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

Privacy-preserving machine learning enables multiple parties to use their data for collaborative data analysis while protecting the privacy of the individual datasets. Trusted execution environments such as Intel SGX can be leveraged to safeguard such computations in practice. SGX enclaves offer excellent performance, but require carefully selected and adapted machine-learning algorithms to prevent side-channel leakage of sensitive information. With this thesis we aim to reduce the gap between theory and real-world application. As a first step, a differentially-private gradient-boosted decision trees (DP-GBDT) implementation was built in C++. This combination has been subject of recent research work, and unites a very successful learning algorithm with differential privacy. The algorithm was subsequently hardened against a broad class of side-channels which we call *digital side-channels*. This includes leakage through the memory access trace, control flow or data size. Several critical bugs were identified and substantial runtime improvements (506x compared to the predecessor implementation) were made. The prediction accuracy of the algorithm was thoroughly evaluated on four UCI-standard datasets. As a final step, we explored solutions for real-world secure deployment of the system. Special emphasis was put on data privacy in the presence of e.g. state rollback attacks, as well as maintaining usability and fault tolerance.

Acknowledgements

First and foremost, I wish to thank my advisors, Dr. Kari Kostiainen and Prof. Dr. Esfandiar Mohammadi, for their continuous advice and encouragement throughout this thesis. Your drive and ability to deconstruct complex problems truly inspired me. Further, I would like to thank Moritz Kirschte for explaining the intricate details of differential privacy to me and always answering my questions.

I am also grateful to my friends for all the discussions, coffee breaks and conversations that greatly helped me to stay focused and motivated.

Last but not least, thanks to my beloved family for being the best family I could ever ask for.

Rudolf Loretan

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 such as hacking incidents, data leaks or IT service outages. Asymmetry in information is one of the major problems in the insurance industry. Insurers have significantly less information about insured objects (e.g. a vehicle or a home) than their customers, which 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. Customers are generally reluctant to fully disclose details of their IT infrastructure and security policies. There are concerns that honest answers about poor IT-security practices might be used to discriminate against them, either at the time of insurance pricing or during potential future claim settlement.

The recent advances in trusted execution environments (TEEs) and privacy-preserving machine learning opens up new avenues for dealing with the aforementioned problem of information asymmetry. A hardware-protected enclave, e.g. using Intel SGX, could be leveraged to collect private customer questionnaires, and subsequently train a machine learning model on the data in a privacy-preserving manner. In order not to leak any secret information about the individual participants, techniques like differential privacy (DP) can be used. This way, the insurance company is able to build a useful aggregate model while protecting the privacy of each customer at the same time. According to recent research [20], users would be more inclined to share private data in such a privacy-preserving setting.

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. A potentially malicious insider at the insurance company must not be able to leak secret data. (ii) The insurer is looking for good learning results from the final 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. Several factors endanger these requirements: First, even though the training process takes place inside a TEE, leakage of secret information can occur. It is a known issue that TEEs are susceptible to side-channel attacks [37]. Second, even if all side-channels during the model 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 or rollback attacks? How can the system cope with data loss and other human or machine related faults?

By combining differential privacy, gradient-boosted decision trees (GBDT) learning and SGX enclaves a potential solution to the aforementioned problems can be assembled. As a result, three main goals were set out:

Goals (i) *Customer data privacy*: First and foremost, the GBDT implementation must provide privacy, more precisely ε-differential privacy, to all customers providing data for the learning algorithm. This necessitates the addition of DP modifications, and porting the implementation into a TEE such as Intel SGX. Further, as SGX enclaves are vulnerable to side-channel attacks, the 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 data size.

(ii) *Efficient implementation*: To take advantage of Intel SGX, a standalone DP-GBDT implementation in C/C++ has to be created. The implementation should be bug-free and able to deliver good prediction results. Ultimately, the algorithm should produce acceptable results for small privacy budgets. 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.

(iii) *Secure deployment*: We further aim to explore the challenges of real-world secure deployment of the system. These include privacy attacks through enclave state rollback or fork attacks, data collection and provisioning over an extended time period, continuous improvement of the previous model with newly acquired data, and fault tolerance in case of human or machine errors.

1.3 Approach

Two prior works form the basis and starting point of this thesis. With DPBoost ([48], 2020), Li et al. 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. After porting the code into an SGX enclave, manual side-channel hardening on source code level was conducted. The process differed from traditional hardening to some extent: We did not eliminate every single bit of leakage. Instead, just enough leakage was removed at the right places to achieve ε-differential privacy. This approach was much more effective than tool assisted hardening as we were able to leverage our knowledge of the DP-GBDT algorithm. Further, several data-oblivious and constant-time building blocks were created to replace common and reoccurring operations. The underlying hardware of the enclave setup was assumed to be uncompromised, patched, and working as expected. Denial of service attacks also fell out of the scope of this thesis. The prediction accuracy of the algorithm was evaluated on four UCI-standard datasets. Moreover, detailed runtime measurements were conducted that show the impact of our hardening methods. Finally, we explored possibilities for secure deployment of the enclave system. With the help of secure monotonic counters, persistent state was achieved. The challenges were highlighted and solutions for enclave initialisation, data collection and enclave replication/migration were assembled.

Results A very fast C++ DP-GBDT implementation was created (506x speedup on average, compared to the predecessor code). Our experiments also show a respectable forecasting accuracy. The DP-GBDT implementation starts beating the naive baseline classifier at around $\varepsilon = 0.7$. The overlying goal is to achieve a useful learning process for very small privacy budgets of around 0.1-0.2. Therefore, further improvement is still required. However, there is still much to be gained through hyperparameter tuning and other algorithm optimisations, which both fell out of the scope of this thesis.

1.4 Contribution

The main contributions of this thesis are:

- Building a C++ DP-GBDT implementation that runs inside an SGX enclave
- Hardening the implementation against side-channels
- Evaluating the accuracy and runtime of the algorithm
- Exploring possibilities for secure deployment

There are further some secondary contributions that developed along the path: (i) Previous results and implementations of the DP-GBDT algorithm 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 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. This includes insights on what kind of code changes are necessary, and how to divide code into inside and outside of the enclave.

Chapter 2

Background

2.1 Decision trees

Machine learning generally consists of two steps, learning and prediction. 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. In other words, they 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 categorise 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).

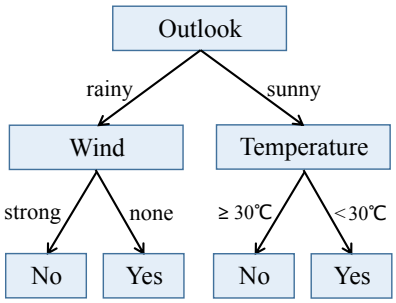

Figure 2.1: Example decision tree

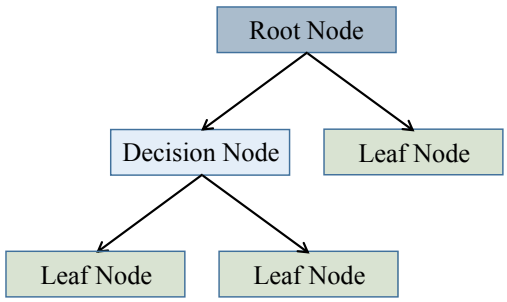

Figure 2.2: Decision tree terminology

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 entropy or *information gain*. Information gain is a statistical property that measures

how well a given attribute separates the training samples according to their target classification. There exist many variations of this recursive building algorithm, such as ID3 [57], C4.5 [58] or CART [13].

As already indicated, decision trees offer several advantages: They are simple to visualise and interpret, usually little to no data preprocessing is required, and they are widely applicable as they make no assumptions about the shape of the data. However, there are also some weaknesses:

- Decision tree learners tend to create overly complex trees that fail to generalise the data well. This is called *overfitting*.
- Being a greedy algorithm, it does not guarantee to return the globally optimal model.
- Decision tree learners are inclined to create biased trees if some target classes dominate [51]. It is thus advised to balance such datasets prior to decision tree fitting.
- 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 [51]. *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 n individual models are weighted and combined to a final, strong classifier.

2.2 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, a multitude of sophisticated and high-performance GBDT frameworks are publicly available, such as LightGBM [43] and XGBoost [18].

A decision tree grows from its root to its 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: [48]

$$G(I_L, I_R) = \frac{(\sum_{i \in I_L} g_i)^2}{|I_L| + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{|I_R| + \lambda} \tag{2.1}$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ is the gradient on a given convex loss function l at the t^{th} iteration, and λ is the regularisation parameter. The exact derivations of the objective and loss function can be found in DPBoost [48]. By traversing all combinations of features and feature values, GBDT finds a split which maximises the gain. If the current node cannot achieve the splitting requirements (i.e. if it is below the maximum tree depth, or all splits have a sub-zero gain), it becomes a leaf node. We define $I = I_L \cup I_R$. The weight of each leaf is computed by the following

function: [48]

$$V(I) = -\eta \frac{\sum_{i \in I} g_i}{|I| + \lambda} \tag{2.2}$$

By applying a shrinkage/learning rate η to the leaf values, we can reduce the impact of each individual tree, allowing future trees to improve the model [48]. The entire GBDT process is shown in Algorithm 1:

Algorithm 1: GBDT training process [33]

