Universiteit
Leiden
The Netherlands

Bachelor Computer Science

Privacy-preserving Explainable AI in the Active Security Model

Lars Selles

Supervisors:
Dr.ir. E. Makri & Sun Shuang

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 1, 2024

Abstract

In this thesis project, a newly created algorithm will be presented. This algorithm is based on the earlier created XorSHAP algorithm [JV23]. The XorSHAP algorithm is a privacy-preserving explainable AI model, which is secure in the passive security model. As the result of this thesis project, the XorSHAP algorithm is transferred and adapted into the compatibility of the active security protocol. In addition to this, another way of calculating the SHAP values is presented.
The SHAP values can be calculated with an approximation, using the Monte Carlo algorithm [LS06, SK14]. The values can also be calculated using an algorithm, which is directly based on the structure of the formula of the SHAP value.
As a result of the thesis, it can be seen that the approximation of the Monte Carlo algorithm is less computational expensive than the direct equation of these values.
It is also stated that the addition of the active security model has an impact on the memory usage and execution time of the model, once the depth of the decision tree is altered. The active security protocol requires significant more memory and execution time compared to the passive security protocol.

Contents

1 Introduction

Currently, there are a lot of developments regarding the usage of machine learning and artificial intelligence models. These models are being trained to behave like human intelligence, performed by machines [Abb21]. Due to developments, these models are becoming more and more intelligent and difficult to understand and predict [TBS+21]. This is likely to result in models which are difficult to predict and control. Especially in applications for fields like healthcare and finance, a model needs to be reliable and cannot show any signs of racism or generalisation. This is important for prevention of certain problems, for example a problem found in the year 2019. It became clear that many black American citizens were discriminated by a biased AI model used in many American hospitals. This resulted in worse healthcare for these people and could have immense consequences for their well-being [Led19].

Because of such problems, it is important to improve the trustworthiness of AI. This can be done by controlling how the models learn with potentially biased data and interpret the decisions made. Developing a transparent AI system and increasing the trust of the user are also crucial [GWS20]. The trustworthiness of AI can be improved by making use of different and new types of models or adding features to the currently available models. This can be done by a relatively new concept, explainable AI (XAI). The goal of XAI is to make the computation of the AI model explainable, by allowing the user to trace back the steps of the AI. With the ability to retrace the steps of AI, more information about the input can be retrieved. This constitutes a breach of privacy, and highlights why supporting *privacy-preserving* XAI is important [JV23]. Especially, for fields like healthcare, finance and criminality predictions. In these fields, the privacy-preserving is needed to be able the use the algorithms and still respect the privacy of the inputting clients.

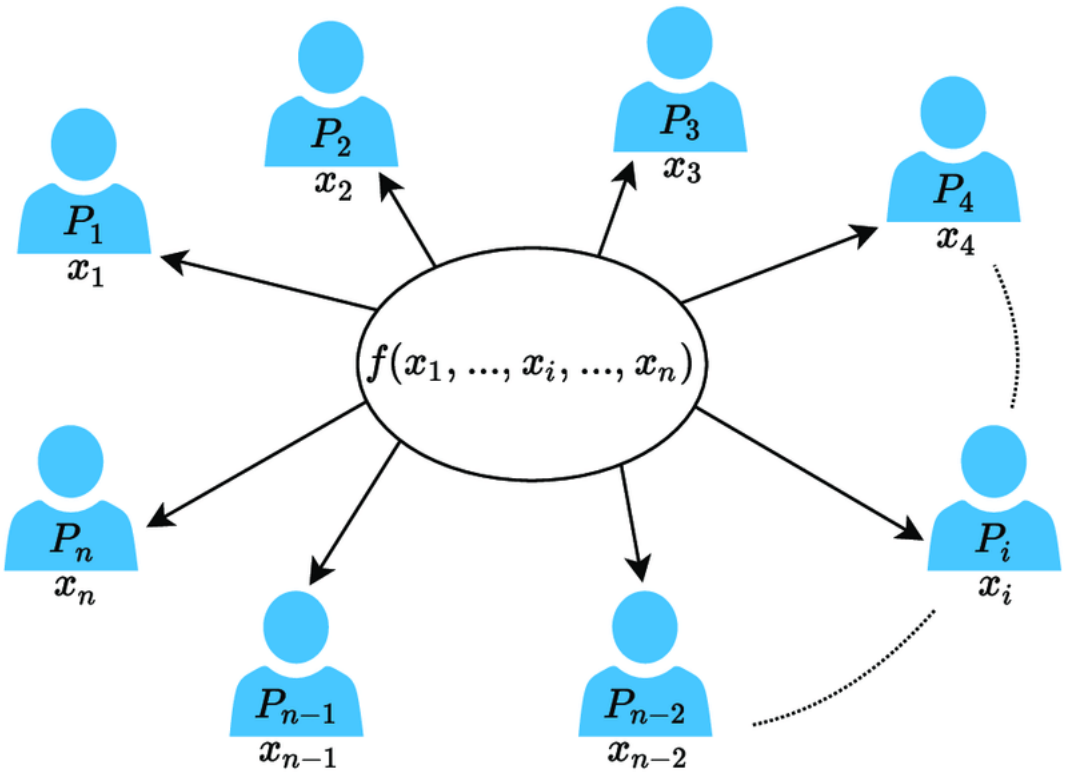

Figure 1: Overview of the secure multiparty computation technique. In this figure, the parties P are numbered 1,2,3,...,n. These clients are computing a joint function f, without revealing their inputting values (x) to the other clients. [Boz21]

A method to preserve the privacy in explainable AI, is by using a technique called Secure Multi-Party Computation (MPC) [Mak21]. The problem solved by this technique is the ability to perform a computation on sensitive data without revealing this data to other parties and without jeopardizing the correctness of the computation. In Figure 1, an overview of the usage and way of computing from the MPC is given. It can be seen that only the result of the joint function will be distributed without showing any secret details or information to the other parties in the computation.

The combination of these techniques shows the possibility to create an algorithm which is able to behave as an AI model, but is explainable and preserves the privacy of the input parties. This has already been done for simple computations and decision-making in decision tree models, the XorSHAP algorithm [JV23].

However, the XorSHAP model can still be improved. As it is stated, the algorithm only supports the passive security protocol. In this research, the goal is to achieve a more extensive version of the XorSHAP algorithm, with the support of more stringent active security. Accordingly, the main research question of this research is: *How can we extend the Privacy-Preserving Explainable AI for Decision Tree Models to support active security?*

1.1 Thesis overview

In the following parts of this thesis, a research about privacy-preserving explainable AI with the support of active security will be set up. First, more information about the preliminary knowledge of these technologies will be given. Afterwards, the methodology for the creation and testing of the algorithms will be shown and elaborated with the view on the experimental setup. Furthermore, the results will be presented and discussed and concluded.
This research is completed as a "bachelor thesis" for the Leiden Institute of Advanced Computer Science, with the support of supervisor Eleftheria Makri.

2 Preliminaries

2.1 Secure multiparty computation

Secure Multi-Party Computation (MPC) is a protocol where multiple parties will calculate the output of a joint function where just the output is revealed, without revealing the input [Lin20]. Within the usage of this protocol the privacy of the input parties can be ensured, because the input data will not be revealed during the calculation of the joint function.
In most of the cases, once the use of MPC protocols are implemented in the creation of new algorithms or software, this application is developed inside the world of a framework with the ability to support MPC. One of these frameworks is the MP-SPDZ framework.

2.2 MP-SPDZ

With the usage of the MP-SPDZ framework, the goal to apply MPC in the program can be achieved [Kel20].
This framework has multiple possibilities and also the possibility to make use of a decision tree model. This model includes a classifier and a trainer. With the classifier, the interaction between the user (the program/algorithm) and the model and its structure can be made. The trainer is used to build the structure of the model and train it with the use of training data.
MP-SPDZ implements MPC by making use of secret-shared values. Each data point of input between different parties, for example the training data for the decision tree model, is converted to a secret data structure. This structure is being used in all different functions and classes of the framework. In this way the secrecy and privacy-preserving of the inputting parties is secured.

2.3 Decision trees

In the field of artificial intelligence and machine learning, there are many different representations to achieve the goal of model. One of these models is the decision tree model.

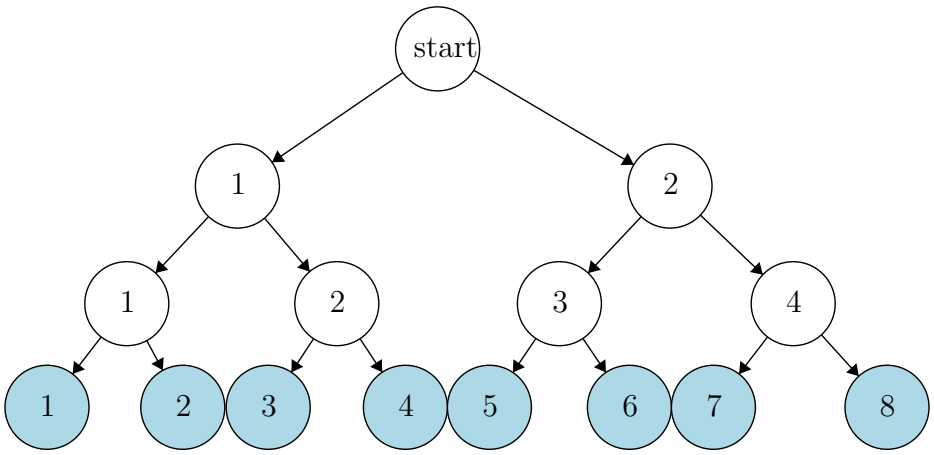

Figure 2: This figure shows a graphical representation of a binary decision tree model. In the bottom level, the nodes are light-blue to represent the leaves in the model.

