FATE

Fuzzing for Adversarial examples in Tree Ensembles

Cas Bilstra

TU Delft

FATE
Fuzzing for Adversarial examples in Tree Ensembles

by

Cas Bilstra

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday July 12, 2021 at 10:00 AM.

Student number:	4381084	
Project duration:	November 9, 2021 – July 12, 2021	
Thesis committee:	Dr. ir. S. E. Verwer,	TU Delft, supervisor
	Dr. C. B. Poulsen,	TU Delft
	D. A. Vos MSc,	TU Delft

TUDelft

Abstract

Machine learning models are increasing in popularity and are nowadays used in a wide range of critical applications in fields such as Automotive, Aviation and Medical. Among machine learning models, tree ensemble models are a popular choice due to their competitive performance and high degree of explainability. Like most machine learning models they however suffer from adversarial examples: slightly perturbed input for which the model makes an unexpected prediction. These can be seen as bugs in the model and in critical applications such a bug may have high impact. We investigate if fuzzers, a popular and effective tool for identifying bugs in software, can be used for finding bugs (adversarial examples) in tree ensemble models as well.

We introduce FATE, a tool based on grey-box fuzzers that is able to find adversarial examples on a multitude of datasets. Using a custom mutator that leverages domain information as well as model-specific information such as splitting thresholds and dataset-specific information such as training samples, FATE is able to find good adversarial examples: for non-image classification models they are within 1 percent-point difference from examples generated by the state-of-the-art (Zhang et al., [48]). However, the coverage-guidance of grey-box fuzzers actually limits the performance of FATE: running the mutator of FATE as a (1+1) Evolutionary Algorithm makes FATE show competitive performance to the state-of-the-art, even outperforming it on some datasets.

Preface

This MSc Thesis describes work I have conducted at the Cyber Analytics lab at the TU Delft. Machine Learning models are increasing in popularity and are nowadays being deployed in safety-critical applications such as automotive and aviation. The safety and security aspects of these models are however easily overlooked. I am proud to have developed a novel method with which the robustness of machine learning models can be evaluated.

Graduating during the Covid-19 pandemic was an extra challenge. Studying sessions that we were able to organize towards the end of the lockdown, show the value of being able to discuss your project and have some fun with other students. Nevertheless I learned a lot and enjoyed the project. I would like to thank my supervisor Sicco Verwer for his enthusiasm, critical attitude and creative ideas. Furthermore, I would like to thank Daniël Vos for his ideas, conversations and help in understanding the field of adversarial machine learning for tree ensemble models. Thanks to Max and Dennis for their reviews of this thesis. Finally, I would like to thank my family for their endless support and my friends for providing some essential distractions from this project.

Cas Bilstra
Delft, July 2021

Contents

1 Introduction

Machine learning (ML) has shown wide adoption in both research and industrial fields. It has shown to be competent at complex tasks such as image recognition and is used in critical applications such as Automotive, Aviation and Medical [23, 49]. Among the many kinds of machine learning algorithms, tree ensembles are a popular choice for their competitive performance and high degree of explainability [24]. Although machine learning models often have high accuracy on their training- and test data, they have shown to be very susceptible to noise in the input. This is not only the case for tree ensembles [11, 24, 48], but also for other machine learning algorithms, such as the widely used Neural Network [38]. Through exploiting this susceptibility for noise, an adversary can create an Adversarial Example (AE): by adding a small amount of forged noise to benign input, it can make the model predict an unexpected class. An example of such an adversarial example for a model trained to recognise digits (0 to 9) can be seen in Figure 1.1. On the first look both images look similar. However when looking closer, small noise can be identified in the left image, making the model predict the left image as an 8 instead of a 1. This can have potentially disastrous consequences when used in critical applications. For example, researchers were able to make the Tesla autopilot switch lanes through stickers on the road [1], or to trick a Tesla into speeding much above the speed limit [34]. Other use cases for adversarial examples may for example be detection evasion in malware classifiers or ML-based Intrusion Detection Systems. It is an active field in research how to create machine learning models that more robust against these adversarial examples. In order to train and evaluate the robustness of machine learning models, fast methods are necessary to create adversarial examples.

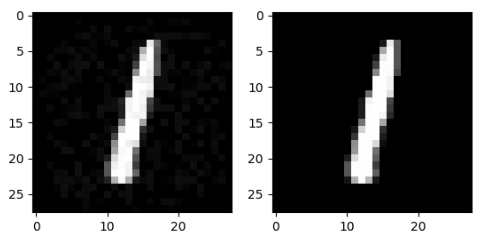


Figure 1.1: An adversarial example. Left: 8 (zoom in to see the noise). Right (original): 1

The presence of a good adversarial example can be seen as a bug in the model: for a certain input, the model produces unexpected output. Why not use one of the most effective tools for finding bugs in software to find adversarial examples: a fuzzer. Starting from a corpus of initial seeds (inputs), fuzzers feed big amounts of randomly or evolutionary generated inputs to a target program. By monitoring which path through the source code or binary is followed, the fuzzer determines if the input reached previously unreached states in the target application. If so, the input is deemed "interesting" and saved. By combining and mutating previously saved seeds it tries to penetrate deep into the target application, with the goal of reaching as many program states as possible to discover potential bugs. Fuzzers may use deep inspection of the target or its execution to guide the mutation of input, by for example monitoring values used in compare instructions or by performing symbolic execution. Many fuzzers also allow the specification of a "custom mutator" through which users get complete control over the mutation; allowing to incorporate domain knowledge into the mutator. This way, many invalid inputs can be avoided (such as inputs requiring checksum fields) and inputs can be created that are more likely to reach new states in the target.

Because of the high degree of efficiency with which the fuzzer can mutate inputs and execute the target program, it can explore inputs and find paths that would be impractical to test using a traditional test suite.

Fuzzers have shown to be very competent in finding bugs in code. For example, Googles "OSS-Fuzz" project [19], which uses different kinds of fuzzers to search for bugs, has found over 25,000 bugs in 375 open source projects from 2016 till January 2021. The famous "Heartbleed" bug in OpenSSL can for example be found by a fuzzer within seconds. Apart from finding bugs, fuzzers have shown to be applicable to other problems as well such as testing Neural Networks [32, 43] and attacking web-services [40].

1.1. Contributions and objectives

The main goal of this research is to reduce the problem of finding adversarial examples to a fuzzing problem. We propose FATE: a tool to translate tree ensemble models such as Gradient Boosting and Random Forests to (C++) code which can then be fuzzed by modern Fuzzing engines such as libFuzzer [29] and AFL++ [16, 46]. We show how FATE efficiently finds good adversarial examples on multiple datasets used in related literature [2, 11, 48]. We improve the performance of FATE by exploiting information contained in the tree ensemble models, training set and through information gathered during execution: for example, we use the splitting thresholds from the decision tree nodes to mutate right on the decision boundaries of single trees, initialize the fuzzer with forged samples using the training set and we bias the (magnitude of) features which should be mutated based on runtime information. We show that the performance of FATE is close to the performance of the current state-of-the-art adversarial example generation method (Zhang et al., [48]) for tree ensembles. However, we show that the performance of FATE depends on its custom mutator instead of the fuzzing architecture itself. The performance of FATE when executing it as a standalone evolutionary algorithm is better than executing it inside a fuzzer and can produce adversarial examples that improve upon the current state-of-the-art. Finally, we show that fuzzers are able to find adversarial examples using their built-in mutators as well using libFuzzer, honggfuzz ([21]) and AFL++. This thesis is divided into the following research questions:

RQ 1 *How can Fuzzers be used to generate adversarial examples for Tree Ensembles?* To use the strength of fuzzers to find bugs in software, a reduction from machine learning model to source code should be developed that can be fed to fuzzing engines such that they can identify adversarial examples.

RQ 2 *How can Fuzzing for adversarial examples in Tree Ensembles be improved by leveraging information sources?* Fuzzing can be sped up considerably by writing a custom mutator and supplying it with good initial input [25]. We achieve this through leveraging information from various sources and show how this impacts the performance of FATE.

RQ 3 *How does FATE perform with different fuzzing engines?* FATE is developed around libFuzzer, but in this research question we evaluate how FATE performs with different fuzzing engines.

The main contributions of this thesis towards the field are the following:

- A public implementation[1] of FATE.
- We show that adversarial examples within 1 percent-point difference from the state-of-the-art can be found by FATE using fuzzers, however this is mainly due to the smart mutator in FATE and not due to the added benefits of using grey-box fuzzers, such as their coverage-guidance mechanism.
- We provide evidence that running the mutator of FATE in a black-box setting as a standalone (1+1) Evolutionary Algorithm yields competitive performance to the state-of-the-art white-box LT-attack for tree ensembles ([48]). As previous black-box attacks performed worse compared to the LT-attack when used on tree ensembles ([48]), this opens up future research on the performance of this black-box attack, which can be tested on other types of machine learning models such as Neural Networks as well.

1.2. Outline

In chapter 2 we provide a background in Fuzzing, adversarial examples and tree ensembles and show the status of related work on the topic. In chapter 3 we introduce FATE. In chapter 4 we show that FATE is able to find adversarial examples on a multitude of datasets and in chapter 5 we improve the adversarial example generation strategy by improving the custom mutator and fuzzer initialization. chapter 6 explores the performance of FATE using various fuzzing engines. In chapter 7 we investigate the influence of the coverage guidance mechanism of grey-box fuzzers on the performance of FATE, we show our main results in section 7.4 and we validate the performance of FATE on two unseen datasets. Furthermore, in section 7.5 we show that FATE has competitive performance to the state-of-the-art when executed as a standalone (1+1) Evolutionary Algorithm. We conclude in chapter 8 with a discussion, conclusion and description of future work.

[1] https://github.com/cbilstra/FATE

2
Background

In this section we will provide an introductory background in tree ensemble models, we will discuss what it means to search for adversarial examples in tree ensemble models and we establish a background on fuzzers. We also describe the related work including the current state-of-the art for finding adversarial examples.

2.1. Decision-tree based models

Decision trees are deterministic classifiers that are fast to train and have a high degree of explainability and interpretability. They are a popular choice in both academic and industrial fields [24]. They have the structure of a tree with at each node a splitting condition, which determines if either the left or right branch of the node should be followed. The leaves of the tree contain the prediction of the decision tree and can be reached through providing input within a certain range. For example in Figure 2.1, the left-most leaf will be reached for any input that has a value <= 25 for the age feature. Unfortunately, decision trees suffer from both bias (underfitting through being unable to grasp the relation between features and the target output) and variance (overfitting on the training data through sensitivity for small perturbations in the data). Apart from classification, decision trees can also be used for regression tasks. For example, based on age and profession a decision tree could estimate the income of a person.

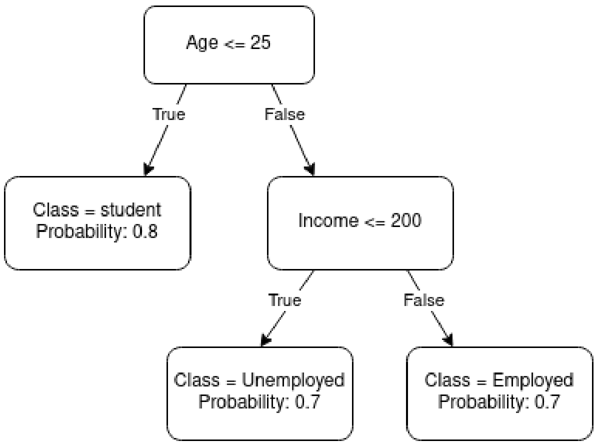


Figure 2.1: A simplified decision tree that predicts main occupation based on age and income

To combat the bias and/or variance of individual decision trees, they are often combined in tree ensemble models, where multiple decision trees are trained and combined in a specific way. Through creating multiple decision trees as smaller individual estimators and combining their predictions, the classifier becomes more robust to small perturbations in the input. Random Forests and Gradient Boosting are the most popular types of tree ensembles.

A **Random Forest** is an ensemble of decision trees where each tree is learned on a subset of the training data and a subset of the available features (a bagging approach to reduce variance). As a prediction, the output of the individual classifiers is combined, which makes the classifier more robust.

Gradient Boosting is an extension over normal boosting (where weak learners are iteratively added to the classifier, focusing on examples that were misclassified by the classifier in the last iteration to reduce bias). It uses a gradient descent algorithm to optimize a differentiable loss function. Individual decision trees are built one by one on different subsets of the training set, where the next tree is trained on the regression error of the previously trained trees. In Gradient Boosting, trees are thus used as regressors that predict values instead of probabilities. The initial prediction is modeled as for example the average value of the target column of the training data. For multi-class classification, at each step a tree is trained for each class separately as a one vs all classifier. The output is produced by summing the predictions (scaled by a learning rate) of the individual decision trees. A value below 0 means a higher probability that the input belongs to class 0 (binary classification) or a high probability that the input does not belong to a specific class (multi-class classification).

2.2. Adversarial examples

An **Adversarial Example (AE)** is a perturbed instance x' which differs a small amount from victim x (the original data point), but when fed to a classifier C, the output for both instances differ, e.g. $C(x) \neq C(x')$. Finding the best possible adversarial example, i.e.

$$r^* = min_{x'}\, d(x, x') \mid C(x) \neq C(x') \tag{2.1}$$

with distance function d is NP-complete for tree ensembles [24]. Generating good adversarial examples is important for assessing the robustness of ML models against unexpected input. It can also assist in adversarial training, which helps making ML models more robust. Furthermore, it can be interesting to visualise concrete adversarial examples to get an understanding for how a model makes its decisions.

Adversarial Attacks are algorithms that have the goal of finding a concrete solution $\bar{r}$ which is an upperbound of r^*. Unless solving r^* exactly, which is usually slow due its NP-completeness, attack algorithms cannot provide any formal guarantee on model robustness other than the model not being robust to perturbations bigger than the smallest $\bar{r}$ found.

Robustness Verification algorithms try to find r^* or a lowerbound $\underline{r}$ for a large set of victims. This guarantees that no adversarial examples exist within $\underline{r}$ distance from x. This is important for fields where machine learning algorithms are used for safety-critical tasks, such as the control system of a self-driving car or for aircraft control systems, where small adversarial perturbations may have fatal results [23].

An adversarial example can be found for any victim: if infinite adjustment is allowed, any point can be changed to any point belonging to another class. It is thus important that the features of an adversarial example do not differ too much from the features of the victim, i.e. they should be similar to each other. The following measures are the most popular for quantifying distance [24] (with x_f feature f of data point x and x' an adversarial example for x):

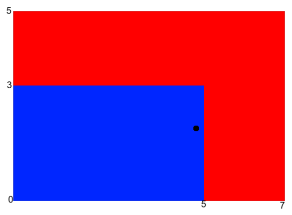
1. L_0 **distance**: the amount of changed features, also called Hamming distance. $|\{f \mid x_f \neq x'_f\}|$.
2. L_1 **distance**: the sum of differences of features. $\sum_f |x_f - x'_f|$
3. L_2 **distance**: euclidean distance. $\sqrt{\sum_f (x_f - x'_f)^2}$
4. L_∞ **distance**: maximum difference of features. $\max_f |x_f - x'_f|$

When optimizing for a different distance measure, the generated adversarial example will often also be different. For example, L_0 distance encourages a small amount of (large) perturbations, whilst L_∞ encourages (a large amount of) small perturbations.

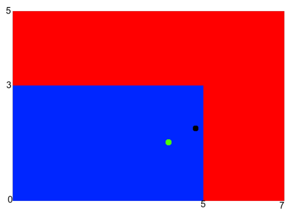
Adversarial attacks can generally be split into two types. **Decision-based attacks** [6, 12, 15], often black-box attacks, only observe the output (e.g. predicted class) of a target program or model. They typically start with an initial adversarial example with large distance and try to minimize the adversarial perturbation along the decision boundary. Due to black-box attacks not being dependent on the internals of the model, they can be used to attack any kind of model. For tree ensembles they are usually ineffective due to the discrete nature of tree ensemble models (many plateaus in the input space), while requiring a large number of queries [48].

Gradient-based attacks cast the attack into an optimization problem on a specially designed loss function, where the gradient is deduced from either back-propagation (in Neural Networks) or based on soft-label output such as confidence scores. A popular approach to generate adversarial examples in Neural Networks is to use the gradient of the loss function, but this is impossible for tree ensembles: For each leaf of a decision tree the prediction is deterministic and constant. This does not change when combining decision trees in a forest. This makes the decision function a non-continuous step function, which means the gradient cannot be estimated because every point in the input space is situated at a "plateau" where the decision does not change. Gradient-based attacks are usually a white-box attack, due to the required information from the model, and are thus often specific to a certain type of ML model.

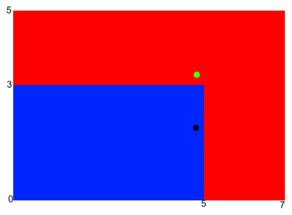
2.2.1. Adversarial examples for tree ensembles



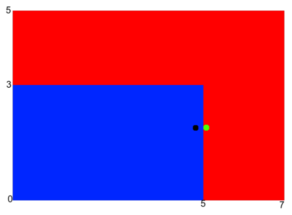
(a) The victim (black dot; original class: benign, predicted class: benign)



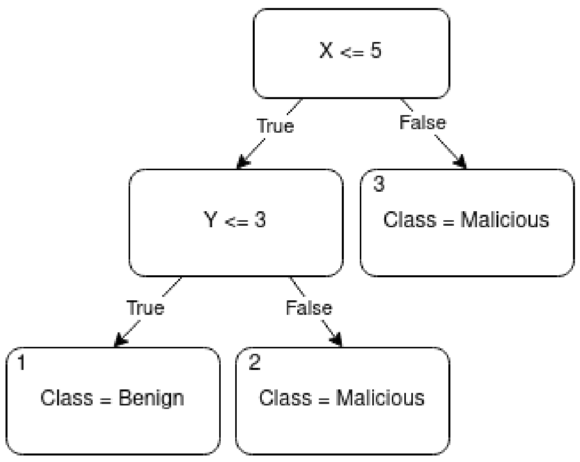
(b) Failed attempt to create an adversarial example (green dot)



(c) Adversarial example (green dot)



(d) Optimal adversarial example



(e) Simple decision tree to attack

Figure 2.2: Adversarial Examples for a simple decision tree

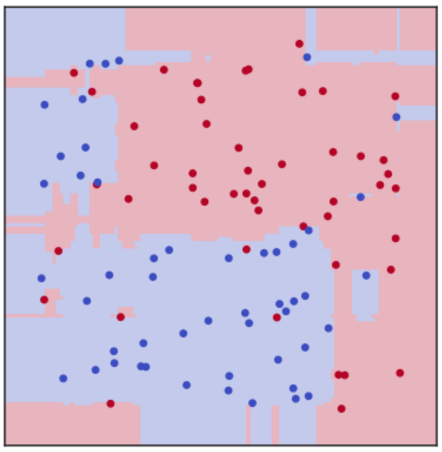


Figure 2.3: Decision regions of a simple Random Forest

Figure 2.3 shows the decision regions of a Random Forest trained on a simple toy dataset with only two features. Blue examples are "benign" while red examples are "malicious". Likewise, blue regions are regions in the input space where "benign" is predicted, while in red regions "malicious" is predicted. Finding the optimal adversarial example for a victim means finding the closest boundary of the closest decision region predicting another class and moving just beyond that boundary. We demonstrate this in Figure 2.2 for a single decision tree: the victim x (lying close to a decision boundary) is shown in 2.2a. A perturbed instance x' based on x is shown in 2.2b. This instance lies close to x, it is however not misclassified and is thus not an adversarial example. The attack algorithm continues searching for adversarial examples and at some point finds 2.2c. This is an adversarial example for x as it lies in a decision region where another class is predicted, however it is not the best adversarial example that can be found. The best adversarial example would be 2.2d, having the shortest possible L_∞ distance to x while having a different prediction.

Looking at the decision tree (Figure 2.2e) that produces the decision regions, the victim lies in leaf 1 (X < 5, Y < 3). The adversarial example from 2.2c lies in leaf 2 (X < 5, Y > 3) and the optimal adversarial example from 2.2d lies in leaf 3 (X > 5). Note that each leaf in the decision tree determines the prediction for inputs in a part of the input space. The path to each leaf poses constraints on possible inputs through which that leaf can be reached. These constraints determine the boundaries of a region in the input space, called a decision region or bounding box. Searching for adversarial examples in a decision tree thus means moving around the input space through adding perturbations to inputs, with the goal of discovering new leaves (decision regions) in the decision tree that lie close (in feature space) to the leaf of the victim, while predicting another class than the original class of the victim.

The victim we just attacked lied very close to a decision region predicting other classes, which means it could be attacked by only modifying its features (moving around the victim) just a little bit. When victims lie further away from decision regions predicting other classes, a larger perturbation is necessary to find an adversarial example and it might thus be more difficult to attack such victims.

Finding adversarial examples for tree ensembles works similar to finding adversarial examples for decision trees. The difference is that for ensembles the decision regions are not determined by single leafs, but by a combination of leaves: the prediction is made collectively through combining the predictions for the leaves that are triggered in each tree t. Each tree (leaf) restricts the input to lie in a certain bounding box in input space. The intersection of the bounding boxes for all leaves that are triggered determines the decision region in which the input lies. Such a decision region exists for each combination of leaves in the ensemble that can be reached, that is, when there is a non-empty intersection of the bounding boxes for that combination of leaves. Each decision region of a tree ensemble classifier thus consists of a combination of leaves from the ensemble. A new combination of leaves generally means a change in class-probability prediction. If this prediction changes sufficiently, an other class will be predicted and an adversarial example is found.

Robust training can be used to make it harder to attack victims. Where normally Tree (Ensemble) training algorithms determine the best splitting conditions based on the Gini impurity or Information Gain criterion, some robust training algorithms also take into account that an adversary can perturb instances when calculating this criterion (Vos et al., [41]; Chen et al., [10]). They optimize for accuracy under attacker influence while determining the best split. TREANT [8] optimizes any convex loss-function under attacker influence but is more costly in runtime because it repeatedly applies a numerical solver. Andriushchenko et al. ([2]) derive a strict upper bound on the robust test error and show how provably robust tree ensembles can be created in epsilon radius with respect to L_∞ norm perturbations. Kantchelian et al. ([24]) use a form of adversarial training by inserting adversarial instances (with the same label as the victim) with small distance to the training points during each training iteration of Gradient Boosting models. However, by hardening the model for L_0 norm perturbations this way, the model became more susceptible for L_1, L_2 and L_∞ perturbations.

2.2.2. Images

Where small perturbations to continuous features may often not make the input invalid, images have the characteristic that adversarial perturbations can alter the image significantly. Even if one perturbation (mutation) may keep the image semantics, sequential mutations (which happens when fuzzing) can still make the image "invalid". [43] performed a user study between perturbing with the L_∞ norm and their proposed conservative setting. Although their conservative setting yielded more valid images, L_∞ turns out to be still quite effective for images. They show that careful design and parameter tuning of the mutation strategy can also help to reduce image invalidity ratio for particular application scenarios.

2.3. Search algorithms

To find adversarial examples, the input space should be searched for inputs that trigger new decision regions. As the input domains are often large and multi-dimensional, performing a depth- or breadth-first exhaustive search is often infeasible. When through the use of a heuristic (fitness function) the fitness of an input sample can be determined, algorithms can be used that are more efficient than performing an exhaustive search, though they cannot guarantee that an optimal solution is found:

Hillclimbing is a local search method that gradually climbs the solution space to an optimal solution. It repeatedly queries the fitness function to determine which neighbour of a point is the best, until no better point can be found (the local optimum is reached). This is only possible for a continuous fitness function that changes its output for any change in the input. Because of only evaluating its direct neighbours, this local search algorithm generally only works well when the neighbourhood is well defined and not too large [47].

Randomized Hillclimbing is an extension to the Hillclimbing search method that changes it to global search: instead of looking at all direct neighbours of an input, the input is mutated slightly and the mutated input is then compared with the original input through the fitness function, continuing with the better of the two.

It is important that through applying mutation successively, theoretically every point in the input space can be reached in order to be able to find the optimal value. On the other hand the mutations should be small enough such that an input is not completely replaced, which would result in random search. A mutation should thus constitute a reasonable change to an individual that still maintains most of its traits [47].

Randomized Hillclimbing is otherwise known as the (1+1) **Evolutionary Algorithm**. Evolutionary algorithms try to mimic the natural processes of evolution. DNA (the input) can mutate, and generally the strongest strains (those with the highest fitness) will survive. This improves the general fitness of the population, which will generally increase until some optimum. The (1+1) Evolutionary Algorithm is a specific case in which there is a population size of 1 which produces 1 new mutation.

The most common evolutionary algorithm however is the **Genetic Algorithm**, which is based on the intuition that problem solutions can be genetically encoded. Each part of the input describes a feature of the individual (e.g. hair color, or in the case of machine learning the general sense of a feature). Either through crossover (combining, removing or adding pieces of strains) or mutating features from individuals, new individuals are created which are then evaluated against the fitness function, keeping only the fittest few individuals. Both crossover and mutation are probabilistic actions, which mean they may or may not happen and the outcome will be different each time. A genetic algorithm is a global search algorithm, as through mutation it can reach all possible inputs. Genetic Algorithms are flexible and generally scale well to larger test problems [47].

2.4. Fuzzing

Fuzzing is a concept first introduced by Miller et al. ([30]) in 1990. They showed that by feeding a lot of random inputs to UNIX utilities, a lot of them would crash due to unexpected inputs. Since the introduction in 1990 a lot of improvements have been proposed and fuzzers have shown their added value in finding bugs in code [19]. Fuzzing is an active field of research, as shown by the amount of papers that propose or study improvements to fuzzers discovered by recent literature studies ([9, 25]).

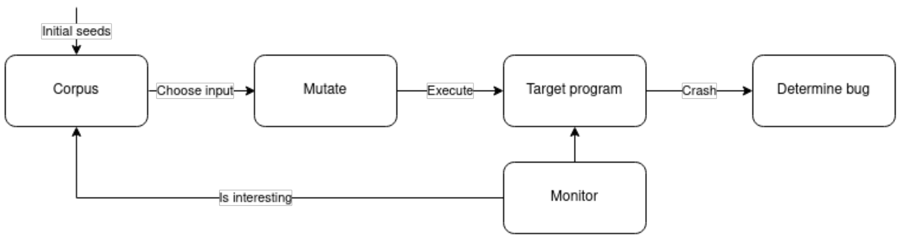


Figure 2.4: The high-level architecture of a typical fuzzer

Modern fuzzers such as AFL [46] and LibFuzzer [29] and their derivatives all use a similar structure [25], which is summarised in Figure 2.4. The fuzzer first chooses the next input to mutate from the corpus, which is in the ideal case seeded with example inputs before the first execution. This input is then mutated, either by adding noise (mutation) or by combining the input with another input from the corpus by performing crossover (inserting, deleting or swapping parts of the input). The new input is then fed to the target program and executed. This execution is monitored by the fuzzer, which determines if the input was interesting (when new behavior is discovered) through e.g. inspecting code coverage, output or instrumentation. If the input was interesting, it is saved in the corpus to be fed in mutated form to the target application again later on. Modern (grey-box) fuzzing can be seen as a genetic algorithm with the corpus being the population and code coverage being the fitness function. Fuzzers are designed to detect bugs and crashes in software that are triggered by unexpected input.

Access models Fuzzers can be divided based on the amount of information and access they use from the target application. **Black-box** fuzzers only monitor input/output behaviour such as crashes, timeouts and used instrumentation. The typically only have access to the binary to be fuzzed. **White-box** fuzzers in contrary, use heavy-weight program analysis, constraint solving and fine-grained execution monitoring [31]. They improve mutation and monitoring by exploiting the semantics of the target application. **Grey-box** fuzzers are a trade-off between black- and white-box fuzzers. They generally act on source code and use lightweight program analysis techniques such as code coverage tracing and try to maintain the high execution rates of black-box fuzzers. Grey-box fuzzers are the type most commonly used, with popular examples being AFL [46], LibFuzzer [29], VUzzer [35] and honggfuzz [21].

In general, fuzzers make a trade-off between better executions (e.g. by better estimating which inputs will trigger new code coverage) and a higher number of executions per second. For example, in the time it takes for a Directed Symbolic Execution fuzzer to execute a single input, a grey-box fuzzer can execute several orders of magnitude more inputs [4]. Below, we describe the core components of a fuzzer in more detail.

2.4.1. Corpus

The corpus is a set of inputs (seeds) that the fuzzer has saved. These inputs should be as good as possible at achieving the objective the fuzzer aims to reach. These inputs are often saved as files, such that they can be shared between fuzzing runs. The corpus is initialized with initial seeds, arrays of bytes that ideally follow interesting paths through the code. The quality of the **initial seeds** is very important for the performance of fuzzers, and individual runs can differ a lot based on the initial seeds used [25, 28]. The seed should be both small (not contain unnecessary information) and valid, to make sure they pass the input parsing stage which is part of most applications and rejects most input. Fundamentally, a fuzzer can also be used without initial inputs, but depending on the complexity of the input the fuzzer may have a lot of difficulty at penetrating deeper into the target application (e.g. passing the parser).

Improving **input selection** can aid the performance of a fuzzer. For example [5, 7, 33] assign different energy values to seeds, increasing the chance that seeds are mutated that show interesting behavior. A concrete implementation of [5] is available directly as an option for LibFuzzer. Based around the intuition that new behaviour equals a gain in information, they use Shannon's entropy to qualify how much is learned from new behaviour. More energy is assigned to seeds that provided more information as it is more likely that mutating those will also reveal more information. This energy determines the probability that a seed is chosen as the next input for mutation. Odena et al. ([32]) show that uniform random seed selection worked acceptably well, but in some cases fuzzing was sped up considerably by favoring more recently discovered corpus elements, prioritizing the exploration of points further from the initial seed.

2.4.2. Mutation

The mutator of a fuzzer generates new input based on an existing seeds which can then be fed to the target application. Its mutation strategies can be generalised into Mutation-based and Generation-based.

Mutation-based strategies are generally simple, fast and rely on random mutations and genetic operators (crossover). Most fuzzers are mutation-based out-of-the-box because of their universal applicability [31]. This type of mutation may however cause the input to be rejected early if the generated input deviates too much from the expected input format (e.g. when input files require a special structure that is parsed in the target). Mutation-based fuzzers have been used extensively for security testing of applications [9].

Generation-based strategies can take into account the complexity of the input format and can therefore drastically improve fuzzing performance. Examples of generation-based strategies are grammar-based mutation, where "words" that can be present in the input which would be difficult for the fuzzer to find by itself are supplied to the fuzzer. This speeds up the fuzzing drastically, especially for short runs [22]. Structure-aware fuzzing uses custom mutators that implement the specific input format of the target application, thereby allowing to for example provide correct checksum information, upon which the target application would otherwise often fail. Others generate inputs according to a model or block [9] or train a Neural Network to generate seeds with complex input format [18, 50]. Mutation strategies can also be combined by "stacking": combining structural (high-level) and bit-level (low-level) operators together to generate interesting inputs [33]. Generation-based mutation usually requires a new mutator or grammar to be implemented for each new input format, which can be time-consuming or difficult due to complex input formats. In practise, generation-based strategies may also be too slow for efficient fuzzing [44].

2.4.3. Monitoring

Monitoring is critical for a modern fuzzer as a way to provide some kind of fitness to inputs. Whereas simply feeding random input to a target program can theoretically unmask any bug, chances that exactly the correct input is guessed may be very small. Building upon an input that is known to have progressed towards the

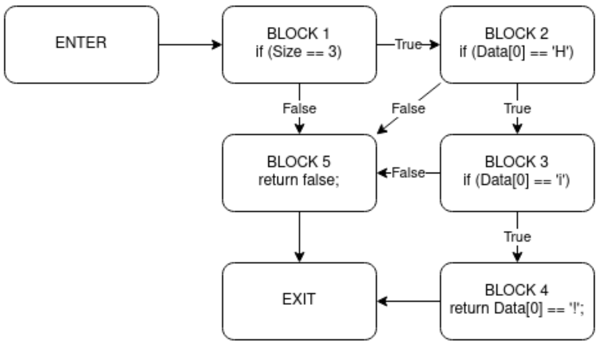


Figure 2.5: An example of a CFG of basic blocks

objective of the fuzzer intuitively improves upon starting with a clean sheet every execution. Different kinds of fuzzers have different objectives and monitoring strategies.