Input: $X = X_1, .., X_n$: instances, $\mathbf{y} = y_1, .., y_n$: 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 2.1
            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 2.2
```

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 [29]. 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 2.1 (ε-Differential Privacy [48]) *Let ε be a positive real number and f be a randomised function. Function f provides ε-differential privacy if, for any two neighboring datasets $D \sim D'$ and every output O of function f:*

$$Pr[f(D) \in O] \leq e^{\varepsilon} \cdot Pr[f(D') \in O] \tag{2.3}$$

The parameter ε, also called *privacy budget*, 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 DP mechanisms. We say a function $f: \mathcal{X}^n \rightarrow \mathbb{R}^m$ has sensitivity Δ, if for all neighboring $D, D' \in \mathcal{X}^n$ it holds that $||f(D) - f(D')||_1 \leq \Delta$ [48].

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. When using the Laplace mechanism, controlled noise is added to the query result before it is returned to the user. The noise is sampled from the Laplace distribution centered at 0 and scaled by factor b [30].

Definition 2.2 (Laplace Distribution) *A random variable has a $Lap(\mu, b)$ distribution if its probability density function is:*

$$f(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x-\mu|}{b}\right) \tag{2.4}$$

Theorem 2.3 (Laplace Mechanism [48]**)** *Let $f: \mathcal{D} \rightarrow \mathcal{R}^d$ be a function. The Laplace mechanism F is defined as:*

$$F(D) = f(D) + Lap(0, \Delta f/\varepsilon) \tag{2.5}$$

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 2.4 (Exponential Mechanism [48]**)** *Let $u: (\mathcal{D} \times \mathcal{R}) \rightarrow \mathbb{R}$ be a utility function, with sensitivity Δu. The exponential mechanism F is defined as:*

$$F(D, u) = \textit{choose } r \in \mathcal{R} \textit{ with probability} \propto \exp\left(\frac{\varepsilon u(D, r)}{2\Delta u}\right) \tag{2.6}$$

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 have to be added up for each randomised mechanism. 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 2.5 (Sequential Composition [48]**)** *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.*

Theorem 2.6 (Parallel Composition [48]**)** *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

Differentially-private gradient-boosted decision trees (DP-GBDT) finally combines the different notions and formulas from the previous background sections. Literature ([71, 48]) proposes two adjustments for Algorithm 1 to build 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 exponential mechanism can choose a split with a proportional probability.

2.4.1 DPBoost

DPBoost [48] is Li et al.'s work on DP-GBDT, which set the theoretical foundation for this thesis. The authors' goal was to improve model accuracy of previous DP-GBDT work [1, 50, 67]. Several optimisations are proposed, such as tighter sensitivity bounds for the leaf nodes and the splitting function, and more effective privacy budget allocation. This results in a better overall prediction performance. 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 to obtain 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, the sensitivities of G (Equation 2.1) and V (Equation 2.2) can be bound: 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 [48], Chapter 3.

Geometric leaf clipping When using GDF, all trees have the same sensitivities. 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 (GLC) mechanism was developed, that can be conducted before the Laplace mechanism (Theorem 2.3) 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 of the current tree that is built.

Adaptive privacy budget allocation Last, the authors suggest allocating half of the privacy budget for the leaf nodes (denoted as ε_{leaf}). The remaining budget is divided equally between each depth of the internal nodes (ε_{node} per level). By Theorem 2.6, because the inputs of each depth level are disjoint, the privacy budget consumption of only one depth has to be counted. Therefore $\varepsilon_{leaf} + \varepsilon_{node} * d_{max} = \varepsilon_t$ holds.

2.5 Intel SGX

Intel Software Guard Extensions (SGX) [39] is a set of architectural extensions that can provide strong security guarantees to software, even when exposed to powerful adversaries. It can be used to provide integrity and confidentiality for security sensitive computation performed on a computer where all privileged software, such as kernel or hypervisor, is potentially malicious. SGX is contained inside modern Intel x86 mainframe CPU packages. Hence, its security properties are strongly tied to the processor's hardware implementation. All other components of the system (including OS, firmware, and memory hardware) are removed from the trusted computing base (TCB) [19]. This is achieved by ensuring that both processor and memory state of an enclave are only accessible by enclave-internal code. Protection from external accesses and physical attacks on enclave memory is prevented by a hardware unit inside the CPU, called the memory encryption engine (MEE). Through stringent encryption and decryption, the MEE ensures that

Algorithm 2: Differentially private GBDT training process [48]

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 → Theorem 2.6
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**
        $G_i \leftarrow \frac{(\sum_{i \in I_L} g_i)^2}{|I_L|+\lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{|I_R|+\lambda}$ ▷ Compute gain, Equation 2.1
        $P_i \leftarrow \exp(\frac{\varepsilon_{node} \cdot G_i}{2\Delta G})$ ▷ Theorem 2.4
      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 2.2 and Theorem 2.3

secret data is only stored in plaintext inside the processor. Furthermore, SGX provides attestation mechanisms, with which enclaves can prove to remote parties that they have been initialised correctly on a genuine (and therefore presumed secure) Intel CPU.

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 by themselves are not secure software modules. They are software modules that can be executed in a TEE. Due to the fact that enclaves cannot be debugged or monitored inside TEEs at runtime, their code has to be verified by users before execution. However, as the untrusted system controls the enclave initialisation, an adequate solution is required. Through *attestation*, a specific software can demonstrate its trustworthiness to an external party in terms of authenticity and integrity [19]. There are two types of enclave attestation supported by SGX, local and remote. During local attestation, a second special enclave has to 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* then signs this information. In order to achieve secure communication between these two enclaves, local attestation is performed [19]. 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. In this way, the remote party can determine that (i) no tampering occurred, and (ii) the attested software is running within an genuine hardware-assisted SGX enclave.

Sealing In practice, programs likely contain 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 [19]. 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 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 [19].

Secure monotonic counters TEEs need protection against rollback and similar attacks. An adversary could, for example, replay old messages or interleave output from multiple enclave instances. To mitigate these kind of attacks, secure monotonic counters (SMCs) have been introduced with Intel SGX SDK 1.8 [38]. 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 has been initialised, `Read()` and `Incr()`. The counter's state and its communication with an enclave are cryptographically secured. SMCs can be leveraged for versioning of sealed data: Whenever data is sealed, the corresponding counter value is included. When the data is unsealed, after e.g. a reboot, the stored counter value can be extracted and compared to the value obtained from reading 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 drawbacks to the current SMC implementations though [52]: (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 wears out after approximately one million write operations.

Side-channel susceptibility The small TCB of SGX has its advantages. Scaling performance with the processor's capabilities, 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 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" and further states that "preventing side-channel attacks is a matter for the enclave developer" [37]. There are two general types of SGX side-channel mitigation approaches. (i) One approach is to harden programs by rewriting the code at compile time and randomizing control flow at execution time. Consequently, monitoring side-channels is much more complicated for the adversary. These measures, however, are often susceptible to more versatile attacks at future points in time [66, 16, 53]. (ii) An alternative approach is to manually eliminate secret-dependent data paths entirely. This method can be very effective at times, but there are also many pitfalls with ensuing consequences [53, 63, 16]. Either way, SGX developers are left with the major task of evaluating the security implications and figuring out practical methods to mitigate 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 [35]. Actually, in efficient RSA implementations, all of the modular multiplications and squaring operations are performed using an algorithm of its own, called Montgomery Multiplication [54], which was itself target of side-channel attacks [61]. 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 [14]. The code in Algorithm 3 has in fact been exploited using

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
```

a variety of other side-channels such as power [44], cache [70], branch prediction [2] and even sound [32].

Digital side-channels We adopt the notion of digital side channels, as proposed in the work of Rane et al. [59]. 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 [59].

Chapter 3

Overview

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 deploys the DP-GBDT enclave on its servers. The enclave must be reachable by external insurance customers. At this point, 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.

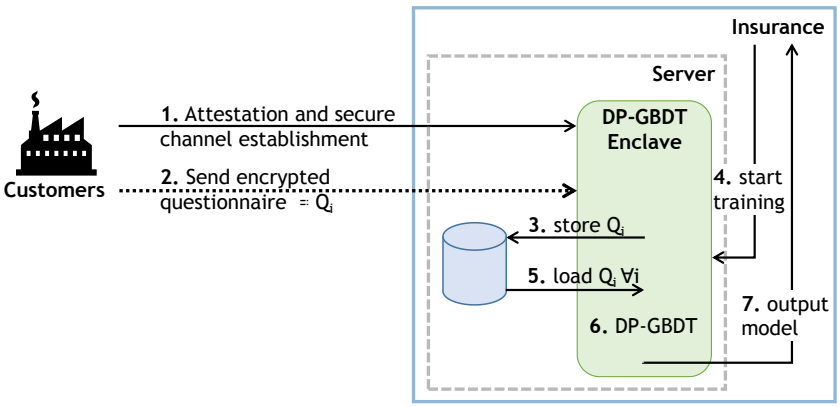

Figure 3.1: Insurance use case overview

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 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 Needless to say, we do not expect the insurance to be malicious. Yet, one of its servers could get compromised, or there might be a malicious insider/administrator. We model all such cases with threat model TM1: This adversary has full control over both the software (including OS or hypervisor) and the hardware of the 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 also 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. Lastly, he has full access to state-of-the-art enclave attack frameworks like SGX-Step [66]. With these techniques the adversary can obtain (i) the full memory access trace, and (ii) the complete algorithm control flow (program counter). The goal of the malicious service provider is to gather as much information as possible about customers participating in the training process.

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 he can initiate multiple session with the DP-GBDT enclave and submit his questionnaire more than once (but not to the extend where the sheer number of messages leads to denial of service). The objective of a malicious customer is to decrease 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 thus provide an isolated execution environment with remote attestation capabilities for any code running inside. Enclave-provisioned secrets also must not be accessible from untrusted code. 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

G1: User data privacy in GBDT Our first and primary goal is to provide privacy, more precisely ε-differential privacy, to customers providing their data to our learning algorithm. To achieve this for GBDT learning three things are required: (i) The algorithm has to be extended with DP modifications. DP guarantees that the existence of every single data point in the training data can be denied. For this reason, noise has to be added to the training data at specific locations in the algorithm. The background chapter contains more details on differential privacy (Section 2.3) and how it can be integrated into GBDT (Section 2.4). (ii) The implementation need to be ported into a trusted execution environment. A TEE, such as SGX acts as a shield around memory and processing, and thus isolates security-sensitive code from the untrusted software stack. Additional information on SGX functionality can be found in Section 2.5. (iii) As SGX enclaves are vulnerable to side-channel attacks, the implementation must be adequately hardened. Our goal is to eliminate all DP relevant leakage through *digital side-channels*. This is a notion that sums up all side-channels that carry information over discrete bits. Examples are the memory address traces, control flow and data size. (See Section 2.6)

G2: Efficient implementation As previously indicated, the starting point of this thesis was *not* zero. The DPBoost paper ([48], 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[2] (submitted Feb. 2021). Due to several weaknesses of the Python code, most notably speed, a new DP-GBDT implementation has to be built. It should meet the following requirements: First of all, and contrary to its predecessor, it should be free of bugs. In terms of prediction accuracy, it should be able to produce good 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, even though training will likely not be performed very frequently, the algorithm's runtime should be competitive.

G3: Secure deployment Even the most highly performing and privacy-preserving algorithm is useless if it cannot 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 previous output model
- Training on old data after output model loss
- Enclave replication and migration for maintaining persistent state

[1] https://github.com/giovannt0/dpgbdt
[2] https://github.com/loretanr/dp-gbdt/blob/master/code/python_gbdt/thesis.pdf

Chapter 4

Algorithm implementation

This chapter summarises the implementation work that was carried out to achieve Goal G2 of this thesis, an *efficient implementation*. This means absence of bugs, good forecasting accuracy and decent runtime. To facilitate future work building upon this thesis, key decisions and features are explained. Readers may skip this chapter if this is not part of their objective. The discussed topics include:

- High-level overview of the DP-GBDT algorithm
- Starting point
- Design decisions and implementation features
- Discovered bugs and fixes
- Transferring the code into an SGX enclave

High-level algorithm To facilitate understanding of the following sections, a brief rundown of the DP-GBDT algorithm is now presented. The core process consists the following steps: (i) The training samples are divided into disjoint subsets, each decision tree receives one subset. (ii) A decision tree can then be built using the following rules:

1. Recursively split the samples into further subsets by using the exponential mechanism (Theorem 2.4) to choose a split that offers a high information gain
2. If no viable split can be found or the maximum depth is reached create a leaf node
3. Add noise to all leaf values using the Laplace mechanism (Theorem 2.3)

(iii) Once the full ensemble of trees is built, predictions for new samples can be made by traversing the trees one by one. The background chapter expands on the details of decision trees (Section 2.1), differential privacy (Section 2.3), the combination of both (Section 2.4) and DPBoost (Section 2.4.1 and Algorithm 2).

