Privacy Preserving Substring Search Protocol with Polylogarithmic Communication Cost

Artifact Documentation

Nicholas Mainardi
nicholas.mainardi@polimi.it

Alessandro Barenghi
alessandro.barenghi@polimi.it

Gerardo Pelosi
gerardo.pelosi@polimi.it

September 19, 2019

Abstract

This artifact provides a software implementation of the privacy-preserving substring search protocol described in the corresponding paper. Specifically, the artifact includes the source code of the implementation as well as a Makefile which can be used to build the executable. This implementation, which is written in C/C++, besides performing the substring search on genomic data both in a private and non-private manner, is already enriched with some additional routines which allow to measure the performance metrics discussed in the experimental evaluation section of the paper. To this end, the artifact provides some bash scripts (described in this document) which allow to run all the tests described in the paper. To the extent of providing a viable dataset to be employed as the text where the substrings are looked-up, the artifacts additionally includes genomic data obtained from publicly available sources (reference [13] in the corresponding paper): this data is stored in a textual file containing DNA sequences of a human chromosome in the FASTA format. The implementation has been tested on Linux systems equipped with x86 Intel processors and it depends on few external software libraries (listed in the documentation). We remark that to obtain results similar to the ones showed in the paper, the build of the referenced external libraries (at least the GNU MultiPrecision one) should be optimized for the specific platform where the implementation is executed (e.g., employing the gcc compiler option: -march=native).

Contents

1 Introduction

The C/C++ source code found in this artifact allows to reproduce and verify the experimental results found in the corresponding paper, where a novel privacy-preserving substring search protocol is proposed. Furthermore, the implementation provided in this artifact can be effectively used to privately retrieve the position of the occurrences of genomic sequences in genomic data stored in a textual file. The genomic format recognized by the current implementation mandates to employ strings over an alphabet of five characters: the four usual nucleotide found in DNA sequences (A, C, G, T) and the wildcard character N, used in FASTA format to denote any possible nucleotide. For user's convenience, the artifact already includes a textual file which represents a portion of a human chromosome; this file, downloaded from [2], was simply processed to remove metadata and newlines, obtaining a genomic string over the aforementioned alphabet. The current implementation also encompasses an implementation of a state-of-the-art substring search algorithm which retrieves the position of the occurrences without any confidentiality guarantees, comparing the results of private and non-private queries in order to assess the correctness of the proposed protocol. We remark that in the current implementation the client and the server resides in the same process instead of being deployed on two different machines. This choice allows to reduce the time required for the experimental evaluation due to the absence of network overhead while retaining the significance of the results, as the application is conceptually partitioned in a client-side part and a server-side one, which are independently profiled. This artifact provides three different implementations of the proposed privacy-preserving substring search protocol: in the first one, the server side computation of the query procedure is sequentially performed on a single core, while in the second one the same computation is split among multiple threads; the third one simulates the multi-user scenario, where several queries are simultaneously performed in different threads (each performing the query on a single core). In this document, we provide all the necessary information in order to build all these three implementations, we describe their command-line interface and how to replicate the experimental results observed in the experimental evaluation of the corresponding paper.

2 Dependencies & Build

We start by describing how to build the executables and the needed external libraries.

Dependencies

All the implementations contained in this artifact hinges upon three external libraries for the arithmetic and cryptographic operations required by the privacy-preserving substring search protocol:

- GNU MultiPrecision library [3] (`libgmp`) is employed as a fast implementation of multi precision modular arithmetic operations. The protocol has been tested with GMP 6.1.
- HCS library [5] (`libhcs`) provides software implementations of some partially homomorphic encryption schemes, including the length flexible version of the Paillier scheme, proposed by Damgard and Jurik, employed in the protocol.
- The cryptographic library of OpenSSL [4] (`libcrypto`) is employed for AES-128 CounTeR (CTR) mode operations. The protocol has been tested with OpenSSL version 1.0.2r.

