different configuration per dataset with no common tuning procedure. We tuned BuDRO per fold as follows. As in the original paper, we use XGBoost [22] as the GBDT learner. We set XGBoost's $\lambda = 10^{-4}$ (L2 penalty) and *min child weight* $= 0.1/n$, where n is the number of training examples. We then tune XGBoost's max depth $D \in \{2, 4, 6\}$ and learning rate $\eta \in \{10^{-4}, 0.005, 0.02, 0.05\}$, as well as BuDRO's perturbation budget $\epsilon \in \{0.01, 0.05, 0.5, 1.0\}$. We run 200 boosting rounds and record validation metrics every 10 rounds, so the number of boosting steps is also tuned without additional fits. We consider the balanced accuracy and the *consistency* of each resulting model, where the latter is the individual fairness measure from Vargo et al. [88], defined as the fraction of individuals whose prediction is unchanged when the protected attribute is flipped. Each fold is tuned separately: models are trained on the fold's training split and evaluated on its validation split, so the test split is never used for selection. Specifically, we select the configuration with the highest validation balanced accuracy among those with consistency $= 1$, i.e., the most accurate configuration that achieves perfect individual fairness according to BuDRO's empirical measurement. The original implementation provides different solvers for the transport problem. Following the authors, we use the Sinkhorn solver for the five smaller datasets, and the dual solver for the five bigger ones. For every trained BuDRO model, Veritas either certified fairness or found at least one counterexample.

D.2 Detailed Results

Table 3 provides a detailed breakdown of Figure 1 (left) by reporting how many verified fair models we obtained for each dataset, using MILP repair and the fairness-aware training methods.

Table 4 reports the average loss in balanced accuracy with respect to the original XGBoost model. On average, the MILP-repaired models pay a loss of 1.5 pp in balanced accuracy. Part of this loss is due to the compression operation, which contributes an average loss of 0.4 pp. FATT trees have an average balanced accuracy that is 9.0 pp lower than XGB. BuDRO ensembles are on average 2.9 pp less accurate than XGB.

Furthermore, Table 5 reports the average number of iterations and the average runtime of each component of our MILP repair loop. We never require more than 22 iterations, and only 5/50 instances run for more than 5 iterations.

Finally, we study how the total MILP repair runtime scales with respect to model size (#leaves) and #counterexamples of the original LOP-compressed model. Figure 4 shows a Spearman correlation

[9] https://github.com/fatt21/fatt
[10] https://github.com/UMich-FATML/BuDRO

Table 3: Number of verified fair models per dataset (5 folds per dataset) starting from a XGBoost (XGB) ensemble. LOP is the compressed XGB model which (if verified unfair) is repaired through MILP. FATT and BuDRO directly train a fairness-aware model.

Dataset	LOP	MILP Repair	FATT	BuDRO
ACSIncome	0/5	5/5	4/5	3/5
ACSEmploy	0/5	5/5	2/5	5/5
ACSPublic	0/5	5/5	4/5	1/5
ACSTravel	0/5	5/5	5/5	4/5
Bank	3/5	5/5	4/5	2/5
Compas	2/5	5/5	0/5	4/5
Credit	4/5	5/5	5/5	3/5
DutchCensus	0/5	5/5	0/5	3/5
LawSchool	3/5	5/5	5/5	2/5
MEPS	0/5	5/5	5/5	4/5
Total	**12/50 (24%)**	**50/50 (100%)**	**34/50 (68%)**	**31/50 (62%)**

Table 4: Average balanced accuracy of the original XGB model. For LOP, repairing the compressed model with our MILP, and training a FATT and BuDRO model we report the average difference in balanced accuracy versus the original XGB model. Folds where the LOP-compressed model is already fair are excluded.

	Bal.Acc.	**Δ Bal.Acc.**			
	XGB	LOP	MILP Repair	FATT	BuDRO
ACSIncome	81.95 ± 0.10	-0.43 ± 0.06	-2.97 ± 1.00	-7.47 ± 1.64	-4.04 ± 0.25
ACSEmploy	82.57 ± 0.12	-0.52 ± 0.12	-0.87 ± 0.33	-1.90 ± 0.47	-1.54 ± 0.46
ACSPublic	69.16 ± 0.14	-0.65 ± 0.26	-1.20 ± 0.75	-11.11 ± 0.70	-4.74 ± 0.73
ACSTravel	68.75 ± 0.30	-0.63 ± 0.29	-0.76 ± 0.24	-10.25 ± 0.87	-7.64 ± 1.50
Bank	85.97 ± 0.57	-0.13 ± 0.05	-0.07 ± 0.29	-7.38 ± 0.64	-3.69 ± 1.73
Compas	66.54 ± 0.91	0.32 ± 0.76	0.08 ± 1.50	-0.55 ± 1.49	-0.81 ± 1.63
Credit	71.40 ± 0.00	-0.94 ± 0.00	-0.61 ± 0.00	-1.56 ± 0.00	-1.09 ± 0.00
DutchCensus	83.95 ± 0.25	-0.52 ± 0.34	-3.66 ± 0.32	-2.70 ± 0.84	-2.97 ± 0.48
LawSchool	78.64 ± 0.90	-0.03 ± 0.39	-0.42 ± 0.47	-28.64 ± 0.90	0.04 ± 1.13
MEPS	75.82 ± 1.56	-0.88 ± 1.00	-4.79 ± 4.45	-18.06 ± 3.31	-2.58 ± 0.82
Average	$\mathbf{76.47 \pm 7.12}$	$\mathbf{-0.44 \pm 0.39}$	$\mathbf{-1.53 \pm 1.67}$	$\mathbf{-8.96 \pm 8.79}$	$\mathbf{-2.91 \pm 2.26}$

Table 5: Average number of iterations and runtime for our MILP repair loop. We report the runtime for each component in our repair loop: XGBoost training (XGB), LOP compression (LOP), counterexample generation (CE) and repair (MILP) as well as the overall runtime. Folds where the compressed model is already fair are excluded.

Dataset	**Repair Loop**	**Runtime**				
	Iterations	XGB	LOP	CE	MILP	Total
ACSIncome	5.4 ± 1.7	$1.9\,\text{s} \pm 0.2\,\text{s}$	$56.4\,\text{m} \pm 4.8\,\text{m}$	$1.3\,\text{m} \pm 0.9\,\text{m}$	$3.7\,\text{h} \pm 1.5\,\text{h}$	$4.7\,\text{h} \pm 1.6\,\text{h}$
ACSEmploy	2.4 ± 0.5	$2.2\,\text{s} \pm 0.1\,\text{s}$	$42.2\,\text{m} \pm 5.7\,\text{m}$	$0.2\,\text{s} \pm 0.1\,\text{s}$	$0.8\,\text{s} \pm 0.3\,\text{s}$	$42.3\,\text{m} \pm 5.7\,\text{m}$
ACSPublic	6.8 ± 4.8	$1.3\,\text{s} \pm 0.4\,\text{s}$	$40.8\,\text{m} \pm 6.1\,\text{m}$	$6.4\,\text{m} \pm 10.9\,\text{m}$	$3.6\,\text{h} \pm 4.2\,\text{h}$	$4.4\,\text{h} \pm 4.3\,\text{h}$
ACSTravel	2.6 ± 0.9	$1.3\,\text{s} \pm 0.2\,\text{s}$	$40.8\,\text{m} \pm 13.2\,\text{m}$	$1.1\,\text{s} \pm 1.0\,\text{s}$	$2.0\,\text{m} \pm 4.0\,\text{m}$	$42.8\,\text{m} \pm 12.9\,\text{m}$
Bank	2.5 ± 0.7	$1.5\,\text{s} \pm 0.0\,\text{s}$	$2.3\,\text{m} \pm 0.0\,\text{m}$	$4.1\,\text{m} \pm 5.7\,\text{m}$	$32.7\,\text{m} \pm 38.8\,\text{m}$	$39.1\,\text{m} \pm 44.4\,\text{m}$
Compas	3.0 ± 1.7	$1.4\,\text{s} \pm 0.1\,\text{s}$	$1.9\,\text{m} \pm 2.2\,\text{m}$	$0.8\,\text{s} \pm 1.3\,\text{s}$	$3.7\,\text{m} \pm 6.4\,\text{m}$	$5.7\,\text{m} \pm 8.7\,\text{m}$
Credit	2.0 ± 0.0	$1.8\,\text{s} \pm 0.0\,\text{s}$	$2.7\,\text{m} \pm 0.0\,\text{m}$	$0.7\,\text{s} \pm 0.0\,\text{s}$	$2.3\,\text{s} \pm 0.0\,\text{s}$	$2.7\,\text{m} \pm 0.0\,\text{m}$
DutchCensus	3.4 ± 0.9	$1.8\,\text{s} \pm 0.3\,\text{s}$	$6.3\,\text{m} \pm 1.3\,\text{m}$	$0.1\,\text{s} \pm 0.2\,\text{s}$	$1.0\,\text{s} \pm 1.6\,\text{s}$	$6.4\,\text{m} \pm 1.3\,\text{m}$
LawSchool	12.0 ± 14.1	$0.5\,\text{s} \pm 0.0\,\text{s}$	$16.4\,\text{m} \pm 18.4\,\text{m}$	$4.9\,\text{m} \pm 6.9\,\text{m}$	$8.2\,\text{h} \pm 11.6\,\text{h}$	$8.5\,\text{h} \pm 12.0\,\text{h}$
MEPS	3.2 ± 1.1	$0.7\,\text{s} \pm 0.2\,\text{s}$	$3.1\,\text{m} \pm 0.6\,\text{m}$	$2.7\,\text{m} \pm 3.0\,\text{m}$	$1.6\,\text{h} \pm 1.5\,\text{h}$	$1.7\,\text{h} \pm 1.6\,\text{h}$

between MILP runtime and model size of 0.76, and a stronger correlation with #counterexamples of 0.93, which makes sense as each counterexample adds a hard constraint to the MILP while a larger model only adds some variables to the MILP.

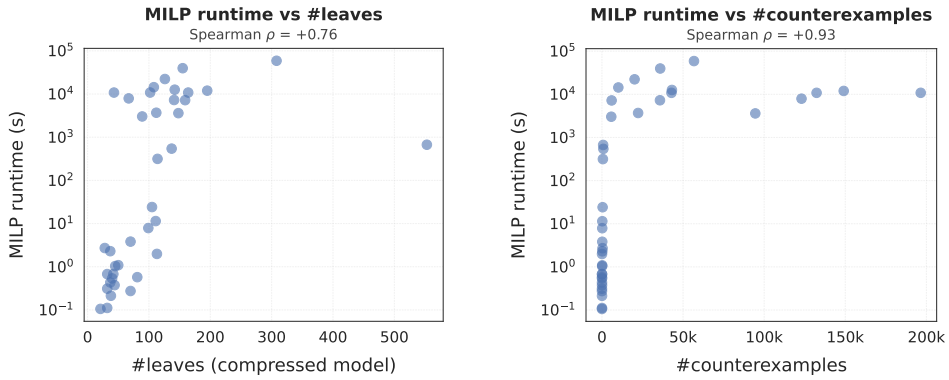

Figure 4: Correlation of MILP repair runtime with respect to the number of leaves of the ensemble (left) and the number of counterexamples for the LOP-compressed model (right).