4.1 Starting point

With DPBoost ([48], 2020) Li et al. provided the basis DP-GBDT algorithm for this thesis. The source code for their experiments is publicly available. However, the implementation is quite convoluted, as it is essentially a fork of the LightGBM [43] framework with modifications scattered accross numerous files. Therefore, Théo Giovanna subsequently built a standalone Python implementation[1] of said algorithm during the course of his master's thesis[2] (submitted

[1] https://github.com/giovannt0/dpgbdt
[2] https://github.com/loretanr/dp-gbdt/blob/master/code/python_gbdt/thesis.pdf

Feb. 2021). Unfortunately, this implementation also has its weaknesses, most notably speed. With the overlying goal of performing DP-GBDT in an SGX 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 a reference, and rebuild the algorithm from scratch 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 and to substantial runtime improvements. We further opted against using any existing C++ machine learning libraries, as it would have complicated the upcoming hardening process. As a result, numerous tasks, that only require a single line of code in Python, had to be manually replicated in C++. Performing cross-validation is one such example. The only library that was utilised is the C++ standard library [41], and the c++11 standard was chosen since it is the latest with SGX compatibility.

Project structure Our project[3] essentially consists of five DP-GBDT versions (Figure 4.1) and some additional infrastructure to run, verify and 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 final output of each variant is the same, equal input means equal output. For

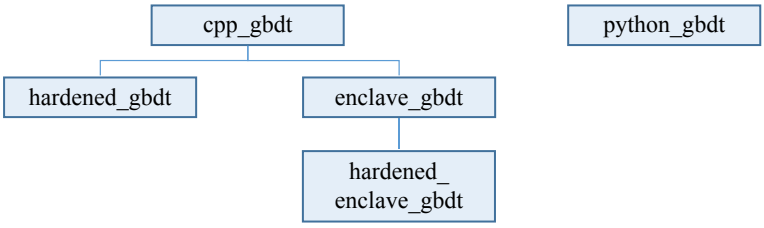

Figure 4.1: Project structure

the remainder of this section we mostly refer to `cpp_gbdt`, the non-hardened C++ version. It is roughly 2500 lines of code in total. The algorithm is clearly visible and the combination of compiler optimisations and multithreading leads to a 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 [48] paper, or in T. Giovanna's thesis[4]. The following features are *not* available:

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

[3] https://github.com/loretanr/dp-gbdt
[4] https://github.com/loretanr/dp-gbdt/blob/master/code/python_gbdt/thesis.pdf

This is of course unfortunate, but if required, they can 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 [25], YearMSD [26], Adult [27] and BCW [28]. To our knowledge, this is first time detailed and correct DP-GBDT results have been generated for these datasets. Further, it is very easy to add other datasets: Given a new dataset (in comma separated form), only about 15 lines of code in the `Parser` class are required to start training on it. An example of this can be found in Appendix A.4.

Comparability and verifiability This was a major challenge during development. Even though a reference implementation was available for comparison, there were multiple obstacles: (i) Naturally, being the first of its kind, the reference implementation contained multiple bugs (Section 4.3), that were only gradually discovered. It ranges from programming slips to relatively important parts of the algorithm that were misinterpreted. (ii) Next, Python and C++ libraries use different sources 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 selective deactivation of every single 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. (iii) Finally, there were numerical issues. It is commonly known, that working with floating point numbers can lead to rounding errors and other mathematical principles, such as associativity in multiplication, not holding anymore [36]. As a result, the C++ implementation frequently produced entirely different output results. Reason being, for instance, that somewhere deep in Python's `sklearn` [56] library, the order of certain arithmetic operations is different. Due to the nature of the DP-GBDT algorithm, there are multiple locations 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 the current tree that is being built. In the following stages, an avalanche effect will occur, resulting in all remaining decision trees in the ensemble looking completely different. Since these kind of issues are absolutely ubiquitous in DP-GBDT, the only viable solution was to regularly clip affected variables to about 12 decimals.

However, given these mitigations, it was possible to eliminate the differences between the Python and C++ implementation. To make future development easier, a verification framework was created. It allows running both Python and the (un)hardened C++ code on a number of different datasets. Intermediate values are continuously compared during the execution of the algorithms. Thus, programming mistakes can be quickly identified given the exact location where the implementations start diverging.

Logging and documentation fmtlib[5] was chosen for logging. Different logging levels can be chosen to obtain output of the desired degree. Additionally, finished decision trees can be printed to the terminal. Doxygen[6] was used to generate code documentation and help programmers quickly getting an overview of the different components of the algorithm.

[5] https://github.com/fmtlib/fmt
[6] https://github.com/doxygen/doxygen

Runtime improvements There are around 20 hyperparameters (see Appendix A.2 for details) in the GBDT algorithm that need to be optimised to achieve good runtime results. Being a compiled language, C++ was naturally already quite fast compared to Python. Additional optimisations that were conducted include multithreading, aggressive compiler optimisations and removing blockers in core/critical functions to incentivise vectorisation. To identify bottlenecks, frequent profiling was done using Intel vTune[7] and Intel Advisor[8]. For illustration purposes you can find the profiling output of (i) the finished (optimised) C++ GBDT algorithm, and (ii) the hardened code in Appendix A.5.

4.3 Discovered bugs and fixes

Python bugs The following issues were identified (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` wrong formula

The idea behind tree rejection is the following: In normal GBDT (no differential privacy) the predictions get closer to their actual value with every single decision tree. However, due to the random noise 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 is a good/useful tree. Bad trees are rejected accordingly and are not part of the final model. This rejection mechanism clearly violates differential privacy. More precisely, the repeated usage of the same samples would either require repeated payment of privacy budget, or alternatively, these samples would need to be split into disjoint subsets, such that they are not reused. This rejection mechanism may be worth exploring however, since it improves prediction performance quite significantly. This was discussed with E. Mohammadi and M. Kirschte, who will likely pursue this idea further in the future. Figure 4.2 shows the vast performance 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.

Grid usage Strictly speaking, this paragraph is not about a bug, but rather an important aspect that was not addressed by the DPBoost [48] paper. When trying out different splits to find the one with the highest gain, iterating over the real feature values that are present in the dataset is a bad idea. Differential privacy itself is *not* violated, but information about which feature values are present in a dataset can be leaked. This can then be used narrow down the range of the affected feature values. Imagine a scenario where all insurance customers answer the first question of a questionnaire with answer $a = A_1$ or $a = A_2$. A naive implementation could leak this information, that only two different answers are present, to a side-channel observer. If the observer further learns the final model, which is an assumption of our differential privacy proof, and both values

[7] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
[8] https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html

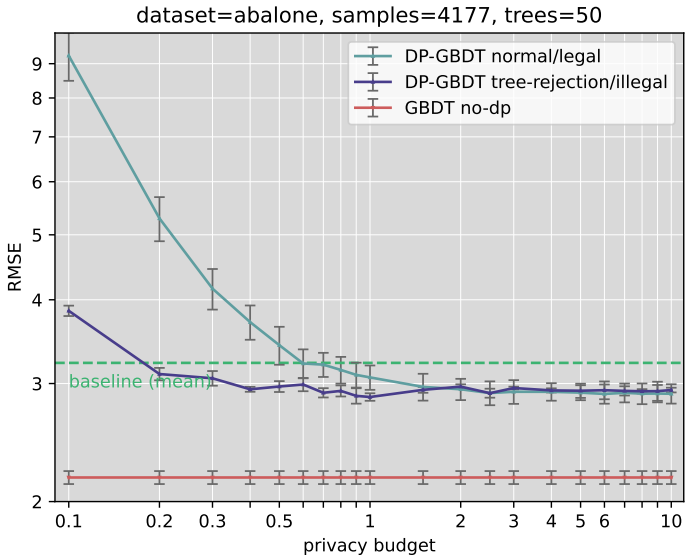

Figure 4.2: Comparison: Tree-rejection vs correct DP-GBDT

were chosen for splitting at some point, the observer knows all customers answered either A_1 or A_2. Solving this problem generally requires adding some form of dummy loop iterations to the process of trying out different splits. For categorical features the problem can be addressed by trying out all possible feature values, even though the values might not be present in the dataset. For numerical features, a grid with constant end points and a fixed step size can be used. Given a small enough step size, 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 value is passed.

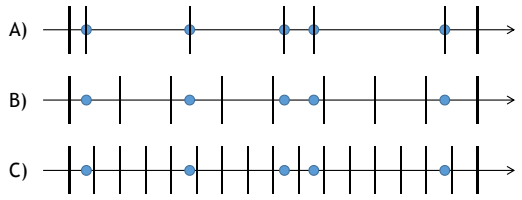

Figure 4.3: Grid step size: A) split on feature values, B) not fine enough, C) catches all splits

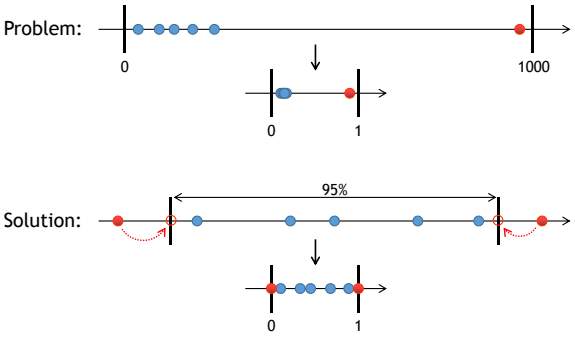

Figure 4.4: Scaling feature values into the grid using the 95% confidence interval

The challenge of such an approach is defining the constant grid borders. As different numerical features can be of vastly different magnitudes, each feature would need its own specific grid to prevent an unreasonably large amount of runtime overhead. To avoid going down that path, we propose a more versatile solution, *feature scaling*. If all numerical feature values of the entire dataset are in e.g. [0,1], the only thing left to be defined is a suitable step size. Unfortunately, there is yet another issue: Consider a feature A, whose values submitted by legitimate customers

usually fall within the range [0,100]. A insurance customer M with malicious intent could intentionally submit false 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 the upper half of 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 (see Figure 4.4). The challenging part is doing this in a differentially private manner. Fortunately, Du et al. [22] propose multiple options on how to achieve this and further provide corresponding code[9]. For this thesis the EXPQ method was adopted from the aforementioned work. According to the paper, it is not the most efficient scheme in terms of privacy budget, but it was the simplest one to understand and port to C++. Of course, the price for this is a small loss of information in your training data, as well as a small amount of privacy budget. In fact, our experiments showed, that this way of scaling feature values into the grid works reliably for $\varepsilon = 0.03$. In the final implementation grid-usage can be selectively enabled. If enough information about the values in a dataset is known, constant grid borders can be used instead.