Furthermore, the standard C++ API for multi-threading applications are employed, which require to link the `pthread` library on Linux based systems. We remark that in our testing environment all the external libraries are compiled from source code and thus optimized for the specific platform where the tests were run; therefore, in order to faithfully replicate the results, all these libraries should be compiled enabling native architecture-specific optimizations. Given that most of the computation is devoted to performing modular arithmetic operations, compiling from scratch only the GMP and HCS libraries it is expected to be sufficient to reproduce performance results similar to the ones reported in the paper. Figures of merit of the proposed protocol are marginally affected by the use of OpenSSL binaries obtained either downloading them from the official repository or compiling locally the source files.

Build

The artifact already provides a Makefile to automate the compilation of the source code. Before using the Makefile, it may be necessary to configure it for the platform where the implementations are built: the two variables `INCLUDE_DIR` and `LIB_DIR` can be modified to specify additional locations where the compiler can look for, respectively, header files and libraries. The `CC` variable can be used to specify the compiler to be used (otherwise g++ compiler is employed). Once these variables are set, the source code can be compiled by running the `make` command. The Makefile has three main targets, each corresponding to one of the three different implementations discussed in the previous section, whose details, including the source files to be compiled and the libraries which need to be linked, are summarized in Table 1. All these targets are built with the `make` command, however it is possible to build only one target by specifying its name as a parameter of the `make` command (e.g., the command `make parallel`

Table 1: Source files and required libraries for each of the three implementations provided in the artifact

Target Name	Implementation	Source Files	Libraries
`single`	single core	• *single_core.c* • *lipmaa.c* • *lfpaillier.c*	• `libgmp` • `libcrypto` • `libhcs`
`parallel`	multi core	• *multi_core.c* • *multithread_lipmaa.c* • *lfpaillier.c*	• `libgmp` • `libcrypto` • `libhcs` • `libpthread`
`multiuser`	multi user	• *multi_user.c* • *lipmaa.c* • *lfpaillier.c*	• `libgmp` • `libcrypto` • `libhcs` • `libpthread`

builds only the multi core implementation). Each of these targets generates an object file, whose name can be customized by modifying the three variables `EXE_SINGLE`, `EXE_PARALLEL` and `EXE_MUSER`, which are located in the first lines of the Makefile.

3 Docker Container

To the extent of easing the build of the implementation, we provide also a Docker container which encloses the three implementations ready to be used with all the required dependencies. In this section, we describe how to setup this container and retrieve the results of the experiments from it.

The artifact includes a *Dockerfile*, which can be used to build a Docker image of the container with the following command:

```
# docker build -t <image_name> <path_of_artifact_folder>
```

where <image_name> is a name to be assigned to the docker image (say, test_ppss). The build process sets up the image for the container, according to the instructions found in the *Dockerfile*. In particular, the build process first bootstraps a docker image [1] based on the Alpine Linux distribution, installing some basic packages required in the container (including the OpenSSL library required by our implementations), and then downloads and builds the GMP and HCS libraries with platform specific optimizations. Lastly, once all the dependencies are satisfied, the three implementations of the privacy-preserving substring search protocol are built. Since the GMP library is downloaded and compiled from sources, the image build process may take a while (e.g., few minutes).

Once the build is finished, a docker container based on the built image can be instantiated with the following command:

```
# docker run -i -t <image_name>
```

This command should start a bash terminal inside the container. All the files of the artifact should be located in the */home/myuser* folder, which should be the working directory of the bash terminal started in the container. In case of a successful build, there should be three executable in the container, called *single_ppss*, *parallel_ppss* and *mu_ppss*, which correspond to the three implementations of our protocol. These implementations, as well as the bash scripts available in the container, can be employed as described in Section 4 to run the experiments. We remark that the Alpine Linux distribution is generally employed to bootstrap Docker images as it is particularly lightweight; however, this feature has the drawback that only strictly necessary tools are already shipped in this distribution. In case additional software is needed, it can be installed through the Alpine package manager with the following command (to be run inside the instantiated container):

```
# apk add <package_name>
```