D.3 Ablation: Impact of the Regularizer λ on the MILP Objective

We study the sensitivity of our MILP encoding to the value of the regularization strength λ in the objective. Figure 5 reports the balanced accuracy loss and MILP runtime for $\lambda \in \{10^{-5}, 10^{-4}, \ldots, 10\}$, averaged over six datasets. We exclude *ACSIncome*, *ACSPublic*, *LawSchool* and *MEPS* due to longer MILP runtimes. The results show that over-regularization of the objective appears for $\lambda > 10^{-1}$,

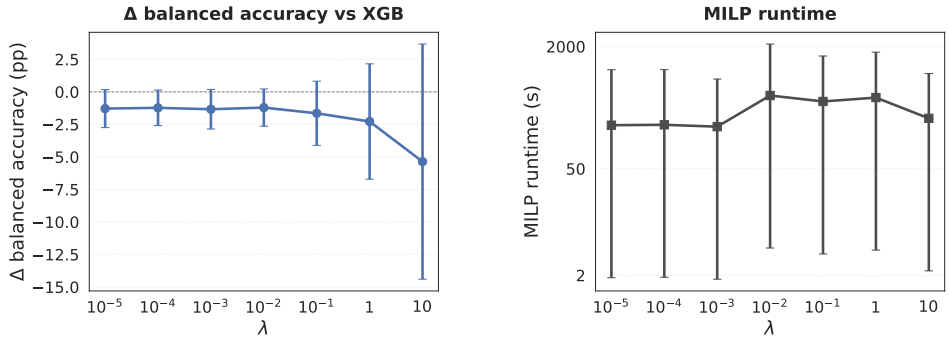

Figure 5: Comparison of balanced accuracy loss (left) and MILP repair runtime (right) using our MILP repair and varying the value of the regularization strength λ in the MILP objective. We use XGB as the original uncompressed model and exclude *ACSIncome*, *ACSPublic*, *LawSchool* and *MEPS* due to long MILP runtimes.

causing an average loss in balanced accuracy of 2.3 pp at $\lambda = 1$ and 5.4 pp at $\lambda = 10$. This is due to the L1-term forcing the MILP to keep leaf values stable, dominating the accuracy-preservation term. However, for $\lambda \leq 10^{-2}$ there is no clear trend with an average balanced accuracy loss in $[1.2, 1.3]$ pp for all the smaller values. Runtime shows no consistent trend. In our main experiment we set $\lambda = 10^{-3}$ everywhere.

D.4 Ablation of the Composite MILP Objective

The objective of our MILP encoding described in Equation (2) is a combination of the accuracy-preservation term (i.e., the minimization over the slack variables) and the L1 term that regularizes the objective by discouraging large variations in the updated leaf values. We tested two alternative MILP objectives where we only keep either the accuracy-preservation term ("Acc. only") or the L1 term ("L1 only").

Figure 6 reports balanced accuracy loss and MILP runtime for the six datasets used in Section D.3. Without L1 regularization, the solver can introduce large variations to some leaf values in order to ensure that a few training examples remain correctly classified. This hurts generalization and results

in considerably larger declines in balanced accuracy for the "Acc. only" variant. As shown in the previous section, any small $\lambda > 0$ prevents this.

Conversely, the "L1 only" objective simply tries to find the smallest perturbations to the leaf values that ensure fairness. Because it does not consider predictive performance, it does not account for the fact that there can be slightly larger perturbations which fix fairness and result in smaller drops in predictive performance. Therefore the loss in balanced accuracy is again larger with respect to our composite objective.

The "L1 only" version is faster (Figure 6, right) because it does not have to model one slack variable for each training example.

We therefore conclude that our accuracy-aware objective successfully models the trade-off between enforcing fairness and retaining model performance.

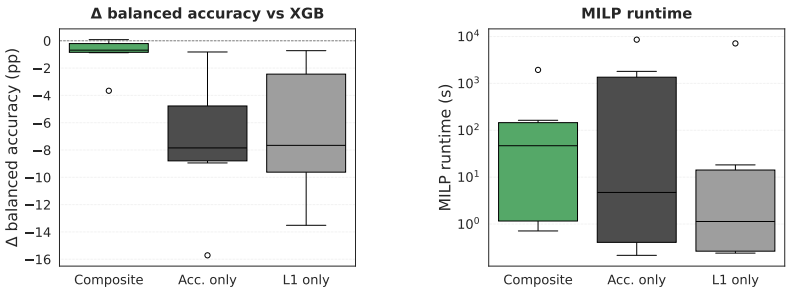

Figure 6: Comparison of balanced accuracy loss (left) and MILP repair runtime (right) using our original MILP objective and two variants that consider only the accuracy-preservation term ("Acc. only") or the L1 term ("L1 only"). We use XGB as the original uncompressed model and exclude *ACSIncome*, *ACSPublic*, *LawSchool* and *MEPS* due to long MILP runtimes.

D.5 Enforcing Accuracy through Hard Constraints

Our MILP formulation from Equation (2) minimizes accuracy loss through the slack variables in the objective. This is an approximate way of modeling accuracy loss, as a slack variable is a continuous variable that records how far a misclassified example is from the decision boundary, meaning that not all misclassifications contribute to the objective in the same way. A more accurate model would use a binary variable for each training example, setting it to 1 if the example is correctly classified with the current leaf values and enforcing a strict bound on the number of allowed misclassifications. Concretely, this "Hard Budget" MILP version substitutes Equation (2) with:

$$\min_{\overrightarrow{\nu}'_{\mathbf{E}}} \quad \|\overrightarrow{\nu}'_{\mathbf{E}} - \overrightarrow{\nu}_{\mathbf{E}}\|_1 \tag{4}$$

$$\begin{aligned}
\text{s.t.} \quad & \forall (x, y) \in \mathcal{D} : \\
& u_x \in \{0, 1\} \\
& u_x = 1 \;\Rightarrow\; \boldsymbol{T}'_y(x) - \max_{y' \in \mathcal{Y} \setminus \{y\}} \boldsymbol{T}'_{y'}(x) \;>\; 0 \\
& u_x = 0 \;\Rightarrow\; \boldsymbol{T}'_y(x) - \max_{y' \in \mathcal{Y} \setminus \{y\}} \boldsymbol{T}'_{y'}(x) \;\leq\; 0 \\
& \frac{1}{|\mathcal{Y}|} \sum_{c \in \mathcal{Y}} \frac{1}{|\mathcal{D}_c|} \sum_{x \,:\, y = c} u_x \;\geq\; \mathrm{BAcc}(\boldsymbol{T}) - b
\end{aligned}$$

where $u_x = 1$ indicates that training example x is correctly classified and viceversa, $\mathcal{D}_c$ indicates the set of training examples belonging to class c, and $\mathrm{BAcc}(\boldsymbol{T})$ and b are respectively the starting LOP-compressed model's training balanced accuracy and the maximum allowed loss. The hard constraints that fix counterexamples remain identical to Equation (1). This alternative MILP is guaranteed to either find a solution within the predefined accuracy loss budget b, or hit infeasibility if such a solution does not exist. However, it has the downside of introducing a large number of binary variables u_x to the MILP, which heavily impacts runtime.

We tested this alternative implementation on the main experiment from Section 4.1, setting 2 pp as the maximum allowed accuracy loss. Within each repair iteration, MILP has a 1h time limit. Results from Figure 7 (left) show that the "Hard Budget" MILP fails to repair 20 of the 38 folds ($\sim$53%) where the LOP-compressed model is unfair. On 15 folds ($\sim$40%), the MILP takes too long and times out, and in the remaining 13% no solution exists within the 2 pp accuracy loss threshold.

Figure 7 (center) shows that when it successfully terminates, the "Hard Budget" MILP achieves a similar performance to our original objective. However, it exhibits a greater variance. Note that in some cases, the "Hard Budget" MILP's balanced accuracy is 2.5 pp lower than the XGB model's because we need to sum the compression loss of $\sim$0.5 pp to the budget $b = 2$ pp.

Crucially, the "Hard Budget" MILP usually takes longer (Figure 7, right), which is also reflected in the 15 timeouts out of 38 instances. Therefore our original formulation has two key advantages: it is a lot faster and it is never infeasible. The "Hard Budget" MILP is too computationally expensive to be widely applied in practice, as it fails to repair the model in more than half of our instances. Moreover, it is not guaranteed that a solution exists within the specified budget. Hence, one would have to tune this drop in performance in order to ensure feasibility, which further adds to the computational cost of this approach.

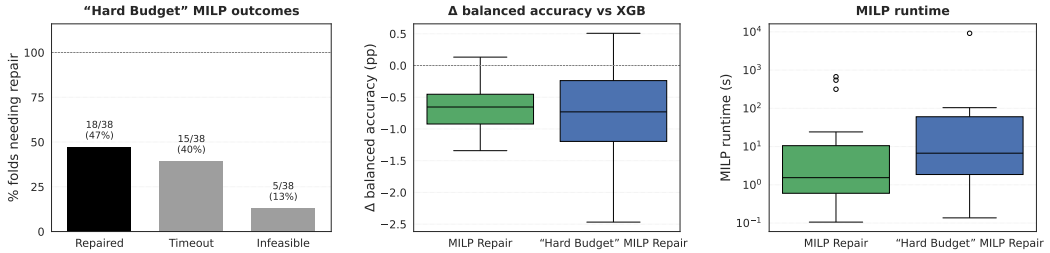

Figure 7: Repair results using an alternative MILP that enforces a hard budget of 2 pp on the training balanced accuracy loss of the model repaired through our loop. Left: fraction of instances where the "Hard Budget" MILP successfully terminates, times out after 1 hour, or is infeasible. We consider all 38 folds where the LOP-compressed model is unfair. Center: balanced accuracy loss with respect to the original uncompressed XGB model for the two MILP versions, considering folds where both methods ended with a fair model. Right: MILP repair runtime for the two MILP versions, on the same folds.

D.6 Generalization of MILP Repair

We next investigate whether repairing a subset of the counterexamples through our MILP also fixes other unseen violations. To do so, we take the LOP-compressed models (i.e., before attempting any repairs) from our main experiment and exhaustively generate all counterexamples for each model using Veritas. We exclude instances with fewer than 10 counterexamples or where the MILP takes > 10m to run, which leaves 22/50 instances. We then randomly split the counterexamples into a repair set and a held-out validation set, and we run one single MILP repair step fixing the repair set only, checking whether the repair also fixes unseen counterexamples from the validation set. We repeat the experiment for various repair set sizes. Figure 8 shows that on average, repairing just 25% of the counterexamples already fixes $> 60\%$ of the unseen ones, which grows to 80% using 50% of the counterexamples. Thus MILP repair exhibits the ability to generalize to unseen violations which could be exploited to reduce computational cost in instances where the MILP is slow due to the counterexample set being particularly large, e.g., by capping the number of counterexamples considered in each MILP iteration. However, runtime was not an issue for our proposed approach as no MILP calls timed out.

