ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Enclave Hardening for Privacy Preserving Machine Learning

Master Thesis

Rudolf Loretan

November 17, 2021

Supervisor: Prof. Dr. S. Capkun

Advisors: Dr. K. Kostiainen, Prof. Dr. E. Mohammadi

Department of Computer Science, ETH Zürich

Abstract

Hello world. This is an abstract. Why am I not showing up?

— Intel's Software Guard Extensions (SGX) is a new technology introduced in recent generations of Intel processors. SGX is supposed to be able to create a trusted execution environment for user-space software that is protected from all privileged software running on the same system. The CPU creates a protected enclave in memory for the software and guards the memory using strict access control and encryption with keys derived from secrets embedded inside the CPU.

Our practical contributions and evaluation branch into three main areas: (1) privacypreserving data release using generative adversarial networks (GANs); (2) private classification using convolutional neural networks and other ML models; and (3) private federated learning —

DP-GBDT (Differentially-Private Gradient Boosted Decision Trees) put the def here?

practical usability demonstrated on standard UCI datasets

Acknowledgements

Rough content:

- that was not so easy with zero ML experience
- combination of really new to me areas (ML,dp, hardening, sgx). 2 ganz new, 2 heard of but no hands on experience.
- thanks to my cat olga

Thesis TODOs, (notes for me)

- upload the whole project to github, and reference it correctly
- make (my) github repo(s) as footnote instead of cite
- clean up security diagrams. names and style consistent with the text
- replication -> mseed distribution, questionnair counter needs ID, retrain counter wrong
- pseudocode width -> inside minipage -> not float anymore -> large, ugly line sep
- should I mention our/the proof? ask esfandiar
- adjust security diagram sizes in latex
- some moreovers instead of furthers
- decide american english or british english
- probably capitalize the words Figure, Table etc.
- i.e. means "that is". not for example!

coding todos

- run other datasets in SGX
- create a light hardened version for measurements
- sgx the hardened
- (adult no-dp measurement)
- hardened adult parser bug with weight column?
- upload the whole project to github,

- into README: init_score auf classification und regression leaked welche werte drin sind wenn zB init score 1.125 dann wissen wir daten sind teilbar durch 8. da muss man ein wenig addieren. + PB anpassen

code issues (doable after submission)

- due to git issues, had to merge runtime_measurements into master. -> good anyways because of the graphs and lots I guess. However, need to use the non ugly src/evaluation etc. and no mape in loss.h etc. Fetch the version from before the runtime_measurements branch
- comparators in constant_time

Contents

Chapter 1

Introduction

1.1 Motivation

In recent years, cyber insurance has emerged as a new form of insurance. A customer seeking cyber insurance is usually looking for protection against a variety of cyber risks, that might include hacking incidents, data leaks, IT service outages, and the like. Asymmetry in information is one of the major problems in the insurance industry. As such, insurers have significantly less information about insured objects (e.g. a vehicle or a home) than their customers. This complicates risk assessment and insurance pricing. To counteract this, insurers would usually hand out questionnaires to their customers to gain more information about the underlying objects. In terms of cyber insurance however, this is not always feasible. Clients are generally reluctant to fully disclose details of their IT infrastructure, and security policies. Customers likely are concerned, that honest answers about poor IT-security practices might be used to discriminate against them, either at the time of insurance pricing or at a potential future claim settlement.

The recent advances in Trusted Execution Environments (TEEs) and privacy-preserving machine learning offer new avenues for dealing with the aforementioned problem of information asymmetry. A hardware-protected enclave (e.g. using Intel Software Guard Extensions (SGX)) could be leveraged to collect private customer questionnaires, and subsequently train a machine leaning model on the data in a privacy-preserving fashion. In order not to leak any secret information about the individual participants, techniques like Differential Privacy (DP) can be used. In this way, the insurance company is able to build a useful aggregate model, while protecting the privacy of each individual customer at the same time. According to recent studies, with such a privacy-preserving system in place, users would be more willing to provide private data to data collectors.

1.2 Problem overview

Continuing with the insurance use case, there are two main parties involved, each with different demands: (i) Insurance customers need data privacy under all circumstances. Even a potentially malicious insider at the insurance company must not be able to leak secret data. (ii) The insurance is looking for good results from the resulting model. Customers or other external parties should not be able to distort the output model or render it useless through targeted interaction with the enclave. There is a number of things that could go wrong, which would lead to these requirements being violated. First, even though the training process takes place inside a TEE, leakage of secret information can occur. It is a known issues that TEEs are susceptible to side-channel attacks.

Second, even if all side-channels during the DP-GBDT training are eliminated, developing a secure deployment strategy is a challenging task: How should data be moved into and out of the enclave in a safe manner? How can persistent state be added to the enclave in order to counteract forking/interleaving/rollback attacks? How can the system cope with data loss and other human or machine related faults?

Goals (i) *Efficient algorithm*: First of all, this requires building a standalone DP-GBDT implementation from scratch. The algorithm should be bug-free and able to produce accurate performance measurements. Ultimately, the algorithm should deliver decent results on a small privacy budget. In other words, the noise added through differential privacy should not completely erase the output model's expressive power. Additionally, even though training will likely not be performed very frequently, its runtime should be reasonable. (ii) *Side-channel resistance*: For secure execution inside the TEE the DP-GBDT implementation must be adequately hardened. The goal is to prevent all leakage through "digital side-channels". This notion sums up all side-channels that carry information over discrete bits. Examples are the memory access trace, control flow, and time. We do not need to get rid of side-channel leakage entirely. We must eliminate just enough leakage to achieve ε-differential privacy. (iii) *Secure deployment*: We further aim to explore the challenges of real-world secure deployment of the system. This includes: Privacy attacks through enclave state rollbacks or fork attacks, data collection and provisioning over an extended time period, ability to continuously improve the previous model with newly acquired data, resilience and fault tolerance in the case of human or machine errors.

1.3 Approach

Two prior works form the basis and starting point for this thesis. With DPBoost, Li et al. ([46], 2020) provide the theory behind DP-GBDT learning. Moreover, Théo Giovanna built a first Python implementation of said algorithm during the course of his master's thesis (submitted Feb. 2021). With the overlying goal of performing DP-GBDT inside an SGX enclave, a new C++ implementation had to be created. This lead to the discovery of several bugs and to substantial runtime improvements. The resulting code was afterwards ported into an SGX enclave. Subsequently, manual side-channel hardening on source code level was conducted. The process differs from traditional hardening to some extent: We do not try to eliminate every single bit of leakage. Instead, we remove just enough leakage at the right places to achieve ε-differential privacy. This is much more effective than tool assisted hardening as we leverage our knowledge of the DP-GBDT algorithm. Further, several data-oblivious and constant-time building blocks are created to replace common and reoccurring operations. We assume that the underlying hardware of the enclave setup is not compromised, patched, and works as expected. Denial of service attacks are also out of scope. The prediction accuracy of the algorithm is evaluated on four real-world UCI standard datasets. Moreover, we provide runtime measurements that show the impact of our hardening methods. Finally we explore possibilities for secure deployment of our enclave setup. With the help of secure monotonic counters, persistent state is achieved. We highlight the challenges and propose solutions for the enclave initialisation, data collection, training, enclave replication and migration.

Do we need some results teasers here? kari:yes

1.4 Contribution

The main contributions of this thesis are:

- Build a C++ DP-GBDT implementation that runs inside an SGX enclave
- Harden the implementation against side-channels
- Evaluate the accuracy and runtime of the algorithm
- Explore possibilities for secure deployment

There are further some smaller contributions that developed along the path: (i) Previous results and implementations of the DP-GBDT algorithm we are using turned out to have several flaws. To our knowledge, this is first time detailed and correct performance results of this algorithm have been generated. (ii) The underlying codebase was designed with a focus on usability and extensibility for future experimentation. (iii) This thesis offers some guidelines on manual (source code level) hardening of a tree based machine learning algorithm. (iv) Similarly, we offer guidance on how port such an algorithm into an enclave. What kind of code changes are necessary and how to divide code into inside/outside of the enclave.

Chapter 2

Background

2.1 Decision trees

Machine learning is generally a two-step process: There is a learning step and prediction step. During the learning phase, a model is built based on the provided training data. During the prediction phase, the previously obtained model is applied to new data in order to forecast the likelihood of a particular outcome. Decision Trees are among the easiest and most popular learning algorithms due to their illustrative nature. They can be used for solving regression and classification problems. This means it can predict both continuous/*numerical* values such as price, salary, etc. and also *discrete/categorical* values, such as gender or nationality. Consider the example in figure 2.1. Using this decision tree, we can categorize an upcoming day according to whether it will be suitable to go rowing. By using the weather outlook, temperature, and wind strength as attributes, the decision tree will return the corresponding classification result (in this case Yes or No).

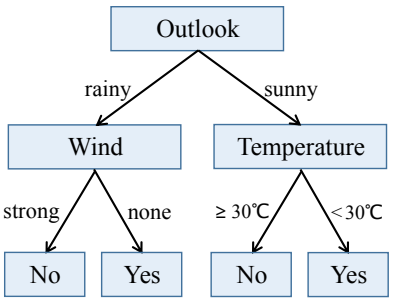

Figure 2.1: Example decision tree

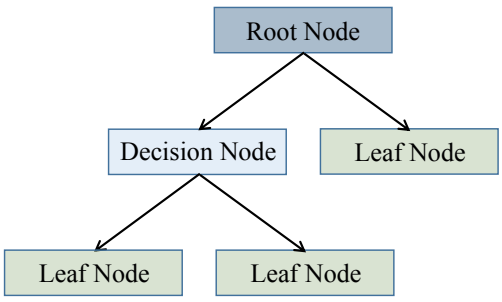

Figure 2.2: Decision tree terminology

Figure 2.2 provides an overview of the terminology related to decision trees. The *root node* is the node that starts the graph. It divides the samples into two parts using the best split that is available on the data. *Splitting* is the process of dividing a node into two (or more) sub-nodes. A *decision/internal node* is formed when a sub-node splits into two (or more) sub-nodes. Nodes where no further splitting takes place are called *leaf/terminal nodes* This is where the predictions of a category or a numerical value are made. In a tree, a node that is divided into sub-nodes is labeled a *parent node*, while its sub-nodes are called *child nodes*.

A decision tree is typically constructed by recursively splitting training samples using the features from the dataset that work best for the specific task at hand. This is done by evaluating certain metrics, like the the entropy or the *information gain*. There exist many variations of this recursive

building algorithm, such as ID3 [56], C4.5 [57] or CART [14].

Definition 2.1 (Information gain [12]) *Let $\boldsymbol{X} = X_1, \ldots, X_n$ be a set of inputs (training samples), where each X_i is of the form $(\boldsymbol{x}, y) = (x_1, \ldots, x_n, y)$. x_j is the value of the j^{th} attribute of $\boldsymbol{x}$, and y is its corresponding label. The information gain for an attribute j is given by:*

$$IG(\boldsymbol{X}, j) = H(\boldsymbol{X}) - H(\boldsymbol{X}|j)$$

Where $H(\cdot)$ is the Shannon entropy. In other words: Information gain or is a statistical property that measures how well a given attribute separates the training examples according to their target classification.

As already indicated, decision trees offer many advantages: They are simple to understand, interpret and visualize. Little to no data preprocessing required. And generally, they are widely applicable as they make no assumptions about the shape of the data. However, there are also some weaknesses:

- Decision-tree tend to create overly complex trees that fail to generalize the data well. This is called *overfitting*.
- Decision tree learners are inclined to create biased trees if some classes dominate [49]. It is thus advised to balance such datasets prior to decision tree fitting.
- Being a greedy algorithm, it does not guarantee to return the globally optimal model.
- It is possible for decision trees to be unstable due to small variations in the data, which can result in completely different trees being produced. Fortunately this effect can be reduced by methods like bagging and boosting.

Bagging and boosting *Ensemble learning* is a concept in which multiple models are trained using the same learning algorithm. Bagging and boosting are both examples of this technique. They combine multiple weak individual learners into one that achieves greater performance than a single learner would. This has been proven to yield better results on many machine learning problems [49]. *Bagging* refers to bootstrapping + aggregation, in which weak learners are trained on a random subset of data sampled with replacement (bootstrapping). Subsequently their predictions are aggregated. Bootstrapping assures independence and diversification as every subset is sampled separately. *Boosting* differs from the aforementioned approach in that it is a *sequential* ensemble method. Let's use decision trees as an illustration. Given a total of n trees, the individual models/trees are added in a sequential way. The second tree is added to improve the performance of the first tree, etc. In the end, the individual models are weighted and combined to a final strong classifier.

2.2 Gradient boosted decision trees

Gradient Boosted Decision Trees (GBDT) algorithms make use of decision trees as the base learner and add up the predictions of several trees. Gradually, new decision trees, that are based on the residual between ground truth and current predictions, are trained and added to the current ensemble. Today, there's a multitude of sophisticated and high-performance GBDT frameworks available, such as LightGBM [42] and XGBoost [19].

In the remainder of this section, we will dive into the formalities of the GBDT algorithm. Let l be a convex loss function and D is a dataset of n instances with d features. That is $X = X_1, .., X_n$,

where $X_i = (\mathbf{x}, y) = (x_1, .., x_d, y)\ \forall i \in [1, n]$. At the t^{th} iteration, GBDT minimizes the following objective function: [63]

$$O(\mathbf{x})^{(t)} = \sum_{i}^{n} \left(g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right) + \Omega(f_t) \tag{2.1}$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ is the first order gradient statistics of the loss function. f_t is the decision tree at the t^{th}, $\Omega(f_t) = \frac{1}{2}\lambda \|V\|^2$ is the regularisation term, V is the leaf weight, and λ is the regularisation parameter.

Trees grow from their roots to their maximum depth. Let I_L and I_R be the instances of the left and right subsets after a split. The gain of a split is given by: [**dpgbdt**]

$$G(I_L, I_R) = \frac{(\sum_{i \in I_L} g_i)^2}{|I_L| + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{|I_R| + \lambda} \tag{2.2}$$

By traversing all combinations of features and feature values, GBDT finds the split which maximizes the gain. If the current node cannot achieve the splitting requirements (e.g. if it's larger than the maximum tree depth, or all splits have a gain < 0), it becomes a leaf node. We define $I = I_L \cup I_R$. The optimal leaf value is then given by: [**dpgbdt**]

$$V(I) = -\eta \frac{\sum_{i \in I} g_i}{|I| + \lambda} \tag{2.3}$$

By applying a shrinkage/learning rate η to the leaf values, we can reduce the influence of each individual tree, allowing future trees to improve the model. The entire GBDT process is shown in algorithm 1:

Algorithm 1: GBDT training process

Input: $X = X_1, \ldots, X_n$: instances, $\mathbf{y} = y_1, \ldots, y_n$: labels
Input: λ: regularisation parameter, d_{max}: maximum depth, η: learning rate
Input: T: total number of trees, l: loss function
Output: An ensemble of trained decision trees.

```
for t = 1 to T do
    Update gradients of all training instances on loss l
    for depth = 1 to d_max do
        forall node in current depth do
            forall split value i do
                G_i ← (Σ_{i∈I_L} g_i)^2 / (|I_L|+λ) + (Σ_{i∈I_R} g_i)^2 / (|I_R|+λ)   ▷ Equation 2.2
            Split node on split value i = arg max_i(G_i)
    forall leaf node i do
        V_i ← −η Σ_{i∈I} g_i / (|I|+λ)                                                ▷ Equation 2.3
```

2.3 Differential privacy

Differential privacy (DP) is a mechanism for provable public sharing of information about a dataset as a whole, while keeping information about the individuals within it private. [30] The

key idea is the following: If a single substitution in the database has a small enough effect on the result of a query to that database, no information about an individual member can be gained. In other words: An attacker is assumed to have almost complete knowledge of the training data, and may only be uncertain about a single point in the training data. Using this, a defender can deny the existence of every single training data point. There are several interesting use cases for differentially private algorithms, such as publishing the results of a survey while ensuring the confidentiality of study participants.

Given a finite data universe $\mathcal{X}$, we define a dataset $D \in \mathcal{X}^n$ as an ordered tuple of n rows $(x_1, .., x_n) \in \mathcal{X}$. We say that two datasets $D, D' \in \mathcal{X}^n$ are *neighboring* if they differ only by a single row. This relationship is denoted by $D \sim D'$.

Definition 2.2 (ε-Differential Privacy [dpgbdt]) *Let ε be a positive real number and f be a randomised function. Function f is said to provide ε-differential privacy if, for any two neighboring datasets $D \sim D'$ and every output O of function f:*

$$Pr[f(D) \in O] \leq e^{\varepsilon} \cdot Pr[f(D') \in O] \tag{2.4}$$

The parameter ε is also called *privacy budget*, and it controls the privacy guarantee level of function f. A smaller ε represents stronger privacy. In practice, ε is usually chosen <1, such as 0.1 or $\ln 2$.

Sensitivity is a parameter determining how much perturbation is required in the differential privacy mechanisms. We say a function $f\colon \mathcal{X}^n \to \mathbb{R}^m$ has sensitivity Δ if for all neighboring $D, D' \in \mathcal{X}^n$ it holds that $||f(D) - f(D')||_1 \leq \Delta$.

To achieve differential privacy, two popular mechanisms exist: the *Laplace mechanism* and the *exponential mechanism*. Numeric queries can be performed by the former, and non-numeric queries by the latter. In the Laplace mechanism, controlled noise is added to the query result before it is returned to the user. [31] The noise is sampled from the Laplace distribution centered at 0 and scaled by factor b.

