JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 25, Number 0, 2018
© Mary Ann Liebert, Inc.
Pp. 1–12
DOI: 10.1089/cmb.2017.0094

Research Article

A Secure Alignment Algorithm for Mapping Short Reads to Human Genome

YONGAN ZHAO, XIAOFENG WANG, and HAIXU TANG

ABSTRACT

The elastic and inexpensive computing resources such as clouds have been recognized as a useful solution to analyzing massive human genomic data (e.g., acquired by using next-generation sequencers) in biomedical researches. However, outsourcing human genome computation to public or commercial clouds was hindered due to privacy concerns: even a small number of human genome sequences contain sufficient information for identifying the donor of the genomic data. This issue cannot be directly addressed by existing security and cryptographic techniques (such as homomorphic encryption), because they are too heavyweight to carry out practical genome computation tasks on massive data. In this article, we present a secure algorithm to accomplish the read mapping, one of the most basic tasks in human genomic data analysis based on a *hybrid cloud computing* model. Comparing with the existing approaches, our algorithm delegates most computation to the public cloud, while only performing encryption and decryption on the private cloud, and thus makes the maximum use of the computing resource of the public cloud. Furthermore, our algorithm reports similar results as the nonsecure read mapping algorithms, including the alignment between reads and the reference genome, which can be directly used in the downstream analysis such as the inference of genomic variations. We implemented the algorithm in C++ and Python on a hybrid cloud system, in which the public cloud uses an Apache Spark system.

Keywords: genome privacy, privacy-preserving techniques, read mapping.

1. INTRODUCTION

The continuous progress in the DNA sequencing technologies, including the next-generation sequencing (NGS) and the third-generation single-molecular sequencing techniques, is rapidly reducing the cost of human genome sequencing, which hit the psychological line of $1000/person last year (van Nimwegen et al., 2016). Already, this technical advance has produced massive human genomic data, which will soon be dwarfed by the new volume to be generated due to many recently launched large-scale genome sequencing projects. For example, the Precision Medicine Initiative is a national study involving the collection of the genome, metabolome, and microbiome, as well as health records from a cohort of one million

Department of Computer Science, School of Informatics, Computing, and Engineering, Indiana University, Bloomington, Bloomington, Indiana.

volunteers (Collins and Varmus, 2015), including the patients of cancer and other complex diseases. With the availability of such big data, it is increasingly difficult for individual data analysis centers to supply the computation power that meets the demands of data analysis. As a result, the trend of moving the sequence analysis to the elastic and inexpensive computing resources, such as clouds, has already been widely recognized by the bioinformatics community (Stein et al., 2010; Datta et al., 2016). However, outsourcing the computation on sensitive human genomic data to such less reliable systems faces significant privacy challenges, as cloud providers, like Amazon EC2, are often reluctant to assume liability.

The confidentiality of clinic DNA sequences has long been considered to be of critical importance (Annas et al., 1995). It is well known that human genomes may reveal information about individuals. For example, methods are being developed to infer biometric and appearance traits (e.g., the eye and hair color) of unknown donors from their biological materials, a technique called *forensic DNA phenotyping* (Kayser, 2015). Furthermore, the understanding of the relationship between genotypes and phenotypes (including the predisposition of diseases such as diabetes or cancer) grows quickly: it was shown that the identity of a participant in a human genome study can be revealed from his genetic profile through searching online genealogy databases (Gymrek et al., 2013); furthermore, statistical techniques have recently been developed to directly link the genetic profile and identifiable phenotypes of human individuals (Harmanci and Gerstein, 2016).

Privacy challenges also arise from the analyses of a large amount of NGS data. It is well known that any human individual can be uniquely identified at high confidence with as few as 10–20 single-nucleotide polymorphisms in his/her genome (Lin et al., 2006). A small fraction of NGS reads in a genome, exome, or transcriptome sequencing dataset contains sufficient variation information to identify the donor of the data, and thus such datasets cannot be stored or analyzed on a public cloud without appropriate protection. We note that this problem cannot be solved by using simple data de-identification approaches, for example, data aggregation (i.e., combining multiple datasets in a single analysis task), because it has been shown that even aggregated DNA data (such as allele frequencies or the presence/absence of minor alleles in a human genome database) can leak out identifiable information (Homer et al., 2008; Braun et al., 2009; Sankararaman et al., 2009; Shringarpure and Bustamante, 2015). As a result, data owners of today become more cautious about utilizing public clouds to analyze their sensitive human genomic data.

Security and cryptographic techniques have been developed for the purposes of protecting data privacy on the cloud. A preeminent example is *homomorphic encryption* (HE; Fontaine and Galand, 2007) that allows users to directly analyze encrypted data without decrypting it on an untrusted server (e.g., a public cloud). The users can also delegate most computation to a public cloud by using a secure multiparty computation (SMC) protocol (Yao, 1986), combined with the oblivious random access memory technique (Goldreich and Ostrovsky, 1996) to hide the data access pattern on the cloud. However, these schemes are designed for general purposes of secure computation, and thus are far too heavyweight to carry out practical genome computation tasks. For example, it takes about 4 seconds to compute the edit distance of 2 strings of 100 characters long (which is computationally equivalent to align two DNA fragments) by using the state-of-the-art SMC implementation (Huang et al., 2011), while the HE implementation is magnitudes slower (Atallah et al., 2003). Therefore, practical solutions that exploit special properties of genome computation tasks are needed to outsource such tasks to clouds (Jiang et al., 2014; Zhao et al., 2015).

Most of NGS data analyses (in particular in biomedical researches) start from the mapping of NGS sequences (*reads*) onto a reference genome (e.g., the human genome), a procedure known as the *read mapping* (Trapnell and Salzberg, 2009; Li and Homer, 2010). Considering the importance of such application, we previously proposed a secure computation algorithm specifically for this task, based on a *hybrid cloud computing* model, which allows users to ''spill-over'' a majority of computation for read mapping to a public cloud, while only conducting a small amount of computation on the users' own server (the *private* cloud; Chen et al., 2012). Our algorithm follows the commonly used *seed-and-extension* approach to approximate sequence alignment, and partitions the read mapping task into two separate subtasks, delegating each of them to public and private clouds, respectively: the public cloud searches for the exact matches between the keyed hash values of seeds and relevant substrings on the reference genome to determine the putative locations of reads, while the private cloud extends these seed matches to optimal alignments and infers the variations between the reads and the reference genome. To shift the workload from the private cloud to the public cloud, we employed the *seed combination* approach that searches for the matches of keyed hash values for two combinations of short (12-bp) seeds (instead of the long, continues seeds), which reduces the computation on extending false seeds.