In Figure 2, a graphical representation of a binary decision tree model can be seen. The model consists of a starting node, this is the starting point of every prediction in the model. Each node represents a decision that needs to be made. When continuing the prediction in the model from the start node in level 0, the decision to continue to node 1 or node 2 in level 1 is determined by an attribute ID in the start node and a threshold value. The attribute ID represents the attribute (feature) in the trained and input data, the value of this feature in the prediction data is compared to the threshold value and when the feature value is lower than the threshold value, the continuation of the prediction will continue to node 1 in level 1, and vice versa once the value of the feature is higher than the threshold. This concludes of a node structure as described in 1.

$$\begin{aligned} node &= level, NID, AID, threshold \\ depth &= tree.height \\ level &= [0, ..., depth - 2] \\ NID &= [1, ..., 2^{level}] \\ AID &= [0, ..., \#features] \\ threshold &= \mathbb{R} \end{aligned} \tag{1}$$

This Equation 1 represents the structure of each node in the decision tree model, where *level* is an integer value representing the level where the node is located. The *depth* is the number of levels in the tree. The NID is the identification number of the node for that current level, these numbers are visible in the nodes in Figure 2. AID represents the index of the feature in the sample for which the model is predicting, so it is important to note that the order of features for the training data and sample data has to be equal as the AID is an index value based on this order. $Threshold$ is the value which is used for comparison between the *threshold* value and the value of the feature from the sample data $SampleData[AID]$, which has the value of the AID^{th} value of the sample. Furthermore, in Figure 2, light-blue nodes can be seen, these nodes represent the leaves of the

decision tree. These leaves are built from the structure described in Equation 2.

$$\begin{aligned} node &= level, NID, result \\ depth &= tree.height \\ level &= [0, ..., depth - 2] \\ NID &= [1, ..., 2^{level}] \\ result &= 0, 1 \end{aligned} \tag{2}$$

In addition to the nodes, the leaves have another value, the *result*. This is the value of the result of the prediction once the prediction has come to this leaf. For the tree model used in the MP-SPDZ, the *result* can only be 0 or 1, so the prediction determines if the result of the prediction is either 'yes' or 'no'.
These representations of the decision tree are based on the decision tree model used in the MP-SPDZ model.

2.4 SHAP values

In order to make an artificial intelligence model explainable, a new technique is required. Therefore, the SHAP and LIME values are common ways to achieve this goal. The Local Interpretable Model-agnostic Explanations, LIME, is developed by Marco Ribeiro in 2016 [RSG16]. LIME approximates the complex model locally with a simpler model and uses this approximation to understand and explain the predictions of the complex model. This is done by comparing results from samples in the dataset and slightly adapted samples to interpret the importance of a feature.
The SHapley Additive exPlanations values, SHAP, was first introduced in the cooperative game theory [Sha53]. After this introduction, the SHAP values were more used for the explainability in AI models and machine learning models [Mas21]. The SHAP value is obtained by iterating over all subsets of features and comparing the results of the prediction.
These values represent the importance of a feature in the trained decision tree model. In this case, we will make use of the SHAP values, because the LIME values are mainly optimised for usage in regression models. For that reason, the SHAP values will perform better in the decision tree model, as the SHAP values will perform well in every model. Furthermore, the LIME method is more based on models that cannot be run in a local only environment.
The main idea behind the computation for SHAP values can be seen as follows; model x predicts the likelihood of being a criminal for person p. If p has grown up in a rich neighbourhood, the features in the data from person p about being raised in a poorer neighbourhood do not affect the prediction of p. As the type of neighbourhood is part of the input data for x. This says the features about being raised in a poorer neighbourhood would be less important than a feature about being raised in a rich neighbourhood. In this case, the SHAP value of features about being raised in a poorer neighbourhood are smaller than the feature for being raised in a rich neighbourhood. In this way, the SHAP value(s) show the importance of a feature in a specific situation and explain how the AI has come to the answer. This is one of the multiple techniques to apply explainability in AI models [MPC23]. When we try to calculate this SHAP value, the following calculation can be used.

$$\phi_i = \sum_{S \subseteq F, \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} \left[f_S(x_{S \cup \{i\}}) - f_S(x_S) \right] \tag{3}$$

In Equation 3, ϕ_i represents the SHAP value. The right part of the formula is computed inside the sum of every possible subset of features in a sample with the deletion of feature i, so the SHAP value is the sum of the computed value of every possible subset of features. The right part is separated into the weight calculation and the difference calculation. In $\frac{|S|!(n-|S|-1)!}{n!}$, the weight is calculated. S represents the subset of features and $|S|$ the length of this subset, n represents the total length of the features in the sample. In the difference calculation part, $\left[f_S(x_{S\cup\{i\}}) - f_S(x_S)\right]$, the difference between the value of the prediction with and without feature i is calculated. Here f_S represents the model and X the sample with features S.
The goal of the weight is to let computations with a large S have a greater part in the total SHAP value. For example, 20 features are in S, if the addition of feature i makes a big difference in the outcome of the prediction, it can be stated that feature i is more important than if the same difference in outcome has been calculated when there are three features in S. The combination of these computations is called the marginal contribution for that subset and the deletion of i.

By making use of this equation, the SHAP value in an AI or machine learning model can be obtained. It is worth noting that the computational complexity of the algorithm's calculation for the SHAP values using this concrete equation, is quite high. Once the number of features grows, the computational time and power needed are growing exponentially. This way of calculating SHAP values is with the KernelSHAP calculation, [VLM22], and so the computational complexity can be stated in the minimum of $O(T2^{D+M})$. Here is T the number of trees in the model. In most of the benchmarking and testing configurations this is set to 1. D represents the depth of the tree and M the number of features in the training data and in the samples used to predict using the model. This shows a high computational complexity, which could result in computational expensive models once multiple and complex trees are used, with high depths and many features [JV23].

2.4.1 SHAP values in decision trees

To be able to calculate the SHAP values (following Equation 3) in decision tree models the computational complexity stated in the previous section should be taken in mind. When the explainable part needs to be applied in real world examples, such as healthcare data, the explainable part should not be limiting the amount of features, trees and/or depth in the trees, as this will limit the performance of the model. This calls for another method to calculate or to approximate the SHAP value.
A frequently used way to achieve this, is by making use of the Monte Carlo algorithm [LS06]. As this algorithm has many usages in the field of game theory, an example in a game is the most appropriate. This algorithm can be explained using the following example: a player is playing a game of checkers against a 'computer'-player. The computer player uses the Monte Carlo algorithm for each step it takes. This is done by taking the current board and playing m random games of checkers, where both the computer and the player take for each turn a random set. The computer calculates for which first step, the number of winning instances is the highest. This is the way how the Monte Carlo algorithm is used in this game of checkers. In this example, m represents the Monte Carlo length, this is the number of random games that will be played in the algorithm.
When applying this algorithm to the approximation of SHAP values, m is the number of random samples that will be taken for the dataset and predicted on the model. For each run of the Monte Carlo approach, the steps are described in Listing 1. Here, the *predict_proba()* function returns

the probability of a correct prediction. This is obtained by dividing the number of samples from the training dataset that have resulted in that specific leaf in the model and the total number of samples in the training data. In this way, the SHAP value is approximated by making use of the Monte Carlo algorithm [SK14]. The rough computational complexity of this structure is $O(m)$, this is a lot lower than the computational complexity of the full calculation of the SHAP value (in Equation 3).

```
1  j = 15 # feature to calculate SHAP value
2  m = 1000
3  total_marginal_contributions = 0
4
5  for i in range(m):
6      sample = X.get_random()
7      prediction_with_j = model.predict_proba(sample)
8      sample[j] = random_value()
9      prediction_without_j = model.predict_proba(sample)
10     total_marginal_contributions += (prediction_with_j - prediction_without_j)
11
12 return (total_marginal_contributions / m)
```

Listing 1: Structure for the approximation of SHAP values using the idea of Monte Carlo algorithms [Ste22].

It is important to note that, in Listing 1 and further shown implementations a special way of deleting features is required. Once a feature is deleted from the sample data array, the length of the array does not comply with the training data of the decision tree model. In this case, the model could not be predicted on when calculating the SHAP values. For that reason, the deleted value is replaced with a random value. This random value will have no relation to the other data in the sample, for which it can be seen as a deleted value, because it makes no sense according to the original data. That is why this way of deleting is used [Mol24].

2.5 Active & Passive security

In the field of MPC, there are two different approaches for the security configurations. These are passive and active security protocols.
In the field of passive security, the MPC protocol will assume that all parties will follow the protocol correctly. However, some parties will try to learn information by analyzing the incoming information. This configuration is honest-but-curious, but not malicious. Except, it can be dangerous for the safety and preserving the privacy of the other parties as one or multiple parties can try to gain information that should be securely saved and not shared when following the protocol strictly.
In active security configurations, parties may try to learn additional information and may also deviate from the protocol in arbitrary ways. The active secure model needs to ensure correctness and privacy, even if some parties act malicious. This is the reason why this model is also called Malicious Security.
In short, the active security model is more secure and preserves the privacy for the inputting parties more.

2.6 XorSHAP

In previous work, the combination of Explainable Artificial Intelligence in a privacy-preserving environment has already been researched. This has been done via the Manticore framework, to achieve the privacy-preserving MPC configurations[CDG+21]. The Manticore framework is an environment where programs can be compiled to ensure the usage of the MPC techniques. The configuration, in the previous work, used the XORBoost technology inside the MPC environment, which causes the algorithm to be called XorSHAP [JV23]. XORBoost is an optimizing technique, used for reducing the number of oblivious permutation evaluations, as well as the quicksort and real number arithmetic algorithms in especially the Manticore framework [DDG+21]. In this thesis research, most parts will be based on the configurations of the algorithm presented in the previous work [JV23]. However, the passive security will be expanded to also support active security. This is the main reason why the Manticore framework would not be sufficient in this case, as the Manticore framework has no ability to support the active security protocols.

3 Methodology & experimental setup

3.1 Use of MPC in Python

To be able to make use of MPC models and their security measures, an environment with the support of MPC has to be used. Regarding the previous related work [JV23], the comparison between the passive security and a active security measure is wanted. In this research, the Manticore framework is used [CDG+21]. However, this framework does not have the support of active security. Therefore, the XorSHAP algorithm will be recreated in the MP-SPDZ framework, as this framework has the ability to apply passive as well as active security measures [Kel20].
To be sure that the decision tree model also uses these MPC configurations, the MP-SPDZ framework has a decision tree classifier class and a trainer included.