Definition 2.3 (Laplace Distribution) *A random variable has a $Lap(\mu, b)$ distribution if its probability density function is:*

$$f(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right) \tag{2.5}$$

Theorem 2.4 (Laplace Mechanism [dpgbdt]) *Let $f : \mathcal{D} \to \mathcal{R}^d$ be a function. The Laplace mechanism F is defined as:*

$$F(D) = f(D) + Lap(0, \Delta f/\varepsilon) \tag{2.6}$$

Where the noise $Lap(0, \Delta f/\varepsilon)$ is drawn from a Laplace distribution with mean $\mu = 0$ and scale $b = \Delta f/\varepsilon$. Then F provides ε-differential privacy.

Theorem 2.5 (Exponential Mechanism [dpgbdt]) *Let $u : (\mathcal{D} \times \mathcal{R}) \to \mathbb{R}$ be a utility function, with sensitivity Δu. The exponential mechanism F is defined as:*

$$F(D, u) = \textit{choose } r \in \mathcal{R} \textit{ with probability} \propto \exp\left(\frac{\varepsilon u(D, r)}{2\Delta u}\right) \tag{2.7}$$

Then F provides ε-differential privacy.

As both of the previous mechanisms only offer privacy guarantees for single functions, two composition theorems exist. Using *sequential composition*, a sequence of differentially private computations can be shown to be private. The privacy budgets can be added up for each randomized mechanism.

Theorem 2.6 (Sequential Composition) *Let $f = \{f_1, .., f_m\}$ be a series of functions performed sequentially on a dataset. If f_i provides ε_i-differential privacy, then f provides $\sum_i^m \varepsilon_i$-differential privacy.*

the *parallel composition* theorem can be applied in a case where f is applied to disjoint subsets of a dataset. The largest privacy budget is then decisive for the final result.

Theorem 2.7 (Parallel Composition) *Let $f = \{f_1, .., f_m\}$ be a series of functions performed separately on disjoint subsets of a dataset. If f_i provides ε_i-differential privacy, then f provides $\{\max(\varepsilon_1, .., \varepsilon_m)\}$-differential privacy.*

2.4 Differentially private gradient boosted decision trees

This section finally combines the different notions and formulas from the previous background sections and puts them to use. Literature ([**dpgbdt**, 71]) proposes two adjustments to algorithm 1 for creating differentially private decision trees. (i) Noise has to be added to each leaf node's value in the decision tree. (ii) To choose the attribute and its value upon which a decision node is split, the exponential mechanism must be used. By ranking the different different splits according to their information gain, the the exponential mechanism can choose a split with a proportional probability.

2.4.1 DPBoost

The remainder of this section summarizes DPBoost [**dpgbdt**], Li et al.'s work on DP-GBDT, which set the theoretical foundation for this thesis. Li et al.'s goal was to improve model accuracy of previous DP-GBDT work [1, 48, 67]. In their work, several optimizations have been made, such as tighter sensitivity bounds for the leaf nodes and the splitting function, and more effective privacy budget allocation. This results in better accuracy of the algorithm. The following three paragraphs elaborate on the paper's new techniques, before the final algorithm is presented in Algorithm 2.

Gradient-based data filtering Gradient-based data filtering (GDF) is a technique that DPBoost uses in order to obtain a tighter sensitivity bounds. Let $g_l^* = \max_{y_p \in [-1,1]} ||\frac{\partial l(y_p, y)}{\partial y}|_{y=0}||$. At the beginning of each iteration, the instances that have 1-norm gradient larger than g_l^* are filtered out. Only the remaining instances are used to build a new differentially private decision tree in this iteration. Using this, we can bound the sensitivities of G (equation 2.2) and V (equation 2.3): By applying GDF in the training of GBDTs, we get $\Delta G \leq \frac{3\lambda+2}{(\lambda+1)(\lambda+2)} g_l^{*2}$ and $\Delta V \leq \frac{g_l^*}{1+\lambda}$. The derivations can be found in DPBoost [**dpgbdt**], chapter 3.

Geometric leaf clipping All trees have the same sensitivities with GDF. During the training process, the gradients tend to decrease with each iteration. Therefore, as the number of iterations increases, one could derive an even tighter sensitivity bound. Based on this, a geometric leaf clipping mechanism was developed, that can be conducted before the Laplace mechanism is applied to the leaf values. In combination with GDF the following bounds hold:

$\Delta V \leq \min(\frac{g_l^*}{1+\lambda}, 2g_l^*(1-\eta)^{t-1})$ and $\Delta G \leq 3g_l^{*2}$, where η is the learning rate, λ is a regularisation parameter and t is the index current tree that is built.

Adaptive privacy budget allocation Specifically, they suggest allocating half of the privacy budget for the leaf nodes (denoted as ε_{leaf}). The remaining budget is equally divided to each depth of the internal nodes (ε_{node} per level). By Theorem 2.7, because the inputs of one depth level are disjoint, the privacy budget consumption of only one depth has to be accounted for. Therefore $\varepsilon_{leaf} + \varepsilon_{node} * d_{max} = \varepsilon_t$ holds.

Algorithm 2: Differentially private GBDT training process [**dpgbdt**]

Input: instances $X = X_1, .., X_n$, labels $y = y_1, .., y_n$
Input: λ: regularisation parameter, d_{max}: maximum depth, η: learning rate
Input: T: total number of trees, l: loss function, ε: privacy budget
Output: An ensemble of trained differentially private decision trees.

$\varepsilon_t = \varepsilon$ ▷ Disjoint training subsets → apply Theorem 2.7
for $t = 1$ **to** T **do**
  Update gradients of all training instances on loss l
  $\varepsilon_{leaf} = \frac{\varepsilon_t}{2}$, $\varepsilon_{node} = \frac{\varepsilon_t}{2d_{max}}$
  for depth $= 1$ **to** d_{max} **do**
    forall node **in** current depth **do**
      forall split value i **do**
        $G_i \leftarrow \frac{(\sum_{i \in I_L} g_i)^2}{|I_L|+\lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{|I_R|+\lambda}$ ▷ Equation 2.2
        $P_i \leftarrow \exp(\frac{\varepsilon_{node} \cdot G_i}{2\Delta G})$ ▷ Theorem 2.5
      Split node on split value i, where i is chosen with probability $P_i / \sum_j P_j$
  forall leaf node i **do**
    $V_i \leftarrow \eta \left(-\frac{\sum_{i \in I} g_i}{|I|+\lambda} + Lap(0, \Delta V / \varepsilon_{leaf}) \right)$ ▷ Equation 2.3 and Theorem 2.4

2.5 Intel SGX

TEEs are a set of architectural extensions that provide software with strong security guarantees even when it is exposed to powerful adversaries. Intel SGX [38, 21] provides secure execution capabilities to Intel x86 mainframe CPUs. Through SGX, secure and verifiable execution environments known as *enclaves* can be used by applications to execute securely on the processor. SGX is contained inside the CPU package. Hence, its security properties are strongly tied to the processors hardware implementation. All other components of the system (including OS, firmware, and memory hardware) are removed from the Trusted Computing Base (TCB). This is achieved by ensuring that both processor and memory states of an enclave are only exposed to the enclave internal code. Protection from software access and physical attacks on enclave memory is provided by a hardware unit inside the CPU, called the memory encryption engine (MEE). Through transparent decryption and encryption, the MEE ensures that secret data is only stored in plaintext inside the processor. Furthermore, SGX provides remote attestation mechanisms, through which enclaves can prove to remote parties that they have been correctly initialized on a genuine (and therefore presumed secure) Intel processor.

The rest of this section elaborates on several SGX mechanics that are of particular relevance for this thesis. We cover (remote) attestation, data sealing and the use of secure monotonic counters.

Remote attestation Enclaves are no secure software modules by themselves. Rather they are software modules that can execute in a TEE. Due to the fact that enclaves can't be debugged or monitored at runtime, their code has to be verified by users before execution. However, as the untrusted system controls enclave initialization, an adequate mechanism is required. Through *attestation*, a specific software can demonstrate its trustworthiness to an external party in terms of authenticity and integrity. [21] There are two types of enclave attestation supported by SGX: local and remote. During local attestation, a second special enclave must be present on the same platform as the attestant. Remote attestation in turn can be performed with any software running on any external platform. During the remote attestation procedure, the CPU creates a measurement for the attested enclave that uniquely identifies it. A special (Quoting Enclave[1]) then signs this information, providing an attestation signature. In order to achieve secure communication between these two enclaves, local attestation is performed. Once the attestation signature has been received by the remote party, it will forward it to the Intel Attestation Service (IAS), which verifies its validity. This way, the remote party can determine that (i) no tampering occurred (ii) the attested software is running within an genuine hardware-assisted SGX enclave.

Sealing Programs in practice likely contain some secrets that need to be preserved during an event where an enclave is destroyed, a computer crashes etc. SGX therefore enables enclaves to have encrypted and authenticated persistent storage [21]. Each SGX-enabled CPU contains a randomly generated root seal key, that was fused into the hardware during production and is not stored by Intel. From this key, an enclave can derive a sealing key that can be used to encrypt enclave data. The resulting blobs can be passed to the operating system for long-term storage. There are two options when sealing data: (i) Seal to current enclave, which means only an enclave with the same exact same measurements will be able to generate the key to unseal the data. (ii) Seal to the enclave author. This means that the sealed data can be accessed by different versions of the same enclave and also by other enclaves belonging to the same vendor.

Secure monotonic counters TEEs needs protection against rollback attacks. Imagine a scenario where an adversary manipulates persistently stored data by replaying old data or by interleaving output from multiple enclave instances. To mitigate these kind of attacks, Secure Monotonic Counters (SMCs) has been introduced with Intel SGX SDK 1.8 [37]. SMCs leverage non-volatile memory storage locations that survive events like a power loss. There are essentially two operations that can be performed on such a counter after it was initialized, `Read()` and `Incr()`. The counters state and its communication with an enclave are cryptographically secured. These counters can be leveraged for versioning of sealed data: Whenever data is sealed, the corresponding SMC value is included. After e.g. a reboot, the data is restored. By extracting the stored counter value and comparing it to the obtained from rading the SMC. Since both the sealed data and the SMC are cryptographically protected, tampering in form of e.g. rollback attacks can be detected. There are a drawbacks to the current SMC implementations though [51]: (i) Incrementing/writing to the counter values is comparably slow (80-250 ms), which can limit its use in high-throughput applications. (ii) The non-volatile memory used by the counters wear out after approximately one million write operations.

Side-channel susceptibility The small TCB of SGX has its advantages. Scaling performance with the processor's capability, for example. This comes at a cost, however. Due to the simplicity of the design, enclaves must rely on shared, untrusted resources like memory, I/O, etc. As

[1] The Quoting Enclave is an Intel-developed secure enclave, which has access to the hardware attestation key required for signing the measurement.

a result, enclaves are susceptible to side-channel attacks. This includes for example timing attacks, cache-attacks, speculative execution attacks, branch prediction attacks, microarchitectural data sampling and software-based fault injection attacks. [55] Intel does acknowledges that "SGX does not defend against this adversary"arguing that “ and further states that "preventing side-channel attacks is a matter for the enclave developer” [36]. There are two general types of SGX side-channel mitigation approaches. (i) One approach is to harden the programs by rewriting the code at compile time and randomizing at execution time. Consequently, monitoring side-channels is much more complicated for the adversary. These measures however are often susceptible to more versatile future attack variants [66, 17, 52]. (ii) An alternative approach is to manually eliminate secret-dependent data paths entirely. eliminate secret-dependent code and data paths altogether. This method can be very effective at times, but there are also many pitfalls for a developer to fall into. [52, 62, 17]. Either way, SGX developers are left with the major task of evaluating the security implications and figuring out practical methods to mitigate the side-channels hazards.

2.6 Side channels

This section offers a very brief introduction to side-channels. We start with an illustrative example. Subsequently we define the notion of digital side channels, which will be important for later chapters.

Consider Algorithm 3 as an example. It depicts the binary version of the classical square-and-multiply algorithm, which can be used to perform exponentiation. Variations of it can be found in old implementations of RSA encryption and decryption. Actually, in efficient RSA implementations, all of the modular multiplications and squaring operations are performed using an algorithm of it’s own, called Montgomery Multiplication [53]. Which was itself target to several side-channel attacks [60]. But to keep things simple, we only focus on the actual square-and-multiply function. The problem lies in the branch on line 4. If the branch is taken, an additional modular multiplication is performed. Through this side-channel, the value of the secret d can be inferred one bit at a time [15].

Algorithm 3: Binary Square-and-Multiply

Input: M, d, N
Output: $M^d \mod N$, where d is an n-bit number $d = (d_0, .., d_{n-1})$

```
1 S = M
2 for i from 1 to n − 1 do
3 |   S = S ∗ S mod N
4 |   if d_i = 1 then
5 |   |   S = S ∗ M mod N
6 return S
```

The code in Algorithm 3 has in fact been exploited with use of a variety of other side-channels, such as power [43], cache [70], branch prediction [2] and even sound [33]. Accordingly many defense techniques have been proposed.

Digital side-channels For the remainder of this thesis we a notion to classify the variety of possible side-channels. We therefore adopt the term of *digital side channels*, as proposed in the work of Rane et al. [58]. Digital side-channels are side-channels that carry information over

discrete bits. An adversarial observer can detect them at the level of either program state or the ISA (instruction set architecture). Therefore, address traces, cache usage, and data size are all examples of digital side-channels. In contrast to elements like power draw, electromagnetic radiation and heat, which are not part of this group.

Chapter 3

Overview

As already indicated, the starting point of this thesis was **not** zero. The DPBoost paper ([46], 2020) provided the theory behind DP-GBDT learning. Moreover, Théo Giovanna built a first Python implementation[1] of said algorithm during the course of his master's thesis (submitted Feb. 2021). Starting from this point, the high level objectives were in short: Porting the code to C++, putting it into an enclave, harden it against side-channels, come up with a realistic strategy for deployment in practice.

To acquire an understanding of the problem we will first give a high level overview of the desired solution for the insurance use case. Subsequently, the adversary model is presented. At last, we outline the goals and requirements our system should meet.

3.1 Solution overview

A visual representation of the setup is given in figure 3.1. Prior to actual operation, the insurance has deployed the DP-GBDT enclave on its servers. The enclave must be reachable by external insurance customers. From there on, the insurance starts selling their cyber insurance policies to companies. In return, customers send back filled out questionnaires about their cyber security policies. They can use a simple client application for this. The client application conducts the survey and sends the encrypted answers directly to the enclave. The questionnaires arrive at the enclave one at a time. Upon reception the enclave saves the acquired data to disk. This continues until enough samples are collected to get a meaningful result from running the DP-GBDT algorithm. At this point the insurance notifies the enclave to initiate the first training. Upon completion the enclave returns the output model to the insurance. The enclave continues collecting samples from customers. And again, once enough samples are collected the insurance can instruct the enclave to train on the newly received data. The corresponding output can be used to improve the previous model. Further, if at any point some part of the model is lost, the insurance can instruct the enclave to recreate the missing part of the model.

3.2 Threat model

Next, we discuss the threat model, which is relevant for side-channel hardening and secure deployment of the system (chapters 4 and 7). We use **two different threat models** to cover the two main security goals. To ensure customer data privacy, which is our main priority, a

[1] https://github.com/giovannt0/dpgbdt

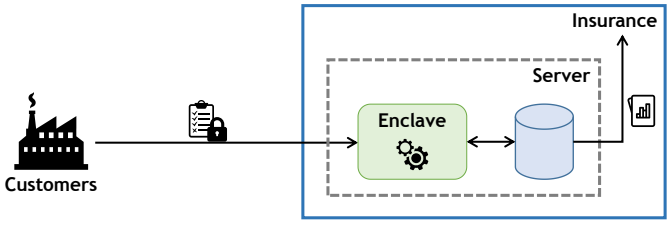

Figure 3.1: Insurance use case overview

very powerful adversary was chosen. To address certain types of denial of service attacks by external customers, we use a weaker adversary model. It is clear that denial of service attacks can occur at virtually any point of an enclave setup. Most of them (e.g. network flooding) are out of scope. However, some cases would inevitably need to be handled in practice. In particular, a malicious external customer must not be able to disrupt the whole setup by crafting and submitting malicious messages to the enclave.

TM1, Goal: Customers want privacy even if the insurance is malicious TM1 represents a malicious service provider. This adversary has full control over both the software (including OS or hypervisor) and the hardware of his system. Thus he is able to start or stop processes at any point, set their affinity and modify them at runtime. He can also read and write to any memory region except the enclave memory, map any virtual page to any physical one and dynamically re-map pages. He can run multiple instances of an enclave in parallel Moreover he has various network control capabilities: He may tamper with, delete, delay, reorder, or replay any network packets exchanged between the server and the participating parties. We also expect him to closely monitor the program execution for side-channels. Lastly, he has full access to state-of-the-art enclave attack frameworks like SGX-Step [66]. The goal of the malicious service provider is to gather as much information as possible about customers participating in the training process. Note that, from a customers perspective, service provider and insurance might very well be the same entity. Or similarly, the two might collude to steal secrets from the insurance customers.

TM2, Goal: Insurance wants good results even if customers are malicious TM2 assumes existence of a legitimate external customer with malicious intentions. We assume that such a customer does not have access to the servers where the enclave resides. Put differently, The enclave is located in a secure area at the insurance company. However, the malicious customer does have full control over his questionnaire data that is sent to the enclave. Further capabilities include: initiating multiple session with the DP-GBDT enclave or sending his questionnaire multiple times (but not to the extend where the sheer number of messages leads to denial of service). The objective of a malicious customer is to decreases the quality of the overall result, or even wipe out previous progress of the learning process entirely.

Out of scope The following attack vectors are considered out of scope for both TM1 and TM2: We assume that the underlying hardware of the enclave setup is not compromised. Further, all known SGX vulnerabilities with a fix available are patched. The TEE hardware must work as expected and provide an isolated execution environment with remote attestation capabilities for any code running inside. Enclave-provisioned secrets are not accessible from untrusted code, etc. As previously indicated, traditional denial of service attacks are also out of scope. Thus, concerns like the network connection being cut, or the OS not scheduling our code are ignored.