After running the experiments, the CSV files storing the output data can be copied outside the container by hinging upon the `docker cp` command. This command requires either the ID or the name of the container, which can be retrieved with the `docker ps` command, which lists all the running containers. Once the ID or the name have been found, the file can be copied outside the container with the following command (to be run on the host machine):

```
# docker cp <container_id>:<path_to_the_file_in_container>
<dest_path_host_machine>
```

For instance, to retrieve the file called *single-core-data.csv* from the container with ID `a3b38d2f271c`, the command to be employed is:

```
# docker cp a3b38d2f271c:/home/myuser/single-core-data.csv ./
```

4 Usage & Experimental Validation

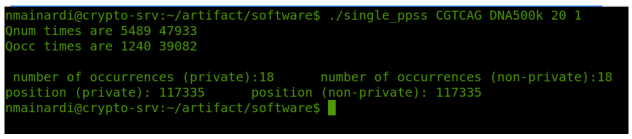
As discussed earlier, three different implementations are included in this artifact, which correspond to three different programs with specific user interfaces. We now describe how to use each of these implementations and how the experiments found in the corresponding paper can be reproduced by hinging upon some simple bash scripts enclosed in the artifact.

How to Use

The single core implementation requires four command line arguments, which must be provided in this order:

1. `substring`: a string over the alphabet $\Sigma = \{A, C, G, N, T\}$, which is the sequence of nucleotide whose occurrences are searched in the text.

2. `input_file`: path of a textual file, which must be formatted as described in the introduction, which stores the string where the occurrences of `substring` are searched. An example file *bigDNA* is already contained in the artifact.



```
nmainardi@crypto-srv:~/artifact/software$ ./single_ppss CGTCAG DNA500k 20 1
Qnum times are 5489 47933
Qocc times are 1240 39082

 number of occurrences (private):18      number of occurrences (non-private):18
position (private): 117335      position (non-private): 117335
nmainardi@crypto-srv:~/artifact/software$ 
```

Figure 1: Execution of single core implementation

3. `b`: an integer ≥ 2 which is employed as the radix to be used for the Lipmaa's PIR protocol.

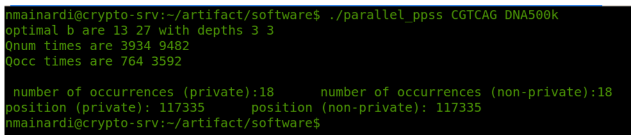
4. `test_type`: an integer (which can be either 0 or 1) used in our experimental setting to determine which experiment is performed. Specifically, as this implementation is used for two different experiments in our experimental validation, this integer allows to distinguish between these two experiments, which is required in order to report on the output file the correct information for the experiment being performed.

Figure 1 shows how to call the single core implementation and its output printed to `stdout`. Given these values, the implementation reads the text in `input_file` and builds the privacy-preserving search index based on the Burrows-Wheeler Transform (BWT) and the Suffix Array as described in the corresponding paper. Then, it retrieves the number of occurrences of `substring` in the text and the positions of one of these occurrences [1] by employing the `Query` procedure of the privacy-preserving substring search protocol; furthermore, the same query is performed in a non-private setting by employing Algorithm 1 of the corresponding paper, which is a known algorithm for substring search which hinges upon unencrypted BWT and suffix array. The results of both these queries are written to `stdout` at the end of the computation, allowing the user to verify the correctness of our protocol. For each of the two phases of the `Query` procedure, referred to as `Qnum` and `Qocc` in the corresponding paper, the implementation measures the client and server costs, which are computed as the execution time of the portion of computation in the `Query` procedure executed by the client (resp. server). These values are both printed to `stdout` and written to a CSV file which contains the result of the experiment; the name and the information found in the file depends on the type of experiment being performed, which is inferred from the `test_type` integer. The implementation computes also the communication cost, which is estimated as the size of the binary representation obtained by serializing the trapdoors which would be sent from the client to the server during each of the two phases of the `Query` procedure[2]. This cost is not printed to `stdout` but it is included in the results of the experiment being written in the CSV file.