3.2 Decision Tree model in MP-SPDZ

As stated before, the decision tree model of the MP-SPDZ framework is used to create the explainable and privacy-preserving decision tree. To start, the framework has to be defined in the use of active and passive security. As the most developments in a MPC framework make the choice of passive or active security beforehand, the framework had to be adapted. In this way, the value stating if the framework needs to work with active security is stored in an instance of the class *program*, this instance is a part of the compilation process of the MP-SPDZ framework.
To start, the decision tree model is built and trained with the data. This is done by the using the structure described in Listing 2.

```
1 from Compiler.decision_tree import TreeClassifier
2 from sklearn.model_selection import train_test_split
3
4 # Load data from dataset
5 X, y = load_data()
6 X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
7
```

```
# Convert data to secure format
X_train = sfix.input_tensor_via(0, X_train)
X_test = sfix.input_tensor_via(0, X_test)
y_train = sint.input_tensor_via(0, y_train)
y_test = sint.input_tensor_via(0, y_test)

# Setup MPC setting for this program
sfix.set_precision_from_args(program)

# Building decision tree model
model = TreeClassifier(max_depth=10, n_threads=2)
model.fit(X_train, y_train)

# Optional to print the output of the model, the tree. It shows if the creation has
    succeeded
model.output()
```

Listing 2: Creating and training a decision tree model inside the MP-SPDZ framework.

By making use of this configuration, the correct use of the MPC protocol is ensured. Until this point, the privacy-preserving decision tree model is built. In the next sections, the way to apply the addition of explainability is explained in the way it is done in this research.

3.2.1 SHAP value calculations

The calculation/approximation of the SHAP values is done in two different ways. It is done via the original formula of the SHAP values (Equation 3) and via the Monte Carlo approximation to the SHAP values (Listings 1).
As described in the previous part, to be able to compute the outcome of the original formula of the SHAP value, a prediction with and without feature j has to be done for every possible subset of features. Here is j the feature, wherefore we want to calculate the feature importance (SHAP value). To get a complete overview of the behaviour of the model, this calculation has to be done for every feature in the sample and training data, and could also be done for multiple samples.

```
import itertools

n_features = len(data[0])
shap_arr = [0] * n_features # Array of length n_features initialized with 0's to
    save the SHAP values

for i in range(len(data)):
    for j in range(0,n_features):
        feature_array = list(range(0,n_features))
        del feature_array[j]
        total_shap_value = 0
        for num in range(len(feature_array)+1):
            for subset in list(itertools.combinations(feature_array, num)):
                for element in subset:
                    if element != j:
                        data[i][element] = random.randint(-1000,1000)
                predict_with_j = tree.predict(data)[i]
```

```
17                      data[i][j] = random.random()
18                      predict_without_j = tree.predict(data)[i]
19                      contribution = sfidata._new(predict_with_j - predict_without_j)
20                      weight = (factorial(num) * factorial(n_features - num - 1))/
                            factorial(n_features)
21                      shap_value = weight * contribution
22                      total_shap_value += shap_value
23
24                      # Resetting the values in data
25                      for element in range(n_features):
26                          data[i][element] = data_samples[i][element]
27              shap_arr[j] += total_shap_value
28
29  for i in range(len(shap_arr)):
30      shap_arr[i] = shap_arr[i] / len(data)
31      print_ln("j: %s = %s",i,shap_arr[i].reveal())
```

Listing 3: Calculating the SHAP values following the Equation 3.

In Listing 3, the calculation of all SHAP values is shown. This algorithm also uses all the samples in the data array and takes the average of the values. This is done to give a best representation of the truthfulness in the SHAP values. In lines 1 to 4, the different arrays and variables are initialized. Afterwards, the first for-loop starts in line 6. This loops takes all the different samples in the data array. The next loop takes numbers from 0 to the amount of features in this sample. This j represents the feature wherefore the SHAP value is being calculated. In the feature array, number from 0 to the amount of features are stored. And the number of the feature j is deleted, as this feature will not be a part of the different subsets. In line 11 & 12, a next loop is started. This loop goes through every different subset of features, the first loop determines the length of the subset and the second loop iterates over every possible subset. In line 13 till 15, the features not in the subset will be eliminated. As the model requires a sample array as input for the prediction from the same format as the training data, the deleted features will be set to a random value. In this way, the truthfulness of this value becomes very low, so it can be seen as 'deleted'.
Afterwards, in the loop which iterates over every subset, the result of the prediction with the feature j and without the feature j is calculated and the difference is measured in the contribution variable. This represents the $\left[f_S(x_{S\cup\{i\}}) - f_S(x_S)\right]$ part of the equation. The calculated weight represents the $\frac{|S|!(n-|S|-1)!}{n!}$ part of the equation. In this way, this algorithm corresponds with the Equation 3. The following part of the algorithm, restores the values changed in the data array and stores the values found. The last section shows the results of the algorithm to the user. It is important to note that the SHAP value is stored inside an secured data structure, so it has to be revealed to the user. This is only for showing the explain-ability data and for testing. The SHAP value is securely saved by the contribution, this is saved inside a secure data-structure during the calculation process.

3.2.2 SHAP value approximations

The approximation of the SHAP value is done by the Monte Carlo algorithm, shown in Listing 1. This is slightly adapted, to ensure that the SHAP value of every feature is calculated.
To be able to execute this algorithm, the MP-SPDZ framework has to be adapted. The *predict_proba*() function is not a standard function implemented in the framework, so this has to be implemented

as following.
The output of the function is determined by Equation 4. Here $\#_{Predict_{Leaf}}$ represents the amount of samples of the training data has ended in the same leaf as current sample does. In the equation $\#_{Total}$ tells the total amount of samples in the training data, so the result of this function represents the probability that the result of the prediction is right, as this fraction is higher, more training data had this 'answer' so more proof that this leaf is right.

$$predict_{proba}(sample) = \frac{\#_{Predict_{Leaf}}}{\#_{Total}} \tag{4}$$

To implement this *predict_proba*() function, the 'fit' function to train the model has to be adapted. This function now has to make an extra value to each leaf. The number of samples that have ended inside this leaf. This is done after the training, because when the model is still being fitted to the training data, nodes and leaves can still shift and for that reason the decision is made to calculate the $\#_{Predict_{Leaf}}$ after the data fitting, so after the fitting and calculations the data can be obtained from the model.

3.3 Experiments

With these algorithms, many tests can be done. First, the difference between different datasets can be tested. As these algorithms have the ability to be explainable and privacy-preserving, they will most likely be used in fields like healthcare and finance. Therefore, there will be an alteration in the number of features and the size of the training dataset.
Furthermore, the performance in computational time and SHAP values will be challenged between the two algorithms.
Most importantly, the difference between the active and passive security models will be tested. To activate the active security protocol or the passive security protocol in the MP-SPDZ framework, the selection between different protocols has to be made before compiling and executing the program. Also, the value *active* of instance *program* has to be set to 1 or 0, depending on the active or passive usage.
In the results, the execution time and memory usage will be tested. The execution time is a result from the execution of the program inside the MP-SPDZ framework, this framework shows the execution time as a benchmarking result after the execution. The memory usage is also a benchmarking result from the MP-SPDZ framework.
In addition to this, the communication costs of the MPC technology used will also be shown.

4 Results

In this thesis, the newly created adapted version of the XorSHAP algorithm is presented. This algorithm has the ability to make use of the active security model. It also has the ability to make use of the equation and the Monte Carlo method for the calculation of the SHAP values.
In the following section, the performance of the algorithms will be tested over different sections and different parameters will be variated. For these tests and benchmarking results, the used processor is an *AMD Ryzen 9 7950X3D*, the RAM memory are 2×16Gb of DDR5-6000. The tests are done on Windows 11, with an installation of WSL2 and the usage of Ubuntu 22.04.

In every testing section, the difference between the performance of active and passive security will be tested.
In the following section, the terms equation algorithm and Monte Carlo algorithm will be used. It is worth noting that the equation method/algorithm referees to the Equation 3 and the Listing 3, The Monte Carlo algorithm/method reference to the method show in Listing 1.
To improve the view on the relations of the data, most of the times, graphs are used. The data of these graphs can be found in Appendix A. For some of the graphs, the data is represented as logarithmic representations, this is done to improve readability.
It is important to note that for all the tests performed, the MP-SPDZ framework is configured with the usage of two parties.

4.1 Performance of the decision tree

In this section, the performance of the creation of the decision tree model inside the MP-SPDZ framework will be tested. This means that the SHAP values are not calculated in the following comparisons and tests. In the first test, the variation will be in the number of features of the training dataset, afterwards the size of the training dataset is altered. As last, the depth of the decision tree is altered. The effects are compared in the active and passive security configuration. For the different tests, the execution time and the memory usage in MB is measured, a MB is seen as 1024^2 bytes.

4.1.1 Number of features

In this first test, the number of features is altered. The number of features is the number of columns in the training dataset.

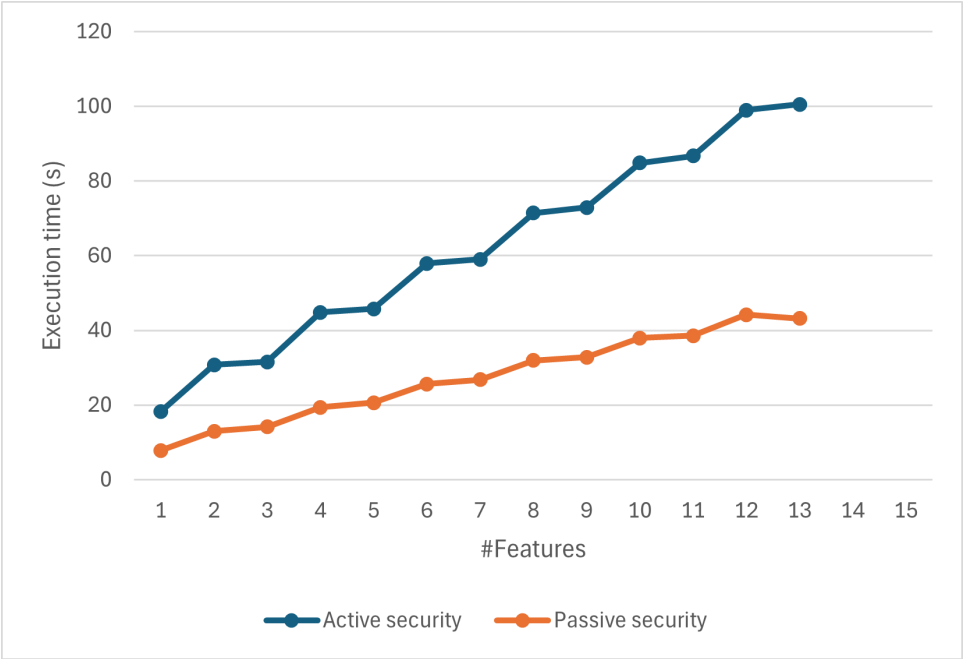

Figure 3: This figure shows the difference in execution time for the creation of the decision tree model in the active and passive security model.