3.3 Goals

Efficient algorithm The new C++ implementation has to meet multiple requirements. First of all and contrary to it's Python predecessor it should be free of bugs. From this point, it should still be able to produce usable results for reasonably small privacy budgets. In other words, the noise added through differential privacy should not completely erase the output model's expressive power. Additionally, the algorithm runtime should be reasonable. Even though training will likely not be performed very frequently.

Side-channel hardening As SGX enclaves are vulnerable to side-channels, the implementation must be adequately hardened. But the hardening process differs from traditional hardening approaches to some extent: First of all, the whole algorithm is quite large and complex. Usually hardening is applied in a smaller frame, for instance to an AES encryption function in a cryptographic library. Second, we are not trying to eliminate every single bit of leakage. We eliminate just enough to achieve ε-differential privacy. For example, in our algorithm the adversary is allowed to learn the final model. For information that must be kept secret in turn, we eliminate all leakage through *digital side-channels*. As already introduced earlier (ref TODO), this sums up all side-channels that carry information over discrete bits. Examples are address traces, cache usage, and data size. In terms of secret dependant memory/cache accesses we decided against specifying a fixed acceptable leakage granularity (e.g. cache line). Admittedly, such an assumption would make the hardening process easier and execution times faster. However, the result would be prone to be broken again once the next better attack vector is discovered. To put it differently and to give an example: In order to retrieve a value at a secret dependent index from an array, every single element should be touched.

Secure deployment Even the most highly performing and privacy-preserving algorithm is useless if it can't be applied to a real-world system. Apart from side-channels, there are several other attack vectors that must be considered for secure deployment. Enclave state rollback or fork attacks, for instance. Additionally, we should be able to deal with potential technical failures due to either human negligence or machine errors. Our goal is therefore to highlight the key problems and present viable new or existing solutions. This includes:

- Data collection over an extended time period
- Training on newly acquired customer data to improve the existing model output[2]
- Training on old data after output model loss[2]
- Enclave replication and migration for maintaining persistent state

[2] This is not trivial whilst not violating differential privacy

Chapter 4

Algorithm implementation

This chapter gives a broad overview of the design and implementation work that was carried out for this thesis. The aim is to explain the key details and features in order to facilitate future work that builds on this thesis. Readers may also skip this chapter if this is not part of their objective. The topics discussed include:

- Starting point
- Design decisions and implementation features
- Discovered bugs and fixes
- Transferring the code into an SGX enclave

Be aware that the design and implementation parts are not strictly separated. This facilitates justifying certain design choices that were made along the way. There are also some remarks about things that did **not** work out as expected.

4.1 Starting point

With DPBoost ([46], 2020) Li et al. set the cornerstone behind DP-GBDT learning. Apart from some slight changes (described in TODO ref background) this is the algorithm that is used in our implementation. The results of their work are promising, yet should be taken with a grain of salt. For instance Moritz Kirschte found a bug in XY calculations TODO. Further, the results of some plots in their evaluation section are not really helpful and apparent. can I say that? Li et al. do provide source code for their implementation. They are essentially using the LightGBM [42] framework but with several modifications scattered throughout a number of files. As this is a bit convoluted, Théo Giovanna subsequently build a first standalone Python implementation[1] of said algorithm during the course of his master's thesis (submitted Feb. 2021). Unfortunately, this implementation has weaknesses as well, most notably speed. It's just a bit too slow for effective testing and tuning. With the overlying goal of performing DP-GBDT in an enclave, a decision had to be made: Do we either use some unofficial framework to squeeze the existing Python code into an enclave, or, do we use the Python code as reference and rebuild the algorithm in C/C++?

4.2 C++ DP-GBDT - Design and implementation

In hindsight, this was a good decision. It lead to the discovery of several bugs, to substantial runtime improvements, and enabled an easier transfer to SGX. Even though there do exist some

[1] https://github.com/giovannt0/dpgbdt

machine learning libraries for C++, we decided against using one. Mainly because it would complicate the upcoming hardening process. It would have been difficult to analyze and assure the absence of side-channels. As a result, many tasks that only require a single line of code in Python had to be replicated in C++. Performing cross validation is one such example. The only library that was utilized in the algorithm is the C++ standard library [40]. The c++11 standard was chosen as it is the latest with SGX compatibility.

Project structure The project[2] essentially consists of five DP-GBDT versions (figure 4.1) and some additional infrastructure to run/verify/evaluate/... them. This separation is a big advantage, since both hardening and running a program inside an enclave tend to clutter the underlying code to some extent. The outcome of running the different variants is the same: equal input means equal output. Details on running and testing them can all be found in the public repository.

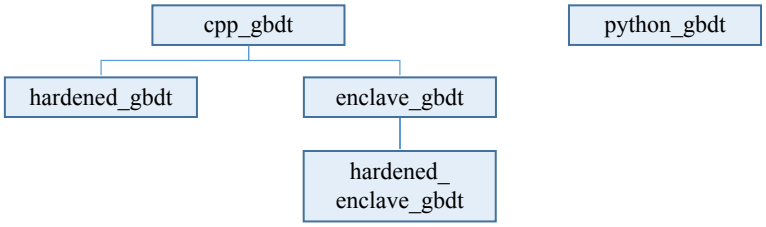

Figure 4.1: Project structure

For the remainder of this section we are mainly referring to `cpp_gbdt`, the non-hardened C++ version. It is roughly 2500 lines of code in total. The algorithm is clearly visible and compiler optimizations and threads lead to solid runtime performance. This version is ideal for experimentation with the underlying algorithm.

Feature selection With future hardening work in mind, it was vital to get a fully functional and correct prototype. To achieve this in the available time frame some compromises had to be made. These came in the form of omitting certain algorithmic options and features that were presented in either the DPBoost paper or in Théo's thesis. The following features are not available:

- Best-leaf first decision tree induction
- "2-nodes" decision tree induction algorithm
- Sequential composition of multiple ensembles
- Generation and use of synthetic insurance datasets
- Multiclass classification

This is of course unfortunate, however, if required they can always be incorporated into the current implementations with relatively small effort.

Datasets and parsing Currently only regression and binary classification prediction tasks are supported. To demonstrate this four datasets are available and ready to use in the project: Abalone [26], YearMSD [27], Adult [28] and BCW [29]. To our knowledge, this is first time detailed and correct DP-GBDT results have been generated for these datasets. This should be a valuable foundation for future experiments and tuning of the algorithm. Further, it is very easy to add other datasets: Given a new dataset (in comma separated form), only about 15 lines of code

[2] https://github.com/loretanr/dp-gbdt/blob/main/TODO.todo

in the `Parser` class are required to start using it. This is just for specifying how the new dataset looks (size, target feature, feature type, regression/classification etc). There's an example of this in appendix A.

Comparability and verifiability This was one of the main challenges during development. How can you ensure that the results are accurate? Even though a reference implementation was available for comparison, there were multiple obstacles:

- Bugs in the reference implementation
- The DP-GBDT algorithm being full of randomness
- The python implementation being painfully slow
- Numerical issues caused by the use of floating point numbers

Let's take a closer look at some of these points. Naturally, being the first of its kind, the reference implementation contained multiple bugs, that were only gradually discovered. It ranges from programming slips to relatively important parts of the algorithm that were misinterpreted. Next, regarding randomness, Python and C++ libraries use different sources and mechanisms for random number generation. This obviously obstructs comparison of Python and C++ DP-GBDT results. Unfortunately, it was not as simple as setting the same seed in both implementations. The problem was solved by enabling deactivation of every bit of randomness. This includes the exponential mechanism, shuffling of the dataset, Laplace Mechanism for leaf clipping, random selection of samples for the next tree and more. Last but not least, a word on numerical details, which were the most aggravating kind of bugs to deal with. Imagine that at some point in the algorithm you need to compute a weighted sum $\sum_{i \in \mathbb{N}} w \cdot a_i$. Being a performance conscious programmer, you unsuspectingly factor out w. You will not notice any problem until you later run the algorithm on some particularly large dataset. In the 4^{th} cross validation fold, from decision tree number 34 onwards, the Python and C++ trees are suddenly not matching anymore. With a few hairs turning gray in the process, you track the difference down to this exact weighted sum. Somewhere deep in the Python sklearn library, this sum was calculated without factoring out w. A friendly reminder that neither addition nor multiplication is associative with IEEE 743 double precision (64-bit) numbers [35]. Because of the nature of the algorithm, there are multiple spots where computation is done on the same values over and over again. For example: all samples have gradients that are constantly updated whenever a tree is built and added to the ensemble. Tiny numerical differences in those gradients can build up to the point where they affect the shape of a new tree. For instance, when searching the next split in an internal node, slightly different gradients lead to a different gain being computed which can result in a different split being chosen. This means different samples continue to the left/right children of that node. As soon as such things happen, the trees in an ensemble will eventually start to look completely different. Since these kind of issues are absolutely ubiquitous in DP-GBDT, the only viable solution was to regularly clip prone values to about 12 decimals.

However, given these hacks, it was possible to eliminate the differences between the python and C++ algorithm. To keep it that way and make future development easier, a verification framework was created. In its core it's a bash script that (compiles) and runs both the Python and (un/hardened) C++ code on a number of different datasets. During execution of the algorithms it will continuously compare intermediate values of the implementations. This way you do not only quickly recognize if a programming mistake was made, but you can also get the exact location where the two implementations started diverging. It will give you convenient and nicely formatted output, an example is shown in appendix A.

Logging and documentation We use fmtlib[3] for logging. You can choose between different logging levels to get output of the desired degree. You can further print finished trees to the terminal. Doxygen[4] was used to generate useful documentation of the algorithm, that will help the programmer quickly getting an overview of the different components of the algorithm.

Runtime improvements Experimentation with a slow implementation is frustrating. The GBDT algorithm has around 20 hyperparameters that need to be optimized to achieve good results (see appendix A for details). Being a compiled language, C++ was naturally already quite a bit faster. There was still much to be gained however. In order to determine the right spots to steer our attention to, frequent profiling was done using Intel vTune[5] and Intel Advisor[6]. For illustration purposes you can find a snippet of the profiling output of the finished (optimized) C++ GBDT algorithm in appendix A. It will also give you an idea of where the current bottlenecks are. Carried out optimizations include multithreading, aggressive compiler optimizations, removing blockers in core/critical functions to incentivise vectorization. The actual performance gains achieved, and put into relation to the other implementation are depicted in chapter 6.3.

4.3 Discovered bugs and fixes

This section summarizes all bugs that were discovered while working on this thesis.

Python bugs List of significant issues encountered (decreasing severity):

- Illegal tree rejection mechanism → big performance boost for small privacy budgets
- Bugs in the parser → performance loss
- `leaf_clipping` never being enabled
- `use_decay` formula wrong

The idea behind tree rejection was the following: In normal GBDT (no differential privacy) the predictions get closer to their actual value with every single tree. However, due to the randomness introduced by DP, this is not always the case for DP-GBDT. In the latter, some of the trees might actually make the current prediction worse. Therefore the author of the Python implementation decided to set aside some samples, that are used each time after a tree is created, to judge whether it's a good/useful tree. Bad trees are rejected accordingly and will not be part of the final model. This rejection mechanism and the reuse of samples of course violates differential privacy. However, as the performance improvements are quite drastic, it may actually be worth paying privacy budget for. This was discussed with E. Mohammadi and M. Kirschte, who will likely explore this idea further in the future. Figure 4.2 shows the drastic difference between the (illegal) usage of the rejection mechanism and the correct way of utilizing all trees. On average around 15 out of 50 trees were rejected. This goes to show how seemingly small tweaks and bugs in the algorithm can have a huge impact on the results and lead to deceptively good results.

All mentioned bugs were of course corrected in the new C++ algorithm. For the sake of completeness this project also includes a fixed version of the Python code.

DPBoost bugs Technically the following point is not a bug, but rather an important aspect that was not addressed by the paper: When trying out different splits to find the one with the

[3] https://github.com/fmtlib/fmt
[4] https://github.com/doxygen/doxygen
[5] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
[6] https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html

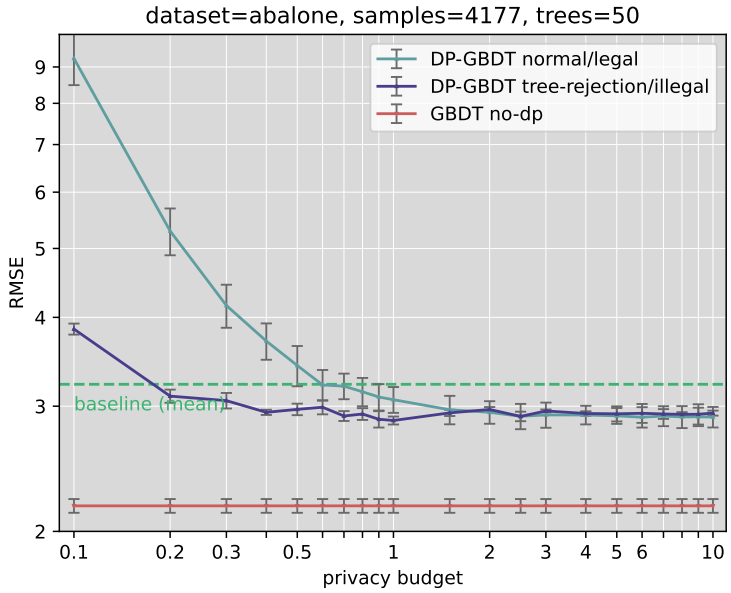

Figure 4.2: Comparison: Tree-rejection vs correct DP-GBDT

highest gain, iterating over the real numerical feature values is a bad idea for the following reason: When an adversary learns the final model (which is an assumption of our differential privacy proof), he can infer what feature values are actually present in the dataset. This problem has to be addressed by either adding enough noise, or by using a grid of fixed-size intervals.

Grid usage Given a fine enough grid spacing, the same splits will be found as if you were iterating over the real feature values. This is depicted in figure 4.3. The key observation is that splits only change whenever an existing feature values is passed.

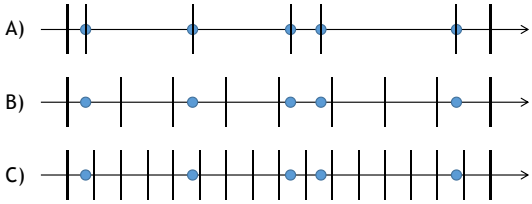

Figure 4.3: Grid step size: A) split on feature values, B) not fine enough, C) catches all splits

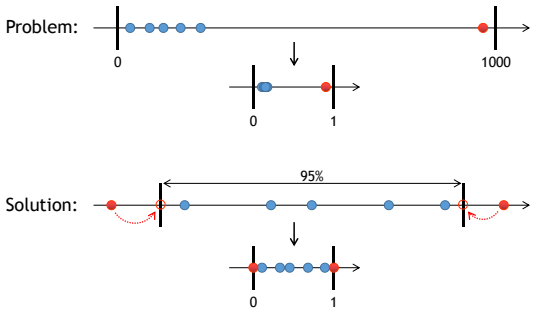

Figure 4.4: Scaling feature values into the grid using 95% confidence interval

Certain datasets however, contain numerical features with values of very different magnitudes. This creates a problem: Imagine a feature *A* with values in [0,1] and feature *B* with values in [0,1000]. You can certainly define a grid that is fine enough for feature A, while simultaneously accommodating all feature values present in B. However that would be extremely inefficient. A natural solution would be scaling the feature values into the grid. If the numerical feature values of the entire matrix X are in [0,1], we could choose our grid borders accordingly and

not waste as much computation power. This comes with another problem though: A malicious entity M, e.g. an insurance customer or even a malicious service provider, could submit bogus questionnaire data: If M chooses a value of 100'000 for feature A, scaling [0,100'000] into [0,1] would cause the legitimate participants' values to lie extremely close to each other. Possibly even to the point where the grid is not fine enough anymore to consider all possible splits. This way an attacker could essentially eliminate certain features from having any effect in the training. This is illustrated in figure 4.4.

One solution to this problem is the use of confidence intervals. A common percentile to choose is 95%. The idea is essentially to clip any outliers to the chosen percentile. In the case of a 95% percentile, the 2.5% and the 97.5% border of the values need to be determined. Subsequently the outliers above the upper boundary and below the lower boundary are clipped to their respective boundary. The challenging part is doing this in a differentially private fashion. Fortunately this paper [23] proposes multiple options on how to achieve this as well as corresponding R code on Github[7]. For this project the EXPQ method was adopted from the aforementioned work. According to the paper, it's not the most efficient scheme in terms of privacy budget, but it was the simplest one to understand and port to C++. The price for this is of course a small loss of information in your training data, as well as a small amount of privacy budget. More details on this in section 6.2.

In the final implementation you have both options. If you know enough about the values in your dataset, i.e. that your data values are going to be in a certain range, You can manually set constant grid borders, and thus not spend any privacy budget. Or, you enable the feature scaling and enjoy a shorter execution time. Of course, the grid usage can also be entirely turned off for experimentation in a non-real-world deployment setting.

4.4 Transferring DP-GBDT into an SGX enclave

This section outlines the steps required to move the DP-GBDT algorithm into an enclave. At this point, the details of attestation, encryption of customer data etc. are ignored (this will happen in chapter 7). The main goal is to run the core algorithm in the enclave, as a proof of concept and for execution time measurements.

After setting up SGX SDK 2.14 according to docs, we started from the included C++ sample enclave, and gradually replaced sections with our own code. As mentioned earlier, the C++11 standard was chosen for enclave compatibility. Albeit now, at the time of writing (October 21), support for C++14 was added to the SGX SDK [37]. An important step in SGX application development is dividing code into outside/inside enclave execution. `App.cpp` contains the outside code, while `Enclave.cpp` contains the internal code, i.e., the actual DP-GBDT algorithm. The basis of this separation is specified in the `Enclave.edl` file. It defines all function calls that are allowed to cross the enclave border. In our case (see listing 4.1) that means 4 calls in total: Three ecalls that go into the enclave, each of them is called exactly once. Two of them bring the dataset resp. the model parameters into the enclave. The other one does not carry any data and just initiates the training. The only ocall and thus way to get data out of the enclave is the ocall_print_string, which is used to print the final cross validation performance. This is sufficient to see that the algorithm works.

