Chapter 5

Side-channel hardening

Rough content: I'm not too happy with this section. I'm struggling a bit to justify our hardening approach

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

Goals The main goal 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 address traces, data size, and cache usage. We are not trying to eliminate every single bit of leakage. Instead, we eliminate just enough leakage to achieve ε-differential privacy.

5.1 Known tools and techniques

As side-channel attacks is generally a well researched topic, there are various established tools and techniques available to the developer. We therefore start by presenting a selection of largely adopted practices. First we'll talk about constant time programming, a well known technique to combat timing side channels. This is followed by a section on memory access pattern side-channels and corresponding countermeasures. Thereafter, an analysis of complications caused by compilers is presented. We will conclude with a brief overview of tools for automatic hardening/verification. The use of such tools can prevent programmers from falling into one of the many traps of manual hardening.

5.1.1 Timing side-channels

Due to performance optimizations, 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 containing implementation, a detailed statistical analysis could even result in full recovery of the secret values.

Constant-time programming Constant-time programming refers to a collection of programming techniques that are used to eliminate timing side-channels. It's almost impossible to hide what kinds of computations a program is performing. What we can do however, is write the code in a way that the 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 which resources will be used and for how long. This means for example: Secret values must not be used in branching conditions or to index memory addresses. But as if this wasn't enough already: CPUs today usually provides a wide 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 several type conversions [22]. Further issues can arise from gaps in the language standards. The C-standard for instance generally doesn't define the relation from source code to hardware instructions. This can result in unexpected non-constant-time instruction sequences. Operations that do not have a directly corresponding assembly instruction are naturally more prone to this issue.

Blinding Another popular mitigation approach for timing side-channels is called blinding. In a nutshell this means that before a non-constant-time operation is executed, the input is obfuscated using random data. After the operation is complete the ouput is de-obfuscated using the same randomness. Two well-known real-world use cases of this approach include RSA decryption and Elliptic Curve Diffie-Hellman [9, 48].

5.1.2 Memory side-channels

Memory side-channels are also a popular concern, and many attacks have been discovered in this direction. Measuring the timing difference between cache hits and misses to infer the memory access of the victim process, for instance. Or similarly, it was also shown that purposely induced page faults can be leveraged to examine the memory access patterns of a victim process. The next section covers two popular mitigation strategies.

Oblivious RAM Oblivious RAM (ORAM) was introduced for the purpose of enabling secure program execution in untrusted environments. The goal is to essentially to make memory accesses appear random and independent of the actual sequence of accesses made by the program. By utilizing ORAM techniques (e.g. [25, 51]), a non-oblivious algorithm can be rendered oblivious, at the expense of a polylogarithmic amortized cost per access [6]. While this is a great result, it also means that algorithms become exceedingly slow when processing large amounts of data, which is inevitable machine learning.

Data-oblivious algorithms A less universal but very effective tactic is to design 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, that are purely based on public information and unrelated to the input data. This means usage of data-oblivious data structures and removal of all data-dependent access patterns. Data-oblivious algorithms tend to be more efficient than using ORAM (sometimes one or more orders of magnitude [56]). But the implementations are more complicated and prone to errors. However, you are not completely on your own. Over the past years, researchers have proposed a large array of specialized oblivious algorithms and building blocks for all kinds of algorithms (e.g. [55, 5, 11]).

5.1.3 Compilation and verification

How can we ensure that the compiler does not optimize away our 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. Depicted are two compilation results from the same piece of source code (top left). 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.

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

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

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

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

As a result it is not just sufficient to ensure that only 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 degree. And keep in mind that changing the compiler or modifying unrelated code parts may suddenly change a compiler's output.

5.1.4 Tool assisted hardening

In theory there are multiple automatic hardening tools available (e.g. Constantine [8], Obfuscuro [1] or Raccoon [45]). 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 will also introduce substantial runtime overhead. On the other hand there are several approaches to that do not actually perform, but instead verify constant-time properties [52, 46, 3]. However, most of these methods again demand meticulous program code instrumentation, to i.e. mark memory locations containing secret values.

5.2 Chosen approach

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

5.2.1 General measures

Closer inspection showed that most of the automatic hardening solutions are still in an early/incomplete stage and not quite ready for usage in practice. Some of the papers did not publish their tool all. Others provide tools that mostly serve as a proof of concept, that work on specific small examples. Then again there are tools that require large amounts of annotations and extra information to be added to the source code. This would be very challenging for an algorithm of our size. Therefore we chose a manual hardening approach on source code level. This way we can leverage the knowledge of the algorithm to pinpoint critical location that require attention. This should also significantly reduce runtime overhead. The most important principles in our case are:

- Avoid branch conditions affected by secret data
- Avoid memory look-ups indexed by secret data
- Avoid secret-dependent loop bounds
- Prevent compiler interference with security-critical operations
- Clean memory of secret data
- Use strong randomness

Apart from these high-level rules, several clearly constant-time violating or non-data-oblivious primitives come to mind:

- logical boolean operators
- comparators
- ternary operator
- sorting
- min/max functions

As these constructs appear quite frequently in code, constant-time and data oblivious versions of all these functions were created and inserted into the algorithm. Details on the implementation of these primitives can be found in section 5.3.

