Trial exercise for the Sharemind team of Cybernetica AS

Madis Ollikainen

November 27, 2016

In the following I'll describe my solution of the trial exercise given to me by the Sharemind team of Cybernetica AS.

1 Task overview

Sharemind is a secure multi-party computation based database/application server, which allows processing of encrypted data without the need for decryption. Applications on the Sharemind server are written in the `SecreC` programming language, which separates private and public data on a type system level. The Sharemind SDK provides the `SecreC` language and compilers as well as a Sharemind server emulator for testing. For my trial exercise I had to familiarise myself with Sharemind SDK and solve a few tasks related to implementing/analysing selection algorithms.

Selection algorithms are algorithms for finding the `k`-th order statistic, *i.e* the `k`-th smallest entry, from a list or array. In my trial exercise, the focus was on 1D arrays. My tasks were:

1. Implement a function `D int64 nthElementSort (D int64[[1]] data, uint64 k)`, which returns the `k`-th smallest element from the input array `data` by first sorting the input array using the sorting function from the `SecreC` standard library.
2. Compare the sorting based `nthElementSort` with the `SecreC` standard library selection algorithm `nthElement`: (a) Which of them is more efficient? (b) Whose execution time leaks more information about the input?
3. Implement an oblivious sorting based selection algorithm `D int64 nthElementSortOblivious (D int64[[1]] data, D uint64 k)`. In this context, oblivious refers to the fact that both of the inputs, `data` and `k`, are private. In the function header this is indicated with specifying the privacy type with `D` (public privacy types do not have to be specified).
4. Implement an oblivious selection algorithm `D int64 nthElementOblivious (D int64[[1]] data, D uint64 k)`, which is based on the `SecreC` standard library function `nthElement`. How does this impact the efficiency of the algorithm and execution time information leakage?

2 The solution repository

My solution can be found from my Github account: `https://github.com/madisollikainen/CyberneticaTrialEx`. Figure 1 illustrates the file structure of the solution repository. This report (both `.tex` and `.pdf` files) can be found from the directory `doc`. The source code of the solutions can be found from the directory `src`. For all three functions there is a file `$func_name$.sc`,

which implements a module called `$func_name$` containing the function itself and a testing function called `test_$func_name$`. The main functions, for each of the tests, are implemented in the files `runTest_$func_name$.sc`. In order to run the tests, the files `runTest_$func_name$.sc` have to be compiled and executed. The necessary files and guides for compiling `SecreC` code and executing it on the Sharemind emulator can be found from `http://sharemind-sdk.github.io/`. In the above the wild-card `$func_name$` has to be substituted with one of the three function names: `nthElementSort`, `nthElementSortOblivious` or `nthElementOblivious`.

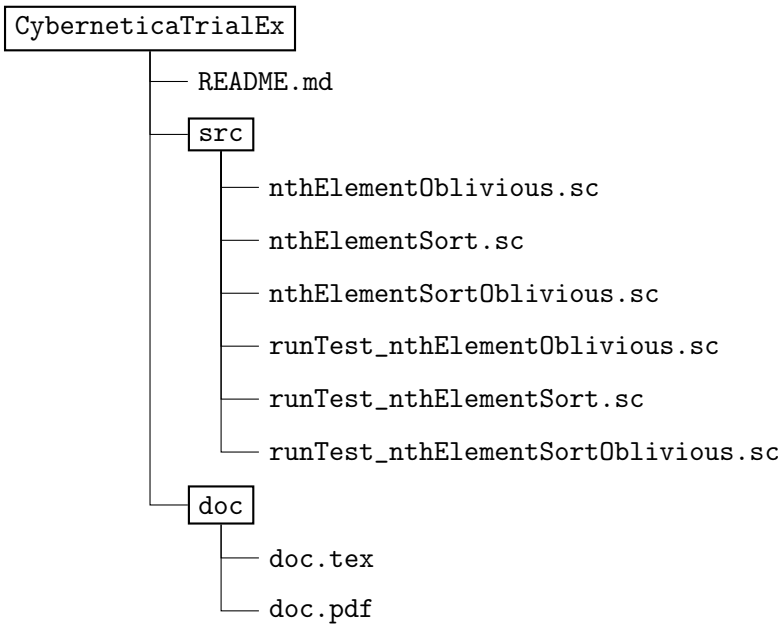

Figure 1: The file structure of the solution repository in Github (`https://github.com/madisollikainen/CyberneticaTrialEx`). Directories have black boxes around them, while files do not.

3 Description of the solution

In this section, I'll shortly present the main ideas of my solution and answer the questions posed in section 1. For the solution code refer to the repository `https://github.com/madisollikainen/CyberneticaTrialEx` and section 2.

3.1 The `nthElementSort` function

The sorting based `k`-th smallest element selection algorithm is rather straight forwards: (1) sort the input using the `SecreC` standard library function `sort` from the `shared3p_sort` module; (2) return the `k`-th element from the sorted array.

