University of California Los Angeles
Department of Computer Science

Securing Masked Short Reads, Identification SNPs, Phenotypes, and Clinical Data using Cryptography

Student: *Sterling Engle, UID 904341227*

Capstone Project for: *MSCS* – Faculty Advisor: *Prof. Sriram Sankararaman*
Date: *February 19, 2022*

***Abstract*—Many people are understandably concerned about protecting their genomic privacy. For this reason, they are unwilling to donate their genomes for genomic research. Huang et al [16] discovered 116 identification (ID) SNPs[1] on the human genome that can be used to uniquely identify every individual. My `snpcrypt` Python [27] program extracts and encrypts these ID SNPs, and can return a subset of the sample genotypes. It encrypts masked extracted short reads of raw genomic data. It can encrypt any file containing phenotypes or clinical data. All files are encrypted and decrypted using a symmetric key that is protected by asymmetric RSA public key cryptography [24]. Multiple RSA public keys are supported for all file types for situations where more than one individual needs to access the same file. Encryption enables this extremely sensitive data to be securely stored in a genomic databank and securely transmitted to researchers and clinicians. I validated ID SNP extraction, optional sample selection, and encryption using a 2,504-sample chromosome 21 variant call format (VCF[2]) file [10] from phase 3 of the Human Genome Project [26]. I tested masked genomic sequence short read extraction and encryption with an indexed binary alignment map (BAM[3]) file [13] of one of the phase 1 human DNA samples. I verified error-free parsing of SAM and VCF versions 4.1, 4.2 [11], and 4.3 [12] files using scripts to run `snpcrypt` against all the "passed" test files in the Samtools `hts-specs` repository [20]. `Snpcrypt.py` source code is available in my public code repository: `https://github.com/sterling-engle/snpcrypt`.**

1. Introduction

1.1. Capstone Project Goals. My first goal was to design a practical means to use strong cryptography to secure human genomic data and maintain its privacy whenever this extremely sensitive data is distributed for research purposes. My second goal was to implement an efficient proof of concept and validate it with a significant amount of actual genomic data.

1.2. Genomic Privacy. Genomic data has a major impact upon privacy because of its extremely sensitive nature. The genome encodes information about an individual's genetic condition and predispositions to serious diseases. Unauthorized disclosure of such information could lead to discrimination, abuse and threats. For example, I would not want a hacker to illegally

[1] SNPs are single nucleotide polymorphisms, or precise locations in the DNA double helix where different base pair encodings are found among individuals.

[2] A variant call format (VCF) file contains many meta information lines, one header line, followed by data lines. Each data line contains information about a position in the genome, followed by genotype information for each DNA sample.

[3] BAM is a compressed binary version of a Sequence Alignment/Map (SAM) file. It contains the output of a DNA sequencing machine mapped to a reference sequence.

obtain my genome, discover that I have SNPs for Autism Spectrum Disorder, and threaten to disclose this to potential employers. Therefore, genomic data must be protected while at the same time it must be made available to researchers and for authorized healthcare purposes.

1.3. Short Read Sequences Aligned to a Reference Genome. DNA sequencing machines produce hundreds of millions of random short reads from a human genome.[4] Each short read is typically 100 to 400 nucleotides in length. Bioinformatic software aligns each short read to its position on a reference genome. The results are stored in a SAM file or its binary equivalent BAM file to save space [1]. Figure 1 shows an example of a short read in a SAM file. The short read data that should be protected for privacy reasons are the reference genome position, the CIGAR string (CS)[5] and the nucleotides in the read itself.

```
SRR035023.26847589  1209  1 9994  25  76M = 9994  0 CTTCCGATCTCCCTAACCCTAACCCTAACCCTAACCCT
AACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAA  >;>782=@@AA>?@@@@A@@??AA@@@@AA@>@@AA@@@@AA@@@AAA@A
@@AA@A@AAA@@@>@@@@???>?@AA>  X0:i:1  X1:i:0  MD:Z:0N0N0N0N0N0N0N1A0A66 RG:Z:SRR035023  AM:i
:0  NM:i:9  SM:i:25 XN:i:7  XT:A:U  BQ:Z:]Z]VWQ\__`NKLMMMMNMMLLNNMMMMNNMKMMNNMMMMNNMMMNNNM
NMMNNMNMNNNMMMKMMMMLLKLMNNK
```

Figure 1: Example short read coded in a SAM file. Each field is separated by a tab character. The first 11 fields are required. They are (in order): "the query template name, a bitwise flag, the reference sequence name, the 1-based leftmost mapping position, mapping quality, CIGAR string, reference name of the mate/next read, position of the mate/next read, observed template length, nucleotide segment sequence, and ASCII of Phred-scaled base quality+33" [13]. They may be followed by optional fields in a TAG:TYPE:VALUE format.

1.4. Universal Individual Identification SNPs. Humans share 99.9% of our genome. A single nucleotide polymorphism (SNP) occurs when one DNA sequence nucleotide[6] is substituted and that alternate allele exists in at least one percent of the population. Universal individual identification (ID) SNPs are those present in at least 35 percent of the population. At a level of 39%, known as the minor allele frequency (MAF), there are only 117. No ID SNP has a MAF $\geq 43\%$. 116 of these ID SNPs uniquely identify all humans, since their cumulative match probability (multiplying the MAFs together) with the rest of the population is between 2.01e-48 to 1.93e-50 [16], and there is no linkage disequilibrium (or correlation) between them.

2. Related Work

Ayday et al proposed and implemented "a privacy-preserving system to protect the privacy of aligned, raw genomic data" [1]. They begin with a SAM file as input, but they encrypt the short read position, CIGAR string (CS), and masked nucleotides in binary format all with different encryption schemes. After decryption, their result is not in a standard SAM or BAM file format. Instead, my program masks the selected regions to a standard SAM or indexed BAM file and encrypts it. My program supports multiple public key encrypted files containing the symmetric decryption key while they do not.

[4] The SAM/BAM file from the Human Genome Project for sample `NA06984` used to develop, debug, and test `snpcrypt` contains almost 335 million short reads.

