

Chapter 3

Overview

As already indicated, the starting point of this thesis was **not** zero. The DPBoost paper ([45], 2020) provided the theory behind DP-GBDT learning. Moreover, Théo Giovanna built a first Python implementation¹ of said algorithm during the course of his master's thesis (submitted Feb. 2021). Starting from this point, the high level objectives were in short: Porting the code to C++, putting it into an enclave, harden it against side-channels, come up with a realistic strategy for deployment in practice.

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

3.1 Solution overview

Figure 3.1, nld figure 3.1

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

3.2 Threat model

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

¹<https://github.com/giovannt0/dpgbdt>

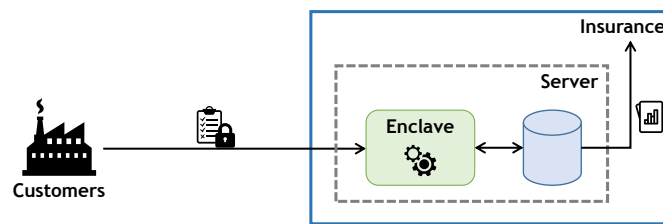


Figure 3.1: Insurance use case overview

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

TM1, Goal: Customers want privacy even if the insurance is malicious TM1 represents a malicious service provider. This adversary has full control over both the software (including OS or hypervisor) and the hardware of his system. Thus he is able to start or stop processes at any point, set their affinity and modify them at runtime. He can also read and write to any memory region except the enclave memory, map any virtual page to any physical one and dynamically re-map pages. He can run multiple instances of an enclave in parallel. Moreover he has various network control capabilities: He may tamper with, delete, delay, reorder, or replay any network packets exchanged between the server and the participating parties. We also expect him to closely monitor the program execution for side-channels. Lastly, he has full access to state-of-the-art enclave attack frameworks like SGX-Step [64]. The goal of the malicious service provider is to gather as much information as possible about customers participating in the training process. Note that, from a customers perspective, service provider and insurance might very well be the same entity. Or similarly, the two might collude to steal secrets from the insurance customers.

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

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

3.3 Goals

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

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

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

- Data collection over an extended time period
- Training on newly acquired customer data to improve the existing model output²
- Training on old data after output model loss²
- Enclave replication and migration for maintaining persistent state

²This is not trivial whilst not violating differential privacy


Side-channel hardening



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

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

5.1 Known tools and techniques



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

Constant-time programming If implemented naively, many cryptographic algorithms perform computations in non-constant time. Such timing variation can cause information leakage if they are dependant on secret parameters. With adequate knowledge of the application, a detailed statistical analysis could even result in full recovery of the secret values. Constant-time programming refers to a collection of programming techniques that are used to eliminate timing side-channels. The key idea is that it  most impossible to hide what kinds of computations a program is performing. What can be done, however, is to produce code whose control flow is independent from any secret information. As a general rule of thumb: The input to an instruction may only contain secret information if the input does not influence *what* resources will be used and *for how long*. This means, for example, that secret values must not be used in branching conditions or to index memory addresses. What further complicates matters: Modern CPUs generally provide a large set of instructions, but not all of them execute in constant-time. Common examples are integer division (smaller numbers divide faster), floating point multiplication, square root and common type conversions [31]. Additional issues can arise from gaps in the language standards. The C-standard, for instance, generally  define the relation from source code to hardware instructions. This can result in unexpected non-constant-time instruction sequences.

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

Blinding Another popular mitigation approach for timing side-channels is blinding. It is essentially the process of obfuscating input using random data before performing a non-constant-time operation. After the operation is complete, the output is de-obfuscated using the same randomness. Put differently, the operation is still non-constant-time, but the secret value is hidden. Two well-known use cases of this approach are old RSA and ECDH implementations [16, 59].

Oblivious RAM Oblivious RAM (ORAM) was introduced for the purpose of enabling secure program execution in untrusted environments. This is accomplished by obfuscating a program's memory access pattern to make it appear random and independent of the actual sequence of accesses made by the program. By utilizing ORAM techniques (e.g. [62]), a non-oblivious algorithm can be compiled to its oblivious counterpart at the expense of a poly-logarithmic amortized cost per access [11]. While this is a great result, it also means that the algorithms become exceedingly slow when processing large amounts of data. As a result, it is generally less suitable for machine learning.

Data-oblivious algorithms A less universal but very effective tactic is to design/rewrite the algorithm itself in a data-oblivious way. In order to prevent an attacker from being able to infer any knowledge about the underlying data, oblivious algorithms need to produce indistinguishable traces of memory and network accesses. Detectable variances must be purely based on public information and unrelated to the input data. This implies the use of oblivious data structures and removal of all data-dependent access patterns. In most cases, manually transformed algorithms are much more efficient than using ORAM (where runtime can easily increase by one or more orders of magnitude [70]). But of course, the rewriting process takes time and is prone to errors. What helps, however, is that over the past years, researchers have produced a large range of oblivious algorithms and building blocks that are at a developer's disposal (e.g. [9, 18]).

5.1.1 Compilation and verification

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

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

<pre> 1 int test(bool b1, bool b2) 2 { 3 bool result = b1 and b2; 4 return result; 5 } </pre>	<pre> 1 test(bool, bool): 2 testb %dil, %dil 3 je ..B1.3 4 xorl %eax, %eax 5 testb %sil, %sil 6 setne %al 7 ret 8 ..B1.3: 9 xorl %eax, %eax 10 ret </pre>
<pre> 1 test(bool, bool): 2 andl %edi, %esi 3 movzbl %sil, %eax 4 ret </pre>	

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

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

5.1.2 Tool assisted hardening

Recent research proposed multiple automatic hardening tools (e.g. Constantine [13], Obfuscuro [3] or Raccoon [56]). The approaches include: Complete linearization of secret-dependent control and data flows, or execution of extraneous "decoy" program paths. Depending on the exact approach, automatic hardening tools introduce substantial runtime overhead. Tool-assisted verification of constant-time properties is also possible [63, 57, 7]. However, most of these methods again demand meticulous program code instrumentation, to i.e. mark variables or memory locations that might contain secret values at some point.