D.7 Fairness Repair for Random Forest

We now explore using random forests [15] (i.e., another popular tree ensemble model class) as starting models in our fairness experiment. Using the same 10 datasets from Section 4.1, we learn a random forest with 50 trees and max depth 8, and compress it with LOP. For all aspects of the repair loop

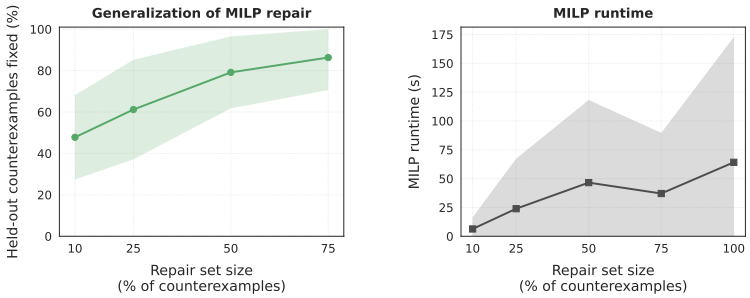

Figure 8: Fraction of unseen counterexamples fixed as a function of repair set size (left) and runtime to fix increasingly larger fractions of counterexamples (right).

with random forests, we use the same hyperparameter values as described in Section 4.1. We again compare to FATT and BuDRO.

Figure 9 (left) and Table 6 report how many verified fair models we obtained for each dataset, using MILP repair and fairness-aware training methods. The results are qualitatively similar to those for XGB models. 40% of the RF models compressed with LOP are fair. Our MILP repair loop generates a provably fair model in all instances. FATT trees are only fair 68% of the time, and BuDRO ensembles are fair on 62% of the instances. Hence, while they offer improved fairness compared to the baseline LOP models, unlike our approach, they are not able to guarantee that each model is fair.

Figure 9 (right) and Table 7 report the average differences in balanced accuracy with respect to the original RF model. On average, our MILP repair approach's balanced accuracy is $\sim$1.4 pp lower than the RF. Conversely, FATT trees and BuDRO ensembles lose on average 5.7 pp and 2.1 pp in balanced accuracy with respect to the original RF. Hence, they pay a greater price in terms of the loss in predictive performance than our approach even though they do not guarantee fairness. These results show that our MILP repair approach is applicable to multiple tree ensemble learners.

Finally, Table 8 reports the average number of iterations and runtime of each component of the MILP repair loop. The repair loop (i.e., counterexample generation + MILP repair) repairs all but three instances in less than 10 minutes, and never requires more than 5 iterations to converge.

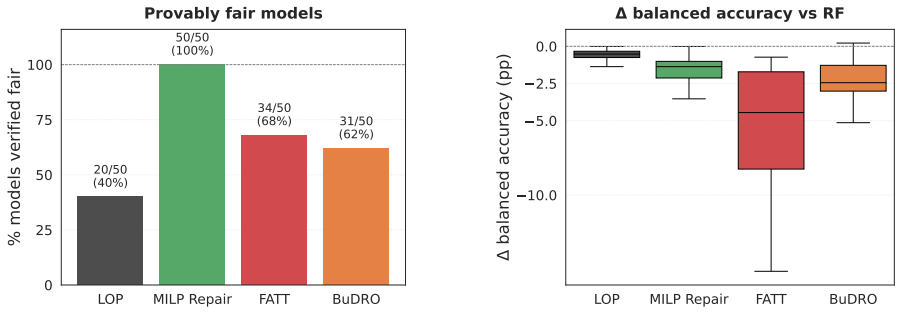

Figure 9: Main experiment results using RF models. Left: fraction of provably fair models among the models generated with LOP, the MILP-repaired models, and the models trained with the fairness-aware baselines. There are 50 models in total (10 datasets $\times$ 5 folds). Right: balanced accuracy loss with respect to the original uncompressed RF model.

E Details on Safety Experiment

In what follows, we first formalize the semantics of our safety-specific context in Section E.1. Next, we provide additional details on our benchmarks and baselines in Section E.2 and Section E.3. In Section E.4 we expand on the experimental results described in Section 4.2.

Table 6: Number of verified fair models per dataset (5 folds per dataset) starting from a random forest (RF) ensemble. LOP is the compressed RF model which (if verified unfair) is repaired through MILP. FATT and BuDRO directly train a fairness-aware model.

Dataset	LOP	MILP Repair	FATT	BuDRO
ACSIncome	0/5	5/5	4/5	3/5
ACSEmploy	1/5	5/5	2/5	5/5
ACSPublic	0/5	5/5	4/5	1/5
ACSTravel	2/5	5/5	5/5	4/5
Bank	3/5	5/5	4/5	2/5
Compas	2/5	5/5	0/5	4/5
Credit	4/5	5/5	5/5	3/5
DutchCensus	0/5	5/5	0/5	3/5
LawSchool	5/5	5/5	5/5	2/5
MEPS	3/5	5/5	5/5	4/5
Total	**20/50 (40%)**	**50/50 (100%)**	**34/50 (68%)**	**31/50 (62%)**

Table 7: Average balanced accuracy of the original RF model. For LOP, repairing the compressed model with our MILP, and training a FATT and BuDRO model we report the average difference in balanced accuracy versus the original RF model. Folds where the LOP-compressed model is already fair are excluded. Results for *LawSchool* are omitted as all LOP-compressed models were fair.

	Bal.Acc.	**Δ Bal.Acc.**			
	RF	LOP	MILP Repair	FATT	BuDRO
ACSIncome	80.81 ± 0.11	-0.54 ± 0.12	-1.64 ± 0.49	-6.34 ± 1.63	-2.90 ± 0.15
ACSEmploy	82.01 ± 0.15	-0.40 ± 0.11	-0.77 ± 0.47	-1.27 ± 0.43	-1.05 ± 0.34
ACSPublic	67.91 ± 0.29	-0.61 ± 0.40	-0.75 ± 0.36	-9.86 ± 0.86	-3.49 ± 0.91
ACSTravel	65.04 ± 0.50	-0.67 ± 0.27	-1.43 ± 0.85	-6.31 ± 1.18	-4.59 ± 1.71
Bank	83.48 ± 0.37	-1.12 ± 0.35	-1.22 ± 0.29	-4.14 ± 0.39	-0.59 ± 1.14
Compas	66.21 ± 0.62	-0.87 ± 0.21	-1.52 ± 0.43	-1.59 ± 0.09	-1.21 ± 1.22
Credit	71.18 ± 0.00	-0.05 ± 0.00	-0.01 ± 0.00	-1.31 ± 0.00	-0.55 ± 0.00
DutchCensus	83.50 ± 0.20	-0.35 ± 0.19	-3.33 ± 0.39	-2.25 ± 1.02	-2.52 ± 0.38
MEPS	75.44 ± 2.10	-1.06 ± 1.23	-2.08 ± 0.53	-18.39 ± 4.61	-1.67 ± 1.21
Average	$\mathbf{75.07 \pm 7.65}$	$\mathbf{-0.63 \pm 0.35}$	$\mathbf{-1.42 \pm 0.94}$	$\mathbf{-5.72 \pm 5.58}$	$\mathbf{-2.06 \pm 1.40}$

Table 8: Average number of iterations and runtime for our MILP repair loop. We report the runtime for each component in our repair loop: random forest training (RF), LOP compression (LOP), counterexample generation (CE) and repair (MILP) as well as the overall runtime. Folds where the compressed model is already fair are excluded. Results for *LawSchool* are omitted as all LOP-compressed models were fair.

Dataset	**Repair Loop**	**Runtime**				
	Iterations	RF	LOP	CE	MILP	Total
ACSIncome	3.6 ± 0.5	$1.4\,\text{s} \pm 0.1\,\text{s}$	$51.3\,\text{m} \pm 10.7\,\text{m}$	$0.4\,\text{s} \pm 0.3\,\text{s}$	$2.5\,\text{m} \pm 4.3\,\text{m}$	$53.8\,\text{m} \pm 11.2\,\text{m}$
ACSEmploy	2.2 ± 0.5	$1.9\,\text{s} \pm 0.1\,\text{s}$	$55.6\,\text{m} \pm 15.8\,\text{m}$	$0.1\,\text{s} \pm 0.0\,\text{s}$	$0.1\,\text{s} \pm 0.0\,\text{s}$	$55.6\,\text{m} \pm 15.8\,\text{m}$
ACSPublic	2.8 ± 1.1	$0.7\,\text{s} \pm 0.1\,\text{s}$	$1.0\,\text{h} \pm 0.3\,\text{h}$	$0.4\,\text{s} \pm 0.4\,\text{s}$	$1.8\,\text{m} \pm 3.5\,\text{m}$	$1.1\,\text{h} \pm 0.3\,\text{h}$
ACSTravel	2.7 ± 1.2	$1.3\,\text{s} \pm 0.0\,\text{s}$	$54.0\,\text{m} \pm 0.9\,\text{m}$	$0.1\,\text{s} \pm 0.0\,\text{s}$	$0.1\,\text{s} \pm 0.1\,\text{s}$	$54.0\,\text{m} \pm 0.9\,\text{m}$
Bank	3.0 ± 0.0	$0.3\,\text{s} \pm 0.0\,\text{s}$	$3.0\,\text{m} \pm 1.5\,\text{m}$	$0.8\,\text{s} \pm 0.9\,\text{s}$	$13.5\,\text{s} \pm 16.0\,\text{s}$	$3.3\,\text{m} \pm 1.8\,\text{m}$
Compas	3.3 ± 1.5	$0.2\,\text{s} \pm 0.0\,\text{s}$	$2.9\,\text{m} \pm 2.1\,\text{m}$	$1.5\,\text{s} \pm 2.6\,\text{s}$	$21.6\,\text{m} \pm 37.4\,\text{m}$	$24.5\,\text{m} \pm 39.5\,\text{m}$
Credit	4.0 ± 0.0	$0.4\,\text{s} \pm 0.0\,\text{s}$	$11.1\,\text{m} \pm 0.0\,\text{m}$	$1.4\,\text{m} \pm 0.0\,\text{m}$	$2.0\,\text{h} \pm 0.0\,\text{h}$	$2.2\,\text{h} \pm 0.0\,\text{h}$
DutchCensus	3.2 ± 1.3	$0.4\,\text{s} \pm 0.1\,\text{s}$	$8.4\,\text{m} \pm 1.2\,\text{m}$	$0.1\,\text{s} \pm 0.1\,\text{s}$	$0.4\,\text{s} \pm 0.4\,\text{s}$	$8.4\,\text{m} \pm 1.2\,\text{m}$
MEPS	2.0 ± 0.0	$0.4\,\text{s} \pm 0.0\,\text{s}$	$3.8\,\text{m} \pm 0.6\,\text{m}$	$0.1\,\text{s} \pm 0.1\,\text{s}$	$0.6\,\text{s} \pm 0.8\,\text{s}$	$3.9\,\text{m} \pm 0.6\,\text{m}$

E.1 Background

We consider action policies in discrete Markov decision processes (MDPs), with states modeled as value assignments to a finite set of state variables. The MDP state space is exponentially large in the number of state variables, or infinite if some of the state variables are continuous-valued. While the policies are learned taking the transition probabilities in the MDP into account, in our safety repair we address worst-case safety, requiring the policy to be safe under all possible circumstances. For the

purpose of the safety analysis and repair, we can drop the probabilities in the MDP and address what is known as fully-observable non-deterministic planning [e.g., 23].