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 be discussed 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 [38]. An important step in SGX application development is dividing code into an inside and outside of the enclave part. `App.cpp` contains the outside code, while `Enclave.cpp` contains the enclave-internal code. This separation is specified in the `Enclave.edl` file, where all function calls that are allowed to cross the enclave border are listed. In our case there are four calls in total (see Listing 4.1): Three of them are ecalls, which means they are directed into the enclave. Each of them is called exactly once. Two are responsible for carrying the dataset resp. the model parameters into the enclave. The remaining one does not carry any data and just initiates the training process.

[9] 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

The only ocall and thus way to get data out of the enclave is currently `ocall_print_string`, which is used to print the final cross-validation performance. This is sufficient to see that the algorithm works.

Parsing a dataset requires various I/O functions that are not readily available inside the enclave. Hence, parsing (as well as defining the model parameters) is easiest to be taken care of on the outside. Since the interface between inside and outside 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.

Since obtaining randomness inside SGX is special, the DP-GBDT algorithm itself had to be adjusted everywhere where randomness is used. SGX provides a function `sgx_read_rand()` which executes the `RDRAND` instruction to generate hardware-assisted unbiased random numbers [42]. Apart from replacing calls that fetch random numbers, several functions, that were previously not much more than a simple std::library call, had to be rewritten from scratch. For example randomly selecting or deleting a subset of rows from a matrix. Of course, external library code for e.g. logging capabilities had to be removed as well. Similarly, no more multithreading is available. Further, there is an important configuration file `Enclave.config.xml` which specifies settings like available stack and heap memory size. Appropriate modifications are indispensable for larger datasets. More information, on compilation for instance, can be found in our repository[10].

[10] https://github.com/loretanr/dp-gbdt/tree/master/code/enclave_gbdt

Chapter 5

Side-channel hardening

This chapter summarizes the DP-GBDT hardening process, which is required to achieve Goal G1 of this thesis, *user data privacy*. As a first step, the goals are reiterated. Subsequently a selection of known hardening tools and techniques is presented. Thereafter, our chosen approach is explained and justified. This is followed by a final section containing additional details and code examples of our hardening efforts.

Goal The aim is to provide ε-differential privacy to customers providing their data to our learning algorithm. This requires protecting the DP-GBDT implementation from leaking secrets through *digital side channels*. As described in Section 2.6, this covers all side-channels that carry information over discrete bits. Examples are the memory access trace, control flow and data size.

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. Afterwards, different complications caused by compilers are analyzed. We conclude with a brief overview of tools for automatic hardening and verification.

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 is 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 [31]. Additional issues can arise from gaps in the language standards. The C-standard, for instance, generally does not 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 [15, 62].

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 [34, 64], a non-oblivious algorithm can be compiled to its oblivious counterpart at the expense of a poly-logarithmic amortised 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, ORAM is usually 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 algorithms 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, 17]).

5.1.1 Compilation and verification

How can a programmer be sure that the compiler does not optimise away his hardening efforts, or create new side channels on its own? Most high-level programming languages do not provide an element that enables specification of desired execution time properties. 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, where 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 instead. 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 hard to understand for the compiler. Otherwise, unwanted and possibly side-channel introducing optimisations have 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 degree. Nevertheless, it is important to

```
int test(bool b1, bool b2)
{
    bool result = b1 and b2;
    return result;
}
```

```
test(bool, bool):
    andl    %edi, %esi
    movzbl  %sil, %eax
    ret
```

```
test(bool, bool):
    testb   %dil, %dil
    je      ..B1.3
    xorl    %eax, %eax
    testb   %sil, %sil
    setne   %al
    ret
..B1.3:
    xorl    %eax, %eax
    ret
```

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

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 several automatic hardening tools, e.g. Constantine [12], Obfuscuro [3] or Raccoon [59]. The offered approaches include: Complete linearisation of secret-dependent control and data flows, or execution of extraneous "decoy" program paths. Depending on the exact approach, the automatic hardening tools introduce substantial runtime overhead. Tool-assisted verification of constant-time properties is also possible [65, 60, 7]. However, most of these methods demand meticulous program code instrumentation, to e.g. mark variables or memory locations that might contain secret values at some point.

5.2 Chosen approach

This section summarises 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 a runtime explosion due to automated tools 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 and 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 followed: (i) Avoid branching 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,

and (v) clean up memory of secret data. 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 called for by our goal to eliminate leakage through digital side channels, we have to remove 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 has to 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 arithmetics, we decided against replacing them entirely with their corresponding constant-time fixed-point counterpart. It is not only a very time consuming task, it also heavily perturbs code readability. We opted for a compromise: we apply this transformation solely to the core functions of the code, where around 95% of time is spent (see Appendix A.5). We use libfixedtimefixedpoint[1], a state-of-the-art library for constant-time mathematics, recommended by multiple research papers [8, 46].

5.2.2 DP-GBDT-specific measures

We would like to direct the reader to Chapter 4 for a high-level overview of the algorithm. Even more detailed information can be found in Section 2.4 and Algorithm 2. There are mainly 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-based 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 the above mentioned points. Later on, in the final paragraph of this section, we cover fully-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 be 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-private 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 are 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 or right. (ii) We must not leak the number of splits that yield a sub-zero 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 which is, of

[1] https://github.com/kmowery/libfixedtimefixedpoint

course, set in an oblivious manner, is passed along to indicate whether a node lies on the real path. In this manner, 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 the formula on line 8 in Algorithm 2 of DPBoost [48]. 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 is sufficient to have indistinguishability in whether a sample was chosen randomly or because of its favourable gradient.

Fully-oblivious tree construction As mentioned, hiding the entire tree building process is not required to ensure ε-differential privacy. It is an extra hardening step that was performed during this thesis. It is still useful since it might allow for a tighter proof in the future.

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 [47]. This means that full binary trees have to be built. 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 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 decision tree hardening process, appendix A.6 shows a pseudocode example of the gain computation function with all side-channel affected regions highlighted.

5.3 Technical details

This section offers more detailed insights into the hardening efforts. We first expand on how compilation and verification was conducted. Afterwards, several oblivious building blocks and code samples are presented.

Compilation and verification The newly created constant-time functions were all defined in a separate file and namespace. Further, function inlining has been disabled with vendor-specific keywords. In this way, the assembly output of the hardened functions can be more conveniently checked as all methods are gathered in one place. Another extra layer of safety measure that we added to the hardened functions is the selective use of the `volatile` keyword. Its purpose is essentially to prevent unwanted compiler optimisation. Regarding compiler optimisation flags, we decided to take a relatively conservative approach: We opted against using very aggressive compiler optimisations, as it significantly lowers traceability and comprehensibility of the assembly code. Compilation results were inspected after usage of both `-O0` and `-O1` `gcc` flags. No side-channel violations were discovered.

Logical operators Logical boolean operators can usually be directly replaced by the corresponding bitwise operators. The logical `NOT`, for instance, can easily be implemented as an `XOR` with 1, as shown in Listing 5.1. The `value_barrier` function is a wrapper for transforming variables to `volatile`.

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

Listing 5.1: Constant-time logical `NOT`

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 subsequently choosing the result with the constant-time ternary operator. Listing 5.2 contains the corresponding code.

```
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 5.2: Constant-time 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 bubblesort approach because it is very easy to harden. The fact that its runtime is in $O(n^2)$ is not a problem, since sorting is not a performance critical task in our algorithm.

```
// 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 5.3: Oblivious sort

Final example For illustration purposes, a hardening before-and-after is depicted in the following paragraph. The underlying code is a slightly simplified version of the prediction function. In other words, this is the function that samples use to traverse a finished decision tree and fetch the corresponding leaf value. Listing 5.4 shows the non-hardened version, and Listing 5.5 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 5.4: 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 5.5: Hardened inference/prediction

Chapter 6

Evaluation

This chapter gives an overview of the achieved results. After covering the methodology and experimental setup, the DP-GBDT prediction performance results are presented. Thereafter, the runtime of the different implementations is compared. This gives an image of the impact that individual hardening measures have. The sections are each concluded with a brief discussion.

6.1 Methodology

We demonstrate the performance of the algorithm on four different UCI-standard 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 [25]	4177	8	regression	no
yearMSD [26]	515345	90	regression	yes
BCW [28]	699	10	binary classification	no
adult [27]	48842	14	binary classification	no

Table 6.1: Used datasets

Experimental Setup The machine that generated the 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 used samples varies amongst the different datasets. The result values are 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 in the plots denote the average measured standard deviation. The regression baseline is achieved by always predicting the average/mean feature value. For binary classification, the baseline is attained through the 0R-classifier (we always predict the majority). For regression tasks both RMSE (penalises errors on outliers) and MAPE (more robust to outliers) are provided.

Hyperparameters In Appendix A.3, we list the hyperparameters used to generate the following results. The `use_grid` functionality (described in Section 4.3) is turned off, since it leads to virtually the same results, given a small enough step size, while substantially increasing runtime. In fact, experiments showed, that our way of scaling feature values into the grid works reliably using a privacy budget of just 0.03. If desired, there are even more cost-efficient ways to perform this task [22]. 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 The goal of this dataset is to predict the age of abalone snails based on measured physical attributes such as diameter and shell weight. The full abalone dataset (4177 samples) can be easily processed. By always predicting the mean of the target feature, we obtain a baseline RMSE of 3.22 and a baseline MAPE of 26.5%. Figure 6.1 shows that all DP-GBDT variations