As described in the requirements section, we completely eliminate leakage through memory access patterns. This means for example: In order to retrieve a value at a secret dependent index from an array, every single element should be touched. Although this entails a significant execution time penalty, it safeguards the implementation from future attacks with even more powerful and finer graded leakage capabilities.

Regarding inherently non-constant-time instructions such as floating point arithmetic, we decided against replacing them entirely. It's not only a major task, it also heavily perturbs code readability. Also for floating point numbers there are fewer constant time libraries/code available. We 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 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 two main problematic parts in the DP-GBDT algorithm. First of all, constant-time and data-oblivious algorithms for both training and inference (prediction) have to be designed. Secondly, gradient data filtering (GDF) causes dependence on secret information (gradients) when choosing the samples for the following trees. The final paragraph of this section covers oblivious tree construction, which can be considered a non-obligatory bonus task. These hardening steps were undertaken in this project, even though they are not necessary for differential privacy. Note, we do not try to hide the number of samples participating in the training. This would complicate matters greatly.

Oblivious training and inference Whether it's for training or inference, the key thing that needs to be hidden is the path that samples take in the decision tree. In GBDT training each new node only uses the data that belongs to that node. In order to conceal which data samples belong to the node, we either need to access each sample obliviously, or, 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 now done on all samples, and only in the end the dummy results are discarded using the indication vector. Oblivious inference 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.

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 (TODO ref background). The second influential factor is GDF. If GDF is enabled, samples with gradients above 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 differential privacy in our design. 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.

For the purpose of preventing data or tree leakage, it is necessary to: (i) add a fixed number of nodes to each tree; and (ii) keep the order in which nodes are added independent of the data [37]. 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, the dummy nodes are indistinguishable from real nodes. We further simulate the creation of a leaf at each node. So internal nodes, dummy nodes and leaf nodes are basically indistinguishable. Further, as we are building the tree in a DFS manner, and always go to the full depth we do not leak any information through order.

Example of hardened and non-hardened function?

In case leaf clipping is enabled, the clipping operation is done on all nodes to hide which ones are real leaf nodes. Further a variety of things need to be hidden during the search for the best split, the most important ones being:

- Testing all possible splits must happen obliviously (we must not leak whether samples would go left/right).
- We must not leak the number of splits with no gain.
- The exponential mechanism has to be hardened to hide which split is chosen.
- We must not leak the number of unique feature values

To illustrate this process in greater detail, appendix C contains pseudocode of several tree-building functions with all side-channel affected regions highlighted.

Compilation and verification What is the best method to verify the compiled output in a manageable manner? The newly created constant-time functions are all defined in a separate file and namespace. Further, inlining is disabled with vendor-specific keywords. This way you can more conveniently check all hardened functions as they are all gathered in one place. Another extra layer of safety measure that we added to the hardened functions is selective use of the `volatile` keyword. Its purpose is essentially to prevent unwanted compiler optimization. Regarding compiler optimization flags, we decided to take a relatively conservative approach: We opted against using very aggressive compiler optimizations, as it significantly decreased traceability and comprehensibility of the assembly code. Compilation results were inspected after usage of both `-0` and `-O1` gcc flags. No hardening or side-channel related violations were visible.

5.3 Technical details

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

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

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

Listing 5.1: Value barrier using `volatile` keyword

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

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

Listing 5.2: Constant-time logical NOT

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

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

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

Listing 5.3: Constant-time ternary operator

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

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

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

Listing 5.4: Oblivious sort

Fault injection This can be seen as bonus hardening task. Fault injection is generally more of a hardware attack, although recently fault injection attacks have also been achieved in software ([43, 13]). It would be very unfortunate if an adversary could simply increase the privacy budget by glitching/flipping one single bit. Even worse, he might be able to switch off differential privacy entirely, or he may change a hyperparameter that the insurance customers did not agree upon. For this reason a solution was implemented that replaces boolean model parameters (internal 1 or 0) with two random integers with hamming distance >25 (see listing 5.5). Then each time a parameter is used, we first check whether it is one of the two values. This way a glitch would need to flip exactly those >25 bits in order to switch off some privacy related model parameter, which should essentially be impossible in practice.

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

Listing 5.5: Parameter fault injection mitigation

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

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

- `samples_left_right_partition`
- ... TODO

All affected floating point operations (A,B,C,D,... TODO) were replaced by their corresponding fixed point counterpart from `libfixedtimefixedpoint` [4].