For safety, the input space $\mathcal{X}$ defines a (possibly infinite) set of **states** and $\mathcal{Y}$ defines a set of **action labels**. Consequently, a **state space** in our framework is a tuple $\Theta = \langle \mathcal{X}, \mathcal{Y}, \mathcal{T}, \mathcal{X}_0, \mathcal{X}_* \rangle$. We assume that states are value assignments to a given finite set of **state variables** $\mathcal{V}$, which may be Boolean, integer, or real-valued. $\mathcal{T} : \mathcal{X} \times \mathcal{Y} \to 2^{\mathcal{X}} \setminus \{\emptyset\}$ is the **transition function**, that maps a given state x and action y to the set of possible successor states x'. If $x' \in \mathcal{T}(x, y)$ is a possible successor of x under action y, we write $x \to_y x'$. $\mathcal{X}_0 \subseteq \mathcal{X}$ is the non-empty set of **initial states** and $\mathcal{X}_* \subseteq \mathcal{X}$ is the set of **goal states**. We denote by $app(x) \subseteq \mathcal{Y}$ the set of actions **applicable** in state x, i.e., those for which $\mathcal{T}(x, y)$ is non-empty. We assume w.l.o.g. that $app(x) \neq \emptyset$ for all $x \in \mathcal{X}$.

We assume that we have access to $\mathcal{V}$, and that $\mathcal{X}_0$ and $\mathcal{X}_*$ are specified in terms of given formulas ϕ_0 and ϕ_* over $\mathcal{V}$. Also, we assume a given **failure condition** ϕ_F in the form of a formula over $\mathcal{V}$. By $\mathcal{X}^F \subset \mathcal{X}$ we denote the set of **failed states** where ϕ_F is true. A **policy** is a function $\pi : \mathcal{X} \mapsto \mathcal{Y}$ such that $\pi(x) \in app(x)$ for all states $x \in \mathcal{X}$. A **run** of π from a state $x^0 \in \mathcal{X}$ is an alternating sequence of states and actions $\sigma^{\pi}_{x^0} = x^0, \pi(x^0), x^1, \ldots$ such that, for all $i \geq 0$, $x^{i+1} \in \mathcal{T}(x^i, \pi(x^i))$ and if $x^i \in \mathcal{X}_* \cup \mathcal{X}^F$ then the path terminates at x^i while otherwise the path continues. If the run ends in $x^n \in \mathcal{X}^F$, we say that it is **unsafe**. A policy π is **unsafe** if there exists an unsafe run for some $x^0 \in \mathcal{X}_0$, and is **safe** otherwise. Note that a safe policy must never reach a failed state, regardless of the initial state x^0 and the outcomes of the probabilistic transitions.

A state x is **safe** if there exists a policy s.t. no run starting from x is unsafe. We denote by $\mathcal{X}^S \subseteq \mathcal{X}$ the set of all safe states, called the **safe region**. All other states form the **unsafe region** $\mathcal{X}^U = \mathcal{X} \setminus \mathcal{X}^S$. Failed states are unsafe, i.e., $\mathcal{X}^F \subseteq \mathcal{X}^U$, but in general $\mathcal{X}^U \not\subseteq \mathcal{X}^F$ (e.g., in $\mathcal{X}^U$ a car may be too fast to brake, while in $\mathcal{X}^F$ it has already crashed). We say that an action y is **unsafe** for a state x iff (1) x is in the safe region, i.e., $x \in \mathcal{X}^S$ and (2) a possible successor is in the unsafe region, i.e., $\exists x' \in \mathcal{T}(x, y) : x' \in \mathcal{X}^U$.

Remark 8. *Note that constructing the repair formula* $\phi_P(\mathbf{e}) = \bigwedge_{y \in \mathcal{Y}^U(\mathbf{e})} \neg(\boldsymbol{T}'_{label}(\mathbf{e}) = y)$ *requires knowing the full set of unsafe actions* $\mathcal{Y}^U(\mathbf{e})$ *in counterexample state* $\mathbf{e}$*. As computing unsafe actions is computationally expensive, in practice we implement an on-demand version of this process. We identify unsafe actions only where chosen by the policy, and use the subset of currently known unsafe actions instead of* $\mathcal{Y}^U(\mathbf{e})$ *in the repair formula, forcing the policy to choose differently.*

E.2 Benchmarks

A benchmark instance is a pair of (1) a non-deterministic planning task described in the JANI language [16] and (2) a neural network policy tackling that problem. Our benchmark set is taken from prior work on fault analysis for action policies [59], and we train a tree ensemble policy by imitation learning from the neural policy. The training set is generated by executing these neural policies from 1000 randomly sampled start states, collecting all pairs of state and action value predictions on these runs. Specifically, we use two sets of benchmarks. We describe them below.

Discrete state space benchmarks These are benchmarks (*Beluga*, *Blocksworld* and *Transport*) from prior work on action-policy safety verification [91, 58] as well as *1 way line* and *2 way line* from work on action-policy testing [59].

Beluga Inspired by a logistics application at Airbus, product parts arriving on a Beluga transport plane must be sent to a factory in a particular order, potentially different from the order in which they are arriving. There is a number of racks in which arriving parts can be stored temporarily. The start condition permits arbitrary orderings of the arriving parts. The goal requires having sent all parts to the factory, under non-determinism which part will be requested next at each step. A state is a failed state if parts are loaded and unloaded on racks too often (modeling the practical guideline to only deliver parts when requested).

Blocksworld Actions moving a block b may non-deterministically fail, and when this happens the cost of moving block b is incremented. The start condition imposes a partial order on the blocks in the initial stacks. A state is a failed state if the number of blocks on the table exceeds a fixed limit. The goal requires arranging the blocks in a predetermined manner. We consider instances with 8 blocks.

Transport A truck must deliver packages on a straight line road to the other side of a bridge, and a failed state occurs if the truck has too many packages while crossing the bridge. The start condition distributes the packages on all locations on the "non-goal" side of the bridge.

1 way line / 2 way line A truck moves along a discrete line in respectively one or both directions. The truck can accelerate and decelerate by one speed unit at a time, and pick up and drop packages if its velocity is 0. We model two variants, one deterministic and one non-deterministic. Non-determinism might cause the truck to drop packages while moving. Moreover, acceleration and deceleration might fail on icy roads. We specifically consider two non-deterministic variants: one with an additional parking action, which we refer to with *I+P* and one disabling parking, suffixed as *Icy*. The safety constraint requires the policy to not drive past either end of the line. The goal constraint is to deliver all packages to a predetermined location at the end of the line.

Continuous state space benchmarks (control benchmarks) These are JANI benchmarks modeling the deterministic variants of control problems *Bouncing Ball*, *Follow Car* and *Inverted Pendulum* often considered in reinforcement learning [7, 8]. They are parameterized by a timestep τ, which determines how frequently the system is updated. A smaller τ enables more fine-grained control, while a larger τ results in less frequent but more substantial updates to the system dynamics. In all domains, the goal constitutes surviving for 2000 timesteps.

Bouncing Ball The agent controls a ball, choosing to hit it with a paddle or do nothing. The ball loses 10% of its energy upon hitting the ground and eventually stops bouncing. Safety requires continuously bouncing.

Follow Car There are two vehicles, *lead* and *ego*. The lead vehicle is moving at a constant speed whilst the agent controls the acceleration of the ego vehicle. The safety constraint is that the two vehicles never crash.

Inverted Pendulum An agent applies left or right rotational force to a pole pivoting around one of its ends, with the aim of balancing the pole in the upright position. Safety constitutes remaining in a range of positions and velocities such that the upright position can be recovered.

E.3 Baselines

We now describe in more detail the baselines considered for safety repair in Section 4.2. These baselines follow two general approaches: given a set of counterexamples, we can either (1) construct a shield that filters out the unsafe model output for each identified counterexample, or (2) retrain a new model using information from the counterexamples, thereby discouraging the learner from reproducing them (but without any built-in guarantee).

Penalty Trees (PT) This baseline simply masks out the model decisions that violate the property. We do so by adding one tree to the ensemble for every counterexample $\mathbf{e}$. Recall that here a counterexample is a state. Specifically, we add a tree $P_{\mathbf{e}}$ that heavily penalizes choosing $\boldsymbol{T}_{\text{label}}(\mathbf{e}) = y^u \in \mathcal{Y}^U(\mathbf{e})$ (i.e., the unsafe actions) for $\mathbf{e}$, so that no unsafe action is selected anymore in that state. A tree $P_{\mathbf{e}}$ that *only applies* to state $\mathbf{e}$ can be constructed by recursively adding an internal node of the form $x_f = \mathbf{e}_f$ for each state variable x_f. The left child (i.e., the test evaluates to true) becomes the subsequent internal node and the right child is a leaf node. Let l_P be the leaf node reached by satisfying all split tests (i.e., the leaf to which only $\mathbf{e}$ is sorted). We set all leaf values to 0, except the value in l_P associated with every unsafe class y^u which is set to some large penalty value P.

The penalty P must be sufficiently negative to ensure that the unsafe action is never selected, i.e., at least one other action is assigned a larger output than y^u by the repaired ensemble. It follows that P must be smaller than the difference between the minimum and maximum possible values of the current ensemble output. We compute P as follows. For each tree T in the original tree ensemble, we recursively traverse the tree and collect, for each class y, the minimum and maximum values assigned to y at any of the leaves. This results in a set of per-class bounds $(\min_{l \in \mathcal{L}_T} \nu_{l,y}, \max_{l \in \mathcal{L}_T} \nu_{l,y})$ where $\mathcal{L}_T$ denotes the set of leaf nodes in tree T, and $\nu_{l,y}$ is the value assigned to class y at leaf l. From this set of bounds, we extract the overall minimum and maximum values across all actions for tree T:

$$\nu^T_{\min} = \min_{y \in \mathcal{Y}} \min_{l \in \mathcal{L}_T} \nu_{l,y}, \quad \nu^T_{\max} = \max_{y \in \mathcal{Y}} \max_{l \in \mathcal{L}_T} \nu_{l,y}.$$

The penalty component related to tree T is then defined as: $P_T = \nu^T_{\min} - \nu^T_{\max}$ which is guaranteed to be non-positive. Finally, the global penalty P is the sum of these values across all trees: $P = \sum_{T \in \boldsymbol{T}} P_T$. This formulation ensures that the employed penalty value is smaller than any possible difference between two class outputs of the ensemble.