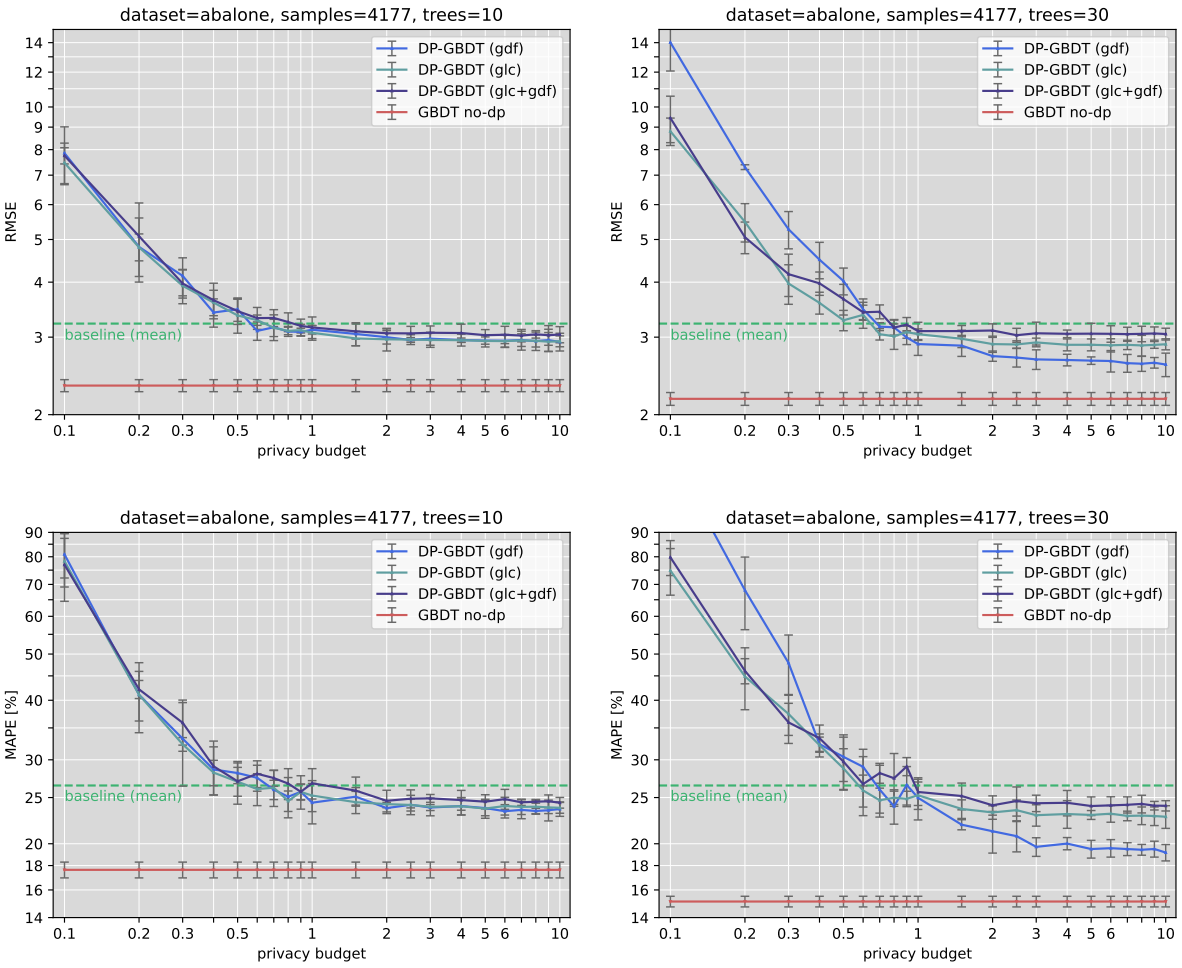

Figure 6.1: Abalone RMSE (top) / MAPE (bottom): Left-hand side ensemble of 10 trees, right-hand side ensemble of 30 trees

start beating the baseline at around $\varepsilon = 0.7$. There is a clear improvement in performance when 30 instead of 10 decision trees are used in the ensemble. The right-hand side plots further show that

using gradient-based data filtering (GDF) leads to worse performance for small privacy budgets, but better performance for higher ε's. At no point however, do the DP-GBDT implementations get reasonably close to the non-DP performance. Geometric leaf clipping (GLC) does not seem to improve performance in any form. As both RMSE and MAPE look very similar, it appears that outliers are forecasted fairly effectively.

yearMSD The goal of this dataset is to predict the release year of songs from their audio features. Since the 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", thus no song from a given artist ends up in both the training and the test set. Figure 6.2 shows that using a subset of 10k samples is just not enough to reliably beat the baseline.

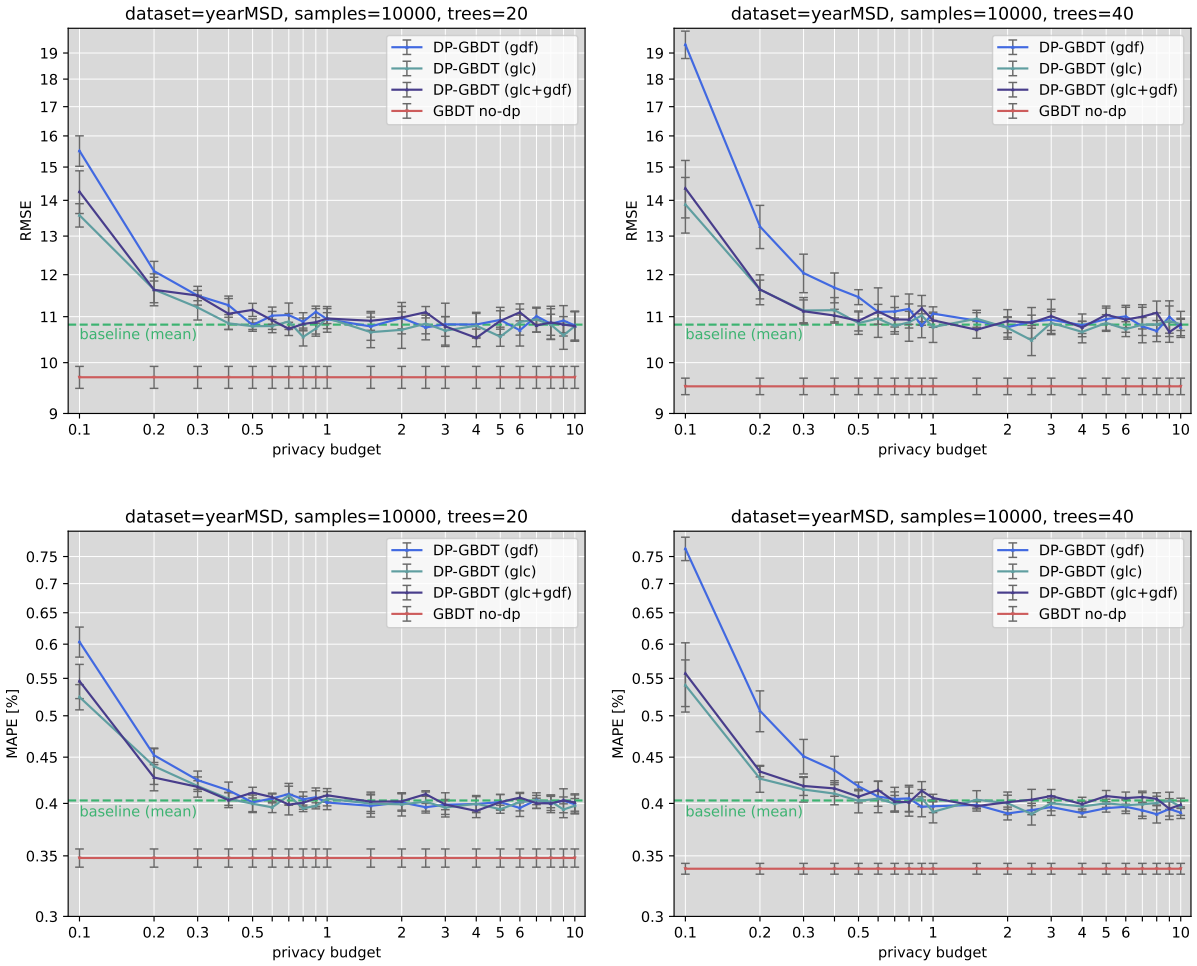

Figure 6.2: YearMSD RMSE (top) / MAPE (bottom): Left-hand side ensemble of 20 trees, right-hand side ensemble of 40 trees

Creating ensembles with 40 instead of 20 trees shows no significant impact. As with the previous dataset, GDF usage tends to negatively affect performance for small privacy budgets.

Breast Cancer Wisconsin This dataset is used for prediction of whether cancer cells are benign or malignant according to features of the cell nucleus. Due to the small size of this dataset, the results were not as stable as those from other datasets. This is evident from the wider errorbars. Consequently, cross-validation was repeated 10 times, and the results were averaged. Figure

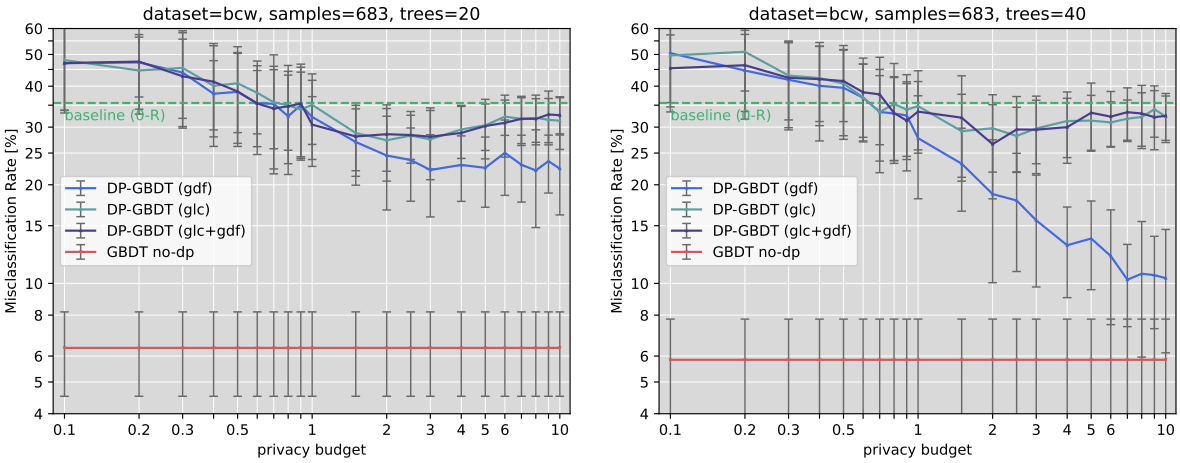

Figure 6.3: BCW misclassification rate: left side ensemble of 20 trees, right side 40 trees

6.3 shows that all DP-GBDT variations start beating 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 around $\varepsilon = 3$. Afterwards, their results deteriorate. Using 40 instead of 20 decision trees mainly improves the performance when using GDF. Once again, GLC does not seem to increase performance in any significant manner.

Adult This dataset can be used to predict whether a person's annual income exceeds \$50k according to census information. Error rates from other, (Non-DP) machine learning algorithms typically range from 14-20% [45]. Figure 6.4 shows that the DP-GBDT algorithm does not really perform well on this dataset, at least not with these specific hyperparameters. Only the GDF variant manages to beat the baseline, though 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 strong 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 distributes more samples to earlier trees in the ensemble, while the balanced version divides samples evenly amongst all trees. 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 is interesting that, unlike DPBoost [48] proclaims, GLC did not improve the overall performance on any of our datasets. The performance was notably worse for the adult dataset. It is 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 [51], which is the case for the adult dataset (25% / 75%). Balancing the dataset prior to fitting might improve this result.