[7] https://github.com/wxindu/dp-conf-int

```
untrusted {
  ocall_print_string([in, string] const char *str);
};

trusted {
  ecall_load_dataset_into_enclave([in] struct sgx_dataset *dset);
  ecall_load_modelparams_into_enclave([in] struct sgx_modelparams *mparams);
  ecall_start_gbdt();
};
```

Listing 4.1: Enclave.edl entries

Parsing the dataset(s) requires various I/O functions that are not available inside the enclave. So the parsing (as well as defining the model parameters) is easiest to be taken care of on the outside. Since the interface between outside/inside of the enclave does not allow passing custom datatypes (not even C++ vectors), the dataset and model parameters have to be converted to C-style arrays. Once passed to the inside, they are converted back to C++ data structures. This procedure is illustrated in figure 4.5

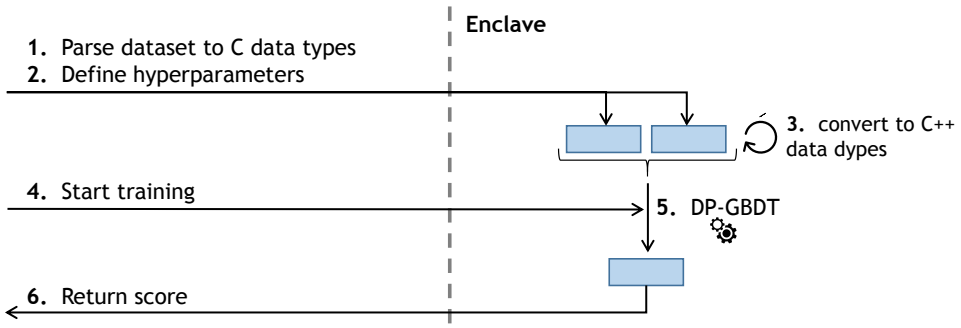

Figure 4.5: Visualisation of procedures inside/outside of the enclave

The DP-GBDT algorithm itself has to be adjusted everywhere where randomness is involved. Reason being that obtaining randomness inside an enclave is a little different than on the outside. SGX provides a function `sgx_read_rand()` which executes the `RDRAND` instruction to generate hardware-assisted unbiased random numbers [41]. Apart from replacing calls that fetch a random number, several functions, that were previously not much more than a simple standard library call, had to be rewritten from scratch. For example randomly selecting/deleting a subset of rows from a matrix. Of course also the external library code for e.g. the logging had to be removed. Similarly no more multithreading is available. Further, there's an important configuration file `Enclave.config.xml` which specifies settings like available memory size. Modifying this according to your needs is indispensable for lager datasets. For information on changes to the Makefile and other details, consider the code in our repository[8].

[8] https://github.com/loretanr/dp-gbdt/blob/main/TODO.todo

Chapter 5

Side-channel hardening

This chapter summarizes the DP-GBDT hardening process. First of all, the goals are reiterated. Subsequently a brief selection of different known hardening tools and techniques is presented. Thereafter, our chosen approach is explained an justified. This is followed by a final section containing additional details, settings and code examples of our hardening efforts.

Goal The aim is to protect the DP-GBDT algorithm from leaking secrets through *digital side channels*. As described earlier, this covers all side-channels that carry information over discrete bits. Examples are the memory access trace, control flow and time. We are not trying to eliminate side-channel leakage entirely. Instead, we eliminate just enough leakage at the right places to achieve ε-differential privacy.

5.1 Known tools and techniques

As side-channel attacks and defense is generally a well researched topic, there are various established techniques and tools available to the developer. We therefore start by presenting a selection of commonly adopted practices. Thereafter, different complications caused by compilers are analyzed. We conclude with a brief overview of tools for automatic hardening/verification. By using such tools, developers can avoid falling into the many traps of manual hardening.

Constant-time programming If implemented naively, many cryptographic algorithms perform computations in non-constant time. Such timing variation can cause information leakage if they are dependant on secret parameters. With adequate knowledge of the application, a detailed statistical analysis could even result in full recovery of the secret values. Constant-time programming refers to a collection of programming techniques that are used to eliminate timing side-channels. The key idea is that it's almost impossible to hide what kinds of computations a program is performing. What can be done, however, is to produce code whose control flow is independent from any secret information. As a general rule of thumb: The input to an instruction may only contain secret information if the input does not influence *what* resources will be used and *for how long*. This means, for example, that secret values must not be used in branching conditions or to index memory addresses. What further complicates matters: Modern CPUs generally provide a large set of instructions, but not all of them execute in constant-time. Common examples are integer division (smaller numbers divide faster), floating point multiplication, square root and common type conversions [32]. Additional issues can arise from gaps in the language standards. The C-standard, for instance, generally doesn't define the relation from source code to hardware instructions. This can result in unexpected non-constant-time instruction sequences.

Operations that do not have a directly corresponding assembly instruction are naturally more prone to this issue.

Blinding Another popular mitigation approach for timing side-channels is blinding. It is essentially the process of obfuscating input using random data before performing a non-constant-time operation. After the operation is complete, the output is de-obfuscated using the same randomness. Put differently, the operation is still non-constant-time, but the secret value is hidden. Two well-known use cases of this approach are old RSA and ECDH implementations [16, 61].

Oblivious RAM Oblivious RAM (ORAM) was introduced for the purpose of enabling secure program execution in untrusted environments. This is accomplished by obfuscating a program's memory access pattern to make it appear random and independent of the actual sequence of accesses made by the program. By utilizing ORAM techniques (e.g. [34, 64]), a non-oblivious algorithm can be compiled to its oblivious counterpart at the expense of a poly-logarithmic amortized cost per access [11]. While this is a great result, it also means that the algorithms become exceedingly slow when processing large amounts of data. As a result, it is generally less suitable for machine learning.

Data-oblivious algorithms A less universal but very effective tactic is to design/rewrite the algorithm itself in a data-oblivious way. In order to prevent an attacker from being able to infer any knowledge about the underlying data, oblivious algorithms need to produce indistinguishable traces of memory and network accesses. Detectable variances must be purely based on public information and unrelated to the input data. This implies the use of oblivious data structures and removal of all data-dependent access patterns. In most cases, manually transformed algorithm are much more efficient than using ORAM (where runtime can easily increase by one or more orders of magnitude [72]). But of course, the rewriting process takes time and is prone to errors. What helps, however, is that over the past years, researchers have produced a large range of oblivious algorithms and building blocks that are at a developer's disposal (e.g. [69, 9, 18]).

5.1.1 Compilation and verification

How can a programmer be sure that the compiler does not optimize away his hardening efforts, or create new side channels on its own? Most high-level programs do not include an execution time element in their semantics. Consequently, compilers provide no guarantee about the runtime of the program. Their sole objective is to make programs run as quickly as possible. In the process, different compilers can produce quite different results. Figure 5.1 shows such an example. Two compilation results from the same piece of source code (top left) are depicted. The source function performs a simple logical `AND` operation. `gcc v11.2` (bottom left) does use the logical and operation as expected. `icc v2021.3.0` (right side), however, inserts a conditional jump that depends on the value of the first operand. The same flags (`-O1`) were used in the process and the compilation was done for the same target architecture (`x86-64`). This phenomenon is called short-circuit evaluation.

As a result it is not just sufficient to ensure that solely constant-time operations are used. We additionally need to make sure, that the program is adequately obfuscated for the compiler not to comprehend its semantics. Otherwise, unwanted and possibly side-channel introducing optimizations are to be expected. In the end, the only reliable way to verify that code written in a high-level language is protected against side-channel attacks, is to check the compiler's assembly output. Preventing the compiler from performing function inlining facilitates this task to some

```
1  int test(bool b1, bool b2)
2  {
3      bool result = b1 and b2;
4      return result;
5  }
```

```
1  test(bool, bool):
2      andl    %edi, %esi
3      movzbl  %sil, %eax
4      ret
```

```
1  test(bool, bool):
2      testb   %dil, %dil
3      je      ..B1.3
4      xorl    %eax, %eax
5      testb   %sil, %sil
6      setne   %al
7      ret
8  ..B1.3:
9      xorl    %eax, %eax
10     ret
```

Figure 5.1: Short-circuit evaluation: bottom left `gcc v11.2`, right side `icc v2021.3.0`

degree. And also keep in mind, that changing the compiler version or modifying unrelated code parts may suddenly change a compiler's output.

5.1.2 Tool assisted hardening

Recent research proposed multiple automatic hardening tools (e.g. Constantine [13], Obfuscuro [3] or Raccoon [58]). The approaches include: Complete linearization of secret-dependent control and data flows, or execution of extraneous "decoy" program paths. Depending on the exact approach, automatic hardening tools introduce substantial runtime overhead. Tool-assisted verification of constant-time properties is also possible [65, 59, 7]. However, most of these methods again demand meticulous program code instrumentation, to i.e. mark variables or memory locations that might contain secret values at some point.

5.2 Chosen approach

This section summarizes the chosen hardening approach and assumptions. We will start by be going over general hardening measures. Afterwards a description of DP-GBDT-specific measures is given. Finally, the compilation and verification techniques are described.

5.2.1 General measures

We chose a manual hardening approach on source code level. There are three main reasons for this choice: (i) In contrast to more suitable auto-hardening applications, which might only contain one single secret key, *all* data is private in DP-GBDT. Hence, secret-dependent accesses spread very quickly which likely results in an automated tool attempting to obfuscate everything. (ii) With manual hardening we can leverage our knowledge of the algorithm to pinpoint critical locations that require attention. We do not need to harden more than necessary, ε-differential privacy is sufficient. (iii) Closer inspection showed that most of the automatic hardening solutions are still in an early/incomplete stage and not quite ready for usage in practice. Some of the papers did not publish their tool all. Others provide tools that mostly serve as a proof of concept, that work on specific small examples. Then again there are tools that require large amounts of annotations and extra information to be added to the source code. This would be very challenging for an algorithm of our size.

In manual hardening, the following general rules have to be be followed: (i) Avoid branch conditions affected by secret data. (ii) Avoid memory look-ups indexed by secret data. (iii) Avoid secret-dependent loop bounds. (iv) Prevent compiler interference with security-critical

operations. (v) Clean up memory of secret data. (vi) Use strong randomness. Apart from these high-level rules, several clearly constant-time violating or non-data-oblivious primitives come to mind: logical boolean operators, comparators, ternary operator, sorting and min/max functions. As these constructs appear quite frequently in code, constant-time and data oblivious versions of all these functions were created and inserted into the algorithm. Details on the implementation of these primitives can be found in section 5.3. As described in the requirements section, we completely eliminate leakage through memory access patterns. This means for example: In order to retrieve a value at a secret dependent index from an array, every single element should be touched. Although this entails a significant execution time penalty, it safeguards the implementation from future attacks with even more powerful and finer graded leakage capabilities. Regarding inherently non-constant-time instructions such as floating point arithmetic, we decided against replacing them entirely. It's not only a major task, it also heavily perturbs code readability. Moreover, for floating point numbers there are significantly fewer constant-time library options available than for integers. We opted for a compromise: we harden the most performance critical section of the code, where 90%+ of time is spent (see A) using `libfixedtimefixedpoint` [8]. Fortunately the code sections that we spend most time in don't contain too many of floating point operations. This way we can circumvent this major implementation feat, while still arguing that it was tested and finishing the job would not incur much more runtime overhead.

5.2.2 DP-GBDT-specific measures

There are three problematic parts in the DP-GBDT algorithm: (i) Training resp. building the decision trees, (ii) inference resp. performing prediction on a finished tree, and (iii) gradient data filtering (GDF), which causes dependencies on secret information (gradients) when choosing the samples for the following trees. Constant-time and data-oblivious substitutes for all three of these have to be designed. We will first expand on training, inference and GDF sample distribution. Later on, in the final paragraph of this section, we cover oblivious tree construction. The latter can be considered a non-obligatory bonus hardening step. In other words: These hardening steps were undertaken in this project, even though they are not necessary for ε-differential privacy.

Oblivious training and inference Whether it's for training or inference, the key thing that needs to be hidden is the path that individual samples take in the decision tree. In normal/non-dp GBDT training, each decision node only uses the data that belongs to that node. In order to conceal which samples belong to this node, we either need to access each sample obliviously, or alternatively, scan through all samples while performing dummy operations for those that do not belong to the node. We chose the second approach: Instead of dividing the set of samples that arrive at an internal node into two subsets (left/right) and continuing on these subsets, we pass along the entire set to both child nodes. Additionally a vector is now passed along which indicates which samples are actually supposed to land in this tree node. In the child nodes, the computation is then performed on all samples. With help of the indication vector, the dummy results are discarded at the end. There a couple more minor details to take into consideration. To name a few examples: (i) When trying out different *potential* splits, to find the one with the highest gain, we must also hide whether samples would go left/right. (ii) We must not leak the number of splits that yield a ≤ 0 gain. (iii) We must not leak the number of unique feature values. Oblivious inference, on the other hand, is simpler. The actual path that a sample takes can easily be hidden by deterministically traversing the whole tree. A boolean indicator, that is, of course, set in an oblivious manner, is passed along to indicate whether a node lies on the real path. This was the entire tree is traversed, while the right leaf value is retrieved.

Oblivious sample distribution There are two factors that affect how samples for the individual trees are chosen: We can use a balanced approach where each tree gets the same amount of samples, or an unbalanced approach, where earlier trees get more samples according to formula (DPBoost [46], Algorithm 2, line 8). The second influential factor is GDF. If GDF is enabled, samples with gradients below a certain threshold will be chosen first. This is of course problematic, as the current gradients depend on the previous tree built from secret data. The key observation that inspired our solution was: We do not need to hide *which* samples are chosen. It is enough to hide *why* a sample was chosen. It's sufficient if it's indistinguishable whether a sample is chosen randomly or because of it's favourable gradient.

Oblivious tree construction As mentioned, hiding the tree building process is not required to ensure ε-differential privacy. It is an extra hardening step that was performed during this thesis. It's still useful as it might allow for a tighter proof in the future. And it also hides information from a potential adversary, that he has no reason to have.

To prevent leaking the shape of the tree under construction it is necessary to: (i) Add a fixed number of nodes to each tree, and (ii) keep the order in which nodes are added independent of the data [45]. This means that we always build a full binary trees. Some nodes will end up being dummy nodes, meaning no sample would actually arrive there. Since, as described in the previous paragraphs, we always scan through all samples at each node, dummy nodes are indistinguishable from real nodes. We further simulate the creation of a leaf node at each node in the tree. Hence a side-channel adversary has no idea whether an internal node, dummy node or leaf node has just been added. Moreover, as we are building the tree in a depth-first (DFS) manner and always go down to the maximum depth, we do not leak any information through order. Furthermore, in case geometric leaf clipping (GLC) is enabled, the clipping operation is done on all nodes to hide which ones are real leaf nodes. Additionally, the exponential mechanism has to be hardened to hide which split is chosen.

To illustrate the entire decision tree hardening process in greater detail, appendix **??** contains pseudocode of several tree-building related functions with all side-channel affected regions highlighted.

Compilation and verification What is the best method to verify the compiled output in a manageable manner? The newly created constant-time functions are all defined in a separate file and namespace. Further, inlining is disabled with vendor-specific keywords. This way you can more conveniently check all hardened functions as they are all gathered in one place. Another extra layer of safety measure that we added to the hardened functions is selective use of the `volatile` keyword. Its purpose is essentially to prevent unwanted compiler optimization. Regarding compiler optimization flags, we decided to take a relatively conservative approach: We opted against using very aggressive compiler optimizations, as it significantly decreased traceability and comprehensibility of the assembly code. Compilation results were inspected after usage of both `-O` and `-O1` gcc flags. No hardening or side-channel related violations were visible.

TODO move this down and clean up technical section

5.3 Technical details

This section offers more detailed insights into the hardening efforts. This is done by presenting several code samples. Even more details can be found in appendix **??**.

Value barrier As already mentioned, we made use of the `volatile` keyword. This identifier is used to inform the compiler that a variable can be changed any time without any task given by the source code. This prevents many optimizations on objects in our source code.

```
template <typename T>
T value_barrier(T a)
{
    volatile T v = a;
    return v;
}
```

Listing 5.1: Value barrier using `volatile` keyword

Logical operators Logical boolean operators can usually just be directly replaced by the corresponding bitwise operators. the logical not can easily be implemented as an xor with 1.

```
__attribute__((noinline)) bool constant_time::logical_not(bool a)
{
    // bitwise XOR for const time
    return (bool) (value_barrier((unsigned) a) ^ 1u);
}
```

Listing 5.2: Constant-time logical `NOT`

Comparators In a similar way to the logical `NOT`, `XOR` can be used to check for (in)equality. To depict order relations, we really only need to define one (for instance >). Following this all other comparators can be constructed by combining it with `NOT` and comparison to zero.

Ternary operator Using a constant time ternary operator (or also called oblivious assign/select), most branches can be transformed to constant-time. This is done by evaluating both branches and choosing the result with the constant-time ternary operator.

```
template <typename T>
__attribute__((noinline)) T select(bool cond, T a, T b)
{
    // result = cond * a + !cond * b
    return value_barrier(cond) * value_barrier(a) +
        value_barrier(!cond) * value_barrier(b);
}
```

Listing 5.3: Constant-time ternary operator

Min/max A min/max function can now easily be built by a combination of the constant-time comparators and ternary operator.

