Privacy Preserving Large Scale DNA Read-mapping in MapReduce Framework using FPGAs

Lei Xu  
University of Houston  
United States  
lxu13@uh.edu

Hanyee Kim  
Korea University  
Republic of Korea  
hanyeemy@korea.ac.kr

Xi Wang  
University of Houston  
United States  
xiwang@cs.uh.edu

Weidong Shi  
University of Houston  
United States  
larryshi@cs.uh.edu

Taeweon Suh  
Korea University  
Republic of Korea  
suhtw@korea.ac.kr

Abstract—Read-mapping, i.e., finding certain patterns in a long DNA sequence, is an important operation for molecular biology. It is widely used in a variety of biological analyses including SNP discovery, genotyping and personal genomics. As next-generation DNA sequencing machines are generating an enormous amount of sequence data, it is a good choice to implement the read-mapping algorithm in the MapReduce framework and outsource the computation to the cloud. Data privacy becomes a big concern in this situation as DNA sequences are very sensitive. In response, encryption may be used to protect the data. However, it is very difficult for the cloud to process cipher texts. In the MapReduce framework, even if values (data to be processed) may be protected by encryption, keys cannot be encrypted using semantic secure encryption schemes as it will affect the MapReduce scheduling mechanism. But if no protection is utilized, attackers may extract useful information from unprotected keys. We propose a solution that can securely outsource read-mapping computations in the MapReduce framework by leveraging inherent tamper resistant properties of FPGAs. We also provide a method to protect the keys generated in this process. We implement our solution using FPGAs and apply it to some data sets. The security evaluation and experimental results show that with this method, DNA sequence privacy is well protected, and the extra cost is acceptable.

I. INTRODUCTION

As the bioinformation technology develops, more and more genetic sequence data are generated. For example, the length of human genome map is about 3000Mb. Here Mb is the unit of length for DNA Fragments equal to 1 million nucleotides. Considering the huge number of difference specifies, it is a big challenge processing these data efficiently. For bioinformation problems that can be processed in parallel, e.g., read-mapping problem, the emergence of MapReduce and cloud computing offer a promising solution to the challenge: MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster and cloud computing provides the cluster with elastic hardware resources such as computation, storage, and communication.

Security is one of the critical concerns that hinders running read-mapping algorithm in the MapReduce framework using cloud computing. The data can be encrypted using AES or 3DES before uploading to cloud to process. But it is very hard for the cloud to process the cipher-texts. Fully homomorphic encryption (FHE [1]) is believed to be a promising solution for such scenario because it supports arbitrary operations on cipher-texts. However, the high overhead is unacceptable in practice.

Genetic sequences are very sensitive data, especially when these sequences are human DNAs. To address the security concern while utilizing the capability of MapReduce and cloud computing, we develop a FPGA based solution for privacy preserving read-mapping in MapReduce framework using cloud computing.

Field programmable gate arrays (FPGAs) receive much attention in recent years for data analytical applications because of its capability to support customized parallel processing. Modern FPGAs also provide protection mechanism for bit-streams, which may contain valuable intellectual properties. By taking advantage of tamper resistant feature of FPGAs and modified usage of bitstream encryption, our solution guarantees the privacy of the genetic sequences to be processed while allowing the user to enjoy the flexibility and convenience of the MapReduce framework and cloud computing. In our solution, data encryption keys that are used to protect the genetic sequence are embedded in bitstream, which is further protected by the bitstream encryption mechanism. Due to the tamper resistant property of FPGAs, the data can be safely decrypted and processed inside FPGAs. Furthermore, we design a special disturbing technique for keys in MapReduce framework that provides further protection for the data. The contribution of this paper includes:

- We propose a solution for privacy preserving read-mapping using MapReduce framework and cloud computing based on FPGAs;
- We analyze the potential information leakage of keys in MapReduce framework and provide a countermeasure;
- We implement our solution using FPGAs and the experimental results show our solution is practical.