This approach can be further accelerated by using the MinHash technique for seed matching, as shown in a recent article (Popic and Batzoglou, 2016). Although these approaches established a hybrid cloud framework for secure read mapping, they require a substantial fraction of computation (i.e., the *extension* subtask) to be conducted on a private cloud. An ideal secure read mapping algorithm, however, should delegate both the seeding and extension subtasks to the public cloud, while only performs the lightweight encryption/decryption operations on the private cloud. In this article, we present a secure read mapping algorithm to achieve this goal.

To perform the secure extension of each seed on a public cloud, we encrypt the l-tuples in the read and the corresponding substring of the reference genome in a *site-wise* manner, using a symmetric encryption function based on the key dependent on the genomic location of the seed, and then conduct a dynamic programming algorithm on the two strings of keyed hash values, representing the encrypted l-tuples in the read and those in the genome substring, respectively. In this case, because different keys are used for reads from different genomic locations, the genome-wide frequency analysis of l-tuples could not infer sufficient information for identifying the donor of the reads, according to our security analysis. We implemented the algorithm in C++ and Python on a hybrid cloud system, in which the public cloud uses an Apache Spark system (Zaharia et al., 2010).

2. APPROACH

In a typical read mapping algorithm, the extension phase starts from each predetermined seed and uses a constrained dynamic programming algorithm to identify the difference between the nucleotide sequence of the read and the corresponding substring of the reference genome. To delegate the extension algorithm to a public cloud, we first represent the read and the reference substring as two sequences of l-tuples, and then modify the conventional dynamic programming algorithm to align these two l-tuple sequences. This algorithm is efficient, accurate, and secure. On the one hand, using a symmetric encryption function, we can encrypt each l-tuple in the read and the reference substring, ensuring two encrypted l-tuples (ciphertexts) are identical if and only if these two l-tuples (plain texts) are the same; on the other hand, when l is sufficiently large, it takes about the same time to align two l-tuple sequences as to align two nucleotide sequences, while the alignment between l-tuple sequences provides a good approximation of the nucleotide sequence alignment (for details see Section 3).

However, if we use the same key to encrypt the l-tuples in all reads, considerable information can be inferred by using the frequency analysis of encrypted l-tuples (*ciphertexts*), a threat that has been analyzed in our previous article (Chen et al., 2012). Briefly, because a symmetric encryption function is used, the frequencies of encrypted l-tuples remain the same as the frequencies of the plain texts in the reference genome (which is publicly known). Hence, an adversary can infer the putative plain texts of an encrypted l-tuple by comparing their frequencies. Although this method cannot determine the exact plain text of an encrypted l-tuple, it can establish the relationship between the *bins* of ciphertexts and plain texts, each containing the encrypted and unencrypted l-tuples with the same frequencies (e.g., all unique l-tuples are in the same bin), respectively, so that the ciphertexts in a bin are inferred to be encrypted from the l-tuples in a corresponding bin. Using this technique, an adversary can further infer some single-nucleotide variations (SNVs) in a NGS dataset (acquired from a patient), in particular, for those located in the repetitive regions in the human genome (Chen et al., 2012).

To address the threat of the frequency analysis, we propose the *side-wise* encryption approach that does not use the same key for all reads. For a predetermined seed (i_r, j_G) of a read r on the reference genome G, where i_r and j_G represent the exact match between the k-mer $r_{i..i+k-1}$ and $G_{j..j+k-1}$, we use a deterministic function $key_1(x)$ to generate a key $key_1(j+k-1)$ dependent on the genome location of the seed $j+k-1$ to encrypt the l-tuples in the read and the reference genome, respectively, using a symmetric encryption function extending from the $3'$-end of the seed, and use another deterministic function $key_2(x)$ to generate a key $key_2(j)$ to encrypt the l-tuples in the read and the reference genome, extending from the $5'$-end of the seed. In this case, the l-tuples extending from the same genomic location of seeds along the same strand ($5'$ or $3'$) are always encrypted by the same key, and thus the ciphertext of the l-tuples in the read and the corresponding reference is identical.

Furthermore, because the extension only involves l-tuples in the short (≤100 bps) flanking regions of a seed, we can precompute the encrypted l-tuples within a small window of 100 bps starting from each position j in each strand c ($c=1$ or 2) of the reference genome by using the key generated by $key_c(j)$. As a result, a total of $\approx 2\times 100$ encrypted l-tuples need to be precomputed for each l-tuple in the reference genome, and a sequence of ≈ 100 encrypted l-tuples is formed from those encrypted by using the same key $key_c(j)$ for a specific location j and strand c. These sequences of l-tuple are used for the extension from position j along the strand c when a seed is determined at the position in a read. The *side-wise* encryption allows different ciphertexts of the same l-tuples to be used in the extension of different seeds, and as a result, each encrypted l-tuple has about the same number of counts (≈ 200) and thus cannot be used to infer the occurrence of the l-tuple in the reference genome (or in the entire set of reads).

3. METHODS

Figure 1 illustrates the workflow of our secure read mapping approach, including the preprocessing of the reference human genome (Fig. 1a), the identification of alignment *seeds* between each read and the reference genome (Fig. 1b), and the extension of the seeds using l-tuple alignment (Fig. 1c). The preprocessing step needs to be carried out only once, while the next two steps are conducted for mapping each set of sequencing reads. Below, we present the algorithms involved in each of these three steps.