Sorting Oblivious sorting finds application in the DP-GBDT algorithm before the actual training. To be more precise, when feature scaling is activated, the exponential mechanism is used to scale the values into the grid. This process involves finding the confidence interval borders, which in turn requires the feature values to be sorted. We decided to use a $O(n^2)$ algorithm, because it's very easy to harden, and it's not a performance critical task in our algorithm (sorting is done only once before training).

```
// O(n^2) bubblesort
for (size_t n=vec.size(); n>1; --n){
    for (size_t i=0; i<n-1; ++i){
        // swap pair if necessary
        bool condition = vec[i] > vec[i+1];
        T temp = vec[i];
        vec[i] = constant_time::select(condition, vec[i+1], vec[i]);
        vec[i+1] = constant_time::select(condition, temp, vec[i+1]);
    }
}
```

Listing 5.4: Oblivious sort

Fault injection This can be seen as bonus hardening task. Fault injection is generally more of a hardware attack, although recently fault injection attacks have also been achieved in software ([54, 20]). It would be very unfortunate if an adversary could simply increase the privacy budget by glitching/flipping one single bit. Even worse, he might be able to switch off differential privacy entirely, or he may change a hyperparameter that the insurance customers did not agree upon. For this reason a solution was implemented that replaces boolean model parameters (internal 1 or 0) with two random integers with hamming distance >25 (see listing 5.5). Then each time a parameter is used, we first check whether it is one of the two values. This way a glitch would need to flip exactly those >25 bits in order to switch off some privacy related model parameter, which should essentially be impossible in practice.

```
// TRUE:  01101100001011110101111000011011
// FALSE: 00000001110001101010000111100100
#define TRUE 1815043611u
#define FALSE 29794788u
```

Listing 5.5: Parameter fault injection mitigation

Compilation and verification As already mentioned, we suggest not using very aggressive compiler optimizations by using the `-O0` or `-O1` gcc flags. To generate assembly code the `-save-temps` flag can be used. The build process is of course automated using Makefile. As the constant-time functions are all defined in the same file, and inlining is disabled, the hardened functions can quickly be checked.

Arithmetic leakage Inherently non-constant-time instructions, such as floating point arithmetic, were only hardened in the most performance critical section of the code. This is where the

majority of execution time is spent, as depicted in A. The affected functions are:

- `samples_left_right_partition`
- ... TODO

All affected floating point operations (A,B,C,D,... TODO) were replaced by their corresponding fixed point counterpart from `libfixedtimefixedpoint` [8].

Example For illustration purpose a just slightly simplified example function: Listing 5.6 shows the non-hardened code of the prediction process. In other words, this is the function that samples use to go down a finished decision tree and get its corresponding leaf value. Listing 5.7 contains the corresponding hardened code.

```
// recursively walk through the decision tree
double predict(vector<double> sample_row, TreeNode *node)
{
    // base case
    if(node->is_leaf()){
        return node->prediction;
    }

    // recurse left or right
    double sample_value = sample_row[node->split_attr];
    if (sample_value < node->split_value){
        return predict(sample_row, node->left);
    }
    return predict(sample_row, node->right);
}
```

Listing 5.6: Non-hardened inference/prediction

```
// recursively walk through the decision tree
double predict(vector<double> sample_row, TreeNode *node)
{
    // always go down to max_depth
    if(node->depth < max_depth){

        double sample_value = sample_row[node->split_attr];

        // hide the real path a sample takes, go down both paths
        double left_result = predict(sample_row, node->left);
        double right_result = predict(sample_row, node->right);

        // decide whether we take the value from the left or right child
        bool is_smaller = constant_time::smaller(sample_value, node->split_value);
        double child_value = constant_time::select(is_smaller, left_result, right_result);
    }
    // if we are a leaf, take own value, otherwise we take the child's value
    return constant_time::select(node->is_leaf, node->prediction, child_value);
}
```

Listing 5.7: Hardened inference/prediction

Chapter 6

Evaluation

This chapter will give an overview over the achieved results. After covering the methodology and experimental setup, the DP-GBDT performance results are presented. Thereafter, the runtime of the different implementations is compared. This gives an image on the impact of individual hardening measures, and the impact of running the code inside an enclave. The sections are both concluded with a brief discussion.

6.1 Methodology

In this section we demonstrate the performance of the algorithm on four different real world datasets. The datasets vary significantly in size. Two of them are regression tasks, the other ones are binary classification tasks.

name	size	features	task	only use subset
abalone [26]	4177	8	regression	no
yearMSD [27]	515345	90	regression	yes
BCW [29]	699	10	binary classification	no
adult [28]	48842	14	binary classification	no

Table 6.1: Datasets

Experimental Setup The machine that generated these results is running x86_64 GNU/Linux, Debian 10, kernel 4.19.171-2 on an Intel Core i7-7600U @ 2.8 GHz Kaby Lake CPU. It has 20GB RAM, 32KB of L1, 256KB of L2 and 4MB of L3 cache. The compiler is g++/gcc version 8.3.0 (Debian 8.3.0-6).

6.2 Performance

The measurements are carried out for 21 distinct privacy budgets between 0.1 and 10. The number of trees per ensemble and the amount of samples used varies amongst the different datasets. The result values represent the average of running 5-fold cross validation. Therefore a `samples` parameter of 5000 indicates that 4000 samples were used for training and 1000 for validation. The errorbars denote the average standard deviation measured. The regression baseline is achieved by always predicting the average/mean feature value. For binary classification the baseline is attained through the 0R-classifier. For regression tasks we provide both RMSE (penalises errors on outliers) and MAPE (more robust to outliers).

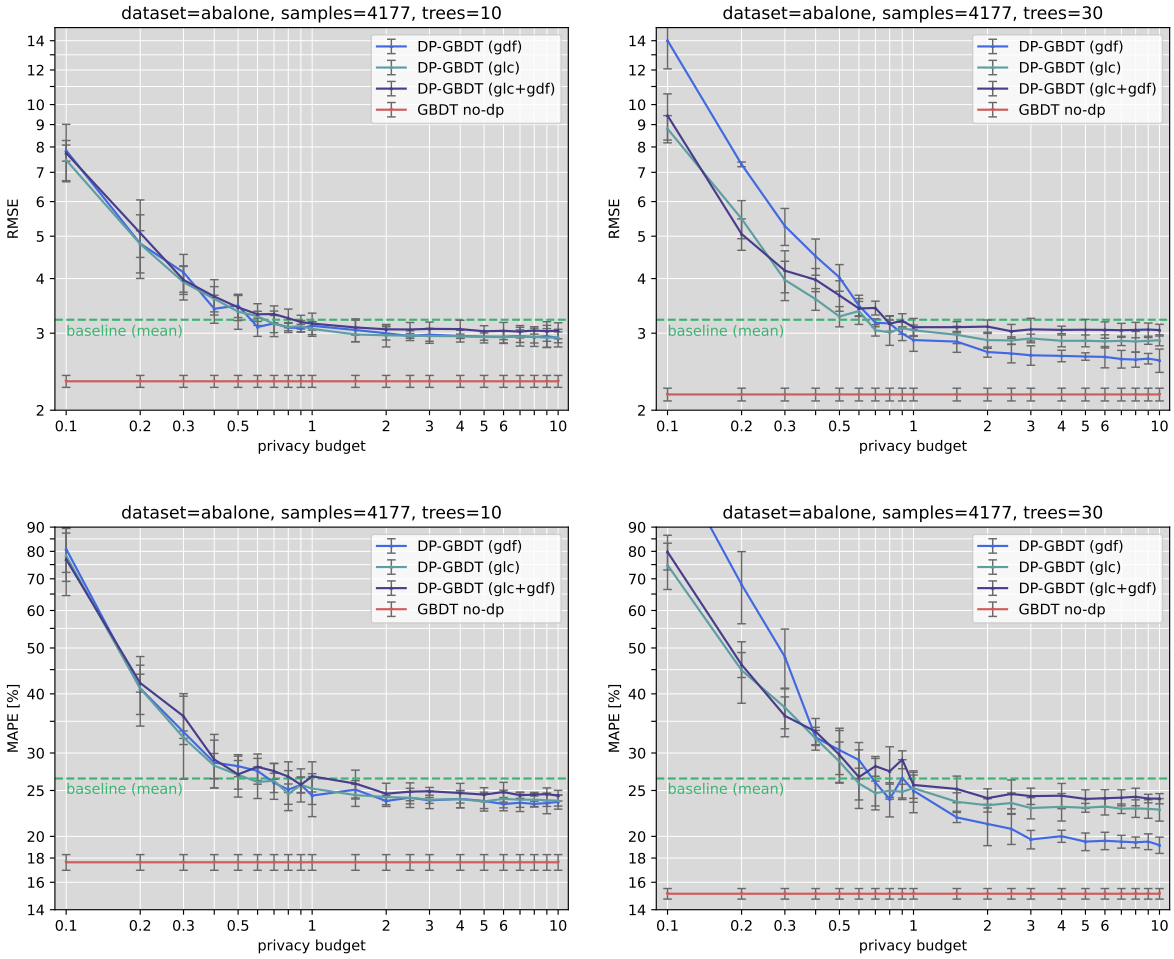

Figure 6.1: Abalone RMSE (top) / MAPE (bottom): Left side ensemble of 10 trees, right side ensemble of 30 trees

Hyperparameters The utilized hyperparameters, are listed in appendix **??**. The `use_grid` option is turned off, since it leads to virtually the same results given a small enough step size while substantially increasing runtime. Experiments showed that our way of scaling feature values into the grid (details can be found in paragraph 4.3) works reliably using a privacy budget of just 0.05. However, there are certainly more effective ways of performing this task that cost less privacy budget (e.g. [23]). Further, note that the chosen hyperparameters are **by no means optimal** in terms of performance. With hyperparameter tuning, even better results should easily be achievable. Such tuning was omitted due to timing constraints and due to the lack of suitable frameworks for this task in C++.

Abalone For abalone we can take the full dataset (4177 samples). As it is a regression task we can take the mean of the target feature as a baseline, which in the case of abalone is around 3.22. Figure 6.1 shows that all DP-GBDT variations start defeating the baseline at a privacy budget of around 0.7. Using more trees (30 instead of 10) clearly improves performance. The right hand side plots also show that using GDF leads to worse for small privacy budgets, but better performance for higher budgets. At no point however, do the DP-GBDT implementations come reasonable close to the non-dp performance. GLC does not seem to improve performance in

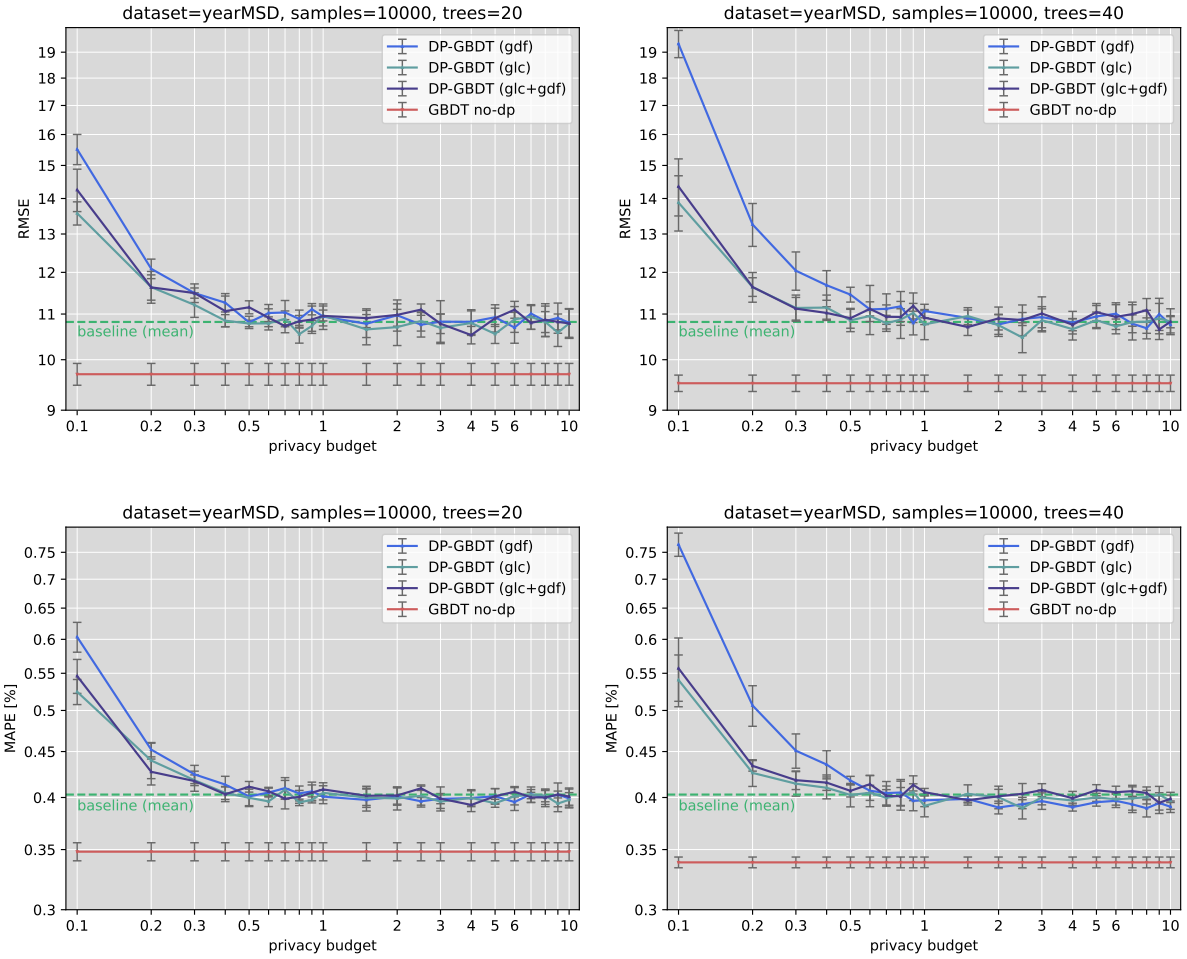

Figure 6.2: Year RMSE (top) / MAPE (bottom): Left side ensemble of 20 trees, right side ensemble of 40 trees

any form. With the RMSE and MAPE looking very similar, it seems like outliers are forecasted pretty effectively.

yearMSD Since this dataset is quite large (over 500k entries and around 90 features) experiments were conducted on subsets of it. Further, following the guidelines on its source webpage, it was ensured that training and test samples were chosen from distinct parts of the dataset. This avoids the 'producer effect', so no song from a given artist ends up in both the train and test set. Figure 6.2 shows that using a subset of 10k samples is just not enough to reliably beat the baseline. Creating ensembles with 40 instead of 20 trees shows no significant impact. As with the previous dataset, usage of GDF tends to negatively impact performance for small privacy budgets.

Breast Cancer Wisconsin Being the smallest dataset, the results were not as stable as those from other datasets. Therefore the result values are the average of 5-fold cross validation repeated 10 times. Figure 6.3 shows that all DP-GBDT variations start defeating the baseline at a privacy budget of around 0.7. The GDF variation is the only version that is able to continuously improve its score with increasing privacy budget. The other variants achieve their best scores at privacy budgets of around 3, afterwards, their results get worse. Using more trees (40 instead of 20)

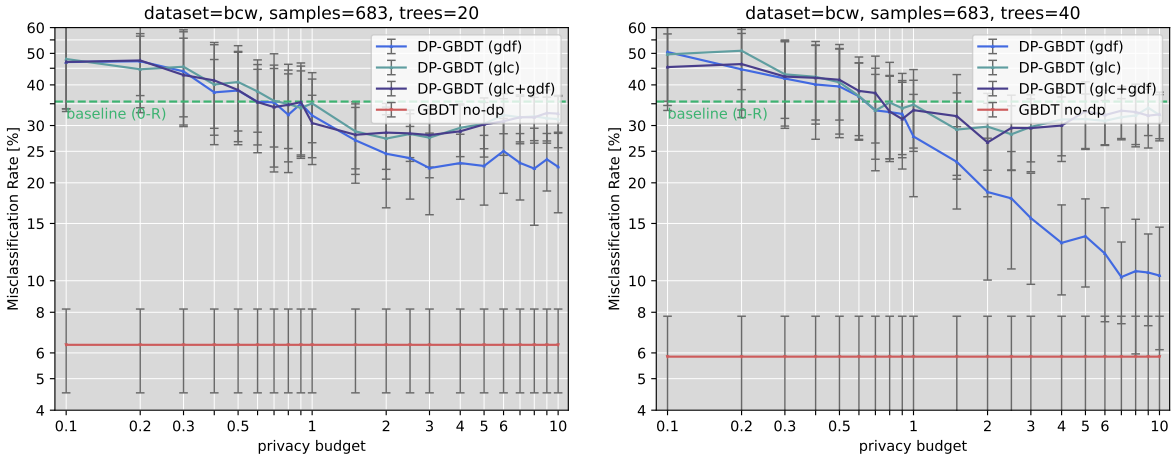

Figure 6.3: BCW misclassification rate: left side ensemble of 20 trees, right side 40 trees

improves the performance mainly when using GDF. Again, GLC does not seem to improve performance in any significant manner.