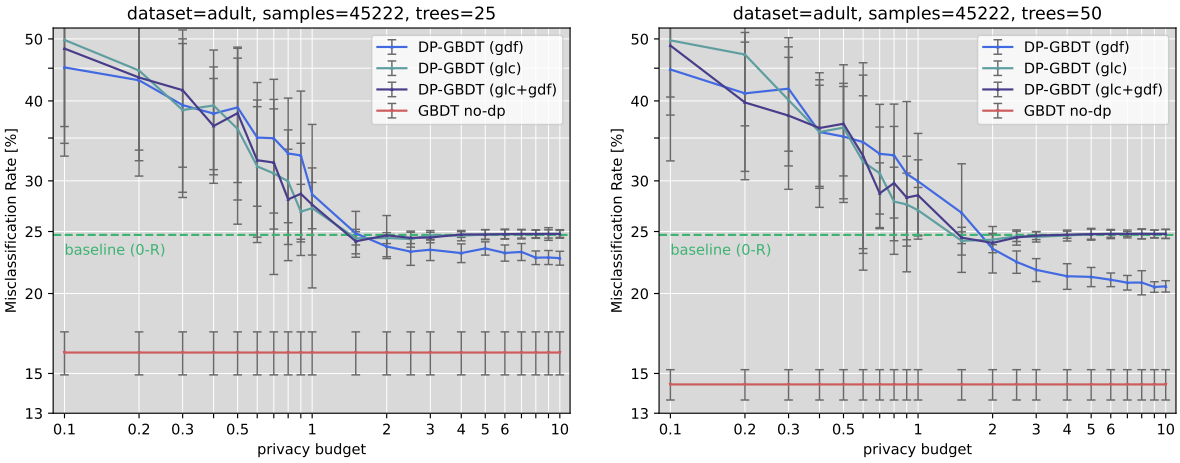

Figure 6.4: Adult misclassification rate: left side ensemble of 25 trees, right side 50 trees

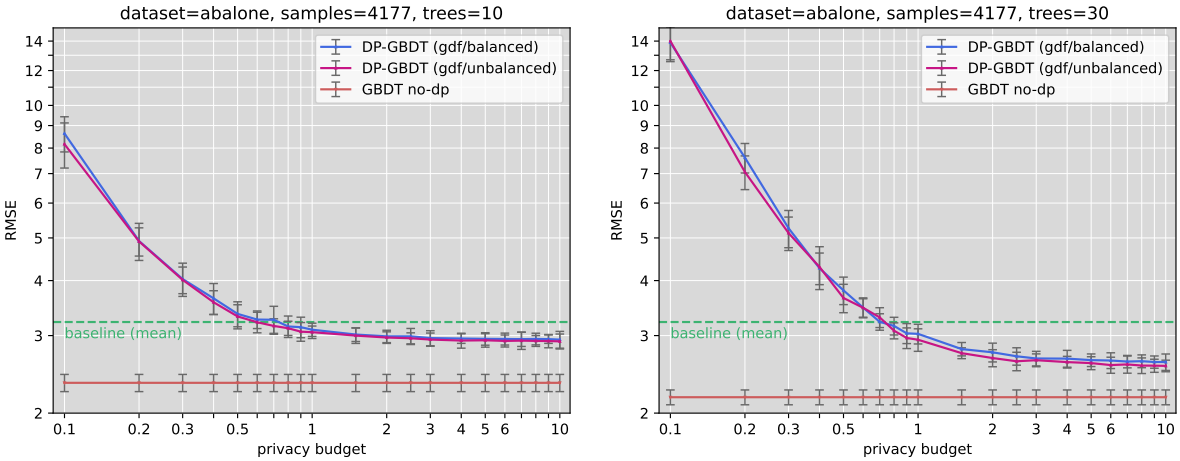

Figure 6.5: Abalone RMSE: Balanced/unbalanced comparison: ensemble of 10 resp. 30 trees

In general, performance for small privacy budgets is quite bad. We seem to be adding so much noise, that using trees with a maximum depth of 6 is more harmful than beneficial. Hence, 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, further improvements are necessary. Also keep in mind, that the current design's performance can certainly be significantly increased by well-versed hyperparameter tuning. In fact, at the time of writing, Moritz Kirschte informed the author that he had achieved better results for certain datasets by using trees of depth one and adaptive thresholds.

6.3 Runtime

Next, we show the variations in training time between the different implementations. Although the DP-GBDT algorithm will likely not be run very frequently in practice, achieving good runtime performance was a key factor for us, and also factored into the choice of a manual hardening approach. Table 6.2 shows the utilised model hyperparameters for the runtime

measurements across all implementations. All runtime results were obtained by performing

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

5-fold cross-validation. Therefore the measured time periods have been divided by 5 in Table 6.3. Implementations inside SGX enclaves were built in pre-release hardware mode. We will highlight the differences between the individual versions first:

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

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

enclave_gbdt: This is essentially `cpp_gbdt` but ported into an SGX enclave. The necessary changes have been described in 4.4.

hardened_gbdt: This is the hardened version of `cpp_gbdt`. It reflects the minimum changes that are needed to achieve ε-differential privacy. Section 5.2 described this process in detail. The code does not run inside an SGX enclave. Even though no side-channel violations were found when compiling with the `-O1` flag, we included the `-O0` measurement to be on the safe side.

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

Implementation	**inside SGX**	**abalone**			**adult**			**yearMSD**	
		300	**1000**	**4177**	**300**	**1000**	**5000**	**300**	**1000**
python_gbdt	no	1.52	2.36	15.2	1.50	3.42	41.8	9.46	88.3
cpp_gbdt	no	<0.01	0.01	0.05	<0.01	0.01	<0.01	<0.01	0.12
enclave_gbdt	yes	0.01	0.04	0.22	0.01	0.03	0.13	0.08	0.43
hardened_gbdt -O0	no	0.26	2.86	52.9	0.30	2.86	69.4	2.52	33.0
hardened_gbdt -O1	no	0.04	0.42	8.02	0.04	0.42	9.84	0.36	4.70
hardened_sgx_gbdt	yes	0.08	0.84	16.7	0.13	1.06	26.6	0.78	9.44

Table 6.3: Runtime results [s]

First of all, compared to the Python reference implementation, our new C++ implementation, `cpp_gbdt`, offers a 506x speedup on average. The speedup increases further with even larger datasets. This is because finding good splits is the bottleneck in training (it is where roughly 95% of runtime is spent, see Appendix A.1). With increasing dataset size, this task grows linearly by a factor n, where n is the number of features. Moving `cpp_gbdt` into an SGX enclave, decreases its runtime by 3.3x on average. The lack of multithreading (2 physical cores) and additional SGX overhead, such as enclave calls and encryption, certainly play a role in this. `hardened_gbdt -O1` is on average 77x slower than `cpp_gbdt`. Turning off compiler optimisations (compile with `-O0`) further increases the runtime by 6.9x on average, which results in a total average slowdown of 534x compared to `cpp_gbdt`. At last, `hardened_sgx_gbdt`, the union of hardened DP-GBDT

and operation inside an enclave, is as expected slightly slower than `hardened_gbdt -O1`. It increases the runtime by 2.3x on average.

Next, Table 6.4 shows the runtime consequences of (i) using constant-time fixed-point arithmetic instead of inherently non-constant-time floating point operations. As explained in 5.2.1, we only replaced affected floating point operations in the core functions of the algorithm, where around 95% (see Appendix A.5) of execution time is spent. (ii) Oblivious tree construction (see 5.2.2): By turning this part of the algorithm into oblivious code, no information about the shape or content of the decision tree is leaked through side-channels. This is not necessary to achieve ε-differential privacy, but might allow for a tighter proof in the future. Tighter bounds essentially means that less noise has to be added to certain sections of the algorithm, which in turn allows spending more privacy budget on other tasks. An overall performance increase would be the result. The results show that using constant-time fixed-point numbers instead of

Implementation	**inside SGX**	**abalone**			**adult**			**yearMSD**	
		300	**1000**	**4177**	**300**	**1000**	**5000**	**300**	**1000**
h_1 := hardened_gbdt -O1	no	0.04	0.42	8.02	0.04	0.42	9.84	0.36	4.70
h_1 + libfixedtimefixedpoint	no	1.18	12.3	220.0	1.44	12.6	278.4	12.7	150.1
h_1 + oblivious_tree_building	no	0.18	1.6	22.0	0.32	2.46	59.5	2.0	16.2

Table 6.4: Runtime implications of using (i) constant-time fixed-point arithmetic, and (ii) oblivious tree construction [s]

floating point numbers substantially increases runtime (31x on average) compared to the baseline hardened implementation. Constructing the decision trees obliviously, i.e. always recursing to the maximum depth and hiding what kind of node/split is created, results in a 6x execution time penalty.

6.3.1 Discussion

As a first step, we produced a very fast baseline implementation, ideal for future experiments. As the results show, hardening then imposes a lot of execution time overhead. Especially, in the case where an "all-inclusive" hardening approach is taken. By opting for just ε-differential privacy, better runtimes are achieved. Nevertheless, even for datasets with thousands of samples, the learning process is a matter of seconds/minutes. This confirms the feasibility of the algorithm for future usage in practice. In hindsight, also a few weaknesses resp. potential for improvement became apparent. For example, using a BFS instead of DFS tree induction strategy 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. Another takeaway is that using constant-time fixed-point numbers significantly decreases performance. The operations themselves but also the conversion back and forth is expensive. It should be noted that this is a state-of-the-art fixed-point library that is recommended by several papers [8, 46]. Finally, there are likely multiple factors that contribute to the fact that the same implementations were 2-3x slower inside SGX compared to outside. (i) The SGX version of the C++ standard library is missing certain functions, such as randomly shuffling the content of a vector. As required by DP-GBDT, these functions had to be recreated. The result is quite likely not as fast as the corresponding library version. (ii) The algorithm constantly requires random numbers. Obtaining such might be slower from within the enclave. (iii) There is an initial ecall and a final ocall that may add a small amount of overhead.

Chapter 7

Secure deployment

This chapter addresses Goal G3, and thus gives a high-level overview of the challenges that accompany the real-world insurance use case. We start with a short recap of 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 chapter.

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. There are essentially two operations that can be performed on such a counter after it has been initialised, `Read()` and `Incr()`. The counter's state and its communication with an enclave are cryptographically secured. Additional details can be found in Section 2.5. As described in Section 3.2, we use two different threat models to cover the 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 Consider a scenario, where the insurance has successfully prepared and started the DP-GBDT enclave on its servers. The questionnaire data collection works as intended, and after a while the insurer instructs the enclave to train a model using the first 500 questionnaires. However, while the model output is being examined, 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 2.5 for sequential composition of DP functions, privacy budget would have to be paid twice for this to be allowed. To address this threat persistent state is added to the DP-GBDT enclave by 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, $\mathcal{A}$ 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. The server containing the enclave could break or crash at the worst possible time, for example during training. Due to its persistent state, the enclave would refuse to train the model again. Hence, all previous progress would be 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, the following issues need to be addressed:

- 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 in case the output model was lost
- Enclave replication and migration while maintaining persistent state
- Prevention of enclave state rollback or fork attacks

7.1 Design

This section is divided into three parts. The first part elaborates on the enclave setup. This includes necessary preparation steps as well as description of the client application and enclave state. Second, the core operations, namely data collection and training, are explained. At last, enclave migration and replication possibilities are explored.

7.1.1 Enclave setup

Numerous settings need to be defined and hardcoded in the enclave source code before attestation and operation. 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.