3.1. l-*Tuple alignment of two nucleotide sequences*

The *l-tuple alignment* is an extension of the conventional dynamic programming algorithm for pairwise sequence alignment. Let $S=s_1s_2\ldots s_N$ be a sequence of N nucleotides, which can be converted into a sequence of $N-l+1$ l-tuples: $R_S=r_1r_2\ldots r_{N-l+1}$, where r_i represents the l-tuple of $s_is_{i+1}\ldots s_{i+l-1}$. Note that when $l=1$, the l-tuple sequence R_S is identical with the original nucleotide sequence S; however, when l is large enough (specifically, when $4^l >> N$), each l-tuple in R_S is likely to be unique. Furthermore, there is a one-to-one correspondence between the nucleotide sequence S and the l-tuple sequence R_S for a fixed l. Given two nucleotide sequences S and T with lengths of N and M, respectively, the *l-tuple alignment* is the pairwise alignment between their l-tuple sequences R_S and R_T (of lengths $N-l+1$ and $M-l+1$, respectively) that rewards matched l-tuples and penalizes unmatched l-tuples and gaps. The l-tuple alignment is reduced to the pairwise sequence alignment when $l=1$. Notably, the optimal l-tuple alignment between two l-tuple sequences is identical with that between the two encrypted l-tuple sequences, in which each l-tuple is encrypted by using a symmetric encryption scheme, and the same key.

Figure 2 illustrates the l-tuple alignment using a schematic example for $l=3$. In the figure, the red substring ACATT represents a *seed* identified between a query read and the reference sequence. Our goal is to extend to seed into the rest of these two sequences: ACCTGGACT (read) and ACCTCGGACT (reference). There is a deletion in the read that can be detected in the sequence alignment, which corresponds to one deletion and two substitutions (of 3-tuples) in the 3-tuple alignment.

In general, one substitution in a nucleotide sequence alignment is equivalent to l consecutive substitutions in the l-tuple alignment, and one insertion (or deletion) is equivalent to one insertion (or deletion) plus $l-1$ consecutive substitutions in the l-tuple alignment. When multiple substitutions (or indels) occur in the sequence alignment between the read and the reference sequence, the corresponding l-tuple alignment depends on how these mutations are distributed in the sequence alignment. If two or more mismatches (substitutions or indels) occur adjacent to each other (with the distance $\leq l$), a subsequence of l-tuples between the first l-tuple, including the first mismatch, and the last l-tuple, including the last mismatch, in the read and the reference sequence cannot be aligned in the l-tuple alignment. On the other hand, if two mismatches are not adjacent (with a distance $> l$), each of them will lead to a series of l consecutive mismatches of l-tuples in the l-tuple alignment. In the practice of read mapping, the mismatches between the reads and the reference (genome) sequence are caused by either sequencing errors or genome variations, both occurring rarely. Therefore, we expect there are no more than five mismatches in the alignment between a read (of 100 nucleotides) and the reference genome. In this case, the pairwise sequence alignment can be well approximated by the l-tuple alignment.

The choice of l reflects the balance between security guarantee and the performance of l-tuple alignment. On the one hand, with larger l, the possibility of observing two identical l-tuples in the same read decreases, and thus the l-tuples are less likely to be recovered through frequency analysis. On the other hand, because one mismatch changes l consecutive l-tuples, larger l reduces the number of matched l-tuple in the l-tuple

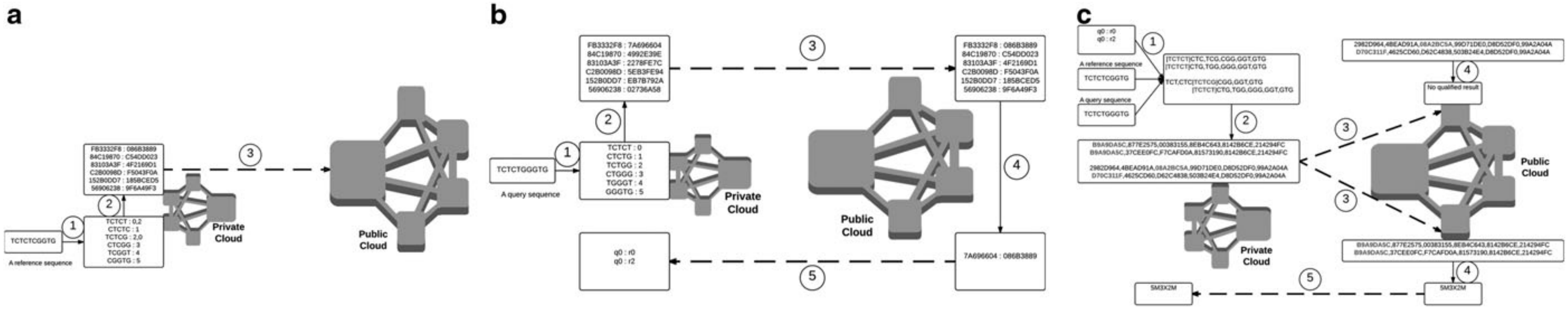


FIG. 1. The workflow of the secure read mapping approach. **(a)** The preprocessing of the reference human genome on the private cloud, and the encrypted *k*-mers and *l*-tuple sequences are transferred onto the public cloud. **(b)** The *k*-mers in the reads are encrypted on the private cloud and then matched against the encrypted k-mers from the reference genome to identify alignment *seeds* on the public cloud. **(c)** Each seed is extended into the *l*-tuple alignment based on the *l*-tuple encrypted using a site-specific key on the public cloud (see Section 3.1 for details).

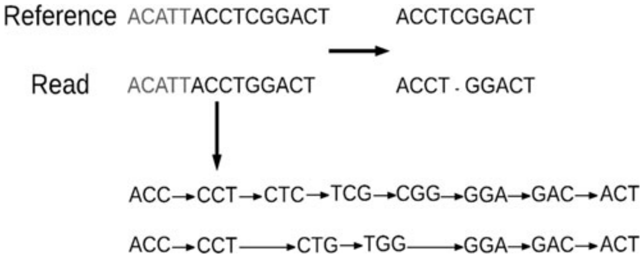

Reference ACATTACCTCGGACT
ACCTCGGACT
Read ACATTACCTGGACT
ACCT - GGACT
ACC→CCT→CTC→TCG→CGG→GGA→GAC→ACT
ACC→CCT→CTG→TGG→GGA→GAC→ACT

FIG. 2. The l-tuple alignment between two nucleotide sequences (bottom) in comparison with their pairwise sequence alignment (top).

alignment, leading to the inaccuracies in the l-tuple alignment. Therefore, we anticipate the l should satisfy two conditions: (1) $4^l >> N$, where $N = 100$ is the read length; and (2) $l << N/\delta$, where $\delta = 5$ is the number of mismatches we expect to observe in the sequence alignment between read and the reference genome. In practice, we often choose $l = 7$–10.