In Figure 3, the effect of the usage of the active and passive security protocol can be seen. In both protocols, the amount of time needed to create the decision tree grows along with the number of features used. It can also be seen that the active security model requires a longer execution time than the passive security model. This can be explained by the structure of these protocols. While the passive security protocol expects that the each party in the model will not try to gain information about the other parties, the computations can be performed faster. Because, the model has not to be aware of the party's behaviour. The active security protocol however, is aware of the fact that many parties may try to gain information and is stopping them from this action. This takes more computational power, as seen in Figure 3.

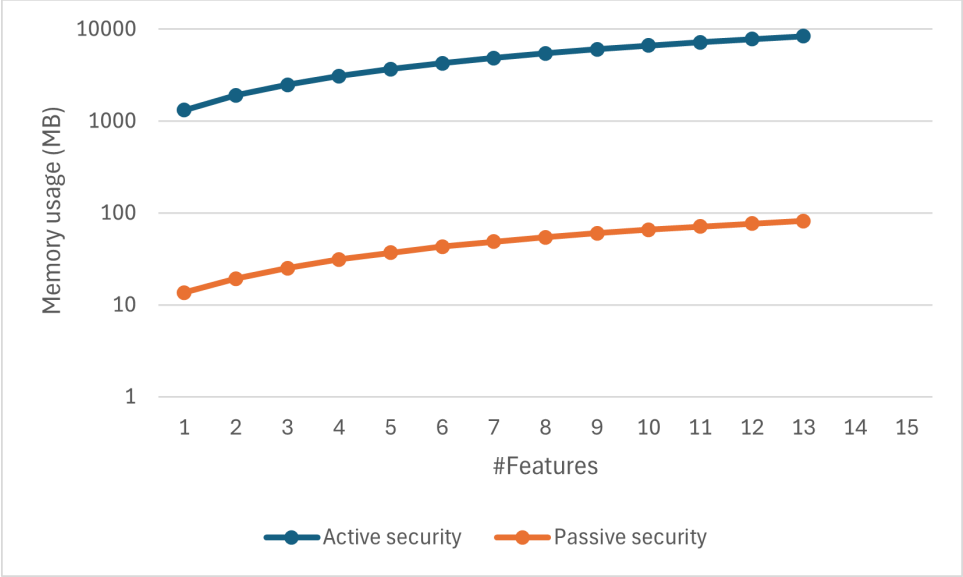

Figure 4: This figure shows the difference in memory usage in the active security protocol and the passive security protocol. It is important to note that the graph uses a logarithmic values on the vertical axis. This is done to improve the readability of the graph, as the difference between the active and passive security model is high.

In Figure 4, the memory usage of the creation for the decision tree model can be seen, when altering over the number of features. It can be seen that the strength of the linear growth for the active and passive security model is the same. However, it can be seen that the active security protocol requires more memory than the passive security model. The reason for this effect can be seen as the same as stated with Figure 3.

4.1.2 Size of training set

In this section, the number of samples in the training dataset is altered. This number has an effect on the training of the decision tree model. As the amount of data for training the model grows, it is expected that the model requires more execution time and memory usage.

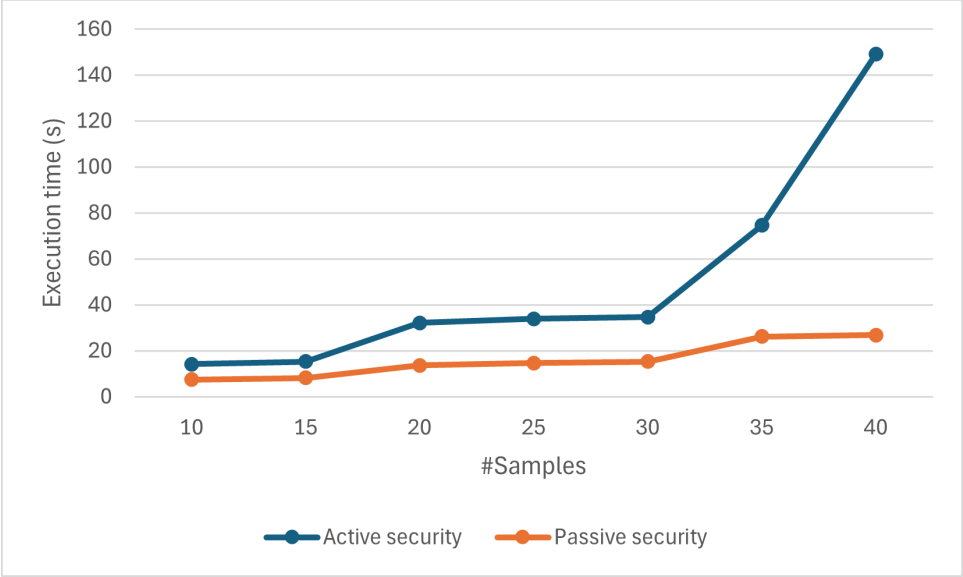

Figure 5: This figure shows the difference in execution time for the active and passive security protocol. In this graph the number of samples (size) in the training dataset is altered.

In Figure 5, the difference in execution time for both security models can be seen. It shows that the active security model requires more execution time than the passive security model in general. It can also be seen that for 30 samples and more, the active security model significantly increases in execution time, compared to the passive security model. This can be explained by the fact that an increase in the number of features will also increase the change of a party trying to gain information, as more data is used. This is the reason for the significant increase of execution time.

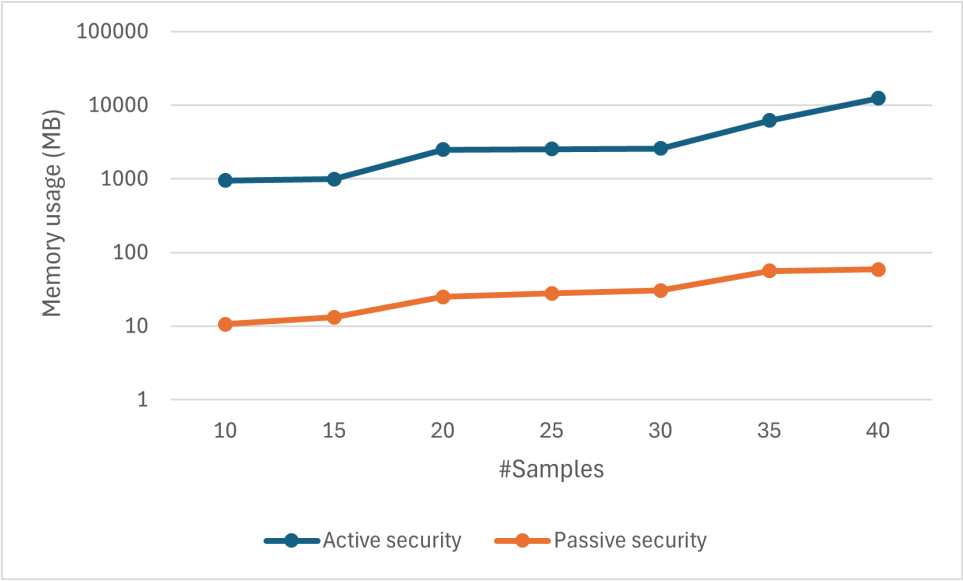

Figure 6: This figure shows the difference in memory usage in the active security protocol and the passive security protocol. It is important to note that the graph uses a logarithmic values on the vertical axis. This is done to improve the readability of the graph, as the difference between the active and passive security model is high.

In Figure 6, the amount of memory usage for both security models can be seen. As in Figure 4, the logarithmic representation is used. In this graph, it can again be seen that the active security model requires more memory usage than the passive security model. The effect, once the number of features exceeds the value of 30, the memory usage of the active security protocol grows significantly faster than the passive security model. This effect is also seen in the execution time, shown in Figure 5.

4.1.3 Depth of the tree

In this section, the depth of the decision tree is altered. The depth of the tree is the number of levels in the tree, once the depth of the tree rises, it is expected that the creation and training of the tree requires more computational power. This will result in a longer execution time and a higher memory usage.

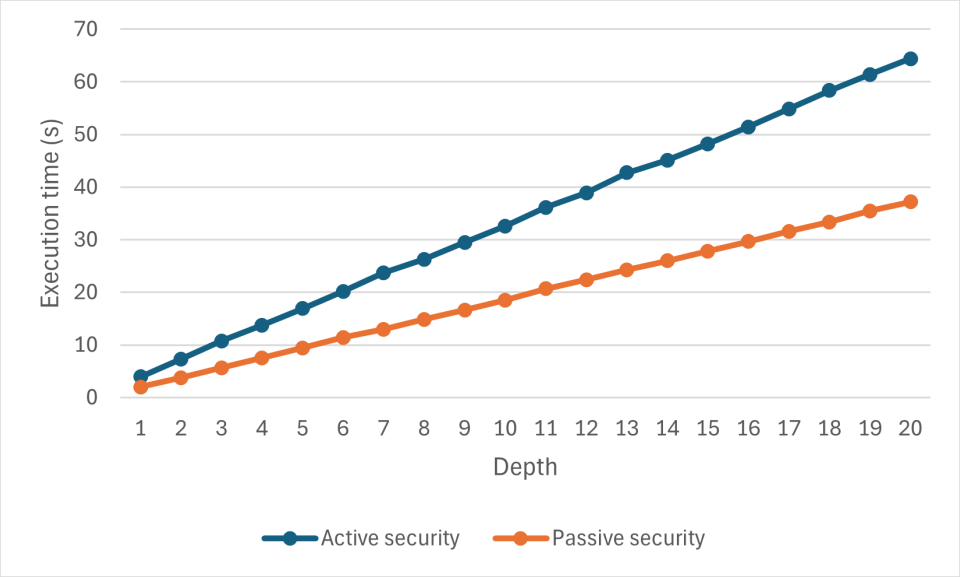

Figure 7: This figure shows the difference in execution time for the active and passive security protocol. In this graph, the depth of the tree is altered.