Adult Using this dataset the task is to predict whether an individual's annual income exceeds $50,000 based on census data. Other machine learning algorithms' scores are generally in the range of 14-20% error rate [44]. Figure 6.4 shows that the DP-GBDT algorithm does not really

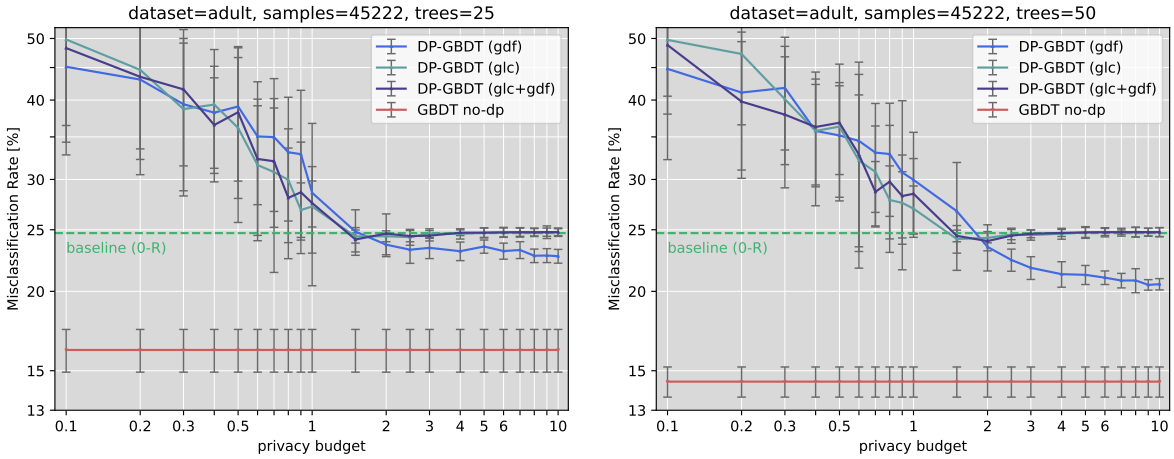

Figure 6.4: Adult misclassification rate: left side ensemble of 25 trees, right side 50 trees

work well on this dataset. At least not with these specific hyperparameters. Only the GDF version manages to beat the baseline. Not by a big margin and only for ε's ≥ 2. Once again, increasing the amount of trees in the ensemble from 25 to 50 does not have a great impact.

Balanced sample distribution This paragraph shows the effect of a balanced vs. unbalanced sample distribution strategy using the abalone dataset as an example. The unbalanced variant uses the formula in (TODO ref to background) to distribute samples (the number is always decreasing). The balanced version hands out the same number of samples to all trees in an

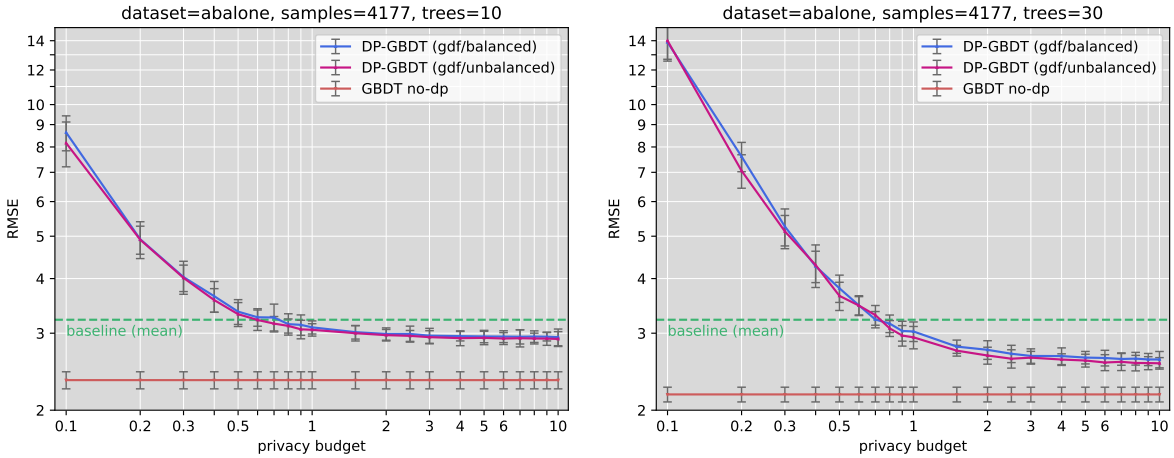

Figure 6.5: Abalone RMSE: Balanced/unbalanced comparison: ensemble of 10 resp. 30 trees

ensemble. As depicted in figure 6.5, using a balanced instead of an unbalanced approach does not significantly affect performance.

6.2.1 Discussion

First of all, it's interesting that, unlike DPBoost [46] proclaims, GLC did not improve the overall performance on any of our datasets. The performance was notably worse for the adult dataset. It's likely that this is at least partially due to a general weakness of many decision tree algorithms: Decision tree learners can create biased trees if some classes dominate [49], which is the case for the adult dataset (25%/75%). Balancing the dataset prior to fitting might improve this result. In general, performance for small privacy budgets is quite bad. We seem to be adding so much noise, that the use of trees of our depth (generally a depth of 6 was used) does more harm than good. So it seems like small $\varepsilon's$ are actually a real problem for decision trees. As the overlying goal is to achieve a useful learning process for privacy budgets around 0.1 or 0.2, there's still quite some work to do. But a couple of improvements down the line this might be possible. Also keep in mind, that the current design's performance can certainly be vastly improved by well-versed hyperparameter tuning.

6.3 Runtime

In this section we demonstrate the training time variations between the different implementations. Even the DP-GBDT algorithm will likely not be run very frequently in practice, achieving a respectable runtime performance was important to us.

Hyperparameters We use the following parameters for all measurements across all implementations:

privacy_budget	0.5	gradient_filtering	false
nb_trees	10	leaf_clipping	true
min_samples_split	2	balance_partition	true
learning_rate	0.1	use_grid	false
max_depth	6	use_dp	true

Table 6.2: Parameters for runtime measurements

Results All runtime results were obtained by performing 5-fold cross validation. Therefore the the measured values in 6.3 have been divided by 5.

python_gbdt: Patched version of the reference DP-GBDT implementation that was created by T. Giovanna prior to this thesis. It utilizes multithreading through `sklearn`.

cpp_gbdt: Base C++ version that was extensively described in section 4.2. It does not run inside an enclave and it is not hardened. It mainly exists for experimentation with the underlying algorithm. Several measures (e.g. multithreading) have been undertaken to boost its performance (details can be found in 4.2).

enclave_gbdt: This is essentially `cpp_gbdt` but ported into an SGX enclave.

hardened_gbdt: Hardened version of `cpp_gbdt`. It is not placed in an enclave however. Chapter 5 described the hardening process. Even though no side-channel violations were found when compiling with the `-O1` flag, we included the `-O0` measurement as well to be on the safe side.

hardened_sgx_gbdt: This is the final combination of `hardened_gbdt` and `enclave_gbdt`.

Implementation	**abalone**			**adult**			**yearMSD**	
	300	**1000**	**4177**	**300**	**1000**	**5000**	**300**	**1000**
python_gbdt	1.5	2.3	15.2	1.5	3.4	41.8	9.4	88.3
cpp_gbdt	<0.01	0.01	0.05	<0.01	0.01	<0.01	<0.01	0.1
enclave_gbdt	0.05	0.26	1.38	todo	todo	todo	todo	todo
hardened_gbdt -O0	1.04	10.3	154.6	1.94	15.8	366	11.9	102
hardened_gbdt -O1	0.18	1.6	22.0	0.32	2.46	59.5	2.0	16.2
hardened_sgx_gbdt	todo	todo	todo	todo	todo	todo	todo	todo

Table 6.3: Runtime results [s]

First of all, compared to the Python reference implementation, `cpp_gbdt` offers a solid 506x speedup on average. The speedup increases further with even larger datasets. This is because its bottleneck in training is finding good splits (see A.1), and this task grows non-linearly with increasing dataset size. Moving `cpp_gbdt` into an enclave, decreases its runtime by [TODO]x on average. This can certainly be contributed to the absence of multithreading, but also to general SGX related overhead like encryption and enclave calls. `hardened_gbdt -O1` is on average 381x slower than `cpp_gbdt`. Turning off compiler optimizations (compile with `-O0`) further increases the runtime by 6.3x on average, which results in a total slowdown of around 2390x. But note that this implementation contains a few more hardening measures than strictly necessary for differential privacy (see 5.2.2). Last but not least, `hardened_sgx_gbdt`, the combination of hardened DP-GBDT and running in an SGX enclave, is as expected once again noticeably slower, [TODO]x on average.

Next, table 6.4 shows the runtime consequences of grid-usage and use of constant-time fixed point arithmetic. For the grid, a step size of [TODO] was employed. To demonstrate the effect of using constant-time fixed point numbers/operations instead of floating point arithmetic, we replaced them in the core functions of the algorithm. Around 95% of execution time is spent these functions (see A).

Implementation	**abalone**			**adult**			**yearMSD**	
	300	**1000**	**4177**	**300**	**1000**	**5000**	**300**	**1000**
h_1 := hardened_gbdt -O1	0.18	1.6	22.0	0.32	2.46	59.5	2.0	16.2
h_1 + use_grid	todo	todo	todo	todo	todo	todo	todo	todo
h_1 + libfixedtimefixedpoint	todo	todo	todo	todo	todo	todo	todo	todo

Table 6.4: Runtime implications of `use_grid` and using constant-time fixed point arithemtic [s]

The results show that. TODO "grid is a big overhead, lftfp is ok."

6.3.1 Discussion

As our runtime results show, hardening can impose a lot of runtime overhead. Especially when a "complete" hardening approach is chosen. If a more minimal way of hardening is applied, like in the our DP-GBDT case, it's much more efficient. In hindsight, at the time of writing this thesis, also a few weaknesses of `hardened_gbdt` became more and more apparent. For example, using BFS instead of DFS tree induction could allow significant execution time improvements for the hardened version. This shows the importance of already reflecting about constant-time properties very early on in such a project. We further showed that porting the DP-GBDT algorithm into the enclave introduced a very manageable overhead. We also produced a fast baseline implementation, ideal for future experiments. In general our runtime experiments underline the feasibility of this setup for future usage in practice.

Chapter 7

Secure deployment

The task of performing DP-GBDT in practice would be easy if, for example, the training data was available all at once, or computers wouldn't break. However, as this is generally not the case these issues need to be addressed. This chapter aims to give a high level overview of the challenges that accompany the real-world insurance use case. We will start with a short reiteration the system components and the threat model. Afterwards a motivating example is presented to highlight some of the challenges present. Then the focus is shifted towards the possible designs of the individual steps. A brief analysis of the overall design concludes this section.

General setup and threat model The DP-GBDT enclave is deployed on the insurance's servers and is reachable by external customers. When a new insurance customer is acquired, said customer sends a filled out questionnaire about their IT-security policies directly to the enclave. To store received data, the enclave has access to a designated hard disk. It has further access to multiple secure monotonic counters on the same machine. These counters leverage non-volatile memory to create persistent state. We use two different threat models to cover the two main security goals. TM1: From a customer's perspective, the goal is data privacy even in presence of an adversary that controls the machine containing the enclave. Such an adversary has full control over software, hardware and network of the system. Most importantly we assume he has side-channel capabilities, which include memory access and program counter traces of the enclave execution. The objective of the adversary is to infer secret information about the participating customers. TM2: From the insurance's perspective, the goal is to obtain good results from the algorithm, even in presence of customers with malicious intent. Such a customer does not have access to the servers where the enclave resides. However, the malicious customer has full control over his questionnaire data that is sent to the enclave, how many times it is sent, etc. The objective of a malicious customer is to decreases the quality of the overall result.

Motivating example We take on the insurance's perspective: Consider a scenario, where we have successfully prepared and started the DP-GBDT enclave on our servers. The questionnaire data collection works as intended, and after a while we instruct the enclave to train a model using the first 500 questionnaires. However, while we are studying the model output, adversary $\mathcal{A}$, a malicious internal system administrator, decides to secretly instruct the enclave to train once again on the same data. By running the algorithm twice and analyzing both execution traces, differential privacy guarantees are violated: According to theorem 2.6 for sequential composition of DP functions, privacy budget would have to be paid twice for this to be allowed. To address this threat we enable the DP-GBDT enclave access to persistent state through the use of a secure monotonic counter. The counter indicates whether there was already a training carried out for 500

questionnaires. $\mathcal{A}$ counteracts this again by setting up a second DP-GBDT enclave on the same server with its own monotonic counter. Again, he can observe the training twice and privacy is violated accordingly. As an answer, through clever use of randomness and the SGX sealing functionality, the insurance decides to ensure that *only* the original enclave is able to train on these questionnaires. While this solves the previous privacy concerns, this massively decreases usability: In case the server containing the enclave breaks (or crashes at the wrong time, for example during training), we are in trouble. Due to it's persistent state, the enclave will refuse to train the model again. Therefore all previous progress is lost.

Goals Our goal is to come up with viable strategies for secure deployment of the DP-GBDT enclave setup. Both the insurance and its customers should be sufficiently protected from incidents covered by our threat model. Simultaneously, general usability has to be maintained. Specifically, we address the following issues:

- Data collection and provisioning over an extended time period
- Training on newly acquired customer data to improve the existing model output
- Training on old data because the output model was lost
- Enclave replication and migration while maintaining persistent state
- Prevention of enclave state rollback or fork attacks

7.1 Design

7.1.1 Enclave setup

This subsection describes preliminary steps and preparations to be completed by the insurance before putting the enclave into operation. In particular, numerous settings need to be defined and hardcoded in the enclave source code **before** attestation. This is important from the client's perspective. For example the client does not want to rely on the insurance choosing a sufficiently small privacy budget, after having already submitted its questionnaire. The aforementioned settings include:

- Model hyperparameters
- ID's of two secure monotonic counters (questionnaire-counter, log-counter)
- The individual points at which training can be performed. Example rule: Training is only possible at $500 + n * 200$, $n \in \mathbb{N}$ questionnaires
- The mapping between these training points and corresponding log-counter value. For example $\{(500, 2), (700, 3), (900, 4), ..\}$
- An upper limit for both secure monotonic counters. Though unlikely to be ever reached, we should not wait to see what happens once the counters are exhausted.

Note that defining the training points in advance serves two purposes: First of all, from an algorithm standpoint it does not make sense to build a decision tree with e.g. just a handful of samples. Second, fixed training intervals makes it easy to ensure that a particular questionnaires can only be used for training in one particular set of questionnaires. More on this later.

Client application and client-server communication The client application will be fairly simple. It allows the client to fill out a questionnaire about their IT-security policies. Upon completion of the questionnaire, attestation between the client and the enclave can be initiated. A trusted third party ensures public access to both the source code and configuration of the enclave and the client application. Afterwards, the client application lets you send the encrypted

questionnaire answers to the trusted application. The encryption key for this is a shared session key that is established during attestation (as in [39]). Further, the client application should ensure that the filled out questionnaires satisfy some minimum quality requirements. For example, it should not be permitted to submit empty questionnaires or ones containing values that are impossible.

Customer legitimacy What happens in the case of a legitimate customer with malicious intent? A customer could, for example, send 100 questionnaires to the enclave instead of one. First of all, we would like to emphasize that this is not a problem from a (differential) privacy standpoint. On the contrary, more questionnaires generally means better privacy, regardless of their content. The problem is that it could decrease the output model's expressive power. To prevent this we see two possibilities: (i) Reception resp. acceptance of a questionnaire is performed by the insurance, which then relays a maximum of one questionnaire per customer to the enclave. (ii) The insurance hands out some kind of access token to each customer, that is signed by the insurance. Questionnaires are sent directly to the DP-GBDT enclave which checks the token's validity and keeps track of used tokens.

Enclave state and initialization Assuming that, for whatever reason, the DP-GBDT enclave was turned off or had to be restarted. After turning back on, how does the enclave know in which state it is in? We suggest the use of two secure monotonic counters to save an enclave's state. As depicted in figure 7.1 the enclave uses the `log-counter` to determine whether it is being started for the very first time. If the counter is greater than zero, initialization has already been done. As a result the enclave then goes into collection resp. waiting mode. This is the default state which will only be left upon reception of a `start_training()` command issued by the insurance.

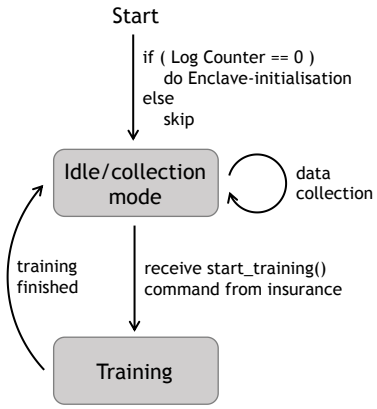

Figure 7.1: Enclave state diagram

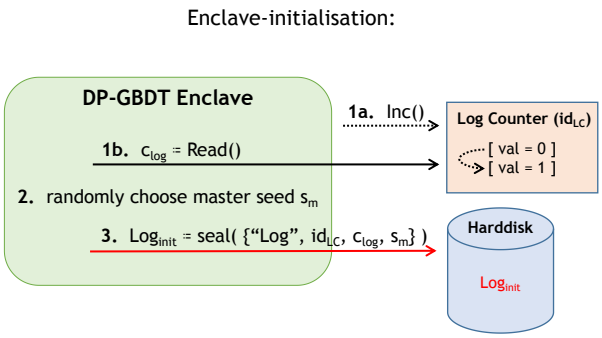

Figure 7.2: Enclave-initialization

Enclave-initialization is depicted in figure 7.2. Note that the secure monotonic counter was already initialized in the setup phase and its ID was hardcoded into the DP-GBDT enclave's source code. Regarding step (2.): The master seed s_m has two main purposes:

- First, it serves as an identifier for all data that was written to disk by this exact enclave. This prevents attacks where an adversary could swap sealed data on the hard disk with sealed data from another clone of the same enclave. This could result in a setting where the same data is used in two different sets of questionnaires, which violates DP. It is therefore important that the master seed s_m stems from a true random number generator (TRNG). In practice this can be accomplished through the `sgx_read_rand()` function [38, 39].

- Second, as indicated before, we offer the functionality to re-compute a lost model. To guarantee differential privacy in that case, we need to produce the exact same model as the previous training did. Hence, the exact same randomness must be used in the DP-GBDT algorithm. We achieve this by using a pseudorandom number generator (PRNG), that is seeded with the master seed s_m, for the actual training.