3.2. Site-wise encryption and security analysis

A important property of the l-tuple alignment is that the optimal alignment remains the same after the l-tuples in both strings are encrypted using a symmetric encryption scheme and the same *key*. This leads to the straightforward application of the l-tuple alignment to read mapping. Specifically, a user can align the encrypted l-tuple sequence of each reads with the encrypted l-tuple sequence of the reference genome, both encrypted using the same key, and then infer the variations between the reads and the reference genome from the l-tuple alignment.

This approach is, however, subject to an inference attack based on the frequency analysis of ciphertexts (encrypted l-tuples). Because the human reference genome is publicly available, an adversary can compute the frequencies and distribution of l-tuples in the reference genome, and can exploit this knowledge to infer the plain text of an l-tuple from its ciphertext by comparing the frequency of the encrypted l-tuple in an NGS dataset against the frequencies of all l-tuples in the reference genome. If we adopt a symmetric encryption algorithm to encrypt the l-tuples in all reads using a fixed key to perform the l-tuple alignment for read mapping, an adversary can easily compute the frequency of each l-tuple in the dataset. Moreover, because a read is represented as an l-tuple sequence in the alignment, an adversary can profile each read by a series of frequencies of l-tuples, and to infer the genome location of the reads (as well as potential variations contained in the reads) through the comparison of such profiles between the reads and the substrings in the reference genome.

To address this privacy issue, we devised a site-wise encryption scheme, which used different encryption keys to encrypt l-tuples for the extension of seeds at different genomic locations. Considering a seed (of alignment) as a k-mer (e.g., $k = 20$) in a read that matches a k-mer in the reference genome located at the position j, our goal is to extend the alignment between the read and the reference genome from the seed, achieved by using the l-tuple alignment between an encrypted l-tuple sequence of a read and the reference genome, using the *location-specific* keys of $f(j, 0)$ and $f(j + k - 1, 1)$, respectively, where f is the *key generation function* (e.g., a collision-free hash function) that takes as input the genomic location and the strand (indicated by 0 or 1) of the first l-tuple in the reference genome and generates a *key*.

Notably, because the keys are independent of the reads, the l-tuples in the reference genome can be encrypted in a preprocessing step (Fig. 1a) and reused for mapping multiple sets of reads. For each l-tuple starting at position j in the reference genome, a total of $2 \times L$ encryptions are needed, using a different key of $f(j - k + 1 - i, 0)$ and $f(j + i, 1)$, respectively, where L is the read length (e.g., $L = 100$). When a read is to be mapped to the reference genome, we first determine the locations of each seed (i.e., a k-mer in the read) in the reference genome. For each location, we then encrypt the l-tuples in the read using a corresponding key (based on the location of the seed), and then align the encrypted l-tuple sequence of the read with the preencrypted l-tuple sequence of the reference genome at the seed's location.

The site-wise encryption allows for l-tuple alignment between a read and the reference genome, while protecting the encrypted l-tuples from inference attacks based on frequency analyses. Because for each seed, a specific l-tuple is encrypted by using a different encryption key, each occurrence of an l-tuple in the reference genome is guaranteed to result in a different ciphertext. Therefore, an adversary cannot collect a sufficient number of identical ciphertexts encrypted using the same key from the same l-tuple to accurately estimate the frequency of the l-tuple in the reference genome and to infer the plain text of the l-tuple.

3.3. The seed-extension workflow

Following the generic seed-extension approach, our read mapping algorithm consists of two phases: (1) the *seeding* (Fig. 1b), where the locations of the *first* k-mer in a read on (both strands of) the reference genome, each referred to as a *seed*, are determined by comparing the encrypted k-mer against a large table of encrypted k-mers with respect to their locations in the reference genome, and (2) the *extension* of each seed in a read (Fig. 1c), where the l-tuples in the read are first encrypted by using the key specific to the location of the seed, and the resulting (encrypted) l-tuple sequence of the read is then aligned with the preencrypted l-tuple sequence (using the same location-specific key) of the reference genome using the l-tuple alignment algorithm.

To reduce the number of putative seeds to be extended in the second phase, we terminate the seed searching after we locate the *first* k-mer from the 5′-end of each read matching with a k-mer in the reference genome. Apparently, if there is one or more SNVs or sequencing errors occurring in the first k nucleotides in a read, the first k-mer in the read may not match with its counterpart in the reference genome, resulting in the read unmapped. In the worst case, if a read contains multiple distributed SNVs or sequencing errors, none of its k-mers will match their counterparts in the reference genome; as a result, no seed is found and thus the read will not be mapped onto the reference genome.

To address this issue, when searching for seeds, we allow up to ε substitutions in the matching between a k-mer in the read and a k-mer in the reference genome. In practice, we search for a k-mer against not only all k-mers in the reference genome but also all of their ε-*neighbors*, that is, the k-mers with $\leq \varepsilon$ substitutions (by default $\varepsilon = 1$; mostly due to SNVs). Note that because the sequencing error is typically low at the 5′-end of the short reads from Illumina sequencers, when we search for matching k-mer from the 5′-end of a read, we expect to identify a seed after the attempt of matching only a few k-mers and afterward, the searching process will be terminated. On the other hand, the k-mers with high occurrences in the reference genome often yield the seeds leading to low-quality alignments in the extending phase. To avoid the number of such noninformative seeds, in our implementation, we skip the k-mers whose *neighbors* occur $\geq \delta$ times in the reference genome. Because not all neighbors of a k-mer actually occur in the reference genome, we set δ as a reasonably low value (e.g., 30), which can effectively filter many noninformative seeds. We note that due to this filtration, some reads in which every k-mer are very frequent, may become unmappable. However, these reads are often not mapped onto a unique locus in the reference genome and thus are not used in the subsequent genomic analysis (e.g., for variation calling).