The multi core implementation exhibits the same execution flow of the single core one, although the modified search procedure of the Lipmaa's PIR protocol (i.e., Algorithm 3 in the corresponding paper) is executed on multiple cores. As mentioned in the paper, the parallel strategy employs one separate thread for

[1]while the substring search functionality defined in the paper requires to retrieve the position of all the occurrences, we decide to retrieve only one of them in our experimental setting, to the extent of reducing the overall time required to perform all the experiments

[2]We do not consider the size of the response of the server to the client in the communication cost as it is negligible w.r.t. the trapdoor size



```
nmainardi@crypto-srv:~/artifact/software$ ./parallel_ppss CGTCAG DNA500k
optimal b are 13 27 with depths 3 3
Qnum times are 3934 9482
Qocc times are 764 3592

 number of occurrences (private):18      number of occurrences (non-private):18
position (private): 117335      position (non-private): 117335
nmainardi@crypto-srv:~/artifact/software$
```

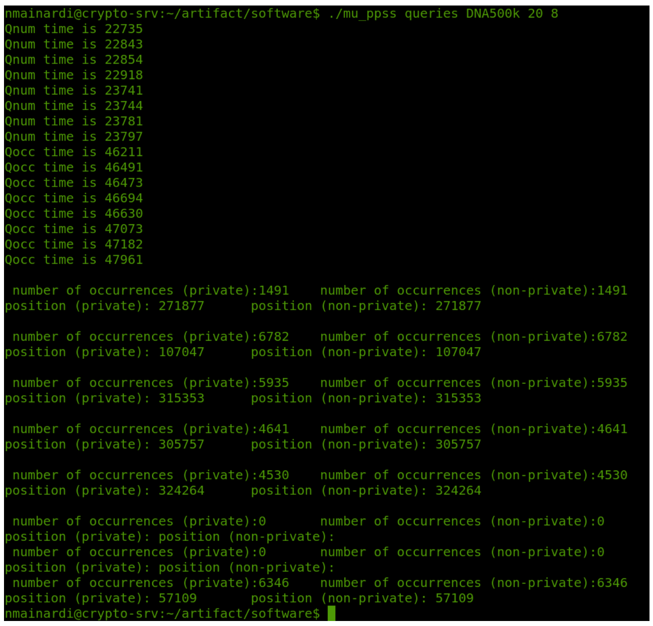
Figure 2: Execution of multi core implementation

each recursive call in Algorithm 3, thus splitting the computation among b cores. Therefore, the value b is chosen by the implementation in order to split as evenly as possible the computational workload among the b threads. In this implementation, two different values, labeled as b_n and b_o, are chosen for the two phases `Qnum` and `Qocc`, respectively, which means that a different number of threads may be used in these phases. However, the user can choose to provide a value b as the third command line argument, as for the single core implementation: in this case, this value is used for both the phases instead of the ones chosen by the implementation. Lastly, the `test_type` integer would be useless in this implementation, as it is employed in only one experiment in our campaign. In conclusion, the multi core implementation has two mandatory arguments (i.e., `substring` and `input_file`) and an optional one (i.e., `b`). Figure 2 shows how to call the multi core implementation and its output printed to `stdout`.

The multi-user implementation simultaneously performs single core queries on the same search index: specifically, each query is executed in a separate thread, simulating multiple independent users performing substring search on the same document. Each query is performed also in the non-private setting, with both the results being printed to `stdout` to let the user verify their consistency. The interface of this implementation has four mandatory arguments and it is designed mainly for our experimental setting:

1. `queries`: a path to a file which contains a set of strings to be searched over the text. Each string is separated by a newline. An example file with 16 queries is already provided in the artifact
2. `input_file`: same as single core implementation
3. `b`: same as single core implementation
4. `num_users`: an integer which specifies the number of users to be simulated, which is equivalent to the number of queries to be performed.