[5] The acronym CIGAR stands for Compact Idiosyncratic Gapped Alignment Report. It is required field number 6 in the SAM/BAM format. It relates the segment sequence read to the reference genome. It contains a list of number of bases followed by an operator: M (alignment match), I (insertion to the reference), D (deletion from the reference), N (skipped region from the reference), S (soft clipping), H (hard clipping), P (padding), = (sequence match), or X (sequence mismatch) [13].

[6] The four constituent bases of nucleic acids are adenine (A), thymine (T), cytosine (C), and guanine (G).

In another paper, Ayday et al proposed and implemented a "privacy-preserving system for storing and processing genomic, clinical, and environmental data by using homomorphic encryption and privacy-preserving integer comparison" [2]. Unlike my project they encrypted the contents of all 50 million SNP positions for each individual. Their system architecture operates between medical and storage processing units, instead of using a cloud model. In contrast to the aforementioned effort, in this paper I describe a system that efficiently extracts and securely distributes to authorized researchers the 116 universal identification SNPs that can be used to uniquely identify an individual [16]. My program can be used to extract and encrypt any desired SNPs and a subset of the samples contained in the file.

In October, 2019, the Global Alliance for Genomics and Health adopted the "GA4GH File Encryption Standard" describing "a file format that can be used to store data in an encrypted state" [9]. The data is encrypted into 65,536 byte blocks. A file header is added including one or more header packets containing the encrypted data encryption key for the data blocks. The fixed-size data blocks can be read using an index. It provides a message authentication code (MAC), which "only protects the contents of each individual block. It does not protect against insertion, removal, or reordering of entire blocks".

It provides these additional features at the cost of additional complexity. GA4GH requires seven different keys: four asymmetric and three symmetric to encrypt each file. My system requires only three keys, two asymmetric and one symmetric. My system has no special encrypted file format. When the encrypted file is decrypted, the original file is recovered. The keys are stored in small external files using a simple naming convention based upon the data file name. Like GA4GH, my system supports multiple secured key access to the key that decrypts the data file.

3. Methods

3.1. Sequence Alignment/Map format file types and `pysam.AlignmentFile` Python object. Sequence Alignment/Map (SAM) files encode the mapping of short DNA sequence reads to a reference genome in a tab-delimited text format as shown in figure 1. BAM files are the binary equivalent of SAM files. The Python modules used by `snpcrypt` are shown in figure 2. A Pysam [15] module `AlignmentFile` object is used to read and write SAM and BAM files. The `pileup()` method is called to return each base of specified regions of an indexed BAM file. A list of desired regions is provided to the `snpcrypt --region` argument using samtools notation. For example, `--region=1:10000-20000,2:2000-3000` returns all the reads containing bases that were mapped to chromosome 1 reference positions 10000-20000 followed by those mapped to chromosome 2 positions 2000-3000.

```
import os
import sys
import argparse  # command line parsing library
import pysam  # ver. 0.18.0 lightweight Python wrapper of htslib C-API version 1.14
from pysam import AlignmentFile  # reads BAM and SAM files
from pysam import VariantFile  # reads VCF and BCF files
from cryptography.fernet import Fernet  # symmetric encryption; cryptography 36.0.0
from cryptography.hazmat.primitives.asymmetric import rsa  # RSA asymmetric encryption
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric import utils
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
```

Figure 2: System, BAM/SAM, VCF/BCF, and cryptography modules imported by `snpcrypt` program.

3.2. Short Read Masking. Often short reads will contain extra nucleotides that are not part of the requested region(s). These must be masked for privacy. `Snpcrypt` provides a `--mask` flag that requests the program to mask this data prior to encryption. The `pysam.PileupRead()` method returns a list of short reads aligned to each nucleotide in the requested region. Its `alignment` attribute is used to access the `pysam.AlignedSegment` object, whose `to_string()` method is called to return each line of the BAM file in SAM format. Each SAM field in the string is separated by a tab character. These tabs are used to split the string into a list of SAM fields. Field 10 (index 9) contains the query sequence. Initially the program replaces the entire sequence with `N` characters, which mean "no read", to mask out all of the nucleotides. Next, the program scans a list of query positions and a list of the sequences in the region returned by the `pysam.PileupColumn` object `get_query_names()` and `get_query_sequences()` methods, respectively. It replaces the `N` values with all of the sequence bases in the region for each short read. Figure 3 shows a typical short read before and after masking.

```
$ python snpcrypt.py --region=1:10011-10020 --outfile=unmask.sam NA06984.mapped.ILLUMINA.bwa.
CEU.low_coverage.20101123.bam

SRR035027.1910635 163 1 10003 23  76M = 10157 229 ACCCTACCCCTACCCCTACCCCTAACCCTACCCCTAACCCTAA
CCCTAACCCTACCCCTAACCCTAACCCTAACCC  >?@@?>=?@@A@?@@@BA?@@@BAB?@@BA?@@@AAB?@@AAB?A@BAB?>@AA>@@@
B>=>@@@;957+@@3:@+  MD:Z:6A5A5A11A23A21 RG:Z:SRR035027  AM:i:23 NM:i:5  SM:i:23 MQ:i:23 XT:A:
M  BQ:Z:@@@@@ABDC@@EFGEB@FFGEB@@@@@@@FFGEB@@@@@@@@@@@@@@@@@@@@EDFDA@@@@@@@@@@@@@@@@@

$ python snpcrypt.py --region=1:10011-10020 --mask --outfile=mask.sam NA06984.mapped.ILLUMINA
.bwa.CEU.low_coverage.20101123.bam

SRR035027.1910635 163 1 10003 23  76M = 10157 229 NNNNNNNNCCTACCCCTANNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN  >?@@?>=?@@A@?@@@BA?@@@BAB?@@BA?@@@AAB?@@AAB?A@BAB?>@AA>@@@
B>=>@@@;957+@@3:@+  MD:Z:6A5A5A11A23A21 RG:Z:SRR035027  AM:i:23 NM:i:5  SM:i:23 MQ:i:23 XT:A:
M  BQ:Z:@@@@@ABDC@@EFGEB@FFGEB@@@@@@@FFGEB@@@@@@@@@@@@@@@@@@@@EDFDA@@@@@@@@@@@@@@@@@
```

Figure 3: A typical short read before and after masking by `snpcrypt`.