We stress that the k-mer matching can be performed on the ciphertext in the same way as in the plain text if the k-mers in the reads and the reference genome are encrypted by using a symmetric encryption algorithm with the same key, as illustrated previously. The k-mers in the reference genome can be encrypted in a preprocessing procedure (Fig. 1a) and reused for multiple read mapping tasks. In this procedure, a table of k-mers and their ε-neighbors in the reference genome (in plain text) is first constructed based on the indexed reference sequence using the Burrows–Wheeler transformation (BWT). Each k-mer in the reference genome is first searched against this table. If its occurrence is not greater than δ, its neighbors are searched iteratively, enumerating the entire set of its ε-neighbors.

To accelerate the substitution-tolerant k-mer search, we implemented it using both the forward and the backward BWT of the reference genome. As the default, ε is set to be 1, the allowed substitution is present in either the first half or the second half of a k-mer; for greater ε, the following search algorithm is only an approximation. The search for the one-neighbors of a k-mer is conducted on both the forward and the backward BWT. The backward BWT is used to identify the target loci that match exactly with the second half of the k-mer, and then putative neighbors are found through the comparison between the target sequence and the k-mer from the last to the first nucleotide of the first half of the k-mer. Similarly, the forward BWT is used to identify the target loci that match exactly with the first half of the k-mer, and then putative neighbors are found through the comparison between the target sequence and the k-mer from the first nucleotide of the first half of the k-mer. At any step, if the number of substitutions exceeds δ, the search is terminated, and the putative neighbor is abandoned. At the end, the occurrences of each k-mer and their neighbors in the reference genome are stored in the table (Fig. 1a-1).

The keyed hash function (i.e., VMAC, a block cipher-based message authentication code) is then adopted for the encryption of k-mers, which consists of a cryptographic hash function, H, and a secret cryptographic key, K_H. Fingerprints of distinctive keys are calculated as $H_{K_H}(s_i, s_i + 1, \ldots, s_{i+k-2}, s_{i+k-1})$. The hash function guarantees that k-mers are securely hashed to a ''random'' value by using a block cipher-based message authentication code and a universal hash function (Katz and Lindell, 2014). Note that both k-mers and their hash values are unique in the corresponding tables. A secret key is used to prevent hash values from being inferred. As the occurrences of k-mers in the reference genome vary drastically, if the locations of each k-mer are encrypted directly, the lengths of different k-mers' ciphertexts are considerably different, which is vulnerable to frequency analysis: an adversary may infer the plain text of k-mers by associating the lengths of their ciphertexts with the occurrences of k-mers in the reference genome. To mitigate this risk, we padded a random string to each k-mer with $\leq \delta$ occurrences (of itself or its ε-neighbors), to ensure the ciphertexts of k-mer locations have equal length across different k-mers. The locations of k-mers are then encrypted by a symmetric encryption function (i.e., AES, advanced encryption standard), E, and a key K_E (Fig. 1a-2).

In addition to generating the location table of k-mers and their neighbors, the second step of preprocessing the reference genome is to construct an encrypted l-tuple sequence at each genomic location, which will be used in the extension phase of the read mapping. The encryption is performed using a symmetric encryption function (AES) and a location-specific key (see previous paragraph for details). The preprocessing of the reference genome is performed only once on a private cloud, and the resulting ciphertexts are distributed on the public cloud (Fig. 1a-3).

Next, we present the procedure for mapping a set of reads onto the reference genome. First, each read is encrypted using the same procedure as the one described above for preprocessing the reference genome on a private cloud (Fig. 1b-1, -2): each k-mer in the read is hashed with the same keyed hash function H and the same hash key K_H; the locations of k-mers are encrypted using the same symmetric encryption function E in combination with the same encryption key K_E; and a read identifier is also hashed and its hash value is appended at the end of the ciphertext of the encrypted locations. After encryption, the list of hashed k-mers and their encrypted locations is transferred to a public cloud (Fig. 1b-3).

The *seeding* phase is then conducted on the public cloud (Fig. 1b-4). Two pairs of ciphertexts with the identical key values (i.e., the ciphertext of two k-mers, one from the reference genome and the other from a read, representing a *seed*) are identified on the public cloud, and then transferred back to the private cloud (Fig. 1b-5). They are then decrypted on the private cloud to obtain the set of seeds. Given the decrypted read identifier and a list of (sorted) locations of all seeds in the read, the seed located at the $5'$-end is selected for the extension phase using l-tuple alignment: the l-tuples in the read are encrypted using the key specific to the location of the seed in the reference genome (Fig. 1c-1, -2), and the resulting l-tuple sequence is transferred to the public cloud and compared with the corresponding encrypted l-tuple sequence from the reference genome there (Fig. 1c-3, -4). Finally, the l-tuple alignment is returned to the private cloud, and the read identifier and the mapping location on the reference genome are decrypted and reported together with the alignment (Fig. 1c-5).

3.4. Implementation on the Spark system

The Apache Spark (Zaharia et al., 2010) is a new MapReduce framework, which first decomposes a task into an acyclic data flow graph of operations and then executes it with the input data. It can recompute failed intermediate results by reexecuting the corresponding operations on the graph. Spark overwhelms Hadoop in two aspects, while retaining the scalability and fault tolerance. First, it provides an iterative MapReduce mode, and thus each iteration becomes a MapReduce job. Users can concatenate a series of MapReduce jobs for a computing task. Second, it enables the load of distributed datasets across the cloud into the memory and uses them for different tasks repeatedly. Spark also allows for retaining data into a local cache if it is used multiple times. Finally, Spark supports *broadcast variables*, which are distributed to each node only once, instead of once for each working thread on a node. Therefore, Spark is extremely useful when a read-only data structure of large size is used in each working thread.