Figure 3 shows how to call the multi-user implementation and its output printed to `stdout`. We remark that this implementation always performs `num_users` queries simultaneously: if there are more strings in the `queries` file, they are ignored, while if there are less strings, they are cyclically re-used. Since the experiments in the multi-user scenario are mainly aimed at showing the limited increasing of the memory consumption when the number of users increases, this implementation employs a separate thread which periodically reads the memory consumption from the Linux's virtual `proc` filesystem and computes the maximum value, which is then reported in a file as the result of the experiment.



```
nmainardi@crypto-srv:~/artifact/software$ ./mu_ppss queries DNA500k 20 8
Qnum time is 22735
Qnum time is 22843
Qnum time is 22854
Qnum time is 22918
Qnum time is 23741
Qnum time is 23744
Qnum time is 23781
Qnum time is 23797
Qocc time is 46211
Qocc time is 46491
Qocc time is 46473
Qocc time is 46694
Qocc time is 46630
Qocc time is 47073
Qocc time is 47182
Qocc time is 47961

 number of occurrences (private):1491    number of occurrences (non-private):1491
position (private): 271877      position (non-private): 271877

 number of occurrences (private):6782    number of occurrences (non-private):6782
position (private): 107047      position (non-private): 107047

 number of occurrences (private):5935    number of occurrences (non-private):5935
position (private): 315353      position (non-private): 315353

 number of occurrences (private):4641    number of occurrences (non-private):4641
position (private): 305757      position (non-private): 305757

 number of occurrences (private):4530    number of occurrences (non-private):4530
position (private): 324264      position (non-private): 324264

 number of occurrences (private):0       number of occurrences (non-private):0
position (private): position (non-private):
 number of occurrences (private):0       number of occurrences (non-private):0
position (private): position (non-private):
 number of occurrences (private):6346    number of occurrences (non-private):6346
position (private): 57109       position (non-private): 57109
nmainardi@crypto-srv:~/artifact/software$
```

Figure 3: Execution of multi-user implementation

Run Experiments

We now describe how to perform the experiments which allow to reproduce the experimental results found in the corresponding paper. We provide here all the details about these experiments and how they can be simply run through the bash scripts which are available in the artifact. We remark that both the overall duration of the experiments reported in this section and the execution times found in the results of the paper depend on the platform being used for these experiments, which is a machine equipped with a dual Intel Xeon E5-2620 CPU clocked at 3 GHz (16 physical cores with hyper-threading overall) and 128 GiB DDR4-2133 memory, with 64-bit Linux Gentoo 17.0 OS.

As most of the experiments require texts of varying size as input data, script *split_file.sh* takes the *bigDNA* genome data, which is available in the artifact, and dumps increasing portions of it in distinct files with different size. Specifically, the script can be simply invoked with no parameters and automatically generates 7 files which contains the first $n \cdot 10^6$ characters of *bigDNA*, with n ranging in $\{0.5, 1, 2, 4, 8, 16, 32\}$; these files have the same size of the ones used in our experimental validation.

There are four different experiments which must be run in order to reproduce the results of the corresponding paper. For each of these experiments, there is a bash script which automatically runs the experiment at hand. All these scripts have a variable called `num_repetitions` which specifies the number of times the experiment has to be repeated: indeed, all the experiments should be performed more than once to make the results more robust against noisy measurement. Nevertheless, as the number of repetitions linearly increases the overall time needed to perform the experiment, we chose to perform only few repetitions. Specifically, the experiments where the execution times are measured are repeated 3 times, while the multi user experiment is performed only once, as it measures only memory consumption, which is expected to be less sensitive to noise. However, we remark that the number of repetitions for each experiment can be changed by modifying the variable `num_repetitions` in the bash script of the experiment at hand.