Coverage-guided fuzzing records which code regions (basic blocks) are reached by inputs. Using a control-flow graph (CFG) such as in Figure 2.5, it can identify basic blocks in the target that have not yet been reached by previous executions. The goal of a coverage-guided fuzzer is to cover every edge in the CFG, which corresponds to visiting every basic block from every basic block from which it can be reached. Through instrumentation inserted at compile-time the fuzzer tracks if new edges in the CFG have been covered during execution [29]. If so, the coverage-increasing input is saved to the corpus. An example fuzz target can be seen in Listing 2.1. A coverage-guided fuzzer will for example remember input which satisfied the condition of branch 0, with the goal to satisfy branch 1 and further. This allows the fuzzer to progress further in the source code step by step. This way the fuzzer can generate fairly structured inputs, although fields like checksums will be very difficult for a standard coverage-guided fuzzer.

```
bool FuzzMe(const uint8_t *Data,
    size_t Size) {
  // Branch 0
  if (Size == 3) {
    // Branch 1
    if (Data[0] == 'H') {
      // Branch 2
      if (Data[1] == 'i') {
        return Data[2] == '!';
      }
    }
  }
  return false;
}
```

Listing 2.1: Example fuzz target

Coverage-guided fuzz testing is more effective in improving coverage than random testing [32], especially for criteria that are difficult to cover [43]. Furthermore, coverage-guided fuzzing is generally more scalable than techniques like symbolic execution (which can also solve path constraints) because the time for one test depends on the execution time of the program rather than the size or complexity of its input [3]. This high efficiency of coverage-guided fuzzing strongly contributes to its performance [4]. In addition, coverage-guided fuzzing is highly parallelizable because the seed files represent the internal state of the program, without any required additional synchronization between fuzzing instances. For grey-box coverage-guided fuzzers, coverage tracing is generally the most time-consuming task [31].

Böhme et al. ([7]) show that there are high- and low density regions in the code coverage exercised by a fuzzer. Most inputs exercise a few high-frequency paths, which is often the path taken for invalid inputs. Just 10% of all paths are exercised by only a few inputs. These represent the valid and interesting inputs. Approaches such as AFLFast [7] prioritize fuzzing low-frequency paths with the goal of discovering more interesting inputs.

Directed fuzzers aim to reach specific locations (code regions) in the target, contrary to coverage-guided fuzzers which aim to reach every code region in the target. The rationale behind directed fuzzers is that it is wasteful to allocate resources to reach code regions which are not interesting or unrelated to finding bugs. They change the fitness function from code-coverage to (basic-block) distance to the target location. When only a small set of pre-determined parts of the code needs to be tested directed fuzzing can greatly speed up the fuzzing process.

Symbolic Execution can be used to aid in traversing conditional guards that are difficult for the fuzzer to guess. For example conditions that contain "magic numbers" may otherwise be difficult to guess for a mutation-based fuzzer: if in Listing 2.1 branch 1 would be `if (x == 42.6662997)`, a fuzzer which does not use some kind of magic number analysis might not find an input with `x == 42.6662997` quickly. Symbolic execution is then used to determine how x can get the value 42.6662997. When using symbolic execution in combination with SMT solvers, the amount of coverage-increasing inputs can be significantly increased at the cost of much lower execution speed of the target [31].

Next to observing code coverage, fuzzing approaches may also observe instrumentation to qualify inputs, or observe any output given by the target (e.g. crashes, return codes, runtime of an input), especially in black-box settings where the source code is not available. Rawat et al. ([35]) use static analysis to assign different rewards to program locations to be reached based on how deep that location resides in the CFG and how likely it is that traveling through that location will result in a triggered vulnerability (through error-handling block detection). They combine static analysis (constant string extraction and the basic block weight calculation) with dynamic taint analysis and basic block tracing, which has a lot less overhead than symbolic execution but achieves similar goals. Chen et al. ([13]) try to improve on symbolic execution fuzzing in solving path constraints by byte-level taint tracking, context-sensitive branch count, path constraint search based on gradient descent and input length exploration.

2.5. Related work

In this section we discuss the state of the art for generating adversarial examples for tree ensembles. We also discuss recent usages of fuzzers for verifying other types of machine learning models.

2.5.1. Adversarial examples for tree ensembles

Kantchelian et al. ([24]) developed a formulation to find optimal adversarial examples using a MILP solver. This formulation is the current state-of-the-art for finding optimal adversarial examples for tree ensemble models. They encode the predicates (if a splitting condition is either true or false), the leaf variables (whether a prediction leaf is active) and an objective variable for the L_∞ norm. They enforce consistency in the ensemble (e.g. the predicates leading to a leaf should be satisfied if that leaf is active, exactly one leaf should be active per tree) using constraints. To find adversarial examples, a constraint is added that the victim should be mislabeled. E.g. if the victim belongs to class 0, the sum of the leaves of the perturbed instance x' should be bigger than 0 (for binary-class gradient boosting). This formulation is then solved using a branch-and-bound approach by a MILP solver. All variables of the formulation are linear except the variables that decode if a node is part of the path to the currently activated leaf. These are first set to linear variables $0 \leq p_i \leq 1$ and solved quickly using Linear Programming. This yields approximate answers for which the approach is executed again using Mixed Integer Linear Programming (MILP), encoding p as a binary variable again (values close to 0 will be treated as 0 and values close to 1 will be treated as 1). The solver proves optimality by approaching the problem from two sides: it finds valid adversarial examples using the Primal while trying to prove that the adversarial example cannot be improved upon using the Dual. The Dual proves that the classification of a victim cannot be changed without perturbing the features with a certain distance. When the primal and dual solution are equal, e.g. an adversarial example is found with distance 0.12 and it is proven that the minimal adversarial perturbation should be 0.12 as well, an optimal adversarial example is found. This approach however takes exponential running time, as Kantchelian et al. showed by a reduction to the NP-complete 3-SAT problem. The runtime of this algorithm can be potentially improved by only calculating the Primal and returning if no improved adversarial example is found within some amount of time.

Zhang et al. ([48]) transform the input space $\mathbb{R}^d$ to a discrete Leaf-tuple space $\{1,2,...,N\}^K$ with N the maximum number of leaves per tree and K the number of trees. Each leaf poses constraints on the possible values of features required for reaching that leaf, corresponding to a bounding box in input space. The bounding box of a leaf tuple is defined as the cartesian product of the bounding boxes of the individual leaves. Within that bounding box, the closest possible point x' based on distance measure d is selected with regards to the victim x. With the number of trees that have different prediction leaves as distance measure, they search the neighbourhood with distance 1 of each tuple, i.e. all leaf combinations with one different prediction leaf that are valid (when there exists a point $x \in \mathbb{R}^d$ that can trigger that leaf combination). Doing this naively, e.g. using a model query to determine if a leaf tuple is valid for each possible tuple, is too costly. They utilize the fact that only the leaves of trees that are bounding x' (when the selected leaf of tree t is constraining the value of x' to move in at least one direction) need to be changed. This way they only need to investigate a subset of the neighborhood at each iteration.

Their LT-attack consists of an outer loop (iterating until no better adversarial example is found) and an inner loop (generating and searching the bound neighborhood with hamming distance 1). The Inner loop computes the trees that are currently bounding x' and runs a top-down traverse for each bound tree with the intersection of the other bounding boxes $B^{-(t)}$. A leaf of that tree forms a valid tuple if it has non-empty intersection with $B^{-(t)}$. $B^{-(t)}$ can be efficiently obtained through caching the K bounding boxes in B' and maintaining a sorted list of right and left bounds. The attack then enumerates all leaf tuples in the bound neighborhood, thus the complexity of the algorithm depends on the size of this neighborhood. They show that the size of this neighborhood is generally acceptably small for real-world datasets.

They seed the algorithm with n random initial adversarial points, which are first improved by a fine-grained binary search. In theory, through multiple smaller neighbourhood updates an adversarial example far away can be reached. Their method can be adapted to any distance metric and is next to the attack from [24] the only white-box attack specifically designed for tree ensembles. By also searching in neighborhoods with larger hamming distances, optimal adversarial examples can be found. This however greatly increases computational time.

Andriushchenko et al. ([2]) propose the CUBE attack, a randomized hillclimbing (1+1 Evolutionary Algorithm) black-box approach for L_∞ norm distances. At every iteration a random subset of the features is mutated with $\pm 2\epsilon$ with probability p, and afterwards projected such that the new data point x' lies within a ball with radius ϵ from victim x. Fitness is calculated by the functional margin, which they minimize (thus working towards an adversarial example of which the classifier is more and more certain it must belong to a different class than the original class). Adversarial examples are always situated at the corner of the feasible set, which is also the disadvantage of CUBE. They however empirically show that in practise not many decision regions are missed because of the small size of the L_∞ balls.

Cheng et al. ([15]) introduced Sign-OPT, a query-efficient decision-based black-box (hard-label) attack. They use the formulation of [14], that allows the problem of finding the minimum adversarial perturbation to be reformulated as another optimization problem which often has a smooth boundary. Cheng improves this by proposing an evaluation of the sign of the directional derivative which only uses a single query, resulting in an attack that is extremely query efficient.

Chen et al. ([12]) introduce the HopSkipJumpAttack (HSJA), another query-efficient decision-based black-box attack. It uses binary search to identify the decision boundary between two points from different classes, and by leveraging gradient estimation of the decision boundary it is able to find adversarial examples with a small amount of queries. It generally performs better (mainly time-wise) than Sign-OPT [48].

Yang et al. ([45]) proposed RBA-Appr. The attack is developed around the fact that for e.g. k-NN and Random Forest classifiers, decision regions can be decomposed in convex sets. The intuition behind the attack itself is that when the closest decision region in the decomposition that predicts another class than the original class of the victim can be identified, an optimal adversarial example can be found. They develop an exact algorithm that searches over all decision regions, however this algorithm is very slow at it solves an NP-hard problem (as shown by Kantchelian et al., [24]). Decision regions predicting other classes reside around training samples of other classes than the class of the victim. They thus perform a search over training points close (in L_p distance) to the victim, creating a subset $S' \subseteq S$ of all polyhedrons. These polyhedrons are then attacked with the exact algorithm, resulting in a much lower (and feasible) running time of the attack.

Comparing all these attacks, Zhang et al. ([48]) show that their LT-attack finds the best adversarial examples (not taking into account the exact MILP attack of [24]) on a wide variety of datasets, while generally also having the fastest runtime.

2.5.2. Coverage-guided fuzzing for verifying (D)NN

Fuzzers have been used before to test machine learning models, namely (Deep) Neural Networks. Odena et al. ([32]) created TensorFuzz, a coverage-guided fuzzing framework for discovering errors in Neural Networks that occur for rare inputs. They show that trivial coverage metrics are too simple for measuring coverage in Neural Networks, so they used an Approximate Nearest Neighbour algorithm to measure coverage. They combine this with Property-Based Testing through objective functions to for example identify broken loss functions.

Xie et al. ([43]) also created a coverage-guided Fuzzing framework for testing DNN. They perform better than TensorFuzz on their datasets for image recognition. They focused specifically on images, and showed that through combining multiple L_p norms images can be generated that have a higher validity ratio than only using the L_∞ norm. They experimented with multiple seed selection algorithms as well as different coverage measures. For their approach Probabilistic seed selection works best.

3 Methodology

In this section we introduce FATE, a tool to translate tree ensemble models to a C++ target which can then be fuzzed for adversarial examples by general-purpose fuzzers. We custom design all parts of the fuzzer that are most important for good fuzzing efficiency ([25, 27]): the initial seeds, the mutator and the way we test the target through an objective function. A high-level overview of the architecture of FATE can be seen in Figure 3.1. A tree ensemble model is first trained on the training data. We use scikit-learn for this step, but models can also be trained using other libraries such as XGBoost [42]. This model is loaded into FATE which translates the model to source code, combining it with an objective function that identifies and saves adversarial examples. This source code is then fed to a fuzzing engine which starts searching for adversarial examples for victims in the testing set that are correctly classified by the model (otherwise the victim itself would be an adversarial example). The fuzzing engine keeps generating large amounts of inputs while the objective function that we injected into the source code checks if the input resembled an adversarial example. The fuzzing engine keeps running until instructed to stop by FATE. FATE then processes all adversarial examples found by the fuzzer, keeping the best adversarial example per victim. The overview shows that FATE is agnostic of training procedure or dataset, although we assume all data used for training and testing is in the range [0, 1] to maintain general applicability between datasets. FATE is designed using libFuzzer [29] as a fuzzing engine, but other fuzzers are evaluated as well in chapter 6. The individual components of FATE are discussed in more detail below. Implementation details of the fuzz target itself together with textual explanations are provided in Appendix D.

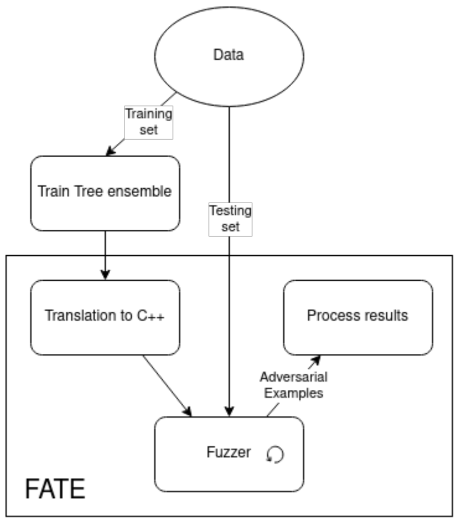


Figure 3.1: A high-level overview of FATE

3.1. Fuzzing

Although fuzzers exist that are designed specifically for certain frameworks, such as TensorFuzz [32] which is designed to be used solely with TensorFlow, we created an approach which is easily transferred between different fuzzers such that FATE can easily make use of new breakthroughs in fuzzing through changing fuzzers in the future if necessary. Fuzzers generally act on either binary programs or source code. When the source code for an application is available that option is preferred, as that gives more options to the fuzzer for e.g. instrumentation. We translate the tree ensemble model that should be fuzzed to C++ source code because popular fuzzers are generally able to fuzz C++ source code directly. Also, C++ is renowned for its speed.

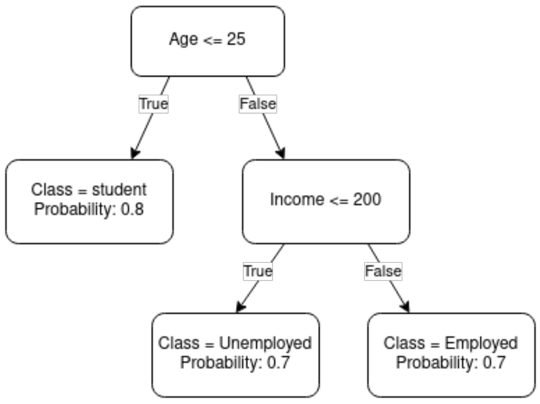
We develop FATE around libFuzzer, a fuzzer that is subject to recent research ([25], [5]), under active development [29] and successful in identifying bugs [19]. For example AFL(++) also satisfies these criteria, but in general libFuzzer is a bit faster in execution speed, despite requiring more setup [22].

Grey-box fuzzers such as libFuzzer try to penetrate deeper in the target through continuously mutating seeds that previously increased basic block coverage. In the source code we generate, increasing basic block coverage should thus be strongly related to finding more and better adversarial examples such that the objective the fuzzer tries to optimize strongly resembles finding good adversarial examples. Recalling from chapter 2, coverage in the tree ensemble is necessary for finding adversarial examples, but also for improving them. A substantial part of the coverage that can be reached in the target should thus be basic-block coverage in the tree ensemble.

For all but the smallest models, the most basic blocks (up to 44 MB of source code for the biggest models) in the produced C++ code are part of the tree ensemble. Code branches other than the ones in the tree ensemble mainly reside in the objective function (ca 450 lines of code) and in the functions used for mutating and producing the correct output of the tree ensemble (ca 400 lines of code). Increasing code coverage thus mainly means that more leaves of the tree ensemble are explored.

3.2. Tree ensemble model to source code

Tree ensembles consist of multiple decision trees as weak learners, which are trained and combined in different ways. We thus need to rewrite both the single decision trees and the way they are combined to C++ code. Translating a single decision tree to a function is trivial: each splitting condition produces an if-else branch, with the leaves determining the values returned from the function. Figure 3.2 shows a simple decision tree (out of a Random Forest) with the produced C++ code next to it. As all data is pre-processed to be in range [0, 1] per feature, age 25 equals a feature value of 0.25 in the first split (the age column has range [0, 100]) and an income of 200 equals a feature value of 0.04 in the second split (The income column has range [0, 5000]). Each of the leaves returns a probability that the input belongs to either one of the classes (Student, Unemployed, Employed).



```
double* tree_0(double features[]) {
  static double res[3];
  // splitting threshold 1
  if (features[0] <= 0.25) {
    // leaf 1
    res[0] = 0.8; res[1] = 0.1;
    res[2] = 0.1; return res;
  } else {
    // splitting threshold 2
    if (features[1] <= 0.04) {
      // leaf 2
      res[0] = 0.3; res[1] = 0.7;
      res[2] = 0.0; return res;
    } else {
      // leaf 3
      res[0] = 0.3; res[1] = 0.0;
      res[2] = 0.7; return res;
    }
  }
}
```

Figure 3.2: Simple decision tree as code

To create a functional tree ensemble model in C++, we need to combine the outputs of these decision trees in the correct way. Please refer to section D.3 for the actual C++ implementation. For **Random Forests**, the class-probability outputs of the individual decision trees are summed and the class with the highest sum of probabilities is predicted.

For **Gradient Boosting**, the outputs of the decision trees are single values as the decision trees are now used as regressors. Outputs of single trees are scaled by a learning rate and added to the initial prediction. The class with the highest sum s is predicted. For binary classification, the class-output probabilities are calculated as $[1-l, l]$ with $l = \frac{e^s}{1+e^s}$. For multi-class classification, for each iteration (tree) a separate decision tree is trained for each class as a one-vs-all classifier. Each of these sub-trees thus computes the regression value for one class. For class i, the prediction for each tree t_i is scaled by the learning-rate and summed to z_i. The probability of the input belonging to class i is then calculated with a softmax function $\sigma_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$. These are the default combination techniques for scikit-learn [36], which we used to train our models. Other combination techniques are currently not supported, but can be easily added to FATE.

3.3. Mutation

The input to the fuzzer is a list of bytes (doubles) representing the values of single features. This can be seen as a sequence genetic encoding with the features being the genes. Grey-box fuzzers like libFuzzer act like a genetic algorithm. They randomly mutate the genes (bytes) and code coverage acts as the fitness function: when new coverage is discovered the seed is saved as a member of the population to be mutated later. This allows the fuzzer to explore beyond the nearest neighbours of seeds: larger steps can be taken around the input space. This is necessary because the decision surface of tree ensembles has a lot of regions (plateaus) in which the decision is not altered by moving towards the direct neighbours of a point: the discrete steps of the single trees combined together form a bounding box which produces the same output.

Fuzzers generally mutate bytes in the input randomly or using information gathered from either the source code of the target or previous executions. For example, some fuzzers are able to record which values are used for compare instructions and can insert those values at the correct locations in the input to try to penetrate deeper into the source. As we a creating the target ourselves, we already have more knowledge than the fuzzer beforehand though: as all input resides in the domain [0, 1] for each feature, we can already improve the fuzzing a lot by only searching on that interval instead of also investigating values outside that domain. To this extend, fuzzers such as libFuzzer allow "custom mutators" that give all the control of mutating bytes to the user. Writing a custom mutator allows us to incorporate knowledge about the model and victim during mutation which we explore in chapter 5. The mutator can either be used in a white-box setting, where we can use information specific to a certain type of model such as splitting thresholds for tree ensemble models, or in a black-box setting where we cannot use this model-specific information and only execute (query) the model and observe the hard-label (prediction) and soft-label (class probabilities) output. Apart from section 7.5, throughout this thesis the white-box setting is assumed. In chapter 7 we further explain what information of the model we use in the white-box setting. Below, we introduce the baseline mutator of FATE. As we only test with L_∞ distance, the mutation function is also optimized for this distance measure. Some of these optimizations may not apply for other distances and when testing with a different distance it could be beneficial to adjust the mutation function or settings.

3.3.1. Baseline

We implement a generation-based mutator: We use the domain knowledge that all data should be in the range [0, 1] to only mutate features within this valid range. Each feature (gene) is mutated with a probability p which is a common approach [47]. p is initially set to 0.1 such that an input will generally not change too much during a mutation. It is unnecessary to mutate features that never occur in splitting conditions: their value does not influence the outcome of the model. We exclude those features from mutation as their original value will be the best value. Fuzzers consecutively apply mutations for 1 to mutation-depth times randomly. For libFuzzer, mutation-depth = 5 by default. This, together with mutating each feature with chance p, allows mutations of multiple features at the same time such that the fuzzer can more easily jump over local extremes in the decision surface. Mutations are drawn from a Gaussian distribution ($\mu = 0, \sigma = 0.34$, shown in Figure 3.3) to favor smaller mutations such that the mutation will create new individuals that mostly maintain its original traits [47], while also allowing to explore points further away with a smaller probability. The standard-deviation is chosen as 0.34 such that it is just possible to have a complete mutation of a feature (± 1), although with a very small probability.

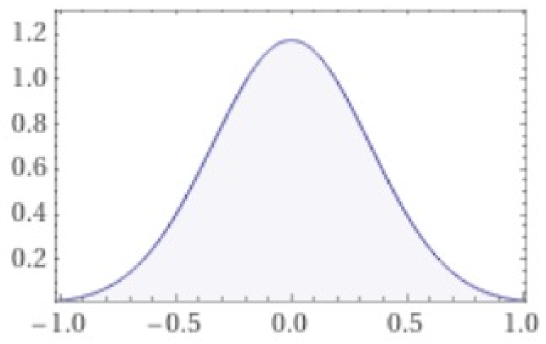


Figure 3.3: Gaussian distribution with $\mu = 0, \sigma = 0.34$

In this work we concentrate on fuzzing datasets consisting of continuous features, even though fuzzing discrete features could also be suitable for a fuzzer due to the limited amount of possible values per feature. The L_∞ norm is less meaningful for discrete features though: what would be the distance between two discrete choices? For attacking discrete features refer to [26]. We also implement an ϵ restriction, which limits all mutations to be within an ϵ radius around the input. To recall, a mutation of 0.1 resembles a 10% point difference of the feature. An ϵ value of 0.1 thus restricts mutations to 10% point from the seed that is mutated. The neighbourhood N of seed s is thus bounded by md consecutive ϵ-balls, with a much higher probability that the mutated input will reside close to the center of the first ϵ-balls because of the Gaussian distribution. Through continuously applying mutations on top of each other, theoretically inputs over the whole input space can be reached. As seeds are saved when they discover new coverage in the source code, mutations can gradually explore the input space by continuously mutating the newly found seeds.

Fuzzers, like genetic algorithms, also use **crossover** functions. The built-in crossover function of libFuzzer inserts, deletes and combines bytes of seeds. For our use-case, inserting or deleting bytes has no meaning, as the input always needs to resemble n features. Insertion and deletion are thus removed and the crossover function only splits and combines two seeds at a random feature index. The crossover function allows combining features of two different seeds which potentially discovers completely different combinations of leaves in the Forest.

3.4. Objective function

Fuzzers generate large amounts of mutated inputs, and we need to distinguish which of these resemble adversarial examples. Odena et al. ([32]) used an objective function to assess whether the neural network they were testing reached some particular erroneous state. This can be seen as a form of Property-Based Testing with the property being violated the behaviour we are interested in. FATE uses such an objective function to identify adversarial examples. As fuzzers such as libFuzzer try to optimize basic-block coverage, the objective function plays an important role in translating basic-block coverage into finding (better) adversarial examples. First of all, much coverage in the objective function should generally mean finding a good adversarial example. Furthermore, the objective function should give a kind of "directedness" to the fuzzer: the fuzzer saves inputs that reach new code branches and those branches should work towards an optimal adversarial example. The fuzzer must also always have the current best adversarial example in its corpus such that it can mutate further on that example. The objective function used by FATE achieves these goals and is shown simplified in Figure 3.4. For the actual code please refer to section D.5. The objective function first calcu-

```
void check(double fuzzed_features[], double original_features[],
           int fuzzed_class) {
  double distance = l_inf(fuzzed_features, original_features);
  if (fuzzed_class != original_class) {
    if (distance <= 1.0) {
      dummy_var = 0;
      if (distance < 0.99) {
        dummy_var = 1;
        ...
          if (distance < 0.01) {
            dummy_var = 7;
          }
        ...
      }
    }
  }
  if (distance < best_distance) {
    best_distance = distance;
    interval = round(distance, 5);
    write_if_not_exist(interval, fuzzed_features);
  }
}
```

Figure 3.4: Objective function

lates the distance between the current features and the victim. If the current input is an adversarial example (`fuzzed_class != original_class`), the seed triggers different distance branches with decreasing adversarial distance. If the distance is lower than previously seen, a new branch will be triggered and the input saved by the fuzzer. The best known adversarial example will thus always be known as a seed by the fuzzer and when the fuzzer reaches more coverage in this objective function, it per definition finds better adversarial examples. In chapter 5 we experiment with different intervals between the distance branches; this is a trade-off between the amount of seeds in the corpus (if a very small improvement triggers an input to be saved the size of the corpus can grow very quickly) and accuracy (the fuzzer will probably find better adversarial examples when the best known adversarial seed is close to the actual best adversarial example found by the fuzzer). In every branch a dummy variable is referenced. We do this to make sure the distance branches are not simplified when the compiler optimizes the code. This dummy variable is printed in a piece of code (shown in section D.6) that the fuzzer will never reach, which the compiler cannot know.

The created adversarial examples need to be saved for later processing in FATE. Fuzzers expect the target to be stateless: saving these adversarial examples in the fuzzing engine itself and communicating them to FATE after execution would require alteration of the fuzzer engine itself and would make the approach much less transferable between different fuzzers. One option is to parse all seeds in the corpus, as the best known adversarial example is just a seed in the corpus. This is however problematic as some of the seeds may not be adversarial examples because new coverage can originate from e.g. new coverage in the forest branches as well. Running all the seeds through the model again to check if they are adversarial examples is costly: there can be tens of thousands of seeds. We can however initialize a bit of memory in the target - before the fuzzer starts testing different inputs - where all individual fuzzing runs can read from and write to. We leverage this to save the distance of the current best adversarial example. A newly discovered adversarial examples is then only saved to a file if it improves upon the current best known adversarial example. Saving the best known adversarial example introduces state in the fuzzer which should generally be avoided. For our use case this works though as we only fuzz each victim with one thread at a time, so concurrent memory writes are not possible. Furthermore, the amount of information we save is limited to a `double` only.

Our hypothesis is that the directedness of the structure of the objective function towards finding smaller adversarial examples may be very suitable for directed fuzzers. When setting the target of a directed fuzzer to the branch with the smallest adversarial distance, the fuzzer will prioritize seeds that work towards triggering that branch. This way coverage-guidance in the Forest can still improve the fuzzing performance, but coverage in branches of the forest that do not influence the capability to trigger the target branch (a very small adversarial example) will distract the fuzzer less. The potential use-case for a directed fuzzer is also why the if-branches are nested. The if-branches do not necessarily have to be nested: for normal fuzzers the same result will be achieved by just putting the if-statements below each other on the same level. However, the nesting can tell a directed fuzzer an accurate basic-block distance from the current best known adversarial example to the target branch. This structure is thus favored for future tranferability to a directed fuzzer. We run all of our experiment for L_∞ distance, but the structure of the objective function works for any kind of monotonically decreasing distance measure: if the distance is lower than previous inputs the input will be saved and improved upon. FATE is extensible for different distance measures.

Saving all adversarial examples the fuzzer discovers can put too much stress on the file-system though and may become a bottleneck. adversarial examples are thus only saved if their distance to the victim (rounded) has not been saved before. This can be easily verified by making the filename of an adversarial example deterministic. By performing a quick (this can easily be executed more than a million times per second on a laptop) `stat` call (requires a POSIX-compliant system) in the `write_if_not_exists` function the target knows whether to perform the more expensive file-write or not. Our objective function thus keeps expensive file-write operations at a minimum while only examples that improve the best known adversarial example will saved to the corpus. Other layouts for the objective are also possible, which we describe and experiment with in chapter 4.

The objective function as specified above acts on one side of the decision surface: when an adversarial example is found. But when finding an adversarial example is difficult, the fuzzer will spend a lot of time performing a random search until the first adversarial example is found. To solve this, an extra descent can be added based on the probability of input belonging to the original class. The rationale behind this is that when the probability of the input belonging to the original class decreases, the probability of the input belonging to one of the other classes will per definition increase. This will thus move the fuzzer towards finding an adversarial example. Pseudocode of the descent is shown in Figure 3.5. The descent acts on the `else` branch of the statement `if (fuzzed_class != original_class)` of the objective function from Figure 3.4. A potential drawback of this approach is that it does not take into account the distance between the current seed and the victim, which means seeds with very bad adversarial distance may be saved by the fuzzer. Again, a dummy variable (printed in a piece of code the fuzzer will never reach) is referenced in each branch to make sure the compiler will not optimize the structure of the branches. Different approaches will be tested and described in chapter 5.

```
void check(double fuzzed_features[],
           double original_features[],
           int fuzzed_class,
           double proba_original) {
  // ...
  else {
    if (proba_original <= 1.0) {
      print_this = 0;
      if (proba_original < 0.95) {
        print_this = 1;
        // ...
        if (proba_original < 0.01) {
            // Something that will not
            // be compiled away
            print_this = 6;
            write_if_not_exist("p0.txt")
        }
      }
    }
  }
}
```

Figure 3.5: Descent on probability of the sample belonging to the original class

In conclusion, the objective function will assist the fuzzer into finding adversarial examples, but it cannot guarantee that it will find an adversarial example as the fuzzer still depends on random mutations. The fuzzer is thus sound, but incomplete. The objective function can be easily modified or changed to test the model for different objectives. It can for example be changed to do a targeted attack by changing `if (fuzzed_class != original_class)` to `if (fuzzed_class == target_class)`.

3.5. Processing results

Adversarial examples identified by the fuzzer are saved to files, which are processed to provide statistics about their quality afterwards. For each victim, the best adversarial example is selected based on L_∞ distance. After each fuzzing run, basic block coverage is measured through SanitizerCoverage feedback (The clang/LLVM method of providing coverage information, also used by libFuzzer) and saved. We verified that the C++ target and the scikit-learn model produce the same predictions. It turns out that very rarely (generally $\ll$ 1% of all examples) an example that is found is not adversarial due to floating-point inconsistencies between the scikit-learn model and the fuzzing target. We allow these floating-point misclassifications as it must mean that an adversarial example can be created by changing one or more features by a negligible amount. We also allow these float misclassifications for the methods we are comparing against.

4 Fuzzing for adversarial examples

In this chapter we verify that with FATE as introduced in chapter 3 adversarial examples can be found for most victims on a multitude of datasets used in related work. We identify a good basic setup for the fuzzer, such as what the initial seeds should be. Grey-box fuzzers act like Genetical Algorithms, for which it is typical to require hyper-parameter tuning. We thus identify the default parameters for settings such as mutation depth.