In Figure 7, the execution time of both security models while altering the depth of the decision tree can be seen. It is expected that, if the depth of the tree grows, the execution time needed to create the tree grows accordingly. This effect can be seen in the graph. Another effect which can be obtained from the graph, is the fact that the execution time of the active security model grows significant faster than the execution time of the passive security model. This effect can also be seen in the previous section. It is presumable that this effect is caused by the same reason.

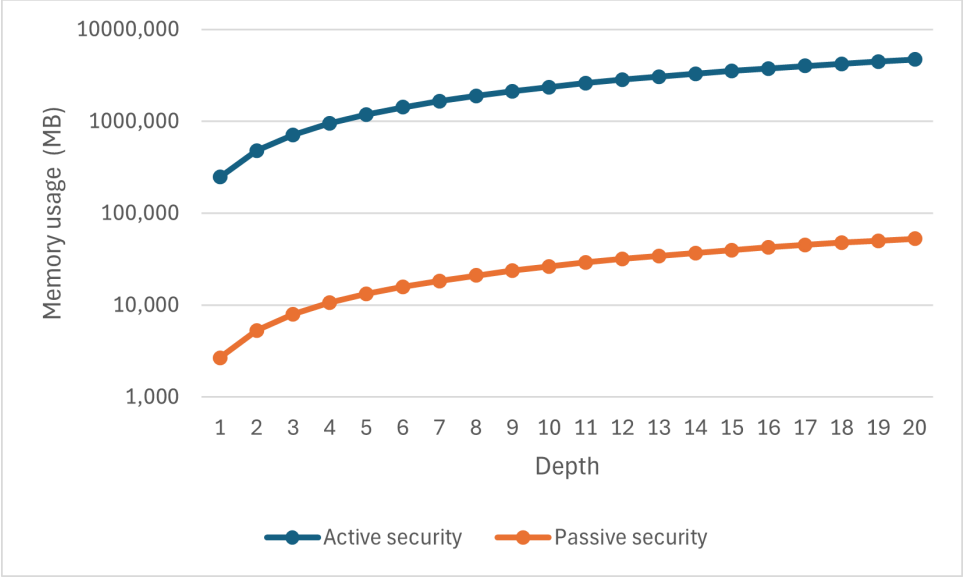

Figure 8: This figure shows the difference in memory usage in the active security protocol and the passive security protocol. It is important to note that the graph uses a logarithmic values on the vertical axis. This is done to improve the readability of the graph, as the difference between the active and passive security model is high.

Figure 8, the memory usage of the passive and active security model can be seen. As in the previous generated graphs, this graph also shows a logarithmic representation of the values. It can be stated from the graph, that there is only a slight increase in growth of the memory usage in the active security model once the depth grows, compared to the passive security model.

4.2 Full integrated model

In this section, the effects of the full integration will be shown. In this full integration, the calculation of the SHAP values and the creation of the decision tree model are combined. This ensures the privacy-preserving and explainable decision tree. In the first section, the Equation method and the Monte Carlo method are compared. Afterwards, the effect of the active and passive security protocols are tested in this full integrated model.

4.2.1 Equation and Monte Carlo method

In section, the comparison between the usage of the Equation and the Monte Carlo method is made.

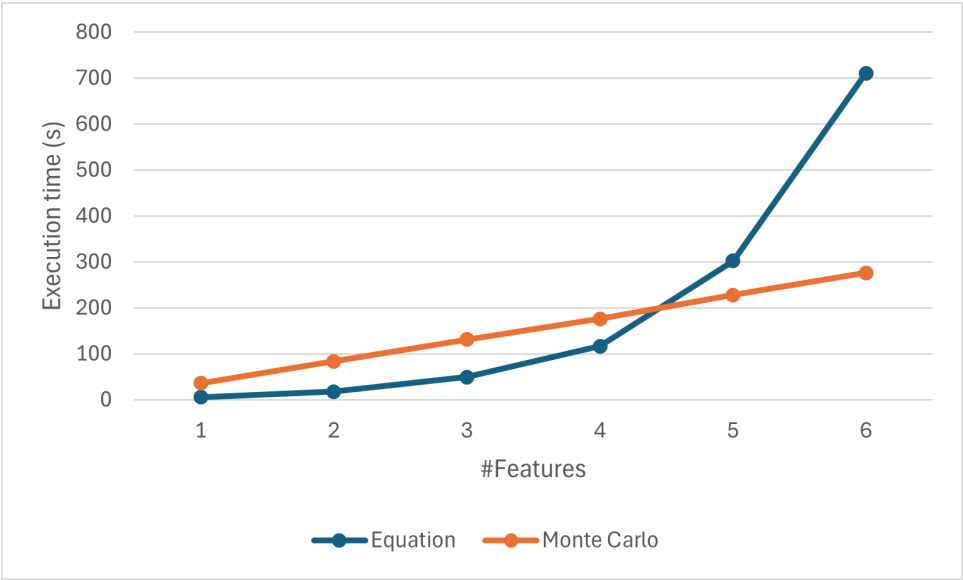

Figure 9: This graph shows the execution time of the Equation algorithm and the Monte Carlo algorithm to calculate the SHAP values in combination with the creation of the decision tree model. These calculations are done in the active security protocol.

In Figure 9, the execution times of the calculation for both methods to calculate the SHAP values are given. It can been seen that the Monte Carlo configuration shows a linear relation to the number of features according to the execution time. The equation method however, shows an exponential graph. According to the expected computational complexity of the equation, $O(T2^{D+M})$, the behaviour in the graph is as expected. Since the number of trees (T) is 1 and the depth (D) is constant, the function of the graph would be 2^M and this behaviour is indeed shown in the graph. The computational complexity of the Monte Carlo algorithm is expected to be $O(M)$, so the execution time should be linear with the amount of features in the model. This effect can also be seen in the figure.

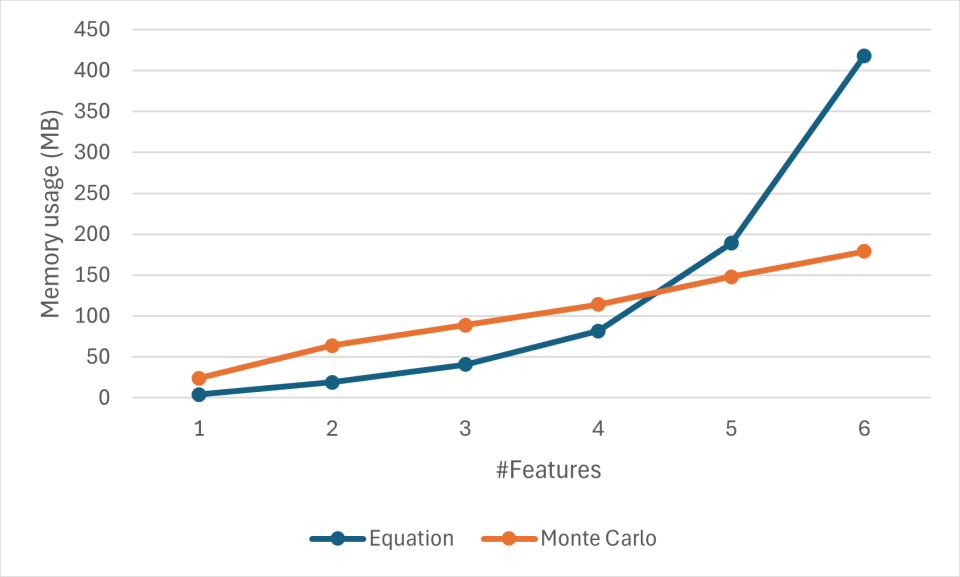

Figure 10: This graph shows the memory usage for both methods for the calculation of the SHAP values.

To compare these methods even further, there can be looked at the difference in memory usage of the SHAP calculation functions. This is shown in Figure 10. The same relation of the methods is represented in the memory usage. However, these relations are not quite well visible as shown in Figure 9. The most likely reason for this, is that the memory usage is also used by many other influencing parties, so the memory usage is not always a direct representative of the computational requirements of an algorithm.

4.2.2 Active and passive security

In this section, the performance of the active security and the passive security protocol are compared.

To start, the usage of the active and passive security protocol is tested in the Monte Carlo method.

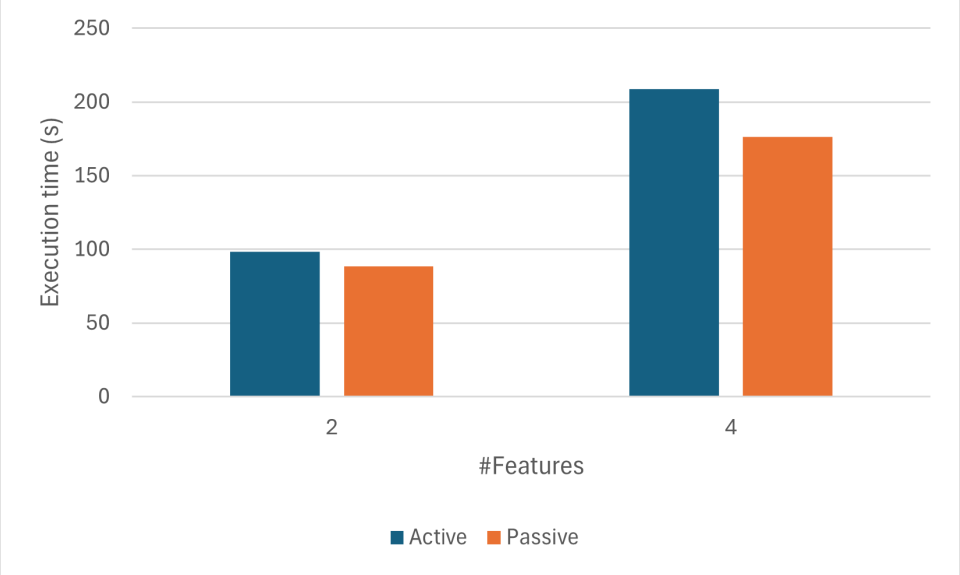

Figure 11: This graph shows the effect of the use of the active security model in combination with the Monte Carlo method. There is a variation between 2 and 4 features.