The first experiment concerns the impact of the radix b employed for the Lipmaa's PIR protocol on the performance of the protocol, whose results are reported in Figure 3 of the paper. Specifically, this experiment invokes the single core implementation with values for b in $\{2,\dots,40\}$ on the smallest genomic data (i.e., the one with $0.5\cdot 10^6$ nucleotide). The experiment can be run through the *test_b.sh* script and it lasts slightly more than 4 hours on our platform. The results of the experiment are appended to a file called *radix_tune.csv*. Specifically, for each value b being tested, the file contains the client, server and communication cost for both the `Qnum` and the `Qocc` phases.

The second experiment aims at assessing the performance of the single core implementation, whose results are represented by the continuous lines in Figure 4. This experiment measures the performance of the single core implementation over the *bigDNA* file and all the other files generated with the *split_file.sh* script. A fixed value $b = 20$ is employed in this experiment. The experiment can be run through the *single-core_test.sh* script and it is the longest one, lasting approximately 15 hours on our platform. The results of the experiment are appended to a file called *single-core-data.csv*. Specifically, for each size of the genomic data being tested, the file contains the client, server and communication cost for both the `Qnum` and the `Qocc` phases.

The third experiment is equivalent to the previous one, but the multi core implementation is employed instead of the single core one. The results are represented by the dashed lines in Figure 4 of the paper. As discussed in the paper, for the multi core implementation we do not employ a fixed value for the radix b, but we let the implementation choose a value which allows to split as evenly as possible the computational workload among the b threads. Therefore, the results of the experiment, which are appended to a file called *multi-core-data.csv*, besides all the values reported also in the previous experiment, additionally include the two values b_n and b_o employed in, respectively, the `Qnum` and `Qocc` phases. The experiment can be run through the *multi-core_test.sh* script and it lasts only a couple of hours on our platform.

The fourth experiment concerns the multi user case, aiming at verifying that the memory consumption does not significantly increase with the number of simultaneous queries being performed. In particular, the experiment analyzes how the memory consumption varies with the number of users when genomic data with different sizes are employed for the substring search, leading to the result reported at the end of the experimental evaluation section of the paper, namely that the additional memory required for each user does not linearly increase with the size of the dataset. To this extent, this experiment employs the multi user implementation and, for each dataset being tested, it simultaneously performs $1, 2, 4, 8$ and 16 queries, hinging upon the information available in the Linux `proc` virtual filesystem to estimate the maximum memory consumption. To reduce the amount of time required for this experiment, we employ only half of the genomic data available, namely the ones containing, respectively, $0.5\cdot 10^6, 2\cdot 10^6, 8\cdot 10^6, 32\cdot 10^6$ nucleotide. A fixed value $b = 20$ is used in these experiment. The experiment can be run through the *muse_test.sh* script and it lasts about 10 hours on our platform. The results are appended to a file called *mem_consumption.csv* which reports, for each dataset and for each number of queries being tested, the base memory consumption (i.e., the memory when no queries are performed) and the maximum memory consumption throughout all queries. The memory consumption data, as given by the `proc` virtual filesystem, is reported as the number of pages allocated in the data and stack frames of the process.

5 Result Analysis

To conclude this document, we show how to process the output of the experiment in order to obtain the results reported in the paper and we provide some insights on the post-processed results.

The artifact includes a python script, called *post_process.py*, which allows to automatically compute the average value over all the repetitions of the experiment for each measurement. The script takes as input the filename of the output file to be processed, compute the average values and store them in a new CSV file with the same name of the original file prefixed with the string *avg-*. For instance, if the file *multi-core-data.csv* is given to the script, the output file will be called *avg-multi-core-data.csv*. The files obtained after this processing should replicate the results reported in the experimental evaluation of the corresponding paper. Each of these CSV files has an header which describes the meaning of each field and, if needed, the unit of measurement. The artifact includes also a latex file *plots.tex*, located in the *tex* folder, which allows to generate the plots found in Figures 3 and 4 of the corresponding paper, which are reported here for reader's convenience in Figure 4 and Figure 5, respectively. By compiling this file with `pdflatex`, a PDF file which contains these plots will be generated; to generate only one the plots for Figure 3 (resp. 4), it is possible to comment on the *plots.tex* source code the line \input{perf_plot.tex} (resp. \input{radixb_plot.tex}).