In our application, because the volume of k-mers from both the reference genome and read is huge, matching them directly (e.g., by using a *merge* algorithm) results in moving a large amount of data through the potentially low-speed network of the public cloud, reducing the computing performance. To address this issue, after the encrypted k-mers are uploaded to the public cloud, a *bloom filter* is built from them. Bloom filter is a memory-efficient data structure supporting membership queries with a low false positive rate, p (Bloom, 1970). It builds a bit array of length m for a set, S, with n elements, which is subsequently used to check if any subsequent input is present in S. The entire bit array is initialized as 0, and then a particular cell is set to 1 if it is equal to the hash value (between 0 and m) of an element computed using one of multiple (w) independent hash functions. To check if an input element is present, the hash values are computed for all hash functions, and the corresponding cells are checked: if some cells are set to 0, the input is definitely absent in S; otherwise, the input is assumed to be present in S, with a small false positive rate. The false positive errors from one or more cells associated with the input may be set to 1 by some other elements in S by chance. The relationship of m, n, w, and p can be derived as $m = n \times p/(ln2)^2$and $w = m \times ln2/n$ (Bloom, 1970).

The bloom filter of the query k-mers (from reads) are uploaded, as broadcast variables in the Spark system, to every node in the public cloud, which is then utilized to check, for each of them, if it is equal to any encrypted reference k-mer predistributed on the local file system. The k-mers with their locations in the reads and reference genome (i.e., the *seeds*) are returned to the private cloud. Note that there are a small number of false positive seeds returned during this phase due to the bloom filter, which will be removed when preparing data for the second phase (i.e., the extension) on the private cloud. In the second phase of extension, site-wise encrypted l-tuple sequences are distributed to the public cloud; l-tuple alignment is applied to these sequences. The resulting cigar strings from the point of view of l-tuple are returned to the private cloud if the alignment scores are above a user-defined threshold.

4. RESULTS

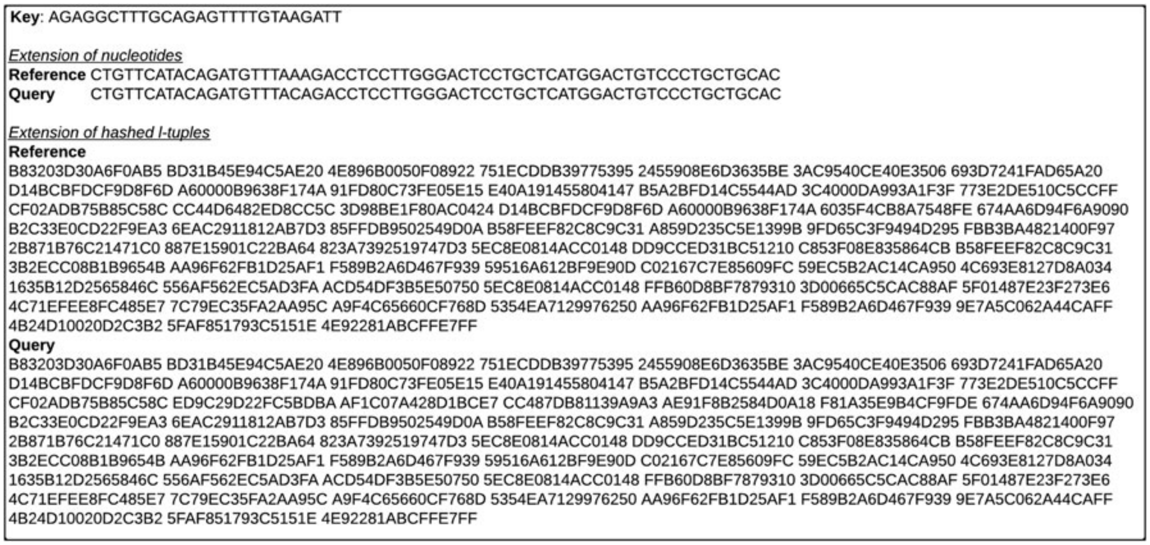
We implemented our methods in C++ for preprocessing the reference genome, and the encryption and decryption of reads on the private cloud, and for the seed and extension steps on the public cloud. We implemented the job scheduling system in Python under the Apache Spark system. All programs running on both the private and public clouds are integrated into a single software package, released as open-source software at github (https://github.com/zhaoyanswill/secureCloudAlign).

We tested the software using a hybrid cloud system consisting of a private and a public cloud. The private cloud is one Linux server equipped with a 32-core 2.60 GHz Intel Xeon CPU and 128 GB memory, whereas the public cloud comprised 14 nodes of FutureGrid, funded by National Science Foundation to provide service to both education and research projects, in which each node is equipped with 48-core 2.30 GHz Intel Xeon CPU and 128 GB of RAM.

The reference k-mer table is constructed on the private cloud for $k = 27$, $\varepsilon = 1$, and $\delta = 30$. All 27-mers on the reference human genome (Build GRCh38) were hashed by using VMAC. Their locations and their ε-neighbors were encrypted by using AES on the private cloud. It takes ≈ 6 hours to preprocess the reference genome sequences using 20 threads on the private cloud. The encrypted key-value pairs are then transferred to the public cloud.

We used human genome sequencing dataset from the 1000 Genome Project (NA21144, downloaded from the short read archive) for the testing purpose. The dataset consists of 10,336,136 reads of length 90 bps acquired by using Illumina Sequencers. The table of all 27-mers in the reads and their locations was generated and encrypted on the private cloud, and then uploaded to pubic cloud for the seed matching. We set the false positive rate of 0.1 when building the bloom filter. The bloom filters are distributed to different nodes as broadcast variables for checking locally if each 27-mer in the reads matches with any of those from the reference genome.