In Figure 11, the effect of the active security model can be seen. The results are as expected, due to the previous results. It can be stated that the active security model requires more execution time to calculate the explainable and privacy-preserving decision tree model, and once the number of features is higher, this difference grows.

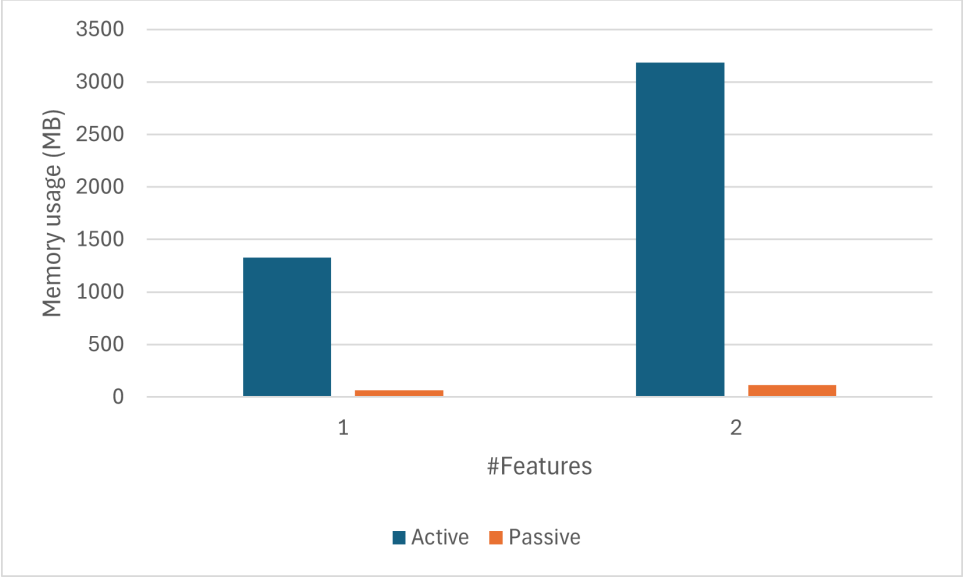

Figure 12: This graph shows the effect of the use of the active security model in combination with the Monte Carlo method. The memory usage of the model is shown.

Figure 12, shows the difference of the memory usage for the active and passive security model in the Monte Carlo method. As in Figure 11, the active security model is computationally more expensive and once the amount of features are higher, the memory usage grows accordingly.

To compare these results with the equation method, the same tests are performed again and the results are shown in the following Figures 13, 14.

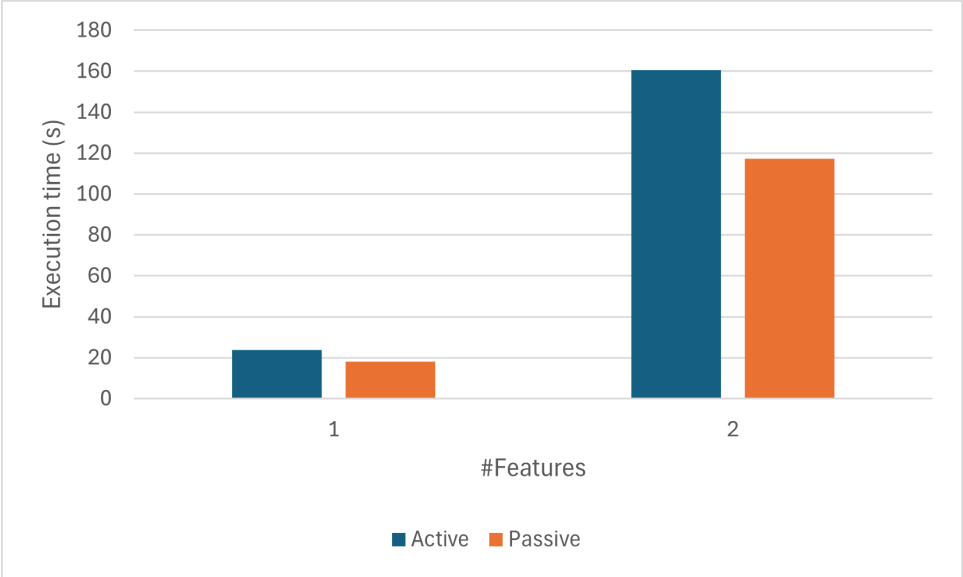

Figure 13: This graph shows the effect of the use of the active security model in combination with the equation method.

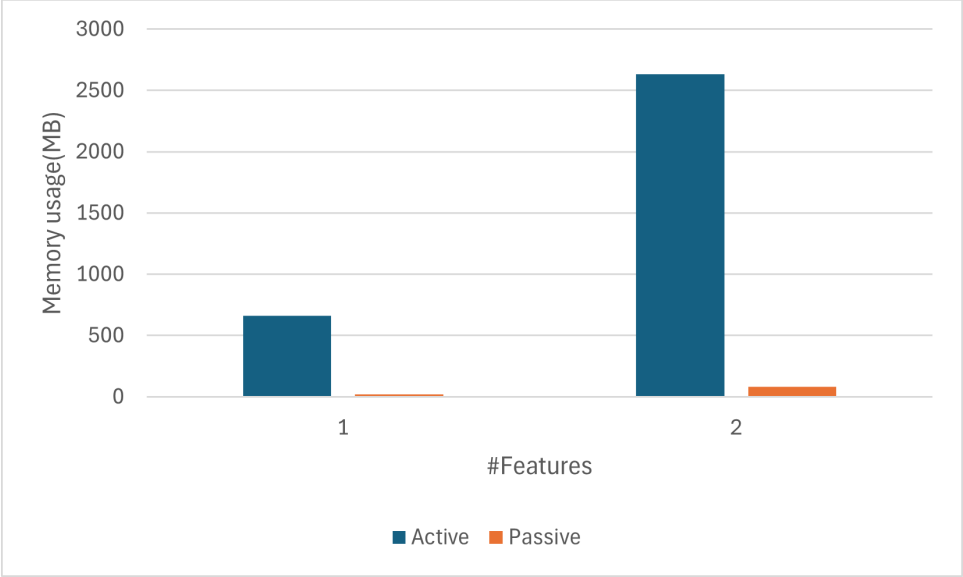

Figure 14: This graph shows the effect of the use of the active security model in combination with the equation method. The memory usage of the model is shown.

In these graphs (shown in Figures 13, 14), the same result as the Monte Carlo method can be obtained. In addition to this, it can be seen that by the usage of the equation method, the difference between the active and passive security method are more significant, especially when compared using the execution time. This is a result of a phenomenon already seen in Figure 9, once the number of features becomes higher, the equation method becomes more computationally expensive.

4.3 Communication costs

The communication costs for the execution of the program inside the MPC framework depend mostly on the message complexity. In this implementation the message complexity refers to the size and number of messages between the parties. This means that, the message complexity will be higher once more complex data is used and once another protocol is used. With the knowledge about the difference between the active and passive security model, described in Section 2, it is expected that the active security model will show a higher communication cost. The communication costs are measured in MB, as the transferred data represents the communication cost, this amount of data is measured in MB. The communication costs shown in this section is the total of communication costs for both parties.
The following tests are done by the usage of the Monte Carlo method and two parties in the configuration.

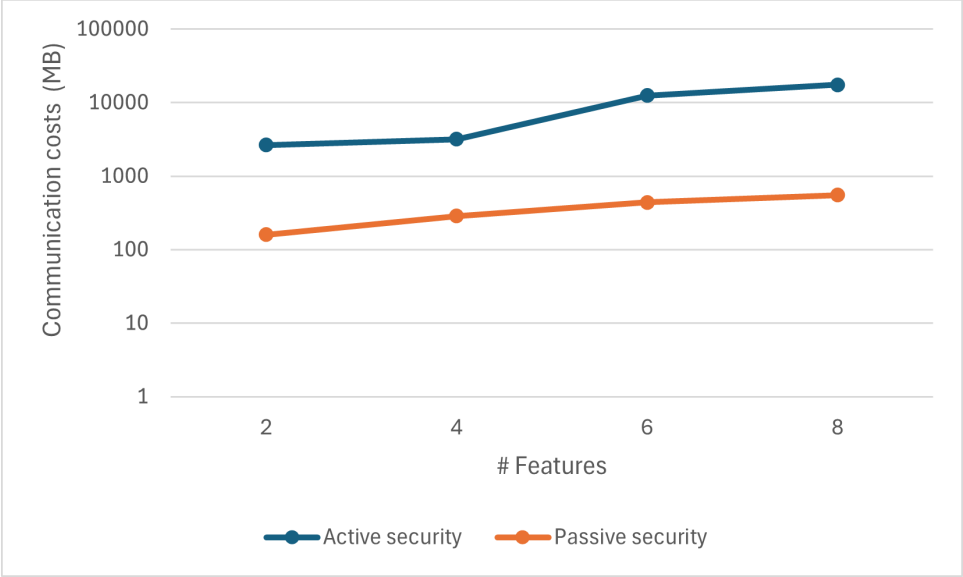

Figure 15: This figure shows the difference in communication costs for the active and passive security model. For this execution the Monte Carlo method is used. Variation is performed with the number of features. It is important to note that the graph uses a logarithmic values on the vertical axis. This is done to improve the readability of the graph, as the difference between the active and passive security model is high.

The expected results can be seen in Figure 15. In this figure, the variation is done over the number of features. Once the number of features is higher, the message complexity also rises. This results in the rise of the communication costs, this effect can be seen in Figure 15.
Another effect can be seen in this graph, the relation between the communication costs and the number of features for the passive and active security model are linear. However, once the execution in the active security model rises above 4 features, the difference in communication costs between both security models is higher. This difference is the result of a more complex decision tree. Once the number of features becomes higher, the number of computations that have to be done in order to train the decision tree and perform calculations with the tree grow alongside with the number of features. It is expected that a computation in the passive security model requires a smaller message between the parties than the active security model. For this reason, the memory complexity grows faster in the active security model, in comparison to the passive security model, once the number of features grows. This is the same effect as seen in Figure 15. The same effect can also be seen in Figure 3.