Remark 9. *The penalty tree construction gives the desired repair guarantee: each counterexample is provably fixed. However, this suffers from two major weaknesses. First, the tree-ensemble size grows linearly in the number of counterexamples, which is inefficient. Second, the penalty terms apply only to these specific counterexamples, so the fixes cannot generalize to other similar counterexamples. Therefore fixing all counterexamples may require adding an exponentially large number of trees, which will fail due to resource limitations. We fix both issues with our MILP repair method.*

Retraining-based methods We next consider two baselines based on the second idea: retraining a model from scratch. At each iteration of the loop, both baselines learn a new tree ensemble with the same depth and number of trees as the original one while biasing the learning procedure away from exhibiting the previously discovered counterexamples. To speed up repair, we associate each counterexample with a weight in the loss function. Each time a counterexample is not fixed during an iteration, its weight is doubled to focus the learner's attention on ensuring it ultimately corrects as many of the counterexamples as is possible [11].

Model Hardening (MH) This is the analog of model hardening for training more robust models [50, 62]. In this approach, we create an augmented training set $\mathcal{D}_{hard}$ based on the original training set $\mathcal{D}_{orig}$ and the identified counterexamples $\mathbf{E}$. For each state in $\mathcal{D}_{orig}$ and $\mathbf{E}$, we obtain the training label for $\mathcal{D}_{hard}$ by querying the neural teacher policy, masking out any action that was previously identified as being unsafe. We then use imitation learning on this hardened training set.

Fault-Aware Loss (FALoss) This method uses only the negative information contained in the counterexamples. That is, for each counterexample $\mathbf{e}$ we only know that the output $\boldsymbol{T}_{\text{label}}(\mathbf{e}) = y$ is currently in $\mathcal{Y}^U(\mathbf{e})$, which should be avoided, but we do not know which action should be chosen instead. We train a new tree ensemble through imitation learning, and incorporate the negative information via a loss function that penalizes selecting a forbidden action for a counterexample, using the found counterexamples as additional negative training samples. This is analogous to safe reinforcement learning techniques that include a penalty term in the objective function that discourages selecting unsafe actions [45, 2, 84], and has recently been proposed for neural network repair [11]. Specifically, we train an XGBoost ensemble on the original training set $\mathcal{D}_{orig}$ and the set of counterexamples $\mathbf{E}$ using the following custom loss function:

$$\mathcal{L}_{\text{retrain}} = \sum_{(x,y)\in\mathcal{D}_{orig}} \mathcal{L}_{\text{CE}}(x,y) + \sum_{\mathbf{e}\in\mathbf{E}, y\in\mathcal{Y}^U(\mathbf{e})} w_{\mathbf{e}} \cdot \mathcal{L}_{\text{NL}}(\mathbf{e},y)$$

where $w_{\mathbf{e}} > 0$ is the weight associated with each counterexample. $\mathcal{L}_{\text{CE}}(x,y)$ is the cross-entropy loss computed on the original training set. $\mathcal{L}_{\text{NL}}$ is a custom penalty term computed on the counterexamples:

$$\mathcal{L}_{\text{NL}}(\mathbf{e},y) = -\log\big(1 - p_y(\mathbf{e})\big),$$

where $p_y(\mathbf{e})$ is the probability assigned by the model to unsafe action y for state $\mathbf{e}$. This term discourages selecting unsafe actions by driving their probability towards zero in the counterexamples.

E.4 Detailed Results

We exclude the 7 policies that were safe from the beginning, leaving us with 31 policies.

E.4.1 Empirical Guarantees of the Safety Repair Loop

Ideally, the repair loop uses an exact verifier to identify all existing property violations and provably fixes all violations in the repair step. If these two conditions hold, loop convergence certifies that the repaired model respects the property. We analyze each of the two conditions empirically in relation to our experiment, using Table 9 to report the repair loop outcomes for all 31 benchmarks using the four different repair methods.

Does the loop fix all found counterexamples? Our MILP repair provably fixes all counterexamples (Thm. 2). Therefore the loop either terminates with a guaranteed repair (29 times), or is stopped due to exhausting resources (2 times). One of the stopped instances exhausts resources during counterexample generation (*CE-OOR*) and therefore fails for reasons outside our MILP repair method. The remaining one reaches the 1h time limit for a MILP repair step (*Repair-TO*). PT is also designed to repair all given counterexamples; however, it exhausts resources on 8 benchmarks as adding one penalty tree per counterexample massively increases model size.

Conversely, the retraining-based methods do *not* guarantee fixing all counterexamples. Indeed, the loop terminates in only 22 instances for MH and 28 for FALoss: in the remaining 9 and 3 instances, it reaches the 1000 iteration limit without having fixed all accumulated counterexamples (*MaxIt*).

Does the loop find all counterexamples? Exact verification for multi-step safety is often infeasible in practice as we discuss in Section 4.2. For reference, Goldberger et al. [49] introduced a counterexample-guided repair loop for single-step neural-network properties, and their results already exposed the scalability limitation of placing exact verification inside a repair loop. This limitation is even more pronounced for multi-step safety, where a violation is witnessed by a policy run that eventually reaches failure, but identifying which model decision should be repaired requires looking back at each decision along the unsafe policy run. This issue was addressed by Jain et al. [59], who introduced fault analysis to pinpoint the model decision along an unsafe run that needs to be repaired. Their approach uses fuzzing to discover unsafe runs. Although fuzzing is incomplete, to the best of our knowledge it is the most effective available approach for finding counterexamples in this setting. We therefore adopt it in our safety repair loop.

A natural question is whether fuzzing is able to identify all counterexamples. We analyze this for our main experiment in the *MILP* section of Table 9. The *Converged* column indicates whether our repair loop converged using MILP repair, which happened in 29/31 instances. In all these cases, the loop fixes all identified counterexamples and no new counterexamples are found by fuzzing. We further attempt to verify the repaired policy with an exact verifier [58], and find that (1) *fuzzing sometimes fails in identifying all existing counterexamples*: in 9 cases, exact verification terminates with an unsafe path found on the repaired policy, meaning that further repair is needed,[11] and (2) *exact verification is infeasible in 19/29 instances* where it runs out of time or memory, which demonstrates the need for approximate counterexample generation.

Therefore *exact verification is often computationally infeasible but approximate techniques can fail to find all counterexamples*. More specifically, fuzzing can miss counterexamples for two reasons: first, it is performed only on the sampled initial states $\mathcal{X}_0^R$ because the set of all possible initial states can be exponentially large or infinite, and second, even from a fixed initial state there may be exponentially many policy runs due to nondeterminism, thus fuzzing cannot explore all of them.

E.4.2 Safety and Performance of the Repaired Policies

Despite fuzzing being incomplete, the best we can do is still use it and accept that convergence does not necessarily imply global safety. We therefore use two metrics to evaluate the repaired policy's safety and performance, assessing whether repair improves behavior beyond the specific unsafe decisions that have been fixed. We examine generalization to both (1) unseen executions from the initial states used for counterexample generation and (2) executions from entirely unseen initial states. Accordingly, we evaluate the repaired policies from both the repair initial states $\mathcal{X}_0^R$ and a disjoint set of held-out initial states $\mathcal{X}_0^E$, which are never used for counterexample generation. For *Beluga* and *Blocksworld*, the number of start states is small. Therefore we use the complete set of start states for repair and evaluation.

%Goal For each initial state, we sample 100 policy runs and report %Goal as the fraction of runs reaching the goal.

%Safe From a fixed initial state, we exhaustively enumerate the *policy envelope*, i.e., the set of all states reachable under the policy, considering every possible transition outcome. The policy is safe from this initial state if no reachable state is a failure state. We report %Safe as the fraction of initial states satisfying this condition. This complete enumeration can identify unsafe runs missed by

[11] If verification finds an unsafe path, we can plug it in as an additional counterexample generator for our loop when fuzzing finds nothing. However, in most cases verification will run OOR and therefore will not help.

Table 9: Results of the repair loop and of the subsequent formal verification of the repaired policy. *Converged*: ✓ means the loop converged to a policy that fixes all counterexamples found with fuzzing. Grey marks instances where the loop fails to converge: *CE-OOR* is out of resources during counterexample generation; *Repair-TO* is MILP timeout after 1h; *MaxIt*/*MaxTime* indicates the loop did not converge within 1000 iterations/48h. *Verif.* reports the verification outcome only for instances where the loop converges. ✓ certifies safety, × means that an unsafe path was found, *OOR* means that the verifier ran out of resources.

	MILP		MH		FALoss		PT	
Benchmark	Converged	Verif.	Converged	Verif.	Converged	Verif.	Converged	Verif.
Discrete state space benchmarks								
Beluga (3 Parts)	✓	×	✓	×	✓	×	✓	×
Beluga (4 Parts)	✓	×	✓	*OOR*	✓	*OOR*	*CE-OOR*	–
Beluga (5 Parts)	✓	*OOR*	*MaxIt*	–	*MaxIt*	–	✓	*OOR*
Beluga (6 Parts)	*Repair-TO*	–	*MaxIt*	–	*MaxIt*	–	✓	×
8Blocks CA	✓	×	✓	×	✓	×	✓	×
8Blocks CI	✓	*OOR*	✓	*OOR*	✓	*OOR*	✓	*OOR*
Transport	✓	×	✓	×	✓	×	✓	×
1WL (15, 10) Det	✓	×	*MaxIt*	–	✓	*OOR*	✓	*OOR*
1WL (17, 10) Det	✓	×	✓	×	✓	×	✓	×
1WL (20, 10) Det	✓	*OOR*	✓	*OOR*	✓	*OOR*	✓	*OOR*
1WL (30, 20) Det	✓	*OOR*	*MaxIt*	–	✓	*OOR*	*MaxTime*	–
1WL (15, 10) Icy	✓	×	✓	×	✓	×	✓	×
1WL (17, 10) Icy	✓	*OOR*	✓	*OOR*	✓	×	✓	*OOR*
1WL (15, 10) I+P	✓	*OOR*	✓	×	✓	*OOR*	✓	×
1WL (17, 10) I+P	✓	×	✓	×	✓	*OOR*	✓	*OOR*
1WL (20, 10) I+P	✓	*OOR*	✓	×	✓	×	✓	*OOR*
1WL (30, 20) I+P	✓	*OOR*	*MaxIt*	–	✓	*OOR*	*MaxTime*	–
1WL (40, 25) I+P	✓	*OOR*	✓	*OOR*	✓	*OOR*	✓	*OOR*
2WL (30, 20) Det	✓	*OOR*	*MaxIt*	–	✓	*OOR*	*MaxTime*	–
2WL (40, 25) Det	✓	*OOR*	*MaxIt*	–	✓	*OOR*	*MaxTime*	–
2WL (30, 20) Icy	✓	*OOR*	✓	*OOR*	✓	*OOR*	✓	*OOR*
2WL (15, 10) I+P	✓	×	✓	*OOR*	✓	×	*CE-OOR*	–
2WL (17, 10) I+P	✓	*OOR*	*MaxIt*	–	✓	×	✓	×
2WL (20, 10) I+P	✓	✓	✓	*OOR*	✓	×	✓	×
2WL (30, 20) I+P	✓	*OOR*	✓	*OOR*	✓	*OOR*	*MaxTime*	–
2WL (40, 25) I+P	✓	*OOR*	✓	*OOR*	✓	*OOR*	*MaxTime*	–
Continuous state space benchmarks								
Bounc. Ball	✓	*OOR*	✓	*OOR*	✓	*OOR*	✓	*OOR*
Bounc. Ball ($\tau = 0.01$)	✓	*OOR*	✓	*OOR*	✓	*OOR*	✓	*OOR*
Bounc. Ball ($\tau = 0.05$)	✓	*OOR*	✓	*OOR*	✓	*OOR*	✓	*OOR*
Inv. Pend.	*CE-OOR*	–	✓	*OOR*	✓	*OOR*	✓	*OOR*
Follow Car	✓	*OOR*	*MaxIt*	–	*MaxIt*	–	✓	*OOR*
Converged (of 31)	**29**		**22**		**28**		**23**	
verified safe (✓)		1		0		0		0
unsafe path found (×)		9		8		10		9
inconclusive (OOR)		19		14		18		14

fuzzing, but is too computationally expensive to repeat for counterexample generation throughout the repair loop. We therefore reserve it for evaluating the repaired policies. When enumeration fails, which occurs only for *Blocksworld*, we instead estimate safety using the same sampling procedure as for %Goal. Even with exhaustive enumeration %Safe=100% does not establish global safety, since the evaluated initial states may not cover all possible initial states. Finally, on some benchmarks being safe implies reaching the goal. For these, the reported results for %Goal and %Safe are identical.

