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. Capkun

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

Department of Computer Science, ETH Zürich

Acknowledgements

Rough content:

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

Thesis TODOs, (notes for me)

- upload the whole project to github, and reference it correctly
- make (my) github repo(s) as footnote instead of cite
- results interpretation discussion points from moritz mail
- clean up security diagrams. names and style consistent with the text
- hard abstand disk, 3b in first trianing redundant resp. load* redundant?, take the 2 on first page apart, replication -> mseed distribution, questionnair counter needs ID, retrain counter wrong, palatino font?, smaller font would look better I think

Contents

Chapter 1

Introduction

Dummy text.

1.1 Motivation

Rough content:
- insurance use case (from thesis description)

Dummy text.

1.2 Problem overview

Rough content: What we have:
- DPBoost paper and it's LighGBM implementation
- DPBoost paper has (1) some undefined holes (2) some questionable results
- Theo's implementation (but it's slow and has bugs)

Goals dummy text

Rough content: What we want:
- Want a working (standalone) C++ implementation that we can put into enclave and harden it against side channels
- Want to get an idea of how much work these steps require
- Want an idea about performance overhead we get

————- We design a set of mechanisms, embodied in a system that we call Raccoon,2 that closes digital side-channels for programs executing on commodity hardware. ————-

1.3 Scope of this thesis

Rough content:
- digital side channels, no hardware attacks
- also no branch prediction shenanigans, assume enclave is patched for all known CVEs
- no implementation of the bigger picture theory (data collection, attestation)
- partial implementation of the really nitpicky hardening tasks.
- no guarantee for side channel non-existence

1.4 Approach

Rough content: not sure if this chapter is necessary
- build C++ implementation from scratch
- put it into enclave

1.5 Contribution

Rough content: Main contributions:
- build hardened C++ DP-GBDT implementation from inside an enclave
- measure performance differences

Other contributions that arose along the way:

- use TEEs for privacy preserving ML, theory is there, but it's not really used in practice
- have DPBoost paper (which has its flaws) and half finished python implementation -> fix stuff, -> bring into C++ -> bring into enclave
- have a reliable code base that is usable and extensible (comparability). Ability to run the code outside the enclave for ML experimentation (because it's fast and not a patched LightGBM chaos), and inside enclave to get a realistic feeling.
- information leakage through side channels (thesis description), timing/memory -> "digital side channels"
- how much work do individual countermeasures take. Where is a reasonable balance of investment and return.
- Offer some guidelines on manual (source code level) hardening of a tree based ML algorithm
- Offer some guidelines on how to put such an algorithm into an enclave. What kind of library functions you need to rewrite. interface between inside/outside.
- show a possible design of the bigger picture: how to collect questionnaires, input/output, balance between reliability/usability and security
- algorithm is too large for automated side channel hardening. Offer a viable alternative

Chapter 2

Background

Rough content: everything GBDT / DP will be very similar to Theos

2.1 Gradient boosted decision trees

Dummy text.

2.2 Differential privacy

Dummy text.

2.3 Differentially private gradient boosted decision trees

Based on DPBoost [26] and previous master thesis from T. Giovanna [15].

Dummy text.

Note that we should put the adapted algorithm pseudocode here.

2.4 Intel SGX

Rough content: not sure yet how in depth, but there's so much literature out there, should be easy

- TEEs
- Basics, Attestation, Sealing
- Limitations, overhead, drawbacks
- secure monotonic counters and their limitations

==== It is crucial that ML training takes place in a safe environment trusted by both the model and data owners. A trusted execution environment (TEE) is a viable option, even if the platform supporting it is not trusted. The Intel SGX (Software Guarded Extensions [19]) implementation is the most widely available hardware-assisted TEE method in comparison to others such as

the ARM TrustZone [118] and AMD Secure Memory Encryption (SME) [119]. ==== todo cites It sets aside a protected memory region,called anenclave, within an application's address space. Code execution and memoryaccess in an enclave are strongly isolated from external programs. The processor ensuresthat only code running in an enclave can access data loaded into it. External programs,including operating system (OS) and hypervisor, can invoke code inside an enclave onlyat statically-defined entry points. SGX also supports remoteattestation, which allows aremote user to verify that the initial code and data loaded into an enclave match a givencryptographic hash, hence ensuring the enclave to perform the expected computation.

==== While hardware-based enclaves protect confidentiality and integrity, those benefits come at a cost. It is important to protect communicating between CPU and enclave memory in order to prevent memory bus snooping. Further, SGX uses Memory Encryption Engine to encrypt and decrypt data transfers via the memory bus. This alone incurs a 2-3x performance overhead over regular execution outside an enclave [16]. ==== Second, the performance of an enclave is usually bounded by the EPC (enclavepage cache) size, a hardware-protected memory region used to host the enclave pages. The EPC size is usually small, e.g., only 168 MB in the most expensive Azure confidential computing instance [121]. Any memory usage beyond the EPC size will cause overhead due to eviction of enclave pages to main memory.

Third, because system calls still need to be facilitated outside of enclaves, there is a substantial context switching overhead. ————————————-

2.5 Side channels

Rough content:

- general intro
- timing side channels -> constant time programming (gilchner)
- memory access pattern (at least no cache problems)
- compilers (gilchner + "2.3.1")

In side-channel attacks, physical characteristics of the execution that are not accounted for in the theoretical model leak secret information to the attacker. There are a variety of physical properties that can be used for the construction of side channels, including timing [25], power consumption [24], sound [14], and electromagnetic emissions [29]. Although most of these attacks require the attacker to be physically close to his or her target, some, for example timing-based side-channel attacks, can even be exploited remotely. The adversary we consider in this thesis (section 3.3) can observe all of the side channels mentioned above and more.

(todo, n1 vanbulck in introduction)

————————————- Microarchitectural timing side-channel attacks, as outlined in the previous section, have traditionally been strictly limited to leaking execution metadata, e.g., the addresses of code or data accesses in a victim program. The preferred way to defend against these attacks is to adopt a "constant-time" programming model [44, 71], which ensures that the victim's code and data memory access patterns never depend on application secrets. For a long time, the side- channel research landscape has been characterized by an ongoing cat-and- mouse game, where attackers developed ever more accurate techniques to leak execution metadata and reconstruct cryptographic keys, prompting developers to refine constant-time code hardening patches in the affected libraries. This fundamentally changed in early 2018, however, when several researchers [67, 101, 146, 162, 249] independently discovered that side channels can

also be abused in an entirely different way to reconstruct secret-dependent traces left in the CPU's microarchitectural state following branch mispredictions or exceptions. 2 Intuitively, this new class of transient-execution attacks broadens the threat of microarchitectural side channels from relatively harmless metadata leakage to direct data extraction of arbitrary victim secrets.

Side channels in SGX "Tutorial: Uncovering and mitigating side-channel leakage in Intel SGX enclaves" Existing SGX side-channel mitigation approaches generally fall down in two categories. One line of work attempts to harden enclave programs through a combination of compile time code rewriting and run time randomization or checks, so as to obfuscate the attacker's view or detect side-effects of an ongoing attack. Unfortunately, as these heuristic proposals do not block the root information leakage in itself, they often fall victim to improved and more versatile attack variants [33, 6, 28]. A complementary line of work therefore advocates the more comprehensive constant time approach known from the cryptography community: eliminate secret-dependent code and data paths altogether. While this approach is relatively well-understood for small applications, in practice even vetted crypto implementations exhibit non-constant time behavior [28, 32, 6]. We conclude that side-channels pose a real threat to enclaved execution, while no silver bullet exists to eliminate them at the compiler or system level. Depending on the enclave's size and security objectives, it may be desirable to strive for intricate constant time solutions, or instead rely on heuristic hardening measures. However, further research and raising developer awareness are imperative to make such informed decisions and adequately employ TEE technology.

2.6 Related work

Rough content: not sure if this fits in my thesis

Dummy text.

Related Work is different from Background

I would make this a separate chapter towards the end (e.g., Chapter 7) and mention there previous works on running ML / DP algorithms inside enclaves

Overview
3.1 Solution overview (system model: insurer with enclave, disk etc; customer)
3.2 Adversary model (for Chapters 4 and 6) -- not only side channel trace but also capabilities like restart, kill, start multiple enclave instances...
3.3 Goals (efficient algorithm implementation, side-channel hardening, secure deployment)

Chapter 3

Requirements

In this chapter we outline the requirements our system should meet as well as our adversary model.

3.1 Security requirements

Rough content:
- questionnaire aka user data confidentiality
- user data privacy in the form of ε-differential privacy
- side channel resistance, especially in terms of memory we want cache line granularity

———————- digital side channel notion described in raccoon paper https://www.usenix.org/system/files/conference paper-rane.pdf In this paper, we introduce a technique that does just this, as we focus on the class of digital side-channels, which we define as side-channels that carry information over discrete bits. These side-channels are visible to the adversary at the level of both the program state and the instruction set architecture (ISA). Thus, address traces, cache usage, and data size are examples of digital side- 432 24th USENIX Security Symposium USENIX Association channels, while power draw, electromagnetic radiation, and heat are not. Our key insight is that all digital side-channels emerge from variations in program execution, so while other solutions attempt to hide the symptoms—for example, by normalizing the number of instructions along two paths of a branch—we instead attack the root cause ———————

We can ASSUME CACHE LINE IS MAX ATTACK GRANULARITY. It's the "state of the art". However that is a stupid assumption to make because it is prone to be broken and already kinda is given the right circumstances. And it's also not much additional work if you're doing the thing anyways for larger data structures.

3.2 Other requirements

Even a perfectly secure and privacy-preserving algorithm is of no use if it is not applicable for any real world system. Thus, there are some other requirements.

Rough content:
- working GBDT algorithm, so decent accuracy achievable
- Ok performance (runtime)
- usability (usable for different datasets, run on every hardware etc.)

3.3 Adversary model

Rough content:
- malicious service provider
- combination of the texts below (from similar work)
- when securing against the grid scaling, we kind of assumed a malicious participant, or a malicious service provider, we didn't know yet

Malicious Service Provider We consider the most powerful adversary: a malicious service provider who has full control over both the software and the hardware of his system. However, we assume that the used hardware is not compromised and is free of vulnerabilities. Specifically, we assume the TEE hardware of the server is not compromised, which results in an isolated execution environment with remote attestation capabilities for any code running inside. Any data that arrives on the server can be monitored, inspected, and edited by a malicious service provider. Further he can monitor the execution for side channels.

Further he has full access to tools like SGX-Step.

The goal of the adversary to gather as much data as possible about customers participating in the training process. Any information about their security infrastructure can be interesting.

———— We consider multiple, mutually distrustful parties involved in data-processing services. A service provider is not trusted by the users of the service to keep data secret;

A service provider might be the same as the insurance or a company, and the two might collude to steal secrets from their input data. A user of a service does not trust the software at any privilege level in the computational platform. For example, the attacker could be the machine's owner and operator; he or she could be a curious or even malicious administrator; he or she could be an invader who has taken control of the operating system and/or hypervisor; he or she might own a virtual machine physically co-located with the VM being attacked; he or she could even be a developer of the untrusted application or OS and write code to record user input directly

Denial of service is outside of the scope of our threat model. Untrusted applications can refuse to run, or the underlying untrusted operating system can refuse to schedule our code.

———— ([16])

Malicious User Do we care if companies collude? TODO

We consider an attacker who has complete control of the server (including the OS or hypervisor) and can thus access or alter any files stored on the server. The attacker may also tamper with, delete, reorder, or replay all network packets exchanged between the server and the client.

Also, we assume the enclave attestation and memory protection features of the SGX hardware function properly: i.e., once the enclave's identity is established, enclave-provisioned secrets are

not accessible from untrusted code. However, SGX does not explicitly defend against software and hardware side-channels

DP This section describes what we assume to be given to the adversary by the DP proof.