4.4 Performance of SHAP values

As the measurement of explainability used in this thesis, which is the usage of SHAP values, the performance of them is important. That is why, in this section, the results of the two methods will be compared and shown.
First, the equation method. By making use of the MP-SPDZ framework, the decision tree model is limited in prediction in only 0 and 1. This makes it more difficult to calculate the SHAP values using the equation method. The cause of this is the final part of Equation 3, where the prediction

of the sample with the value of the feature and without the value of the feature are being compared. Once these predictions meets the same result, the value of this comparison is equal to 0 and this results into the fact that the final outcome for the calculation for that subset is 0. However, if a model has many features, it is likely that the deletion of one feature might often give the same result in the prediction. Due to the fact that many features also result in a great amount of subsets of these features, once a difference is found, the difference will likely not be significant as the other values will take over. This gives a problem with the direct equation method for the calculation of the SHAP values. However, the results are a representation of the reality, as the deletion of a feature will not result in a change of result of the prediction. This means that the equation method will show the correct answer, but will not always give the values wherefore the maximal explainability can be observed. The solution for this problem is the usage of the Monte Carlo algorithm. Yet, it should be noted, that this algorithm is an approximation and the equation method will still give the correct answer to the calculation of the SHAP value.
For this reason, first the comparison between the obtained SHAP values will be given. Here, the comparison is show for samples with 4 features. This shows the difference in SHAP values.

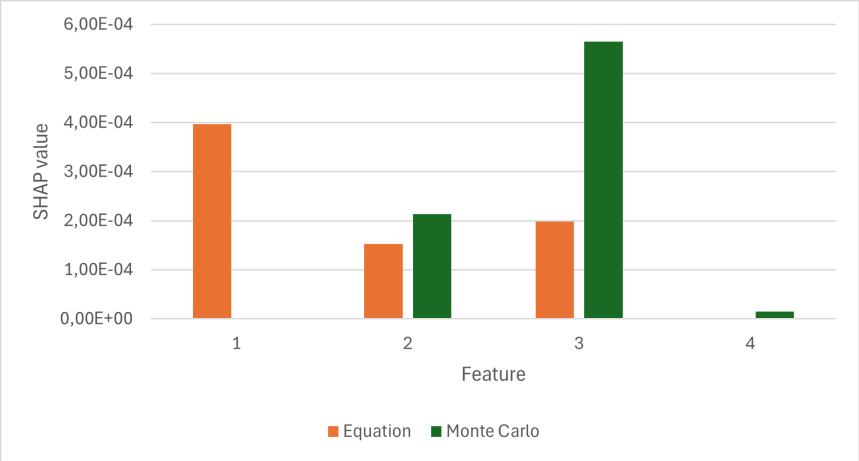

Figure 16: This figure shows the difference in SHAP values for both calculation methods. The training dataset with 4 features is used, with a tree depth of 3.

Figure 16 shows the difference in SHAP values that are calculated. It can be seen that the value of the SHAP values is very low. This means that there is not one specific feature which making a big difference in the prediction of the dataset. However, a big difference in feature importance can be seen in feature 1, where the equation method find the most importance, but the Monte Carlo method sees feature 3 as most important.
To research the well behaving of the SHAP value calculation, a new dataset is created. In this dataset, feature 1 is made important, as if the value of the first feature meets the threshold > 50 the outcome of the target in the training data will always be 1. The outcome of this test can be seen in Figure 17. As expected, it shows that both methods find the first feature important, the other features do not have any importance. There is a small difference between the equation method and the Monte Carlo method, this can be explained by the fact that the Monte Carlo method uses a approximation and the equation show the result of the direct formula of the SHAP value.

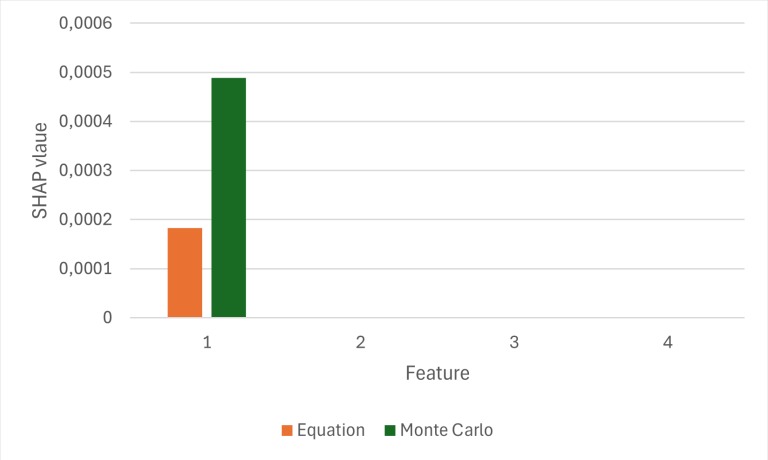

Figure 17: This figure shows the difference in SHAP values for both calculation methods. The training dataset with 4 features is used, with a tree depth of 3. The dataset is manipulated so the target of the dataset is always 1, if the first feature has a value > 50.

In general, it can be seen that the results of the SHAP values do give the same result. This means that the approximation of the Monte Carlo method can also be used to give an overview of the importance for each feature and so give the possibility to explain the results of the algorithm.

5 Conclusion

In this thesis, the newly created version of the privacy-preserving explainable AI with the ability to make use of active security protocols, is presented. It is also shown that the calculation of the SHAP values, for expanding the AI with explainability, can be done in two different ways. This can be done via the presented Monte Carlo algorithm or via the equation method, where the Monte Carlo method is based on the structure of a Monte Carlo algorithm and the equation method is based on the direct formula (Equation 3) of the SHAP value.
By taking a look at the differences between the these models, it can be stated that the Monte Carlo method is less computationally expensive than the equation method, but the equation method will always show the abstract calculated SHAP value. The Monte Carlo method is an approximation of the SHAP value. In most cases the Monte Carlo method will succeed in the goal to create explainability of the decision tree model.
The main goal of this thesis was to expand the privacy-preserving explainable AI with the support of the active security protocols. This goal is accomplished by the adapted usage of the MP-SPDZ framework.
In addition to this, the usage of the passive and the active security protocols is compared. In this comparison it can be seen that the active security model is computationally more expansive, as the model requires in all situations more memory and more execution time for privacy-preserving explainable decision tree model.

References

[Abb21] Hussein Abbass. Editorial: What is Artificial Intelligence? *IEEE transactions on artificial intelligence*, 2:94–95, 2021.

[Boz21] Beyza Bozdemir. *Privacy-preserving machine learning techniques*. PhD thesis, 12 2021.

[CDG+21] Sergiu Carpov, Kevin Deforth, Nicolas Gama, Mariya Georgieva, Dimitar Jetchev, Jonathan Katz, Iraklis Leontiadis, M. Mohammadi, Abson Sae-Tang, and Marius Vuille. Manticore: Efficient Framework for Scalable Secure Multiparty Computation Protocols. Cryptology ePrint Archive, Paper 2021/200, 2021. https://eprint.iacr.org/2021/200.

[DDG+21] Kevin Deforth, Marc Desgroseilliers, Nicolas Gama, Mariya Georgieva, Dimitar Jetchev, and Marius Vuille. XORBoost: Tree Boosting in the Multiparty Computation Setting. Cryptology ePrint Archive, Paper 2021/432, 2021. https://eprint.iacr.org/2021/432.

[GWS20] Randy Goebel, Wolfgang Wahlster, and Jörg Siekmann. *Lecture Notes in Artificial Intelligence Subseries of Lecture Notes in Computer Science Series Editors Founding Editor*. 2020.

[JV23] Dimitar Jetchev and Marius Vuille. XorSHAP: Privacy-Preserving Explainable AI for Decision Tree Models. Cryptology ePrint Archive, Paper 2023/1859, 2023.

[Kel20] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. Cryptology ePrint Archive, Paper 2020/521, 2020. https://eprint.iacr.org/2020/521.

[Led19] Heidi Ledford. Millions of black people affected by racial bias in health-care algorithms. *Nature (London)*, 574:608–609, 2019.

[Lin20] Yehuda Lindell. Secure Multiparty Computation (MPC). Cryptology ePrint Archive, Paper 2020/300, 2020. https://eprint.iacr.org/2020/300.

[LS06] H.W. Lenstra and C.P. Schnorr. A Monte Carlo factoring algorithm with linear storage, 2006.

[Mak21] Eleftheria Makri. Secure and Efficient Computing on Private Data, 2021.

[Mas21] Serg Masis. *Interpretable machine learning with Python : learn to build interpretable high-performance models with hands-on real-world examples* . Packt, Birmingham, England ;, 2021.

[Mol24] Christoph Molnar. *Interpretable Machine Learning*. 2 edition, 2024. Accessed: 2024-06-28.

[MPC23] Mayuri Mehta, Vasile Palade, and Indranath Chatterjee. *Explainable AI : foundations, methodologies and applications* . Intelligent systems reference library ; 232. Springer, Cham, Switzerland, 2023.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?" Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. ACM, 2016.

[Sha53] Lloyd S. Shapley. A Value for N-Person Games. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games*, volume 2, pages 307–317. Princeton University Press, 1953.

[SK14] Erik Strumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems*, 41(3):647–665, December 2014.

[Ste22] Tobias Sterbak. How to calculate shapley values from scratch, Jul 2022.

[TBS+21] Amirhosein Toosi, Andrea G. Bottino, Babak Saboury, Eliot Siegel, and Arman Rahmim. A Brief History of AI: How to Prevent Another Winter (A Critical Review). *PET clinics*, 16:449–469, 2021.

[VLM22] Mattia Villani, Joshua Lockhart, and Daniele Magazzeni. Feature Importance for Time Series Data: Improving KernelSHAP, 2022.

Appendix

A Tables

# Features	Active security (s)	Passive security (s)
2	18,3102	7,84887
3	30,7722	13,0038
4	31,5578	14,1868
5	44,7812	19,3899
6	45,7548	20,5879
7	57,9584	25,6204
8	59,02	26,7761
9	71,4457	31,9072
10	72,8659	32,7727
11	84,8236	37,9718
12	86,7424	38,5501
13	98,9428	44,1977
14	100,492	43,1532

Table 1: This table represents the data used for Figure 3.