We remark that, because of the different platforms where the experiments are performed, the actual values in terms of execution times and memory consumption may differ from the ones reported in the paper; nonetheless, the behavior of the implementations showed by the results should fit the one reported on the paper. In other words, the shape and the trend of the plots obtained from these results should resemble the ones reported in the paper. For the multi user experiment, which has no plot in the paper due to space constraints, the results should provide two evidences:

1. the increase of memory consumption per user becomes more negligible w.r.t. the base memory consumption when the dataset size increases

2. the increase of memory consumption per user does not increase significantly (i.e., linearly) with the dataset size

For instance, on our platform, for all the dataset being tested, the increase of memory consumption per user is always approximately 70 MB independently from the size of the dataset, thus proving that the additional memory consumption due to new allocated data in the computation is negligible w.r.t. other system overhead (e.g., data structures for handling more threads).

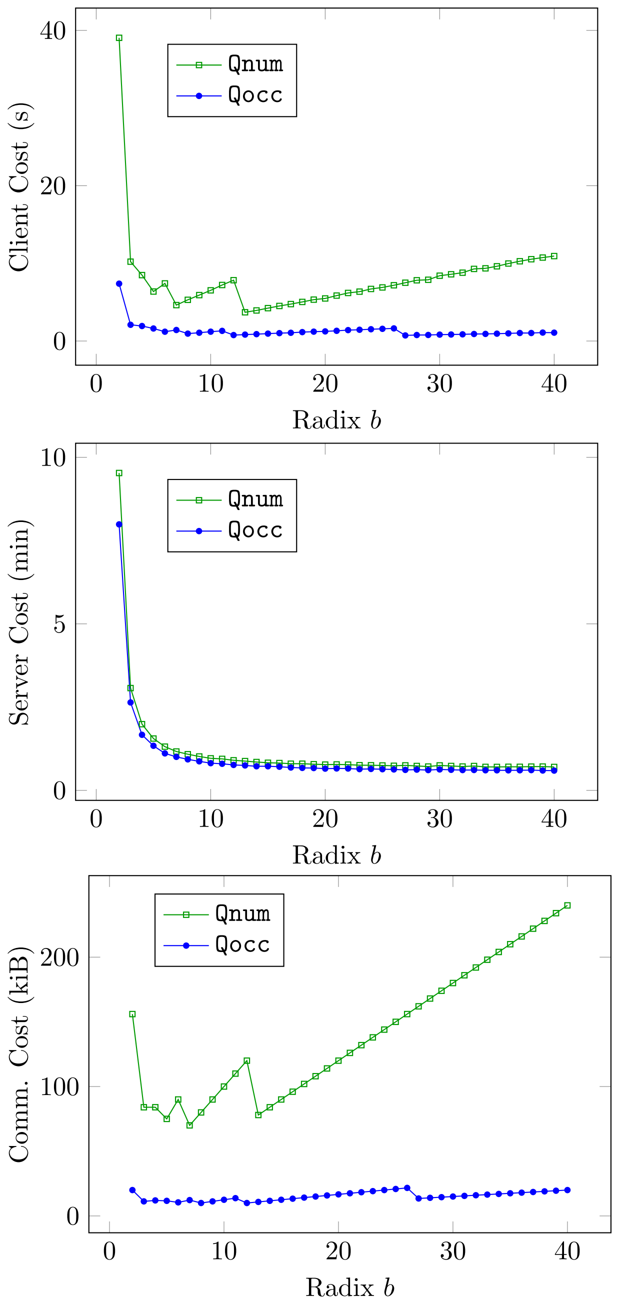


Figure 4: Performance of our PPSS protocol as a function of the radix b employed in the PIR algorithms. Private search of $q = CTGCAG$ in a genome with $500k$ nucleotides