3.3. Encryption types and Python modules. `Snpcrypt` utilizes both symmetric and asymmetic encryption methods provided by mature Python `cryptography` [6] modules. The program uses symmetric encryption provided by `Fernet` [5] to encrypt plaintext files with a randomly-generated 32-byte key. Fernet uses the Advanced Encryption Standard [7] in Cipher Block Chaining mode (it writes one fixed-size block at a time) with a 128-bit encryption key.

Asymmetric encryption by the `rsa` module using an RSA public key of the 32-byte key that encrypts each data file protects that key from compromise. The researcher or clinician uses `snpcrypt` to generate a secure, password-protected 4,096-bit RSA private key file and matching public key file for this purpose, supplying the public key to the databank. The remaining `cryptography` modules in figure 2 support RSA. `Snpcrypt` uses them to generate and load keys; and to encrypt and decrypt the symmetric key data.

3.4. Symmetric key encryption with researcher's public RSA key. I utilized the RSA public key cryptographic algorithm to encrypt the symmetric key since it is far more secure than symmetric methods for two reasons. Since the private key is never shared, even the `snpcrypt` program cannot decrypt the symmetric key file it creates for the researcher. The random symmetric key is generated once and never stored in plaintext form. Mathematical complexity also provides a higher level of security for information encrypted asymmetrically using an RSA public key, as described in the next section.

3.5. Private and public RSA key pair creation. In order to obtain encrypted files from the genomic databank, a researcher first runs `snpcrypt` with the `--genkeypath=`*key_id* and `--password=`*secret* options to generate a password-encrypted RSA private key saved to *key_id*`.key.private` and public key to *key_id*`.key.public`. The databank uses the RSA public key to encrypt files containing symmetric keys the researcher needs to decrypt requested data files. `Snpcrypt` calls the `getPrivatePublicKeys()` function, which calls:

```
private_key = rsa.generate_private_key(public_exponent=65537, key_size=4096)
public_key = private_key.public_key()
pem = private_key.private_bytes(encoding=serialization.Encoding.PEM,
                                format=serialization.PrivateFormat.PKCS8,
          encryption_algorithm=serialization.BestAvailableEncryption(password)
 )
pemPublic = public_key.public_bytes(encoding=serialization.Encoding.PEM,
                        format=serialization.PublicFormat.SubjectPublicKeyInfo)
```

to generate them. The `private_key.private_bytes()` method encrypts the private key with the password, returning bytes written to the private key file. The `public_key.public_bytes()` method does the same for the public key, except it is not encrypted.

I chose a longer 4096-bit RSA key length for additional security. `Snpcrypt` creates a public key for the user based upon two very large prime numbers,[7] along with an auxiliary value. These primes are kept secret. Messages (in this case symmetric keys) can be encrypted by anyone using the public key, but can only be decoded by someone who knows the prime numbers.

3.6. Generating multiple private and public RSA key pairs for testing. As shown in figure 4, the `snpcrypt` options `--genkeypath=`*key_id_list* and `--password=`*password_list* combine to generate pairs of password-protected private and public RSA keys for testing. For example, given *key_id* `user2` and simple test password `user2`, the program saves a 4,096-bit RSA private key encrypted by the password to file `user2.key.private`. The program writes the corresponding RSA public key to file `user2.key.public`. Each RSA public or private key is referenced on the `snpcrypt` command line by *key_id*, `user2` in this case.

```
(py39) sterline@Zeus:~/capstone$ time python snpcrypt.py --genkeypath=user2,user3,user4
--password=user2,user3,user4 --verbose
                  snpcrypt.py: verbose mode on
 generating RSA private key to: user2.key.private
  generating RSA public key to: user2.key.public
  reading RSA private key from: user2.key.private
   reading RSA public key from: user2.key.public
 generating RSA private key to: user3.key.private
  generating RSA public key to: user3.key.public
  reading RSA private key from: user3.key.private
   reading RSA public key from: user3.key.public
 generating RSA private key to: user4.key.private
  generating RSA public key to: user4.key.public
  reading RSA private key from: user4.key.private
   reading RSA public key from: user4.key.public

real    0m2.206s
user    0m1.946s
sys     0m0.000s
(py39) sterline@Zeus:~/capstone$ ls -l user[234]*
-rwxrwxrwx 0 sterline sterline 3434 Feb  6 02:03 user2.key.private
-rwxrwxrwx 0 sterline sterline  800 Feb  6 02:03 user2.key.public
-rwxrwxrwx 0 sterline sterline 3434 Feb  6 02:03 user3.key.private
-rwxrwxrwx 0 sterline sterline  800 Feb  6 02:03 user3.key.public
-rwxrwxrwx 0 sterline sterline 3434 Feb  6 02:03 user4.key.private
-rwxrwxrwx 0 sterline sterline  800 Feb  6 02:03 user4.key.public
```

Figure 4: Generating multiple password-protected private and public RSA key pairs for testing.

[7] "The security of RSA relies on the computationally challenging factorization of RSA modulus $N = p_1 p_2$ with N being a large semi-prime consisting of two primes p_1 and p_2, for the generation of RSA keys in commonly adopted cryptosystems" [21].

3.7. Encrypting masked short reads with a unique symmetric key protected by multiple RSA public keys. As shown in figure 5, the options `--region=`*region_list*, `--encrypt`, `--keypath=`*key_id_list*, `--outfile=`*output_path*, and `--mask` are combined to extract and mask a list of regions from an indexed BAM file to an encrypted BAM or SAM file. The randomly-generated symmetric encryption key is itself encrypted by each public key read from files whose names are generated from the *key_id_list*. For example, if the *key_id* is `user2`, the program appends `.key.public` to it and opens public key file `user2.key.public` for reading. The `snpcrypt` program `--keypath` option accepts a list of one or more key ids. Each public key is read into memory. All the short reads containing the requested regions are extracted from the indexed BAM file and the bases outside those regions are masked out with `N` characters. The results are written to a temporary file. The reference genome alignments are sorted by their leftmost coordinates and the results are saved to the BAM or SAM output file given by the `--outfile=`*output_path* command line argument. If the output file is a BAM file, indicated by a `.bam` extension, then it is also indexed, creating a file with the extension `.bam.bai`, which is not encrypted since it does not contain nucleotides.