Safety and Performance on the Repair States $\mathcal{X}_0^R$. We report per-benchmark results in Table 10 and summarize them visually in Figure 10. All four repair methods substantially improve safety on $\mathcal{X}_0^R$, with MILP, MH and FALoss showing similar safety for the repaired policies. Specifically, mean %Safe increases from 56.4% for the original policies to 91.2% for MILP ($+34.7$ pp), 90.8% for MH ($+34.3$ pp), 87.0% for FALoss ($+30.6$ pp), and 74.4% for PT ($+17.9$ pp). MILP reaches %Safe=100% on 16/31 benchmarks, compared to 16/31 for MH, 19/31 for FALoss, and 8/31 for PT. However, for some repaired policies %Safe remains well below 100%, indicating that fuzzing has

missed unsafe executions even from the initial states. %Goal is also slightly improved on average by all methods, indirectly as unsafety can be a reason for not reaching the goal.

Table 10: Goal-reaching performance (%Goal) and safety (%Safe) on the *repair* initial states $\mathcal{X}_0^R$ for the original policies and policies returned by the repair loop using MILP, the two retraining-based methods (MH and FALoss), and Penalty Trees (PT). For benchmarks where the loop does not converge, we report the performance of the last available policy. In some PT instances, the repaired policy becomes too large to evaluate computationally; in these cases, we report the original policy.

Benchmark	*%Safe*					*%Goal*				
	Original	MILP	MH	FALoss	PT	Original	MILP	MH	FALoss	PT
Discrete state space benchmarks										
Beluga (3 Parts)	81.33	82.33	80.33	80.50	80.50	81.33	82.33	80.33	80.50	80.50
Beluga (4 Parts)	84.75	94.12	93.00	92.21	91.12	84.75	94.12	93.00	92.21	91.12
Beluga (5 Parts)	69.08	95.84	76.66	24.42	93.51	69.08	95.84	76.66	24.42	93.51
Beluga (6 Parts)	86.01	80.17	80.73	12.25	98.01	86.01	80.17	80.73	12.25	98.01
8Blocks CA	99.93	100.00	100.00	99.99	99.98	92.03	86.60	92.10	92.08	92.11
8Blocks CI	99.93	100.00	100.00	100.00	100.00	92.01	90.86	92.10	92.06	92.06
Transport	98.76	100.00	100.00	100.00	100.00	0.09	0.09	0.09	0.09	0.09
1WL (15, 10) Det	0.01	100.00	100.00	100.00	100.00	0.01	100.00	100.00	100.00	100.00
1WL (17, 10) Det	99.37	100.00	100.00	100.00	100.00	99.37	100.00	100.00	100.00	100.00
1WL (20, 10) Det	0.01	100.00	100.00	100.00	100.00	0.01	100.00	100.00	100.00	100.00
1WL (30, 20) Det	0.01	100.00	100.00	100.00	91.29	0.01	100.00	100.00	100.00	91.29
1WL (15, 10) Icy	0.50	0.50	0.50	0.50	0.50	90.48	92.06	92.21	92.15	91.76
1WL (17, 10) Icy	0.01	0.01	0.01	0.01	0.01	85.90	92.23	91.35	92.19	92.23
1WL (15, 10) I+P	0.50	99.99	99.81	99.99	86.86	92.21	92.17	91.87	92.17	92.19
1WL (17, 10) I+P	0.01	98.42	99.81	100.00	76.36	91.50	91.52	91.51	91.52	91.55
1WL (20, 10) I+P	0.11	86.98	99.73	88.46	4.64	88.32	88.25	86.07	88.15	88.27
1WL (30, 20) I+P	0.06	99.95	99.79	99.99	6.86	62.41	62.19	55.36	73.98	79.41
1WL (40, 25) I+P	84.43	97.59	99.99	99.79	85.88	61.93	61.90	60.18	72.16	61.88
2WL (30, 20) Det	7.07	100.00	99.91	100.00	7.07	0.01	0.03	0.00	0.00	0.00
2WL (40, 25) Det	14.96	100.00	100.00	100.00	14.96	0.01	0.01	0.01	0.01	0.01
2WL (30, 20) Icy	99.95	100.00	100.00	100.00	99.98	2.04	2.10	1.92	3.18	2.06
2WL (15, 10) I+P	78.31	99.96	100.00	100.00	78.31	0.01	0.01	0.01	0.01	0.01
2WL (17, 10) I+P	77.98	100.00	99.05	100.00	100.00	0.00	0.01	0.00	0.01	0.00
2WL (20, 10) I+P	78.70	100.00	100.00	100.00	100.00	0.00	0.01	0.01	0.01	0.01
2WL (30, 20) I+P	64.08	95.46	84.10	100.00	64.08	0.00	0.00	0.00	0.00	0.00
2WL (40, 25) I+P	56.33	99.97	100.00	100.00	56.33	0.01	0.00	0.00	0.00	0.00
Continuous state space benchmarks										
Bounc. Ball	99.95	100.00	100.00	100.00	100.00	99.95	100.00	100.00	100.00	100.00
Bounc. Ball ($\tau = 0.01$)	94.69	100.00	100.00	100.00	95.43	94.69	100.00	100.00	100.00	95.43
Bounc. Ball ($\tau = 0.05$)	97.08	100.00	100.00	100.00	97.94	97.08	100.00	100.00	100.00	97.94
Inv. Pend.	97.55	95.00	100.00	100.00	97.56	97.55	95.00	100.00	100.00	97.56
Follow Car	77.76	100.00	99.99	99.97	77.76	77.76	100.00	99.99	99.97	77.76
Average	**56.43**	**91.17**	**90.76**	**87.03**	**74.35**	**53.11**	**64.76**	**64.05**	**61.26**	**64.73**

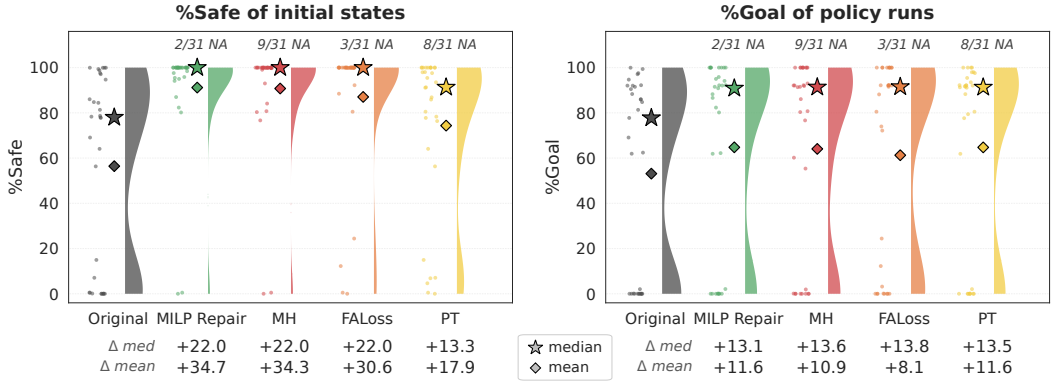

Figure 10: %Safe (left) and %Goal (right) before and after repair on the repair initial states $\mathcal{X}_0^R$, showing per-benchmark values (dots) and the kernel density estimate (violin). ★: median, ♢: mean; Δ reports the change from the original to the repaired model. $k/31$ *NA* above a column: k of 31 benchmarks failed to converge. In these cases, we evaluate the last policy generated from the repair loop.

Safety and Performance on the Evaluation States $\mathcal{X}_0^E$. Switching to the held-out evaluation start states (Table 11 and Figure 2 in Section 4.2) reveals a substantial difference between the repair methods. While on $\mathcal{X}_0^R$ MILP and MH achieve almost identical safety improvements with FALoss close behind, on $\mathcal{X}_0^E$ MILP is clearly the method yielding the safer policies. Specifically, the %Safe improvement from MILP remains essentially unchanged from $\mathcal{X}_0^R$ at $+35.1$ pp, whereas it drops to $+26.5$ pp for MH and $+20.3$ pp for FALoss. A similar effect appears for goal-reaching performance.

Table 11: Goal-reaching performance (%Goal) and safety (%Safe) on the *evaluation* initial states $\mathcal{X}_0^E$ for the original policies and policies returned by the repair loop using MILP, the two retraining-based methods (MH and FALoss), and Penalty Trees (PT). For benchmarks where the loop does not converge, we report the performance of the last available policy. In some PT instances, the repaired policy becomes too large to evaluate computationally; in these cases, we report the original policy.