Features	Active security (MB)	Passive security (MB)
2	1312.8	13.5564
3	1901.06	19.2817
4	2486.14	25.133
5	3079.08	31.128
6	3672.16	37.004
7	4256.59	42.8812
8	4844.01	48.5796
9	5436.5	54.4592
10	6025.89	60.1315
11	6613.66	65.6678
12	7199.78	71.0083
13	7788.93	76.446
14	8374.99	81.8245

Table 2: This table represents the data used for Figure 4

#samples	Active security (s)	Passive security (s)
10	14,1729	7,43323
15	15,3171	8,23061
20	32,1876	13,7153
25	33,8425	14,7145
30	34,7102	15,2944
35	74,5399	26,2016
40	149,0798	26,8705

Table 3: This table represents the data used for Figure 5.

samples	Active security (MB)	Passive security (MB)
10	949.472	10.6251
15	994.027	13.1096
20	2486.14	25.0651
25	2542.11	27.8525
30	2586.52	30.5478
35	6189.16	56.1386
40	12378.32	59.0162

Table 4: This table represents the data used for Figure 6.

Depth	Active security (s)	Passive security (s)
1	3,961	2,016
2	7,303	3,795
3	10,797	5,629
4	13,742	7,531
5	16,923	9,448
6	20,212	11,469
7	23,723	12,975
8	26,247	14,847
9	29,524	16,668
10	32,627	18,561
11	36,176	20,683
12	38,958	22,377
13	42,739	24,308
14	45,157	26,009
15	48,196	27,874
16	51,427	29,658
17	54,854	31,627
18	58,375	33,415
19	61,431	35,487
20	64,411	37,243

Table 5: This table represents the data used for Figure 7.

Depth	Active security (MB)	Passive security (MB)
1	246.901	2.679
2	480.629	5.308
3	714.073	7.957
4	949.472	10.625
5	1187.510	13.286
6	1419.200	15.748
7	1653.170	18.376
8	1887.970	21.004
9	2121.350	23.632
10	2359.180	26.260
11	2591.150	29.011
12	2827.590	31.651
13	3061.130	34.292
14	3292.540	36.933
15	3530.120	39.574
16	3764.600	42.378
17	3996.070	45.029
18	4229.090	47.668
19	4462.540	50.228
20	4703.240	52.789

Table 6: This table represents the data used for Figure 8.

# Features	Equation (s)	Monte Carlo (s)
1	5,91146	36,082
2	17,8407	83,3213
3	49,1036	130,887
4	116,189	175,718
5	302,24	228,125
6	710,807	276,136

Table 7: This table represents the data used for Figure 9.

# Features	Equation (MB)	Monte Carlo (MB)
1	4,11885	23,9012
2	18,8378	63,7328
3	40,7783	88,897
4	81,728	114,228
5	189,065	147,932
6	417,964	178,94

Table 8: This table represents the data used for Figure 10.

# Features	Active security (s)	Passive security (s)
2	98,1717	88,5848
4	208,763	176,184

Table 9: This table represents the data used for Figure 11.

# Features	Active security (MB)	Passive security (MB)
2	1325,38	63,7328
4	3184,21	114,228

Table 10: This table represents the data used for Figure 12.

# Features	Active security (s)	Passive security (s)
2	23,7074	18,0884
4	160,569	117,374

Table 11: This table represents the data used for Figure 13.

# Features	Active security (MB)	Passive security (MB)
2	659,505	18,8378
4	2632,25	81,728

Table 12: This table represents the data used for Figure 14.

# Features	Active security (log base 10)	Passive security (log base 10)	Active security (MB)	Passive security (MB)
2	3,423673	2,204828	2652,61	160,261
4	3,503002	2,458107	3184,21	287,149
6	4,096455	2,6433	12486,9	439,845
8	4,240437	2,740626	17395,5	550,334

Table 13: This table represents the data used for Figure 15.

Feature	Equation	Monte Carlo
1	3,97E-04	0
2	1,5E-04	2,1E-04
3	1,98E-04	0,00056458
4	0	1,5E-05

Table 14: This table represents the data used for Figure 16.

Feature	Equation	Monte Carlo
1	0,000183	0,00048828
2	0	0
3	0	0
4	0	0

Table 15: This table represents the data used for Figure 17.

B Code

```
# Importing libraries
from sklearn.model_selection import train_test_split
from Compiler.decision_tree import TreeClassifier
import pandas as pd
import random
import itertools

# Change the active or passive security usage
program.active = True # Active security is enabled

# Getting the data from the test.csv file
data = pd.read_csv("data/test.csv")
# Getting the first 4 columns as sample data
X=data.values[:,0:4]
# The last comlumn contains the value of the targets
y=data.values[:,4]
# Splitting the dataset into training and testing datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# Convert data to secure format
X_train = sfix.input_tensor_via(0, X_train)
X_test = sfix.input_tensor_via(0, X_test)
y_train = sint.input_tensor_via(0, y_train)
y_test = sint.input_tensor_via(0, y_test)

# Setting the settings for the compilation and execution inside the framework
sfix.set_precision_from_args(program)

# Creating the decision tree model
model = TreeClassifier(max_depth=3, n_threads=2)
# Training and fitting the model with the training samples
model.fit(X_train, y_train)

print_ln("STARTING SHAP CALCULATION: ...")
# Function used to calculate the factorial of the input
def factorial(num):
    result = 1
    for i in range(1, num + 1):
        result *= i
    return result

# Setting up the variables used in the calculation function
n_features = len(X_train[1])
```

```
43 shap_arr = [0] * n_features
44 x_copy = sint._new(X_train)
45
46 # Function to create the SHAP values using the Equation method
47 def calculateShapEquation(length):
48     for i in range(length):
49         for j in range(0,len(X_train[0])): # Looping over the features in the
               sample
50             feature_array = list(range(0,n_features)) # Generating a array of the
                   subsets
51             del feature_array[j] # Making sure the feature j chosen will not be an
                   element in the subset
52             total_shap_value = 0
53             for num in range(len(feature_array)+1): # Looping over the lengths of
                   the subsets
54                 for subset in list(itertools.combinations(feature_array, num)): #
                       Looping over the subsets in the subset array
55                     for element in feature_array: # Looping over the elements in
                           the feature_array and deleting the features which are not in
                            the subset
56                         if element != j and not element in subset:
57                             x_copy[i][element] = random.randint(-1000,1000)
58                     predict_with_j = model.predict(x_copy)[i] # Prediction with
                           feature j
59                     x_copy[i][j] = random.randint(-1000,1000)
60                     predict_without_j = model.predict(x_copy)[i] # Prediction
                           without feature j
61                     contribution = sfix._new(predict_with_j - predict_without_j) #
                           Calculating the contribution in secure format
62                     weight = (factorial(num) * factorial(n_features - num - 1))/
                           factorial(n_features) # Calculating the weight
63                     shap_value = weight * contribution # Calculating the SHAP value
64                     total_shap_value += shap_value
65                     for element in range(n_features): # Resetting the values
                           changed in the x_copy array
66                         x_copy[i][element] = X_train[i][element]
67             shap_arr[j] += total_shap_value
68
69     for i in range(len(shap_arr)): # Printing the results
70         shap_arr[i] = shap_arr[i] / length
71         print_ln("Feature %s has SHAP value %s",i,shap_arr[i].reveal())
72
73 calculateShapEquation(len(X_train[0])) # Making sure the SHAP values are being
       calculated over all the elements in x_train
```

Listing 4: This code is the Python code used for the execution of the algorithm by making use of the equation method.

```
# Importing libraries
from sklearn.model_selection import train_test_split
from Compiler.decision_tree import TreeClassifier
import pandas as pd
import random

# Change the active or passive security usage
program.active = True # Active security is enabled

# Getting the data from the test.csv file
data = pd.read_csv("data/test.csv")
# Getting the first 4 columns as sample data
X=data.values[:,0:4]
# The last comlumn contains the value of the targets
y=data.values[:,4]
# Splitting the dataset into training and testing datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# Convert data to secure format
X_train = sfix.input_tensor_via(0, X_train)
X_test = sfix.input_tensor_via(0, X_test)
y_train = sint.input_tensor_via(0, y_train)
y_test = sint.input_tensor_via(0, y_test)

# Setting the settings for the compilation and execution inside the framework
sfix.set_precision_from_args(program)

# Creating the decision tree model
model = TreeClassifier(max_depth=3, n_threads=2)
# Training and fitting the model with the training samples
model.fit(X_train, y_train)

# The function totalOfArr calculates the sum of the values in the input array
def totalOfArr(arr):
    total = 0.0
    for obj in range(len(arr)):
        total += arr[obj]
    return total

print_ln("STARTING SHAP CALCULATION: ...")

def calculateShapValue(length):
    shap_arr = sfix.Array(len(X_train[0])) # Array to save the SHAP values
    for j in range(len(X_train[0])): # Looping over the different features
        M = length # Setting monte carlo length
        marginal_contributions = sfix.Array(M) # Storage for the totals of the
            marginal contributions
        x_copy = sint._new(X_train) # copy of the x_train array
        for obj in range(M): # Performing calculation for as long the as the Monte
            Carlo length
            # Getting a random sample
            i = random.randint(0,len(x_copy))
            # Calculating the probability of the prediction with and without the
                feature j
            predWithJ = model.predict_proba(x_copy)[i]
```

```
52              x_copy[i][j] = random.randint(-10000,10000)
53              predWithoutJ = model.predict_proba(x_copy)[i]
54              # Calculate marginal contribution
55              marginal_contribution = predWithJ - predWithoutJ
56              marginal_contributions[obj] = marginal_contribution
57              # Resetting the x_copy variable
58              x_copy[i][j] = X_train[i][j]
59          shap_arr[j] = totalOfArr(marginal_contributions.reveal()) # Calculating the
                SHAP value
60      for i in range(len(shap_arr)): #Printing the results of the SHAP value
            calculation
61          shap_arr[i] = shap_arr[i] / length
62          print_ln("Feature %s has SHAP value %s",i,shap_arr[i].reveal())
63
64  calculateShapValue(1000) # Making sure the SHAP values are being calculated with
        the Monte Carlo length of 1000
```

Listing 5: This code is the Python code used for the execution of the algorithm by making use of the Monte Carlo method.