7.1.2 Enclave operation

We now describe the DP-GBDT enclave's two core operations: data collection and training. As you will see, data collection stays essentially the same, regardless of whether we're collecting the first or the 942th questionnaire. In terms of training however, we will differentiate between (i) the very first one, (ii) later trainings with new data (refinement trainings), and (iii) the repetition of training that was already done in the past (re-training).

Data collection phase Let's start with an overview of how receiving and saving customer data looks like from the enclave's perspective. Figure 7.3 illustrates this. Some remarks on selected

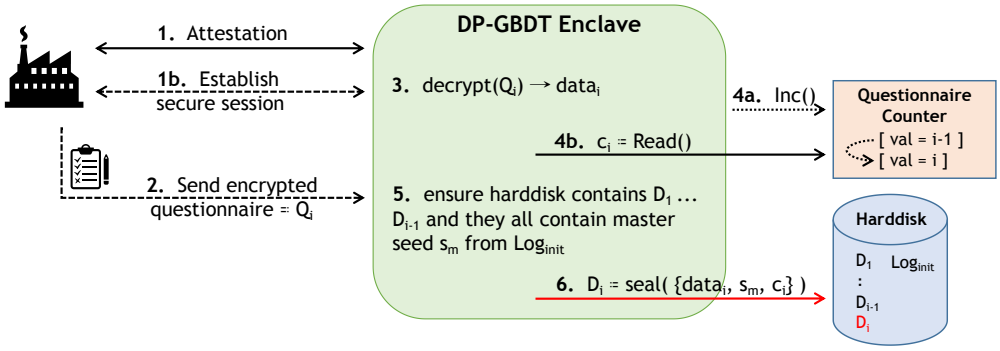

Figure 7.3: Data collection phase

steps: (1.&2.): As already mentioned, the encryption key for the questionnaire data is a shared session key that is established during attestation. This ensures confidentiality, integrity and replay protection of the sent data. (5.) This step is not strictly necessary, but it helps the insurance identify problems or tampering early. (6.) The sealed data consists of the questionnaire, the master seed and the `questionnaire-counter`. The counter determines the order of samples in the dataset. It ensures that, for example, an adversary cannot observe training with 100 samples, drop one sealed data piece from disk, and then monitor training again with the original 99 samples plus another sample in place of the dropped one. Again we would have a differential privacy problem if the same data is used in two different training sets. The downside of such an approach is that we absolutely must not loose any sealed data piece. Otherwise all progress is lost and we back at the very start (before data collection!). The same situation occurs if the enclave was to crash after increasing the questionnaire counter but before sealing the received data to disk.

First training As soon as enough samples are collected, the insurance can initiate the first training with the `start_training()` command. The ensuing sequence of steps is pictured in figure 7.4. Please be aware that some messages contain redundancy. The goal was to create comprehensible diagrams and not to save every byte of space. Also notice the `load`* primitive in the diagram. This is an abbreviation for the following tasks:

- Unseal the data from disk

- Perform different checks to recognize whether tampering occurred. This includes: (i) checking whether all sealed data pieces D_i in our desired training interval are present (and contain the right counter values c_i). (ii) Check whether all D_i contain the master seed s_m from Log_{init}.
- Abort training in case any check fails

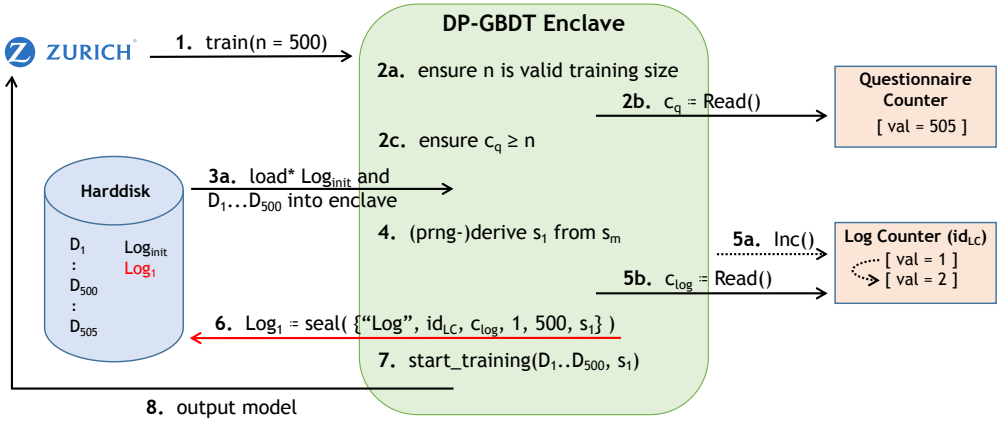

Figure 7.4: First training

Remarks: Generally, because enclave-initialization already increases the log-counter, it will always be one step ahead. Once the first training was completed it will read 2, once another refinement training is completed it will read 3, etc. (6.): To enable possible re-training in the future, the pseudorandomly generated seed s_1 needs to be saved as well. (7.): s_1 is the seed for the randomness that will be used in the training.

Refinement training This paragraph demonstrates how the existing model can be improved with newly received data. For DP-GBDT there are generally two options: (i) The new samples could be used in combination with old ones to train the entire model again. (ii) The new samples are used separately to create new trees that are appended to the previous ensemble. We will only discuss the latter one here, since the first option is still under research. Another reason is that this approach does not require an extra payment of privacy budget, because we operate on fresh data. The process is visualized in figure 7.5.

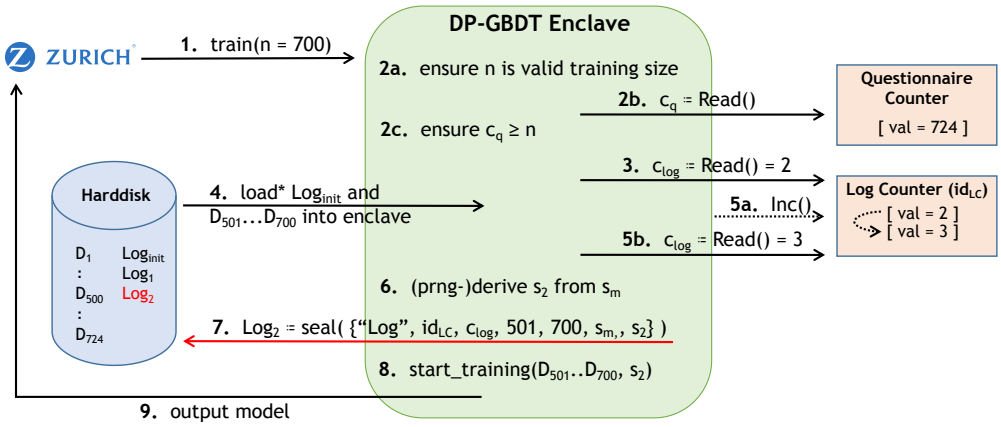

Figure 7.5: Refinement training

Model re-training Finally, let's consider how a previously trained piece of the model can be recreated. This is depicted in figure 7.6. Remarks: (3.) As described in section 7.1.1, the

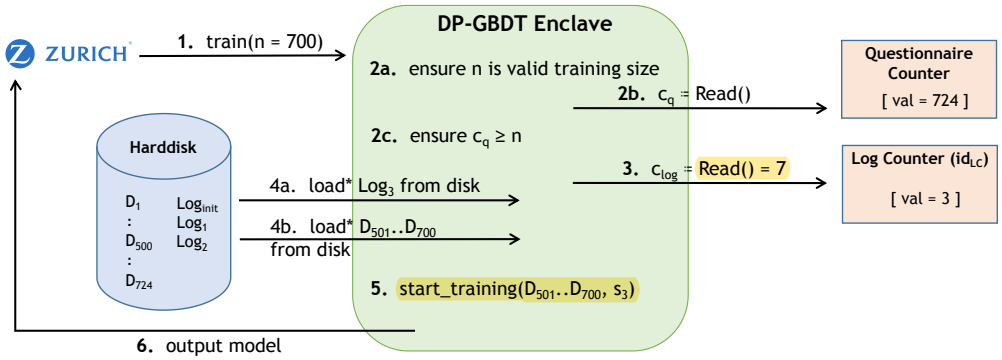

Figure 7.6: Re-training

mapping (700→Log_3) is hardcoded in the enclave's source code.

7.1.3 Enclave migration/replication

While the previous setup works in theory, it entails some major drawbacks in practice. The main problem with the design being that it is bound to one single computer. If the machine broke, we would have to start all over again. Also, the insurance might want to transition to newer hardware at some point etc. For this reason a migration as well as a replication approach are now presented.

Enclave migration Fortunately there already exists research on this exact subject. This paragraph presents a short summary of the work of Alder et al. [5]. They propose an enclave migration approach that guarantees the consistency of its persistent state (such as sealed data and monotonic counters). This is achieved by using a migration library and by use of a separate migration enclave on both the source and destination machine. A high level overview of the migration process can be found in figure 7.7. The main steps are:

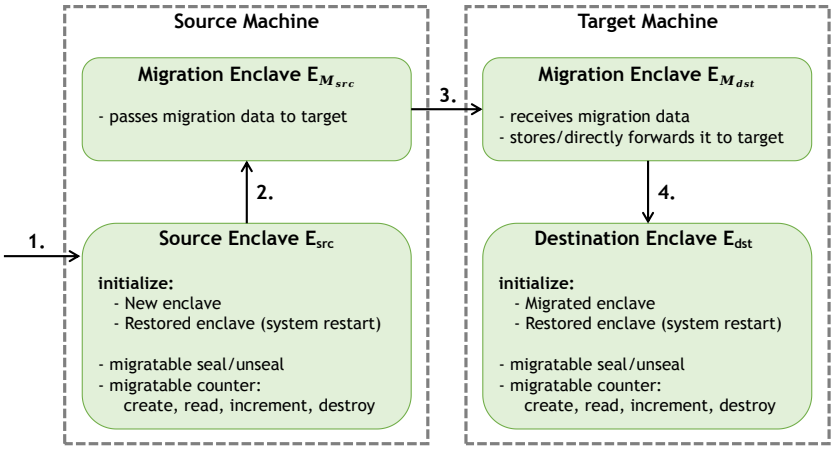

Figure 7.7: Enclave migration [5]

1. A migration notification is sent to the enclave.

2. As soon as the migration enclave is locally attested, it receives the data that should be migrated. During the same period, the migration library disables the source enclave from operating further.
3. The source migration enclave then performs mutual remote attestation with the destination migration enclave. This process entails the establishment of a secure channel and mutual authentification of the migration enclaves.
4. To complete the process, the migration enclaves check whether the remote destination enclave matches the local destination enclave. If this is successful the migration data is handed over and the procedure completes.

Enclave replication Nevertheless, enclave migration might not be enough of a safety margin. What if the computer containing the GBDT enclave and it's corresponding monotonic counters explodes? To address this concern we suggest a simple form of enclave replication that is undertaken by the insurance before the data collection even starts. Given a secure environment we can create a setup of multiple enclaves using the master/slave principle. The process looks as follows (figure 7.8): First, the desired number of DP-GBDT enclave replicas is created. They are essentially clones with the exception of their public/private keys, and the monotonic counter ID's that are hardcoded in their source code. In a secure environment the DP-GBDT enclaves are then started up. Next, they announce their individual public keys. The public keys are then hardcoded into the master enclave's source code. The master enclave is started. The master enclave now takes over the part of generating a master seed s_m and distributes it to the other enclaves. The data collection phase now looks a bit different (figure 7.9): Insurance customers will not directly communicate with the DP-GBDT enclaves anymore, but with the master enclave instead. Similarly, attestation will be carried out between the master enclave and the customers. But of course the customers still have to verify that the slave enclaves are all legitimate DP-GBDT enclave clones. Upon reception of a customer questionnaire, the master enclave will distribute it to the GBDT enclaves. For this it has to decrypt and re-encrypt the questionnaire with the public keys of the slave enclaves. Remarks:

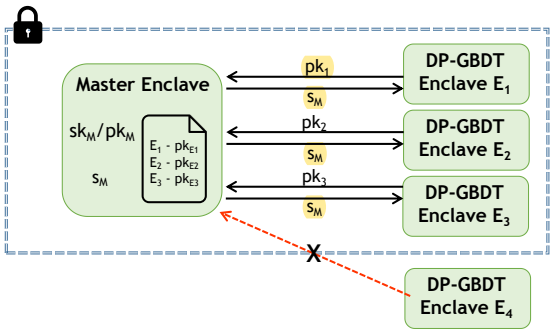

Figure 7.8: Replication setup

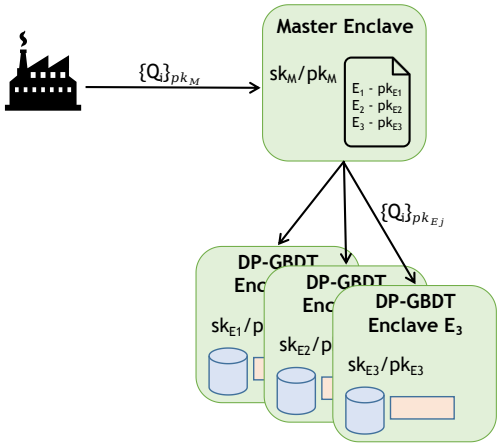

Figure 7.9: Enclave replication

- The reason for carrying out the initial replication setup in a secure environment is to simplify the replication procedure. If required, there actually exists research that describes this process in an insecure environment [22]. Keep in mind, that our goal should always be

to prevent an adversary from adding his own local enclave to the list of slaves. This is an extra precaution measure that should prevent a setting where an adversary could monitor the DP-GBDT enclave training on his own machine from the comfort of his home.

- The master enclave itself does not have persistent state. Therefore it itself is not prone to most of the issues the DP-GBDT enclaves suffer from.
- The crux of such a setup is ensuring privacy resp. resilience against an insurance-internal adversary that undermines the distribution process between master and slave enclaves. Imagine a scenario where the adversary cuts the connection to half of the enclaves while one questionnaire is being distributed. Subsequently, for the next questionnaire, he inverts the setup by cutting off the other half. As a result, the counters of all enclaves would be equal again, even though different data was received. This would result in the same data being used in two different sets of questionnaires, which violates DP. There are two solutions that come to mind to address these kind of attacks: (i) We add persistent state to the master enclave with help of a monotonic counter. The counter value is then included when forwarding questionnaires to the slave enclaves. Thus, slave enclaves can detect missing information and react accordingly. This approach however calls for appropriate replication of the master enclave. (ii) Some form of consensus / multi-party agreement protocol between the master and slave enclaves is installed. Upon detection of tampering, slave enclaves will exclude themselves from further participation. Bezerra et al. [10] propose an example solution for state-machine replication.

7.2 Analysis

First and foremost, we argue that customer data privacy is ensured in such a deployment setup. To our knowledge, and given an appropriately hardened DP-GBDT implementation, we are protected from the different variations of enclave state/rollback attacks that could endanger the differential privacy guarantees. We further prevent external customers with malicious intent from perturbing the result of the output model. To achieve a reasonable resilience to (un)foreseeable events, such as computer crashes or data loss, we offer a migration and replication solution. This adds a certain amount of complexity, but significantly decreases the likelihood of disaster (through e.g. an enclave crash at the worst possible moment, or loss of sealed data). There is still the risk of denial of service attacks, but we consider this out of scope. Overall this chapter should offer a good overview for future discussion and implementation work.

Chapter 8

Related Work

Privacy-preserving machine learning is not a novel concept. A number of proposals exist that combine decision tree based classification of private data in a multi-party setting (e.g. [24, 47, 25]). One possible approach is to leverage homomorphic encryption schemes [4, 68]. Another approach is the use of Differential Privacy (DP), as introduced by Dwork et al. [30]. It can be argued, that this is the only mathematically rigorous definition of privacy in the context of machine learning and big data analysis. Through extensive research and growing industry acceptance, DP has become the standard of privacy over the past decade [6]. Multiple DP-GBDT solutions [46, 1, 48, 67] have been proposed since then. The combination of DP-GBDT with SGX enclaves has not been as thoroughly researched however. To our knowledge, there are two works that take a relatively similar approach to this thesis:

Allen et al. (2019) "*An Algorithmic Framework For Differentially Private Data Analysis on Trusted Processors*" [6]. Their high-level goal and architecture are the same as ours: Run DP algorithms inside SGX enclaves and eliminate leakage. Specifically, this work proposes a mathematical model for designing DP algorithms in TEE-based setting. They assume that the leakage only consists of (i) the output model and (ii) memory access trace. In this setting they ensure that DP guarantees hold for three selected algorithms. Decision trees are not among them. Further their focus lies more on the algorithmic side, which means they don't consider an entire system with data provisioning from users, disk as persistent storage, and so on.

Law et al. (2020) "*Secure collaborative training and inference for XGBoost*" [45]. The authors present a privacy-preserving system for multiparty training and inference of XGBoost [19] (efficient, open-source GBDT library) models. Their goal is to protect the privacy of each party's data as well as the integrity of the computation with SGX enclaves. However, they only consider side-channel leakage through memory access patterns. For this purpose, multiple data-oblivious building blocks for GBDT are created. Another difference to our approach is that no DP mechanisms are used. In other words, they aim for complete obliviousness of the entire algorithm, while we only selectively harden certain areas to achieve DP.