References

[1] Natanael Copa. Alpine linux – a minimal docker image based on alpine linux with a complete package index and only 5 mb in size! `https://hub.`

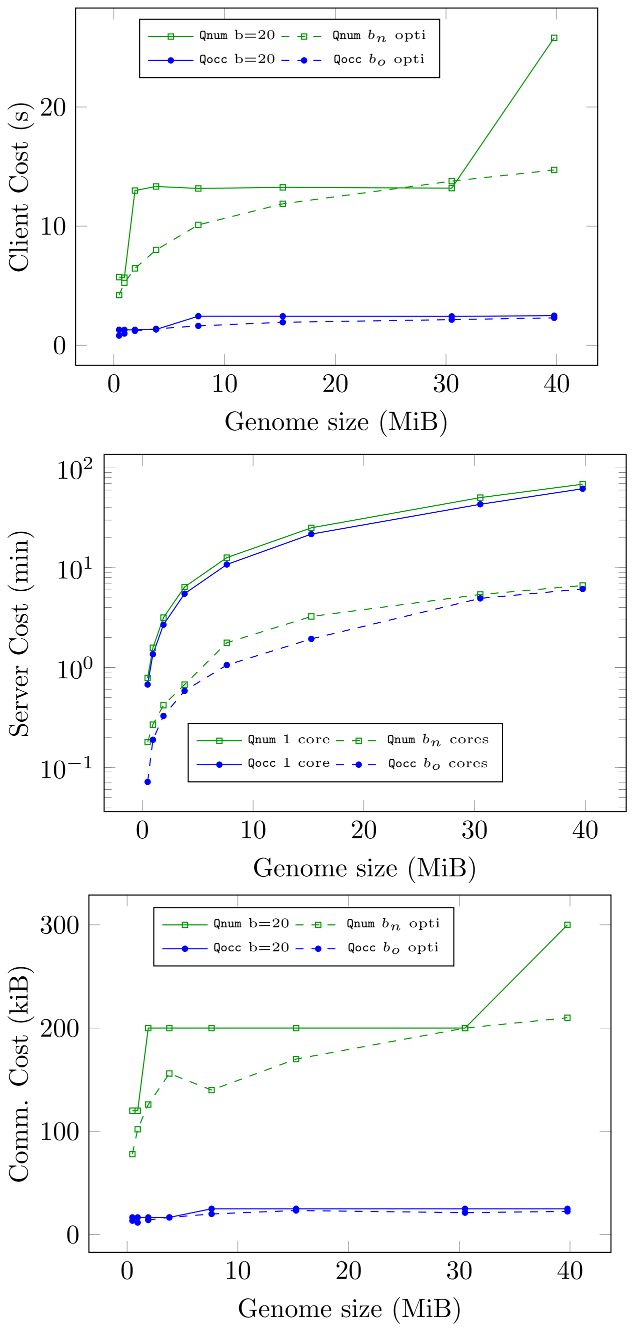


Figure 5: Performance of our PPSS protocol as a function of the genomic document size to find one occurrence of the substring $q = CTGCAG$. Considering each document size in increasing order, the optimal values of radixes b_n and b_o employed during the experiments are $\{13, 17, 21, 26, 14, 17, 20, 21\}$ and $\{27, 14, 17, 20, 24, 28, 17, 18\}$, respectively

docker.com/_/alpine, 2019.

[2] Paul Flicek et. al. Ensembl Genome Browser. http://www.ensembl.org/, 2000.

[3] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. http://gmplib.org/.

[4] Ben Kaduk et. al. OpenSSL – Cryptography and SSL/TLS Toolkit. https://www.openssl.org, 2015.

[5] Marc Tiehuis. libhcs: A partially Homomorphic C library. https://github.com/tiehuis/libhcs/tree/master/include/libhcs, 2015.