Goal: user privacy, decent accuracy, manageable performance Not model confidentiality, right? TODO

Chapter 4
Algorithm implementation
- starting point (paper, Python prototype)
- design and implementation
- discovered bugs and fixes
- verification

Chapter 5
Enclave hardening
- goal: special type of hardening (DP)
- known tools and techniques
- chosen approach: manual...
- technical details

Chapter 4

Design and Implementation

This chapter gives a broad overview of the design and implementation work that was carried out for this thesis. This includes:

- overview of the project structure
- design decisions and implementation features
- transferring the code into an SGX enclave
- side channel hardening details and takeaways

This is a lot of content for one chapter. However, as these elements are all quite tightly connected, we opted against strictly separating them. This facilitates justifying certain design choices that were made along the way. There are also some remarks about things that did **not** work out as expected. So hopefully, this part offers some guidance for future DP-GBDT and hardening endeavors.

Project structure The project[1] essentially consists of five DP-GBDT versions and some additional infrastructure to run/verify/evaluate/... them. This separation is a big advantage, since both hardening and running a program inside an enclave tend to clutter the underlying code to some extent. The outcome of running the different variants is the same: equal input means equal output. Details on running and testing them can all be found in the public repository. The next section introduces `cpp_gbdt`, the non-hardened C++ version. It is roughly 2500 lines of source code in total. The algorithm is clearly visible and compiler optimizations and threads lead to solid runtime performance. Ideal for experimentation with the underlying algorithm.

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

Figure 4.1: TODO, project structure

4.1 C++ GBDT from scratch

Rough content:

- assumption (only DFS, no 3-nodes etc.)
- datasets and parsing
- Comparability and Verifiability
- Logging
- performance improvements
- Evaluation component and plotting
- Documentation

Note, the following parts of this chapter are not nicely articulated! content is roughly there though.

This section offers a summary over the different tasks involved in creating a first working C++ DP-GBDT implementation.

While python is nice and easy for machine leaning tasks, and benefit of many frameworks. Theo implementation was just too slow to be used and tested and tuned.

Even though there are machine learning libraries for C++, we decided agains using them. Not only would we end up with a patchy construct like DPBoost's LightGBM implementation. It would also complicate the upcoming hardening process. We would have to rely on their side channel freeness, or rewrite and patch the code. And we might not fully understand what's going on.

The only library used in the algorithm is the C++ standard library [22]. c++11 version was chosen because of future enclave compatibility. So can use library functions for non secret dependent part of the code.

Challenge was to reconstruct parts of the python libraries. Need cross validation.

Reducing to the essentials 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 the DPBoost paper or in Theo's thesis:

- best-leaf first decision tree induction
- "2-nodes" decision tree induction algorithm
- sequential composition of multiple ensembles
- generation and use of synthetic insurance datasets
- multiclass classification

This is of course unfortunate, however, if required they can always be added to the current implementations with a modest amount of work. On the bright side, also new features were added. The next paragraph denotes such an example.

use_grid This not obvious sounding paragraph title is the name of a hyperparameter of the algorithm. They are all described in appendix B. The DPBoost paper ignores an implementation

detail: When trying out different splits to find the one with the highest gain, you cannot iterate over the real feature values that occur in the dataset. Otherwise you can learn something about dataset by closely examining the split values of final model that is produced. One way of solving this is to iterate over a fixed grid instead. Given a fine enough granularity, the same splits will be found as if you were iterating over the real feature values. This is depicted in figure 4.2.

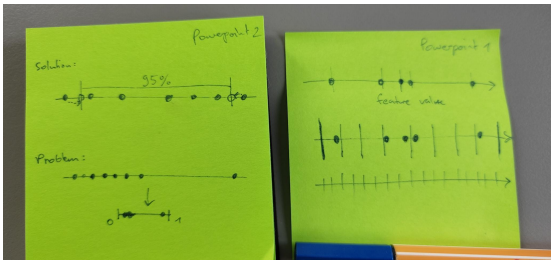


Figure 4.2: use_grid and feature scaling, TODO

Certain datasets however, will have numerical features with feature values of really different magnitudes. This creates a problem: Imagine feature A with values in [0,1] and feature B with values in [0,1000]. You can certainly define a grid that is fine enough for feature A while simultaneously accommodating all feature values present in B. However that would be extremely inefficient. A natural solution would be scaling feature values into the grid. If the numerical feature values of the entire matrix X are in [0,1], we could choose our grid borders accordingly and not waste so much computation power. This comes with another problem though: A malicious entity M, e.g. a training data supplier or even a malicious service provider, could submit bogus questionnaire data: taking again feature A∈[0,1]. If M chooses a value of 100'000 for feature A, scaling [0,100000] into [0,1] would cause the legitimate participants values to be extremely close to each other, to the point where the grid is not fine enough to consider all possible splits. This way an attacker could essentially eliminate certain features from having any effect in training.

One solution to this is to use confidence intervals. A common percentile to choose is 95%. Therefore you basically need to search the 2.5% and the 97.5% border of your data. Then the outliers values above the 97.5% border and below the 2.5% border are clipped to their respective border. The challenging part is doing this in a differentially private fashion. Fortunately this paper [8] proposes multiple options on how to achieve this as well as corresponding R code on Github [9]. I used the EXPQ method, which is not the most efficient one in terms of privacy budget according to the paper, but it seemed like the simplest one to understand and port to C++. The price for this is of course a small loss of information in your training data, as well as a small amount of privacy budget. More details on this in section 5.2.

In my implementation you have both options. You can either skip scaling, save some privacy budget, and set grid borders and step size accordingly. Or, if you know enough about the values in your dataset i.e. that your data values are going to be in a certain range, you can enable the scaling and save precious runtime.

Datasets and parsing Currently only regression and binary classification prediction tasks are supported. To demonstrate this there are four datasets are available and ready to use in my project: Abalone, YearMSD, Adult and BCW. As already mentioned, it's unfortunate that there was no time for experimenting with Theo's synthetic insurance datasets. However, we now for the first time have detailed and realistic results for 4 of the datasets mentioned in DPBoost and Theo.

This should be a valuable foundation for future experiments and tuning of the algorithm. Further, it is very easy to add more: There might be a small amount of manual preprocessing necessary to get a clean input file with comma separated values from a dataset. Once this is achieved, about 15 lines of code in the `Parser` class are required, to specify how the new dataset looks (size, target feature, feature type, regression/classification etc.). You can find an example of this in appendix B.

Comparability and verifiability During development it was a constant struggle. How to make sure our results are correct? How to verify that the feature we just implemented is bug free? Sure, there was the reference implementation in python that I was trying to rebuild.

There were multiple obstacles:

- Naturally, being the first of its kind, the python reference implementation contained multiple bugs, that were only gradually discovered. It ranges from programming slips to relatively important parts of the algorithm that were misinterpreted. There is a list of it in appendix B
- the algorithm is full of randomness
- the python implementation was painfully slow
- numerical differences

The most annoying kind of bugs were numerical differences. You would for example do multiple mathematical operations on a value, like $\log \sum \exp(x)$ only to discover, that the python math library's result differs from the 12th decimal onwards from the result from the C++ std library. This does not seem like a big deal on the first glance, but it is. Because of the nature of the algorithm, there is 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 will grow larger and larger up to the point where they affect the shape of a new tree. E.g. when searching the next split in an internal node, slightly different gradients leads to a different gain being computed which can result in a different split being chosen. This means different samples will continue to the left/right children of that node and so on. Once such a thing has happened, the trees will start to look completely different. And there's no way to tell whether you just made a mistake while programming, or whether it's the math libraries.

As python and C++ libraries use different sources and mechanisms to generate randomness, it was not as simple as giving the same seed to python and C++. solution: every single bit of randomness had to be turned off. This can be, shuffle the input dataset, exponential mechanism, laplace mechanism for leaf clipping, random selection of samples for the next tree etc. Plus at certain points, afetr certain math operations, values had to be clipped to 12 decimals.

Given these hacks it was possible to eliminate the differences between the python and C++ algorithm. And we would finally get the exact same result for the same inputs. To keep it that way and make the developers life easier in the future a verification framework was set up. In its core it's a bash script that (compiles) and runs both the C++ and python code on a couple of small datasets. During execution of the algorithms it will continuously compare intermediate values of the implementations. This way one can not only quickly recognize if you made a programming mistake. But you can also get the exact location where the two implementations started diverging. It will give you convenient and nicely formatted output "dataset x, ensemble 4, tree 2, split 5 was different". This makes it a lot easier to debug and develop.

Can be used to compare python, C++, and hardened C++ implementation to see that they all give exactly the same result. More details on running/tweaking in my github [27].

Logging We use fmtlib [34] for logging. It's very convenient. You can select from multiple different logging levels, that will get you output of the desired degree. You can further print the finished tree to the terminal.

Performance Enhancement At this point the porting of the core algorithm to C++ was done. Even though the main goal was to create a hardened SGX version of the algorithm, a small detour was made at this point. Our GBDT algorithm has around 20 hyperparameters that can be tweaked to your needs (see appendix B for details). I noticed myself, that experimenting with a slow implementation is frustrating. Being a compiled language, C++ was naturally already quite a bit faster. However, in order to encourage further work like experimentation with the algorithm and parameters, I wanted to improve performance even more. Note this is not something that can and will be used in the hardened version.

In order to determine the right spots to steer your attention to, regular profiling was done with Intel vTune [21] and Intel Advisor [17]. For illustration purposes you can find a snipped of the profiling output of the finished (optimized) C++ GBDT algorithm in appendix B. It will also give you an image of where the current bottlenecks are in the finished product.

The actual performance gains achieved, and put into relation to the other implementation are depicted in chapter 5.3.

What kind of optimizations are we talking about? First of all multithreading. As we are always doing 5-fold cross validation, the partition of resources among threads happens accordingly. Next up, compiler optimizations. By enabling aggressive optimization, general and floating point specific we gain quite a bit. Further, by removing blockers in core/critical functions we alleviate the compilers job of vectorization.

Note also that this was done rather quickly. For the performance inclined reader still much to be gained if you think more about caching/locality, vectorization etc.

Evaluation component and plotting For convenient experimentation there's the evaluation framework. You can use it to create expressive performance measurements (as in section 5.2). The results can be exported to csv format, where they can be plotted with python.

Documentation doxygen, TODO probably leave this away.

4.2 Moving the code into the enclave

Rough content:

- App.cpp Encalve.cpp bundary
- .edl file
- bring data into the enclave (+ sketch) either custom serializ or c types
- removal of logging / libraries except stl *enclave version of stl
- randomness
- other changes?
- caveat enclave parameters (memory bound), specify enclave setting file

This section outlines the steps required to bring the algorithm into an enclave. It is not the smallest project, so not trivial. This should also serve as guideline, and show what need to be taken care of.

Figure 4.3: Enclave.edl entries, + visualisation of inside outside arrows etc.

We do not care about the details of attestation, encryption of customer data and so on (this will happen in chapter 6). The goal is to run the core algorithm in the enclave, as a proof that it works and for runtime/overhead measurements.

Set up SDK 2.14 according to docs, we start from the included Cpp example enclave and extend it. We used C++11 from the start, though now at the time of writing (October 21) Support for C++14 in SGXSDK came was added [18].

First of all the application needs to be divided into outside/inside the enclave. In practice the files App.cpp describes the outside actions, and then it switches to the inside Enclave.cpp, where our algorithm is then started.

Important is the Enclave.edl file. It specifies all function calls that are allowed to cross the border. In our case (see figure 4.3) that means 4 calls: 3 ecalls that go into the enclave, each of them is called exactly once. 2 of them bring the dataset resp. parameters into the enclave (where the data is put into C++ datastructures and saved as global variables). The other one does not carry any data and just initiates the training. The only ocall and thus way to get data out of the enclave is the ocall_print_string, which is used to produce the output in the first image I sent today. Which is sufficient to see that the algorithm works.

Parsing the dataset(s) requires a bunch of I/O functions and other calls that are not available inside the enclave. So the parsing (as well as defining the model parameters) is done on the outside, then they are passed to the enclave. Since the interface between outside/inside does not allow passing custom datatypes or even C++ vectors, that means converting the dataset (and model parameters) to C-style arrays and then back to C++ structures on the inside.