The remainder of this paper is organized as follows. In Section II we describe the detailed design of privacy preserving map-reading in MapReduce framework using cloud computing. Security analysis is also given in this section. Section III discusses the implementation issues and evaluates the performance. Section V concludes the paper.

II. PRIVACY PRESERVING READ-MAPPING WITH FPGAS

In this section we discuss FPGA based privacy preserving BLAST in MapReduce framework using cloud computing.

A. Overview and Threat Model

Three parties are involved when a documents clustering task is outsourced to FPGA cloud: (1) FPGAs. FPGAs are the devices used to run map and reduce functions for read-mapping task. FPGAs will not leak the data and are trusted because they are tamper resistant. We also suppose a key for bitstream
encryption is safely saved inside the FPGAs; (2) Cloud service provider (CSP). CSP manages a resource pool which contains a bunch of FPGAs, and provides computation service to users. Besides managing the FPGAs, CSP is also responsible for some scheduling work of MapReduce framework such as *collect*, *spill*, *merge* and *shuffle*. CSP is semi-trusted, i.e., it will follow pre-defined protocols to use the FPGAs to finish the read-mapping task while it may try its best to learn about the genetic sequences being processed; (3) User. A user has some genetic sequences and wants to utilize the resources of CSP to finish the work of read-mapping using BLAST. A user always encrypts his data before transferring to CSP. As data owner, the user is always totally trusted.

The security goal of our solution is to protect the genetic sequences while allowing the CSP to process them. The basic idea is simple. User encrypts and uploads documents to CSP. CSP runs BLAST algorithm in MapReduce framework to process the task. When plain texts are needed, these operations are executed inside FPGAs. At the end of the process, CSP sends back encrypted results to the user. The user can decrypt to get the final results.

### B. Detailed Working Process

#### System Initialization.

CSP holds two types of FPGAs, one for *map* function and the other one is for *reduce* function. In order to make it easier for CSP to schedule, each type of FPGAs has the same public/private key pair, denoted as $(pk_{map}, sk_{map})$ and $(pk_{reduce}, sk_{reduce})$. $pk_{map}$ and $pk_{reduce}$ are published, while $sk_{map}$ and $sk_{reduce}$ are embedded in bitstreams and protected by bitstream encryption mechanism. If higher level of security is required that different FPGA has different private key, more sophisticated encryption scheme such as CP-ABE [2] or KP-ABE [3] can be used to make sure that each FPGA possesses different private key. Parameters of BLAST are also decided in this step, e.g., the number of allowed errors when matching (denoted as $k$).

#### Data uploading.

The user encrypts both the *read* sequences and *reference* sequences using AES data key $dek$. The ciphertexts are denoted as $c_{read}$ and $c_{ref}$. $dek$ is further encrypted using $pk_{map}$, and the ciphertext is denoted as $c_{dek}$. The user also randomly picks a key $hk$ for HMAC. This key is also encrypted using $pk_{map}$, and the ciphertext is denoted as $c_{hk}$. The user uploads all these cipher-texts to CSP.