A unique Fernet symmetric key is generated in line 1 of the code listing below. In lines 2-8, the symmetric key is encrypted by each RSA public key and the result is stored in the name of the output BAM or SAM file with a *.key_id*`.key` extension appended to it. For example, the `user2` RSA public key encryption of the symmetric key used to encrypt `test.bam` is stored in `test.bam.user2.key`.

```
1  key = Fernet.generate_key()  # Fernet generates a random 32-byte symmetric key
2  for i in range(len(RSAkeyFiles)):  # encrypt key with each public RSA key to its file
3    with RSAkeyFiles[i] as ek:
4      ciphertext = public_keys[i].encrypt(key,
5                    padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
6                    algorithm=hashes.SHA256(),
7                    label=None))  # returns RSA public key encrypted symmetric key
8      ek.write(ciphertext)  # save in corresponding .key file, e.g. 'test.bam.user2.key'
9  f = Fernet(key)  # get Fernet object for encryption using key
```

The output BAM or SAM file is read and encrypted using the unique symmetric key object obtained in line 9 above by calling its `encrypt()` method. The result is stored in a file with the `.bam.crypt` or `.sam.crypt` extension, respectively. The unencrypted `.bam` or `.sam` file is then deleted.

3.8. Decrypting masked short read BAM and SAM files. As shown in figure 6, the options `--decrypt`, `--keypath=`*key_id*, and `--password=`*secret* are combined to decrypt encrypted masked short read BAM and SAM files. In this example, the private key for researcher "user3" is read from file `user3.key.private`. On line 2 of the code listing below, the public key encrypted symmetric key used to encrypt the data file is read from file `2.bam.user3.key`. Decrypting the RSA public key encrypted symmetric key is performed on lines 3-7 using the researcher's private key. The encrypted BAM or SAM file, `2.bam.crypt` in this example, is decrypted to `2.bam` using the symmetric key object obtained in line 10 by calling its `decrypt()` method. The checksum verifies the integrity of the BAM file.

```
1  with RSAkeyFiles[0] as ek:  # opened file.bam.keyid.key or file.sam.keyid.key file
2    ciphertext = ek.read(512)  # read RSA public key encrypted symmetric key
3    key = private_keys[0].decrypt(
4                      ciphertext,
5                      padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
6                      algorithm=hashes.SHA256(),
7                      label=None))  # returns RSA private key decrypted symmetric key
8    if verbose and trace:
9      printlog(f"            Fernet symmetric key: {key}")
10 f = Fernet(key)  # get Fernet object for decryption using key
```

```
(py39) sterline@Zeus:~/capstone$ python snpcrypt.py --verbose --region=1:10000-10002
--encrypt --keypath=user2,user3,user4 --outfile=2.bam --mask N*.bam
                    snpcrypt.py: verbose mode on
   reading RSA public key from: user2.key.public
   reading RSA public key from: user3.key.public
   reading RSA public key from: user4.key.public
extracting region(s): ('1:10000-10002',) from
NA06984.mapped.ILLUMINA.bwa.CEU.low_coverage.20101123.bam to 2.bam
masking short reads with N's
sorting 2.bam
indexing 2.bam
encrypting extracted regions to: 2.bam.crypt
     with unique symmetric key protected by RSA public key in: 2.bam.user2.key
     with unique symmetric key protected by RSA public key in: 2.bam.user3.key
     with unique symmetric key protected by RSA public key in: 2.bam.user4.key
2.bam extracted short reads encrypted to 2.bam.crypt
     with unique symmetric key protected by RSA public key in 2.bam.user2.key
     with unique symmetric key protected by RSA public key in 2.bam.user3.key
     with unique symmetric key protected by RSA public key in 2.bam.user4.key
(py39) sterline@Zeus:~/capstone$ ls -l 2.bam.*
-rwxrwxrwx 0 sterline sterline  760 Feb  6 03:07 2.bam.bai
-rwxrwxrwx 0 sterline sterline 6244 Feb  6 03:07 2.bam.crypt
-rwxrwxrwx 0 sterline sterline  512 Feb  6 03:07 2.bam.user2.key
-rwxrwxrwx 0 sterline sterline  512 Feb  6 03:07 2.bam.user3.key
-rwxrwxrwx 0 sterline sterline  512 Feb  6 03:07 2.bam.user4.key
```

Figure 5: Encrypting masked short reads with a unique symmetric key protected by multiple RSA public keys.

```
(py39) sterline@Zeus:~/capstone$ python snpcrypt.py --verbose --decrypt --keypath=user3
--password=user3 2.bam
                    snpcrypt.py: verbose mode on
  reading RSA private key from: user3.key.private
   reading RSA public key from: user3.key.public
decrypting extracted regions to: 2.bam from 2.bam.crypt
     using unique symmetric key protected by RSA public key in: 2.bam.user3.key
2.bam extracted short reads decrypted from 2.bam.crypt
     using unique symmetric key protected by RSA public key in 2.bam.user3.key
(py39) sterline@Zeus:~/capstone$ sum 2.bam
35080     5
```

Figure 6: Decrypting masked short reads BAM file example.

3.9. Variant Call Format file types and `pysam.VariantFile` Python object. Variant Call Format (VCF) files encode genomic data in plaintext or a compressed binary format with an index. The compression ratio is approximately 51 to 1. A VCF file contains many meta information lines, one header line, followed by data lines each containing information about a position in the genome and genotype information on the DNA samples. The Python modules used by `snpcrypt` are shown in figure 2. The pysam module `VariantFile` object reads Variant Call Format (VCF) and their binary equivalent BCF files and writes the header. `Snpcrypt` writes the genotype VCF lines. It writes ASCII VCF files that can be compressed afterwards by the `bgzip` [17] program, and indexed by the `bcftools` [8] program.