The gbdt algorithm itself had to be adjusted everywhere where randomness is involved. Because obtaining randomness inside an enclave is a bit different than on the outside. SGX provides a function sgx_read_rand that executes the RDRAND instruction to generate hardware-assisted unbiased random numbers [23]. Apart from just replacing calls that fetch a random number, this also meant writing several function from scratch that were previously not much more than a simple std::library call. For example randomly selecting/deleting a subset of rows from a matrix.

Of course also the external library code for e.g. the logging had to be removed. Similarly no more multithreading is available.

Further, there's a config file Enclave.config.xml with important setting such as available memory size. Which is necessary for lager datasets. Few changes to the Makefile and we're ready. More details in my repo [27].

4.3 Side channel hardening

Rough content:
- intro
- hardware vs software adv/disadv
- hardening thoughts algorithm level
- overview hardening branches, loops, then some concrete examples with pseudocode.
- voluntary additional hardening (fault injection / recursion hiding)
- arithmetic leakage
- compiling / verifying
- takeaways

Various approaches possible: In theory there are automatic hardening tools available [5, 1, 30]. However, investigation showed that the solutions are still in an early stage and not really feasible for usage in practice. Either the papers did not publish their tool, or the tools are more of a proof of concept, that work on very small examples and require a ton of annotations and extra information to be added to the source code. Complete auto tools, e.g. branch removal on assembly level, will also bring a ton of overhead. Surely not feasible for our project.

Therefore we chose to manually harden by hand. This way we can leverage the knowledge of the algorithm to pinpoint critical location that need hardening. This should mean significantly less runtime overhead. On the flip side this means we can't give any security guarantees apart from "we did it to the best of our ability/knowledge".

Manual hardening can be done on different levels. You can harden C++ code on source code level or on assembly code level. Working on the assembly code directly is hard though. Difficult to maintain overview and a lot of work. The benefit is however that there aren't any unwanted surprises. At least GCC for example passes your assembly source through the preprocessor and then to the assembler. At no time are any optimisations performed (TODO cite https://stackoverflow.com/a/64670604/6576122).

Professionals also choose a mixture by including their own assembly code snippets into source code.

With the algorithm being rather large, and the secret data is pretty much everywhere, we chose to harden on the source code level. This obviously brings with it the problem of the compiler. How can we ensure that it does not optimize away our hardening efforts, or create new side channels on its own? These questions are addressed later in paragraph 4.3.

As described in the requirements section (3.1) we're trying to fully hide access patterns, not only cache line granularity.

Fixing general issues These constant time violating constructs appear frequently in code and need to be replaced with oblivious operations.

- logical boolean operators (and, or, not)
- comparators
- ternary operator
- oblivious sort
- std::min/max leak through memory access pattern

Some compilers (depending on which one and what version) will translate code with logical operators to assembly code with jumps and thus side channels. Take the snippet in figure 4.4 for example.

```
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 4.4: Short-circuit evaluation x86-64 icc 2021.3.0 -O1 vs x86-64 gcc 11.2 -O1

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

a constant time ternary operator or oblivious assign/select most branches can be done in constant time. for this both branches are evaluated, and the result is chosen with the constant time ternary operator.

In terms of comparators: Checking (in)equality can also be implemented using xor similarly to the logical not. The only comparators missing are the order relations on integers. Strictly speaking we only need to find one (e.g. $<$) and can then express all the remaining ones by combining it with not and comparison with zero.

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

Listing 4.1: Constant time ternary operator

Another example is oblivious sorting, which finds application in the scaling of the grid before the actual training. Using the $O(n^2)$ algorithm, because it's very easy to harden, and it's not a performance critical task (done only once before trainng).

```
// 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 4.2: Oblivious sort

DP-GBDT hardening In this section the we'll give an overview of the side channels that occur in DP-GBDT with examples and how to harden them. More details can be found in appendix C.

Things that need to be hidden:

- path that samples take in the tree

Arithmetic leakage ———————— glichner Hardware instructions provided by the CPU are a source of additional issues when trying to implement constant-time cryptography. Modern CPUs provide a big set of instructions, however, not all of them execute in constant-time. The canonical example of this is integer division (which usually extends to mod, often using the same instruction). While addition and subtraction are easy to implement fast, division is until today still comparatively slow. For this reason many division implementations take shortcuts wherever they can, which results in execution times that are highly dependent on the inputs. Other issues stem from gaps in the language standards. The C-standard usu- ally does not define any mapping between high-level C-operations and hardware instruction. This leads to the fact that operations might be implemented in non-constant-time. While this is potentially the case for any operation, some operations like comparators or logical boolean operators, are more prone to non- constant-time implementations than others, due to a lack of directly corresponding assembly in- structions. In some cases seemingly simple operations are even defined inherently with non-constant-time semantics. The most import example of this are the two logical operators && and ||, whose short-circuit-evaluation semantics 12 necessitate a jump depending on the value of the left operand. The best way to solve these problems depends on the assumption that the most basic operations (addition, subtraction, and bitwise logical operators) will always be translated to directly corresponding hardware instructions. In the following we will present solutions to these problems, these are mostly based on Pornin's BearSSL [40] as well as the Guidelines for low-level cryptography software by Aumasson [7]. ————————

The ones just described are among easy hardening cases. However replacing all arithmetic operations everywhere in the code is not only a major task, it also worsens readability of the hardened code. Also for floating point numbers it's naturally more difficult and there are fewer libraries/code available. We therefore opted for a compromise: we harden the most performance critical section of the code, where 90-95%+ of time is spent using libfixedtimefixedpoint [4]. Fortunately the code sections that we spend most time in don't have a lot of floating point arithmetics. So we could get away with adding it only there. That way we can spare ourselves

most implementation headaches, while still arguing that “we did it” and "finishing the job” would not incur much more runtime overhead.

Fault injection on parameters This can be seen as bonus hardening task. Because fault injection is more of a hardware attack, although recently fault injection attacks have also been achieved in software (TODO cite Plundervolt resp. VoltPillager, 2020/21). It would be very unfortunate if an adversary could simply turn off differential privacy by glitching and flipping one single bit. Or increase the privacy budget. Or glitch another hyperparameter that the insurance customers did not agree upon. The solution implemented replaces boolean model parameters (internal 1 or 0) with two random integers with hamming distance >25 (see listing 4.3). Then each time a parameter is used, we first check whether it is one of the two values. This way a glitch would need to flip exactly those >25 bits in order to switching off some privacy related model parameter, which should essentially be impossible in practice.

```
1 // TRUE:  01101100001011110101111000011011
2 // FALSE: 00000001110001101010000111100100
3 #define TRUE 1815043611u
4 #define FALSE 29794788u
```

Listing 4.3: Parameter fault injection mitigation

Compilation and verification ————————- (gilchner) As explained above high-level programming languages usually do not include run- time in their semantics. Hence compilers provide no guarantees about the runtime of a program. Instead they try, depending on the compiler settings very agres- sively, to make programs as fast as possible. The results can differ vastly between compilers. This can be beneficial to performance of cryptographic software, but its unpredictability is detrimental to security. Take as example the functions in XYZ. Thus we not only need to take care that we do not use any non-constant-time operations. But we also have to take care that our implementation is obfuscated enough so that the compiler cannot fully understand its semantics and thus produces constant time assembly. With this knowledge, it is also apparent why we must get rid of branches, even though one might think it sufficient to ensure that all branches take the same time. A compiler will not take time differences between branches into consideration and thus will reduce the time needed to take one branch whenever it can. ————————

So, having no guarantees, how can we still verify our assembly code with a somewhat manageable effort?.

There are tools for constant time programming verification (e.g. [31, 3]) However it requires some effort + what about the other side channels we care about?

I decided to take a relatively conservative approach: Most Compiler optimizations are disabled with the -O0 flag. Constant time functions are all defined in a new file and namespace. Further, inlining is disabled with vendor-specific keywords. This way you can conveniently check all hardened functions once and they are all gathered in one place. Another extra layer of safety measure that was added to the hardened functions is the `volatile` keyword. Its purpose is essentially to prevent unwanted compiler optimization.

General rules and takeaways

- Avoid table look-ups indexed by secret data
- Avoid branchings controlled by secret data
- Compare secret strings in constant time
- Avoid secret-dependent loop bounds
- Prevent compiler interference with security-critical operations
- Clean memory of secret data
- Use strong randomness
-
- guidelines from https://github.com/veorq/cryptocoding, n1

based on these we came up with a set of rules that should be sufficient to harden a DP-GBDT application against side channels.

In the ehat of the moment hardening was just robotically applied. In hindsight some things were hardened that were not strictly necessary. Obviously this is not a bad thing, as we don't want an adversary to get any information at all at best. But this prolonged the very important task for future endeavors: coming up with minimal and easy to follow hardening rules for DP-GBDT

- bla

secret dependence in dp_ensemble:

GDF with balanced: it should be enough to hide whether a sample was chosen randomly or whether it was chosen because of its favorable gradient. GDF no balance: same, we have our numebr of rows that we have to fill up no GDF: randomly select fixed number of rows. should not require hardening

so we hide why it was chosen but not which sample. That facilitates choosing a subset of rows from the matrix a lot. And also deleting from the matrix.

Training/predict - basically need to hide which path a sample takes. - the OP hardenings everywhere where secret dependent decisions are made. (Tree building, sample distribution, prediction)

Class dp_tree DFS recursion - we don't need to hide when the recursion stops and which type of node was created. - we hide how many and which samples are live. -> Rest of the computation has lots of overhead. - find best split, use (whole) grid - compute gain -> sample-left-right-p obliviously - hide number of useable candidates for the exponential mechanism - exponential mechanism -> hide which one was chosen

Leaf clipping if we know what samples end up in a leaf, need to hide which leaves have to be clipped. Couldn't we just circumvent this by doing an oblivious matrix row shuffle in the beginning? Yeah should work and not be impossible to achieve. But not trivial!

prediction hide the path that this samples goes down into by traversing the whole tree. if we didn't then one could observe how 1 matrix row behaves in each tree. So we could kinda build anonymous profiles of the rows. But we wouldn't know who it is.

Can we save all this hassle if we just oblivious shuffle at the start? - Or maybe after every tree? - but I guess, if we left away hardening stuff like: iterate over all possible splits instead of those apearing in the dataset. we would learn something about the dataset.

obliviously shuffle 1 array would be enough. Could then have 2 matrices and obliviously assign each row from the src matrix to the target matrix $O(n^3)$

Chapter 5

Evaluation

Rough content:
- TODO which plots to put here and which ones in appendix

5.1 Methodology

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

name	size	features	task	only use subset
abalone [10]	4177	8	regression	no
yearMSD [11]	515345	90	regression	yes
BCW [12]	699	10	binary classification	no
adult [13]	48842	14	binary classification	TODO decide

Table 5.1: Datasets

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

5.2 Performance

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

Hyperparameters The utilized hyperparameters, are listed in appendix A. The `use_grid` option is turned off, since it leads to virtually the same results given a small enough step size while substantially increasing runtime. As a matter of fact, experiments showed that the grid implementation presented in paragraph 4.1 works reliably using a privacy budget of just 0.05. There are certainly smarter ways that cost less privacy budget though.

Note that the chosen hyperparameters are by no means optimal. With hyperparameter tuning, even better results should easily be achievable. Such tuning was omitted due to timing constraints and due to the lack of suitable frameworks for this task in C++.

Abalone For abalone we can take the full dataset (4177 samples). As it is a regression task we can take the mean of the target feature as a baseline, which in the case of abalone is around 3.22. Figure 5.1 shows TODO.