#### CSP side procedure.

After receiving the cipher-texts from the user, CSP works on the received cipher-texts, which is summarized in Fig. 1. In the following description, mapper means a FPGA running the *map* function and reducer means a FPGA running the *reduce* function; (1) Read. CSP reads the encrypted sequence data from file system and divides it into blocks. CSP dispatches blocks to different mappers, $c_{dek}$ is sent together with $c_{read}$ and/or $c_{ref}$; (2) Map. Mappers are responsible for evaluating Map function. As $c_{dek}$ and $c_{hk}$ is encrypted with $pk_{map}$, mappers can decrypt it with $sk_{map}$ which is embedded in bitstream. Then mappers use $dek$ to decrypt received data block to get plain-texts of reads or/and references. For a *read* sequence, *map* function divides it into $k + 1$ subsequences with same length $\ell$. For a *reference* sequence, it is divided into subsequences of length $\ell$ by enumerating all possible consecutive subsequences. For example, if $\ell = 5$ then a *reference* sequence ACAAGATGCCC of length $len = 10$ will be divided into $6 = len - \ell + 1$ subsequences: ACAAG, CAAGA, AAAGAT, AGATG, GATGC, ATGCC. These subsequences of *reads* and *references* are called seeds. The output of *map* function are a set of pairs $(key, value)$, where $key = seed$ and $value = info_{seed}$. *values* are used to decide where to send these pairs. Before sending the pairs to reducers, the original $(key, value)$ is encrypted using AES and $dek$, the result is denoted as $c_{pair}$. $hMAC_{hk}(\cdot)$ is applied to $key$ to generate $hkey$, $(hkey, c_{pair})$ is sent to CSP. In order to enable reducers to process these pairs, $dek$ and $hk$ are encrypted using $pk_{reduce}$. Fig. 2 describes this process from the perspective of hardware implementation; (3) Collect, Split, and Merge. These jobs are done by scheduler, which is a component of CSP. Pairs with the same $hkey$ value are dispatched to the same reducer; (4) Reduce. Receiving data from the CSP, the reducer first decrypts cipher-texts of $dek$ and $hk$ using $sk_{reduce}$. Then the reducer decrypts $c_{pair}$’s and gets a bunch of *values*.
that some of the values come from read sequence and some come from reference sequences. Reduce function try to match these two sets of values. Let \( v_1 \) and \( v_2 \) be two values from the read and reference sequence respectively. \( v_1 \) is expanded to the original read sequence and we try to match it with the reference sequence at the position provided by \( v_2 \). The similarity value is calculated and returned as the result. Reduce function encrypts all the results using \( 
abla \text{dek} \) and outputs the results; (5) Write. After receiving results from reducers, CSP writes the result to storage system for the user. As the results are encrypted using \( 
abla \text{dek} \), only the legitimate user can get the read-mapping result.

Result gathering. After the processing procedure is finished, the user can read the result from CSP, which is encrypted using \( 
abla \text{dek} \). As \( 
abla \text{dek} \) is chosen by the user himself, he can decrypt and get plain-text of the result.

C. Security Analysis

According to our design, both the read sequences and reference sequences are encrypted before sending to CSP. At the map step, the plain-texts only appear inside the FPGAs. Because of the tamper resistant property, an attacker cannot look inside. The outputs of the mappers are also protected. HMAC is applied to keys to hide the original values. We do not apply hash function to keys because if the domain of key is not large enough, the attacker may guess the original keys from hash values. All the pairs are protected using AES, and the AES key is protected by reducers’ public key. An attacker can access these information but he cannot recover the plain-texts. For the reduce step, the situation is similar. Decryption operation only occurs inside the FPGAs, and the attacker has no way to learn the plain-texts.

D. Statistical Leakage and Countermeasures

When a genetic sequence, especially a long reference sequence is processed by map function, it is very likely that some seeds appear more times than the others. Note that the distribution of the seeds is directly reflected by the distribution of \( h_{\text{key}} \). By applying HMAC, the original keys’ values can be hidden but the distribution will not be affected because the HMAC function always outputs the same result for the same input (when the same HMAC key is applied). An attacker with background knowledge may be able to establish the relationship between probability values and seed values so it can interpret the content of the sequence being processed. A straightforward idea to prevent such information leakage is to involve some randomness in the process of \( h_{\text{key}} \) generation. However, this method is infeasible as it will disrupt the scheduling of MapReduce framework. Instead, we propose to add a cyclic counter in the process of key generation for reference sequences. Specifically, each mapper maintains a counter, and this counter is used as another input to the HMAC function.

As an example, we use a data set from UCI (https://archive.ics.uci.edu/ml/datasets/Molecular+Biology+Splice-junction+Gene+Sequences) as a reference sequence and show the distribution of the seeds with different values. Fig. 3(a) is the original distribution and Fig. 3(b) shows the distribution when proposed countermeasure is applied. It can be seen that the original distribution is greatly disturbed.