3.10. Non-identification SNP extraction. Sankararaman et al developed a likelihood ratio (LR) test that can be applied to determine "an upper bound on the power of [detecting an individual genotype], which yields guidelines as to which set of SNPs can be safely exposed for a given pool size with a maximal allowable power β and false-positive level α" [25]. This test may be applied to determine the SNPs which do not require encryption.
Extracting non-ID SNPs into a VCF/BCF file is time-consuming but it only has to be done once per file. Figure 7 shows the `snpcrypt` program took about 2.3 minutes elapsed time to read the compressed VCF file, and write a 190 MB binary BCF file containing 1,105,529 SNPs with the

9 ID SNPs from chromosome 21 removed.

```
(py39) sterline@Zeus:~/capstone$ time python snpcrypt.py --verbose --pos=`cat
id_SNPs_pos.txt` --remove=c21_noIDs.bcf
                  snpcrypt.py: verbose mode on
   removing identity SNPs from:
   1000-genomes-phase-3_vcf-20150220_ALL.chr21.phase3_shapeit2_mvncall_integrated_v5a.
   20130502.genotypes.vcf.gz
                           to: c21_noIDs.bcf
          removed identity SNP: rs4539869 at pos 15479537 from output file
          removed identity SNP: rs4541312 at pos 15479678 from output file
          removed identity SNP: rs7278737 at pos 15481365 from output file
          removed identity SNP: rs1556277 at pos 15482718 from output file
          removed identity SNP: rs2826388 at pos 21926137 from output file
          removed identity SNP: rs2826390 at pos 21927356 from output file
          removed identity SNP: rs2826399 at pos 21935119 from output file
          removed identity SNP: rs10470220 at pos 21935399 from output file
          removed identity SNP: rs2831350 at pos 29389882 from output file
1105529 non-identity SNPs output to c21_noIDs.bcf and 9 identity SNPs removed

real    2m16.685s
user    2m18.402s
sys     0m10.712s
(py39) sterline@Zeus:~/capstone$ ls -l c21_noIDs.bcf
-rwxrwxrwx 0 sterline sterline 190887294 Feb 20 02:12 c21_noIDs.bcf
```

Figure 7: `Snpcrypt` command removing 9 chromosome 21 individual identification SNPs from a BCF output file.

Figure 8 shows the program took about 6.6 minutes elapsed time to read the same compressed VCF file, and write an 11 GB plaintext VCF file with the 9 ID SNPs from chromosome 21 removed.[8] Identity SNPs removal verification was accomplished by trying to extract them from the VCF/BCF files, which returned no data.

```
(py39) sterline@Zeus:~/capstone$ time python snpcrypt.py --verbose --pos=`cat
id_SNPs_pos.txt` --remove=c21_noIDs.vcf
                  snpcrypt.py: verbose mode on
   removing identity SNPs from:
   1000-genomes-phase-3_vcf-20150220_ALL.chr21.phase3_shapeit2_mvncall_integrated_v5a.
   20130502.genotypes.vcf.gz
                           to: c21_noIDs.vcf
          removed identity SNP: rs4539869 at pos 15479537 from output file
          removed identity SNP: rs4541312 at pos 15479678 from output file
          removed identity SNP: rs7278737 at pos 15481365 from output file
          removed identity SNP: rs1556277 at pos 15482718 from output file
          removed identity SNP: rs2826388 at pos 21926137 from output file
          removed identity SNP: rs2826390 at pos 21927356 from output file
          removed identity SNP: rs2826399 at pos 21935119 from output file
          removed identity SNP: rs10470220 at pos 21935399 from output file
          removed identity SNP: rs2831350 at pos 29389882 from output file
1105529 non-identity SNPs output to c21_noIDs.vcf and 9 identity SNPs removed

real    6m37.737s
user    2m51.031s
sys     0m18.411s
(py39) sterline@Zeus:~/capstone$ ls -l c21_noIDs.vcf
-rwxrwxrwx 0 sterline sterline 11243150879 Feb 20 02:32 c21_noIDs.vcf
(py39) sterline@Zeus:~/capstone$ wc c21_noIDs.vcf
    1105782  2778197919 11243150879 c21_noIDs.vcf
```

Figure 8: `Snpcrypt` command removing 9 chromosome 21 individual identification SNPs from a VCF output file.

3.11. Universal individual identification SNPs extraction and encryption with a unique symmetric key protected by multiple RSA public keys. As shown in figure 9, the options `--pos=`*base_ref_pos_list*, `--encrypt`, `--keypath=`*key_id_list*, and `--file=`*output_path* are combined to extract the list of ID SNPs identified by their base reference positions to an encrypted VCF file. A unique symmetric key encrypts the VCF header and extracted data to

[8] I increased performance by a factor of 9 writing the VCF file and added support for BCF file output during my capstone project.

output file *output_path*`.vcf.crypt`. As detailed in section 3.7, *key_id_list* contains a list of one or more key ids. Their RSA public keys are read and used to encrypt the symmetric key that decrypts the ID SNP VCF file. They are saved in files named *output_path*`.vcf.`*key_id*`.key` for later decryption. Extracting and encrypting 9 ID SNPs created a 170 KB binary file and three 512-byte RSA-secured symmetric key files in 1.073 seconds.

```
(py39) sterline@Zeus:~/capstone$ time python snpcrypt.py --verbose --pos=`cat
id_SNPs_pos.txt` --encrypt --keypath=user2,user3,user4 --file=c21ids
                  snpcrypt.py: verbose mode on
   reading RSA public key from: user2.key.public
   reading RSA public key from: user3.key.public
   reading RSA public key from: user4.key.public
   encrypting selected SNPs to: c21ids.vcf.crypt
     with RSA-protected key in: c21ids.vcf.user2.key
     with RSA-protected key in: c21ids.vcf.user3.key
     with RSA-protected key in: c21ids.vcf.user4.key
9 selected SNPs encrypted with unique symmetric key protected by RSA public key

real    0m1.073s
user    0m0.409s
sys     0m0.305s
(py39) sterline@Zeus:~/capstone$ ls -l c21ids.vcf.*[yt]
-rwxrwxrwx 0 sterline sterline 170926 Feb 20 17:31 c21ids.vcf.crypt
-rwxrwxrwx 0 sterline sterline    512 Feb 20 17:31 c21ids.vcf.user2.key
-rwxrwxrwx 0 sterline sterline    512 Feb 20 17:31 c21ids.vcf.user3.key
-rwxrwxrwx 0 sterline sterline    512 Feb 20 17:31 c21ids.vcf.user4.key
```

Figure 9: `Snpcrypt` command extracting and encrypting the 9 chromosome 21 ID SNPs to .crypt file.