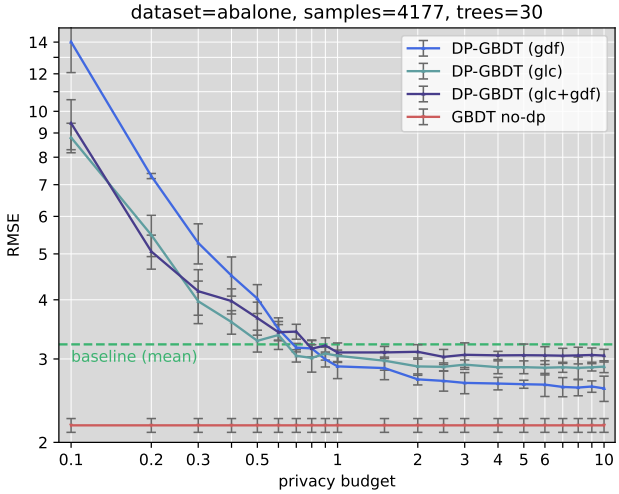


Figure 5.1: Abalone RMSE for 1 ensemble of 30 trees

yearMSD Since yearMSD is quite large (over 500k entries and around 90 features) experiments were conducted on subsets of it. Further, following the guidelines on its source webpage, it was ensured that training and test samples were chosen from distinct parts of the dataset. This avoids the 'producer effect', so no song from a given artist ends up in both the train and test set. Figure 5.2 shows TODO.

Breast Cancer Wisconsin Being the smallest dataset, the results were not as stable as those from other datasets. Therefore the result values are the average of 5-fold cross validation repeated 10 times. Figure 5.3 shows TODO.

Adult TODO, full dataset instead of only 5000 samples. but will take a while for non-dp. figure 5.4.

5.3 Runtime

In this section we talk about runtime

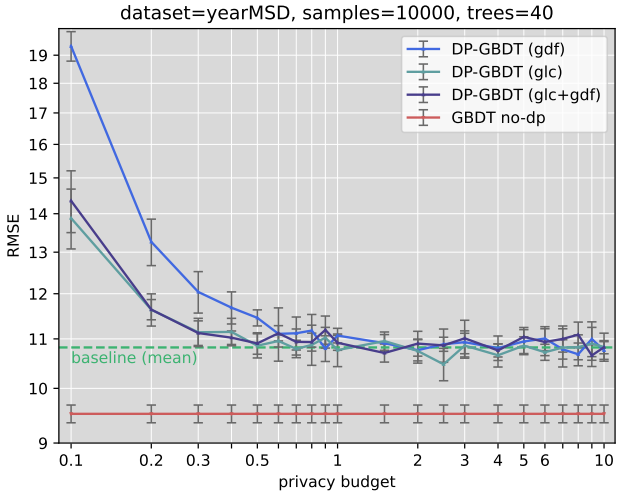


Figure 5.2: yearMSD RMSE for 1 ensemble of 40 trees

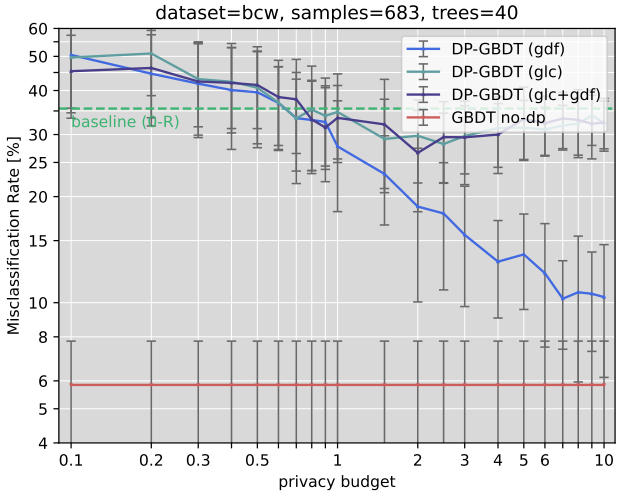


Figure 5.3: BCW misclassification rate for 1 ensemble of 40 trees

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 5.2: Parameters

Hyperparameters Note, all runtime results describe the amount of time spent to perform 5-fold cross validation on a dataset. Note, many hyperparameters don't have a huge influence on runtine. But some do. Note, python sklearn uses multithreading for cross validation. Note, other stuff

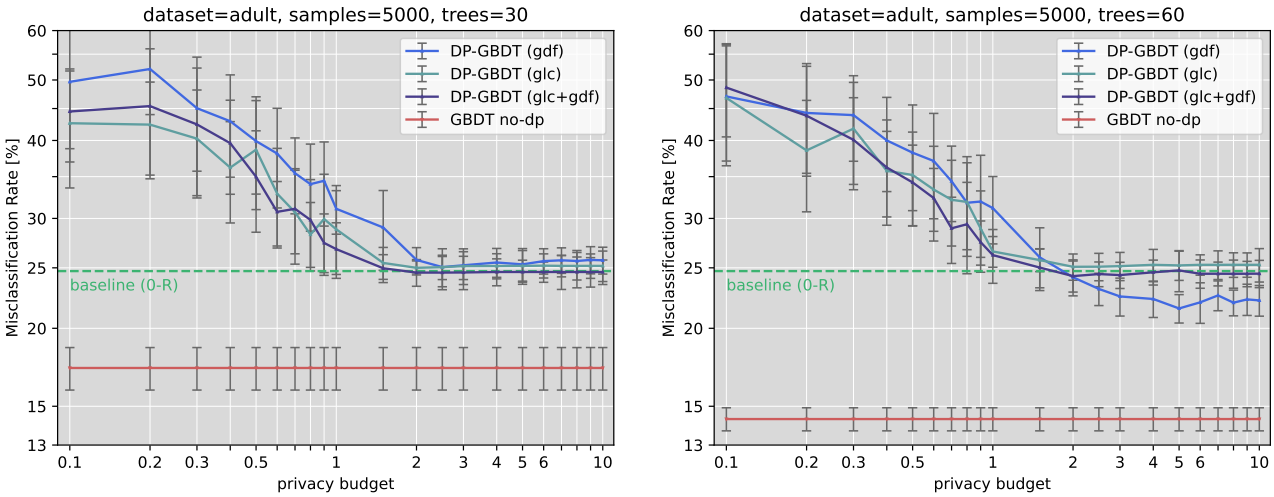


Figure 5.4: Adult misclassification rate: ensemble of 30 resp. 60 trees

was running.

Implementation	abalone			adult			yearMSD	
	300	1000	4177	300	1000	5000	300	1000
python_gbdt	7.4	11.5	75.7	7.3	16.6	209	46.8	441
cpp_gbdt	0.02	0.07	0.26	0.02	0.04	0.16	0.14	0.55
enclave_gbdt	0.27	1.3	6.9	todo	todo	todo	todo	todo
hardened_gbdt -O0	12.7	71.3	813	31.8	158	2170	65.6	578
hardened_gbdt -O1	2.1	11.2	123.8	4.8	22.3	313	9.3	81.6
hardened_sgx_gbdt	todo	todo	todo	todo	todo	todo	todo	todo

Table 5.3: Runtime [s] 5-fold cross validation

What is nice: for datasets with mainly categorical values we're pretty fast actually. bsp adult with 10 features with values in [1,10] -> we can easily do 40k samples.

use_grid overhead resp. more abstufungen when hardening.

Secure deployment
- recall system model (enclave, disk, ...) and adversary model (restart, manipulate disk, ...)
- motivating example / strawman deployment (that breaks DP)
- goal: deployment scheme that preserves DP but is also practical (data may get lost, enclave may crash...)
- design (setup, operation, migration...)
- brief analysis (why design ensures the goal)

Chapter 6

Real-world deployment

The task of performing DP-GBDT in practice would be easy if, for example, the training data was available all at once. Or, if our computers would never break and humans wouldn't loose things. However as this is generally not the case in reality these issues need to be addressed. This chapter aims to give a high level overview of the challenges that accompany the real world insurance use case. For each problem an appealing solution is presented and the reasoning behind it is explained. We will start by replaying the general scenario and then shift our focus towards the individual steps.

Scenario The insurance starts by preparing and deploying the DP-GBDT enclave on its servers. The enclave has to run and must be reachable by external customers. At this point the insurance starts marketing their cyber insurance policies to companies. In return, customers will send them 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 to the enclave. The questionnaires arrive at the enclave one at a time. Upon reception the enclave saves the acquired data to disk. This is done until enough samples are collected to get a meaningful result from running the DP-GBDT algorithm. At this point the insurance can initiate the first training and receive the resulting model from the enclave. The enclave continues collecting samples from customers. Again, once enough samples are collected the insurance can instruct the enclave to train on the newly received data. The resulting trees can be used to refine the previous model. Further, if at any point a part of the model is lost, the insurance can instruct the enclave to recreate the missing part of the model.

6.1 Enclave setup

This section describes preliminary steps and preparations to be completed by the insurance before putting the enclave into operation. In a nutshell, numerous settings need to be defined and hardcoded in the enclave source code **before** attestation. This is important from the client's perspective. For example the client does not want to rely on the fact that the insurance chooses 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 shouldn't wait to see what happens once the counters are exhausted.

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

Client application and client↔server communication The client application will be fairly simple. It allows the client to fill out a questionnaire about their cybersecurity policies. Upon completion of the questionnaire, attestation between the client and the enclave can be initiated. A trusted third party ensures public access to both the source code and configuration of the enclave and the client application. After that the client application lets you send the encrypted questionnaire answers to the trusted application. The encryption key for this is a shared session key that can be established during attestation [20]. Further, the client application should ensure that the filled out questionnaires satisfy some minimum quality requirements. For example, it should not be permitted to submit empty questionnaires or ones containing values that are impossible.

Customer legitimacy TODO

Rough content:

- what if a company sends 100'000 questionnaires? Could render all legit companies' data useless.
- how to stop companyC, that isn't even a customer of the insurance, from sending data to our enclave?

Possible but ugly solution: insurance sets up list of tokens, signs it, encrypt it with the enclaves public key. Whenever a new customer appears, they can hand out one such token. The customer sends their token via secure channel to the enclave during remote attestation. The enclave enclave compares the token to a predefined list, signed by the insurance

Enclave state and initialization Assuming that, for whatever reason, the enclave was turned off or had to be restarted. After turning back on, how does the enclave know in which state it is in? In essence, the enclave uses two secure monotonic counters to save its state. As depicted in figure 6.1 the enclave can use the log-counter to determine whether it is being started for the very first time. If the counter is greater than zero, initialization has already been done. The enclave then goes into collection resp. waiting mode. This is the default state which will only be left upon reception of a `start_training()` command issued by the insurance.

Enclave-initialization is depicted in figure 6.2 on the right hand side. The log-counter was already initialized in the setup phase and its ID was hardcoded into the DP-GBDT enclave's source code. Regarding step (2.): The master seed s_m has two main purposes:

- First, it serves as an identifier for all data that was written to disk by this exact enclave. This prevents attacks where an adversary could swap sealed data on the hard disk with sealed data from another clone of the same enclave. It is therefore important that the

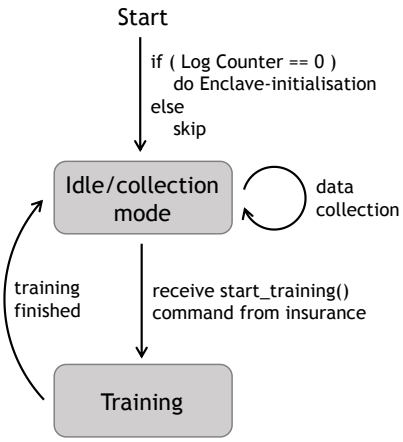


Figure 6.1: Enclave state diagram

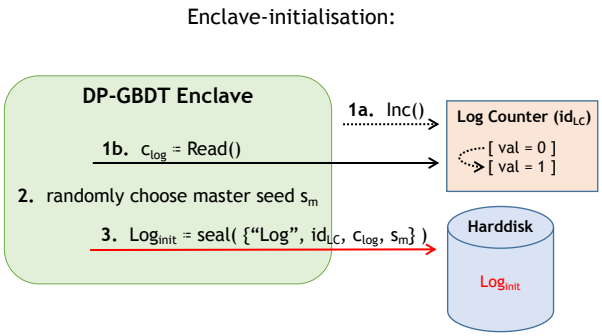


Figure 6.2: Enclave-initialization

master seed s_m stems from a true random number generator (TRNG). In practice this can be accomplished through the `sgx_read_rand()` function [19, 20].