4.1. Experimental setup

All experiments are conducted on a HP Zbook studio G5 x360 laptop with an Intel i7-8750H CPU @ 2.20GHz (6 cores, 12 threads). Unless mentioned otherwise, experiments are repeated 5 times for the same 50 victims per dataset. Victims are only attacked if they are correctly classified by the model: as models often do not have 100% accuracy, some testing set samples are misclassified with their original features which would make them an adversarial example of itself. Repeating the experiments is important to attribute for OS noise and randomness in the fuzzer. To reduce the influence of OS noise even more, 10 threads are used such that 2 threads stay available to handle general system tasks. The datasets are introduced in Appendix C. They are selected based on their occurrence in related work [11, 48] and vary in the size of their respective models, number of features and number of classes. An overview of the training procedure for the models is shown in Appendix A.

Most experiments are deducted on the Gradient Boosting models only. We show the performance of the Random Forest models only for the final setup in chapter 7 to limit the number of experiments that needs to be executed. Multiple victims are attacked concurrently on m threads. When a victim is fuzzed for s seconds, the average execution time shown is calculated as $\frac{s}{m}$ as in s seconds m victims can be attacked simultaneously. In the tables showing the experimental results in this and the following chapters, $\bar{r}$ shows the average L_∞ norm adversarial distance over the victims that were attacked, with a lower $\bar{r}$ meaning that on average better adversarial examples are found. The value reported is the average adversarial distance over 5 runs. n shows the number of victims that could be attacked, again averaged over 5 runs.

In this chapter we will tweak basic settings of FATE. We start using a baseline with sensible default values:

- Mutation chance: 0.1 (to keep most traits of an input)
- Mutation depth: 5 (default of libFuzzer)
- Epsilon radius: 0.5 (a trade-off between being able to attack many victims but with not too large adversarial distances)
- Custom Mutation and crossover functions as described in chapter 3
- Corpus initialization with the victim features only
- Gaussian mutation distribution
- Distance intervals of 0.005 for the objective function for fine-grained directedness in this function.

All datasets are fuzzed for 2 seconds per victim except MNIST

Table 4.1: Baseline performance

	Baseline	
Dataset	$\bar{r}$	n
Breast-cancer	0.2244	50
Diabetes	0.095	50
IJCNN1	0.1756	50
Covertype	0.2229	50
Higgs	0.1563	50
MNIST 2 vs 6	0.5113	22.8
MNIST	0.538	38.2
FMNIST	0.5537	20.2
Average	0.3097	41.4

and MNIST 2 vs 6 which are fuzzed for 10 seconds due to their higher difficulty and larger size. The performance of the baseline is shown in Table 4.1.

4.2. Objective function

As explained in chapter 3, the objective function plays a role in directing the fuzzer towards good adversarial examples. The layout and performance of this function are thus important for the performance of FATE. We experiment with three different layouts:

Listing 4.1 shows "Include else branches" where `else` branches are included in the distance descent. A drawback of this structure is that when an adversarial example is found with distance < 0.005, but not yet within 0.015 < distance < 0.01, the fuzzer will try to find an adversarial example within that range, even though we have already found a better adversarial example.

Listing 4.2 shows the "Write in each branch" layout where the redundant `else` branches are not present, but the drawback now is that `write_if_not_exist` is called in each branch, potentially forming a bottleneck for the execution speed.

Listing 4.3 shows "Only write if better"; the best of both worlds, without the redundant `else` branches and with only one call to `write_if_not_exist`. This call is optimized further by only calling this function if the previous best adversarial example is improved upon by the current input. This is possible through saving the current best adversarial example distance in memory which is initialized before the fuzzer calls the `LLVMFuzzerTestOneInput` function as described in chapter 3.

Looking at the results in Table 4.2, it is clear that "Write in each branch" performs much worse than the other two layouts. Upon inspecting the results, this layout takes a big performance hit compared to the other two layouts with only 80.000 executions per second for the Breast-cancer dataset. The other two layouts perform similar, but as the "Only write if better" layout has higher executions per second (380.000) compared to the "Include else branches" (300.000), and the fuzzer will not spend its time on unimportant branches in the objective function, the "Only write if better" layout should thus be used.

Through increasing or decreasing the stepsize between the branches (0.005 in the previous examples), one can make a trade-off between the number of branches (and thus the amount of seeds in the corpus) and the accuracy at which new branches are triggered for better adversarial examples. The experiments in Table B.3 (Appendix) show that a different step size does not impact the performance significantly. We also experiment with different intervals, starting from distance 0.8 instead of 1.0 with increasing accuracy. For example, the "distance decreasing ++" setup shown in Table B.4 (Appendix) uses the intervals [0.8, 0.7, 0.6, 0.59, ..., 0.25, 0.249, ..., 0.002, 0.0019, ..., 0.0001]. Diabetes is the only dataset that seems to benefit from this decreasing step size, so we keep the distance intervals at 0.005. This has the added benefit that the accuracy for finding adversarial examples is equal throughout the distance range [1, 0], thus working equally well for models that have higher average adversarial distance. We add an extra branch 0.000001 as last step, such that the fuzzer will keep trying to trigger new branches in the objective function for datasets which have very small adversarial distances, such as Higgs.

Table 4.2: Different layouts of the objective function

	Include else Branches		**Write in each Branch**		**Only write If better**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.1579	42.2	0.2508	42.8	0.1523	42
Diabetes	0.0776	50	0.177	50	0.0739	50
IJCNN1	0.0839	50	0.1509	50	0.086	50
Covertype	0.1085	50	0.1718	50	0.1069	50
Higgs	0.069	50	0.123	50	0.069	50
MNIST 2 vs 6	0.259	21	0.3138	21.2	0.2595	21
MNIST	0.2865	46.4	0.335	46.4	0.289	46.8
FMNIST	0.2877	44.2	0.3431	44.8	0.2953	45
Average	0.1663	44.2	0.2332	44.4	0.1665	44.4

```
distance = calculate_distance()
...
if (distance < 0.01) {
    if (distance < 0.005) {
        write_if_not_exist(0.005,
            fuzzed_features);
    } else {
        write_if_not_exist(0.01,
            fuzzed_features);
    }
} else {
   write_if_not_exist(0.015,
    fuzzed_features);
}
...
```

Listing 4.1: Objective function with else branches

```
distance = calculate_distance()
...
if (distance < 0.015) {
    write_if_not_exist(0.015,
        fuzzed_features);
    if (distance < 0.01) {
        write_if_not_exist(0.01,
        fuzzed_features);
        if (distance < 0.005) {
            write_if_not_exist
               (0.005,
                fuzzed_features);
        }
    }
}
```

Listing 4.2: Objective function that writes adversarial examples in each branch

```
distance = calculate_distance()
...
if (distance < 0.01) {
  dummy_var = 1;
  if (distance < 0.005) {
    dummy_var = 2;
  }
}
if (distance < best_ae_dist) {
  best_ae_dist = distance;
  dist_interval = round(distance, 5);
  write_if_not_exist(dist_interval,
      fuzzed_features);
}
```

Listing 4.3: Only write if better

For some datasets it is difficult for the fuzzer to find adversarial examples. To counter this we add the descent from Figure 3.5 based on class-output probabilities. The fuzzer reaches this descent when the current seed is not an adversarial example. By looking for input for which the model is less sure that the input belongs to the original class of the victim, the fuzzer will come closer to an adversarial example: if the probability of the current input belonging to the original class is sufficiently low, an adversarial example will eventually be found (the sum of class probabilities is always 1). This climbs the other side of the decision surface: from the side where input is not an adversarial example.

Only looking at a decreasing probability of the input belonging to the original class does however not take into account the adversarial distance of such an example, which may become very large. In Listing 4.5 we add the class output probabilities to the adversarial distance to make the distance-descent dependent on the probability and distance at the same time.

When the model receives input that is not similar to input it was trained on, the model will generally be less sure what to predict. Looking at Figure 7.3, adversarial examples are often samples that are not similar to samples from the training set, as the decision regions around samples from the training set should be evaded. The model will often be less sure about the right prediction for adversarial examples than for normal samples, as it did not have a chance to train on similar examples. We are thus looking for input for which the fuzzer predicts another class, but the class probabilities of the input belonging to the original class and the input belonging to the adversarial class should be close to each other. In Listing 4.6 we show an approach that only looks at the difference between the two largest class probabilities. If this difference is small enough (optionally also taking into account the adversarial distance), the seed is saved.

```
...
else {
  if (proba_original <= 1.0) {
    print_this = 0;
    if (proba_original < 0.95) {
      print_this = 1;
      // ...
      if (proba_original < 0.01) {
        // Something that will not
        // be compiled away
        print_this = 6;
        write_if_not_exist("p0.txt");
      }
    }
  }
}
```

Listing 4.4: Probability steps

```
distance = calculate_distance();
proba_original = probabilities[
    original_class];
distance += proba_original;
if (original_class !=
    fuzzed_class) {
  if (distance < 1.0) {
  dummy_var = 0;
    if (distance < 0.995) {
      dummy_var = 1;
      ...
    }
  }
}
```

Listing 4.5: Combine distance and probability

```
probabilities.sort("descending")
if (distance < ...) { // either 1.0 or epsilon
  if (probs[0] - probs[1] < 0.2) {
    print_this = 0;
    ...
    if (probs[0] - probs[1] < 0.01) {
      print_this = 7;
    }
  }
}
```

Listing 4.6: Catch input close to resembling an adversarial example

The results of the approaches for taking into account class probabilities are shown in Table 4.3. To increase the influence of the probability steps, coverage was enabled for only the objective function for this experiment. Seeds are thus only saved if new coverage is triggered in the objective function. Combining distance and probability clearly performs worse than not using probability information. The probability descent makes it easier to attack the image datasets while not increasing the adversarial distances for the other datasets too much. When starting this descent from 0.5 instead of 1.0, thus not taking into account high original class probabilities, the added value of the probability descent of being able to attack more victims for the image datasets mostly disappears. However, intuitively, seeds for which the original class probability is high are not valuable to be saved by the fuzzer as they will not be close to resembling an adversarial example. It may be the case that saving these seeds helps in exploring more leaves of the forest thus enabling the fuzzer to attack more victims. Another explanation is that it is difficult for the fuzzer to find original class probabilities lower than 0.5 for the victims that are difficult to attack, thus performing better when seeds with higher probabilities of belonging to the original class are saved as well.

Adding the branches that act on small probability differences mainly improves the amount of victims that can be attacked for the Breast-cancer dataset. Limiting these branches to a distance within epsilon generally works best. Using these branches within limited epsilon distance is the best trade-off between adversarial distance and the amount of victims that can be attacked.

4.3. Fuzzer configuration

There are many settings and opportunities for custom implementations in fuzzers such as libFuzzer. Through experimentation we define an acceptable baseline for the fuzzer configuration.

In Table 4.4 experimental results for different fuzzer configurations are shown. For the "empty seed" configuration we initialize the corpus with one seed containing the value 0.0 for each feature, as at least one input is required by the fuzzer and the mutator expects input with the correct length. The fuzzer shows clearly worse distances than the baseline for this configuration, while finding adversarial examples for almost

Table 4.3: Taking into account soft-label output

Dataset	No probability Branches $\bar{r}$	No probability Branches n	Combine distance And probability $\bar{r}$	Combine distance And probability n	Probability Descent $\bar{r}$	Probability Descent n	Probability descent From 0.5 $\bar{r}$	Probability descent From 0.5 n	Small proba Difference $\bar{r}$	Small proba Difference n	Small proba Difference limited $\bar{r}$	Small proba Difference limited n
Breast-cancer	0.1272	20.4	0.1343	21.4	0.1556	36.6	0.1248	20.4	0.1562	34.8	0.1552	36.6
Diabetes	0.0644	49.4	0.1109	49.4	0.0648	50	0.0636	49.4	0.0637	49	0.0629	49.4
IJCNN1	0.0529	50	0.0821	50	0.0512	50	0.0518	50	0.0534	49.8	0.0515	50
Covertype	0.0614	48.4	0.1042	48.2	0.068	50	0.0624	48.2	0.0637	48.6	0.0635	48.6
Higgs	0.0223	50	0.062	50	0.0233	50	0.0227	50	0.0263	50	0.0226	50
MNIST 2 vs 6	0.1797	22.8	-	-	0.2151	30.4	0.1794	22.4	0.1813	22.4	0.1738	22.2
MNIST	0.1948	43.4	-	-	0.2264	48.2	0.2084	43	0.2061	43.8	0.1911	43.8
FMNIST	0.2105	44.4	-	-	0.2338	48.2	0.2275	44.8	0.2191	45	0.2066	44.4
Average	0.1142	41.1	0.0987	43.8	0.1298	45.4	0.1176	41.0	0.1212	42.9	0.1159	43.1

all victims. The 0 initialization is probably either already an adversarial example by itself (although a very bad one) or very close to an adversarial example. This explains why almost all victims can be attacked with high distances for the image datasets, as often a pixel that would otherwise be white is now completely 0 (black).

For "no custom mutate", the built-in libFuzzer mutator was tested with the rest of the code being equal to the baseline. The built-in mutator mutates bytes of the input (instead of features), which results in input that can be far outside the original input domain [0, 1]. Most input will thus be treated as either 0.0 or 1.0 by the forest. Although the fuzzer is still able to find some adversarial examples, they are clearly worse and not all victims can be attacked, even for the easier datasets.

For "Uniform random" we change the Gaussian distribution ($\mu = 0, \sigma = 0.34$) to a uniform random distribution on the interval [-1, 1]. This results in a larger mutation on average, which we can see back in higher adversarial distances, although also more victims are attacked for the image datasets.

For "No crossover" we remove the crossover capabilities from the fuzzer, so it can only mutate and not combine seeds. This changes the fuzzing process from a Genetic Algorithm to an Evolutionary Algorithm approach. This generally produces higher distances compared to the baseline, although again it is easier to attack the image datasets. Apparently the unnatural looking images that crossover will generate confuse the fuzzer.

We keep the baseline configuration, as the other configurations yielded no (clear) improvements in both adversarial distance and number of victims attacked.

Table 4.4: Different fuzzer configurations

Dataset	Baseline $\bar{r}$	Baseline n	Empty seed $\bar{r}$	Empty seed n	No custom mutate $\bar{r}$	No custom mutate n	Uniform random $\bar{r}$	Uniform random n	No crossover $\bar{r}$	No crossover n
Breast-cancer	0.2244	50	0.2547	50	0.5417	35.8	0.2318	50	0.2202	50
Diabetes	0.095	50	0.1094	50	0.3712	45.6	0.1039	50	0.0852	50
IJCNN1	0.1756	50	0.4317	50	0.65	48.4	0.2889	50	0.1998	50
Covertype	0.2229	50	0.6487	50	0.9522	25.8	0.372	50	0.2579	50
Higgs	0.1563	50	0.7258	50	0.6436	25.8	0.2535	50	0.1781	50
MNIST 2 vs 6	0.5113	22.8	0.9957	50	0.9993	9.4	0.7949	33.6	0.5726	25.2
MNIST	0.538	38.2	0.9964	50	0.9992	20	0.7492	45.8	0.5658	44.6
FMNIST	0.5537	20.2	0.9668	47.8	0.9756	13.8	0.7667	32.4	0.6048	32.4

4.3.1. Custom mutator

The custom mutator is important for supplying the fuzzer with domain knowledge. We discuss settings which are generally defined for such mutators.

Each feature is mutated with a certain chance. This chance can greatly influence fuzzing performance as can be seen in Table 4.5. We can see a trade-off between average adversarial distance and the ability to attack victims for the image datasets: Average distances are lowest for the smallest mutation chance, while for the highest mutation chance the most victims can be attacked for the image datasets. Apart from the easy Breast-cancer and Diabetes datasets the average distance rises drastically for higher mutation chances. Using mutation chance 0.1 seems a good default trade-off between distance and the average number of victims that is attacked, but ideally this chance should be set to a higher value when attacking image datasets and a lower value for the other datasets (except Breast-cancer). A possibility to determine a good value for the mutation chance for a certain dataset would be to try multiple values for the mutation chance on a subset of all victims before starting the actual execution of FATE. For now we use 0.5 for Breast-cancer and the image datasets and 0.05 for the other datasets.

Table 4.5: Mutation chance

	0.05		**0.1**		**0.2**		**0.3**		**0.5**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2294	50	0.2244	50	**0.2224**	50	0.2229	50	0.223	50
Diabetes	0.0994	50	0.095	50	**0.0913**	50	0.0923	50	0.0943	50
IJCNN1	**0.1536**	50	0.1756	50	0.2113	50	0.2413	50	0.2849	50
Covertype	**0.1847**	50	0.2229	50	0.2719	50	0.3011	50	0.3438	50
Higgs	**0.1344**	50	0.1563	50	0.1891	50	0.2146	50	0.2596	50
MNIST 2 vs 6	**0.4923**	21.4	0.5113	22.8	0.6007	24.8	0.6537	26.6	0.7013	**28**
MNIST	**0.4701**	18	0.538	38.2	0.5885	47.6	0.6143	48.4	0.6562	**49.8**
FMNIST	**0.5067**	9.2	0.5537	20.2	0.6199	39.4	0.6395	44.4	0.6612	**46**
Average	0.283825	37.325	0.30965	41.4	0.3493875	45.225	0.3724625	46.175	0.4030375	46.725

Another option is to set the mutation depth, the maximum number of consecutive mutations before feeding the input to the target. This changes the potential neighborhood of an input seed. By default this is set to "5" for libFuzzer, which means that randomly 1 to 5 mutations are performed. Table 4.6 shows results for different mutation depths. The best result per dataset is shown in bold. For IJCNN1, Covertype and Higgs a mutation depth of 1 seems to provide a little better results than higher mutation depths, but for the other datasets higher numbers of mutations are better. The differences in $\bar{r}$ and n are often small, and may very well be contributed to the randomness of fuzzers. We decided to keep the mutation depth on the default value of "5".

Table 4.6: Mutation depth

	Depth 1		**Depth 3**		**Depth 5**		**Depth 7**		**Depth 10**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2317	50	0.2255	50	0.2244	50	**0.2218**	50	0.2226	50
Diabetes	0.097	50	0.0921	50	0.095	50	**0.0879**	50	0.0891	50
IJCNN1	**0.1594**	50	0.1742	50	0.1756	50	0.1742	50	0.177	50
Covertype	**0.21**	50	0.2154	50	0.2229	50	0.2261	50	0.2301	50
Higgs	**0.1489**	50	0.153	50	0.1563	50	0.1535	50	0.1569	50
MNIST 2 vs 6	0.5361	24.2	0.5445	25.8	**0.5113**	22.8	0.5562	26	0.5436	24.8
MNIST	0.5378	39.8	0.5521	38.8	0.538	38.2	0.5527	38.8	**0.5366**	36
FMNIST	**0.5433**	19.6	0.5525	21.8	0.5537	20.2	0.5698	21.2	0.5682	20.6

In FATE we allow to restrict the size of mutations using an ϵ value. All mutations are scaled by this ϵ, such that mutations lie in a ball with radius ϵ around the input s that is currently being mutated. For example when $\epsilon = 0.2$, only adversarial examples that have L_∞ distance < 0.2 to s can be reached within this mutation. Setting $\epsilon = 1.0$ removes this restriction. Smaller values of ϵ thus make the neighbourhood, all possible inputs that can be "reached" when mutating s, smaller. This increases the similarity between s and its mutations. Results for different epsilon values are shown in Table 4.7. Again we can see a trade-off between average adversarial distance and the number of victims that can be attacked. adversarial examples with smaller distances are found for smaller ϵ values, while more victims can be attacked (mainly for the image datasets) with higher epsilon values. The breast-cancer model performs significantly worse for smaller epsilon values, both on distance and the amount of victims attacked. Due to the small model size for this dataset, probably no other leaves of the forest can be reached for some victims within the smaller epsilon ranges, which means that they cannot be attacked. Manual investigation also shows that optimal adversarial examples can have a distance as much as 0.6 for Breast-cancer. It is very difficult or impossible to reach such adversarial examples by continuously mutating with a Gaussian distribution in neighbourhoods with epsilon radius 0.05 or 0.1. $\epsilon = 0.2$ seems the best trade-off between the average adversarial distance and number of victims attacked. However, like for the mutation chance hyper-parameter, the epsilon value should ideally be tweaked per dataset that is being attacked.

Table 4.7: Different ϵ values

	0.05		**0.1**		**0.2**		**0.3**		**0.5**		**0.7**		**1.0**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.6313	21	0.3809	38.6	0.2855	50	0.2576	50	**0.2266**	50	0.2289	50	0.2378	50
Diabetes	**0.067**	50	0.0711	50	0.0788	50	0.0851	50	0.0962	50	0.1052	50	0.1132	50
IJCNN1	**0.0698**	50	0.0646	50	0.0902	50	0.1196	50	0.1706	50	0.2248	50	0.2809	50
Covertype	**0.0526**	48.2	0.0745	49.6	0.1101	50	0.1487	50	0.2258	50	0.3034	50	0.3995	50
Higgs	**0.0255**	50	0.043	50	0.0749	50	0.1031	50	0.1565	50	0.1977	50	0.2554	50
MNIST 2 vs 6	**0.0851**	3.2	0.1514	16.2	0.2415	20.4	0.3272	21	0.5204	22.8	0.713	28.6	0.8971	**30.8**
MNIST	**0.0591**	7	0.1127	10.8	0.2275	21	0.3421	29.4	0.5388	38	0.7134	43	0.9153	**47.6**
FMNIST	**0.0616**	4	0.1197	6.4	0.2314	9.8	0.3421	12.8	0.5554	19.6	0.7167	26.4	0.909	**34.2**
Average	0.1315	29.2	**0.1272**	34.0	0.1675	37.7	0.2157	39.2	0.3113	41.3	0.4004	43.5	0.5010	**45.3**

The previous results have shown that generally we need small mutations for small distances, but sometimes we need a large mutation to be able to attack the victim at all. We try to solve this problem by squaring the draw from the Gaussian distribution, keeping the sign: if the draw would originally be -0.08, we change this to $-0.08 * -0.08 = 0.0064$, but keeping the minus sign of the original mutation such that the new mutation becomes -0.0064. Larger mutations can now still happen, only with a much smaller likelihood. We show the results for different ϵ values in Table 4.8. None of these results are better than the results with $\epsilon = 0.2$ from Table 4.7, so this option will not be used.

Table 4.8: Different ϵ values combined with a steeper mutation curve

	0.2		**0.5**		**1.0**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2797	50	**0.2176**	50	0.2214	50
Diabetes	**0.0736**	50	0.0809	50	0.0876	50
IJCNN1	**0.0684**	50	0.1025	50	0.168	50
Covertype	**0.0838**	50	0.1419	50	0.2412	50
Higgs	**0.0456**	50	0.088	50	0.1364	50
MNIST 2 vs 6	**0.2323**	12.8	0.4638	21.8	0.751	**24.8**
MNIST	**0.1711**	9	0.4635	18.2	0.8234	**32**
FMNIST	**0.2061**	3.6	0.4958	9.4	0.8105	**9.4**
Average	**0.1451**	34.4	0.2568	37.4	0.4049	**39.5**

4.4. Compiler optimization

To improve the speed with which the fuzzer can execute the target, we experiment with always enabling the optimization options of the compiler through the "-O3" option instead of not allowing optimization ("-O0"). The reason not to use optimization options of the compiler by default is that compilation takes longer if optimization is enabled as can be clearly seen in Table 4.9 in the t_c columns: compilation takes on average for about 4 times longer than without optimization enabled. As expected, the executions per second are higher when optimization is enabled, which mainly makes a big difference in the number of executions per second for the image datasets. This results in a small number of extra victims that can be attacked. The difference in the quality of the adversarial examples found is not big, but if the compilation time is not an issue enabling optimization should improve the performance of FATE a bit through higher execution speeds. If runtime is of importance though, the optimization option should not be used. We did a best-effort to verify the branches in the objective function are not compiled away when the optimization option is used, but this required inserting print statements which of course can also make the compiler not optimize away the branches. However, as a variable is set in the branches of the objective function that is printed after the branches are visited (see section D.6), the compiler should not optimize this structure.

Table 4.9: Difference between always/never optimize

	Never optimize				**Always optimize**			
Dataset	$\bar{r}$	n	exec/s	t_c (s)	$\bar{r}$	n	exec/s	t_c (s)
Breast-cancer	0.2870	50	90985	1.0	0.2942	50	105339	1.9
Diabetes	0.0727	50	63223	1.2	0.0713	50	85723	2.1
IJCNN1	0.0704	50	22075	2.1	0.0665	50	25502	7.2
Covertype	0.0978	50	7500	2.9	0.0949	50	9774	10.0
Higgs	0.0588	50	999	5.0	0.0589	50	1147	22.1
MNIST 2 vs 6	0.2337	21.6	663	3.0	0.2248	21.2	1115	9.7
MNIST	0.2521	39.4	9.5	48.7	0.2486	41.4	17.0	235.5
FMNIST	0.2691	39.6	9.1	54.2	0.2624	43.6	17.9	244.0
Average	0.1688	43.8	21638	14.8	0.1668	44.5	26509	66.6

4.5. Conclusion

We showed that FATE is able to find adversarial examples with reasonable average adversarial distances on datasets used in related work. We verified the design of the objective function of FATE. The search for adversarial examples was extended by including a search on the non-adversarial example side of the decision function through the use of class-output probabilities. The custom mutator is important for the performance of FATE, with the mutation_chance and ϵ hyper-parameters strongly influencing the quality and quantity of adversarial examples. A mutation chance of 0.5 for the image datasets and 0.1 for the other datasets was chosen as a default, and ϵ is set to 0.2. We also enable compiler optimization to achieve a higher number of executions per second. The performance of the new baseline is shown in Table 4.10. The performance of this new baseline is better on all datasets except for Breast-cancer, where the adversarial distance is lower but not all victims could be attacked.

Table 4.10: Performance after defining the basic setup for FATE

	New Baseline		**Old Baseline**	
Dataset	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.1511	46	0.2244	50
Diabetes	0.0652	50	0.095	50
IJCNN1	0.0703	50	0.1756	50
Covertype	0.085	50	0.2229	50
Higgs	0.0482	50	0.1563	50
MNIST 2 vs 6	0.2238	21.4	0.5113	22.8
MNIST	0.2854	46.4	0.538	38.2
FMNIST	0.2917	45.2	0.5537	20.2
Average	0.1526	44.9	0.3097	41.4

5 Improving FATE by fuzzing smarter

We identify three possible sources of extra information with which we can potentially improve the performance of FATE: the training set, the model itself, and information from single executions of the fuzzer. Through experimentation we identify which of this information is usable and we show the performance improvements of FATE when using this information.

5.1. Sources of information

In this section we briefly identify possible sources of information and how they can be used. The different possible ways to use this information are further explained later in this chapter.

The **Training set** gives information about benign samples with which the model is trained. It contains samples of points of the same class as the victim, but also points that belong to other classes. This means we can initialize the fuzzer with adversarial examples: every point (correctly classified as) belonging to another class will already be an adversarial example. Statistics and heuristics can be calculated for these subsets: for example we can calculate the average member of each adversarial class. As we can only assume access to the training set, the testing set cannot be used for this type of information.

The **model** is the most fundamental source of information as we are fuzzing it directly. Because of the white-box access model we use, we have full access to all information contained in the model. In the model itself there are splitting conditions at each non-leaf node we call thresholds (`if (feature <= threshold)`). The values of these thresholds determine the borders of bounding boxes around decision regions. Mutating towards a value right at this border will produce the best possible adversarial example within that decision region. Some training artifacts are contained in the model as well, for example how important each feature is for classifying the input correctly based on impurity during training. An alternative for measuring feature importance is counting the feature occurrence: how often each feature is used in a splitting condition.

Another artifact from training is the number of training samples in each leaf, which could potentially be used as a measure of the importance of a leaf. It is out of the scope of this project to identify individual leaves to target, so we do not use this information. The model also gives information about its layout, size and the depth of each leaf but we do not see a clear way to leverage this kind of information. Lastly, the models allows us to inspect the decision path followed to classify an input example, potentially enabling us to only enable coverage-guidance in nodes close (in feature distance) to the path followed when classifying the victim. As coverage information is inserted at compile-time, this would require us to compile the target for each victim, which would be very time-consuming. We thus choose not to use this information.

Lastly, information about the current **execution** can be recorded. For example, when mutating we can take into account the differences in features between the current input and the victim, allowing to influence which features are mutated and with which magnitude, should the current input be an adversarial example. Also, previously found adversarial examples for similar victims can be used to initialise future runs of the fuzzer. We decided to opt for the simpler approach to initialize with adversarial examples through variations on nearest neighbours of other classes in the training set, as this can be computed beforehand which simplifies the execution process and decreases the execution time of the fuzzer. The fuzzer also gives heuristics about an execution such as the reached coverage, and can produce detailed coverage information for single runs. This information could be used in the runs after the first victim is attacked, but it can be very complex to use this information. For example, we would need to know which branches/leaves are actually important for finding adversarial examples. Also, this would require a lot of processing in FATE between fuzzing runs which would make the management of the execution more complex.

5.2. Training set

A seed is a file consisting of bytes that resemble "interesting" inputs to the fuzzer. The performance of individual fuzzing runs can differ a lot based on the initial seeds used [25, 28]. By default we only provide the victim point as an input to the fuzzer. This way the fuzzer knows the original path through the tree ensemble and will search for adversarial examples using the victim features as a starting point. At the start the fuzzer will thus explore leaves relatively close to the original path through the forest. The best adversarial example will look much like the victim, and by providing the victim to the fuzzer the fuzzer is already close to the optimal solution.

Table 5.1: Initial adversarial example quality with different distance measures

Dataset	L_1	L_2
Breast-cancer	0.8667	0.7800
Diabetes	0.3541	0.2734
IJCNN1	0.3286	0.2783
Covertype	0.5183	0.4711
Higgs	0.6844	0.5546
MNIST 2 vs 6	1.0	1.0
MNIST	1.0	1.0
FMNIST	0.9954	0.9932

We develop several other corpus initialization strategies using the samples from the training set. They allow the fuzzer to be initialized with an adversarial example, instead of having to look for one using random mutation until a first adversarial example is found. The fuzzer then thus starts in the distance-descent of the objective function right away. Results for the following strategies are shown in Table 5.2.

1. Providing the k approximate closest points from other classes: k **AE**. This way the fuzzer already starts with some (hopefully good) adversarial examples and can spend its time on improving those. Computing the exact k closest instances is computationally expensive due to quadratic complexity. We thus use the k-ANN (Approximate Nearest Neighbour) implementation Annoy ([37]) to find the approximate closest neighbours. This strategy guarantees that the fuzzer is always initialized with a true adversarial example. However, the model may predict this adversarial example wrongly thus we cannot guarantee that an adversarial example will always be identified by the fuzzer. We test with $k \in \{1, 5, 10\}$. Annoy does not support L_∞ distance, however they do support L_0, L_1 and L_2. For the image datasets the distance between two features is often 1, as often a pixel will either be filled (1) or not (0). For the image datasets we thus use L_0 distance (hamming distance) and argue that the images with the least different pixels will look most similar. For the other datasets, we experimented with both L_1 and L_2, both of which produced similar results for the adversarial examples FATE finds. However, as can be seen in Table 5.1, the L_2 distance produces better initial adversarial examples (distances noted with the L_∞ norm). As the L_2 distance also resembles L_∞ distance the most, we use the L_2 distance. Producing the ANN lookup also takes time as shown in Table 7.4 (Lookup init, next chapter). The time it takes to initialize the lookup mainly depends on the size (number of features) and amount of samples, both of which are big for the image datasets. The time it takes to initialize the lookup for the image datasets is still reasonable however, and for the other datasets it is negligible.

2. Providing all training points of opposite classes: "**Full train set**". This way the fuzzer knows right from the start how to reach a lot of the leaves in the tree ensemble that predict other classes. We limit this to 5000 instances, to not overload the corpus.

3. Providing the average adversarial example: "**Average AE**". For the binary classification datasets we calculate the average value a of the samples from the opposite class. Note that a is not guaranteed to be classified as an adversarial example by the model.