5.2 Chosen approach

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

5.2.1 General measures

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



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


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


5.2.2 DP-GBDT-specific measures


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

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

Oblivious sample distribution There are two factors that affect how samples for the individual trees are chosen: We can use a balanced  approach where each tree gets the same amount of samples, or an unbalanced approach, where earlier trees get more samples according to formula (DPBoost [45], Algorithm 2, line 8). The second influential factor is GDF. If GDF is enabled, samples with gradients below a certain threshold will be chosen first. This is of course problematic, as the current gradients depend on the previous tree built from secret data. The key observation that inspired our solution was: We do not need to hide *which* samples are chosen. It is enough to hide *why* a sample was chosen. It's sufficient if it's indistinguishable whether a sample is chosen randomly or because of it's favourable gradient. 





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

To prevent leaking the shape of the tree under construction it's necessary to: (i) Add a fixed number of nodes to each tree, and (ii) keep the order in which nodes are added independent of the data [44]. This means that we always build a full binary trees. Some nodes will end up being dummy nodes, meaning no sample would actually arrive there. Since, as described in the previous paragraphs, we always scan through all samples at each node, dummy nodes are indistinguishable from real nodes. We further simulate the creation of a leaf node at each node in the tree. Hence a side-channel adversary has no idea whether an internal node, dummy node or leaf node has just been added. Moreover, as we are building the tree in a depth-first (DFS)  manner and always go down to the maximum depth, we do not leak any information through order. Furthermore, in case geometric leaf clipping (GLC) is enabled, the clipping operation is done on all nodes to hide which ones are real leaf nodes. Additionally, the exponential mechanism has to be hardened to hide which split is chosen.

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

5.3 Technical details

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

Compilation and verification What is the best method to verify the compiled output in a manageable manner? The newly created  time functions were all defined in a separate file and namespace. Further, inlining is disabled with vendor-specific keywords. This way you can more conveniently check the assembly output of the hardened functions, as they are all gathered in one place. Another extra layer of safety measure that we added to the hardened functions is selective use of the volatile keyword. Its purpose is essentially to prevent unwanted compiler optimization. Regarding compiler optimization flags, we decided to take a relatively conservative approach: We opted against using very aggressive compiler optimizations, as it significantly lowered traceability and comprehensibility of the assembly code.  Compilation results were inspected after usage of both `-O0` and `-O1` gcc flags. No hardening  resp. side-channel violations were visible. 

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



```

1  __attribute__((noinline)) bool constant_time::logical_not(bool a)
2  {
3      // bitwise XOR for const time
4      return (bool) (value_barrier((unsigned) a) ^ 1u);
5  }

```

Listing 5.1: Constant-time logical NOT

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

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



```

1  template <typename T>
2  __attribute__((noinline)) T select(bool cond, T a, T b)
3  {
4      // result = cond * a + !cond * b
5      return value_barrier(cond) * value_barrier(a) +
6             value_barrier(!cond) * value_barrier(b);
7  }

```

Listing 5.2: Constant-time ternary operator

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



```

1  // O(n^2) bubblesort
2  for (size_t n=vec.size(); n>1; --n){
3      for (size_t i=0; i<n-1; ++i){
4          // swap pair if necessary
5          bool condition = vec[i] > vec[i+1];
6          T temp = vec[i];
7          vec[i] = constant_time::select(condition, vec[i+1], vec[i]);
8          vec[i+1] = constant_time::select(condition, temp, vec[i+1]);
9      }
10 }

```

Listing 5.3: Oblivious sort

Arithmetic leakage Inherently non-constant-time instructions, such as floating point arithmetic, were only hardened in the most performance critical section of the code. This is where the majority of execution time is spent, as depicted in A. The affected functions are:

- samples_left_right_partition
- ... TODO

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

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

```

1  // recursively walk through the decision tree
2  double predict(vector<double> sample_row, TreeNode *node)
3  {
4      // base case
5      if(node->is_leaf()){
6          return node->prediction;
7      }
8
9      // recurse left or right
10     double sample_value = sample_row[node->split_attr];
11     if (sample_value < node->split_value){
12         return predict(sample_row, node->left);
13     }
14     return predict(sample_row, node->right);
15 }

```

Listing 5.4: Non-hardened inference/prediction

```
1 // recursively walk through the decision tree
2 double predict(vector<double> sample_row, TreeNode *node)
3 {
4     // always go down to max_depth
5     if(node->depth < max_depth){
6
7         double sample_value = sample_row[node->split_attr];
8
9         // hide the real path a sample takes, go down both paths
10        double left_result = predict(sample_row, node->left);
11        double right_result = predict(sample_row, node->right);
12
13        // decide whether we take the value from the left or right child
14        bool is_smaller = constant_time::smaller(sample_value, node->split_value);
15        double child_value = constant_time::select(is_smaller, left_result, right_result);
16    }
17    // if we are a leaf, take own value, otherwise we take the child's value
18    return constant_time::select(node->is_leaf, node->prediction, child_value);
19 }
```

Listing 5.5: Hardened inference/prediction