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

6.2 Enclave operation

This section describes the enclave's two core operations: data collection and training. As you will see, data collection stays essentially the same, regardless of whether we're collecting the first or the 974th sample. In terms of training however, we will differentiate between the very first one, later trainings with new data (refinement trainings) and the repetition of training that was already done in the past (re-training).

Data collection phase Let's start with an overview of how receiving and saving customer data looks like from the enclave's perspective. Figure 6.3 illustrates this.

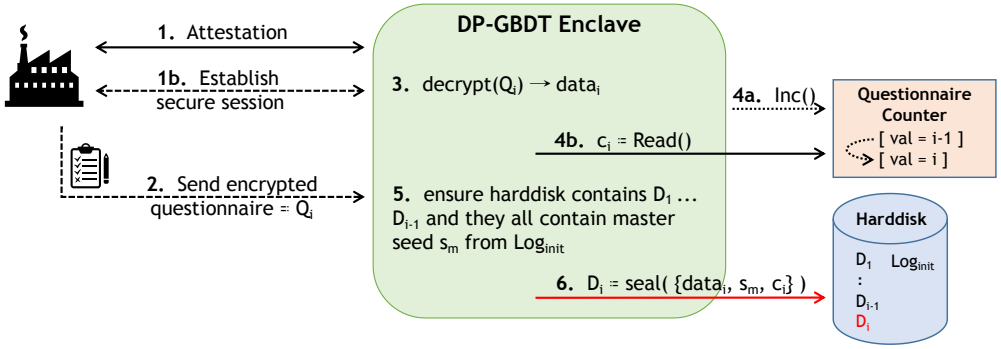


Figure 6.3: Data collection phase

Some remarks: (1. & 2.) As already indicated, the encryption key for this is a shared session key

that is established during attestation. This ensures confidentiality, integrity and replay protection of the sent data. (5.) This is not strictly necessary, but it would help the insurance identify problems or tampering early. (6.) The sealed data consists of the questionnaire, the master seed and the questionnaire counter. The counter determines the order of samples in the dataset. It ensures that, for example, an adversary cannot observe training with 100 samples, drop one sealed data piece, and then monitor training again with 99 samples. Or with another sample in place of the dropped one etc. this would be a dp problem, right? The downside of such an approach is that we absolutely must not loose any sealed data piece. Otherwise all progress is lost and we back at the very start. 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 following sequence of steps is pictured in figure 6.4. Please be aware that some messages contain redundancy. The goal was to create comprehensible diagrams and not to save every byte of space whenever possible. Also note the load* primitive in the diagram. This is an abbreviation for the following tasks:

- Unseal the data from disk.
- Perform different checks to recognize whether tampering occurred. This includes: (1) checking whether all D_i in our desired interval are present (and contain the right c_i). (2) Check whether all D_i contain the master seed s_m from Log_{init}.
- Abort training in case any check fails.

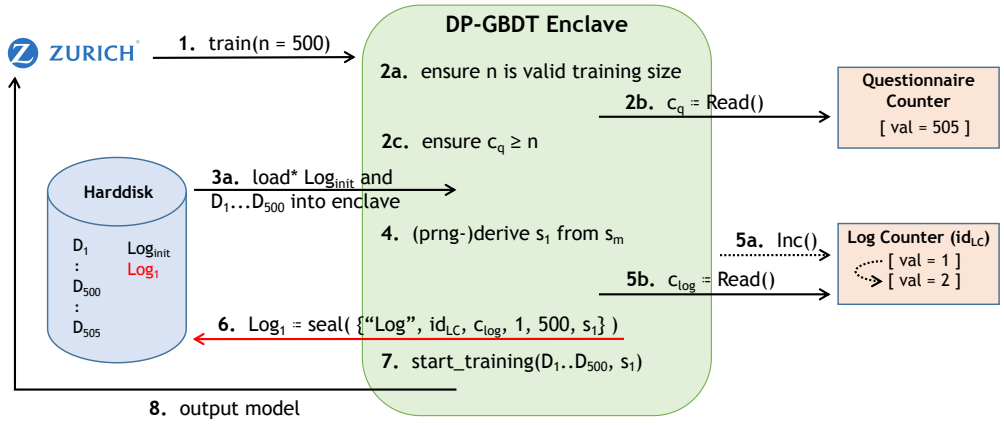


Figure 6.4: First training

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

Refinement training This paragraph demonstrates how the existing model can be improved with newly received data. For DP-GBDT there are generally two options: First, the new samples could be used in combination with old ones to train the entire model again. Second, the new samples are used separately to create new trees that are appended to the previous ensemble. We will only discuss the latter one here, since the first option is still being developed. Another reason

is that this approach does not require an extra payment of privacy budget, because we operate on fresh data. The process is visualized in figure 6.5.

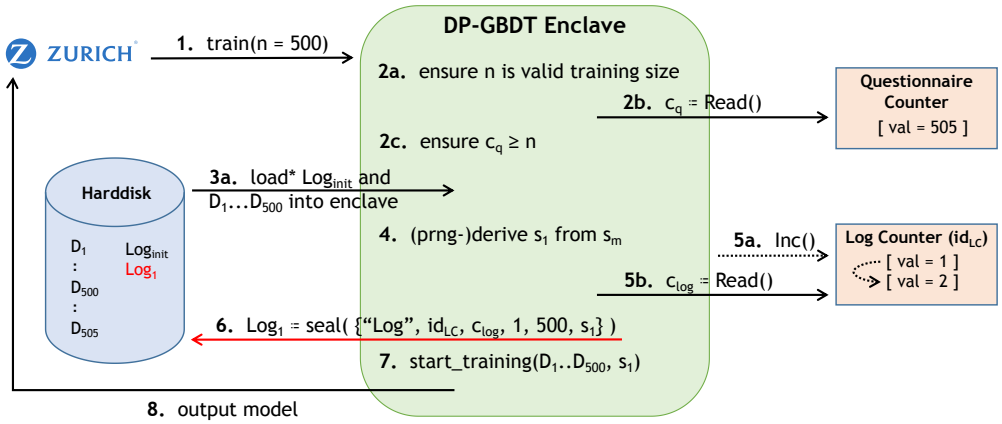


Figure 6.5: Refinement training

Model re-training Finally, let's consider how a previously trained piece of the model can be recreated. This is depicted in figure 6.6.

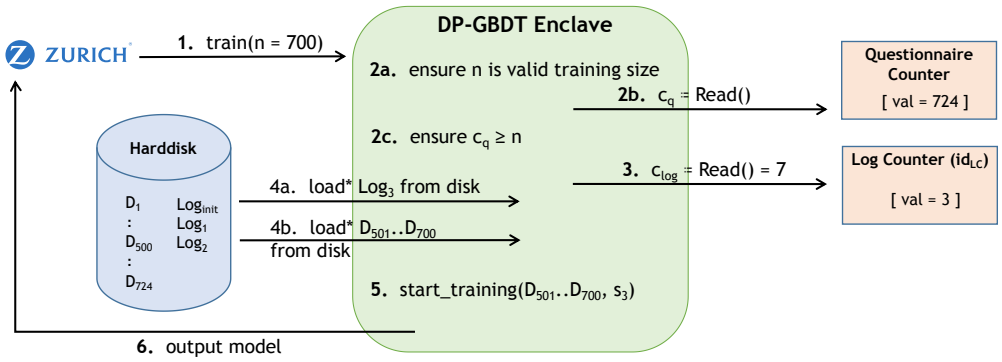


Figure 6.6: Re-training

Remarks: (3.) As described in section 6.1, the mapping (700→Log_3) is hardcoded in the enclave's source code.

6.3 Enclave migration/replication

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

Enclave migration Fortunately there already exists research on exactly this subject. This paragraph presents a short summary of the work of Alder et al. [2]. They propose an enclave migration approach that guarantees the consistency of its persistent state (such as sealed data and monotonic counters). This is achieved by using a migration library and by use of a separate

migration enclave on both the source and destination machine. A high level overview of the migration process can be found in figure 6.7. The main steps are:

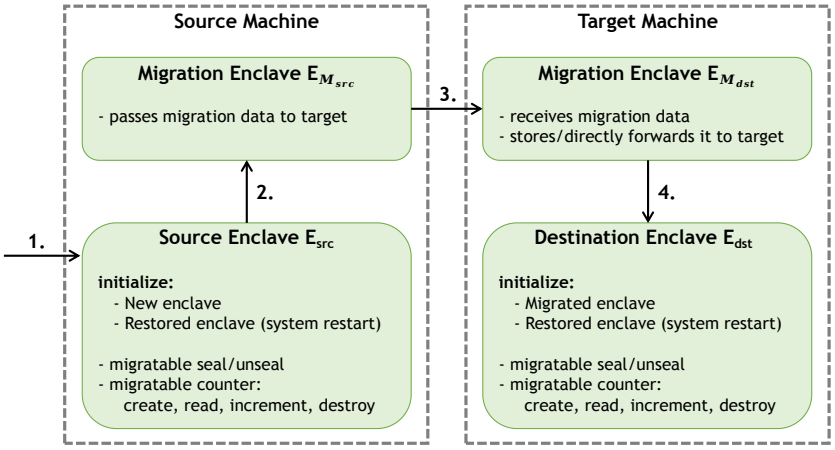


Figure 6.7: Enclave migration

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

Enclave replication Nevertheless, enclave migration might not be enough of a safety margin. What if the computer containing the GBDT enclave and it's corresponding monotonic counters explodes? To address such issues 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 6.9): First, the desired number of DP-GBDT enclave replicas is created. They are essentially clones with the exception of their public/private keys, and the monotonic counter ID's that are hardcoded in their source code. In a secure environment the DP-GBDT enclaves are then started up. Next, they announce their individual public keys. The public keys are then hardcoded into the master enclave's source code. The master enclave is started. The master enclave now takes over the part of generating a master seed s_m and distributes it to the other enclaves. The data collection phase now looks a bit different (figure 6.8): Insurance customers will not directly communicate with the DP-GBDT enclaves anymore, but with the master enclave instead. Similarly attestation will be carried out between the master enclave and the customers. But of course the customers still have to verify that the master's slaves are all legitimate DP-GBDT enclave clones. Upon reception of a customer questionnaire, the master enclave will distribute it to the GBDT enclaves. For this it has to decrypt and re-encrypt the questionnaire with the public keys of the slave enclaves.

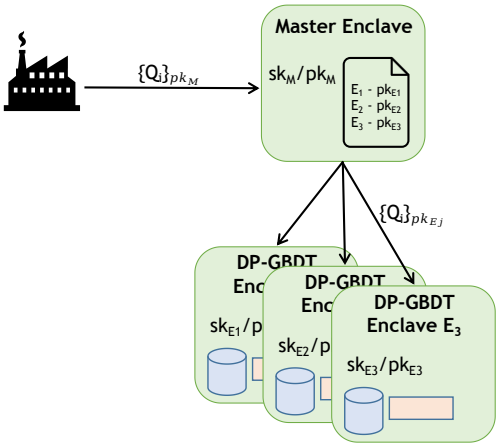


Figure 6.8: Enclave replication

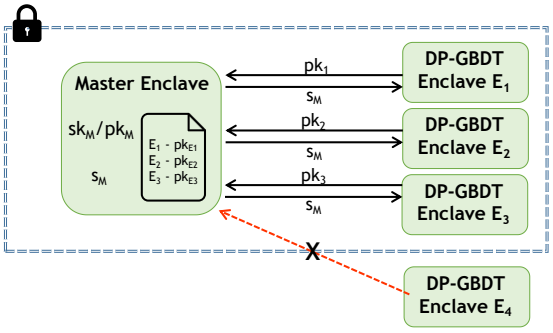


Figure 6.9: Replication setup

Some more remarks:

- The reason of doing the initial replication setup in a secure environment is to simplify the underlying process. If required, there actually exists research work that describes this process in an insecure environment [7]. Important is to keep the common goal in mind: We want to disallow an adversary to add his own local enclave to the list of slaves. So it's an extra precaution measure that should prevent a setting where an adversary could monitor the 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.
- What if the adversary drops one msg from the master enclave to the slaves? Then 1 has 1 less What must be prevented in such a system is a scenario, where an adversary briefly cuts the connection to one or more enclave, while a questionnaire is being distributed. This would lead to the affected DP-GBDT enclaves having different questionnaire counters when receiving future samples. Which in turn could be abused by an adversary that exchanges/duplicates etc. sealed data pieces among the enclaves. And subsequent training on different sets of questionnaires would then violate the differential privacy properties. right? A solution for this would be to have the master enclave check each slave enclave's questionnaire counter before distributing a new questionnaire. If one or more counters drag behind, the corresponding enclaves won't receive the new questionnaire. Houston, we still have a problem.