4. "**Double fuzz**". This is a hybrid approach where the fuzzing is started without any adversarial examples and if the fuzzer is able to find no adversarial example for a victim, which mainly happens for the image datasets, the victim is fuzzed again but now initialized with an adversarial example, guaranteeing that the victim will be attacked.

Table 5.2: Initialization with adversarial examples

	Baseline		1 AE		5 AE		10 AE		Full train set		Double Fuzz		Average AE	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2841	50	0.2813	50	0.2327	50	0.2336	50	0.2367	50	0.2867	50	0.2542	50
Diabetes	0.0697	50	0.0745	50	0.0735	50	0.0745	50	0.0768	50	0.072	50	0.0722	50
IJCNN1	0.0713	50	0.0722	50	0.0757	50	0.0773	50	0.1184	50	0.0717	50	0.0866	50
Covertype	0.0844	50	0.088	50	0.096	50	0.1014	50	0.2326	50	0.0866	50	0.1039	50
Higgs	0.0562	50	0.0639	50	0.0861	50	0.0943	50	0.3213	50	0.0548	50	0.0652	50
MNIST 2 vs 6	0.259	21.8	0.7195	50	0.7915	50	0.8182	50	0.9589	50	0.6834	50	0.6533	50
MNIST	0.2781	37.6	0.7377	50	0.8307	50	0.9011	50	-	-	0.5018	50	-	-
FMNIST	0.2925	36	0.669	50	0.7075	50	0.7847	50	-	-	0.5235	50	-	-
Average	0.1744	43.2	0.3383	50	0.3617	50	0.3856	50	-	-	0.2851	50	-	-

For "k AE" we see that initializing the fuzzer with adversarial examples is beneficial for the image datasets where no adversarial examples can be found for many victims in the baseline setup. The more adversarial examples are fed to the fuzzer though, the worse the average distance gets for all datasets.

Due to the slow execution speed of (F)MNIST (see chapter 4), it took very long to initialize the fuzzer with the full training set of the opposite class for these datasets. Those runs were thus canceled and this setting is deemed infeasible. Providing the full training set also yielded clearly worse results than the baseline for the other datasets.

Providing the average adversarial example also decreased performance for most datasets. Double fuzz is the only feasible option, providing baseline performance when adversarial examples can be found and having the benefit of finding adversarial examples for each victim. This does take more time though, as some victims are now fuzzed twice.

Feeding more adversarial examples to the fuzzer generally made average distance much worse. This may be the case because decision regions around samples in the training set are generally well-defined, as the model was trained using those samples. The fuzzer may now thus get distracted by seeds that are not worth the time investment. We thus try to improve the adversarial example: instead of only providing an adversarial example a we also provide the point exactly in between a and the victim v: $a_1 = \frac{a+v}{2}$. This will either be a better adversarial example or be an example that is closer than the victim to the decision region of the nearest training sample of another class. Table 5.3 shows the results of this approach. "5 AE +" means that 5 approximate Nearest Neighbours are identified, and that they are used, together with the 5 artificial points and the victim, as initial seeds.

Table 5.3: Adversarial example initialization with an artificial point

	Baseline		1 AE +		5 AE +		10 AE +		Double Fuzz 10 AE+	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2841	50	0.2831	50	0.2314	50	0.2342	50	0.2845	50
Diabetes	0.0697	50	0.0709	50	0.0715	50	0.0716	50	0.0688	50
IJCNN1	0.0713	50	0.0679	50	0.0663	50	0.068	50	0.071	50
Covertype	0.0844	50	0.0865	50	0.085	50	0.0869	50	0.0858	50
Higgs	0.0562	50	0.0656	50	0.0788	50	0.0821	50	0.0552	50
MNIST 2 vs 6	0.259	21.8	0.6308	50	0.6289	50	0.584	50	0.5317	50
MNIST	0.2781	37.6	0.5657	50	0.5152	50	0.4963	50	0.3422	50
FMNIST	0.2925	36	0.5847	50	0.5294	50	0.5035	50	0.3856	50
Average	0.1744	43.2	0.2944	50	0.2758	50	0.2658	50	0.2281	50

Generally this approach improves a lot upon simply providing the adversarial example itself. It is now beneficial to provide more adversarial examples to the fuzzer, contrary to previous results. Feeding this kind of adversarial example is now even beneficial for some of the non-image datasets such as IJCNN1 and Breast-cancer. We further experiment, also providing adversarial examples a_2 in between a_1 and v and a_3 in between a_1 and a in Table 5.4. For "10 AE ++", the input corpus now exists of $10(\text{ANN}) * 4 + 1$ (the victim) = 41 inputs. This further improves the performance. Double fuzzing with 10 adversarial examples and its artificial in-between points will be used in FATE. It seems to work very well for all datasets except Breast-cancer and IJCNN1, that perform best when always seeded with multiple adversarial examples.

Table 5.4: Adversarial example initialization with multiple artificial points

	Baseline		**1 AE ++**		**5 AE ++**		**10 AE ++**		**Double Fuzz 10 AE++**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2841	50	0.2795	50	0.2258	50	0.2303	50	0.2815	50
Diabetes	0.0697	50	0.069	50	0.0691	50	0.0694	50	0.069	50
IJCNN1	0.0713	50	0.0636	50	0.0635	50	0.0632	50	0.0711	50
Covertype	0.0844	50	0.0801	50	0.084	50	0.0844	50	0.085	50
Higgs	0.0562	50	0.0623	50	0.0662	50	0.0706	50	0.0561	50
MNIST 2 vs 6	0.259	21.8	0.5901	50	0.5462	50	0.5051	50	0.4735	50
MNIST	0.2781	37.6	0.4998	50	0.3928	50	0.3575	50	0.3189	50
FMNIST	0.2925	36	0.5103	50	0.4605	50	0.4179	50	0.3841	50
Average	0.1744	43.2	0.2693	50	0.2385	50	0.2248	50	0.2174	50

5.3. Model

The **thresholds** in a decision tree specify the boundary values of decision regions within decision trees. We consider all threshold values T_f for feature f. For Gradient boosting, we take into account all sub-trees that are produced in each iteration. After producing a mutated value m_f, it can be optimized by moving it towards the first threshold t between m_f and the original value for that feature o_f without changing the prediction. E.g. with e =small_perturbation_threshold:

- If $m_f < o_f$: mutate to $t - e$ with t = the smallest threshold $t \in T_f \mid m_f <= t < o_f$.
- If $m_f > o_f$: mutate to $t + e$ with t = the largest threshold $t \in T_f \mid o_f < t < m_f$.

When such a threshold t cannot be found, mutate to o_f. The addition or subtraction of e is performed to eliminate potential floating-point inconsistencies due to different precision between the thresholds in the model and the thresholds that are injected in the C++ target. Looking at Table 5.6, the produced adversarial distances are marginally better using the thresholds for mutation.

Looking at Table 5.5, there can be as many as 2198 thresholds per feature (for the Higgs dataset). This means mutated values will generally only be moved towards the nearest threshold very slightly, thus resulting in only a small difference in the created adversarial examples. We investigated if using these thresholds has a negative impact on the number of executions per second for the Covertype, Higgs and MNIST datasets. Only for MNIST there is a significant difference in the number of executions per second: 124.6 without thresholds and 96.8 with thresholds enabled, a 28.7% difference. To try to improve the performance we remove all thresholds which are within 0.0001 from each other (using a bigger interval would counter the effect of using the thresholds at all). This only marginally increases the number of executions per second for MNIST. For Covertype this removes almost half of the thresholds for some features, but again the number of executions per second is not impacted. Using thresholds generally means that the adversarial examples become slightly better, which can help in finding the optimal adversarial example for a victim. We thus enable the thresholds by default, keeping in mind that they may impact the number of executions for (F)MNIST.

Feature importances. Some features may be more important than others for classifying input. When looking at Figure 5.1 which visualises how often a feature is used for classification on the MNIST 2 vs 6 dataset, it can be clearly seen that there is a limited amount of features that is used extensively, a larger amount of features that is sometimes used, and a lot of features that are never used for classification at all. This visualisation was created with the number of occurrences of a feature in a splitting condition, but some training algorithms, such as the ones in scikit-learn, also record (impurity-based) feature importances. These are measures on how important a feature is for distinguishing inputs belonging to other classes. Using these feature importances i, we calculate the chance to mutate feature f as $\frac{i_f}{\sum_{j=0}^{n-1} i_j}$.

Table 5.5: Number of thresholds per feature

Dataset	Min	Max	Average	Median	Std
Breast-cancer	0	11	3.4	2	3.2
Diabetes	9	57	28.6	23	18.4
IJCNN1	1	1008	314.9	311	342.3
Covertype	0	1978	125.2	1	401.7
Higgs	3	2198	1275.1	1308	589.7
MNIST 2 vs 6	0	108	7.0	3	10.8
MNIST	0	492	198.8	197	166.0
FMNIST	0	450	275.5	309	110.4

Using the feature importance metric to bias which features should be mutated improves the performance on most datasets as can be seen in Table 5.6. Worth noting is that the Covertype dataset has a lot of features (38) with only one threshold. Using the feature importance metric, FATE can make a better prediction if mutating these features is worth it or not. Much less victims can be attacked for (F)MNIST. For MNIST 2 vs 6 using the feature importances results in slightly better adversarial distances, though not more victims could be attacked than with the baseline. Using feature importance is a better metric for FATE than using feature occurrence as the adversarial distances are generally lower with at least as many victims that can be attacked.

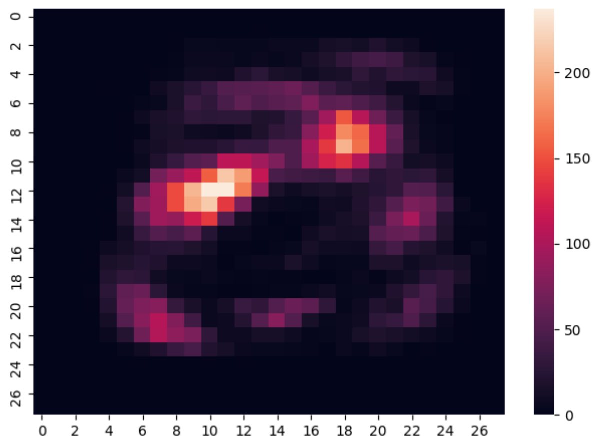


Figure 5.1: Feature occurrence in a splitting condition for MNIST 2 vs 6

5.4. Execution

Not all mutations will be equally interesting. It is important that as many mutations as possible examine new regions of the input space that are beneficial for finding good adversarial examples, and the mutations should thus be small enough not to change too much of the current seed/victim. We implement different mutation strategies that adjust the neighbourhood of a seed s if it already is an adversarial example, keeping in mind that we test with the L_∞ distance, which only depends on the feature with the largest difference d between the adversarial example and the victim. These strategies should help in exploring leaves closer to the victim and biasing the mutation of the most important features:

1. **Bias large differences**. As it is more likely that the features with large differences between the mutated and the victim point will have to be mutated to reach a lower adversarial distance (as L_∞ only depends on the feature with the largest perturbation), we adjust the chance of mutating a feature with difference δ_f by adding an extra factor $\frac{\delta_f}{\sum_j \delta_j}$ to the default mutation chance.

2. **Mutate all largest differences**. When an adversarial example is found, not mutating the feature which has the largest distance to the victim will never result in finding a better adversarial example, as the L_∞ norm distance is equal to the feature with the largest distance d. With probability 0.5, we mutate all

Table 5.6: Using model information

Dataset	Baseline $\bar{r}$	Baseline n	Thresholds $\bar{r}$	Thresholds n	Feature importance $\bar{r}$	Feature importance n	Feature occurrence $\bar{r}$	Feature occurrence n
Breast-cancer	0.2841	50	0.289	50	0.2692	50	0.2854	50
Diabetes	0.0697	50	0.0676	50	0.0703	50	0.0699	50
IJCNN1	0.0713	50	0.0698	50	0.0643	50	0.0675	50
Covertype	0.0844	50	0.0818	50	0.0747	50	0.0781	50
Higgs	0.0562	50	0.0554	50	0.0465	50	0.0503	50
MNIST 2 vs 6	0.259	21.8	0.2285	21.2	0.2378	21	0.2428	21
MNIST	0.2781	37.6	0.2717	36	0.2466	21.6	0.2439	19
FMNIST	0.2925	36	0.2922	35.2	0.2629	19.6	0.2516	17
Average	0.1744	43.2	0.1695	42.8	0.1590	39.0	0.1612	38.4

features that have distance $> d - 0.01$. The term 0.01 is necessary because when only the feature with the largest difference is mutated, it is likely that the objective function will not signal an improved adversarial example because other features close to the largest feature will then determine the adversarial distance, due to the nature of the L_∞ norm.

3. **Mutate less when closer**. For adversarial examples, the maximum mutation distance is scaled with the largest difference δ of any feature with the victim, allowing for smaller mutations when a good adversarial example was found already, thus decreasing the neighborhood of a victim and increasing precision.

4. **Only mutate towards victim**. For adversarial examples, all features are only mutated towards the victim features, allowing for mutations "past" the victim features but with smaller difference than the original difference of the feature. Mutual exclusive with the previous strategy.

Table 5.7 shows experiments with the mutation bias strategies mentioned above. Biasing mutating the features with the largest difference with the original features performs a bit better than the baseline, and will thus be used. Mutating less when closer to the victim does not seem to benefit the performance and is thus not used. Mutating all largest differences clearly benefits the performance for most datasets. The "Only mutate towards victim" strategy mainly improves the performance for the image datasets, but will be used. We also tested with all mutation bias strategies enabled, which has good performance but at the cost of some victims that cannot be attacked for the image datasets and will thus not be used.

Table 5.7: Mutation Bias

Dataset	Baseline $\bar{r}$	Baseline n	Bias largest diffs $\bar{r}$	Bias largest diffs n	Mutate less when closer $\bar{r}$	Mutate less when closer n	Mutate all largest $\bar{r}$	Mutate all largest n	Mutate towards victim $\bar{r}$	Mutate towards victim n	All $\bar{r}$	All n
Breast-cancer	0.2841	50	0.2872	50	0.2989	50	0.2857	50	0.298	50	0.286	50
Diabetes	0.0697	50	0.0679	50	0.0717	50	0.0639	50	0.0681	50	0.066	50
IJCNN1	0.0713	50	0.0671	50	0.0699	50	0.0585	50	0.0694	50	0.0575	50
Covertype	0.0844	50	0.0819	50	0.0857	50	0.0687	50	0.0827	50	0.0691	50
Higgs	0.0562	50	0.0519	50	0.0521	50	0.0485	50	0.0536	50	0.0458	50
MNIST 2 vs 6	0.259	21.8	0.2491	21.4	0.2452	21.2	0.215	21.4	0.2225	21.4	0.2105	21
MNIST	0.2781	37.6	0.2701	34.6	0.2744	33.6	0.2636	37.2	0.2558	33.8	0.2598	32.4
FMNIST	0.2925	36	0.2872	33.8	0.2909	35	0.2843	35.4	0.2814	34	0.2833	29.8
Average	0.1744	43.2	0.1703	42.5	0.1736	42.5	0.1610	43	0.1664	42.4	0.1598	41.7

Only using the biased mutation strategies when the current input is an adversarial example requires a model query for each mutation that is performed, which can be up to mutation_depth times per run (5). This can negatively impact the executions per second, mainly for the big models such as (F)MNIST. We implemented different adversarial example recognition systems in Table 5.8. Model Query performs a model query each time the mutator is called for an input. Save last prediction saves the last prediction such that an input is identified as adversarial example if the last input that was fed to the objective function was an adversarial example. This may produce a wrong result if it is not the previous input that is being mutated but a different seed from the corpus. In "AE chance 0.5" we model that an input resembles an adversarial example with probability 0.5. In "Always AE" we treat every input as an adversarial example and in "Never AE" we treat every input as not being an adversarial example, thus not performing the biased mutation techniques. The most precise approach of performing a model query for every input performs the best. The performance penalty in executions per second is not significant for the Breast-cancer dataset while MNIST performs 17% more executions (116 vs 99) per second when using "Save last prediction". The performance on MNIST is however better when performing a model query each mutation, so we keep using that exact approach.

Table 5.8: Adversarial example identification techniques in the mutator

	Model query		**Save last prediction**		**AE chance 0.5**		**Always AE**		**Never AE**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2123	50	0.2119	50	0.2119	50	0.2114	50	0.2141	50
Diabetes	0.06	50	0.0604	50	0.0602	50	0.0606	50	0.0643	50
IJCNN1	0.0503	50	0.0504	50	0.0502	50	0.0477	50	0.0629	50
Covertype	0.0567	50	0.0583	50	0.0604	50	0.0585	50	0.0748	50
Higgs	0.0314	50	0.0338	50	0.033	50	0.0605	50	0.0441	50
MNIST 2 vs 6	0.3618	49.8	0.3874	48.6	0.4174	48.6	0.4517	50	0.4576	49
MNIST	0.2468	50	0.2597	50	0.2611	50	0.3593	49.4	0.2876	50
FMNIST	0.2695	48.8	0.2856	49.4	0.2818	48.8	0.3955	48.8	0.2988	49.6
Average	0.1611	49.8	0.1684	49.8	0.1720	49.7	0.2057	49.8	0.1880	49.8

5.5. Conclusion

We identified three sources of information: the training set, the model and information directly from the execution of the fuzzer. We showed that with each of these types of information FATE can be improved. Through making use of samples in the training set the fuzzer can be initialized with both adversarial examples and forged input. This guarantees that every victim can be attacked through using double fuzzing if the first run did not produce an adversarial example. The model allows us to use both feature importance and threshold information, which both improve the performance of FATE. Through using information from current executions, we are able to improve the mutations that are made in FATE thus improving the created adversarial examples. When an adversarial example is found we bias mutating features with bigger distances to the original victim feature, we mutate only towards the victim and with probability 0.5 we mutate all features that are limiting the current seed to its distance branch in the objective function.

We test the influence of coverage guidance with the updated setup. The results are shown in Table 5.9. Using coverage guidance only in the objective function is clearly the best option: it attacks the most victims with the smallest adversarial distances for all datasets except the two easiest datasets. In chapter 7 we further investigate the influence of coverage guidance.

Comparing the performance of FATE before (old baseline) and after leveraging more information (FATE smarter) in Table 5.10, leveraging more information is clearly beneficial for FATE. For all datasets except MNIST 2 vs 6 and Breast-cancer the average adversarial distance is better while attacking all victims. For breast-cancer and MNIST 2 vs 6 the average adversarial distance has increased because the last victims that can now be attacked have a high adversarial distance. For easy comparison we also show FATE final (the final default setup for FATE) in this table, where coverage guidance is enabled only in the objective function. The performance is a little worse for the easy Breast-cancer and Diabetes datasets compared to FATE smarter, but the performance on the other datasets has clearly improved.

Table 5.9: Influence of coverage guidance for FATE when using the final custom mutator

	Full coverage		**Coverage in The Forest**		**Coverage in the objective function**		**No coverage Instrumentation**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2123	50	0.2114	50	0.2194	50	0.2181	50
Diabetes	0.06	50	0.0637	50	0.0619	50	0.0672	50
IJCNN1	0.0503	50	0.0539	50	0.0442	50	0.0496	50
Covertype	0.0567	50	0.064	50	0.0515	50	0.0584	50
Higgs	0.0314	50	0.0365	50	0.0132	50	0.0171	50
MNIST 2 vs 6	0.3618	49.8	0.4071	49.2	0.2233	49.8	0.4301	48.2
MNIST	0.2468	50	0.2492	50	0.1234	50	0.2386	50
FMNIST	0.2695	48.8	0.2669	49.4	0.1284	49.8	0.2701	49.8
Average	0.1611	49.8	0.1691	49.8	0.1082	50.0	0.1687	49.8

Table 5.10: Performance improvement of FATE when fuzzing smarter

	Old Baseline		**FATE smarter**		**FATE final**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.1511	46	0.2123	50	0.2194	50
Diabetes	0.0652	50	0.06	50	0.0619	50
IJCNN1	0.0703	50	0.0503	50	0.0442	50
Covertype	0.085	50	0.0567	50	0.0515	50
Higgs	0.0482	50	0.0314	50	0.0132	50
MNIST 2 vs 6	0.2238	21.4	0.3618	49.8	0.2233	49.8
MNIST	0.2854	46.4	0.2468	50	0.1234	50
FMNIST	0.2917	45.2	0.2695	48.8	0.1284	49.8
Average	0.1526	44.9	0.1611	49.8	0.1082	50

6
FATE with different fuzzing engines

In this section we compare the performance of FATE when fuzzing with AFL++, honggfuzz and libFuzzer. These are all grey-box coverage guided fuzzers, however they differ slightly in their approaches of executing the target, the way they mutate input in their mutators, and in the way coverage is recorded. Also, they perform differently when searching for bugs in open-source libraries as shown by fuzzbench [1]. In the previous chapters we concluded that the performance of FATE heavily depends on its custom mutator. We explore if finding adversarial examples is also possible with the built-in mutators of fuzzers. When fuzzing with the built-in mutators, "Double fuzz" is turned off: It may well be the case that the adversarial examples with which the fuzzer is initialized will be the best adversarial examples the fuzzer finds, such that the performance of the mutator will depend on the initial adversarial examples instead of the performance of the mutator. To be able to better compare the built-in mutators of the fuzzers with each other, the fuzzers are executed without any initial adversarial examples or "double fuzzing".

6.1. AFL++

AFL++ ([17]) is a fork of the popular AFL ([46]) fuzzer which incorporates recent advances in fuzzing research such as the ability to use custom mutators and different seed selection schedules. By default AFL++ feeds the fuzzed inputs via stdin or files that can be parsed by the target. For extra speedup, we implement their persistent mode, which skips many calls to the forkserver and passes mutated inputs via an in-memory buffer. The custom mutator is provided through their custom mutator API, which does not support crossover. Another difference with other fuzzers is that the mutator should be compiled and supplied as a separate library.

AFL divides mutations in two categories: "deterministic" and "havoc". The deterministic stage performs single deterministic mutations on the input seed. In the havoc stage, mutations are randomly stacked and also change the size of the testcase. AFL++ applies bit/byte flips, insertion of interesting words/bytes (through a dictionary), arithmetic operations such as addition or subtraction on integers, randomly adjusting bytes, block deletion and duplication and crossover [13]. Edge coverage is measured directly and stored in an approximate hash table along with coarse branch-taken hit counts [31].

AFL++ recommends making use of afl-clang-lto with CMPLOG functionality. This logs the operands of the last 256 executions for each comparison in a shared table between the target and the fuzzer. The CMPLOG functionality however introduces a serious memory leak when starting and stopping many fuzzing instances as we need to do with FATE, and can thus not be used. Instead, we use the Dict2File feature (`AFL_LLVM_DICT2FILE=dict_path`) that records interesting words during compilation and saves them to a file that can be used to guide mutation. We furthermore enable the splitting of comparison operations (`AFL_LLVM_LAF_ALL=1`) for floats, integers and strings as well as the splitting of switch statement in smaller sub-steps such that coverage guidance can be used to tackle these statements step by step. Every fuzzing instance is started as a Master without any Slave processes. When we fuzz with timeout the environment variable `AFL_FAST_CAL=1` is used to speedup the calibration stage of AFL++. When fuzzing with the custom mutator trimming is disabled (`AFL_DISABLE_TRIM=1`) because the testcases should maintain their length.

The performance of AFL++ is shown in Table 6.1. The performance of the baseline setup using AFL++ is generally worse than the performance of the baseline using libFuzzer. It is likely that a lower execution speed

[1] https://www.fuzzbench.com/reports/2021-04-11/index.html

is the cause of the worse performance: for the Breast-cancer dataset the execution speed is 12800 compared to 127000 for libFuzzer. Also for MNIST AFL++ is slower: 67 compared to 85 for libFuzzer. Extra tests with different power schedules (we use the default 'fast') did not increase the number of executions per second. AFL++ does perform better than libFuzzer on the Higgs dataset though, and it has comparable performance on the IJCNN1 dataset. The running times t are higher due to more victims being "double fuzzed".

For the no timeout setting, AFL++ was instructed to stop if no new paths were found for 6 seconds (60 seconds for the image datasets) with a maximum of 200 seconds per victim. The fuzzer keeps on finding new paths for a long time: for many datasets the timeout of 200 seconds is reached. Except for MNIST 2 vs 6, the performance is better than when fuzzing shorter for all datasets as expected. The performance without timeout is worse for MNIST 2 vs 6 because the fuzzer is now able to attack many victims on its first run with high adversarial distances. Not attacking some victims and performing "double fuzz" while initializing the fuzzer with an adversarial example performs better for these victims.

We also test AFL++ with only out-of-the-box features and mutator, so we disable the custom mutator of FATE and "double fuzz". It stands out that this improves the average adversarial distance on the Higgs dataset compared to the baseline, however when allowing the baseline to also run without timeout the baseline performs better. Nevertheless, it is impressive that the built-in mutator can find adversarial examples within 3 percent-point difference from the optimal adversarial examples (0.0022), as it does not have any domain knowledge. For the other datasets, the performance is clearly worse when using the built-in mutator of AFL++. AFL++ has problems in finding additional coverage for the Covertype dataset as the runtime for this dataset is quite low compared to the other datasets, which means that generally early on no new paths could be explored for 6 seconds and the fuzzer quits.

Table 6.1: AFL++ performance

	Baseline (libFuzzer)			**AFL++**			**AFL++ no timeout**			**AFL++ no custom mutator no timeout**		
Dataset	$\bar{r}$	n	t	$\bar{r}$	n	t	$\bar{r}$	n	t	$\bar{r}$	n	t
Breast-cancer	0.275	50	0.21	0.3575	50	0.31	0.228	50	6.60	0.3072	43.6	7.42
Diabetes	0.0645	50	0.21	0.0942	50	0.36	0.0671	50	8.91	0.101	50	8.72
IJCNN1	0.054	50	0.21	0.0572	50	0.33	0.0519	50	18.58	0.0753	50	14.18
Covertype	0.0636	50	0.22	0.0848	50	0.37	0.0629	50	20.25	0.1074	50	3.52
Higgs	0.0382	50	0.21	0.0289	50	0.31	0.019	50	20.26	0.0295	49.8	20.24
MNIST 2 vs 6	0.4284	50	2.00	0.5659	50	2.71	0.8287	50	17.17	-	0	20.12
MNIST	0.3408	50	2.09	0.3607	50	2.75	0.3322	50	43.37	-	0	20.17
FMNIST	0.3877	50	2.23	0.4356	50	2.70	0.3652	50	45.32	-	0	20.17
Average	0.2065	50.0	0.92	0.2481	50.0	1.23	0.2444	50.0	22.56	0.1241	30.4	10.85

6.2. libFuzzer

LibFuzzer is an in-process (it operates entirely in memory), coverage-guided evolutionary fuzzing engine ([29]). It is part of the LLVM project and ships directly with recent versions of Clang. LibFuzzer requires a bit of setup: at least the `LLVMFuzzerTestOneInput` function should be implemented that feeds input generated by libFuzzer to the program under test. This input is then executed and coverage information is provided through SanitizerCoverage ([39]) instrumentation. Coverage-increasing seeds are saved to the corpus with the goal to penetrate deeper into the source code.

The built-in mutator of libFuzzer performs several mutations:

- EraseBytes removes bytes from the input thus decreasing the size of the seed
- InsertByte inserts a byte and thus increases the size of the input
- InsertRepeatedBytes inserts multiple bytes to the input
- ChangeBit flips one of the bits in the input
- ChangeByte replaces one of the bytes with a random byte
- ShuffleBytes is a crossover function that randomly re-arranges the bytes of the current input
- ChangeASCIIInteger randomly adjust an ASCII integer in the input
- ChangeBinaryInteger randomly adjust a binary integer in the input
- CopyPart returns only a part of the current input

- Crossover combines the current input with a random input from the corpus or from itself
- CMP adds values of a CMP instruction to the dictionary
- AddWordPersistAutoDict replaces a part of the input with that of a seed that previously increased code coverage.
- AddWordTempAutoDict replaces a part of the input with that of a seed that recently increased code coverage.
- AddWordFromTORC replaces a part of the input with the contents of a recently executed compare instruction

The target is compiled using clang++-11 with the `-fsanitize=fuzzer -fbracket-depth=1100 -fsanitize-coverage=bb,trace-cmp` command-line arguments. The increased bracket depth is necessary for some layouts of the objective function where the bracket depth can become more than 1000. The 'bb' in the sanitize-coverage argument stands for instrumentation on the basic-block level (instead of function/edge) and the trace-cmp flag adds instrumentation around comparison instructions and switch statements. The trace-cmp flag is a recommended option by the creators of libFuzzer [29]. The command-line arguments supplied to the binary compiled as fuzz target all have to do with the input and its length: `-reduce_inputs=0` such that the fuzzer will not try to minimize testcases, `-reload=0` such that the fuzzer will not periodically reload the corpus as this is unnecessary as we only fuzz the same corpus with one instance, `-max_len=(num_features * BYTES_PER_FEATURE)` - the expected number of input bytes, `-prefer_small=0` as we always look for inputs with max_len bytes and `-len_control=0` such that inputs up to max_len are tried right away.

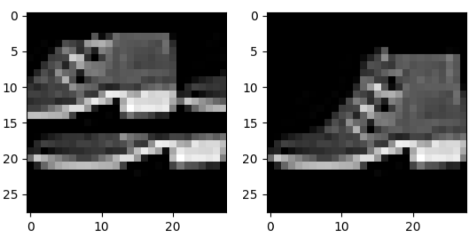


Figure 6.1: Best adversarial example produced by libFuzzer for FMNIST. Original (right): Ankle boot. Adversarial example (left): Bag

We test the performance of libFuzzer with their out-of-the-box mutator and 'without a timeout'. Because libFuzzer has no support to quit the fuzzer when for a certain amount of time no new paths are found, like AFL++ does, we set the timeout to 200 seconds per victim, which is 100x as long as the default fuzzing time of 2 seconds. We also set the `use_value_profile=1` flag, which collects values profiles for the parameters of compare instructions and treats some new values as new coverage. We also set the `-use_cmp=1` flag, which is similar to the LAF feature of AFL++. Lastly, we also enable the entropic power schedule such that the setup is mostly similar to AFL++, only an alternative for their autodict feature is missing from the libFuzzer setup.

Experimental results are shown in Table 6.2. Using "no timeout" improves the performance of FATE as expected. For each dataset the average distance if smaller while attacking all victims. The out-of-the-box mutator is able to find adversarial examples for most of the victims. The average distances are much worse though than with the custom mutator and not all victims can be attacked. For the image datasets only adversarial examples with very high distance can be found. We show one of the created adversarial examples in Figure 6.1. It looks like the example is generated by performing crossover, possibly with itself. The generated example is clearly different from the original and may not be recognised as being an ankle boot by humans either.