Because the counter is cyclic, original HMAC value for a given key is divided into several sub HMAC values. According to the scheduling rule of MapReduce, pairs that go the same reducer now go to different reducers. This may lead to missing of matches, as the seed of a read sequence only appears for one reducer. To avoid such omission, we also generate multiple HMAC values for each seed coming from read sequence. Read sequence is usually short and the number of different seeds is small, so this operation will not add much cost.

III. IMPLEMENTATION AND EVALUATION

We implemented our solution using a Xilinx Zynq-7000 (xc7z020clg484-1). The map and reduce functions were originally written in C. As there are two types of inputs (read sequences and reference sequences), the map function is further divided into two types of functions, one is map_read_kernel() for processing of read strings, the other is map_reference_kernel() for processing of reference strings. To convert the C code to a hardware description, we used the Xilinx Vivado High Level Synthesis (HLS) [4]. For performance, we turned on the ARRAY_PARTITION and PIPELINE directives in Vivado HLS. The ARRAY_PARTITION directive allows a string declared as char in C to be expanded, so the data can be accessed in one clock cycle. The PIPELINE directive enables a higher throughput by adding pipeline stages in hardware. Fig 4 shows the implemented hardware block diagrams of the three key functions: map_read_kernel(), map_reference_kernel(), and reduce_kernel(), which are originally written in C. HMAC() is also written in C and converted to a Verilog description via Vivado HLS. The AES encryption and decryption IP cores written in Verilog were downloaded from OpenCores.org. In Fig. 4(a), the security enhanced map_read module in outer dotted line is composed of two AES decryption modules, four AES encryption modules, two HMAC modules and map_read_kernel marked in inner dotted line. The security enhanced map_read generates two sets of (hkey, cpair) simultaneously, where hkey is the result of applying HMAC to the original key and cpair is the ciphertext of the original pair. The module consumes 4521 FFs, 6306 LUTs, 88 Mem LUTs, and 4 BRAMs. The latency to process map_read is 522 cycles. The security enhanced map_reference...
in the outer dotted line of Fig. 4(b) is composed of two AES encryption modules, one AES decryption module, one HMAC module and map_reference_kernel. The map_reference generates one set of (hkey, cpair). It utilizes 3751 FFs, 5632 LUTs, 522 Mem LUTs, and 2 BRAMs. The latency to process is 560 cycles. The security enhanced reduce in the outer dotted line of Fig. 4(c) is composed of one AES encryption, two AES decryption modules and reduce_kernel. It utilizes 396 FFs and 1093 LUTs. The latency is 26 cycles.

IV. RELATED WORK

A lot of works have been done on utilizing cryptographic tools to provide secure outsourced computation such as searching and DNF evaluation on cipher-texts. However, it is often infeasible to apply them in the real world because they either support limited application scenarios or suffer from high computation/storage cost. From the perspective of secure FPGA, some researchers have proposed to use physical unclonable functions (PUFs) to protect the FPGA bitstream [5], [6]. Other techniques such as watermarking and signature are also used to protect the FPGA IPs [7], [8]. In [9] the authors proposed a FPGA based oblivious RAM processor, the aim of which is to protect the memory access pattern and is different from our purpose. There are recent efforts on applying FPGAs as accelerators for parallel data analytics [10], [11]. In [12], the authors present a framework to use FPGAs to accelerate MapReduce processing, but all these related efforts do not consider the problem of data privacy. In [13] the authors discussed FPGA based secure cloud computing but did not consider the application of DNA read-mapping.

V. CONCLUSION

In this paper we propose a FPGA based solution for running BLAST algorithm securely in the MapReduce framework using cloud computing. By leveraging the tamper resistant property of FPGAs and bitstream encryption mechanism, we protect the data from CSP. We also consider the risks of information leakage and statistical attack caused by distribution of keys, and propose corresponding countermeasures. The security analysis indicates our solution achieves the security goal of protecting the genetic sequences. The implementation and evaluation results demonstrate our solution is practical.

REFERENCES