Chapter 9

Conclusion

In this thesis we present the implementation and hardening process of DP-GBDT, a relatively new variation of privacy preserving machine learning with promising applications. Compared to predecessor implementations, runtime performance was drastically improved and multiple bugs were identified and removed. The implementation was further ported into an SGX enclave and hardened to achieve ε-differential privacy. Affected code sections were protected from leaking secrets through *digital side-channels*, a notion which sums up all side-channels that carry information over discrete bits (such as the memory access trace, control flow or time). We saw that eliminating 100% of leakage is hard to achieve due to inherently non-constant-time hardware instructions such as several floating point arithmetic operations. The entire implementation and hardening process is captured in detail and should offer some guidelines for future work. The prediction accuracy of the DP-GBDT algorithm was evaluated on four real-world UCI standard datasets. Although we are not far from achieving good forecasting results with smaller privacy budgets, further research is required. Runtime performance of the training process was increased by over 500x compared to the predecessor implementation. This offered some cushion for the hardening overhead that subsequently added. The execution time of the fully hardened enclave implementation is well into the acceptable range for an algorithm that is not supposed to be run very frequently. In addition, we proposed a secure real-world deployment scheme of the DP-GBDT enclave setup in the insurance use case. Both the insurance and its customers are sufficiently protected from incidents covered by our threat model. At the same time, we placed emphasis on general usability and fault tolerance. Ultimately, we further reduced the gap between privacy preserving state-of-the-art machine learning frameworks and their real-world application.

9.1 Future Work

While this work shows encouraging results, there are still several paths to be explored in future work. On the theoretical side, further thought has to be put into making the algorithm more efficient in terms of privacy budget. On the practical side there is a broad selection of things that would add value to the project:

- Model hyperparameter tuning
- Experimentation with insurance specific synthetic datasets
- Visualisation of the overall output model
- Tool-assisted verification of constant-time properties
- Python framework/wrapper for more convenient usage

Appendix A

Implementation Details

C++ GBDT parameters

nb_trees	The total number of trees in the model.
privacy_budget	The privacy budget available for the model.
learning_rate	The learning rate.
max_depth	The maximum depth for the trees.
min_samples_split	The minimum amount of samples required to split an internal node.
gradient_filtering	Whether or not to perform gradient based data filtering during training (only available on regression).
leaf_clipping	Whether or not to clip the leaves after training (only available on regression).
balance_partition	Balance sample partition for training among the trees. If set to True, all trees within an ensemble will receive an equal amount of training samples. If set to False, each tree will receive <x> samples where <x> is given in line 8 of the algorithm in the DPBoost paper.
use_decay	If True, internal node privacy budget has a decaying factor.
cat_idx	List of indices for categorical features.
num_idx	List of indices for numerical features.
l2_threshold	g_{L^*} from [50] TODO
l2_lambda	The regularization parameter.
use_grid	Use grid to test out possible split values. Necessary for dp, but brings a reasonable performance penalty.
grid_borders	Specify grid borders.
grid_step_size	Grid step size.
cat_values	When using a grid for testing out different splits, this parameter can be used to specify the different feature values categorical features could theoretically take. This way splits on all possible values will be tested even though they don't actually appear in the dataset. As a result we can hide the fact whether a value is part of the dataset from a side channel observer.

scale_X If we do use a grid, we should efficiently use it. Therefore all feature values of should be scaled into the grid range. This can be done (not trivial though), but costs some privacy budget.

scale_X_percentile Select the percentile for the confidence intervals for scale_X.

scale_X_privacy_budget Select the amount of privacy budget you're willing to pay for the scale_X. The more you pay, the more reliable and accurate the result.

Adding new datasets to the project

Adding a new dataset to the project is very simple. Once you have saved your new dataset in a file in comma separated form. All you have to do is create a function, similar to the one in listing A.1, where you specify the basic properties of the dataset.

```
1  DataSet *Parser::get_adult(std::vector<ModelParams> &parameters, size_t
       num_samples, bool use_default_params)
2  {
3      std::string file = "datasets/real/adult.data";
4      std::string name = "adult";
5      int num_rows = 48842;
6      int num_cols = 15;
7      std::shared_ptr<BinaryClassification> task(new BinaryClassification());
8      std::vector<int> num_idx = {0,4,10,11,12};
9      std::vector<int> cat_idx = {1,3,5,6,7,8,9,13};
10     std::vector<int> target_idx = {14};
11     std::vector<int> drop_idx = {2};
12     std::vector<int> cat_values = {};
13     return parse_file(file, name, num_rows, num_cols, num_samples, task,
           num_idx, cat_idx, cat_values, target_idx, drop_idx, parameters,
           use_default_params);
14 }
```

Listing A.1: How to add a new dataset

Profiling output

Figures A.1 and A.2 illustrate the output from Intel VTune. This shows current bottlenecks of both the "regular" C++ implementation and the hardened C++ implementation.

Verification output

TODO scrot

Function	CPU...	CPU Tim...
clone	98.2%	0s
DPEnsemble::train	98.2%	0.028s
start_thread	98.2%	0s
func@0xbbb20	98.2%	0s
DPTree::make_tree_DFS	95.5%	0.203s
DPTree::fit	95.5%	0s
DPTree::find_best_split	94.5%	2.519s
DPTree::compute_gain	77.8%	20.069s
operator new	16.7%	6.778s
_int_free	7.9%	3.195s
__memmove_avx_unaligned_erms	3.6%	1.471s
__GI_	3.2%	1.304s
std::_Rb_tree<double, double, std::_Id	2.3%	0.915s
std::_Rb_tree_insert_and_rebalance	2.2%	0.902s
DPTree::exponential_mechanism	2.1%	0.150s
__libc_start_main	1.8%	0s
main	1.8%	0s

Figure A.1: `cpp_gbdt`: Time spent in %

Function	CPU...	CPU Tim...
_start	100.0%	0s
main	100.0%	0s
__libc_start_main	100.0%	0s
DPEnsemble::train	99.6%	0s
DPTree::fit	98.1%	0s
DPTree::make_tree_dfs	97.6%	0s
DPTree::find_best_split	96.5%	0.040s
DPTree::compute_gain	80.7%	0.208s
DPTree::samples_left_right_partition	61.6%	0.588s
constant_time::select<bool>	19.3%	0.336s
constant_time::select<int>	18.0%	0.232s
DPTree::exponential_mechanism	8.6%	0.024s
constant_time::value_barrier<bool>	4.1%	0.112s
constant_time::select<double>	4.1%	0.096s
operator new	3.7%	0.100s
__GI___exp	3.7%	0.100s
log_sum_exp	3.2%	0.012s
std::vector<int, std::allocator<int>>::ve	3.1%	0s
constant_time::value_barrier<bool>	2.7%	0.072s
constant_time::value_barrier<int>	2.5%	0.068s
constant_time::value_barrier<int>	2.2%	0.060s
constant_time::value_barrier<bool>	2.2%	0.060s

Figure A.2: `hardened_gbdt`: Time spent in %

Bibliography

[1] Martin Abadi et al. “Deep Learning with Differential Privacy”. In: July 2016, pp. 308–318. DOI: 10.1145/2976749.2978318.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “On the Power of Simple Branch Prediction Analysis”. In: *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’07. Singapore: Association for Computing Machinery, 2007, pp. 312–320. ISBN: 1595935746. DOI: 10.1145/1229285.1266999. URL: https://doi.org/10.1145/1229285.1266999.

[3] Adil Ahmad et al. “OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX”. In: Jan. 2019. DOI: 10.14722/ndss.2019.23513.

[4] Adi Akavia et al. “Privacy-Preserving Decision Tree Training and Prediction against Malicious Server”. In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 1282.

[5] Fritz Alder et al. “Migrating SGX Enclaves with Persistent State”. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 195–206. DOI: 10.1109/DSN.2018.00031.

[6] Joshua Allen et al. *An Algorithmic Framework For Differentially Private Data Analysis on Trusted Processors*. July 2018.

[7] José Bacelar Almeida et al. “Verifying Constant-Time Implementations”. In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC’16. Austin, TX, USA: USENIX Association, 2016, pp. 53–70. ISBN: 9781931971324.

[8] Marc Andrysco et al. “On subnormal floating point and abnormal timing”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 623–639.

[9] Michael Bader and Christoph Zenger. “Cache oblivious matrix multiplication using an element ordering based on a Peano curve”. In: *Linear Algebra and its Applications* 417 (Sept. 2006), pp. 301–313. DOI: 10.1016/j.laa.2006.03.018.

[10] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. “Scalable State-Machine Replication”. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 331–342. DOI: 10.1109/DSN.2014.41.

[11] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. “Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing”. In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS ’13. Hangzhou, China: Association for Computing Machinery, 2013, pp. 207–218. ISBN: 9781450317672. DOI: 10.1145/2484313.2484341. URL: https://doi.org/10.1145/2484313.2484341.

[12] Avrim Blum et al. “Practical privacy: The SulQ framework”. In: Jan. 2005, pp. 128–138. DOI: 10.1145/1065167.1065184.

[13] Pietro Borrello et al. “Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization.” In: Nov. 2021. DOI: 10.1145/3460120.3484583.

[14] L. Breiman et al. *Classification and Regression Trees*. Wadsworth and Brooks, 1984.

[15] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *12th USENIX Security Symposium (USENIX Security 03)*. Washington, D.C.: USENIX Association, Aug. 2003. URL: https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical.

[16] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *12th USENIX Security Symposium (USENIX Security 03)*. Washington, D.C.: USENIX Association, Aug. 2003. URL: https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical.

[17] Jo Van Bulck et al. “Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1041–1056. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[18] T.-H Chan et al. “Oblivious Hashing Revisited, and Applications to Asymptotically Efficient ORAM and OPRAM”. In: Nov. 2017, pp. 660–690. ISBN: 978-3-319-70693-1. DOI: 10.1007/978-3-319-70694-8_23.

[19] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *KDD*. ACM, 2016, pp. 785–794.

[20] Zitai Chen et al. “VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 699–716. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[21] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 86.

[22] Aritra Dhar et al. “ProximiTEE: Hardened SGX Attestation by Proximity Verification”. In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. CODASPY ’20. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 5–16. ISBN: 9781450371070. DOI: 10.1145/3374664.3375726. URL: https://doi.org/10.1145/3374664.3375726.

[23] W. Du et al. “Differentially Private Confidence Intervals”. In: *(NeurIPS* (2020). arXiv: 2001.02285. URL: https://arxiv.org/abs/2001.02285.

[24] Wenliang Du and Zhijun Zhan. “Building Decision Tree Classifier on Private Data”. In: *Proceedings of the IEEE International Conference on Privacy, Security and Data Mining - Volume 14*. CRPIT ’14. Maebashi City, Japan: Australian Computer Society, Inc., 2002, pp. 1–8. ISBN: 0909925925.

[25] Wenliang Du and Zhijun Zhan. “Using Randomized Response Techniques for Privacy-Preserving Data Mining”. In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’03. Washington, D.C.: Association for Computing Machinery, 2003, pp. 505–510. ISBN: 1581137370. DOI: 10.1145/956750.956810. URL: https://doi.org/10.1145/956750.956810.

[26] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: https://archive.ics.uci.edu/ml/datasets/abalone.

[27] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd.

[28] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: https://archive.ics.uci.edu/ml/datasets/adult.

[29] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original).

[30] Cynthia Dwork. “Differential Privacy”. In: *Encyclopedia of Cryptography and Security (2nd Ed.)* Springer, 2011, pp. 338–340.

[31] Cynthia Dwork et al. “Calibrating Noise to Sensitivity in Private Data Analysis”. In: vol. Vol. 3876. Jan. 2006, pp. 265–284. ISBN: 978-3-540-32731-8. DOI: 10.1007/11681878_14.

[32] Agner Fog. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. URL: https://www.agner.org/optimize/instruction_tables.pdf (visited on 10/13/2021).

[33] Daniel Genkin, Adi Shamir, and Eran Tromer. “Acoustic Cryptanalysis”. In: *J. Cryptol.* 30.2 (Apr. 2017), pp. 392–443. ISSN: 0933-2790. DOI: 10.1007/s00145-015-9224-2. URL: https://doi.org/10.1007/s00145-015-9224-2.

[34] Oded Goldreich and Rafail Ostrovsky. “Software Protection and Simulation on Oblivious RAMs”. In: *J. ACM* 43.3 (May 1996), pp. 431–473. ISSN: 0004-5411. DOI: 10.1145/233551.233553. URL: https://doi.org/10.1145/233551.233553.

[35] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[36] Intel. *Intel SGX and Side-Channels*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sgx-and-side-channels.html (visited on 11/10/2021).

[37] Intel. *Intel SGX SDK releases*. URL: https://github.com/intel/linux-sgx/releases (visited on 10/10/2021).

[38] Intel. *Intel Software Guard Extensions*. URL: https://software.intel.com/en-us/sgx (visited on 10/10/2021).

[39] Intel. *Intel software guard extensions SDK for linux*. URL: https://download.01.org/intel-sgx/linux-1.9/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.9_Open_Source.pdf (visited on 10/10/2021).

[40] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Third. Sept. 2011. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[41] Yaoqi Jia et al. “Robust P2P Primitives Using SGX Enclaves”. In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2020.

[42] Guolin Ke et al. “LightGBM: A Highly Efficient Gradient Boosting Decision Tree”. In: *Advances in Neural Information Processing Systems 30 (NIP 2017)*. 2017. URL: https://www.microsoft.com/en-us/research/publication/lightgbm-a-highly-efficient-gradient-boosting-decision-tree/.

[43] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Advances in Cryptology — CRYPTO’ 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. ISBN: 978-3-540-48405-9.

[44] Ronny Kohavi and Barry Becker. *The Adult dataset*. URL: http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html (visited on 10/30/2021).

[45] Andrew Law et al. “Secure Collaborative Training and Inference for XGBoost”. In: (Oct. 2020).

[46] Qinbin Li et al. “Privacy-Preserving Gradient Boosting Decision Trees”. In: *CoRR* abs/1911.04209 (2019). arXiv: 1911.04209. URL: http://arxiv.org/abs/1911.04209.

[47] Yehuda Lindell and Benny Pinkas. “Privacy Preserving Data Mining”. In: *Advances in Cryptology — CRYPTO 2000*. Ed. by Mihir Bellare. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 36–54. ISBN: 978-3-540-44598-2.

[48] Xiaoqian Liu et al. “Differentially private classification with decision tree ensemble”. In: *Applied Soft Computing* 62 (2018), pp. 807–816. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2017.09.010. URL: https://www.sciencedirect.com/science/article/pii/S1568494617305495.

[49] Wei-Yin Loh. “Fifty Years of Classification and Regression Trees 1”. In: 2014.

[50] Rudolf Loretan. *TODO*. URL: https://github.com/loretanr/dp-gbdt/blob/main/TODO.todo (visited on 10/13/2021).

[51] Sinisa Matetic et al. “ROTE: Rollback Protection for Trusted Execution”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1289–1306. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic.

[52] Ahmad Moghimi et al. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations”. In: *Int. J. Parallel Program.* 47.4 (Aug. 2019), pp. 538–570. ISSN: 0885-7458. DOI: 10.1007/s10766-018-0611-9. URL: https://doi.org/10.1007/s10766-018-0611-9.

[53] Peter L. Montgomery. “Modular multiplication without trial division”. In: *Mathematics of Computation* 44 (1985), pp. 519–521.

[54] Kit Murdock et al. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057.

[55] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. “A Survey of Published Attacks on Intel SGX”. In: *ArXiv* abs/2006.13598 (2020).

[56] J. R. Quinlan. “Induction of decision trees”. In: 1.1 (Mar. 1986), pp. 81–106. DOI: 10.1007/bf00116251.

[57] John Quinlan. *C4.5: programs for machine learning*. Elsevier, 2014.

[58] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 431–446. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane.

[59] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. "Sparse Representation of Implicit Flows with Applications to Side-Channel Detection". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 110–120. ISBN: 9781450342414. DOI: 10.1145/2892208.2892230. URL: https://doi.org/10.1145/2892208.2892230.

[60] Werner Schindler. "A Timing Attack against RSA with the Chinese Remainder Theorem". In: *Cryptographic Hardware and Embedded Systems — CHES 2000*. Ed. by Çetin K. Koç and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 109–124.

[61] Werner Schindler. "Efficient Side-Channel Attacks on Scalar Blinding on Elliptic Curves with Special Structure". In: 2015.

[62] Michael Schwarz et al. "Malware Guard Extension: abusing Intel SGX to conceal cache attacks". In: *Cybersecurity* 3 (Dec. 2020). DOI: 10.1186/s42400-019-0042-y.

[63] Si Si et al. "Gradient Boosted Decision Trees for High Dimensional Sparse Output". In: *ICML*. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 3182–3190.

[64] Emil Stefanov et al. "Path ORAM: An Extremely Simple Oblivious RAM Protocol". In: *J. ACM* 65.4 (Apr. 2018). ISSN: 0004-5411. DOI: 10.1145/3177872. URL: https://doi.org/10.1145/3177872.

[65] *Valgrind*. URL: https://valgrind.org/ (visited on 10/13/2021).

[66] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: Oct. 2017, pp. 1–6. DOI: 10.1145/3152701.3152706.

[67] Tao Xiang et al. "Collaborative ensemble learning under differential privacy". In: *Web Intelligence* 16 (Mar. 2018), pp. 73–87. DOI: 10.3233/WEB-180374.

[68] Justin Zhan. "Using Homomorphic Encryption For Privacy-Preserving Collaborative Decision Tree Classification". In: Jan. 2007, pp. 637–645. ISBN: 1-4244-0705-2. DOI: 10.1109/CIDM.2007.368936.

[69] Bingsheng Zhang. "Generic Constant-Round Oblivious Sorting Algorithm for MPC". In: vol. 6980. Oct. 2011, pp. 240–256. DOI: 10.1007/978-3-642-24316-5_17.

[70] Yinqian Zhang et al. "Cross-VM Side Channels and Their Use to Extract Private Keys". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 305–316. ISBN: 9781450316514. DOI: 10.1145/2382196.2382230. URL: https://doi.org/10.1145/2382196.2382230.

[71] Lingchen Zhao et al. "InPrivate Digging: Enabling Tree-based Distributed Data Mining with Differential Privacy". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2018, pp. 2087–2095. DOI: 10.1109/INFOCOM.2018.8486352.

[72] Wenting Zheng et al. “Opaque: An Oblivious and Encrypted Distributed Analytics Platform”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 283–298. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.