6.3. Honggfuzz

Honggfuzz is a security oriented, feedback-driven and evolutionary fuzzer [21]. Like libFuzzer, it uses the `LLVMFuzzerTestOneInput` and `LLVMFuzzerInitialize` APIs for testing a target. Coverage feedback is received through SanitizerCoverage's compile-time instrumentation. Honggfuzz adds a layer over Clang and automatically adds the `-fsanitize-coverage=trace-pc-guard,indirect-calls,trace-cmp` arguments to the fuzzer that instrument comparison instructions, indirect calls, and add a guard variable to each edge. Next to inputs that increase code coverage, honggfuzz also identifies inputs that increase instruction/branch counters and edge coverage is measured indirectly using basic block coverage. In contrast to AFL++ (which implements multiple types of schedules) and libFuzzer (with an entropic seed schedule), seeds are selected randomly from the in-memory corpus. Honggfuzz does not support the custom mutator API like libFuzzer, however it does support custom mutation through calling a script that mutates input files. The current version (2.4) has a bug for this feature though where the file provided to the mutator is non-existent, so we do not test with the custom mutator. We set `--mutations_per_run 5` (the mutation depth) to the same

Table 6.2: libFuzzer performance

	Baseline (Custom Mutator)			Custom mutator No timeout			No custom mutator No timeout		
Dataset	$\bar{r}$	n	t	$\bar{r}$	n	t	$\bar{r}$	n	t
Breast-cancer	0.2750	50	0.21	0.2157	50	20.11	0.4764	37.2	20.11
Diabetes	0.0645	50	0.21	0.0596	50	20.11	0.2677	49.2	20.11
IJCNN1	0.0540	50	0.21	0.0463	50	20.13	0.2119	50.0	20.12
Covertype	0.0636	50	0.22	0.0548	50	20.15	0.4000	43.6	20.13
Higgs	0.0382	50	0.21	0.0235	50	20.17	0.1368	50.0	20.14
MNIST 2 vs 6	0.4284	50	2.00	0.2672	50	34.61	0.9973	41.8	20.12
MNIST	0.3408	50	2.09	0.2227	50	24.33	0.9981	30.2	20.28
FMNIST	0.3877	50	2.23	0.2421	50	24.34	0.9707	25.8	20.29
Average	0.2065	50	0.92	0.1415	50	22.99	0.5574	41.0	20.16

value of libFuzzer and `-F (num_features * BYTES_PER_FEATURE)` to specify the maximal (and expected) input size. Honggfuzz does not support exiting when no paths are found for n seconds like AFL++, so similar to libFuzzer we set the timeout to 200 seconds.

Honggfuzz produces a lot of adversarial examples with non-finite input, which means the produced features are NaN (Not a Number), ∞ or $-\infty$. This input is discarded by FATE, so many inputs that reached new coverage in the target are inputs that are actually invalid. This decreases the performance of honggfuzz. Results are shown in Table 6.3. Honggfuzz is able to attack most victims, and the adversarial distances are similar to libFuzzer and worse than AFL++. This is true especially for the image datasets, where the distances are close to 1 which means that every feature can be adjusted over almost the whole feature range. The best adversarial example found by honggfuzz for FMNIST is shown in Figure 6.2. One can argue that this could still be a usable adversarial example in the real world: for example putting some tape over a shirt at the right location might trick the model into classifying wrong. This adversarial example shows that using the builtin mutators, it might be more feasible to attack models on L_0 distance. Many of their operators consist of removing or changing bytes which result in big changes in single features and thus large L_∞ distances, but not necessarily many features that are changed at one time which is important for the L_0 norm.

Table 6.3: Honggfuzz performance

	No custom mutator No timeout		
Dataset	$\bar{r}$	n	t
Breast-cancer	0.4627	36.4	20.1032
Diabetes	0.2029	49.4	20.1021
IJCNN1	0.2745	50	20.1356
Covertype	0.3039	49.2	20.1751
Higgs	0.1078	50	20.2654
MNIST 2 vs 6	0.9961	48.2	20.139
MNIST	0.9961	49.4	20.2309
FMNIST	0.909	47.4	20.2954
Average	0.5316	47.5	20.18

6.4. Conclusion

In this chapter we showed that FATE is not limited to using libFuzzer as a fuzzing engine. FATE does however perform best when using libFuzzer, both when fuzzing with or without timeout. The reason for this is the much lower number of executions per seconds that is reached when fuzzing with AFL++ (while implementing all performance boosting options that they offer through their "Persistent mode" fuzzing). LibFuzzer, AFL++ and honggfuzz are all able to find adversarial examples using their built-in mutators, however the adversarial examples that are found are generally much worse than when using the custom mutator. This is no surprise:

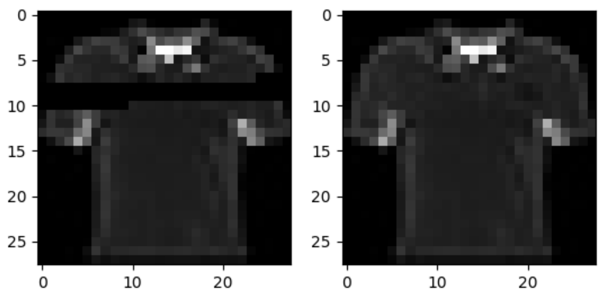


Figure 6.2: Best adversarial example produced by honggfuzz for FMNIST. Original (right): t-shirt. Adversarial example (left): shirt/top

the custom mutator incorporates domain knowledge that the default mutators do not have. Considering the default mutators, AFL++ finds the best adversarial examples on the datasets we are testing with, however it is unable to find adversarial examples for the image datasets where libFuzzer and honggfuzz are able to find adversarial examples for these datasets. The clearly worse performance with the built-in mutators for all these fuzzers shows the importance of the custom mutator of FATE.

7
Fuzzing Efficiency

In previous chapters we showed how fuzzers can be used to find adversarial examples in tree ensembles using FATE and how the performance of FATE can be improved by leveraging information from various sources. In this chapter we investigate how efficient it is to fuzz for adversarial examples: is the objective grey-box fuzzers try to optimize, code coverage, equal to finding more and better adversarial examples? Also we investigate the scalability of FATE for the various model sizes in our datasets and show our main results through comparing the performance of FATE with the state-of-the-art in section 7.4. We verify the performance of FATE on two previously unseen datasets. Finally, we execute FATE as a standalone Evolutionary Algorithm to investigate if FATE will perform better without fuzzing overhead and coverage-guidance distractions.

7.1. Coverage guidance

Fuzzers aim to increase code coverage. As mentioned in chapter 3, for FATE this should be strongly related to finding better adversarial examples. For all but the smallest models, the most basic blocks in the produced C++ code are part of the individual decision trees. Increasing code coverage thus mainly means that more leaves of the tree ensemble are explored. Recalling from chapter 2, coverage in the tree ensemble is necessary for finding adversarial examples, but also for improving them. We investigate the relation of code coverage and adversarial example quality & quantity using the Breast-cancer dataset. The Breast-cancer dataset is a fairly easy dataset to attack so for readability of the plot the ϵ value was set to 0.1 which makes it more difficult for the fuzzer to find new leaves. The fuzzing target was executed with increasing amounts of runs in the range [1.000, 800.000]. After each number of runs, the adversarial examples are removed and the fuzzer is started with a clean sheet. At each run one mutated input is generated (through up to 5 consecutive calls to the mutator). 1.000 runs thus equal 1.000 mutated inputs which are tested against the tree ensemble. Figure 7.1 shows the development of the quality and quantity of adversarial examples over the amount of fuzzing runs. It also shows the Approximate Nearest Neighbour distance to samples from the training set.

The curves showing the approximate closest distances to other instances show that the examples the fuzzer finds are quite different from samples in the training set. This is expected as the fuzzers needs to evade the decision regions around points in the training set where the model is quite certain what the prediction should be. In the plot we see that forged adversarial examples generally lie closer to points from the original class of the victim (distance to own class) than to points of other classes (distance to other classes). Intuitively, good adversarial examples look as much like the victim (and other points of the victim class) as possible as that generally means the distance to the victim is as small as possible. For optimal adversarial examples for the Breast-cancer dataset, the average distance to the nearest sample of the class of the victim is 0.287 and the average distance to samples from the other class is 0.415, so also the optimal adversarial examples are quite different from instances from the training set.

It seems that more coverage means an higher number of successfully attacked victims and a lower average L_∞ distance. We further explore these relations. The relation between L_∞ distance and coverage is shown in Figure 7.2a. The figure shows the average coverage and L_∞ distance over the breast-cancer test-set. We show different coverage levels through increasing the number of runs (which on average corresponds to increasing the coverage). Lower coverage generally means a higher L_∞ distance. This will never be a perfect correlation though: because the fuzzer makes random mutations, it is possible that the fuzzer discovers a good/optimal

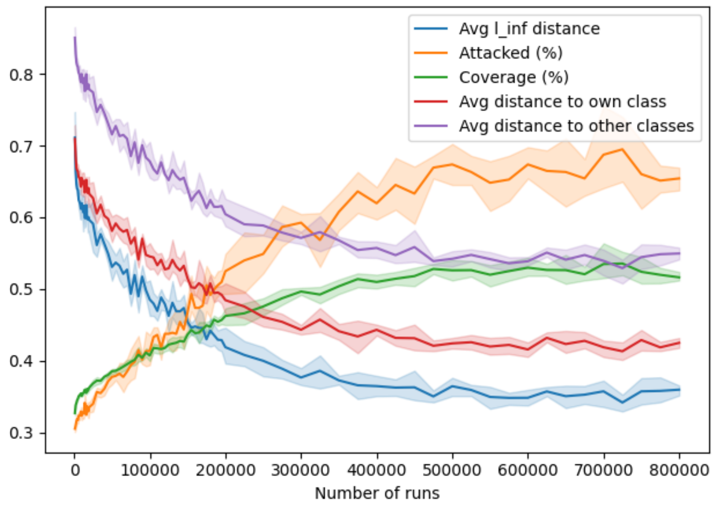
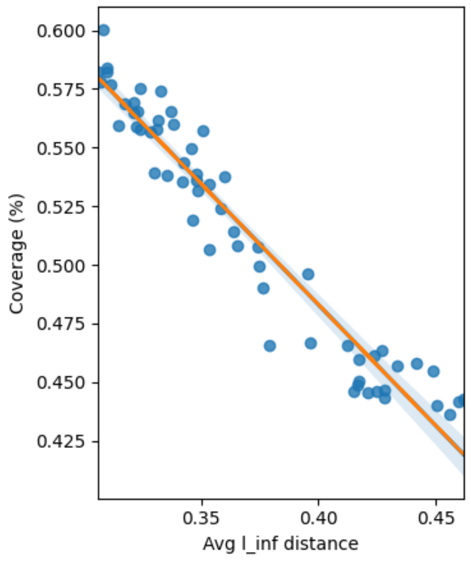
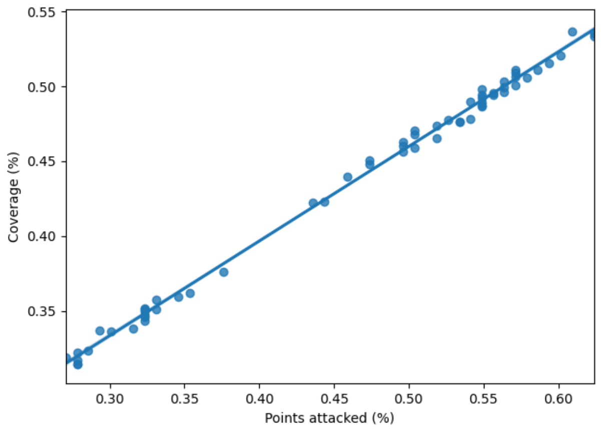


Figure 7.1: Fuzzing progress over an increasing amount of runs for the Breast-cancer dataset ($\epsilon = 0.1$)



(a) Relation between L_∞ distance and coverage



(b) Relation between points attacked and coverage

Figure 7.2: Coverage guidance for the Breast-cancer dataset ($\epsilon = 0.1$)

adversarial example right at one of the first mutations. The coverage will then still be low, with a relatively good L_∞ distance. This variance is limited through taking the average of the whole test-set. This makes the relation for the breast-cancer dataset quite clear: higher coverage generally means a lower adversarial distance.

The average number of points attacked and average coverage show an even clearer relation (Figure 7.2b). Again, coverage was increased by increasing the amount of runs. This relation is expected as the more (combinations of) leaves of the tree ensemble are explored, the more victims can be attacked.

We show the same plot as Figure 7.1 but with settings that easily generate optimal adversarial examples for the breast-cancer dataset in Figure 7.3. The optimal approach produces not only adversarial examples with much lower average distance, it is also a lot more consistent (this plot shows 5 separate executions as well). The fuzzer is now able to attack each victim much easier, thus relying less on the randomness of the fuzzer. The optimal setup reaches high coverage much quicker: already after for about 50.000 runs optimal adversarial examples are found where the previous setup stopped improving after about 600.000 runs while not producing optimal adversarial examples. The average percentage of coverage reached is also much higher: around 85% versus a little over 50%. The average L_∞ distance that is decreasing at the beginning of the plot and then increasing again can be explained by the lower amount of victims that can be attacked for the lower

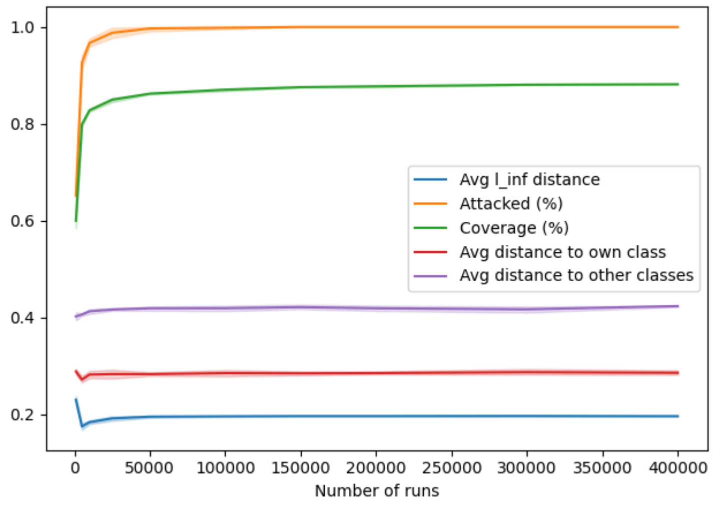


Figure 7.3: Fuzzing progress over an increasing amount of runs for the optimal Breast-cancer setup ($\epsilon = 0.4$)

number of runs: some victims with high optimal adversarial distance (up to 0.61) could not yet be attacked after that limited amount of runs. The lines describing the average distances to other instances deviate a small bit (especially the distance to other classes line at 400.000 runs). This can be explained by the fact that two adversarial examples with equal adversarial distance can theoretically be located at completely different locations in the input space, thus having different distances to instances of the training set. These lines can thus deviate without the optimal adversarial example distance changing significantly.

We investigate the added value of coverage guidance further by instructing the compiler to either compile coverage guidance for the full target, only for the forest, only for the objective function or not at all. The results are shown in Table 7.1. On the breast-cancer dataset the performance of FATE is best when using coverage guidance as expected with the previous findings. It is however remarkable that using no coverage guidance at all drastically improves the performance for the IJCNN1, Covertype and Higgs datasets with the current setup. When no coverage guidance is used, the fuzzer spends all of its time fuzzing the initial seed, which is the victim point. When the fuzzer is able to find an adversarial example this way, the distance of this adversarial example to the victim will generally be relatively small because the adversarial example will directly originate from the victim features itself. This also explains why less victims could be attacked for the more difficult image recognition datasets when no coverage guidance is used: it is difficult to mutate exactly the right combination of features with the right perturbation starting from the victim features itself.

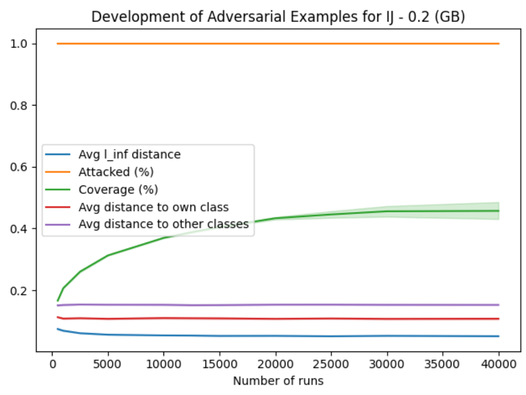
Because seeds that trigger new branches in the forest are also saved to the corpus for further mutation when using coverage guidance in the forest, the fuzzer is able to explore more combinations of leaves in the forest and is thus able to attack more victims. This is however at the cost of the average adversarial example distance for the other datasets, which is generally clearly worse compared to using no coverage guidance: the fuzzer is distracted by the seeds that trigger additional coverage in the forest (which may be far away from the victim) and thus spends less time fuzzing seeds that actually resemble good adversarial examples. Coverage guidance in the objective function only also improves the average adversarial distance compared to full coverage guidance, although the improvement is smaller than when using no guidance at all. Coverage guidance only for the objective function still has the problem of not being able to successfully attack most victims for the image datasets, because it can only explore a limited amount of combinations of leaves in the forest.

We further investigate the influence of coverage guidance on the IJCNN1 dataset, which performed clearly better without coverage guidance for the baseline setup as shown in Table 7.1. Maybe the custom mutator and crossover functions do not produce the right mutations with the baseline setup, which makes saving seeds that trigger new coverage less powerful as those seeds might be worse starting points than the victim itself. In Figure 7.4 we show results for the IJCNN1 dataset with a much better setup for the mutator (as identified in chapter 5). The differences in produced average adversarial examples are now much smaller: 0.0517 (full coverage guidance), 0.0554 (coverage guidance in the forest only), 0.0470 (coverage guidance for the objective function only) and 0.0535 (without coverage guidance). Coverage guidance in the objective function

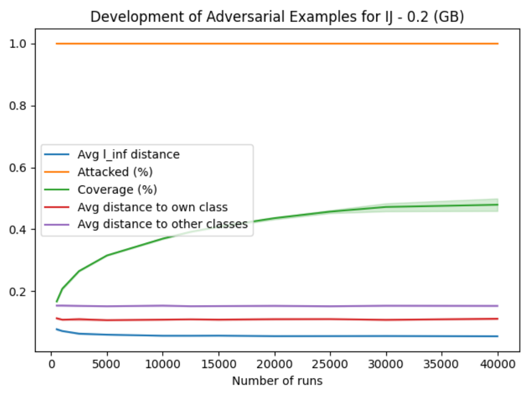
Table 7.1: Coverage Guidance

	Full guidance		**Guidance in Forest**		**Guidance in objective function**		**No guidance**	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.2189	50	0.2325	50	0.2174	50	0.2329	50
Diabetes	0.0841	50	0.1058	50	0.0796	50	0.0811	50
IJCNN1	0.194	50	0.2036	50	0.115	50	0.0591	50
Covertype	0.2565	50	0.2755	50	0.1691	50	0.0723	50
Higgs	0.1603	50	0.1592	50	0.1361	50	0.0306	50
MNIST 2 vs 6	0.4988	23.4	0.5247	24.2	0.4709	20	0.4037	19.2
MNIST	0.5251	44.6	0.5214	44.8	0.5225	35.2	0.4433	34.6
FMNIST	0.5329	45.2	0.5258	44.4	0.5202	32	0.4436	31.8
Average	0.3088	45.4	0.3186	45.4	0.2789	42.2	0.2208	42.0

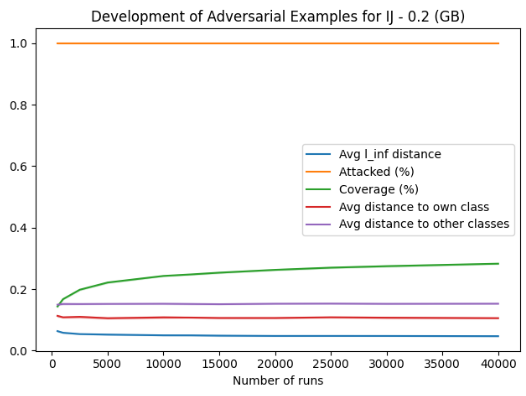
thus helps for finding better adversarial examples when the fuzzer makes the right mutations. There are big differences in the coverage that the fuzzer reaches with the different coverage guidance setups: enabling coverage-guidance in the forest results in a higher percentage of basic blocks covered (around 50%) versus 30% when fuzzing without coverage guidance. As coverage guidance in the forest does not assist the fuzzer with finding better adversarial examples, we can conclude that the fuzzer in that case spends its execution time on exploring irrelevant cases. In the plots in Figure 7.4 the fuzzer in general spends much of its effort in trying to trigger branches that do not aid in finding better adversarial examples as coverage increases much more than the quality of the created adversarial examples.



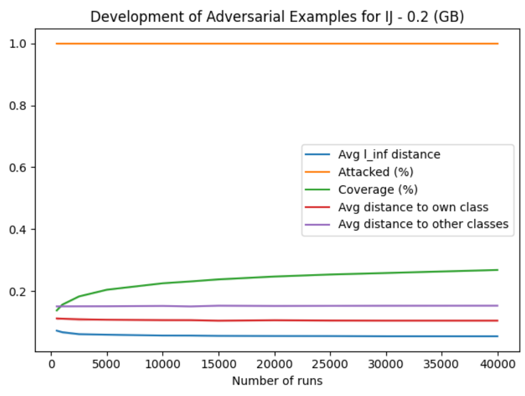
(a) IJCNN1 with coverage guidance over the full target



(b) IJCNN1 with coverage guidance in the Forest



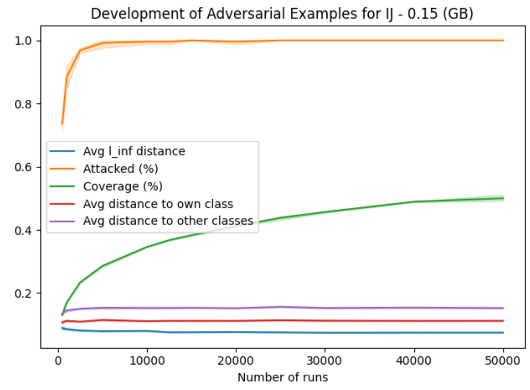
(c) IJCNN1 with coverage guidance in the objective function



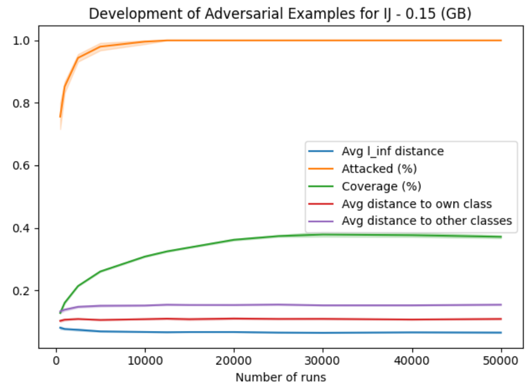
(d) IJCNN1 without coverage guidance

Figure 7.4: Coverage guidance influence on IJCNN1 ($\epsilon = 0.2$)

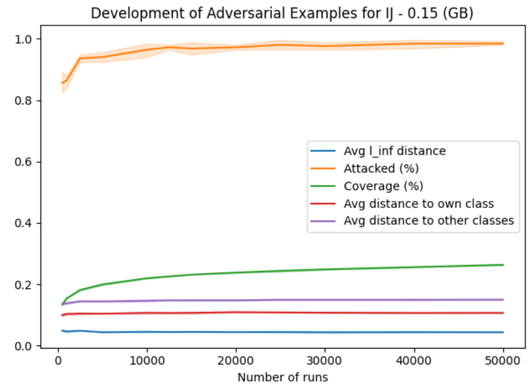
We investigate the development of the coverage curve (green line) when the quality of the created adversarial examples increases in Figure 7.5. The upper two figures (medium setup) produce examples with adversarial distances that are worse (0.06) than the distances in the lower two figures (0.045). The entropic power schedule in the right two figures can be neglected for now, it is introduced in the next section. In the figure we can clearly see that the relation between adversarial example quality and coverage does not hold: the upper figures reach more coverage than the lower figures. More coverage can thus actually mean that the best adversarial examples are worse. This means that the branches the fuzzer is investigating are not working towards the goal of finding better adversarial examples and can actually distract the fuzzer from identifying the most important combinations of branches. Similar results can be seen for the Covertype dataset in Figure B.2 (Appendix). The influence of the coverage that is reached depends per dataset: for the breast-cancer dataset the average coverage is quite high for the optimal adversarial examples and quite low for setups that produce worse examples (Figure B.1, Appendix). The trick for performance seems to be to generate the least amount of coverage for which all victims can still be attacked.



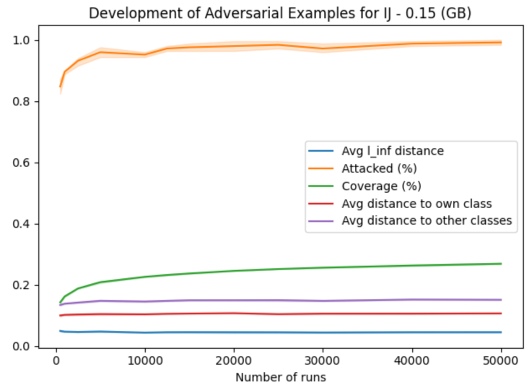
(a) IJCNN1 medium setup no entropic power schedule



(b) IJCNN1 medium setup with entropic power schedule



(c) IJCNN1 good setup no entropic power schedule



(d) IJCNN1 good setup with entropic power schedule

Figure 7.5: Performance of FATE over increasing runs for different setups on the IJCNN1 dataset

7.2. Seed selection

When investigating coverage we have to keep in mind that between different runs, 30% coverage does not necessarily mean that the same 30% of the leaves is covered. Because the fuzzer makes random mutations, different inputs will be found for each run, resulting in different leaves being triggered. Also, for some victims much more coverage may be achieved than for other victims. One reason that the relation between coverage and the number of victims attacked is so clear may be that taking the average over attacking multiple victims counters the variance in coverage between individual runs. Another explanation that this relation is so clear could be that certain leaves are more difficult to reach in the tree ensemble than others. Easy-to-reach leaves (given certain areas of seed input) will be quickly discovered by every fuzzing run, whereas difficult-to-reach leaves will generally take more time. If these leaves are necessary to attack a certain victim, the increased coverage, increased by more runs, also increases the number of victims that can be attacked. If this is true, there will be certain paths through the tree ensemble that are occurring often, whereas other paths will be much rarer. This phenomenon is known to occur when fuzzing normal code as well [7, 33].

LibFuzzer has an option to enable an entropic power schedule as described in [5], which assigns more energy to seeds that maximize information. That is, the entropy (information gain) is small when the fuzzer exercises mostly the same few program transitions, whereas if the input triggered previously unseen program behaviors (branches) the entropy is high and these inputs will generally be better at discovering new behaviors (branches). Using an entropic power schedule, priority will thus be given to seeds that are more likely to discover new program paths and less time will be spent on the "uninteresting" members of the population that generally follow the same paths. Table 7.2 shows the results of using an entropic power schedule. Using such a schedule clearly improves fuzzing compared with the baseline for non-image datasets. For image datasets less victims can be attacked.

Table 7.2: Entropic power schedule

Dataset	Baseline $\bar{r}$	Baseline n	Entropic power Schedule $\bar{r}$	Entropic power Schedule n
Breast-cancer	0.2189	50	0.2186	50
Diabetes	0.0841	50	0.0771	50
IJCNN1	0.194	50	0.1439	50
Covertype	0.2565	50	0.1894	50
Higgs	0.1603	50	0.1151	50
MNIST 2 vs 6	0.4988	23.4	0.4463	21.8
MNIST	0.5251	44.6	0.5076	27.2
FMNIST	0.5329	45.2	0.529	27.2
Average	0.3088	45.4	0.2784	40.8

Looking at Figure 7.5 though, conversely to what we would expect the entropic power schedule does not necessarily increase coverage. Inspecting the difference between Figure 7.5a and Figure 7.5b, the setup with the entropic power schedule enabled actually reaches lower coverage. We further investigate the use of the entropic power schedule with a better setup in Table 7.3. We can clearly see why the entropic power schedule performs worse for the image datasets than without the entropic power schedule: the executions per second are drastically lower. It takes too much time to compute the entropy for the large models of the image datasets. It is interesting to note that the entropic power schedule has a lot of impact on the number of executions per second on other datasets as well. For example, for the IJCNN1 and MNIST 2 vs 6 datasets the number of executions per second is much higher which generally means less coverage-increasing test-cases are found: As we see in section 7.3, the coverage tracing mechanism of libFuzzer much time, and the entropic schedule impacts how many test-cases coverage should be traced. The idea of this schedule is that coverage should be traced for more inputs, which would result in a lower amount of executions per second, but as can be seen in the higher executions per second for some datasets, this is not always the case.

Table 7.3: Entropic power schedule with a better setup

Dataset	No entropic $\bar{r}$	No entropic n	No entropic exec/s	Entropic $\bar{r}$	Entropic n	Entropic exec/s
Breast-cancer	0.1618	44.2	283393	0.1715	45.2	282702
Diabetes	0.0605	50	202144	0.0611	50	182490
IJCNN1	0.0532	50	17978	0.0506	50	24954
Covertype	0.0616	50	22808	0.0576	50	20154
Higgs	0.0368	50	4346	0.0295	50	2928
MNIST 2 vs 6	0.1223	21	1699	0.1267	21.6	2263
MNIST	0.2167	43.4	99	0.2271	33.8	16
FMNIST	0.2401	43.4	91	0.2392	33.2	20
Average	0.1191	44.0	66570	0.1204	41.7	64441

7.3. Scalability

Following the rationale of Böhme et al. ([4]), the efficiency with which fuzzers can execute targets is very important and a main reason why grey-box fuzzing works so well. Table 7.4 shows the influence of model size on C++ file generation, ANN-lookup creation (used in chapter 5) and compilation times. It also shows how many runs per thread per second the fuzzer is able to perform on the generated C++ file. Size is calculated by

$t * 2^d * c$ with t the number of trees, d the maximum depth of the tree (2^d resembles the maximum amount of nodes per tree) and c the number of trees per iteration (1 for Random Forests and binary-classification Gradient Boosting, num_classes for multi-class Gradient boosting). This is just an estimation, as for unbalanced trees the maximum depth may not be reached for all leaves, thus resulting in less nodes. We can see that the file generation and compilation times are not completely linear in the size of the model, but there is a clear relation. The Lookup initialization depends on the number of samples in the lookup and the size of the sample (number of features). The image datasets have a lot of features (784) which explains the higher initialization times for those datasets. The number of executions per second per thread in the last column is very low for both MNIST and FMNIST. For MNIST 2 vs 6, the standard deviation is quite high, but this remained when performing the test again. The reason for this can be a big difference in the amount of new coverage found, which influences the runtime (through the fuzzer needing to trace where the increased coverage is located). The number of executions per second for MNIST 2 vs 6 thus depends a lot on the mutations made by the fuzzer. We investigate the executions per second for MNIST as 99 is (probably) too low for FATE to find good

Table 7.4: Influence of model size

Dataset	**Approximate model size**	**File Size**	**File generation (s)**	**Compilation (s)**	**Lookup init (s)**	**Exec/s**
Breast-cancer	256	48.4 kB	0.019	1.33	0.003	283393 (11937)
Diabetes	640	82 kB	0.023	1.46	0.003	202144 (7236)
IJCNN1	15360	871.6 kB	0.13	4.40	0.027	17978 (264)
Covertype	20480	1.4 MB	0.21	2.22	0.053	22808 (892)
Higgs	76800	3.6 MB	0.48	3.94	0.026	4346 (173)
MNIST 2 vs 6	16000	1.3 MB	0.18	2.28	0.41	1699 (583)
MNIST	1024000	44.5 MB	6.0	40.1	3.5	99.5 (3)
FMNIST	1024000	48.1 MB	6.3	42.5	3.5	90.6 (4)

adversarial examples in a reasonable time. We have to keep in mind that this resembles the number of executions per second per thread, so the actual executions per second is 10 times higher as we use 10 concurrent threads.

We tried to improve the execution speed inside the fuzzer in several ways. In chapter 5 we showed how calls to predict can be faked to remove the necessity to query the model for each mutation. While increasing the execution speed, the results were actually better by using the exact model query. Furthermore, in chapter 4 we improved how often a call to write_if_not_exist needs to be performed and always turn on optimization. This resulted in a 71% speedup and when using no coverage instrumentation even a speedup of 650% (598 exec/s), but these are still quite low numbers of executions per second. The MNIST model contains 530865 leaves that can be reached, with only 400 (trees) $*$ 10 (number of classes) = 4000 leaves triggered with the first seed and each execution changing only a number of leaves. The (F)MNIST models should thus be fuzzed longer, to make up for the low amount of executions per second.