Defining the training points in advance serves two purposes: First, 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. As indicated before, this would violate DP if privacy budget is not paid accordingly.

Client application and client-server communication The client application is fairly simple and allows the customer to fill out the questionnaire about its IT-security policies. Upon completion of the questionnaire, attestation between the client and the enclave is performed. 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 sends the encrypted questionnaire answers to the trusted application. The encryption key is a shared session key that can be established during attestation [40]. Further, the client application ensures 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.

Malicious customers A customer with malicious intent could, for example, send 100 questionnaires to the enclave instead of one. From a differential privacy point of view this is not a problem. On the contrary, more questionnaires generally means better privacy, regardless of their content. The problem is that such actions could decrease the output model's expressive power. This could be prevented by the following two options: (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 first start 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, the steps in Figure 7.2 have already been conducted. As a result the enclave then goes into collection resp. waiting mode. This is the default state which is only left upon reception of a `start_training()` command issued by the insurance.

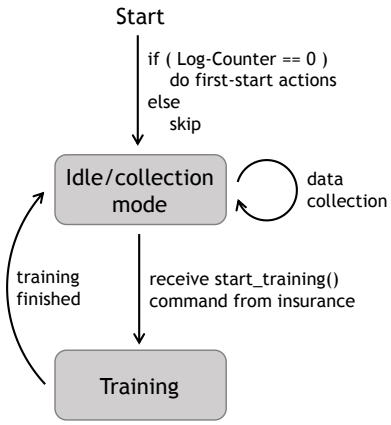

Figure 7.1: Enclave state diagram

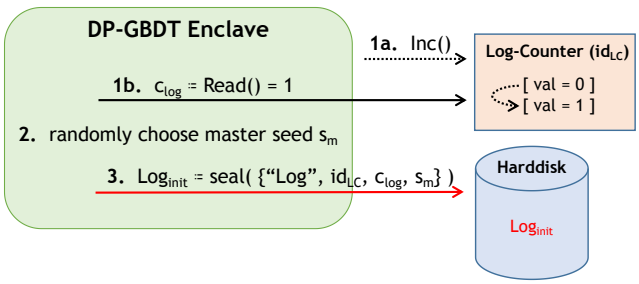

Figure 7.2: First start of the enclave

The actions, that an enclave performs on its very first start are depicted in Figure 7.2. Note that the secure monotonic counter was already initialised in the setup phase and its ID is hardcoded into the DP-GBDT enclave's source code. Regarding step (2.): The master seed s_m has two main purposes: (i) 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 [39, 40]. (ii) Second, as indicated before, we offer the functionality to re-compute a lost model. To guarantee differential privacy in that case, the exact same model as obtained from the previous training need to be produced. Hence, the exact same randomness must be used in the DP-GBDT algorithm. This can be achieved 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. Data collection stays essentially the same, irrespective of the number of questionnaires that were already collected. With regard to training, however, we 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 Figure 7.3 gives an overview of how receiving and saving customer data is conducted from the enclave's perspective. Remarks: (1.+2.) As already mentioned,

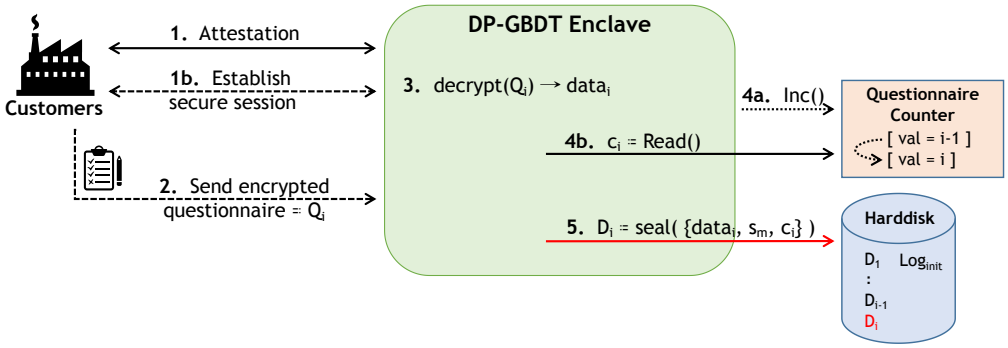

Figure 7.3: Data collection

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.) 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 DP 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 piece of data. In that case, all progress would be lost and we are back at the very start, i.e. even 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. The `load`* primitive in the diagram is an abbreviation for the following tasks:

- Unsealing the data from disk
- Performing different checks to recognise if 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), and (ii) checking whether all D_i contain the master seed s_m from Log_{init}
- Aborting the training in case any of the previous checks fail

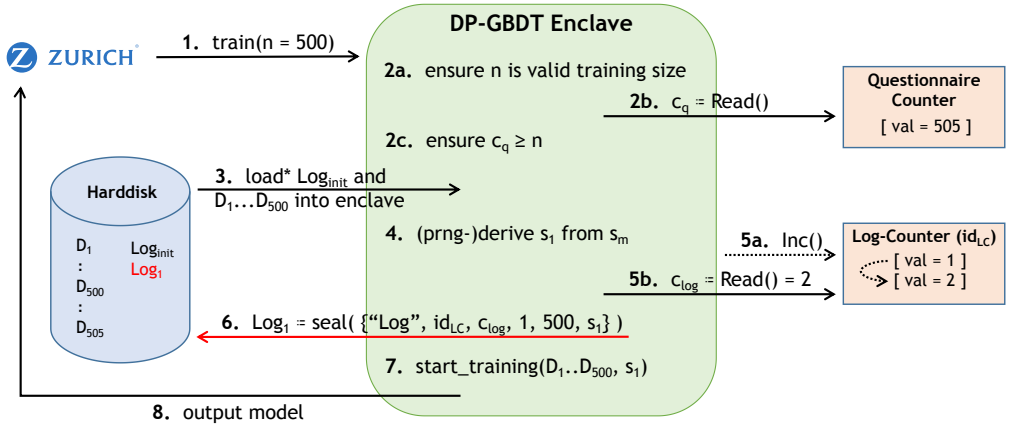

Figure 7.4: First training

Remarks: Generally, as the log-counter is already increased by one during the first-start procedure (Figure 7.2), it is always one step ahead. Once the first training is completed it reads a value of 2, once another refinement training is completed it reads 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 is going to be used during 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 only consider option (ii), as the first option is still under research. Another reason is, that this approach does not require an extra payment of privacy budget, since we operate on fresh data. The process is visualised in Figure 7.5.

Model re-training Figure 7.6 depicts how previously trained parts of the output model can be recreated. Remarks: (3.) As described in Section 7.1.1, the mapping (700→Log_2) is hardcoded in the enclave's source code.

7.1.3 Enclave migration and replication

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

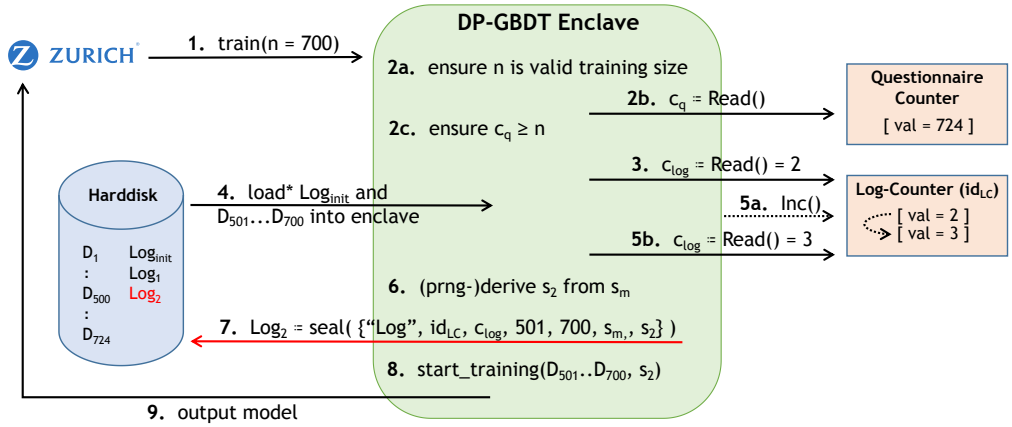

Figure 7.5: Refinement training

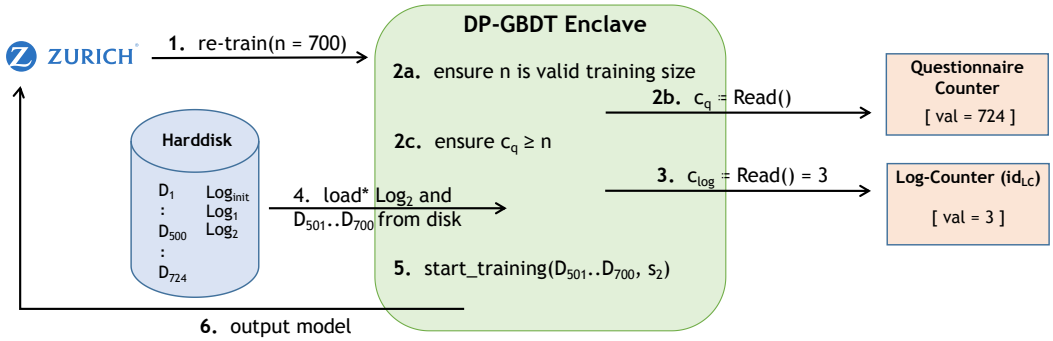

Figure 7.6: Re-training

Enclave migration Alder et al. [5] propose an enclave migration approach that is now briefly summarized. It guarantees the consistency of an enclave's persistent state, such as sealed data and monotonic counters. This is achieved by using a migration library to provide the sealing and monotonic counter functions with extended functionality. Further, a separate migration enclave finds application 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:

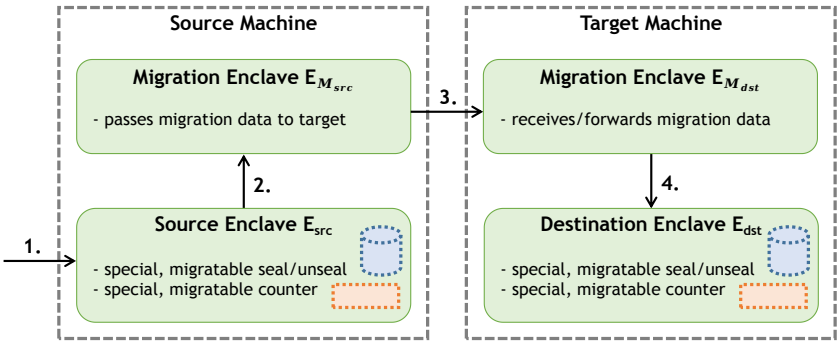

Figure 7.7: Enclave migration process[5]

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, as the machine containing enclave and counters might break unexpectedly. 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 exact clones. As they reside on separate physical platforms, they use separate monotonic counters. In a secure environment the DP-GBDT enclaves and the master enclave are then started. When started for the very first time, each enclave generates a key pair for asymmetric encryption. The master enclave takes over the part of generating a master seed s_m. Each replica subsequently performs attestation with the master enclave. During attestation, the identities are validated through source code hashes. Each DP-GBDT enclave transmits its individual public key and receives the master seed s_m in return. The data collection phase now looks a bit different (Figure 7.9): Insurance customers do not directly communicate with the DP-GBDT enclaves anymore, but with the master enclave instead. Similarly, attestation is now carried out between the master enclave and the customers. Upon reception of a customer questionnaire, the master enclave performs a decryption and re-encryption step and subsequently distributes the data to the DP-GBDT enclaves.