3.2 Comparison of `nthElementSort` and `nthElement`

In this section, I'll compare the efficiency and information leakages from execution time of the sorting based selection algorithm `nthElementSort` and the `SecreC` standard library selection algorithm `nthElement`. As seen in section 3.1, the function `nthElementSort` consists of calling the function `sort` and a single array access. As the array access is a constant time operation, the comparison

can be done between the two standard library functions: `sort` and `nthElement`. I'll start by shortly describing the logic behind these two functions and then move onto comparing their efficiency and possible information leakage from their execution time.

3.2.1 The `sort` function

The `sort` function in the `shared3p_sort` module is an implementation of a $O(N^2)$ sorting function. For the following discussion, there are two important thing to note about this sorting function:

1. For any input with the same size N, the execution time of the function is the same, *i.e* the best, average and worst case scenarios are all the same and scale quadratically with N.
2. The sorting function is data oblivious, *i.e* it doesn't leak information about the input data (besides the size of the input array).

3.2.2 The `nthElement` function

The `nthElement` function in the `shared3p_statistics_common` module is an implementation of the `QuickSelect` algorithm, whose main workhorse is the partitioning subroutine. The partitioning function takes an array together with an index of a certain entry in the array, called the "pivot", and re-arranges the entries of the array such that all entries smaller than the pivot are on the left-hand side of the pivot and everything larger than the pivot is on the right-hand side of the pivot. Thus, the pivot is set to the position it would be in an sorted array. This re-ordering of the array is done with a single sweep over the array.

Before starting the selection procedure the `nthElement` function shuffles the input, unless the input variable `bool shuffle` is set to `false`. During the selection, `nthElement` first picks the middle element of the shuffled array as the pivot and then partitions the input array. If the final position of the pivot is at the `k`-th place in the array, the pivot is returned. Otherwise, the search is recursively continued on the correct sub-array: (a) left-hand side sub-array, if the pivot index is larger than `k`; (b) right-hand side sub-array, if the pivot index is smaller than `k`. Unlike the `sort` function, where the execution time is always the same for the same input size N, the execution time of the `nthElement` function can vary. The best and average case scenarios of the `nthElement` scale linearly with the input size, but the worst case scenario scales quadratically. The execution time depends on the structure of the shuffled array.

3.2.3 Efficiency comparison

We have already seen that the execution time of the `sort` function always (best, average and worst cases) scales quadratically with the input size N. In contrast, the execution time of the `nthElement` scales linearly in the best and average case scenarios, but quadratically in the worst case. Thus, for the same input size, on average the `nthElement` function scales better. But, besides simple scaling, there is also the question of the constant in front of the scaling term. When testing the functions, one can witness that the constant is much larger for the `nthElement` function. Therefore, for smaller inputs `sort` is more efficient, while for larger inputs the `nthElement` becomes more efficient.

3.2.4 Information leakages from execution time

As stated before, the execution time of the `sort` function is always the same for the same sized input array. Thus the execution time leaks the size of the input array. Otherwise, the `sort` function is data oblivious. The size of the input array is also leaked in the `nthElement` function: the left- and rightmost indices of the initial input array are public variables. Now, when considering the execution time of the `nthElement` function, then it is clear that it is dependent on the initial (post shuffle) ordering of the array. Therefore, the execution time leaks information on the ordering of the shuffled array. However, this doesn't really give information on the original data array. Nevertheless, if the function is executed without shuffling (`bool shuffle = false`) then some information is leaked about the original input array.

3.3 The `nthElementSortOblivious` function

This section discusses the function `D int64 nthElementSortOblivious (D int64[[1]] data, D uint64 k)`, which sorts the input array and then returns the `k`-th smallest entry, while keeping both the input array and `k` private. The key idea here is to notice that the variable `k` is irrelevant for the sorting: it only comes into play when accessing the `k`-th entry of the sorted array. Thus, we just have to use the `SecreC` standard library function for oblivious vector access `vectorLookup(D bool[[1]] vec, D uint index)` from the module `oblivious`. The oblivious vector access is much more expensive than public access. Nevertheless, as it is used only once the execution time of the algorithm stays effectively the same. Noticeable changes come into play only for very small input sizes.

3.4 The `nthElementOblivious` function

This section discusses the `D int64 nthElementOblivious (D int64[[1]] data, D uint64 k)` function, which uses the same `QuickSelect` algorithm as the `SecreC` standard library function `nthElement`, but keeps the variable `k` private. Similarly as for the `nthElementSortOblivious`, we have to use functions implemented in the `oblivious` module. However, while the `sort` function uses `k` only once, the `nthElement` function uses `k` throughout the function run time for comparison in the control flow. It is also important to note that the boundaries of the sub-arrays, on which the partitioning function is executed, reveal information about the variable `k`. Thus, these boundaries also must be kept private. Consequently, most of the vector accessing and updating in the function has to be made via the oblivious `vectorLookup` and `vectorUpdate` functions from the `oblivious` module. Besides the oblivious vector accessing and updating, the condition for the `for loop` in the partitioning function has to be changed, so that the exact boundaries of the sub-arrays are kept private and only the sizes of the sub-arrays are published.

Such an heavy usage of the oblivious subroutines `vectorLookup` and `vectorUpdate` has a strong impact on the efficiency of the function. Now the constant in front of the scaling term is even larger, making this function very inappropriate for small input arrays. However, as almost nothing is publicly published, the information leakages (even for when the array is unsorted) becomes much smaller.