When executing FATE as a standalone Evolutionary Algorithm, as we show later in this chapter, the number of executions per second is also limited and similar to executing FATE inside a fuzzer. It is thus not the fuzzer that produces a lot of overhead, but the model size that is very big or unnecessary complexity in the source code. Recalling from chapter 2, the main benefit of fuzzers is the efficiency with which they generate and execute new test input. It might be the case that currently these models are too large to be used efficiently with FATE.

However, executing the important parts of the target outside the fuzzer as standalone programs (e.g. by calling the predict and mutation functions on random features) yields much higher executions per second. Predicting the outcome for random features in python (model in memory) for example gives 3095 predictions per second. To test if writing the model to source code decreased performance we call the target with a forged main function calling LLVMFuzzerTestOneInput with Data consisting of random features in the range [0, 1]. This produces an even higher amount of 10781 predictions per second. Producing the output of the tree ensemble is thus not likely to be a bottleneck. Another possibility is that the mutation functions are the bottleneck. Like before, we execute the target within a main function that calls mutate and crossover on random features repeatedly. This target is able to consecutively perform mutation and crossover 5028 times per second on the MNIST data. The mutations functions are thus also not a bottleneck. One reason why calling these functions "standalone" performs much better is that these experiments were only executed on one thread instead of 10 threads. This allows the CPU to "Turbo-boost" one CPU core which improves the performance. This is however unlikely to attribute for the big difference in executions per second. Another possible explanation is that caching plays an important role. These results however show that performance improvements may be possible, and would benefit FATE a lot.

FATE can be executed with a virtually unlimited amount of concurrent threads. Table 7.5 shows the number of executions per second per thread for the Breast-cancer dataset on the Gradient Boosting model. FATE scales well with the number of threads. These experiments are performed on a laptop with 6 cores (12 threads), which explains the drop in performance from 6 to 12 threads as multiple threads have to be executed on the same core. Fuzzing with one thread is faster because of less overhead and probably also because the processor can "Turbo-boost" when only a small number of cores is under load. The total executions per second, multiplying the executions per second per thread with the number of threads, grows steadily with a larger number of threads.

Table 7.5: Multithreading scalability

Number of Threads	**Executions per second per thread**	**Total executions per second**
1	182773	182773
6	139282	835692
10	107212	1072120
12	100343	1204116

7.4. FATE in comparison with the state-of-the-art

We show our main results through comparing the Performance of FATE with both the exact best solution (MILP approach) [24] and the state-of-the-art LT-attack for tree ensembles by Zhang et al. ([48]). The MILP implementation is taken from [1], which is an adapted version of the MILP formulation of [24] implemented by Hongge Chen at [2]. This formulation is solved using the Gurobi solver [20].

The LT-attack implementation is compiled from source using the most recent commit on main from their github page [3]. We changed one line: in their version they seed the random generator in NormalRandomPoint with the same seed each run, which means that the same initial points will be generated for each run. This may produce seeds that either consistently improve of decrease performance. As we repeat our experiments 5 times, we added a random device for the generator in NormalRandomPoint (neighbour_attack.cc line 1283/1288) to make sure new random points are generated each run thus allowing for a fair comparison. For FATE we discard all adversarial examples with distance > 0.9 as not having identified an adversarial example. This is reasonable, as a value of 0.9 would mean we would be able to almost completely change any victim.

Table 7.6 shows the performance of default FATE on Gradient Boosting models. We increase the number of victims to 500 (previous experiments were conducted on 50 victims, see chapter 4). This is used in related work as well ([48]) and further decreases the possibility that we get invalid results due to attacking victims that are especially suitable for one of the attacks. The Breast-cancer and Diabetes test-sets do not contain 500 valid (the victims should be correctly classified by the model) victims, so we tested on all available valid victims (133 and 124 respectively). Entries with (*) in the MILP column are not repeated due to long running times, while entries with (**) are not repeated and only executed for 50 victims due to very long running times. The tables mention the average adversarial example found ($\bar{r}$), the execution time per victim (t) and possibly also the number of victims that could be attacked (n). Standard deviations are mentioned in brackets behind individual values for the tables containing the main results. This value represents the standard deviation in the result of 5 different runs. For example, FATE may find average adversarial distances {0.1957, 0.1962, 0.1967, 0.1962, 0.1962} in the different runs. The standard deviation is then 0.0004.

FATE is able to find adversarial examples for all victims and its performance is a clear improvement over the baseline performance (Table 4.10). The adversarial distances are not yet on par though with the MILP and

[1] https://github.com/tudelft-cda-lab/GROOT
[2] https://github.com/chenhongge/RobustTrees
[3] https://github.com/chong-z/tree-ensemble-attack

LT-attack attacks, which are generally better with around 1% point for the non-image datasets. The performance of FATE on the image datasets is much worse. For (F)MNIST the reason is that these models are too big to be fuzzed efficiently. The various datasets need different settings for FATE to perform optimally.

Table 7.6: Main result: performance of FATE compared to the state of the art on Gradient Boosting models

	MILP		**LT-attack**		**FATE**	
Dataset	r^*	t	$\bar{r}$	t	$\bar{r}$	t
Breast-cancer	0.1957	0.0070 (0.0001)	0.1957 (0)	0.0008 (0)	0.1962 (0.0005)	0.237 (0.008)
Diabetes	0.0627	0.11 (0.004)	0.0656 (0.0008)	0.0015 (0.0001)	0.0687 (0.0005)	0.231 (0.0001)
IJCNN1	0.0337	24.6 (4.8)	0.0387 (0.0001)	0.011 (0.0006)	0.0453 (0.0002)	0.209 (0.0001)
Covertype	0.0416	50.0 (*)	0.0442 (0.0002)	0.017 (0.0006)	0.0507 (0.0003)	0.207 (0.0001)
Higgs	0.0022 (**)	667 (**)	0.0037 (0)	0.034 (0.0006)	0.0147 (0.0002)	0.206 (0.002)
MNIST 2 vs 6	0.0532 (**)	24.3 (**)	0.0837 (0.0004)	0.476 (0.002)	0.213 (0.003)	1.63 (0.01)
MNIST	-	-	0.0246 (0.0003)	0.919 (0.004)	0.141 (0.002)	1.31 (0.01)
FMNIST	-	-	0.0253 (0.0001)	2.00 (0.02)	0.123 (0.0009)	1.21 (0.01)

Table 7.7 shows that FATE benefits from longer fuzzing times. For "long", victims were fuzzed 5 to 10 times longer than for "short". As expected, this improves the performance of FATE, although the performance is still not on par with the LT-attack.

Table 7.7: Performance of FATE when increasing fuzzing time

	Short			**Long**		
Dataset	$\bar{r}$	n	t	$\bar{r}$	n	t
Breast-cancer	0.1962	133	0.24	0.1957	133	1.25
Diabetes	0.0687	124	0.23	0.0667	124	1.16
IJCNN1	0.0453	500	0.21	0.0427	500	1.10
Covertype	0.0507	500	0.21	0.0476	500	1.11
Higgs	0.0147	500	0.21	0.0115	500	1.10
MNIST 2 vs 6	0.213	499.2	1.63	0.166	500	14.8
MNIST	0.141	499.2	1.31	0.111	500	11.5
FMNIST	0.123	499.6	1.21	0.099	500	10.9

We also investigate how FATE performs on Random Forest models compared to the state-of-the-art. As the setup is tweaked based on the Gradient Boosting models, FATE may perform slightly worse relative to the state-of-the-art. The results in Table 7.8 show that FATE is able to find good adversarial examples on Random Forests as well, and is able to find better adversarial examples than the LT-attack for the Covertype dataset. Entries with (**) are executed only once and for 50 victims, entries with (***) are executed only once for 25 victims due to very long running times. The performance of MNIST for the RF model is much better than the performance for the GB model. Because the RF model has only 1 tree per iteration instead of num_classes trees per iteration for the GB models, the execution speed of the RF model is also much higher due to the smaller model size: 4940 executions per second.

Adversarial examples FATE generated for the image datasets are shown in Figure 7.6. The optimal adversarial example FATE has found is shown on the right together with the optimal adversarial L_∞ distance.

7.4.1. FATE on unseen datasets

To validate if FATE can also find adversarial examples on datasets that were not used to tweak its parameters, we perform experiments on two "unseen" datasets. We choose webspam, the only dataset used in related work that was not used to tweak FATE. The second dataset is Vowel, a multi-class classification problem that is not based around image recognition, such that we can show that multi-class classification also works for datasets that are not based around image recognition. Both datasets are described in Appendix C and the training of their models in Appendix A. The results for these datasets for both Random Forest (RF) and Gradient Boosting (GB) models are shown in Table 7.9. The adversarial distances for the Vowel dataset are close to the adversarial distances the LT-attack produces for both the RF and GB models. The webspam dataset

Table 7.8: Performance of FATE compared to the state-of-the-art on Random Forest models

	MILP		LT-attack		FATE	
Dataset	r^*	t	$\bar{r}$	t	$\bar{r}$	t
Breast-cancer	0.2405	0.010 (0.0003)	0.246 (0.001)	0.0008 (0)	0.260 (0.004)	0.277 (0.007)
Diabetes	0.0692	0.98 (0.02)	0.0713 (0.0003)	0.0023 (0)	0.079 (0.0005)	0.239 (0.0003)
IJCNN1	0.0560 (**)	125 (**)	0.0517 (0)	0.020 (0.0005)	0.059 (0.0004)	0.216 (0.003)
Covertype	0.0777 (**)	566 (**)	0.098 (0.001)	0.104 (0.002)	0.095 (0.001)	0.219 (0.002)
Higgs	0.0084 (***)	2134 (***)	0.0125 (0)	0.062 (0.001)	0.020 (0.0004)	0.207 (0.002)
MNIST 2 vs 6	0.0570 (**)	83.3 (**)	0.1454 (0.0001)	0.462 (0.004)	0.283 (0.003)	1.65 (0.01)
MNIST	-	-	0.032 (0.0005)	0.90 (0.02)	0.079 (0.003)	1.22 (0.002)
FMNIST	-	-	0.0457 (0.0003)	1.72 (0.01)	0.125 (0.002)	1.29 (0.0002)

is more difficult for FATE, probably because the average adversarial distances are very small which would require both the ϵ value to be tweaked. For MILP, (*) means 50 victims are attacked instead of 500.

Table 7.9: Performance of FATE for unseen datasets compared to the state-of-the-art

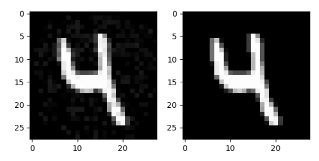
		MILP		LT-attack		FATE	
Dataset	**Model type**	$\bar{r}$	t	$\bar{r}$	t	$\bar{r}$	t
Vowel	GB	0.0417	38.6	0.0436 (0.0003)	0.047 (0.004)	0.0451 (0.0002)	0.206 (0.0002)
Vowel	RF	0.0372	68.0	0.0387 (0.0001)	0.025 (0.0006)	0.0422 (0.0002)	0.208 (0.0001)
Webspam	GB	0.002844 (*)	89.5 (*)	0.0037 (0)	0.048 (0.008)	0.031 (0.001)	0.211 (0.004)
Webspam	RF	0.003565 (*)	113 (*)	0.0054 (0)	0.094 (0.003)	0.028 (0.0004)	0.236 (0.003)

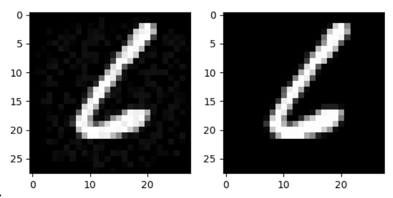
7.5. FATE as standalone Evolutionary Algorithm

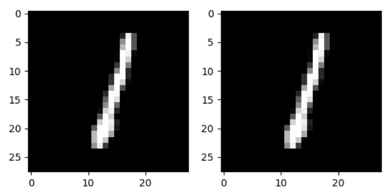
FATE can also be executed outside fuzzers as a standalone Evolutionary Algorithm. A simple main function was developed that saves the best population_size examples and randomly chooses one of the samples in the population to mutate using the custom mutator of FATE. The objective function is used to identify and save adversarial examples, but the structure of this objective function does not influence the result as we do not perform coverage-guided fuzzing anymore. Likewise, the descent on the class-output probabilities from the objective function will not be used anymore and no alternative was implemented for the standalone algorithm. Crossover is implemented between members of the population. However, with a small chance crossover is executed by combining the current seed with random features in the interval [0, 1] to escape local minima. The exact same functions are called as when libFuzzer fuzzer would be used for execution (`LLVMFuzzerCustomMutator`, `LLVMFuzzerCustomCrossOver` and `LLVMFuzzerTestOneInput`). A feature missing from the standalone implementation is the ability to initialize the execution with data points other than the original features of the victim: the "corpus initialization". This would help for being able to attack most victims (as the execution can be seeded with adversarial examples) and could also improve the adversarial examples found (as seen in chapter 5).

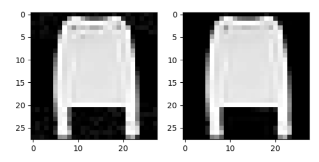
FATE by default adopts a white-box approach where information specific to the type of model is used. In our case this is information about the splitting thresholds and feature importances that are specific to tree ensemble models. FATE can be applied in a black-box setting as well where only the model can be executed ("queried") and its output observed (class-output probabilities and prediction). This makes the approach "portable": any model (i.e. Neural Networks) can be attacked this way as the black-box method does not depend on model internals. We note that the black-box approach still uses the translation from tree ensemble to source code which would fall under the white-box model as this is specific to tree ensembles. This enables "compilation" of the model and high execution speeds. We thus argue that FATE standalone can be a well-performing black-box attack, should it be possible to efficiently query the model, like we are able to do through the compilation of the model. The translation to source code itself, and thus a white-box approach, is by no means necessary for FATE to work.

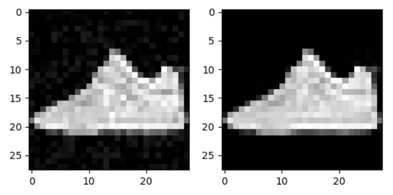
Through experimentation, a good baseline for FATE standalone is identified: mutation chance = 0.5, ϵ = 0.1, population size = 1 (only mutating the best known adversarial example and thus disabling normal

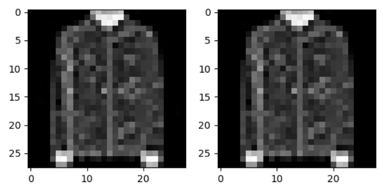

(a) Original (right): 4; adversarial example (left): 6

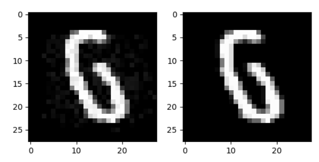

(b) Original (right): 6; adversarial example (left): 2

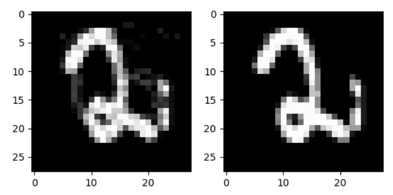

(c) Original (right): 1; adversarial example (left): 7; $L_\infty = 0.010$


(d) Original (right): Pullover; adversarial example (left): shirt


(e) Original (right): Sneaker; adversarial example (left): bag


(f) Original (right): Coat; adversarial example (left): Shirt; $L_\infty = 0.009$


(g) Original (right): 6; adversarial example (left): 2


(h) Original (right): 2; adversarial example (left): 6

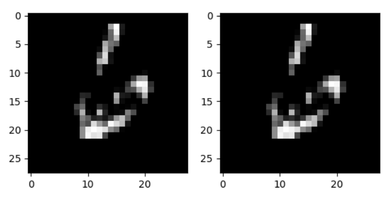

(i) Original (right): 6; adversarial example (left): 2; $L_\infty = 0.002$

Figure 7.6: Adversarial examples generated by FATE for MNIST (first row), FMNIST (second row) and MNIST 2 vs 6 (Third row)

crossover as there are no samples in the population to perform crossover with), crossover with random features chance = 0.001. The performance of FATE as standalone Evolutionary Algorithm is shown in Table 7.10. The number of victims that could be attacked (n) is also mentioned, as not all victims can be attacked due to the absence of "double fuzz": FATE standalone does not support corpus initialization. White-box long (†), which is executed for about 5 to 10 times longer than the normal variants, was only executed once due to time constraints. It is important to keep in mind that the mutation chance and epsilon are not tweaked for the individual datasets, which could potentially improve performance.

Both the black-box and white-box versions of FATE standalone are competitive with the LT-attack for the non-image datasets, finding adversarial examples with comparable distances while even finding better adversarial examples for the Diabetes dataset. The adversarial distances between the white-box and black-box approaches are very similar, with the black-box approach even being able to attack more victims for the MNIST dataset. Both the black- and white-box attacks are unable to attack more than around 50% of the victims for the MNIST 2 vs 6 dataset. This is due to the absence of "double fuzz" (that seeds the execution with adversarial examples, see chapter 5). To solve this, we tried increasing the chance that crossover with random features occurs for this dataset such that more parts of the input space are explored. This however did not increase the number of victims that can be attacked for MNIST 2 vs 6. Note that potentially adversarial examples were found for the other victims (especially when the input consists of many random features through random crossover), but FATE discards all adversarial examples with distance $r > 0.9$ as mentioned before.

When allowing FATE standalone to run longer (only executed for the white-box approach), even adversarial examples with better distances than the state-of-the-art LT-attack on the Diabetes, IJCNN1 and Covertype datasets can be found. As we have established in section 2.5, the LT-attack performs much better than previous black-box approaches on tree ensembles. The comparable performance of FATE standalone (black-box) with the LT-attack suggests that FATE standalone black-box can improve upon currently known black-box attacks for tree ensembles.

We provide the differences between FATE executed in a fuzzer and as standalone Evolutionary Algorithm in Table 7.11. The standalone approach consistently produces better adversarial distances. Note that for the image datasets less victims can be attacked, again due to the absence of "double fuzz".

The distributions of the adversarial distances for the different models for both FATE and FATE standalone are shortly described in Appendix B.

Table 7.10: Important results: performance of variants of FATE standalone vs the LT-attack

	Black-box			White-box			White-box long (†)			LT-attack		
Dataset	$\bar{r}$	n	t	$\bar{r}$	n	t	$\bar{r}$	n	t	$\bar{r}$	n	t
Breast-cancer	0.1958 *(0)*	133	0.213	**0.1957** *(0)*	133	0.213	**0.1957** -	133	1.16	**0.1957** *(0)*	133	0.001
Diabetes	**0.0627** *(0)*	124	0.212	0.0629 *(0.0001)*	124	0.212	**0.0627** -	124	1.16	0.0656 *(0.0008)*	124	0.002
IJCNN1	0.0398 *(0)*	500	0.203	0.0395 *(0.0001)*	500	0.203	**0.0386** -	500	1.10	0.0387 *(0.0001)*	500	0.011
Covertype	0.0443 *(0)*	500	0.205	0.0449 *(0.0003)*	500	0.208	**0.0431** -	500	1.10	0.0442 *(0.0002)*	500	0.017
Higgs	0.0077 *(0)*	500	0.203	0.0079 *(0)*	500	0.204	0.007 -	500	1.10	**0.0037** *(0)*	500	0.034
MNIST 2 vs 6	0.0399 *(0)*	258.2	1.10	0.0362 *(0.0003)*	257.2	1.10	0.0332 -	261	10.1	**0.0837** *(0.0004)*	**500**	0.476
MNIST	0.062 *(0.0006)*	497	1.11	0.0517 *(0.0007)*	466	1.11	0.0434 -	500	10.1	**0.0246** *(0.0003)*	500	0.919
FMNIST	0.0411 *(0)*	461.8	1.11	0.0414 *(0.001)*	457	1.11	0.0324 -	462	10.1	**0.0253** *(0.0001)*	**500**	2.00

Table 7.11: Performance of FATE when executed inside a fuzzer and as standalone Evolutionary Algorithm on both Gradient Boosting (GB) and Random Forest (RF) models

	Fuzzer GB		Standalone black-box GB		Fuzzer RF		Standalone black-box RF	
Dataset	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n	$\bar{r}$	n
Breast-cancer	0.1962	133	0.1958	133	0.2601	135	0.2426	135
Diabetes	0.0687	124	0.0627	124	0.079	121	0.0699	121
IJCNN1	0.0453	500	0.0398	500	0.059	499.4	0.0539	500
Covertype	0.0507	500	0.0443	500	0.0954	499.6	0.0918	490.4
Higgs	0.0147	500	0.0077	500	0.0202	500	0.0152	500
MNIST 2 vs 6	0.2133	499.2	0.0399	258.2	0.2828	493.4	0.0463	252.8
MNIST	0.1411	499.2	0.0619	497	0.0792	496.6	0.0133	457
FMNIST	0.1231	499.6	0.0411	461.8	0.1253	497.8	0.0564	464.2

7.6. Conclusion

In this chapter we have shown that increased code coverage over the full fuzz-target does not imply improved adversarial example distance. There is however a relation between increased coverage and the ability to attack more victims. Using the coverage-guidance of grey-box fuzzers over the full source code to identify interesting seeds increases the average distance of adversarial examples and should thus not be used. Enabling coverage guidance only for the objective function does improve the adversarial examples. This however reduces the fuzzing approach to a normal Evolutionary Algorithm where the increased coverage in the objective function (when a better adversarial example is found) determines the population and the mutation and crossover functions determine the new species. Furthermore, a seed selection algorithm using entropy as a measure of importance was deployed and found effective in improving adversarial examples for the non-image datasets. A limitation of FATE was identified: FATE is less suitable to attack models that are very big due to low executions per second. When only enabling coverage for the objective function running FATE inside a fuzzer is unnecessary as increased "coverage" can then be much more efficiently determined by simply comparing the distance of an adversarial example with the best known (k) distances directly. Running FATE as a standalone Evolutionary Algorithm performs clearly better than when executing FATE inside a fuzzer and comparable with the state-of-the-art LT-attack. Using FATE in a black-box setting, thus disabling features that use knowledge specific to tree ensemble models, FATE standalone performs comparable with using the white-box access model and allows FATE standalone to be used to attack other machine learning models as well.

8 Conclusion

8.1. Discussion

To the best of our knowledge, with FATE we have created the first approach that uses fuzzers to find adversarial examples in machine learning models. FATE has shown the versatility of fuzzers: even without providing domain knowledge, fuzzers are able to find adversarial examples using their built-in mutators through the high rate of model queries that can be performed when compiling machine learning models as source code. Incorporating domain knowledge through a custom mutator however greatly improves the performance of FATE. By leveraging model information such as decision thresholds and feature importances, smart initialization using training-set samples and mutation guidance based on the current adversarial examples, FATE is able to efficiently find good adversarial examples.

Fuzzers optimize code coverage, and there is a strong relation between increasing code coverage and finding more adversarial examples. However, increasing code coverage does not imply finding better adversarial examples. In contrary, we show that lower code coverage is sometimes beneficial for finding better adversarial examples. The fuzzer then puts too much effort in investigating seeds that are not contributing towards finding better adversarial examples. This is partly due to the limited time available: when allowing the fuzzer to run for an unlimited amount of time, at some point exploring all branches in the forest will benefit the quality of the adversarial examples. Experiments however showed that even when allowing much more time than other methods take for finding adversarial examples, (close to) optimal adversarial examples can still not be found.

By only enabling code coverage in the objective function of FATE, adversarial examples are created that are close to state-of-the-art LT-attack [48]. This however reduces fuzzing to a Genetic Algorithm which can also be executed outside fuzzing engines. We explored this with FATE standalone. Through slightly changing the mutator, FATE standalone can be cast into a black-box attack, allowing it to attack other machine learning models as well. FATE standalone (black-box) shows competitive performance with the LT-attack and is able to improve upon its adversarial examples, even finding optimal adversarial examples for the Breast-cancer and Diabetes models. Because of its competitive performance with the white-box LT-attack, which has shown to perform much better than previous black-box attacks when attacking tree ensemble models, there is evidence that FATE standalone can be a well-performing generic black-box attack. The approach of this black-box attack is different than many other black-box attacks that focus on query efficiency; we argue that the number of queries is not important for investigating model robustness when a query can be performed quick enough.

A benefit of FATE over the state-of-the-art is that it allows any kind of perturbation and distance norm, which enables property-based testing and restricting mutations in any way. Odena et al. ([32]) showed the potential of this through Property-Based Testing on Neural Networks. Due to time constraints, experiments were only conducted on the generated L_∞ norm distances. A drawback of FATE is that is requires compilation of the fuzzing target which can be as much as 4 minutes for the largest models we have tested with ((F)MNIST). In general, FATE also requires more running time than the current state-of-the-art.

8.1.1. Limitations

While producing valid and reasonably good adversarial examples, FATE has a number of limitations. Its performance depends a lot on hyper-parameters, which can be cumbersome to tune for each dataset individually. We define an acceptable baseline for the parameters but performance may be much better with hyper-parameters that are tweaked to the respective dataset. Furthermore, finding the right combination of parameters and settings is very much a "chicken and egg" problem: the performance of the mutator depends on fuzzer settings such as mutation depth, but the right mutation depth depends on the performance of the mutator. It may thus be that other default parameters will perform better. The parameters were tweaked on the Gradient Boosting models, but it may be the case that FATE will perform better with different parameters on the Random Forest models.

Fuzzing depends a lot on the quality of random mutations. Through repeating experiments multiple times and for a multitude of victims we tried to limit the influence of this randomness, but it can always be the case that a result is either much better or worse than it would be on average. Furthermore, experiments were conducted on a laptop. Laptops are not the most ideal machines to conduct experiments on due to their cooling problems and resulting CPU throttling. When possible, extra airflow was created by lifting the laptop to limit the cooling problems. To conclude, performance was tested on a wide variety of datasets, validating its performance on datasets that were not used for tweaking its settings. However, it can still be the case that FATE will not perform well on different (types of) datasets.

8.2. Conclusion

To answer **"How can Fuzzers be used to generate adversarial examples for Tree Ensembles?"** we introduced FATE, a tool to reduce the problem of finding adversarial examples in tree ensembles to a fuzzing problem. FATE is able to find adversarial examples for all victims on datasets that are commonly used in the field, while being able to find optimal adversarial examples for the Breast-cancer dataset. Although for the rest of the datasets the adversarial examples FATE creates are comparable to the state-of-the-art in distance, they generally have a slightly increased distance and take more time to compute. FATE has a clear limitation on model size: the very large (F)MNIST models have very low execution speeds resulting in high adversarial distances (0.13 vs 0.025).

How can Fuzzing for adversarial examples in Tree Ensembles be improved by leveraging information sources? Three potential sources of information were identified: the model itself, the training set and information from the execution. FATE was improved by initializing the fuzzer with adversarial examples through searching over samples of the opposite class with an ANN approach. Splitting condition information was used to move mutations right to decision boundaries and feature importances are used to bias mutation. During execution information about the current input and features of the victim are used to bias (the magnitude of) mutations of features.

How does FATE perform with different fuzzing engines? FATE was tested with AFL++, libFuzzer and honggfuzz. All mentioned fuzzing engines are able to find adversarial examples using their built-in mutators, which inspect the source code at compilation time for interesting input and use information (such as input used for comparison instructions) from single runs to guide mutation. The fact that adversarial examples can be found using the built-in mutator shows that breakthroughs in fuzzing engines may improve the performance of FATE in the future. The created adversarial examples are generally much worse though than when using the custom mutator of FATE while AFL++ was not able to attack the image datasets. LibFuzzer and AFL++ are compared when using the custom mutator, with libFuzzer yielding the best results because of a much higher number of executions per second. Running the fuzzer longer per victim improves the created adversarial examples for both fuzzers.

8.3. Future Work

Currently additional coverage in the fuzz-target does not necessarily contribute towards finding better adversarial examples: additional code coverage can even mean worse adversarial examples. This is problematic, as fuzzers optimize code coverage. Future work can focus on developing a metric to determine if a leaf is worth the time investment for the fuzzer to try to reach. This way coverage guidance in the forest can be beneficial as well. Through dividing the tree into branches that are "interesting" and "uninteresting", all uninteresting parts of the tree can be put in separate functions that are excluded from coverage guidance at compile-time through SanitizerCoverage's `__attribute__((no_sanitize("coverage")))`. A potential metric can be the probability p of input belonging to the original class (for Random Forests), leaves are then interesting if $p < t$ (experiment with different values) or as an alternative when the highest probability of the input belonging to another class is close to the probability of the input belonging to the original class. For Gradient Boosting the equivalent of this would be the regression value $r < -t$ (for original class 1) or $r > t$ (for original class 0). This can be combined with an ϵ criterion that only instruments the leaves that are within an ϵ ball around the victim. The metric of reached coverage will now tell if all interesting leaves could be reached and thus also how far the fuzzer has progressed. This approach would however require a new target to be compiled for every victim, which can be too time-consuming. Ideally, an approach should thus be created that can change coverage guidance for branches in the forest at runtime, allowing a single target to be compiled for all victims.

Furthermore, some of the models (at least MNIST and FMNIST) are too big to be efficiently executed by the fuzzer as overhead (coverage tracing) takes too much time. Future work could try to attack these models in smaller steps, such as attacking the model per tree, and then combine these sub-results in a final stage, leaving the fuzzer to perform expensive computations on the complete model for a shorter amount of time. A way to limit execution time that can be added can be to implement an early time-out mechanism, for example when no better adversarial example was found for t milliseconds.

FATE is dependent on many hyper-parameters. Through experimentation a baseline setup was identified, but results show that the quality of the generated adversarial examples is still very dependent on both the chance that a single feature is mutated and the ϵ restriction of mutations. Through performing a search over these parameters on a small subset of all victims, good parameters can be determined that then can be used to fuzz all victims. This would make FATE generally applicable without the need for parameter tuning by the user.

Directed fuzzers are designed to designate most execution time (energy) towards seeds that seem to contribute towards reaching certain targets in the code. When appointing the branch in the objective function with the lowest adversarial distance as target, priority will be given to seeds that seem to contribute towards finding better adversarial examples. This currently is a bottleneck for FATE is shown by the performance difference between FATE and FATE standalone. FATE standalone only mutates the best currently known adversarial example, while FATE also mutates examples that were previously saved to the corpus. FATE was implemented for AFLGo ([4]), the current baseline for directed fuzzers. AFLGo does however not support custom mutators which is essential for good performance of FATE, so this approach was abandoned. Future work can either merge AFLGo with for example AFL++ such that custom mutators can be used or implement directed fuzzing in a different fuzzer that supports custom mutators to investigate if this can give FATE the performance boost it needs to improve upon the state-of-the-art.

FATE allows for any type of perturbation or distance norm. Using the objective and mutation functions, only specialized and restricted mutations can be allowed for finding adversarial examples. This can be useful for validating robustness of models for which only certain type of input is possible or allowed. Future work can focus on investigating how this can be beneficial for testing real-world applications. Also, performance of FATE and FATE standalone for different distance norms such as $L_2, L_1 \text{and} L_0$ can be tested.

We conclude with the black-box approach of FATE standalone, that shows competitive performance on multiple datasets to the white-box LT-attack. As the LT-attack performed better than multiple previous black-box attacks, FATE standalone is a promising new black-box attack. The most promising direction for future work would be to see how FATE standalone performs on the popular Neural Networks, where gradient-based methods (that may take a long time) are the current baseline attack methods.

Bibliography

[1] Evan Ackerman. Three small stickers in intersection can cause tesla autopilot to swerve into wrong lane, 04 2019. URL https://spectrum.ieee.org/cars-that-think/transportation/self-driving/three-small-stickers-on-road-can-steer-tesla-autopilot-into-oncoming-lane. Accessed on 2021-03-12.

[2] Maksym Andriushchenko and Matthias Hein. Provably robust boosted decision stumps and trees against adversarial attacks. *arXiv preprint arXiv:1906.03526*, 2019.