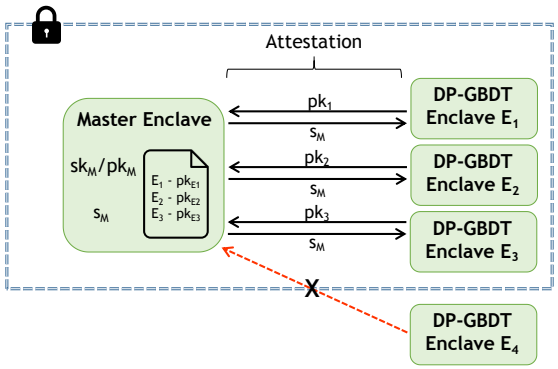

Figure 7.8: Replication setup phase

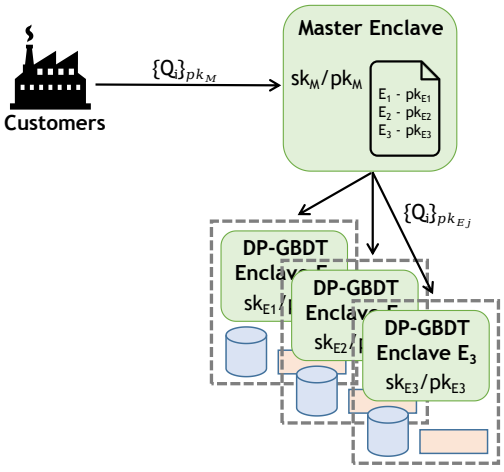

Figure 7.9: Data collection in a replication setting

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 [21]. Regardless of whether a secure/insecure environment is chosen, the goal is 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 data collection and distribution process between master and slave enclaves. Imagine a scenario where an adversary $\mathcal{A}$ cuts the connection to half of the enclaves while one questionnaire is being distributed. Subsequently, for the next questionnaire, $\mathcal{A}$ 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 the help of a monotonic counter. The counter value is incremented and included when forwarding questionnaires to the slave enclaves. Thus, slave enclaves can detect missing information and react accordingly. The drawback of such an approach is that the master enclave becomes a very critical component. In case this link breaks, the insurance would be left with a partially working setup: The individual DP-GBDT could still train on previously received data, but no new data could be collected and distributed anymore. Therefore, this approach would call for appropriate replication of the master enclave. Bezerra et al. [10] propose a similar solution for state-machine replication. (ii) The other solution for more broad fault tolerance would be to introduce some form of consensus/multi-party agreement protocol between DP-GBDT enclaves. Upon detection of tampering, slave enclaves could exclude themselves from further participation. Even the existence of a master enclave would essentially become unnecessary. The obvious drawback is the significant increase in complexity.

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, the system is 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 [23, 49, 24]. 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. [29]. 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 [48, 1, 50, 67] have been proposed since then. The combination of DP-GBDT and SGX enclaves has not been as thoroughly researched however. To our knowledge, there are only 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]. The high-level goals and architecture of Allen et al. 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 a TEE-based setting. The authors assume that the leakage only consists of (i) the output model and (ii) memory access trace. In this setting it is ensured that DP guarantees hold for three selected algorithms. Decision trees, however, are not among them. In general, the focus lies more on the algorithmic side, which means not an entire system with data provisioning from users, disk as persistent storage etc. is considered.

Law et al. (2020) "*Secure collaborative training and inference for XGBoost*" [47]. This work presents a privacy-preserving system for multiparty training and inference of XGBoost [18] (efficient, open-source GBDT library) models. The goal is to protect the privacy of each party's data as well as the integrity of the computation with SGX enclaves. However, the authors 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, the paper aims 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 data size). 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, more research needs to be done in this field. 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, such as model hyperparameter tuning, experimentation with insurance-specific synthetic datasets, visualisation of the overall output model, tool-assisted verification of constant-time properties, an the creation of a Python DP-GBDT wrapper for more convenient usage.

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.

[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.

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

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

[14] 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.

[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] 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.

[17] 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.

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

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

[20] Rachel Cummings, Gabriel Kaptchuk, and Elissa Redmiles. *"I need a better description": An Investigation Into User Expectations For Differential Privacy*. Oct. 2021.

[21] 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.

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

[23] 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.

[24] 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.

[25] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: https://archive.ics.uci.edu/ml/datasets/abalone (visited on 11/11/2021).

[26] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd (visited on 11/11/2021).

[27] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: https://archive.ics.uci.edu/ml/datasets/adult (visited on 11/11/2021).

[28] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2019. URL: https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original) (visited on 11/11/2021).

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

[30] 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.

[31] 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).

[32] 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.

[33] Theo Giovanna. "Privacy-Preserving Machine Learning for Cyber Insurance". MA thesis. ETH Zurich, 2021.

[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.

[35] Lucas Chi Kwong Hui and K.-Y. Lam. "Fast square-and-multiply exponentiation for RSA". In: *Electronics Letters* 30 (1994), pp. 1396–1397.

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

[37] 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).

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

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

[40] 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).

[41] 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.

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

[43] 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/.

[44] 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.

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

[46] David Kohlbrenner and Hovav Shacham. "On the effectiveness of mitigations against floating-point timing channels". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 69–81. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/kohlbrenner.

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

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

[49] 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.

[50] 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.

[51] Wei-Yin Loh. "Fifty Years of Classification and Regression Trees 1". In: 2014.

[52] 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.

[53] 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.

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

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

[56] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

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

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

[59] 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.

[60] 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.

[61] 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.

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

[63] 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.

[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.

[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.

[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.

Appendix A

Appendix

A.1 Acronyms

TEE trusted execution environment

SGX Software Guard Extensions

GBDT gradient-boosted decision trees

TCB trusted computing base

SMC secure monotonic counter

DP differential privacy

ORAM oblivious RAM

GDF gradient-based data filtering

GLC geometric leaf clipping

DP-GBDT differentially-private gradient-boosted decision trees

A.2 C++ DP-GBDT parameters

nb_trees	Total number of trees in the model
privacy_budget	Privacy budget available for the model
learning_rate	Learning rate
max_depth	Maximum depth for the decision trees
min_samples_split	Minimum amount of samples required to split an internal node
gradient_filtering	Whether or not to perform gradient-based data filtering during training
leaf_clipping	Whether or not to clip the leaves during training
balance_partition	Balanced sample partition, If set to `true`, all trees within the ensemble receive an equal amount of training samples. If set to `false`, each tree receives x samples, where x is determined by the formula in (DPBoost [48], Algorithm 2, line 8)

use_decay If True, internal node privacy budget has a decaying factor.

cat_idx List of indices of categorical features

num_idx List of indices of numerical features

l2_threshold g_{L*} from Section 2.4.1 on GDF

l2_lambda Regularisation parameter

use_grid Enables grid functionality described in Section 4.3

grid_borders Specifies constant 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 all values that a categorical features could theoretically take. As a result, splits on all possible values will be tested even though they do not actually appear in the dataset

scale_X If a grid is used, this enables feature value scaling, as described in Section 4.3

scale_X_percentile Select the percentile for the `scale_X` confidence interval

scale_X_privacy_budget Select the amount of privacy budget paid for `scale_X`. Higher ε results in more reliable and accurate confidence intervals

A.3 Evaluation hyperparameters

learning_rate	0.1
min_samples_split	2
max_depth	6
scale_y	false
balance_partition	true
use_grid	false
use_decay	false
l2_threshold	1.0
l2_lambda	0.1

use_dp	variable
privacy_budget	variable
nb_trees	variable
gradient_filtering	variable
leaf_clipping	variable

Table A.1: Model parameters for prediction accuracy experiments

A.4 Adding new datasets

Including a new dataset in the project[1] is very simple once the dataset is in comma separated form. The only thing left is creating a function, similar to the one in Listing A.1, that specifies the basic properties of the dataset.

[1] https://github.com/loretanr/dp-gbdt

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

Listing A.1: Adding a new dataset

A.5 Profiling output

Figures A.1 and A.2 illustrate the output from Intel vTune[2]. This shows current bottlenecks of both the unhardened and the hardened C++ implementation.

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 [%]

Function	CPU...	CPU Tim...
main	100.0%	0s
__libc_start_main	100.0%	0s
_start	100.0%	0s
DPEnsemble::train	99.9%	0s
DPTree::fit	99.4%	0s
DPTree::make_tree_dfs	99.4%	0s
DPTree::find_best_split	99.3%	1.964s
DPTree::compute_gain	94.8%	16.792s
DPTree::samples_left_right_partition	64.5%	20.842s
constant_time::select<int>	27.2%	15.163s
constant_time::select<double>	1.0%	0.534s
DPTree::exponential_mechanism	0.7%	0.092s
DPTree::_predict	0.6%	0.080s
DPEnsemble::predict	0.6%	0s
DPTree::predict	0.6%	0s
DPEnsemble::update_gradients	0.6%	0s
log_sum_exp	0.4%	0.048s
constant_time::max<double>	0.3%	0.064s
std::vector<SplitCandidate, std::allocator<SplitCan	0.1%	0.052s
constant_time::select<unsigned long>	0.1%	0.052s
constant_time::logical_or	0.1%	0.036s

Figure A.2: `hardened_gbdt`: Time spent [%]

[2] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

A.6 Hardening example

Caller/call graph

Pseudocode 4: DPTree::compute_gain

```
1  function compute_gain(X, gradients, feature_index, feature_value)        ▷ X, gradients
       // partition into lhs/rhs
2      lhs, rhs = samples_left_right_partition(X, feature_index, feature_value)  ▷ lhs/rhs size
3      lhs_size = lhs.size()
4      rhs_size = rhs.size()
       // return on useless split
5      if lhs_size == 0 or rhs_size == 0 then                                ▷ no split possible
6          return -1
       // sums of lhs/rhs gains
7      lhs_gain = sum(gradients[lhs])                  ▷ memory access pattern of left/right gradients
       rhs_gain = sum(gradients[rhs])
8      lhs_gain = lhs_gain^2 / (lhs_size+params.l2_lambda)        ▷ floating point arithmetics leakage
9      rhs_gain = rhs_gain^2 / (rhs_size+params.l2_lambda)
10     total_gain = max(lhs_gain + rhs_gain, 0)          ▷ max leaks whether total_gain < 0
11     return total_gain
```