3.12. Identification SNP file decryption. To decrypt the ID SNP file, the user gives the options shown in figure 10 to `snpcrypt`, including `--decrypt`, their RSA private *key_id* and password, and output *file* path. The program opens *file*`.vcf.crypt` (encrypted ID SNP VCF file) and *file*`.vcf.`*keyid*`.key` (RSA public key encrypted symmetric key file) for reading, and *file*`.vcf` for writing the decrypted file. It calls `readPrivatePublicKeys(keyPrivateFile, keyPublicFile, password)` to obtain the private key. The `decrypt` method provided by the RSA private key object returned from its instantiation is passed the 512 bytes of ciphertext read from the RSA public key encrypted file containing the Fernet symmetric key. The encrypted ID SNPs VCF file is read and each line (one per ID SNP) is decrypted using the Fernet key and written to the VCF output file. The elapsed time for this example was 0.457 seconds.

```
$ time python snpcrypt.py --verbose --decrypt --keypath=user4 --password=user4 --file=c21ids
                  snpcrypt.py: verbose mode on
  reading RSA private key from: user4.key.private
   reading RSA public key from: user4.key.public
 decrypting selected SNPs from: c21ids.vcf.crypt
     with RSA-protected key in: c21ids.vcf.user4.key
               decrypting to: c21ids.vcf
9 selected SNPs decrypted with unique symmetric key protected by RSA public key

real    0m0.457s
user    0m0.123s
sys     0m0.062s
(py39) sterline@Zeus:~/capstone$ ls -l c21ids.vcf
-rwxrwxrwx 0 sterline sterline 127499 Feb 20 22:51 c21ids.vcf
```

Figure 10: `Snpcrypt` command decrypting the 9 chromosome 21 ID SNPs to .vcf file.

3.13. Encryption and decryption of phenotype and clinical data files. `Snpcrypt` encrypts and decrypts all phenotype, clinical, and any other type of data file. If the input file extension is not recognized (i.e. not .bam, .sam, .vcf, or .bcf), then the program encrypts or decrypts the file depending upon the command line options. For example, the file extension

`.ped` is used for files containing pedigrees with numeric phenotype values [3]. In figure 11, using the command line:

```
$ python snpcrypt.py --verbose --keypath=user2,user3,user4 --encrypt sample.ped
```

a `sample.ped` file is encrypted with a unique symmetric key to `sample.ped.crypt` then deleted. That symmetric key is encrypted by three different RSA public keys, each stored in a file ending in `.key`. This provides secure file storage, transmission, and access to three different individuals. Each may run `snpcrypt` with their respective private key and password, which enables the program to decrypt the symmetric key and use it to decrypt the file.

```
(py39) sterline@Zeus:~/capstone$ sum sample.ped
34362    20
(py39) sterline@Zeus:~/capstone$ python snpcrypt.py --verbose --keypath=user2,user3,user4
--encrypt sample.ped
                    snpcrypt.py: verbose mode on
   reading RSA public key from: user2.key.public
   reading RSA public key from: user3.key.public
   reading RSA public key from: user4.key.public
sample.ped encrypted to sample.ped.crypt then deleted
     with unique symmetric key protected by RSA public key in sample.ped.user2.key
     with unique symmetric key protected by RSA public key in sample.ped.user3.key
     with unique symmetric key protected by RSA public key in sample.ped.user4.key
(py39) sterline@Zeus:~/capstone$ ls -l sample*
-rwxrwxrwx 0 sterline sterline 26252 Feb  9 20:10 sample.ped.crypt
-rwxrwxrwx 0 sterline sterline   512 Feb  9 20:10 sample.ped.user2.key
-rwxrwxrwx 0 sterline sterline   512 Feb  9 20:10 sample.ped.user3.key
-rwxrwxrwx 0 sterline sterline   512 Feb  9 20:10 sample.ped.user4.key
```

Figure 11: `Snpcrypt` command encrypting pedigree file `sample.ped` for three users.

Figure 12 provides an example of "user3" running `snpcrypt` using the command line:

```
$ python snpcrypt.py --verbose --keypath=user3 --password=user3 --decrypt sample.ped
```

to decrypt the `sample.ped` file encrypted in figure 11. In this case, the private key is read from file `user3.key.private` and the password given in the `--password` argument is used to decrypt it. An incorrect or missing password returns an error message. The RSA private key decrypts `sample.ped.user3.key`, returning the symmetric key. The symmetric key decrypts `sample.ped.crypt`, returning `sample.ped`. A checksum verifies decrypted file integrity.

```
(py39) sterline@Zeus:~/capstone$ python snpcrypt.py --verbose --keypath=user3
--password=user3 --decrypt sample.ped
                    snpcrypt.py: verbose mode on
  reading RSA private key from: user3.key.private
   reading RSA public key from: user3.key.public
          decrypting to: sample.ped from sample.ped.crypt
using RSA-protected key in: sample.ped.user3.key
sample.ped decrypted from sample.ped.crypt
  using unique symmetric key protected by RSA public key in: sample.ped.user3.key
(py39) sterline@Zeus:~/capstone$ sum sample.ped
34362    20
```

Figure 12: `Snpcrypt` command decrypting pedigree file `sample.ped` with "user3" private RSA key.