Benchmark	*%Safe*					*%Goal*				
	Original	MILP	MH	FALoss	PT	Original	MILP	MH	FALoss	PT
Discrete state space benchmarks										
Beluga (3 Parts)	81.33	82.33	80.33	80.50	80.50	81.33	82.33	80.33	80.50	80.50
Beluga (4 Parts)	84.75	94.12	93.00	92.21	91.12	84.75	94.12	93.00	92.21	91.12
Beluga (5 Parts)	69.08	95.84	76.66	24.42	93.51	69.08	95.84	76.66	24.42	93.51
Beluga (6 Parts)	86.01	80.17	80.73	12.25	98.01	86.01	80.17	80.73	12.25	98.01
8Blocks CA	99.93	100.00	100.00	99.99	99.98	92.03	86.60	92.10	92.08	92.11
8Blocks CI	99.93	100.00	100.00	100.00	100.00	92.01	90.86	92.10	92.06	92.06
Transport	97.70	98.08	98.39	97.84	97.70	0.01	0.01	0.01	0.01	0.01
1WL (15, 10) Det	0.00	98.98	61.98	69.02	24.53	0.00	98.98	61.98	69.02	24.53
1WL (17, 10) Det	98.86	99.64	96.84	98.98	99.03	98.86	99.64	96.84	98.98	99.03
1WL (20, 10) Det	0.00	98.53	99.38	98.30	73.98	0.00	98.53	99.38	98.30	73.98
1WL (30, 20) Det	0.00	96.89	31.07	35.03	9.03	0.00	96.89	31.07	35.03	9.03
1WL (15, 10) Icy	0.40	0.40	0.34	0.40	0.40	88.16	91.30	91.23	91.52	90.36
1WL (17, 10) Icy	0.00	0.00	0.00	0.00	0.00	87.17	91.67	90.00	91.63	91.67
1WL (15, 10) I+P	0.40	98.94	98.14	98.94	82.13	91.61	91.65	91.13	91.65	91.60
1WL (17, 10) I+P	0.00	96.96	99.46	100.00	67.88	90.38	90.33	90.36	90.45	90.31
1WL (20, 10) I+P	0.10	80.54	93.32	66.96	5.67	87.52	87.54	83.55	87.31	87.49
1WL (30, 20) I+P	0.07	85.19	97.09	91.74	0.62	47.79	47.42	36.78	66.06	47.38
1WL (40, 25) I+P	55.28	62.80	45.66	82.99	55.28	46.85	46.87	35.18	65.14	46.89
2WL (30, 20) Det	19.08	95.06	99.98	49.65	19.08	0.00	0.02	0.00	0.00	0.00
2WL (40, 25) Det	16.94	49.94	98.89	36.71	16.94	0.00	0.00	0.00	0.00	0.00
2WL (30, 20) Icy	99.87	100.00	100.00	100.00	99.94	4.51	4.56	2.90	6.61	4.53
2WL (15, 10) I+P	58.84	98.14	65.77	65.77	58.84	0.00	0.00	0.00	0.00	0.00
2WL (17, 10) I+P	59.55	99.50	58.97	65.45	59.55	0.00	0.00	0.00	0.00	0.00
2WL (20, 10) I+P	60.14	100.00	66.24	66.24	60.14	0.00	0.00	0.00	0.00	0.00
2WL (30, 20) I+P	29.58	81.40	33.38	49.25	29.58	0.00	0.00	0.00	0.00	0.00
2WL (40, 25) I+P	13.67	98.02	44.90	46.09	13.67	0.00	0.00	0.00	0.00	0.00
Continuous state space benchmarks										
Bounc. Ball	99.92	99.99	100.00	100.00	99.92	99.92	99.99	100.00	100.00	99.92
Bounc. Ball ($\tau = 0.01$)	95.06	100.00	100.00	100.00	95.06	95.06	100.00	100.00	100.00	95.06
Bounc. Ball ($\tau = 0.05$)	97.21	100.00	100.00	100.00	97.21	97.21	100.00	100.00	100.00	97.21
Inv. Pend.	97.08	94.64	100.00	100.00	97.08	97.08	94.64	100.00	100.00	97.08
Follow Car	77.61	99.99	99.98	99.83	77.61	77.61	99.99	99.98	99.83	77.61
Average	**51.56**	**86.65**	**78.08**	**71.89**	**61.42**	**52.10**	**63.55**	**58.88**	**57.58**	**57.13**

Therefore *policies repaired with our MILP are safer than those repaired with the baselines*, which suggests that fixes produced through MILP repair generalize better to initial states that are not seen during counterexample generation.

E.4.3 Computational Cost of the Repair Loop

MILP takes the fewest iterations to converge Table 12 reports the number of accumulated counterexamples and repair iterations of the repair loop using the four different repair methods. Among the 29 benchmarks where the MILP-based loop terminates, 26 require fewer than 10 iterations and none requires more than 25. In contrast, retraining may require more iterations to eliminate the same unsafe choices, and sometimes fails in doing so (i.e., it is stopped after 1000 iterations). Specifically, MILP needs fewer iterations than MH in 19/29 cases, than FALoss in 18/29, and than PT in 17/29. In most of the remaining cases, the methods require the same number of iterations. For example, in *1WL (30, 20) Det*, MILP repair converges in 7 iterations compared to 28 for FALoss, while MH reaches the 1000-iteration limit.

Repair takes less time than counterexample generation Table 13 separates the total loop runtime into counterexample generation and repair runtimes. Of the 29 completed MILP runs, 19 complete the entire loop within 10 minutes. Moreover, usually most of the runtime is spent searching for counterexamples and not repairing the model: the median MILP share of the loop runtime is 9.2% and the average is 31.5%. The latter is inflated by the few harder instances: MILP runtime exceeds counterexample generation runtime on only 8 benchmarks.

Therefore converging in fewer iterations (which MILP does) is crucial: a single retraining iteration is often cheaper than solving the MILP, but retraining can require many more loop iterations to converge. Finally, PT takes long because penalty trees make the model size explode, which massively increases the complexity of the counterexample generation step (e.g., on *1WL (30,20) I+P*, PT spends only 35s on repair but over 37h on counterexample generation).

Table 12: Statistics for running the entire repair loop. $|E|$ is the total number of counterexamples found during the repair loop. We also report the number of loop iterations.

Benchmark	$\|E\|$				#Iterations			
	MILP	MH	FALoss	PT	MILP	MH	FALoss	PT
Discrete state space benchmarks								
Beluga (3 Parts)	4	10	6	4	4	9	13	4
Beluga (4 Parts)	32	89	57	17	19	56	48	9
Beluga (5 Parts)	246	1926	3067	108	23	1000	1000	12
Beluga (6 Parts)	1763	9148	25602	660	25	1000	1000	38
8Blocks CA	2	7	6	3	2	5	10	3
8Blocks CI	1	1	1	1	2	3	4	2
Transport	20	19	28	28	3	2	3	4
1WL (15, 10) Det	11246	24854	11236	13067	7	1000	17	6
1WL (17, 10) Det	30	98	30	30	2	4	2	2
1WL (20, 10) Det	7793	3475	4276	3921	7	14	13	5
1WL (30, 20) Det	16476	63492	16854	29974	7	1000	28	8
1WL (15, 10) Icy	23	39	23	27	2	3	2	3
1WL (17, 10) Icy	8	8	8	8	2	2	2	2
1WL (15, 10) I+P	27	27	27	37	2	2	4	3
1WL (17, 10) I+P	49	48	48	76	3	3	3	5
1WL (20, 10) I+P	92	124	231	132	2	6	9	4
1WL (30, 20) I+P	3311	38332	2457	8666	17	1000	30	12
1WL (40, 25) I+P	195	187	389	353	3	2	5	9
2WL (30, 20) Det	15195	57822	21156	38933	9	1000	9	12
2WL (40, 25) Det	11633	42655	17277	28594	8	1000	6	12
2WL (30, 20) Icy	3	3	11	4	2	2	4	3
2WL (15, 10) I+P	6911	19338	5782	13017	5	8	4	16
2WL (17, 10) I+P	13826	87641	6605	6509	7	1000	4	4
2WL (20, 10) I+P	4259	4259	12105	6388	3	3	6	4
2WL (30, 20) I+P	8347	5063	5795	16863	5	4	22	22
2WL (40, 25) I+P	1269	988	1028	12795	7	2	5	30
Continuous state space benchmarks								
Bounc. Ball	5	5	5	5	2	2	2	2
Bounc. Ball ($\tau = 0.01$)	531	531	613	588	2	3	3	3
Bounc. Ball ($\tau = 0.05$)	292	292	292	292	2	4	6	2
Inv. Pend.	245	268	859	340	2	19	11	7
Follow Car	2224	3182	3851	2230	2	1000	1000	7

Therefore (1) MILP repair is preferable with respect to the baselines because it converges in fewer iterations and (2) future work should focus on improving verification and counterexample generation techniques, which usually represent the bottleneck of the repair loop.

F Extended Related Work

There are two closely related areas of work: approaches for repairing models and algorithms that consider a specific property at training time. For repair approaches that are property-agnostic, the

Table 13: Total repair loop runtime, split into counterexample generation (CE) and repair runtimes. – indicates runs that exceeded the maximum time limit of 48 hours.

Benchmark	CE time				Repair time				Total time			
	MILP	MH	FALoss	PT	MILP	MH	FALoss	PT	MILP	MH	FALoss	PT
Discrete state space benchmarks												
Beluga (3 Parts)	<1 s	2 s	3 s	1 s	2 s	2 s	3 s	<1 s	2 s	4 s	7 s	1 s
Beluga (4 Parts)	7 s	23 s	18 s	23 s	1.8 m	22 s	21 s	<1 s	1.9 m	45 s	39 s	23 s
Beluga (5 Parts)	32 s	29 m	49 m	35 s	34 m	9.8 m	13 m	<1 s	35 m	39 m	1.0 h	35 s
Beluga (6 Parts)	8.6 m	11 h	19 h	24 m	25 h	23 m	40 m	17 s	25 h	12 h	20 h	25 m
8Blocks CA	32 s	2.2 m	6.7 m	1.4 m	2 s	2 s	6 s	<1 s	34 s	2.3 m	6.8 m	1.4 m
8Blocks CI	31 s	46 s	1.1 m	35 s	2 s	<1 s	1 s	<1 s	33 s	46 s	1.1 m	35 s
Transport	42 s	19 s	22 s	45 s	2 s	<1 s	<1 s	<1 s	44 s	19 s	22 s	45 s
1WL (15, 10) Det	11 m	3.4 h	15 m	3.1 h	1.4 m	6.6 m	6 s	6 s	13 m	3.5 h	16 m	3.1 h
1WL (17, 10) Det	38 s	56 s	32 s	27 s	3 s	1 s	<1 s	<1 s	41 s	58 s	32 s	27 s
1WL (20, 10) Det	11 m	15 m	28 m	1.1 h	12 m	5 s	5 s	3 s	23 m	16 m	28 m	1.1 h
1WL (30, 20) Det	1.1 h	19 h	5.6 h	35 h	17 m	28 m	31 s	29 s	1.4 h	20 h	5.6 h	—
1WL (15, 10) Icy	5.3 m	8.2 m	5.3 m	7.8 m	3 s	1 s	<1 s	<1 s	5.3 m	8.2 m	5.3 m	7.8 m
1WL (17, 10) Icy	10 m	10 m	13 m	12 m	3 s	<1 s	<1 s	<1 s	10 m	10 m	13 m	12 m
1WL (15, 10) I+P	45 s	1.2 m	1.8 m	1.4 m	3 s	<1 s	2 s	<1 s	48 s	1.2 m	1.9 m	1.4 m
1WL (17, 10) I+P	1.2 m	1.6 m	1.5 m	2.0 m	6 s	1 s	1 s	<1 s	1.3 m	1.6 m	1.5 m	2.0 m
1WL (20, 10) I+P	1.3 m	2.2 m	2.9 m	1.8 m	3 s	1 s	4 s	<1 s	1.3 m	2.2 m	3.0 m	1.8 m
1WL (30, 20) I+P	1.1 h	20 h	56 m	37 h	1.1 h	20 m	28 s	35 s	2.1 h	20 h	57 m	—
1WL (40, 25) I+P	9.9 m	8.3 m	41 m	49 m	16 s	1 s	9 s	1 s	10 m	8.3 m	41 m	49 m
2WL (30, 20) Det	3.5 m	4.7 h	5.4 m	5.7 h	1.1 h	20 m	5 s	2.0 m	1.1 h	5.0 h	5.4 m	—
2WL (40, 25) Det	5.0 m	3.2 h	8.8 m	14 h	3.0 h	9.9 m	2 s	2.2 m	3.0 h	3.4 h	8.9 m	—
2WL (30, 20) Icy	53 s	52 s	1.9 m	1.7 m	3 s	<1 s	2 s	<1 s	56 s	52 s	1.9 m	1.7 m
2WL (15, 10) I+P	1.1 m	3.6 m	1.4 m	1.5 h	37 s	2 s	<1 s	5 s	1.7 m	3.6 m	1.4 m	1.5 h
2WL (17, 10) I+P	3.2 m	5.0 h	1.4 m	1.3 m	2.8 m	25 m	<1 s	1 s	5.9 m	5.4 h	1.4 m	1.3 m
2WL (20, 10) I+P	1.5 m	1.7 m	2.9 m	2.6 m	1.0 h	<1 s	1 s	3 s	1.0 h	1.7 m	3.0 m	2.7 m
2WL (30, 20) I+P	3.9 m	3.2 m	21 m	15 h	43 s	1 s	12 s	2.0 m	4.6 m	3.2 m	21 m	—
2WL (40, 25) I+P	3.0 m	1.2 m	2.7 m	19 h	32 s	<1 s	1 s	2.1 m	3.5 m	1.2 m	2.7 m	—
Continuous state space benchmarks												
Bounc. Ball	1.1 m	1.1 m	1.2 m	1.1 m	6 s	4 s	6 s	<1 s	1.2 m	1.2 m	1.3 m	1.1 m
Bounc. Ball ($\tau = 0.01$)	2.3 m	3.1 m	3.2 m	7.8 m	6 s	8 s	12 s	<1 s	2.4 m	3.2 m	3.4 m	7.8 m
Bounc. Ball ($\tau = 0.05$)	58 s	2.5 m	3.0 m	1.4 m	6 s	12 s	29 s	<1 s	1.1 m	2.7 m	3.5 m	1.4 m
Inv. Pend.	40 m	30 m	39 m	2.1 h	25 s	1.6 m	1.2 m	<1 s	40 m	32 m	40 m	2.1 h
Follow Car	4.4 m	9.4 h	9.5 h	1.2 h	8 s	42 m	55 m	<1 s	4.6 m	10 h	10 h	1.2 h