[3] Marcel Böhme and Soumya Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, 42(4):345–360, 2015.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134020. URL https://doi.org/10.1145/3133956.3134020.

[5] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 678–689, 2020.

[6] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248*, 2017.

[7] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019. doi: 10.1109/TSE.2017.2785841.

[8] Stefano Calzavara, Claudio Lucchese, Gabriele Tolomei, Seyum Assefa Abebe, and Salvatore Orlando. Treant: training evasion-aware decision trees. *Data Mining and Knowledge Discovery*, 34(5):1390–1420, 2020.

[9] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2018.02.002. URL https://www.sciencedirect.com/science/article/pii/S0167404818300658.

[10] Hongge Chen, Huan Zhang, Duane Boning, and Cho-Jui Hsieh. Robust decision trees against adversarial examples. In *International Conference on Machine Learning*, pages 1122–1131. PMLR, 2019.

[11] Hongge Chen, Huan Zhang, Si Si, Yang Li, Duane Boning, and Cho-Jui Hsieh. Robustness verification of tree-based models. In *Advances in Neural Information Processing Systems*, pages 12317–12328, 2019.

[12] Jianbo Chen, Michael I Jordan, and Martin J Wainwright. Hopskipjumpattack: A query-efficient decision-based attack. In *2020 ieee symposium on security and privacy (sp)*, pages 1277–1294. IEEE, 2020.

[13] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[14] Minhao Cheng, Thong Le, Pin-Yu Chen, Jinfeng Yi, Huan Zhang, and Cho-Jui Hsieh. Query-efficient hard-label black-box attack: An optimization-based approach. *arXiv preprint arXiv:1807.04457*, 2018.

[15] Minhao Cheng, Simranjit Singh, Patrick Chen, Pin-Yu Chen, Sijia Liu, and Cho-Jui Hsieh. Sign-opt: A query-efficient hard-label adversarial attack. *arXiv preprint arXiv:1909.10773*, 2019.

[16] Marc Heuse et al. Aflplusplus, 2021. URL https://aflplus.plus/. Accessed on 2021-05-28.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[18] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.

[19] Google. Oss-fuzz: Continuous fuzzing for open source software, 06 2020. URL `https://github.com/google/oss-fuzz`. Accessed on 2020-11-25.

[20] LLC Gurobi Optimization. Gurobi - the fastest solver, 2021. URL `https://www.gurobi.com/`. Accessed on 2021-06-04.

[21] Hongfuzz. Honggfuzz, 02 2021. URL `https://github.com/google/honggfuzz`. Accessed on 2021-03-12.

[22] Noora Hyvärinen. Super awesome fuzzing, part one, 06 2017. URL `https://blog.f-secure.com/super-awesome-fuzzing-part-one/`. Accessed on 2021-03-12.

[23] Kyle D Julian, Shivam Sharma, Jean-Baptiste Jeannin, and Mykel J Kochenderfer. Verifying aircraft collision avoidance neural networks through linear approximations of safe regions. *arXiv preprint arXiv:1903.00762*, 2019.

[24] Alex Kantchelian, J Doug Tygar, and Anthony Joseph. Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning*, pages 2387–2396, 2016.

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[26] Bogdan Kulynych, Jamie Hayes, Nikita Samarin, and Carmela Troncoso. Evading classifiers in discrete domains with provable optimality guarantees. *arXiv preprint arXiv:1810.10939*, 2018.

[27] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[28] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 521–532, 2019.

[29] LLVM. libfuzzer – a library for coverage-guided fuzz testing, 2021. URL `https://llvm.org/docs/LibFuzzer.html`. Accessed on 2021-03-10.

[30] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[31] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[32] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.

[33] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. doi: 10.1109/TSE.2019.2941681.

[34] Katyanna Quach. Researchers trick tesla into massively breaking the speed limit by sticking a 2-inch piece of electrical tape on a sign, 02 2020. URL `https://www.theregister.com/2020/02/20/tesla_ai_tricked_85_mph/`. Accessed on 2021-03-21.

[35] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[36] scikit learn. scikit-learn: Machine learning in python, 01 2021. URL https://scikit-learn.org/stable/. Accessed on 2021-03-25.

[37] Spotify. Annoy, 2021. URL https://github.com/spotify/annoy. Accessed on 2021-05-03.

[38] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[39] The Clang Team. Sanitizercoverage - clang 13 documentation, 2021. URL https://clang.llvm.org/docs/SanitizerCoverage.html. Accessed on 2021-06-15.

[40] Hoang Ngoc Thanh and Tran Van Lang. Evaluating effectiveness of ensemble classifiers when detecting fuzzers attacks on the unsw-nb15 dataset. *Journal of Computer Science and Cybernetics*, 36(2):173–185, 2020.

[41] Daniël Vos and Sicco Verwer. Efficient training of robust decision trees against adversarial examples. *arXiv preprint arXiv:2012.10438*, 2020.

[42] XGBoost. Xgboost documentation, 2020. URL https://xgboost.readthedocs.io/en/latest/. Accessed on 2021-03-25.

[43] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: A coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 146–157, 2019.

[44] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[45] Yao-Yuan Yang, Cyrus Rashtchian, Yizhen Wang, and Kamalika Chaudhuri. Robustness for non-parametric classification: A generic attack and defense. In *International Conference on Artificial Intelligence and Statistics*, pages 941–951. PMLR, 2020.

[46] Michal Zalewski. american fuzzy lop (2.52b), 2021. URL https://lcamtuf.coredump.cx/afl/. Accessed on 2021-03-10.

[47] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Search-based fuzzing. In *The Fuzzing Book*. Saarland University, 2019. URL https://www.fuzzingbook.org/html/SearchBasedFuzzer.html. Retrieved 2019-12-21.

[48] Chong Zhang, Huan Zhang, and Cho-Jui Hsieh. An efficient adversarial attack for tree ensembles. *Advances in Neural Information Processing Systems*, 33, 2020.

[49] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.

[50] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Camtepe Seyit, and Yang Xiang. Defuzz: Deep learning guided directed fuzzing. *arXiv preprint arXiv:2010.12149*, 2020.

A
Training Procedure

Table A.1: Details of the trained models

			Trees		**Depth**		**Test accuracy**	
Dataset	**features**	**classes**	RF	GB	RF	GB	RF	GB
Breast-cancer	10	2	4	4	6	6	0.985	0.971
Diabetes	8	2	25	20	8	5	0.786	0.805
IJCNN1	22	2	100	60	8	8	0.946	0.957
Covertype	54	2	160	80	10	8	0.791	0.877
Higgs	28	2	300	300	8	8	0.702	0.725
MNIST 2 vs 6	784	2	1000	1000	4	4	0.984	0.998
MNIST	784	10	400	400	8	8	0.93	0.978
FMNIST	784	10	400	400	8	8	0.824	0.899
Webspam (unigram)	256	2	100	100	8	8	0.960	0.989
Vowel	10	11	50	50	10	10	0.574	0.450

Before training, the data is scaled per column to the range [0, 1]. For the image data (MNIST, FMNIST and MNIST 2 vs 6) the scaling was performed over all columns at the same time, as the feature ranges are equal across the columns. We assume the absolute highest and lowest values per feature are present in the training set. When no separate training/testing sets are specified, 20% of the data was used for testing, the rest for training. For both Gradient Boosting and Random Forest min_samples_leaf are set to 5 and min_samples_split to 10. We consider this good practise, to make the classifier a bit more robust (there will be less very small high-confidence decision regions that make adversarial attacking easier). For Gradient Boosting, the learning_rate is set to 0.1. The models with which we are testing are trained with scikit-learn [36], but FATE can for example also handle Random Forests in the XGBoost [42] JSON format. Support for models trained with different methods can be added easily. Details about the created models can be found in Table A.1. We can see that the Gradient Boosting models generally have a little higher accuracy on the testing set than the Random Forests models. There is a notable difference between Random Forest (RF) and Gradient Boosting (GB) accuracy for the Covertype dataset.

B

Extra visualisations and tables

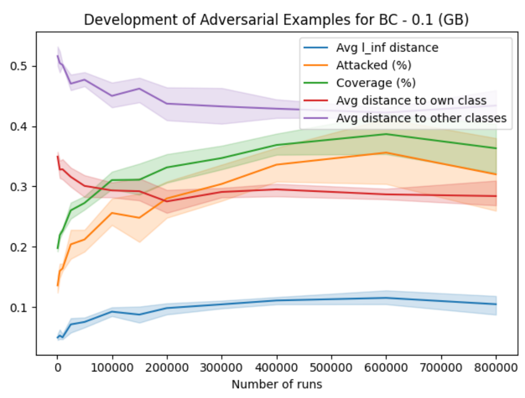
Interesting to note is the distribution of the distances of adversarial examples, which we show for 50 victims in Table B.1 for FATE and Table B.2 for FATE standalone. These tables show the minimal, maximal, mean, median and standard deviation for the adversarial distances of 50 victims. The median generally lies (much) lower than the mean, which means that there are generally more victims that can be attacked with lower adversarial distance with fewer victims that require larger perturbations to attack. Looking at the table for FATE standalone, we can see that for most datasets at least one victim could be attacked with maximal 0.5% point difference of features. Only MNIST 2 vs 6 requires a larger perturbation of around 3% point.

Table B.1: Distribution of the identified adversarial examples for FATE

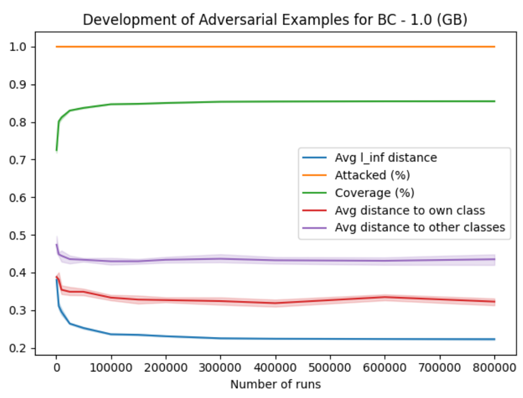
Dataset	**Min**	**Max**	**Mean**	**Median**	**Std**
Breast-cancer	0.0041	0.6111	0.2159	0.1667	0.1644
Diabetes	0	0.1885	0.0601	0.0594	0.0425
IJCNN1	0.0026	0.1678	0.0445	0.0308	0.0376
Covertype	0.0003	0.2884	0.0507	0.0398	0.0521
Higgs	0.0001	0.0475	0.014	0.0119	0.0093
MNIST 2 vs 6	0.041	0.8549	0.2216	0.1794	0.171
MNIST	0.0235	0.4549	0.1217	0.0873	0.1088
FMNIST	0.0294	0.6059	0.1318	0.1098	0.1092

Table B.2: Distribution of the identified adversarial examples for FATE standalone

Dataset	**Min**	**Max**	**Mean**	**Median**	**Std**
Breast-cancer	0.0041	0.6111	0.2115	0.1668	0.1595
Diabetes	0	0.1884	0.057	0.0528	0.0408
IJCNN1	0.0014	0.1559	0.0394	0.025	0.0338
Covertype	0.0003	0.2742	0.0458	0.0315	0.0496
Higgs	0.0002	0.0189	0.0072	0.0062	0.0046
MNIST 2 vs 6	0.029	0.0649	0.0451	0.042	0.0116
MNIST	0.0053	0.5557	0.0654	0.0401	0.0924
FMNIST	0.0046	0.1171	0.0399	0.0414	0.0252

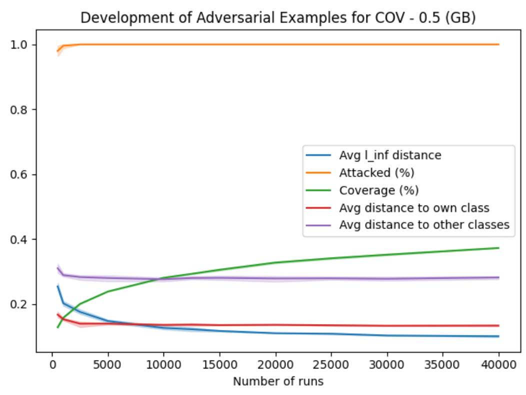


(a) Breast-cancer curve with sub-optimal setup

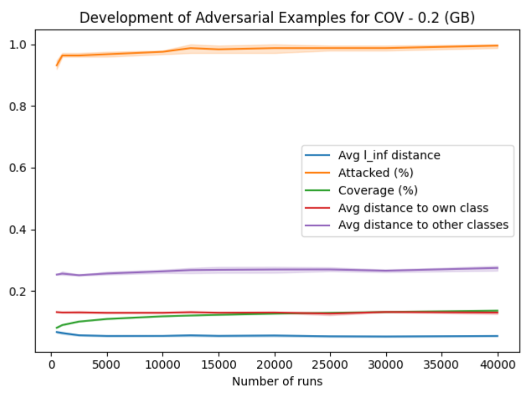


(b) Breast-cancer curve with a setup producing optimal Adversarial examples

Figure B.1: Coverage curves with optimal and less optimal setups for the Breast-cancer dataset



(a) Covertype curve with sub-optimal setup (adversarial example distance ≈ 0.098)



(b) Covertype curve with a setup producing good Adversarial examples (adversarial example distance ≈ 0.053)

Figure B.2: Coverage curves with optimal and less optimal setups for the Covertype dataset

Table B.3: Different step sizes for the distance descent

	0.001		**0.005**		**0.01**		**0.1**	
Dataset	r	n	r	n	r	n	r	n
Breast-cancer	0.1566	43	0.1523	42	0.1577	43	0.1568	42.2
Diabetes	0.0726	50	0.0739	50	0.0711	50	0.08	50
IJCNN1	0.0843	50	0.086	50	0.0851	50	0.0858	50
Covertype	0.1102	50	0.1069	50	0.1098	50	0.1091	50
Higgs	0.0691	50	0.069	50	0.0709	50	0.0678	50
Average	0.0986	48.6	0.0976	48.4	0.0989	48.6	0.0999	48.4

Table B.4: Different step intervals

	Decreasing		**Decreasing +**		**Decreasing ++**	
Dataset	r	n	r	n	r	n
Breast-cancer	0.1551	42.6	0.1557	43	0.1537	42.6
Diabetes	0.0678	50	0.0709	50	0.0662	50
IJCNN1	0.0828	50	0.0848	50	0.0834	50
Covertype	0.1101	50	0.109	50	0.1097	50
Higgs	0.0682	50	0.0658	50	0.0673	50
Average	0.0968	48.5	0.0972	48.6	0.0961	48.5

C Datasets

The following datasets were mainly selected because they were used in related work [11, 48]. It is a combination of binary and multi-class datasets with varying amounts of features and model sizes. The Breast-cancer and Diabetes datasets are very small and generally easy to attack. MNIST 2vs 6, MNIST and FMNIST are image recognition datasets and have a lot of features (784). They have the largest models and are generally the most difficult to attack for FATE.

C.1. Breast-cancer

Link: `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#breast-cancer`. Winsconcin Breast-Cancer database. The dataset consists of 546 samples in the training set and 137 samples in the testing set. There are 2 classes and each sample has 10 features. The distribution of samples among the classes can be seen in Figure C.1.

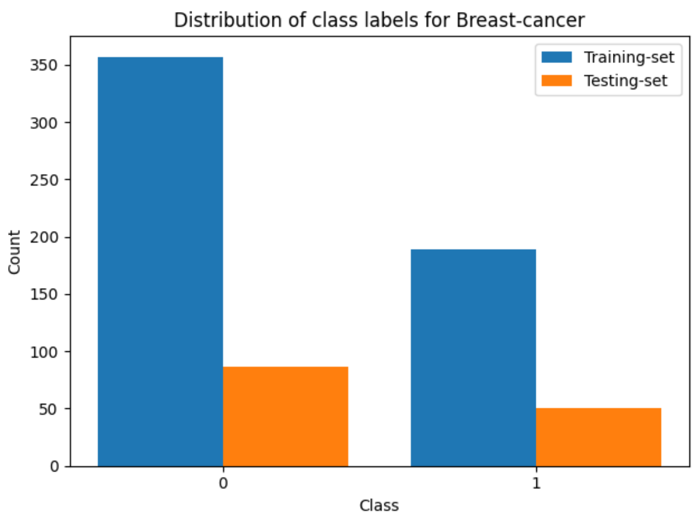


Figure C.1: Distribution of class labels for Breast-cancer

C.2. Diabetes

Link: `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#diabetes`. AIM ’94 Diabetes dataset. The dataset consists of 614 samples in the training set and 154 samples in the testing set. There are 2 classes and each sample has 8 features. The distribution of samples among the classes can be seen in Figure C.2.

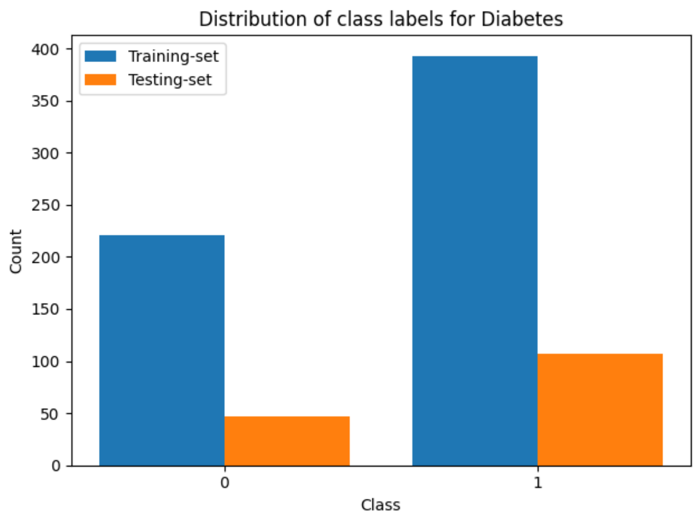


Figure C.2: Distribution of class labels for Diabetes

C.3. IJCNN1

Link: `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#ijcnn1`. IJCNN 2001 neural network competition dataset. The dataset consists of 49990 samples in the training set and 91701 samples in the testing set. There are 2 classes and each sample has 22 features. The distribution of samples among the classes can be seen in Figure C.3.

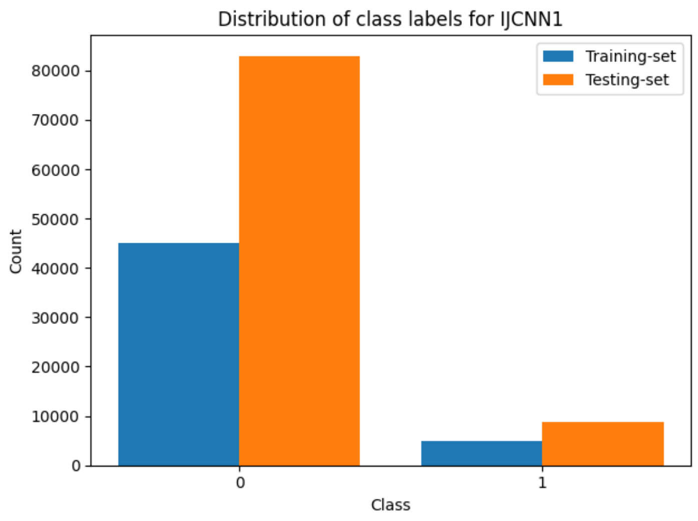


Figure C.3: Distribution of class labels for IJCNN1

C.4. Covertype Binary

Link: `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#covtype.binary`. The dataset consists of 464809 samples in the training set and 116203 samples in the testing set. There are 2 classes and each sample has 54 features. The distribution of samples among the classes can be seen in Figure C.4.

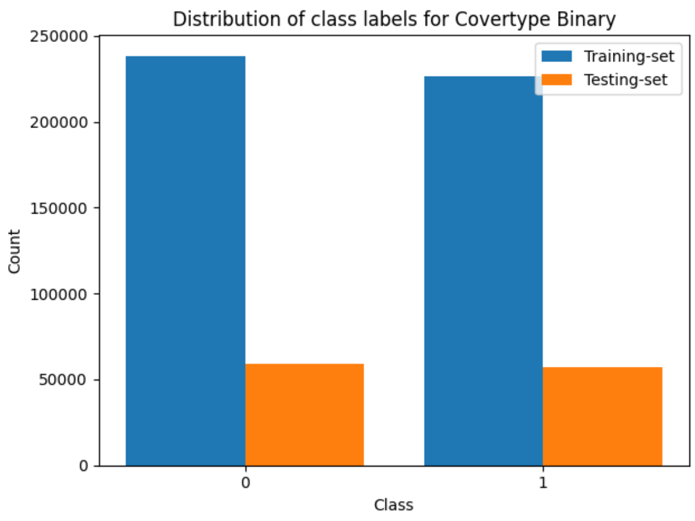


Figure C.4: Distribution of class labels for Covertype Binary

C.5. Webspam

Link: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#webspam. Unigram (occurrences of single words) over web pages that are created to manipulate search engines and deceive web users by De Wang, Danesh Irani, and Calton Pu. "Evolutionary Study of Web Spam: Webb Spam Corpus 2011 versus Webb Spam Corpus 2006". In Proc. of 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2012). Pittsburgh, Pennsylvania, United States, October 2012. The dataset consists of 280000 samples in the training set and 70000 samples in the testing set. There are 2 classes and each sample has 256 features. The distribution of samples among the classes can be seen in Figure C.5.

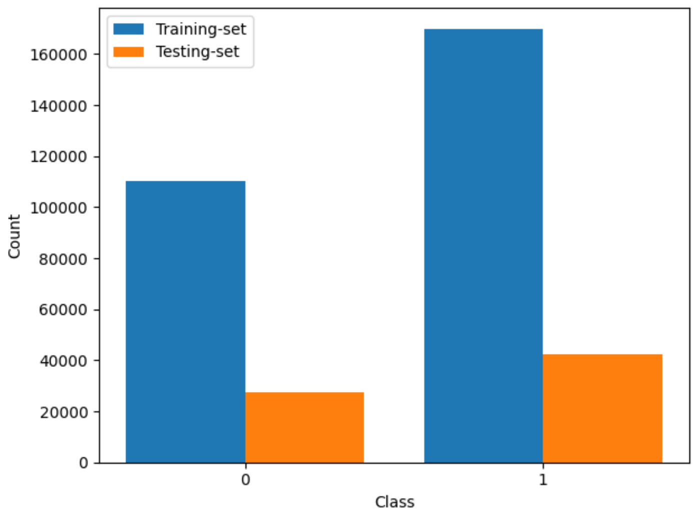


Figure C.5: Distribution of class labels for webspam

C.6. Higgs

Link: https://www.openml.org/d/23512. The original, full Higgs dataset is too big for our hardware to train a model. We thus used this smaller version with 98050 instances. The dataset consists of 78440 samples in the training set and 19610 samples in the testing set. There are 2 classes and each sample has 28 features. The distribution of samples among the classes can be seen in Figure C.6.

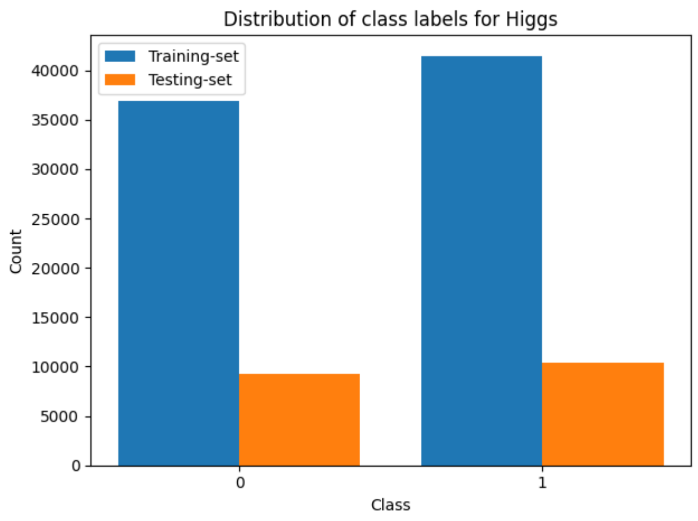


Figure C.6: Distribution of class labels for Higgs

C.7. MNIST 2 vs 6

MNIST with only the digits '2' and '6'. Easier to classify and more difficult to attack than the original MNIST dataset. The dataset consists of 11876 samples in the training set and 1990 samples in the testing set. There are 2 classes and each sample has 784 features. The distribution of samples among the classes can be seen in Figure C.7.

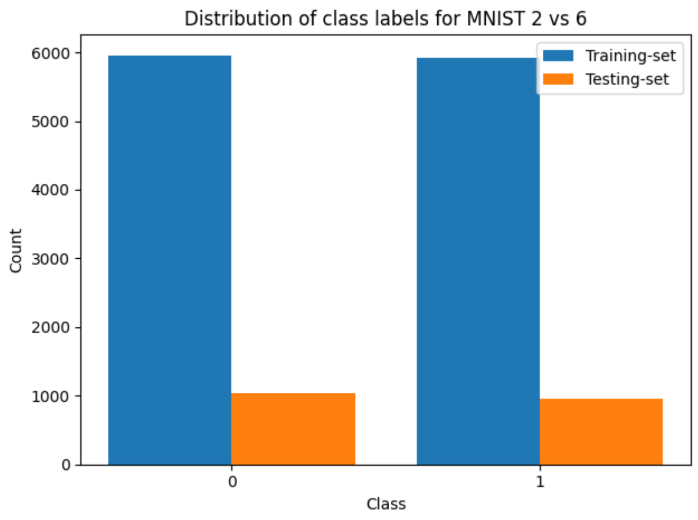


Figure C.7: Distribution of class labels for MNIST 2 vs 6

C.8. Vowel

Link: `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#vowel`. The dataset consists of 528 samples in the training set and 462 samples in the testing set. There are 11 classes and each sample has 10 features. The distribution of samples among the classes can be seen in Figure C.8.

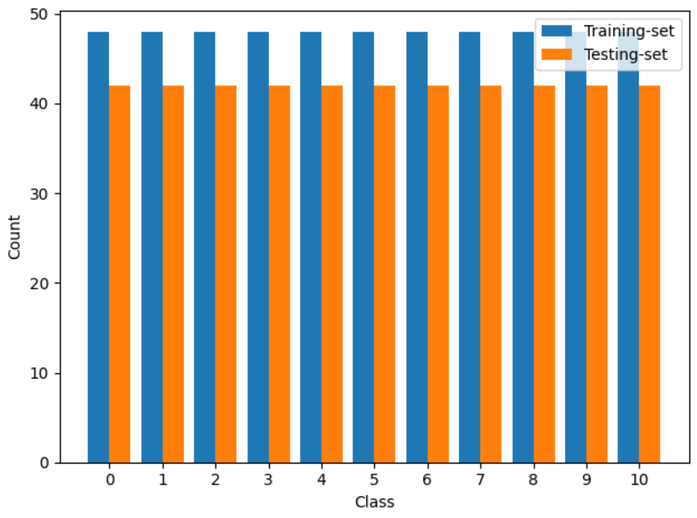


Figure C.8: Distribution of class labels for Vowel

C.9. MNIST

Link: `https://deepai.org/dataset/mnist`. Image recognition / Digit classification challenge. The dataset consists of 60000 samples in the training set and 10000 samples in the testing set. There are 10 classes and each sample has 784 features. The distribution of samples among the classes can be seen in Figure C.9.

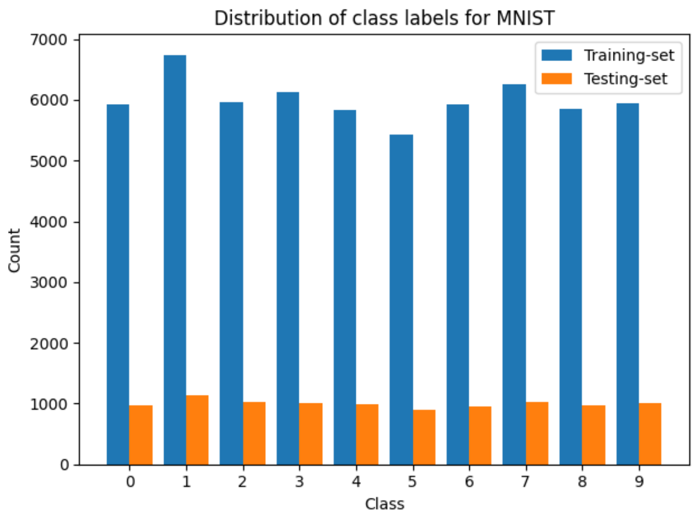


Figure C.9: Distribution of class labels for MNIST

C.10. FMNIST

Link: `https://github.com/zalandoresearch/fashion-mnist`. More difficult drop-in replacement for MNIST created by Zalando Research. The dataset consists of 60000 samples in the training set and 10000 samples in the testing set like MNIST. There are 10 classes and each sample has 784 features. The distribution of samples among the classes can be seen in Figure C.10.

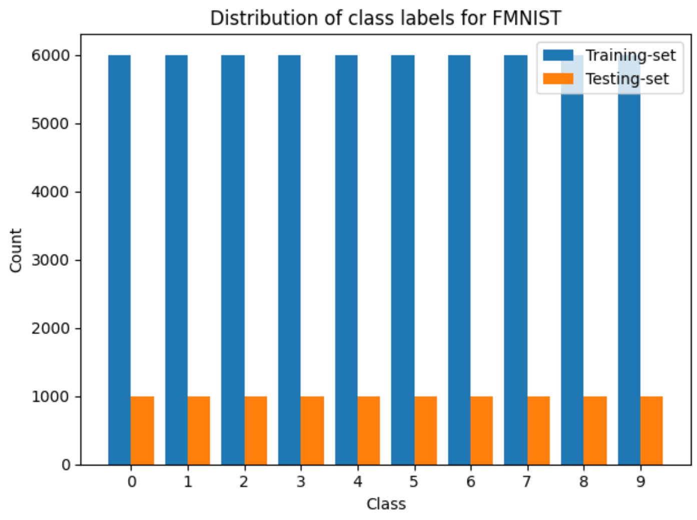


Figure C.10: Distribution of class labels for FMNIST

D Full fuzz target

This chapter shows the target (based on the Gradient Boosting model for the Breast-cancer dataset) as generated by FATE for libFuzzer. Some parts of the code are re-ordered for readability; some code will occur in a different order in the generated C++ file. Also, in the code a structure like "{% if pred %} ... " may be present. This signals that the respective code is only included if "pred" is True; it may also include an else branch. The breast-cancer dataset has 10 features and 2 classes. Its model consists of 4 trees.