Those matched 27-mers (i.e., the seeds) from all nodes are merged together and returned to the private cloud. The seed is then decrypted, and the l-tuple sequences are prepared and encrypted in a site-wise manner on the private cloud. The resulting l-tuple sequences are transferred and distributed on the public cloud for the extension step using l-tuple alignment. Figure 3 shows an example of the alignment of two encrypted l-tuple sequences, in comparison with the pairwise alignment of two nucleotide sequences. Finally, for the alignments with scores above a threshold, the encrypted alignment along with the read indices and their locations on the reference genome are returned back to the private cloud. The private cloud will then decrypt the information to report the final alignments (Fig. 3).



```
Key: AGAGGCTTTGCAGAGTTTTGTAAGATT

Extension of nucleotides
Reference CTGTTCATACAGATGTTTAAAGACCTCCTTGGGACTCCTGCTCATGGACTGTCCCTGCTGCAC
Query     CTGTTCATACAGATGTTTACAGACCTCCTTGGGACTCCTGCTCATGGACTGTCCCTGCTGCAC

Extension of hashed l-tuples
Reference
B83203D30A6F0AB5 BD31B45E94C5AE20 4E896B0050F08922 751ECDDB39775395 2455908E6D3635BE 3AC9540CE40E3506 693D7241FAD65A20
D14BCBFDCF9D8F6D A60000B9638F174A 91FD80C73FE05E15 E40A191455804147 B5A2BFD14C5544AD 3C4000DA993A1F3F 773E2DE510C5CCFF
CF02ADB75B85C58C CC44D6482ED8CC5C 3D98BE1F80AC0424 D14BCBFDCF9D8F6D A60000B9638F174A 6035F4CB8A7548FE 674AA6D94F6A9090
B2C33E0CD22F9EA3 6EAC2911812AB7D3 85FFDB9502549D0A B58FEEF82C8C9C31 A859D235C5E1399B 9FD65C3F9494D295 FBB3BA4821400F97
2B871B76C21471C0 887E15901C22BA64 823A7392519747D3 5EC8E0814ACC0148 DD9CCED31BC51210 C853F08E835864CB B58FEEF82C8C9C31
3B2ECC08B1B9654B AA96F62FB1D25AF1 F589B2A6D467F939 59516A612BF9E90D C02167C7E85609FC 59EC5B2AC14CA950 4C693E8127D8A034
1635B12D2565846C 556AF562EC5AD3FA ACD54DF3B5E50750 5EC8E0814ACC0148 FFB60D8BF7879310 3D00665C5CAC88AF 5F01487E23F273E6
4C71EFEE8FC485E7 7C79EC35FA2AA95C A9F4C65660CF768D 5354EA7129976250 AA96F62FB1D25AF1 F589B2A6D467F939 9E7A5C062A44CAFF
4B24D10020D2C3B2 5FAF851793C5151E 4E92281ABCFFE7FF
Query
B83203D30A6F0AB5 BD31B45E94C5AE20 4E896B0050F08922 751ECDDB39775395 2455908E6D3635BE 3AC9540CE40E3506 693D7241FAD65A20
D14BCBFDCF9D8F6D A60000B9638F174A 91FD80C73FE05E15 E40A191455804147 B5A2BFD14C5544AD 3C4000DA993A1F3F 773E2DE510C5CCFF
CF02ADB75B85C58C ED9C29D22FC5BDBA AF1C07A428D1BCE7 CC487DB81139A9A3 AE91F8B2584D0A18 F81A35E9B4CF9FDE 674AA6D94F6A9090
B2C33E0CD22F9EA3 6EAC2911812AB7D3 85FFDB9502549D0A B58FEEF82C8C9C31 A859D235C5E1399B 9FD65C3F9494D295 FBB3BA4821400F97
2B871B76C21471C0 887E15901C22BA64 823A7392519747D3 5EC8E0814ACC0148 DD9CCED31BC51210 C853F08E835864CB B58FEEF82C8C9C31
3B2ECC08B1B9654B AA96F62FB1D25AF1 F589B2A6D467F939 59516A612BF9E90D C02167C7E85609FC 59EC5B2AC14CA950 4C693E8127D8A034
1635B12D2565846C 556AF562EC5AD3FA ACD54DF3B5E50750 5EC8E0814ACC0148 FFB60D8BF7879310 3D00665C5CAC88AF 5F01487E23F273E6
4C71EFEE8FC485E7 7C79EC35FA2AA95C A9F4C65660CF768D 5354EA7129976250 AA96F62FB1D25AF1 F589B2A6D467F939 9E7A5C062A44CAFF
4B24D10020D2C3B2 5FAF851793C5151E 4E92281ABCFFE7FF
```

FIG. 3. An example of the reported alignment on the encrypted *l*-tuple sequences (bottom) extended from 3′-end of a matched *k*-mer (top, also used to generate the key for the encryption), in comparison with the corresponding pairwise sequence alignment (middle).

The running time of these steps is summarized in Table 1. Note that the running time on the private and public clouds is not directly comparable, as the jobs on the public cloud were executed on 14 nodes, with 38 threads on each node (i.e., a total of 532 threads), while the jobs on the private cloud were executed with 20 threads. The entire process spends a total of 11 minutes on the private cloud using 20 threads (i.e., 220 minutes CPU time), except the preprocessing step, and 92 minutes on the public cloud using 532 threads (i.e., 48,944 minutes CPU time). These results indicate a majority of computation (48,944 out of 49,164 minutes CPU times, or 99.6%) were shifted to the public cloud.

We compared the reads alignments obtained by using our software with those obtained by using BWA-mem (Li, 2013). Among the 9,916,394 alignments reported by BWA-mem, 9,867,764 (99.5%) were also identified by our software, on which the alignments are identical. The remaining 48,630 (0.5%) alignments from BWA-mem were missed by our software. In each of these unmapped reads, every *k*-mer has more than δ occurrences on the reference genome, and thus was not considered in the extension step. Out of these reads, 17,410 are mapped to multiple locations in the reference genome (and thus are from a repetitive region), as reported by BWA-mem.

5. DISCUSSION

The secure read mapping on clouds arises as an urgent demand due to the long-term privacy risks of human genomic data. Conventional security protocols are not practical in this scenario. Some of them have

Table 1. The Running Time of the Secure Mapping Algorithm

Steps	*Preprocessing (one time)*	*Encryption of* k-*mers*	*Seed matching*	*Decryption of seeds*	*Encryption of* l-*tuples*	*Extension of alignments*	*Decryption of alignments*
Execution cloud	Private[a]	Private	Public[b]	Private	Private	Public	Private
Running time (minutes)	360	2	87	4	4	5	1

The jobs on the public cloud were executed on 14 nodes, with 38 threads on each node (i.e., a total of 532 threads), while the jobs on the private cloud were executed with 20 threads.

[a] The run time on the private cloud was measured using 20 threads.

[b] The run time on the public cloud was measured using a total of 532 threads (on 14 nodes, with 38 threads on each node).

to leverage either intensive computation or massive communication between multiple computing resources, which pushes human genome researchers away from utilizing public clouds and other elastic computing resources in human genome computation. On the other hand, existing approaches provide only approximate solutions, sometimes without reporting the actual alignment between the reads and the reference genome that are needed in downstream analyses (e.g., for variation calling).

