

École polytechnique de Louvain

Block Based Compression and Encryption of Genetic Information in VCF Files for Controlled Access

Author: **Tom GAUTOT**

Supervisor: **Benoit MACQ**

Readers: **Benoit MACQ, François-Xavier STANDAERT, Karim
EL'KHOURY, Pauline HERMANS**

Academic year 2023–2024

Master [120] in Computer Science and Engineering

Abstract

In recent years, the development of new technologies has brought down the cost of genome sequencing by several orders of magnitude. This has ushered in a new era of genomic analysis and has significantly increased the amount of genomic data that needs to be handled. Genomic variants are of utmost importance for research owing to some of them influencing biological functions. The most common way to work with these variants is through the Variant Call Format (VCF) files, which are often impractical to work with because of their size. Despite many compression algorithms having been developed for these files over the last few years, none of them offer a free, open-source solution that compresses the VCF file in its entirety while allowing for random access of the data. Additionally, despite genomic information being considered as sensitive data, security is often not incorporated in these algorithms. We introduce the Secure VCF Compressor (SVC) which fills both those holes: an open-source algorithm designed for compression and encryption of VCF files capable of random access. SVC's novel approach allows users to freely define regions of the files that will be independently compressed and encrypted, and thanks to its random access capabilities, SVC is capable of only decrypting and decompressing small parts of its archives, only those relevant to the users' queries, while keeping the rest of the file confidential. SVC combines GNU zip and GVC, an algorithm specialized in compressing genotype information, to compress the data, and then makes use of AES-GCM 256-bit to provide confidentiality and integrity to the data. While currently not able to compete with commercial solution compression-wise, SVC still consistently performs better than the de-facto solution and goes up to compression ratios of 200:1 on files from the 1000 Genome Project. Its security features are currently un-matched by any other academically available algorithm. The main contribution of this work is to provide a free, open-source, random-access capable algorithm for VCF compression and encryption, to enable an easier and much more secure handling, sharing and processing of sensitive genetic information.

Contents

abstract	1
1 Introduction	3
1.1 The Problem	3
1.2 VCF files	4
1.3 State of the art	7
1.4 Contribution	9
2 Secure VCF Compressor	11
2.1 Methods	11
2.1.1 The Original GVC Algorithm	12
2.1.2 GVC's re-implementation for SVC	15
2.1.3 SVC Pipeline	19
2.1.4 VCF to SVC	19
2.1.5 SVC to VCF	22
2.1.6 SVC file format	24
2.2 Security	27
2.3 Example Usage	29
2.4 Performances	30
2.4.1 Losslessness	30
2.4.2 SVC's performance depending on its parameters	31
2.4.3 SVC compared to other solutions	41
2.5 Future improvements	44
2.5.1 Security	44
2.5.2 Features	45
2.5.3 Compression Ratio	45
2.5.4 Time Performance	46
3 Conclusion	47

A	Appendices	48
A.1	Code Availability	48
A.2	Common INFO and FORMAT fields	48
A.3	Test Data	50
A.4	Test Machine Specification	50
A.4.1	Operating System	50
A.4.2	CPU	50
A.4.3	RAM	51
A.4.4	Disk	53
	References	54

Chapter 1

Introduction

1.1 The Problem

In the recent years, the improvements and reduction of costs brought to genomic sequencing technologies, coupled with the rising interest in genome analysis for human health, disease analysis, genetic selection and other, has led to an ever more complex problem of data storage among other things.

In the last decades, medical research has easily reached the state of "big-data" sector [17]. This data comes in multiple shape: biologic (e.g. biobanks), biometric and electronic (like medical records or images). Much research has been carried out into handling the multi-modal nature of this data, mainly for machine-learning usage [21] [22]. One modality of this data is the genome. A lot of genomic analysis is done using variations in gene sequences. These variations are stored in a Variant Caller Format (VCF) file. These files contain genomic data at multiple **loci** (a loci is a genomic position, defined by the chromosome, plus the position in this chromosome) in the sequenced genome, for potentially multiple individuals at once. The exact nature of the data contained in these files will be discussed later. This means that a single VCF file can grow quite large. For example, the VCF produced by the 1000 genome project for chromosome 22 (which is the second smallest one in humans, accounting for only 2% of the total DNA) sequenced for 2500 people already weighs 11GB when stripped of most of the sample-wise metadata and up to 15GB when filled with all the metadata (from [4]). And VCF aren't limited in use to human genomes, meaning that fast and efficient compression codecs are needed for this file format.