4. Evaluation

4.1. `Snpcrypt` validation using 1000 Genomes Project BAM data file. I downloaded the phase 1 subject `NA06984` BAM file:

`NA06984.mapped.ILLUMINA.bwa.CEU.low_coverage.20101123.bam`

and its associated `.bai` index file from a Google mirror site [22]. This file contains 334,736,893 short DNA sequence reads of this subject's genome mapped to a reference genome. I ran the `samtools` [8] utility program to convert the 24 GB BAM file to a human-readable 120 GB SAM file. I used these files to develop and validate the masked short read extraction, encryption, and decryption features of `snpcrypt`.[9]

4.2. `Snpcrypt` validation using 1000 Genomes Project VCF data file. I downloaded the phase 3 file:

```
1000-genomes-phase-3_vcf-20150220_ALL.chr21.phase3_shapeit2_mvncall_integrated_v5a
                                        .20130502.genotypes.vcf
```

from a Google mirror site [23]. I used this VCF file to develop and validate the SNP-related features of `snpcrypt`. The file contains 2,504 genotype samples of 1,105,538 SNPs located on chromosome 21. I choose chromosome 21 for practical reasons because the VCF file is only about 11 GB (213 MB compressed) while containing 9 ID SNPs as shown in figure 13 out of a total of 116 in the entire human genome. The `rs` dbSNP identifier prefix stands for "reference SNP". They are labels from the dbSNP database uniquely identifying SNPs at particular chromosomal offsets along reference human genome build 37 (GRCh37) [4].

dbSNP Identifier	GRCh37 Location
rs4539869	chr21:15479537
rs4541312	chr21:15479678
rs7278737	chr21:15481365
rs1556277	chr21:15482718
rs2826388	chr21:21926137
rs2826390	chr21:21927356
rs2826399	chr21:21935119
rs10470220	chr21:21935399
rs2831350	chr21:29389882

Figure 13: Nine chromosome 21 universal individual identification SNPs.

4.3. Error-free SAM and VCF file parsing verification. I verified error-free parsing of SAM and VCF versions 4.1, 4.2, and 4.3 files using small shell scripts to run `snpcrypt` against all the "passed" test files in the Samtools `hts-specs` repository [20].

4.4. `Snpcrypt` accuracy. I ran the Ubuntu Linux `sum(1)` command to calculate file checksums to verify the accuracy of BAM, SAM, BCF, VCF, and other data files that were encrypted and then decrypted by `snpcrypt` by comparing the checksums of encrypted and unencrypted versions. I used checksums to verify the accuracy of the `makeBytes(*args, sep="tab")` function that converts a variable list of VCF text arguments into bytes. I verified RSA and Fernet key generation by successfully using these keys for numerous encryption and decryption operations.

4.5. `Snpcrypt` performance. I measured `Snpcrypt` performance (see figure 14) on an idle PC.[10] Generating each RSA key pair consumes about 0.57s user time due to the mathematics involved. The short read masking algorithm performs slowly: $O(q*m*s)$ where for all extracted regions: q is the number of short read query names, m is the average number of matched

[9] `Snpcrypt` was developed on Ubuntu release 18.04.6 LTS [19] with Python version 3.9.7, cryptography 36.0.0, pysam 0.18.0, samtools 1.14, and htslib 1.14. I obtained the last three packages from bioconda [14].

[10] An Intel Core i5 @ 3.00Ghz PC with 64GB RAM @ 1337MHz and 894GB ADATA SX8200NP SSD.

sequences in the query names, and s is the number of short reads containing nucleotides. Masking 2,000 short reads from 22 regions spread across each non-sex chromosome averaged 24 seconds due to extraction of all regions into one set of lists, followed by a tri-level nested `for` loop that re-inserts the nucleotides that are not masked-out.[11] The algorithm can be improved significantly by masking each region separately. Other operations all exhibit good performance. One reason for this is the `pysam` module `snpcrypt` uses is a lightweight wrapper of the `htslib`[18] C-API. `Snpcrypt` takes advantage of `pysam` indexed read access to large BAM and compressed VCF files. Its command line arguments enable the program to be used in an automated, scripted workflow. The decryption operations perform exceptionally well.

`Snpcrypt` Operation	Elapsed	User Time	System
SAM: extract/mask 2,000 short reads (no encryption)	0m24.06s	0m21.40s	0m0.19s
SAM: extract/mask/encrypt/1 key 2,000 short reads	0m24.76s	0m21.82s	0m0.13s
SAM: extract/mask/encrypt/10 keys 2,000 short reads	0m24.75s	0m21.61s	0m0.16s
SAM: decrypt 2,000 short reads	0m00.29s	0m00.14s	0m0.00s
BAM: extract/mask 2,000 short reads (no encryption)	0m23.44s	0m21.38s	0m0.13s
BAM: extract/mask/encrypt/1 key 2,000 short reads	0m23.72s	0m21.64s	0m0.07s
BAM: extract/mask/encrypt/10 keys 2,000 short reads	0m23.94s	0m21.53s	0m0.12s
BAM: decrypt 2,000 short reads	0m00.28s	0m00.13s	0m0.01s
remove ID SNPs to BCF	2m16.69s	2m18.40s	0m10.71s
remove ID SNPs to VCF	6m37.74s	2m51.03s	0m18.41s
bgzip VCF file	1m33.54s	1m52.39s	0m18.99s
index VCF file	1m00.81s	0m40.76s	0m11.88s
generate 1 RSA key pair	0m01.51s	0m01.15s	0m00.19s
generate 3 RSA key pairs	0m02.21s	0m01.95s	0m00.00s
generate 10 RSA key pairs	0m06.50s	0m05.70s	0m00.04s
encrypt ID SNP VCF file for 1 user	0m00.55s	0m00.21s	0m00.14s
encrypt ID SNP VCF file for 3 users	0m01.07s	0m00.41s	0m00.31s
decrypt encrypted ID SNP VCF file	0m00.36s	0m00.11s	0m00.04s
make plaintext VCF	0m00.47s	0m00.28s	0m00.04s

Figure 14: `Snpcrypt` performance measurements.