In this article, we presented a practical approach to the secure read mapping problem on a hybrid cloud system. Our method enables human genome researchers to utilize computation and storage resources of public cloud computing platform without compromise of patients' privacy. While shifting most (>99.5%) computation to the public cloud, our method can also provide highly accurate read alignment results (depending on the choice of *l*), comparable with conventional read mapping tools. Our method leverages the site-wise encryption approach and the *l*-tuple alignment to mitigate the threats of inference attacks based on frequency analysis. Our implementation is ready to be practically applied to human genomic data analysis on the public cloud computing platforms.

One limitation of our approach is that intensive computation is required on the public cloud. Surprisingly, the major bottleneck is the seed matching step (Table 1), which took about 87 minutes when even 532 threads were used. This is largely due to the overhead of distributing data and jobs across nodes through the Spark system. In the future, we will optimize our implementation in the Apache Spark system to speed up the computation on the public cloud.

ACKNOWLEDGMENTS

This work was funded, in part, by the NIH/NHGRI (Grant No. R01HG007078, U01EB23685) and National Science Foundation (Grant No. CNS-1408874). The authors thank Diyue Bu for helpful discussions.

AUTHOR DISCLOSURE STATEMENT

No competing financial interests exist.

REFERENCES

Zhao, Y., Wang, X., and Tang, H. Secure reads mapping on a hybrid cloud: https://github.com/zhaoyanswill/secureCloudAlign

Annas, G.J., Glantz, L.H., and Roche, P.A. 1995. Drafting the genetic privacy act: Science, policy, and practical considerations. *J. Law Med. Ethics*. 23, 360–366.

Atallah, M.J., Kerschbaum, F., and Du, W. 2003. Secure and private sequence comparisons. Proceedings of the 2003 ACM workshop on Privacy in the electronic society, Washington, DC, pp. 39–44.

Bloom, B.H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*. 13, 422–426.

Braun, R., Rowe, W., Schaefer, C., et al. 2009. Needles in the haystack: Identifying individuals present in pooled genomic data. *PLoS Genet*. 5, e1000668.

Chen, Y., Peng, B., Wang, X., et al. 2012. Large-scale privacy-preserving mapping of human genomic sequences on hybrid clouds. NDSS Symposium 2012.

Collins, F.S., and Varmus, H. 2015. A new initiative on precision medicine. *N. Engl. J. Med*. 372, 793–795.

Datta, S., Bettinger, K., and Snyder, M. 2016. Secure cloud computing for genomic data. *Nat. Biotechnol*. 34, 588–591.

Fontaine, C., and Galand, F. 2007. A survey of homomorphic encryption for nonspecialists. *EURASIP J. Inform. Secur.* 2007, Article No. 15.

Goldreich, O., and Ostrovsky, R. 1996. Software protection and simulation on oblivious rams. *J. ACM*. 43, 431–473.

Gymrek, M., McGuire, A.L., Golan, D., et al. 2013. Identifying personal genomes by surname inference. *Science*. 339, 321–324.

Harmanci, A., and Gerstein, M. 2016. Quantification of private information leakage from phenotype-genotype data: Linking attacks. *Nat. Methods*. 13, 251–256.

Homer, N., Szelinger, S., Redman, M., et al. 2008. Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays. *PLoS Genet*. 4, e1000167.

Huang, Y., Evans, D., Katz, J., et al. 2011. Faster secure two-party computation using garbled circuits. Proceeding SEC'11 Proceedings of the 20th USENIX conference on Security, San Francisco, CA.

Jiang, X., Zhao, Y., Wang, X., et al. 2014. A community assessment of privacy preserving techniques for human genomes. *BMC Med. Inform. Decis. Mak*. 14(Suppl 1), S1.

Katz, J., and Lindell, Y. 2014. *Introduction to Modern Cryptography*. CRC Press, Florida, USA.

Kayser, M. 2015. Forensic DNA phenotyping: Predicting human appearance from crime scene material for investigative purposes. *Forensic Sci. Int. Genet*. 18, 33–48.

Li, H. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-mem. *arXiv preprint* arXiv:1303.3997.

Li, H., and Homer, N. 2010. A survey of sequence alignment algorithms for next-generation sequencing. *Brief. Bioinform*. 11, 473–483.

Lin, Z., Altman, R.B., and Owen, A.B. 2006. Confidentiality in genome research. *Science (New York, NY)*. 313, 441–442.

Popic, V., and Batzoglou, S. 2017. A hybrid cloud read aligner based on MinHash and kmer voting that preserves privacy. *Nature Communication*. 8, 15311.

Sankararaman, S., Obozinski, G., Jordan, M.I., et al. 2009. Genomic privacy and limits of individual detection in a pool. *Nat. Genet.* 41, 965–967.

Shringarpure, S.S., and Bustamante, C.D. 2015. Privacy risks from genomic data-sharing beacons. *Am. J. Hum. Genet*. 97, 631–646.

Stein, L.D. 2010. The case for cloud computing in genome informatics. *Genome Biol*. 11, 207.

Trapnell, C., and Salzberg, S.L. 2009. How to map billions of short reads onto genomes. *Nat. Biotechnol*. 27, 455.

van Nimwegen, K.J., van Soest, R.A., Veltman, J.A., et al. 2016. Is the $1000 genome as near as we think? A cost analysis of next-generation sequencing. *Clin. Chem.* 62, 1458–1464.

Yao, A. 1986. How to generate and exchange secrets. 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 162–167.

Zaharia, M., Chowdhury, M., Franklin, M.J., et al. 2010. Spark: Cluster computing with working sets. Proceeding HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, Boston, MA.

Zhao, Y., Wang, X., and Tang, H. 2015. Secure genomic computation through site-wise encryption. *AMIA Summits Transl. Sci. Proc.* 2015, 227.

Address correspondence to:
Prof. Haixu Tang
Department of Computer Science
School of Informatics, Computing, and Engineering
Indiana University, Bloomington
Bloomington, IN 47405

E-mail: hatang@indiana.edu