D.1. Initialization

This part of the code (shown in Listing D.1) consists of including the required headers, initializing some variables and processing information about the victim that is currently being fuzzed. The feature importances, thresholds and prior mutate chances are determined by FATE and inserted on compile-time. As we fuzz one victim at a time, not all information that is needed about the victim can be compiled into the program. In LLVMFuzzerInitialize we read the id of the victim (check_num), its original class and its original features. We also initialize the thresholds_per_feature vector. The arguments to the fuzzer are then updated to be used further by libFuzzer/honggfuzz itself. AFL++ does not have the same initialization interface as libFuzzer and honggfuzz. The information about the victim is thus initialized via an alternative method described in subsection D.6.1, and the thresholds are initialized in the mutator (Listing D.6).

```
#include <stdint.h>
#include <math.h>
#include <stdio.h>
#include <array>
#include <stdlib.h>
#include <stddef.h> // size_t
#include <string>
#include <sys/types.h>
#include <sys/stat.h> // check if file exists
#include <unistd.h>   // POSIX OS API
#include <dirent.h>   // access to directories

#include <iostream>
#include <fstream>
#include <random>
#include <sstream>
#include <iterator>

#include <chrono>
#include <ctime> // time

#include <bits/stdc++.h> // all standard headers and STL include file
using namespace std;
```

```
int ppp = 0; // is set in the probability steps to avoid simplification
    by the optimizer
int ddd = 0; // is set in the distance steps to avoid simplification by
    the optimizer
double min_e = 1e-05; // minimal perturbation from the feature
   threshold

vector<double> original_features; // victim features, set on init
int check_num;                    // id of the victim
int original_class;               // original class of the victim
double best_ae_dist = 1.1;        // all AE will have distance < 1.0
double cur_ae_dist = 1.1;         // special value that resembles "no
   AE"

// feature importances as probabilities with sum 1
double feature_importances[] = {0.02051328891276369,
   0.013680900869554249,
                                0.7984335118650101,
                                   0.06693435249146118,
                                0.003751919312307732,
                                   0.014649060041497394,
                                0.07321704078129188,
                                   0.004224237171287643,
                                0.004595688554826086, 0.0};

// 0 if feature is not used in a splitting condition,
// else DEFAULT_MUTATE_CHANCE
double prior_mutate_chances[] = {0.1, 0.1, 0.1, 0.1, 0.1,
                                 0.1, 0.1, 0.1, 0.1, 0};

// for producing semi random numbers
std::default_random_engine en;
// used whn something should happen with a certain chance
std::uniform_real_distribution<double> uniform_dist(0, 1);
// in general, abs(draw) < 1
std::normal_distribution<double> mutation_dist(0, 0.34);
// at which index to combine two victims
std::uniform_int_distribution<int> crossover_dist(0, 10);

// initialized in LLVMFuzzerInitialize
vector<vector<double>> thresholds_per_feature;
vector<double> th_f_0 = {0.0721409, 0.0767784, 0.0813804,
                         0.0845354, 0.0848987,
                         0.0853971, 0.0862544, 0.0868369,
                         0.0916151, 0.0916932, 0.0925671};
vector<double> th_f_1 = {0.5, 0.6111111};
vector<double> th_f_2 = {0.1666667, 0.3888889};
vector<double> th_f_3 = {0.0555556, 0.1111111, 0.1666667};
vector<double> th_f_4 = {0.0555556, 0.5};
vector<double> th_f_5 = {0.1666667, 0.2777778};
vector<double> th_f_6 = {0.0555556, 0.1666667, 0.2777778,
                         0.3888889, 0.5, 0.7222222, 0.8333333};
vector<double> th_f_7 = {0.2777778, 0.3888889, 0.6111111};
vector<double> th_f_8 = {0.1666667, 0.3333333};
// feature 9 is never used as splitting condition for this model
```

```
vector<double> th_f_9 = {};

// initialization function used for initializing the thresholds vector
// and reading information about the victim from the supplied arguments
extern "C" int LLVMFuzzerInitialize(int *argc, char ***argv)
{
  thresholds_per_feature.push_back(th_f_0);
  thresholds_per_feature.push_back(th_f_1);
  thresholds_per_feature.push_back(th_f_2);
  thresholds_per_feature.push_back(th_f_3);
  thresholds_per_feature.push_back(th_f_4);
  thresholds_per_feature.push_back(th_f_5);
  thresholds_per_feature.push_back(th_f_6);
  thresholds_per_feature.push_back(th_f_7);
  thresholds_per_feature.push_back(th_f_8);
  thresholds_per_feature.push_back(th_f_9);

  int num_args = *argc;
  char **args = *argv;
  check_num = atoi(args[1]);
  int num_features = atoi(args[2]);
  original_class = atoi(args[3]);
  int start_index = 4;
  int num_fuzzer_args = num_args - num_features - start_index;
  vector<double> original(num_features, 0.0);

  for (int i = start_index; i < start_index + num_features; i++)
  {
    original[i - start_index] = atof(args[i]);
  }
  original_features = original;

  char **new_argv = 0;
  new_argv = new char *[num_fuzzer_args + 1];
  new_argv[0] = args[0];
  for (int i = 0; i < num_fuzzer_args; i++)
  {
    new_argv[i + 1] = args[start_index + num_features + i];
  }

  *argv = new_argv;
  *argc = num_fuzzer_args;

  return 0;
}
```

Listing D.1: Initialization

D.2. Adversarial Example handling

When an adversarial example is found, it is saved to a file. Recalling from chapter 3, saving all adversarial examples we find would overflow the OS I/O system: tens of thousands of adversarial examples can be generated in short periods of time. We thus save adversarial examples only if an adversarial example was found in a new interval. As the filename for an interval is consistent, the fuzzer can this way determine if a similar adversarial example was found already in an earlier run. The stat method of file_exists is very fast: it can easily stat more than 1 million times per second so this will not likely form a bottleneck. The adversarial features are written with 15 digits of precision, to limit floating-point inconsistencies between the scikit-learn and C++

models as much as possible. Also the class probabilities are written to the output file for debugging purposes.

```
// returns true if name exists
inline bool file_exists(const std::string &name)
{
  struct stat buffer;
  return (stat(name.c_str(), &buffer) == 0);
}

// function that writes an AE (and some debugging info) to filename
// if filename does not yet exist
void write_if_not_exist(const std::string &filename, double fuzzed[],
                        int expected, int actual,
                        int num, double probs[])
{
  std::string filebase = ".ADVERSARIAL_EXAMPLES/";
  std::string full_filename = filebase.append(filename);
  if (file_exists(full_filename))
  {
    return;
  }

  ofstream myfile;
  myfile.open(full_filename);
  for (int i = 0; i < 10; i++)
  {
    myfile << std::setprecision(15) << fuzzed[i] << ",";
  }
  myfile << to_string(expected) << "," << to_string(actual)
  << "," << to_string(num);

  for (int i = 0; i < 2; i++)
  {
    myfile << "," << std::setprecision(15) << probs[i];
  }
  myfile.close();
}
```

Listing D.2: Adversarial Example handling

D.3. Predict

Listing D.3 shows the code required to resemble a prediction from a Gradient Boosting model. It shows the binary classification case, for multi-class classification tree_n is represented by multiple one-vs-all sub-trees which are combined in tree_n such that num_classes values can be returned. For binary classification the probabilities are calculated using the log-likelihood, for multi-class classification a softmax function is used. For brevity, only a part of tree_0 is shown while for the other trees only the signature is shown. The return value for each leave is either a class probability (Random Forest) or a regression values (Gradient Boosting, combined into a list of num_classes regression values for multi-class classification).

```
// predicts a class from raw probabilities
int predict(double probs[])
{
  double greatest_sum = probs[0];
  int greatest_class = 0;
  for (int i = 0; i < 2; i++)
  {
    if (probs[i] > greatest_sum)
    {
      greatest_sum = probs[i];
      greatest_class = i;
    }
  }
  return greatest_class;
}

// return np.exp(val)/(1+np.exp(val))
double log_likelihood(double v)
{
  double e = exp(v);
  return e / (1 + e);
}

// s = sum([np.exp(v) for v in vals])
// return [np.exp(v)/s for v in vals]
double *softmax(double *decisions)
{

  static double res[{{nc}}];
  double s = 0.0;
  for (int i = 0; i < {{nc}}; i++)
  {
    s = s + exp(decisions[i]);
  }
  for (int i = 0; i < {{nc}}; i++)
  {
    res[i] = exp(decisions[i]) / s;
  }
  return res;
}

// returns class probabilities from raw predictions
double *calc_class_prob(double **arr, int num_trees)
{
  double decisions[1] = {-0.5718681934644387}; // the initial
     prediction

  for (int i = 0; i < num_trees; i++)
  {
    auto counts = arr[i];
    for (int ii = 0; ii < 1; ii++)
    {
      decisions[ii] = decisions[ii] + (0.1 * counts[ii]);
    }
  }

  // For multiclass classification, a softmax function is used
  double l = log_likelihood(decisions[0]);
  static double sums[2];
  sums[0] = 1.0 - l;
  sums[1] = l;
  return sums;
}
```

```
// a part of tree_0 (for brevity)
double *tree_0(double fs[])
{
  // res has size num_classes. In the case of binary
  // classification, one spot is redundant
  static double res[2];
  if (fs[2] <= 0.1666667)
  {
    if (fs[6] <= 0.5)
    {
      if (fs[1] <= 0.5)
      {
        if (fs[6] <= 0.3888889)
        {
          if (fs[0] <= 0.0848987)
          {
            res[0] = -1.5644699140401146;
            return res;
          }
          else
          {
            if (fs[0] <= 0.0862544)
            {
              res[0] = -1.5644699140401148;
              return res;
            }
            else
            {
              res[0] = -1.5644699140401142;
              return res;
            }
          }
        }
        ...
      }
    }
  }
}

double *tree_1(double fs[]) { ... }
double *tree_2(double fs[]) { ... }
double *tree_3(double fs[]) { ... }

// predict class probabilities for fuzzed_features
double *predict_probs(double fuzzed_features[])
{
  double **tree_outputs = 0;
  tree_outputs = new double *[4];
  tree_outputs[0] = tree_0(fuzzed_features);
  tree_outputs[1] = tree_1(fuzzed_features);
  tree_outputs[2] = tree_2(fuzzed_features);
  tree_outputs[3] = tree_3(fuzzed_features);
  auto probs = calc_class_prob(tree_outputs, 4);
  delete[] tree_outputs;
  return probs;
}
```

Listing D.3: Predict outcome

D.3.1. Random Forest

When fuzzing Random Forest models, the calc_class_prob function is different as shown in Listing D.4.

```
double* calc_class_prob(double** arr, int num_trees) {
        // Insert all elements in hash.
        static double sums[{{ num_classes }}];
        for (int i = 0; i < {{ num_classes }}; i++) {
                sums[i] = 0.0;
        }

        for (int i = 0; i < num_trees; i++) {
                auto counts = arr[i];
                for (int ii = 0; ii < {{ num_classes }}; ii++) {
                        sums[ii] = sums[ii] + (counts[ii] / (double)
                            num_trees);
                }
        }
        return sums;
}
```

Listing D.4: Random Forest predict

D.4. Mutation

Listing D.5 shows the custom-mutator (LLVMFuzzerCustomMutator) and crossover (LLVMFuzzerCustom-CrossOver) functions for libFuzzer. There are two helper functions that mutate the actual features: mutate (performs normal Gaussian mutation) and mutate_towards_original (only mutates towards the victim, typically only used when the fuzzed features already resemble an adversarial example). The find_closest_threshold function is a function that moves a mutated value towards a better position in the current bounding box as described in chapter 5. The custom-mutator function first unpacks the input data into doubles. It then either mutates all features with the biggest distances from the victim towards the victim, or it mutates single features with a certain chance. This chance can be influenced by multiple settings. The basic chance is defined by the prior_mutate_chances variable as inserted by FATE. If feature importances are used to guide mutation, the mutation chance is biased by adding the feature importance (scaled such that the sum of all feature importances is 1) to the basic chance. Likewise, the chance to mutate a feature can be influenced by how large the difference is between the fuzzed and original features. The basic chance is then divided by the amount of sub-chances it consists of which results in a probability $0.0 <= p_f <= 1.0$ to mutate feature f. The feature is mutated with chance 0.5 towards the victim if AE_MUTATE_TOWARDS_VICTIM is true or otherwise scaled by both epsilon and biggest_dist when MUTATE_LESS_WHEN_CLOSER is true or just within epsilon range otherwise. For honggfuzz, fuzzing with custom mutators is not supported by FATE, so the CustomMutator and CustomCrossover functions and their helper functions are excluded in that case.

```
// crossover function combines Data1 and Data2 on a random index
extern "C" size_t LLVMFuzzerCustomCrossOver(const uint8_t *Data1,
    size_t Size1,
                                            const uint8_t *Data2,
                                                size_t Size2,
                                            uint8_t *Out, size_t
                                                MaxOutSize,
                                            unsigned int Seed)
{
  en.seed(Seed);
  int split_index = crossover_dist(en);
  int split_length = split_index * sizeof(double);
  memcpy(Out, Data1, split_length);
  memcpy(Out + split_length, Data2 + split_length, Size2 - split_length
     );
  return Size2;
}

// finds first threshold between the mutated and original value,
// looking from the mutated value
double find_closest_threshold(double mutated, int fi)
{
  vector<double> thresholds = thresholds_per_feature[fi];
  double original = original_features[fi];
  if (mutated < original)
  {
    for (double threshold : thresholds)
    {
      if (threshold > original)
      {
        break;
      }
      if (mutated < threshold - min_e && threshold - min_e < original)
      {
        return threshold - min_e;
      }
    }
    return original;
  }
  else if (mutated > original)
  {
    for (auto thres = thresholds.rbegin();
        thres != thresholds.rend(); ++thres)
    {
      if (*thres < original)
      {
        break;
      }
      if (original < *thres + min_e && *thres + min_e < mutated)
      {
        return *thres + min_e;
      }
    }
    return original;
  }
  else
  {
    return original;
  }
}

// mutates original scaled by biggest_dist
double mutate(double original, int index, double biggest_dist, bool
    is_ae)
{
  double md = mutation_dist(en);

  // if STEEP_CURVE
  double m = pow(md, 2.0);
  if (md < 0)
  {
    m = -m;
  }
  // else
  double m = md;

  // mutating 1.0 up has no effect
  if (original > 0.9999)
  {
    m = -abs(m);
  }
  // mutating 0.0 down has no effect
  if (original < 0.0001)
  {
    m = abs(m);
  }

  double mutated = original + m * biggest_dist * 0.1; // 0.1=epsilon
  mutated = min((double)1.0, mutated);
  mutated = max((double)0.0, mutated);

  if (is_ae)
  {
    double nearest_threshold = find_closest_threshold(mutated, index);
    return nearest_threshold;
  }
  return mutated;
}

// similar to mutate, but only allows mutations towards the victim
    feature
// if the current features are an AE
double mutate_towards_original(double f_mutated, double f_original,
                               int index, bool is_ae)
{
  // 2 * difference because it may be the case
  // that we have to "overshoot" the original
  double perturbation = abs(mutation_dist(en)) * 0.1 * 2
                      * abs(f_mutated - f_original);
  double mutated;
  if (f_original > f_mutated)
  {
    mutated = f_mutated + perturbation;
  }
  else
  {
    mutated = f_mutated - perturbation;
  }
  mutated = min((double)1.0, mutated);
  mutated = max((double)0.0, mutated);

  if (is_ae)
  {
    double nearest_threshold = find_closest_threshold(mutated, index);
    return nearest_threshold;
  }
  return mutated;
}

// Calculates the sum of differences between the fuzzed and original
   features
double sum_distance(double fuzzed[])
{
  double summed = 0.0;
  for (int i = 0; i < 10; i++)
  {
    double f_original = original_features[i];
    double f_fuzzed = fuzzed[i];
    double diff = abs(f_original - f_fuzzed);
    summed += diff;
  }
  return summed;
}

// l_inf norm distance function
double distance_linf(double fuzzed[])
{
  double max_diff = 0.0;
  for (int i = 0; i < 10; i++)
  {
    double f_original = original_features[i];
    double f_fuzzed = fuzzed[i];
    double diff = abs(f_original - f_fuzzed);
    if (diff > max_diff)
    {
      max_diff = diff;
    }
  }
  return max_diff;
}

// mutates all fuzzed features that have difference > (biggest_diff -
   0.01)
// with the original features
double *mutate_biggest_diff_features_l_inf(double fs[], double
   biggest_diff)
{
  for (int i = 0; i < 10; i++)
  {
    double f_original = original_features[i];
    double f_mutated = fs[i];
    double diff = abs(f_original - f_mutated);
    if (diff > (biggest_diff - 0.01))
    {
      // In this function, it is always an AE to is_ae is set to true
      double better = mutate_towards_original(f_mutated, f_original, i,
          true);
      fs[i] = better;
    }
  }
  return fs;
}

// function recognized by libFuzzer as the custom mutator, mutates *
   Data
extern "C" size_t LLVMFuzzerCustomMutator(uint8_t *Data, size_t Size,
                                          size_t MaxSize, unsigned int
                                             Seed)
{
  en.seed(Seed);

  // Sanity check
  if (Size != MaxSize)
  {
    // the first run is always without input,
    // so make sure we do not crash on no input
    if (Size != 0)
    {
      printf("size␣%ld\n", Size);
      abort(); // sanity check
    }
    return Size;
  }

  // Unpack data to doubles
  double fs[10];
  for (int i = 0; i < 10; i++)
  {
    fs[i] = *(double *)(Data + i * sizeof(double));
  }

  bool is_ae = false; // default

  // the following lines are only inserted when necessary
  auto probs = predict_probs(fs);
  int fuzzed_class = predict(probs);
  is_ae = fuzzed_class != original_class;

  double biggest_dist = 1.0; // default

  // only use biggest_dist for scaling when necessary
  if (is_ae)
  {
    biggest_dist = distance_linf(fs);
  }

  // total difference
  double sum_of_dist = sum_distance(fs);
  double mutate_biggest_chance = 0.5; // setting in constants.py
  if (is_ae && biggest_dist > 0.001 && uniform_dist(en) <
     mutate_biggest_chance)
  {
    // with mutate_biggest_chance, we mutate all biggest differences
    mutate_biggest_diff_features_l_inf(fs, biggest_dist);
    for (int i = 0; i < 10; i++)
    {
      ((double *)(Data + i * sizeof(double)))[0] = fs[i];
    }
  }
  else
  {
    for (int i = 0; i < 10; i++)
    {
      // We treat each feature separately
      double chance = prior_mutate_chances[i];
      double chance_count = 1.0;

      // If USE_FEATURE_IMPORTANCES
      chance_count++;
      chance += feature_importances[i];

      // If BIAS_MUTATE_BIG_DIFFS
      if (is_ae && biggest_dist > 0.00001)
      {
        chance_count++;
        double diff = abs(fs[i] - original_features[i]);
        chance += diff / sum_of_dist;
      }
      // recalculate mutation chance based on biases introduced above
      chance = chance / chance_count;
      if (uniform_dist(en) < chance)
      {
        double mutated_val;

        // if MUTATE_LESS_WHEN_CLOSER
        mutated_val = mutate(fs[i], i, biggest_dist, is_ae);
        // elif AE_MUTATE_TOWARDS_VICTIM
        if (is_ae)
        {
          mutated_val = mutate_towards_original(fs[i],
             original_features[i],
                                                i, is_ae);
        }
        else
        {
          mutated_val = mutate(fs[i], i, 1, is_ae);
        }
        // else
        mutated_val = mutate(fs[i], i, 1, is_ae);
        // update Data with mutated feature i
        ((double *)(Data + i * sizeof(double)))[0] = mutated_val;
      }
    }
  }
  return Size;
}
```

Listing D.5: Mutation

D.4.1. AFL++ mutation library

When fuzzing using AFL++ with the custom mutator, the custom mutator is compiled into a separate mutation library which does not support crossover functionality. Because we now do not have access to the predict functions, we mimic the is_ae check of LLVMFuzzerCustomMutator: with probability 0.5, the input is treated as an adversarial example. Apart from the code shown in Listing D.6, the code from Listing D.5 is also included (with the `LLVMFuzzerCustomMutator` signature changed to:

```
extern "C"  size_t afl_custom_fuzz(void *mutator_instance, uint8_t *
    Data, size_t Size, uint8_t **outData, uint8_t *additional_buf,
    size_t add_buf_size, size_t MaxSize)
```

). Because the command-line arguments to not reach the AFL++ mutation library, information about the victim is communicated through environment variables.

```
std::string getEnvVar( std::string const & key ) {
    char * val = getenv( key.c_str() );
    return val == NULL ? std::string("") : std::string(val);
}

template <typename Out>
void split(const std::string &s, char delim, Out result) {
    std::istringstream iss(s);
    std::string item;
    while (std::getline(iss, item, delim)) {
        *result++ = item;
    }
}

std::vector<std::string> split(const std::string &s, char delim) {
    std::vector<std::string> elems;
    split(s, delim, std::back_inserter(elems));
    return elems;
}

extern "C" void *afl_custom_init(void *param, unsigned int seed) {
    en.seed(seed);

    // Thresholds are initialize here if USE_THRESHOLDS
    thresholds_per_feature.push_back(th_f_0);
    thresholds_per_feature.push_back(th_f_1);
    thresholds_per_feature.push_back(th_f_2);
    thresholds_per_feature.push_back(th_f_3);
    thresholds_per_feature.push_back(th_f_4);
    thresholds_per_feature.push_back(th_f_5);
    thresholds_per_feature.push_back(th_f_6);
    thresholds_per_feature.push_back(th_f_7);
    thresholds_per_feature.push_back(th_f_8);
    thresholds_per_feature.push_back(th_f_9);

    check_num = stoi(getEnvVar("CHECK_NUM"));
    int num_features = stoi(getEnvVar("NUM_FEATURES"));
    original_class = stoi(getEnvVar("ORIGINAL_CLASS"));
    auto of = getEnvVar("ORIGINAL_FEATURES");
    auto splitted_of = split(of, '␣');
    vector<double> original(num_features, 0.0);

    for (int i = 0; i < num_features; i++) {
        original[i] = stod(splitted_of[i]);
    }
    original_features = original;
    return calloc(1, sizeof(double)); // Initialize some dummy memory
}

extern "C" void afl_custom_deinit(void *data) {}
```

Listing D.6: AFL mutation library

D.5. Objective function

The objective function is shown in Listing D.7, with only a subset of the distance- and probability steps for brevity. To make sure the individual branches are not optimized away by the compiler, the ddd variable is set and printed in a part of the code that will never be reached (although the compiler cannot know that). Furthermore, a file is written in the last branch, again to make sure the code will not be optimized. We do the same for the probability branches with the ppp variable and a file write.

```
// the objective function
void check_for_ae(double fuzzed_features[], int fuzzed_class, double
    probs[])
{
  int num_features = 10;
  double proba_original = probs[original_class];
  // For comparing the highest two probabilities in the probability
     descent
  std::sort(probs, probs+2, std::greater<double>{});
  std::string check_base = "check_";
  std::string file_base = check_base.append(to_string(check_num)).
     append("-");
  double distance = distance_linf(fuzzed_features);
  if (fuzzed_class != original_class){
    cur_ae_dist = distance;
    } else {
    cur_ae_dist = 1.1;
  }
  if (fuzzed_class != original_class)
  {
    if (distance <= 1.0)
    {
      ddd = 0;
      if (distance < 0.995)
      {
        ddd = 1;
        if (distance < 0.99)
        {
          ddd = 2;
          if (distance < 0.985)
          {
            ...
            ddd = 5;
            if (distance < 0.01)
            {
              ddd = 6;
              if (distance < 0.005)
              {
                ddd = 7;
                if (distance < 1e-06)
                {
```

```
                write_if_not_exist(file_base.append("1e-06.txt"),
                   fuzzed_features, original_class, fuzzed_class,
                   check_num, probs);
                }
              }
            }
          }
        }
      }
    }
    if (distance < best_ae_dist)
    {
      best_ae_dist = distance;
      double new_dist = (double) round(distance*100000.0) / 100000.0;
      std::stringstream ss;
      ss << std::fixed << std::setprecision(5) << new_dist;
      std::string new_dist_str = ss.str();
      write_if_not_exist(file_base.append(new_dist_str).append(".txt"),
          fuzzed_features, original_class, fuzzed_class, check_num,
         probs);
    }
  }
  else
  {
    if (probs[0] < probs[1]) {abort();} // sanity check, should not
       happen
    if (distance < 0.2)
    {
      if (probs[0] - probs[1] < 0.2)
      {
        ppp = 0;
        if (probs[0] - probs[1] < 0.19)
        {
          ppp = 1;
          ...
          if (probs[0] - probs[1] < 0.01)
          {
            ppp = 2;
            write_if_not_exist(file_base.append("probasmall.txt"),
                                                    fuzzed_features,
               original_class,
                               fuzzed_class, check_num, probs);
          }
        }
      }

    }
  }
}
```

Listing D.7: Objective function

D.6. Fuzz function

The fuzz function is the function that is repeatedly called by the fuzzer. For AFL++ the signature is slightly different than for libFuzzer but it is similar in nature, while honggfuzz uses the same function signature as libFuzzer for this function. The function first checks if we are dealing with input that has the correct length. If not, we simply return right away. When the custom mutator option is used, we also add a sanity check: if the size of the input is not the expected size or 0 (libFuzzer always first executes the target with an empty seed), we add a crash (abort()). The fuzzer will quit when this happens so that we know something went wrong in the mutator, that is responsible for keeping the length of the input equal between runs. The fuzz function furthermore produces the prediction of the Tree Ensemble for the current input and calls the objective function that checks for Adversarial Examples.

```
  // the function that actually fuzzes an input
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)
{
  if (Size != 80)
  {

    if (Size != 0)
    {
      printf("size␣%ld\n", Size);
      abort(); // this line is a sanity check; it is removed
      // when the custom mutator is not used
    }
  }
  double fuzzed_features[10];
  int count = 0;
  while (count < Size / sizeof(double))
  {
    fuzzed_features[count] = *(double *)(Data + count * sizeof(double))
        ;
    count++;
  }

  auto probs = predict_probs(fuzzed_features);
  int fuzzed_class = predict(probs);

  check_for_ae(fuzzed_features, fuzzed_class, probs);

  // The following never happens, but is used such that the structure
  // of the objective function and the variables ppp and ddd will not
  // be compiled away or optimized
  if ((double) fuzzed_class - probs[0] == -10000.42)
  {
    printf("never␣happens␣%d\n", ppp);
    printf("never␣happens␣2␣%d\n", ddd);
  }
  return 0;
}
```

Listing D.8: Fuzz function

D.6.1. AFL++ persistent mode

AFL++ features a "persistent mode" that can drastically improve performance by supplying mutated input through in-memory buffers and by only initializing the target once every 1000 runs. Listing D.9 shows how the persistent mode is implemented in FATE. For AFL++ the information about the victim is passed as environment variables instead of command-line arguments, as this is needed for the AFL++ mutator anyway (see subsection D.4.1). ORIGINAL_FEATURES is a string with the original features separated by a space. After __AFL_LOOP is executed 1000 times, the target is fully re-initialized to e.g. combat memory leaks. The code in the loop is similar to the code from the libFuzzer/honggfuzz LLVMFuzzerTestOneInput function.

```
int main() {
    check_num = stoi(getEnvVar("CHECK_NUM"));
    int num_features = stoi(getEnvVar("NUM_FEATURES"));
    original_class = stoi(getEnvVar("ORIGINAL_CLASS"));
    auto of = getEnvVar("ORIGINAL_FEATURES");
    auto splitted_of = split(of, '␣');
    vector<double> original(num_features, 0.0);

    for (int i = 0; i < num_features; i++) {
        original[i] = stod(splitted_of[i]);
    }
    original_features = original;
#ifdef __AFL_HAVE_MANUAL_CONTROL
    __AFL_INIT(); // This is where AFL++ initializes
    // after __AFL_LOOP iterations
#endif
    unsigned char *Data = __AFL_FUZZ_TESTCASE_BUF;
    while (__AFL_LOOP(1000)) {
        int Size = __AFL_FUZZ_TESTCASE_LEN;

        if (Size != 80) {
            {% if not use_custom_mutator %}
            continue;
            {% else %}
            if (Size != 0) {
                printf("size␣%d\n", Size);
                printf("digit␣%d\n", ddd);
                abort();
            }
            {% endif %}
        }

        double fuzzed_features[10];
        int count = 0;
        while (count < Size / sizeof(double)) {
            fuzzed_features[count] = *(double *)(Data + count*sizeof(
                double));
            count++;
        }

        auto probs = predict_probs(fuzzed_features);
        int fuzzed_class = predict(probs);
        check_for_ae(fuzzed_features, fuzzed_class, probs);

        // The following never happens, but is used such that the
            structure of the objective function and the variables ppp
            and ddd will not be compiled away or optimized
        if ((double) fuzzed_class - probs[0] == -10000.42) {
          printf("never␣happens␣%d\n", ppp);
          printf("never␣happens␣2␣%d\n", ddd);
        }
    }
    return 0;
}
```

Listing D.9: AFL++ fuzz function and persistent mode

D.7. Standalone

Listing D.10 shows the standalone Genetic Algorithm. It calls the `LLVMFuzzerInitialize`, `LLVMFuzzerCustomCrossOver`, `LLVMFuzzerCustomMutator` and `LLVMFuzzerTestOneInput` functions to make sure it performs the same as when executed in libFuzzer. The population is implemented as a 2d-array of bytes, with the adversarial distances of the population implemented as a separate array of doubles. The population is initialized with the original features of the victim for each population member. The function returns after runtime_seconds or max_runs, whichever comes first. Variables such as `mutation_depth` and `population_size` that are not initialized represent settings of the Genetic Algorithm.

```
int main(int argc, char **argv) {
    std::uniform_int_distribution<int> num_mut(1, mutation_depth); //
        uniform, unbiased
    std::uniform_int_distribution<int> pop_dist(0, population_size-1);
        // uniform, unbiased

    // reads information about the victim
    LLVMFuzzerInitialize(&argc, &argv);

    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    int Size = num_features*sizeof(double);
    uint8_t OrigData[Size];
    // victim features to byte array
    for (int i = 0; i < num_features; i++) {
        ((double *)(OrigData + i*sizeof(double)))[0] =
            original_features[i];
    }

    // Start the timer
    auto t1 = high_resolution_clock::now();

    // initial population consists of the victim features
    uint8_t** population = 0;
    population = new uint8_t*[population_size];
    for (int i = 0; i < population_size; i++) {
        uint8_t Entry[Size];
        memcpy(Entry, OrigData, Size);
        population[i] = Entry;
    }

    // we keep the distances of the population members in a separate
        array
    double population_dist[population_size] = {1.0};
    std::fill_n(population_dist, population_size, 1.0);

    int iter = 0;
    while (true) {
        /* exit if the runtime or max_runs is exceeded */
        auto t2 = high_resolution_clock::now();
        auto ms_int = duration_cast<milliseconds>(t2 - t1);
        if (ms_int > std::chrono::milliseconds(runtime_seconds*1000)) {
            break;
        }
        iter++;
        if (iter > max_runs) {
            break;
        }

        uint8_t Data[Size];
        memcpy(Data, population[pop_dist(en)], Size);

        // we call the mutator maximal mutation_depth times
        for (int nm = 0; nm < num_mut(en); nm++) {
            if (uniform_dist(en) < 0.001) {
                // unlikely
                uint8_t CrossoverData[Size];
                if (uniform_dist(en) < random_crossover_chance) {
                    // crossover with random features
                    for (int j = 0; j < num_features; j++) {
                        ((double *)(CrossoverData + j*sizeof(double)))
                            [0] = uniform_dist(en);
                    }
                } else {
                    // crossover with another member of the population
                    memcpy(CrossoverData, population[pop_dist(en)],
                        Size);
                }
                uint8_t indata[Size];
                memcpy(indata, Data, Size);

                // perform crossover
                LLVMFuzzerCustomCrossOver(indata, Size, CrossoverData,
                    Size, Data, Size, en());
            } else {
                // perform mutation
                LLVMFuzzerCustomMutator(Data, Size, Size, en());
            }
        }
        LLVMFuzzerTestOneInput(Data, Size); // call the objective
            function
        // itr == -1 if current input is not better than one of the
            members of the population
        int itr = index_to_replace(population_dist, cur_ae_dist);
        if (itr != -1) {
            // Save input to population
            memcpy(population[itr], Data, Size);
        }
    }
    auto t3 = high_resolution_clock::now();

    /* Getting number of milliseconds as an integer. */
    auto ms_int = duration_cast<milliseconds>(t3 - t1);

    /* Output runtime information to be parsed by FATE */
    std::cout << "Performed␣" << iter << "␣runs␣in␣" << (double)ms_int.
        count()/1000 << "␣seconds.\n";

    return 0;
}
```

```c
// finds the member of the population with the highest adversarial
   distance
// if higher than the current input.
// cur_dist = 1.1 if the current input is not an adversarial example.
// the population distances are initialized with the value 1.0 (for the
// original features of the victim)
int index_to_replace(double distances[], double cur_dist) {
    int res = -1;
    double largest_dist = 0.0;
    for (int i = 0; i < 1; i++) {
        if (distances[i] > cur_dist && distances[i] > largest_dist) {
            res = i;
            largest_dist = distances[i];
        }
    }
    return res;
}
```

Listing D.10: Standalone Genetic Algorithm function