vast majority of the work has focused on (deep) neural networks [49, 80, 83, 86, 36, 11, 42, 79, 14]. Within this literature, the approach of Goldberger et al. [49] is the most similar to ours. It uses an LP formulation to modify the weights in the last layer of the network to ensure that it makes the correct prediction for a set of counterexamples. Beyond tackling a different model class, there are several important differences with our work. First and foremost, we also incorporate a term in our objective to encourage repairs that minimize the decrease in predictive performance. Second, we can exploit properties of the tree structure to simplify our encoding. This is based on the fact that the output for a counterexample is purely determined by the set of leaves in its OC. Therefore we only need to optimize a subset of the leaves to repair the model, i.e., leaves that appear in the OC of at least one counterexample.

The only existing property agnostic repair method for tree-based models is PyXAI [6]. Specifically, their rectification operation modifies the trees such that the model outputs a class specified by the user for a given input. This is insufficient for properties such as fairness and safety, which do not specify a target output per counterexample but only require that outputs of similar inputs agree (fairness) or that the output lies in a set of admissible values (safety). Moreover, PyXAI does not support multiclass ensembles.

In addition to generic repair approaches, there are algorithms that are tailored towards addressing a specific property. In the context of fairness, one typically distinguishes between *group fairness*, which requires that predictions do not systematically differ across groups defined by a protected attribute, and *individual fairness* [38], which requires that similar individuals must receive similar outcomes, where similarity is defined by a task-specific metric that ignores the protected attribute. We define fairness as the (lack of) causal discrimination [18, 44], which is a specific instantiation of individual fairness where any two inputs identical in all non-protected features must receive the same label. For group fairness, there has been work about post-hoc repair of (ensembles of) decision trees through greedy leaf relabeling [1], structural optimization [95] or pruning [9]. No existing works specifically repair tree ensembles for individual fairness. Related works exist for neural network repair and typically rely on weight modification [67, 43, 40].

It is also possible to incorporate property-specific requirements directly in the learning procedure. Recent works have proposed methods to train individually fair neural networks [77, 96]. In terms of tree ensembles, group fairness-aware training algorithms have been proposed by using fair splitting criteria [61, 73], constrained optimization [27] or adversarial gradient boosting [51]. Only two methods exist to train individually fair ensembles of decision trees: FATT [74] and BuDRO [88]. FATT uses a genetic algorithm guided by abstract interpretation to balance accuracy and fairness at training time. BuDRO trains a GBDT ensemble using a distributionally robust loss. At each boosting round, it solves an optimal transport problem that adversarially maximizes the loss by reweighting the training points within neighborhoods defined by a fair metric. This penalizes individual fairness violations over the training examples, but does not consider regions of the input space that are not represented by the training data. We use these two methods as baselines in our experiment. These algorithms simply use a fairness-aware objective at training time and thus cannot provide a hard fairness guarantee. In contrast, a valid solution to our MILP repair loop generates a model that is provably fair on the whole input space. A similar approach is that of Aghaei et al. [3], who employ a MILP formulation for training a fair decision tree; however, their approach operates over the training data only and does not guarantee fairness over unseen inputs.

There are two families of algorithms addressing safety in sequential decision making problems: shielding and safe reinforcement learning. Shielding [5, 65, 60, 93, 26, 54] synthesizes a component which masks unsafe actions at runtime, such that any policy composed with the shield is guaranteed to be safe. To scale the shields to massive state spaces, symbolic representations such as BDDs [65, 20] have been used to represent these shields. The second approach is safe reinforcement learning which instead modifies the training procedure so the learned models avoid unsafe behavior. Prominent algorithms here are constrained policy optimization [2] and PPO-Lagrangian [75] which extend standard RL algorithms like PPO [78] by penalizing unsafe behavior during training. No existing works specifically tackle obtaining safe tree ensembles.

Finally, existing work [76, 17, 31] has looked at using optimization to change the leaf values of a tree ensemble. Ren et al. [76] showed that a pretrained random forest can be finetuned by further optimizing all the leaf values simultaneously. Buschjäger and Morik [17] reduce model size by combining leaf refinement with L1-regularized ensemble pruning, which removes entire trees. Devos et al. [31] compress an ensemble level by level, pruning subtrees at any depth while refining the remaining leaf values, subject to a bound on the loss in validation accuracy. Inspired by these results, we formulate tree ensemble repair as a leaf refinement task and add hard constraints to fix the counterexamples.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction claim that we can enforce user-defined properties on tree ensembles by performing post-hoc repair through a MILP encoding. We also state that this outperforms property-aware training baselines. Section 3 explains how we encode tree ensemble repair as a MILP, and provides formal soundness and termination guarantees (proofs in Appendices A-B-C). Section 4.1 and Section 4.2 support these claims by empirically showing that our method repairs models to enforce fairness and safety, and yields stronger results than the baselines.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss two main limitations. (1) MILP runtime: while typically fast, the MILP can be expensive on harder instances, occasionally hitting our predefined timeout. This is discussed in the *Repair time* paragraphs of both fairness (Section 4.1) and safety (Section 4.2) experiments, with detailed runtimes in Appendices D and E respectively. (2) Verification/counterexample generation: the overall scalability of the repair loop is bounded by the verifier's capability to extract all counterexamples. In the fairness experiment, we compress XGBoost ensembles with LOP to enable exact verification with Veritas (which could be infeasibly slow on a large ensemble). In the safety experiment, exact verification is usually infeasible, so we rely on (approximate) policy testing methods to find counterexamples. We summarize these limitations in the experimental setup and in Section 6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.

- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: The paper contains four theoretical results: Thm. 1 (a repair formula for a given counterexample always exists), Thm. 2 (any feasible MILP solution yields an ensemble satisfying the property on all given counterexamples), Thm. 3 (the IIS-driven tree-augmentation procedure recovers feasibility in finitely many steps), and Thm. 4 (the iterative repair loop terminates under finiteness assumptions, with a stronger variant for fairness). All theorems state their assumptions explicitly, and complete proofs are given in Appendices A-B-C.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification:

   Our MILP encoding is reported in Sections 3.1 and 3.2, and the overall repair loop is outlined in Section 3.4 (full algorithm in Appendix B). The *Experiments Setup* and *Baselines* paragraphs in Section 4.1 and Section 4.2 provide all information needed to reproduce our fairness and safety experiments, pointing to Appendix sections where needed. For fairness, the datasets (standard from the fairness literature) are listed in Table 2; Section 4.1 reports the XGBoost and LOP hyperparameters, the employed verifier (Veritas), and the two baselines with the adopted configurations (FATT and BuDRO). Additional details and pointers to the baselines' source codes are provided in Appendix D.1. For safety, the benchmark set is described in Appendix E.2; Section 4.2 outlines the imitation learning procedure, XGBoost hyperparameters and counterexample generation methods. Baseline details are provided in Appendix E.3. The MILP solver (Gurobi 12.0) and regularization hyperparameter λ are reported at the beginning of Section 4.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.

- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: At the beginning of Section 4 we share a link to the codebase that reproduces all our experiments.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification:

Training and test details are specified for both experiments. For fairness (Section 4.1), we report the number of folds for cross validation, the XGBoost hyperparameters, the LOP compression settings, and the FATT/BuDRO baselines configurations. For safety (Section 4.2), we report the XGBoost hyperparameters, the imitation learning setup, and the use of disjoint repair and evaluation start-state sets to assess safety generalization.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: For fairness, results are aggregated over 5-fold cross-validation; Table 4 and Table 5 in Appendix D report mean and standard deviation across folds for balanced accuracy and runtime per dataset. Box plots in Figure 1 show the distribution of balanced accuracy loss across all unfair models. For safety, cross-validation is not applicable as each benchmark is a distinct MDP with a single neural policy. Therefore Figure 2 reports per-benchmark values, kernel density estimate and mean/median across all benchmark instances; per-benchmark numbers are reported in detail in Table 11 in Appendix E.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification:

The compute worker characteristics and memory are specified in Section 4.1 for the fairness experiment and in Section 4.2 for the safety experiment.

Runtimes to execute our repair loop are discussed in the *Repair time* paragraphs of both Section 4.1 and Section 4.2, with detailed tables in Appendices D-E.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We are aware of the code of ethics and do not violate any clause.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We included an impact statement in Section 6.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.

- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: -

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Only the authors contributed to this paper. Adopted third-party methods for learning and compressing tree ensembles and generating counterexamples are properly referenced in Section 4, with details in the supplementary material. All datasets are referenced in Table 2 (fairness) and Appendix E.2 (safety). The license has been selected for submission in OpenReview.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

Justification: The code implementing our new method (described in Section 3) as well as all the presented experiments (Section 4) is publicly distributed and referenced in Section 4.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: -

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: -

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: -

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.