5. Conclusions

My MSCS capstone project provided me the opportunity to study and learn several important genomic file format standards in great detail in order to design and code `snpcrypt`. I learned how to integrate cryptographic functions with low-level genomic data file access methods in Python. I designed and implemented genetic data extraction and masking functions. I designed and implemented a scheme that provides secure data storage, transmission, and access by multiple authorized individuals to extremely sensistive human genetic, phenotype, and clinical data without unnecessary design complexity.

Unlike passwords and credit card numbers, one cannot change their genetic code after it has been compromised. Therefore, ongoing research efforts worldwide to strengthen human genomic data security against attacks are critical to protect our privacy.

[11] The implementation goal was functionality instead of performance for this operation.

References

[1] Erman Ayday, Jean Louis Raisaro, Urs Hengartner, Adam Molyneaux, and Jean-Pierre Hubaux. Privacy-preserving processing of raw genomic data. *Data Privacy Management and Autonomous Spontaneous Security*, pages 133–147, 2013.

[2] Erman Ayday, Jean Louis Raisaro, Paul J. McLaren, Jacques Fellay, and Jean-Pierre Hubaux. Privacy-preserving computation of disease risk by using genomic, clinical, and environmental data. In *2013 USENIX Workshop on Health Information Technologies (HealthTech 13)*, Washington, D.C., August 2013. USENIX Association.

[3] Christopher Chang. File format reference - PLINK 1.9. https://www.cog-genomics.org/plink/1.9/formats, 2022.

[4] Genome Reference Consortium. Genome reference consortium human build 37 (grch37). https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.13, 2009.

[5] Individual contributors. Fernet (symmetric encryption). https://cryptography.io/en/latest/fernet, 2021.

[6] Individual contributors. Welcome to pyca/cryptography (version 37.0.0 - main). https://cryptography.io/en/latest, 2021.

[7] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, 1999.

[8] Petr Danecek, James K Bonfield, Jennifer Liddle, John Marshall, Valeriu Ohan, Martin O Pollard, Andrew Whitwham, Thomas Keane, Shane A McCarthy, Robert M Davies, et al. Twelve years of SAMtools and BCFtools. *Gigascience*, 10(2):giab008, 2021.

[9] Global Alliance for Genomics & Health. GA4GH File Encryption Standard. http://samtools.github.io/hts-specs/crypt4gh.pdf, 2019.

[10] Global Alliance for Genomics & Health. The Variant Call Format (VCF) Version 4.1 Specification. http://samtools.github.io/hts-specs/VCFv4.1.pdf, July 2021.

[11] Global Alliance for Genomics & Health. The Variant Call Format (VCF) Version 4.2 Specification. http://samtools.github.io/hts-specs/VCFv4.2.pdf, July 2021.

[12] Global Alliance for Genomics & Health. The Variant Call Format (VCF) Version 4.3 Specification VCFv4.3 and BCFv2.2. http://samtools.github.io/hts-specs/VCFv4.3.pdf, July 2021.

[13] The SAM/BAM Format Specification Working Group. Sequence Alignment/Map Format Specification. http://samtools.github.io/hts-specs/SAMv1.pdf, June 2021.

[14] Björn Grüning, Ryan Dale, Andreas Sjödin, Brad A Chapman, Jillian Rowe, Christopher H Tomkins-Tinch, Renan Valieris, and Johannes Köster. Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nature methods*, 15(7):475–476, 2018.

[15] Andreas Heger and Kevin Jacobs et al. pysam: htslib interface for python. https://pysam.readthedocs.io/en/latest/index.html, 2021.

[16] Erwen Huang, Changhui Liu, Jingjing Zheng, Xiaolong Han, Weian Du, Yuanjian Huang, Chengshi Li, Xiaoguang Wang, Dayue Tong, Xueling Ou, Hongyu Sun, Zhaoshu Zeng, and Chao Liu. Genome-wide screen for universal individual identification SNPs based on the HapMap and 1000 genomes databases. *Scientific Reports*, 8, 2018.

[17] Heng Li. Tabix: fast retrieval of sequence features from generic TAB-delimited files. *Bioinformatics*, 27(5):718–719, 2011.

[18] Genome Research Limited. Samtools. http://www.htslib.org, 2021.

[19] Canonical Ltd. Ubuntu 18.04.6 lts (bionic beaver) release. https://releases.ubuntu.com/18.04, 2021.

[20] Samtools organization. Specifications of SAM/BAM and related high-throughput sequencing file formats (hts-specs) testing files. https://github.com/samtools/hts-specs/tree/master/test, 2022.

[21] Anthony Overmars and Sitalakshmi Venkatraman. Mathematical Attack of RSA by Extending the Sum of Squares of Primes to Factorize a Semi-Prime. *Mathematical and Computational Applications*, 25(4):63, 2020.

[22] Human Genome Project. genomics-public-data: Na06984.mapped.illumina.bwa.ceu.low_coverage.20101123.bam and .bai. https://console.cloud.google.com/storage/browser/genomics-public-data/ftp-trace.ncbi.nih.gov/1000genomes/ftp/phase1/data/NA06984/alignment, 2010.

[23] Human Genome Project. genomics-public-data: 1000-genomes-phase-3/vcf/ALL.chr21.phase3_shapeit2_mvncall_integrated_v2.20130502.genotypes.vcf. https://console.cloud.google.com/storage/browser/genomics-public-data/1000-genomes-phase-3/vcf, 2015.

[24] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[25] Sriram Sankararaman, Guillaume Obozinski, Michael I Jordan, and Eran Halperin. Genomic privacy and limits of individual detection in a pool. *Nature genetics*, 41(9):965–967, 2009.

[26] Peter H Sudmant, Tobias Rausch, Eugene J Gardner, Robert E Handsaker, Alexej Abyzov, John Huddleston, Yan Zhang, Kai Ye, Goo Jun, Markus Hsi-Yang Fritz, et al. An integrated map of structural variation in 2,504 human genomes. *Nature*, 526(7571):75–81, 2015.

[27] Guido van Rossum and FL Drake. Python language reference, version 3. *Python Software Foundation*, 2019.