  Could switch half off one round, then the other half next round, then they would be even again
  We don't care about the case where the connection to all DP-GBDT enclaves is interrupted, as this corresponds to a DoS attack that might as well happen elsewhere and is out of scope.

small conclusion **Rough content:** do we need some small conclusion now? even though the conclusion chapter is following

also many things that you must not lose

Combining both migration and replication should be another challenge but doable. And with these measures now in place we should be ready for real world deployment.

Could someone increase the counter? yes. Would it be useful? No, it would render all previous work useless though.

Chapter 7

Conclusion

Rough content:
- accuracy bad for small pb, runtime ok
- future work (some theory (not perfectly dp), some implementation stuff)

7.1 Future Work

THINGS THAT ARE NOT PERFECTLY DP:

init_score auf classification und regression leaked welche werte drin sind

wenn zB init score 1.125 dann wissen wir daten sind teilbar durch 8

da muss man ein wenig addieren. + PB anpassen

GDF ist problematisch. Worst case wir ändern dataset um genau 1 datenpunkt und dann wandert der zum nächsten tree. "ein datenpunkt kann exakt 2 bäume beeinflussen, und wir rechnen momentan nur mit einem" TODO ??

- output the model, need a graphical representation at some point
- we need a PRNG for the algorithm.
- verification of constant time / other properties (gilchner)
- some statistical analysis on whether side channels are gone resp. use of a constant time verification tool like [31]

Appendix A

Detailed Performance Results

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: Hyperparameters for performance experiments

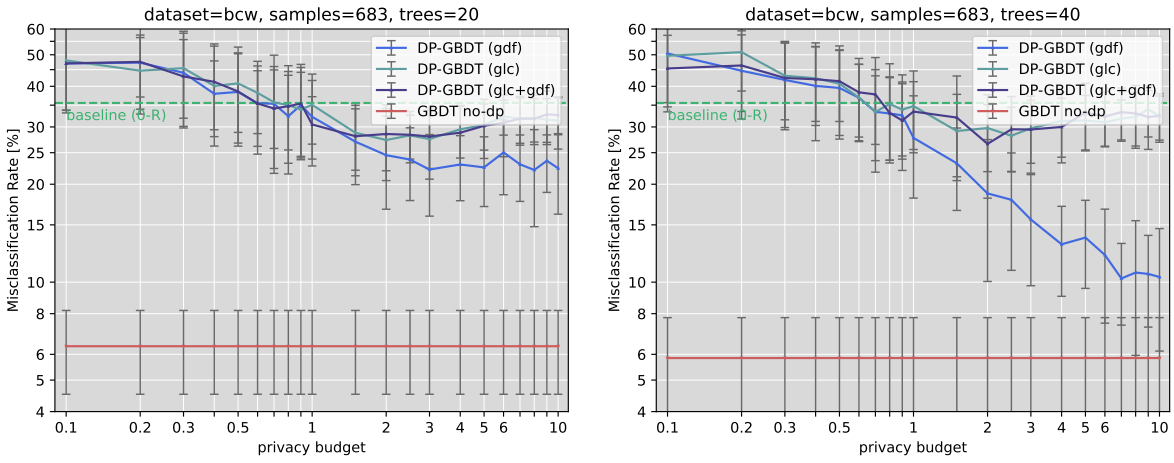


Figure A.1: BCW misclassification rate: ensemble of 20 resp. 40 trees

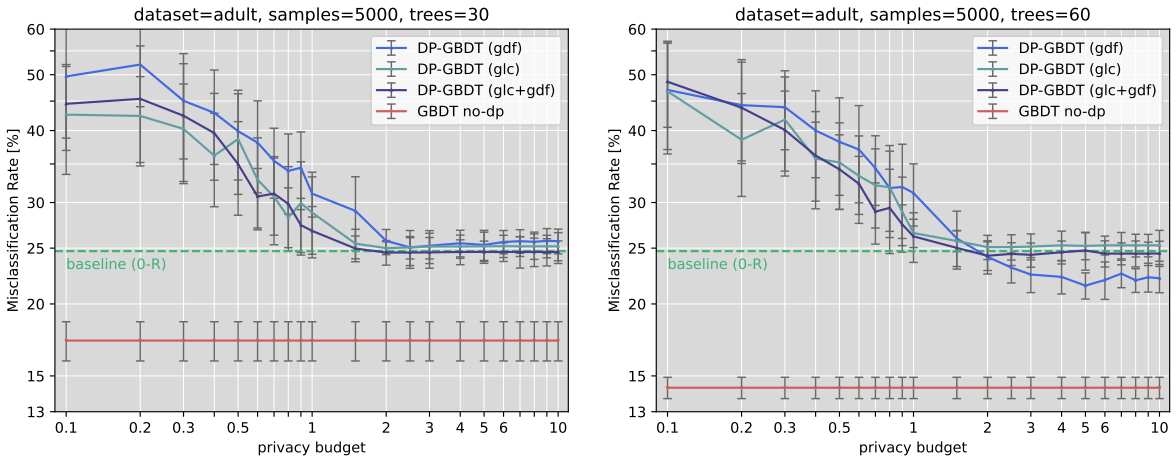


Figure A.2: Adult misclassification rate: ensemble of 30 resp. 60 trees

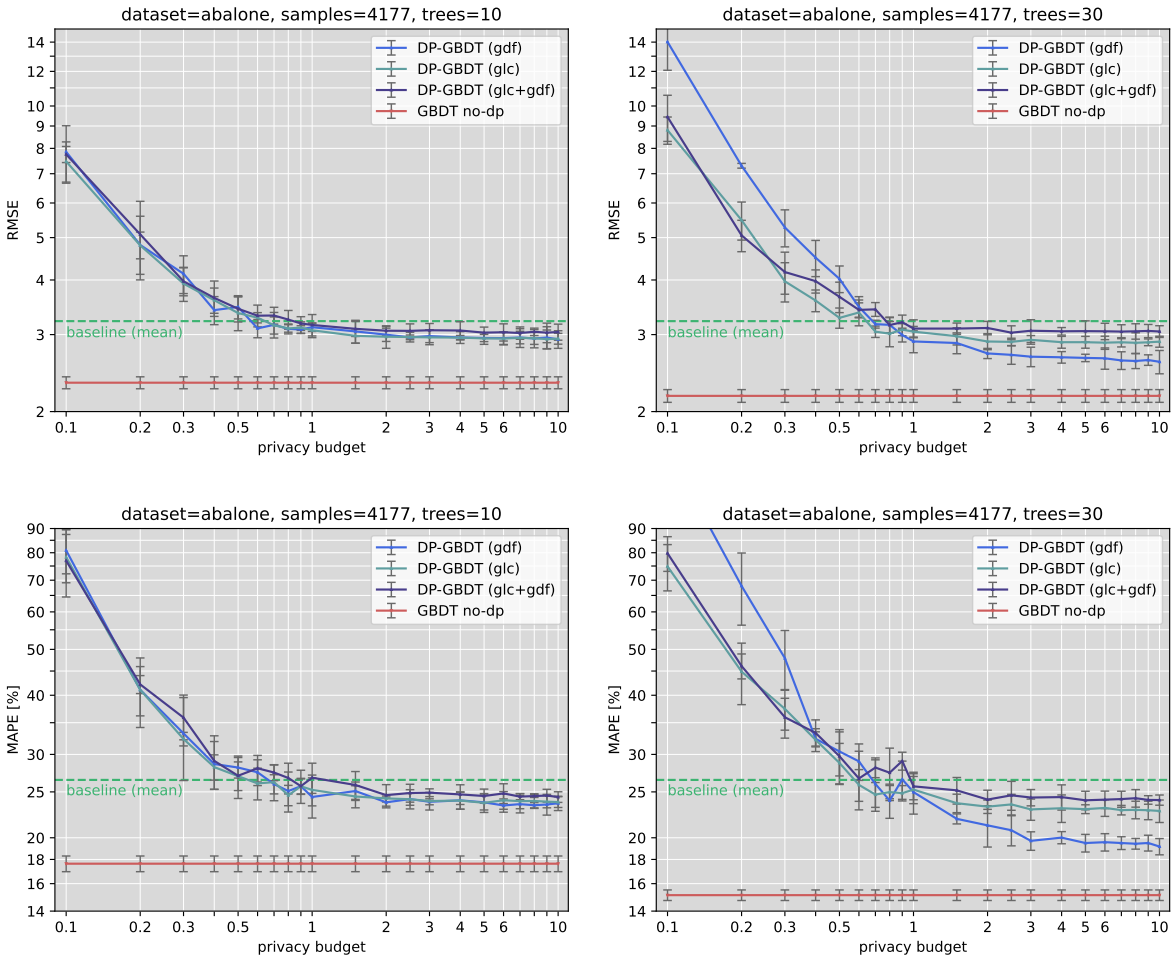


Figure A.3: Abalone RMSE/MAPE: Left side ensemble of 10 trees, right side 30 trees

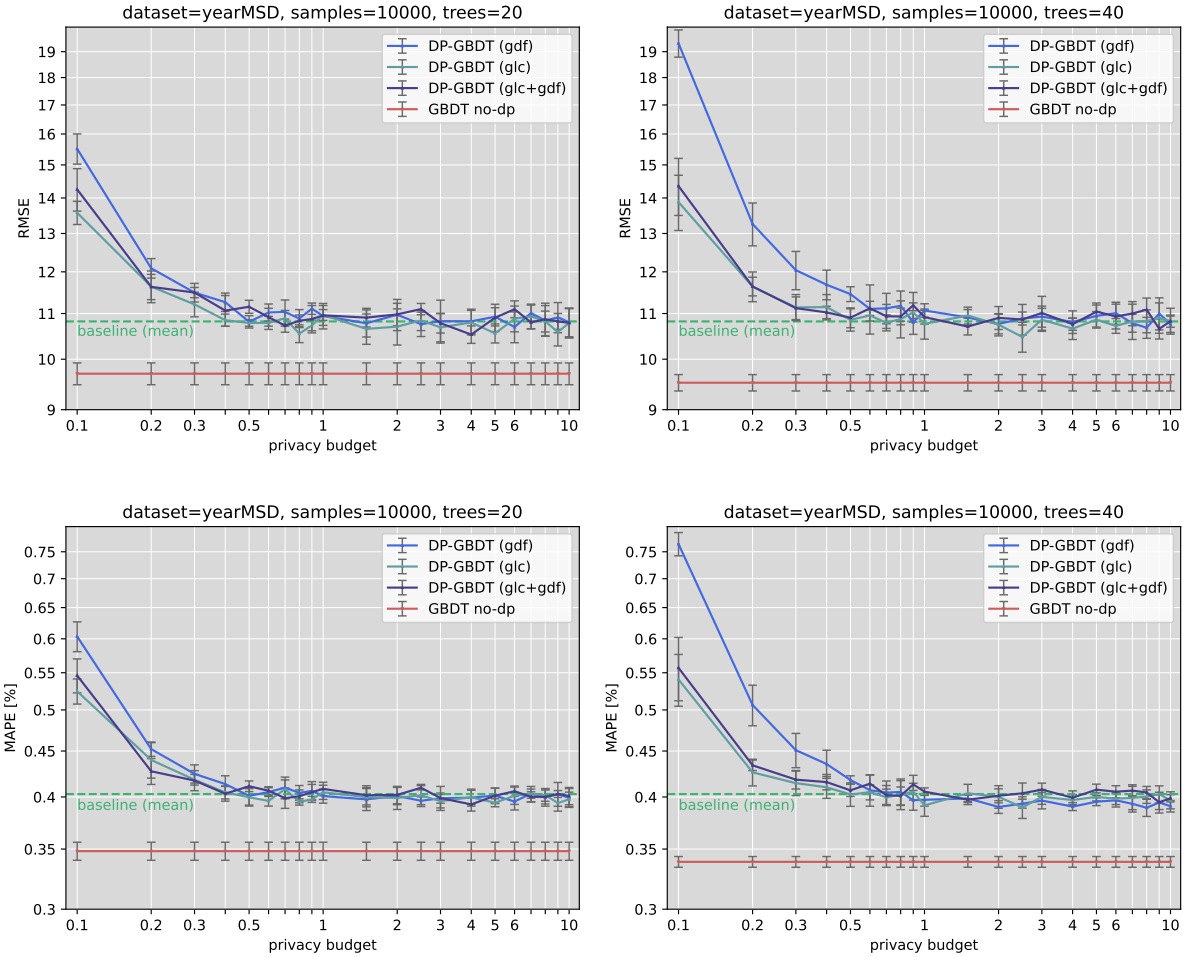


Figure A.4: Year RMSE/MAPE: Left side ensemble of 20 trees, right side 40 trees

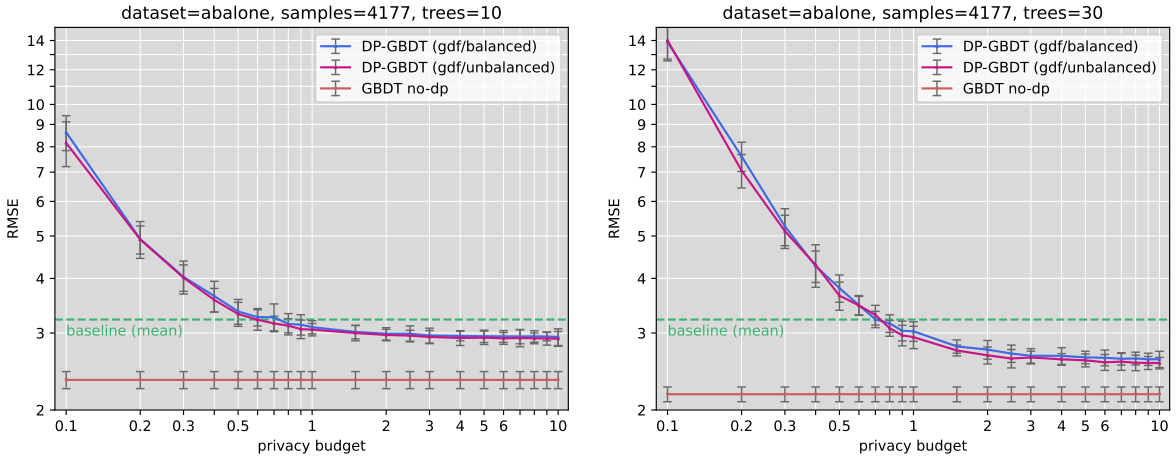


Figure A.5: Abalone RMSE balanced/unbalanced number of samples comparison: ensemble of 10 resp. 30 trees

Appendix B

Implementation Details

C++ GBDT parameters

nb_trees	The total number of trees in the model.
privacy_budget	The privacy budget available for the model.
learning_rate	The learning rate.
max_depth	The maximum depth for the trees.
min_samples_split	The minimum amount of samples required to split an internal node.
gradient_filtering	Whether or not to perform gradient based data filtering during training (only available on regression).
leaf_clipping	Whether or not to clip the leaves after training (only available on regression).
balance_partition	Balance sample partition for training among the trees. If set to True, all trees within an ensemble will receive an equal amount of training samples. If set to False, each tree will receive <x> samples where <x> is given in line 8 of the algorithm in the DPBoost paper.
use_decay	If True, internal node privacy budget has a decaying factor.
cat_idx	List of indices for categorical features.
num_idx	List of indices for numerical features.
l2_threshold	g_{L*} from [27] TODO
l2_lambda	The regularization parameter.
use_grid	Use grid to test out possible split values. Necessary for dp, but brings a reasonable performance penalty.
grid_borders	Specify grid borders.
grid_step_size	Grid step size.
cat_values	When using a grid for testing out different splits, this parameter can be used to specify the different feature values categorical features could theoretically take. This way splits on all possible values will be tested even though they don't actually appear in the dataset. As a result we can hide the fact whether a value is part of the dataset from a side channel observer.

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

scale_X_percentile Select the percentile for the confidence intervals for scale_X.

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

Adding new datasets to the project

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

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

Listing B.1: How to add a new dataset

Python GBDT bugs

Issues encountered with the python implementation [15] from the previous work .

- illegal second split, tree rejection, huge performance boost
- bugs in the parser (performance loss)
- changed gradient filtering
- bug with leaf_clipping not being passed
- compute_gain on real features
- use_decay formula wrong
- noifbinary

Mostly fixed in python

Profiling output

This illustrates the output from Intel VTune. Further you can see where the current bottlenecks now are of the finished C++ implementation.

TODO get a good snipped

Appendix C

Hardening Details

Rough content:
- This is just the raw content from 2 months ago, needs to be cleaned up a lot
- probably not going to keep all of it, just a few for illustration

TODO

- clean up color usage (probably use yellow to mark my “unnecessary hardening”)
- clean up right hand side comments
- check function definitions, calls, arguments etc to be matching with each other
- setstretch for smaller pseudocode font etc not working. (works in other overleafs)

Secrecy

Dataset and parameters

entity	secret
content of X	✓
X_cols_size	×
X_rows_size	×
content of y	✓
y_rows_size	×

parameter	secret
nb_trees	×
learning_rate	×
privacy_budget	×
task	×
max_depth	×
min_samples_split	×
balance_partition	×
gradient_filtering	×
leaf_clipping	×
scale_y	×
use_decay	×
l2_threshold	×
l2_lambda	×
cat_idx	×
num_idx	×

inferrable from those	secret
nb_samples per tree	×

While building a single tree

entity	secret	
X_subset	✓	
X_subset_cols_size	×	
X_subset_rows_size	✓	
y_subset	✓	
y_subset_rows_size	✓	
gradients	✓	
gradients_size	✓	

DP imperfections of the algorithm

- `init_score`, (=mean in regression, =most common feature in classification) leaks information about which feature values are in the dataset. Would need to add noise.
- `compute_gain` is done on the real data points. This was not adressed by the DPBoost paper. Would also need to add noise there.
- GDF is also problematic. changing one data point could have an impact on 2 trees.

main

No data dependent operations are done, therefore no side channel leakage. Securely getting the model parameters and dataset into and the resulting model out of the enclave is not among the challenges of this thesis.

Pseudocode 1: main

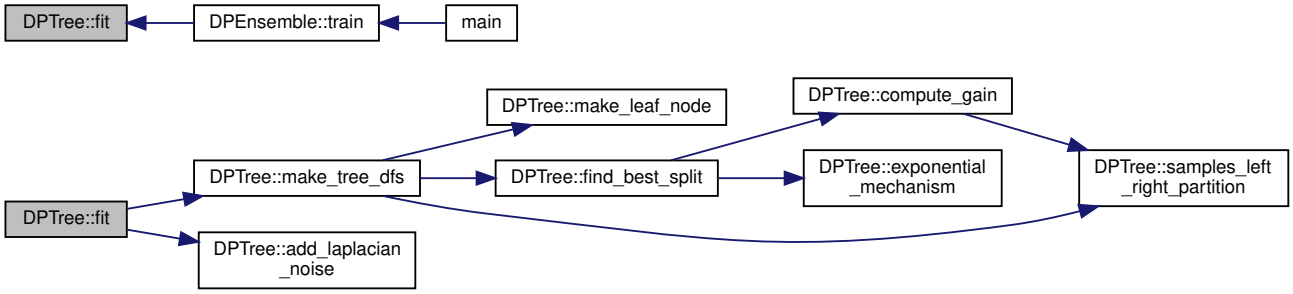
```
1 function main()
      // get parameters and dataset
2     parameters = get_params()
3     dataset = get_dataset()
      // create 5 train/test splits for cross validation
4     cv_splits = create_cross_val_inputs(dataset, 5)
      // do cross validation
5     for split in cv_splits do
6         ensemble = DPEnsemble(parameters)
7         ensemble.train(split.train)
          // predict using the test set
8         y_pred = ensemble.predict_ensemble(split.test.X)
          // compute score
9         score = compute_score(split.test.y, y_pred)
```

class DPTree

Methods

fit

Caller/call graph



Variables

- must not leak:

`dataset, leaves`

- can leak:

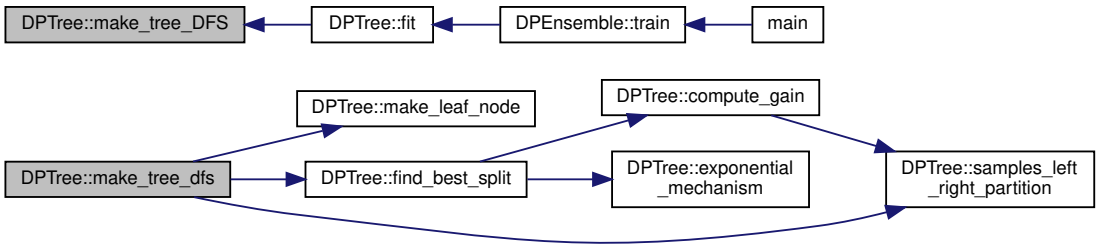
`params.*`

Pseudocode 2: DPTree::fit

```
1  function fit()
       // all samples are live at the start
2      live_samples = [1,2,3,...,dataset.length]
       // build tree
3      this→root_node = make_tree_dfs(live_samples, 0)
       // leaf_clipping
4      if params.leaf_clipping or !params.gradient_filtering then        ▷ params
5          threshold = params.l2 * (1 − η)^tree_index
6          for leaf in this→leaves do        ▷ number of leaves, which nodes are leaves
7              leaf.prediction = clamp(leaf.prediction, -threshold, threshold)
       // add laplace noise to leaf values
8      privacy_budget_for_leaf_nodes = tree_privacy_budget / 2
9      laplace_scale = params.Δv / privacy_budget_for_leaf_nodes
10     add_laplacian_noise(laplace_scale)
```

`make_tree_dfs`

Caller/call graph



Arguments / variables

- must not leak:

`dataset, X_transposed, live_samples, gradients`

- can leak:

`params.min_samples_split, params.max_depth, curr_depth`

Pseudocode 3: DPTree::make_tree_dfs

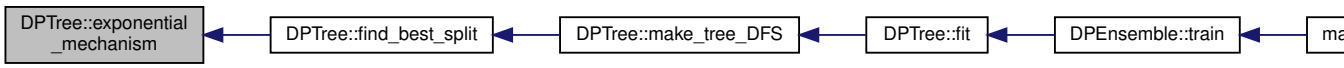
```
1  function make_tree_dfs(live_samples, curr_depth)                ▷ size of live_samples
       // max depth reached or not enough samples -> leaf node
2      if curr_depth == params.max_depth or len(live_samples) < params.min_samples_split
       then
3          TreeNode *leaf = make_leaf_node(curr_depth, live_samples)  ▷ both branch conditions
4          return leaf
       // find best split
5      TreeNode *node = find_best_split(X, gradients, live_samples, curr_depth)
       // no split found
6      if node.is_leaf then                                          ▷ number of leaves
7          return node
       // prepare the new live samples to continue recursion
8      lhs,rhs = samples_left_right_partition(X, node.feature_index, node.feature_value)
       ▷ sizes
9      for sample in live_samples do                                 ▷ size of live_samples
10         if lhs.contains(sample) then                              ▷ which samples go left/right
11             lhs_live_samples.insert(sample)
12         else
13             rhs_live_samples.insert(sample)
       // recurse
14     node→left = make_tree_dfs(lhs_live_samples, curr_depth+1)
15     node→right = make_tree_dfs(rhs_live_samples, curr_depth+1)
16     return node
```

Recursion leakage

- number of splits in the tree
- number of splits/leaves observable by watching memory allocations

`exponential_mechanism`

Caller graph



Arguments / variables

- must not leak:

`candidates`

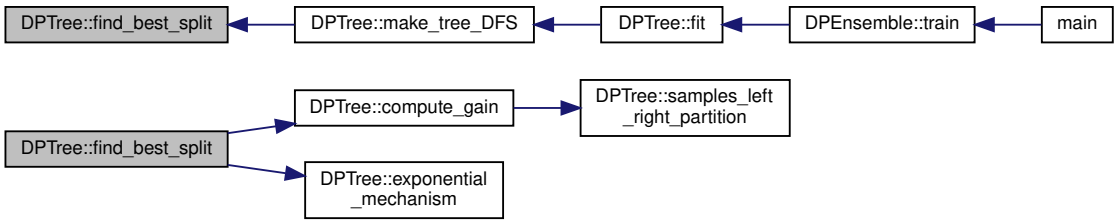
- can leak:

Pseudocode 4: DPTree::exponential_mechanism

```
1  function exponential_mechanism(candidates)                    ▷ number of candidates
       // if no split with positive gain, return, node will become a leaf
2      if cand.gain <= 0 forall cand in candidates then          ▷ no good split exists, leaf creation
3          return-1
       // calculate probabilities from the gains
4      for candidate in candidates do                            ▷ number of candidates
5          gains.append(candidate.gain)
6          if candidate.gain <= 0 then                           ▷ number of candidates with viable splits
7              probabilities.append(0)
8          else
9              lse = log ∑_i exp(gains_i)
10             probabilities.append(exp(candidate.gain − lse))
       // create a cumulative distribution from the probabilities, its values add up to
          1
11     partials = std::partial_sum(probabilities)
       // choose random value in [0,1]
12     rand_val = std::rand()
       // return the corresponding split
13     for i = 0 to i = partials.size() - 1 do
14         if partials[i] >= rand_val then
15             return i
16     return -1
```

`find_best_split`

Caller graph



Arguments / variables

- must not leak:

`X, gradients, live_samples`

- can leak:

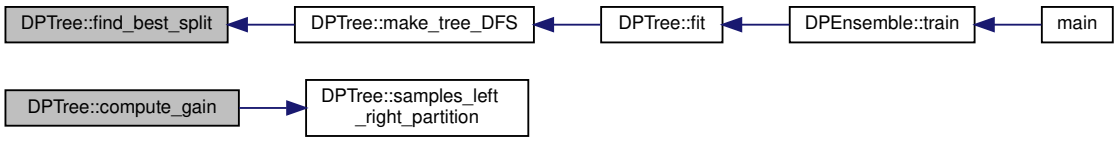
`params.*, tree_budget, curr_depth`

Pseudocode 5: DPTree::find_best_split

```
1  function find_best_split(X, gradients, live_samples, curr_depth)   ▷ size of X, gradients
       // determine node privacy budget
2      if params.use_decay then                                        ▷ params.use_decay
3          if curr_depth == 0 then                                     ▷ curr_depth == 0
4              node_budget = tree_budget / (2*(2^(max_depth+1)+2^(curr_depth+1)))
5          else
6              node_budget = tree_budget / (2*2^(curr_depth+1))
7      else
8          node_budget = tree_budget / (2*max_depth)
       // iterate over all possible splits
9      for feature_index in features do                                ▷ number of cols in X
10         for feature_value in X[feature_index] do                    ▷ number of rows in X
11             if "already encountered feature_value" then             ▷ number of unique feature values
12                 continue
13             gain = compute_gain(X, gradients, live_samples, feature_index, feature_value)
14             if gain < 0 then                                        ▷ number of splits with no gain
15                 continue
16             gain = node_budget*gain / (2*Δg)
17             candidates.insert(Candidate(feature_index, feature_value, gain))   ▷ number of candidates
       // choose a split using the exponential mechanism
18     index = exponential_mechanism(candidates)
       // construct the node
19     TreeNode *node = new TreeNode(candidates[index])                ▷ internal node vs. leaf
20     return node
```

compute_gain

Caller/call graph



Arguments / variables

- must not leak:

`X, gradients, live_samples`

- can leak:

`params.l2_lambda, feature_index, feature_value`

Pseudocode 6: 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                                 ▷ useless split
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)
9      rhs_gain = rhs_gain^2 / (rhs_size+params.l2_lambda)
10     total_gain = max(lhs_gain + rhs_gain, 0)        ▷ max might leak whether total_gain < 0
11     return total_gain
```

make_leaf_node

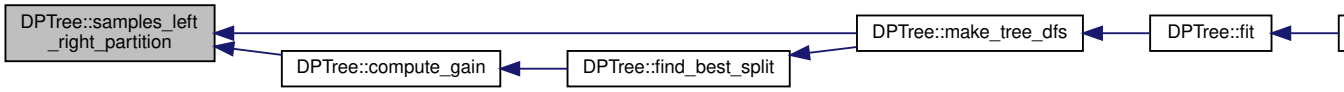
Caller graph

Pseudocode 7: DPTree::make_leaf_node

```
1  function make_leaf_node(curr_depth, live_samples)                ▷ live_samples size
2      leaf = new TreeNode(curr_depth)
       // compute prediction
3      for sample_index in live_samples do                          ▷ live_samples size
4          gradients.append(dataset.gradients[sample_index])        ▷ dataset.gradients memory
               access pattern
5      leaf.prediction = -∑gradients / (gradients.size()+params.l2_lambda)
6      return leaf
```

samples_left_right_partition

Caller graph



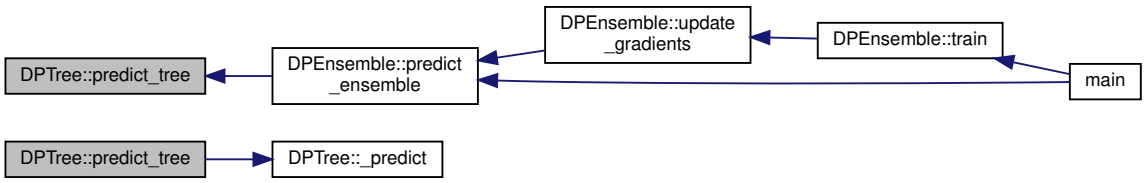
Pseudocode 8: DPTree::samples_left_right_partition

```
1  function samples_left_right_partition(X, feature_index, feature_value)   ▷ X size
2      if feature_index in params.cat_idx then                               ▷ cat/num feature split
           // categorical feature
3          for sample_index = 0 to sample_index < X.num_rows do              ▷ X size
4              if X[sample_index][feature_index] == feature_value then       ▷ lhs,rhs sizes and
                   access pattern
5                  lhs.append(sample_index)
6              else
7                  rhs.append(sample_index)
8      else
           // numerical feature
9          for sample in X[feature_index] do                                 ▷ X size
10             if X[sample_index][feature_index] < feature_value then        ▷ lhs,rhs sizes and access
                   pattern
11                 lhs.append(sample_index)
12             else
13                 rhs.append(sample_index)
14     return lhs,rhs
```

Whether the yellow box is secret depends on the context from where the function is called. If it's from find_best_split / compute_gain where we're just trying out all possible splits, then it's not secret. However if we recreate the split that we found and chosen before, then it's secret.

predict_tree

Caller/call graph

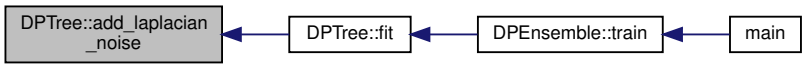


Pseudocode 9: DPTree::predict_tree

```
1  function predict_tree(X)                                              ▷ X size
2      for row in X do
3          predictions.append(_predict(row, this→root_node))
4      return predictions
5
6  function _predict(row, node)          ▷ must not leak whether a sample row goes left/right
7      if node.is_leaf then
8          return node.prediction
9      if node.split_attribute in params.cat_idx then
           // categorical feature
10         if row[split_attribute] == node.split_value then
11             return _predict(row, node.left_child)
12     else
           // numerical feature
13         if row[split_attribute] < node.split_value then
14             return _predict(row, node.left_child)
15     return _predict(row, node.right_child)
```

add_laplacian_noise

Caller/call graph



Pseudocode 10: DPTree::add_laplacian_noise

```
1  function add_laplacian_noise(laplace_scale)
2      lap = new LaplaceDistribution(laplace_scale)
       // add noise from laplace distribution to leaves
3      for leaf in this.leaves do                                  ▷ number of leaves
4          noise = lap.return_a_random_variable()
5          leaf.prediction += noise
```

class DPEnsemble

TODO can I get around doing it for DPEnsemble? because it should be largely data independent.

- I think it would be consequent to hide the GDF stuff as well, since it gradients kinda depend on data. However, possible that it is also "unnecessary" and already assumed leaking in the proof.

Methods

train

Caller graph

TODO

Call graph

TODO

Variables

- must not leak:

TODO

- can leak:

params.*, tree_params.*

Pseudocode 11: DPEnsemble::train

```
1  function train(dataset)
       // compute initial prediction
2      init_score = compute_init_score(dataset.y)
       // each tree gets the full budget since they train on distinct data
3      tree_privacy_budget = params.privacy_budget
       // train all trees
4      for tree_index = 0 to tree_index = nb_trees - 1 do                    ▷ bla
           // init/update gradients
5          update_gradients(dataset.gradients, tree_index)
           // sensitivity for internal nodes
6          tree_params.Δg = 3 * (params.l2_threshold)^2
           // sensitivity for leaf nodes
7          if params.gradient_filtering or !params.leaf_clipping then
8              tree_params.Δv = params.l2_threshold / 1+params.l2_lambda
9          else
10             tree_params.Δv =
                   min(params.l2_threshold / 1+params.l2_lambda, 2 * params.l2_threshold * (1 − η)^tree_index)
           // determine number of rows
11         if params.balance_partition then
12             number_of_rows = |D| / nb_trees−tree_index
13         else
14             number_of_rows = |D|η(1−η)^tree_index / 1−(1−η)^nb_trees
```

`predict`

`update_gradients`

`add_laplacian_noise`

`remove_rows`

`get_subset`

other classes

Bibliography

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

[2] 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.

[3] Jose Bacelar Almeida et al. “Verifying Constant-Time Implementations”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

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

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

[6] 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.

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

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

[9] Wenxin Du. *Differentially Private Confidence Intervals*. https://github.com/wxindu/dp-conf-int. 2019.

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

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

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

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

[14] Daniel Genkin, Adi Shamir, and Eran Tromer. *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis*. Cryptology ePrint Archive, Report 2013/857. https://ia.cr/2013/857. 2013.

[15] Theo Giovanna. *Differentially Private Gradient Boosted Decision Trees*. URL: https://github.com/giovannt0/dpgbdt (visited on 10/13/2021).

[16] Tyler Hunt et al. "Ryoan". In: *ACM Transactions on Computer Systems* 35.4 (Dec. 2018), pp. 1–32. DOI: 10.1145/3231594. URL: https://doi.org/10.1145/3231594.

[17] Intel. *Intel Advisor*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html (visited on 10/15/2021).

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

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

[20] 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).

[21] Intel. *Intel VTune Profiler*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html (visited on 10/15/2021).

[22] 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.

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

[24] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology — CRYPTO' 99*. Springer Berlin Heidelberg, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25. URL: https://doi.org/10.1007/3-540-48405-1_25.

[25] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Springer Berlin Heidelberg, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9. URL: https://doi.org/10.1007/3-540-68697-5_9.

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

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

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

[29] E. De Mulder et al. “Differential power and electromagnetic attacks on a FPGA implementation of elliptic curve cryptosystems”. In: *Computers & Electrical Engineering* 33.5-6 (Sept. 2007), pp. 367–382. DOI: 10.1016/j.compeleceng.2007.05.009. URL: https://doi.org/10.1016/j.compeleceng.2007.05.009.

[30] 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.

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

[32] 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.

[33] 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.

[34] V. Zverovich and J. Mueller. *fmt formatting library*. https://github.com/fmtlib/fmt. 2021.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

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

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

With my signature I confirm that

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

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

Place, date	**Signature(s)**

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