There is also a question of security, most notably of confidentiality (making sure that information remains secret and cannot be known by unauthorized parties)

and of integrity (detecting when data has been wrongly tempered with or simply corrupted). Genetic data is obviously considered sensitive, research has shown in the last few years that many diseases could be linked with specific gene variations, things such as cancers, diabetes, neurological diseases, etc. And with future research many more things may be uncovered about a person from their genome. DNA information should be encrypted to avoid it being accessed by people without authorization. There is also a need for integrity, some bad actors might want to change genetic information of some people in an attempt to undermine them; or try to specifically edit data of competing research firms to hinder them. This is why there needs to be secure storage, designed to not hinder the potential need of analysis by health professional or researchers.

The solution we propose is a program that divides all the records in the VCF into blocks, which can be compressed and, most importantly, encrypted independently. It also allows for random access (i.e. the capacity of accessing any block in constant time) making it possible to quickly retrieve the few blocks necessary to whatever purpose, and only decrypt those (using an authenticated encryption for integrity), leaving the rest secret.

1.2 VCF files

VCF files are the final product of 3-step pipeline, see Figure 1.1. The first step is the actual sequencing of the genome. When a genome is sequenced, the raw reads are first stored in a FASTA or FASTQ file, which simply contain the ATGC(X) sequence. The data contained in these files directly comes from the sequencing of genomes, this is usually done some form of Next Generation Sequencing (NGS). There exists different NGS platforms, some major figures (cost, read length, run time) can be found online [9]

The sequencing, at a very high level, works by first multiplying the DNA and dividing it into very small strands (depends on the technology, goes from a 100 to a few thousand [9]). These small strands are what is being "read". Because the DNA was first multiplied, each base in the original genome is read many times (the actual number being called the **coverage**) in different strands. Since these file contain the whole genome multiple times over but not in any orderly fashion, we use a **reference sequence** to align them. A reference sequence is simply a human genome that has been "standardized", and is being used by researcher when comparing DNA sequences. Once the reads are aligned to a reference, a SAM/BAM file is obtained. From these file, variants (loci where the reference and the read sequence differ) can be called, creating a VCF file. The variants can be 1 of 3:

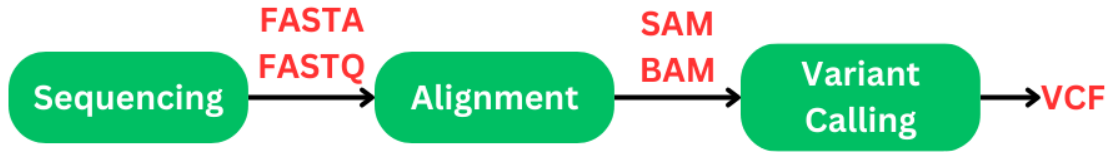


Figure 1.1: Pipeline to obtain a VCF

1. SNP: Single-nucleotide polymorphisms
2. INDEL: Insertion or deletion
3. SV: Large structural variants

Here is a brief explanation of the VCF file format and its content. Following is a very simple sample VCF file. Note that the line numbers aren't actually part of the file, they have been added for the sake of clarity and future reference in later explanations.

```

0 | ##fileformat=VCFv4.3
1 | ##fileDate=20090805
2 | ##source=myImputationProgramV3.1
3 | ##reference=file:///seq/0 | references/1000GenomesPilot-NCBI36.fasta
4 | ##contig=<ID=20,length=62435964,assembly=B36,md5=f126cdf8a6e0c7f379d618ff66beb2da,species="Homo sapiens",taxonomy=x>
5 | ##phasing=partial
6 | ##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
7 | ##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
8 | ##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
9 | ##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
10 | ##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
11 | ##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
12 | ##FILTER=<ID=q10,Description="Quality below 10">
13 | ##FILTER=<ID=s50,Description="Less than 50% of samples have data">
14 | ##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
15 | ##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
16 | ##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
17 | ##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
18 | ##FORMAT=
19 | #CHROM POS ID REF ALT QUAL FILTER INFO FORMAT NA000001 NA000002 NA000003
20 | 20 14370 rs6054257 G A 29 PASS NS=3;DP=14;AF=0.5;DB;H2 GT:GQ:DP:HQ 0/0:48:1:51,51 1/0:48:8:51,51 1/1:43:5:...
21 | 20 17330 . T A 3 q10 NS=3;DP=11;AF=0.017 GT:GQ:DP:HQ 0/0:49:3:58,50 0/1:3:5:65,3 0/0:41:3
22 | 20 1110696 rs6040355 A G,T 67 PASS NS=2;DP=10;AF=0.333,0.667;AA=T;DB GT:GQ:DP:HQ 0/2:21:6:23,27 2/1:2:0:18,2 2/2:35:4
23 | 20 1230237 . T . 47 PASS NS=3;DP=13;AA=T GT:GQ:DP:HQ 0/0:54:7:56,60 0/0:48:4:51,51 0/0:61:2
24 | 20 1234567 microsat1 GTC G,GTCT 50 PASS NS=3;DP=9;AA=G GT:GQ:DP 0/1:35:4 0/2:17:2 1/1:40:3
  
```

A VCF file starts with a *header*. The header lines are prefixed with **##** and give information such as which chromosomes are present, from which fasta reference file this VCF has been created, explanation on the INFO and FORMAT fields present, and more.

On line number 19 you can find the separation between the header and all the records. This line shows what metadata is present for each record, a record simply being a non-header line in the VCF file that contains information about a *variant* (a variant being a base where there was a difference with the reference sequence). The rest of the file is formatted as a Tab-separated-value file, with the column described on line 19:

- CHROM: On which chromosome this variant can be found.
- POS: At what position on the chromosome the variant is located.
- ID: An ID for the variant.

- REF: The allele of the reference sequence at this position.
- ALT: The alternative alleles possible at this position.
- QUAL: Quality score of the reading
- FILTER: PASS if all the filters were passed, a flag indicating the failed ones otherwise.
- INFO: A list of **key=value** pair, giving info about the variants, multiple pairs are separated with semicolon. The keys are described in the header. For more information on this, you can find common fields in the annex, Section A.2.
- FORMAT: Gives the format of the information that will be given for each sample for the current record. This format is a list of fields separated with colons. Some common fields are GT (genotype, present on all records) and DP (Read Depth). You can find more common format fields in the annex, Section A.2.
- Samples: After all the previous columns come a list of sample names, for each of which there will be the genotype, phasing and format information in the following records, in our case there are 3 samples: NA00001, NA00002 and NA00003.

To potentially clarify the FORMAT column and to explain the formatting of the SAMPLE columns, here is a quick rundown of line 22 from the sample VCF file.

- The FORMAT column gives **GT:GQ:DP:HQ** meaning that these 4 format fields should be present on every sample. GT means genotype and is pretty much always present. GQ is *Conditional Genotype Quality*, DP is *Read Depth* and HQ is *Haplotype Quality*. The definition of those acronyms can in fact be found in the header, at lines 15 to 18. For each of these fields, there will be a corresponding value on each sample, the value of different fields being separated by a colon and the order should be the same.
- The first sample (NA00001) shows **0|2:21:6:23,27**
 - The 0|2 represent the genotype on both of the chromosome. The first chromosome, having the value 0 means that it has the reference allele, the second chromosome having the value 2, means that it has the second alternative allele (T). The separator between these two values being a straight bar instead of an oblique one (/) means that the genotype is *phased* (instead of unphased, the meaning of phased/unphased is irrelevant for our task).

- The following values, separated by colons, represent the values associated with the formats listed previously. Since we have `21:6:23,27` we can for example know that this sample, for this variant, has a *conditional genotype quality* of 21 and a *read depth* of 6.

1.3 State of the art

The most commonly used tool for compressing VCF file is simply **GZip**, it is notably used for the 1000 genome project [4] files. Since it is a general use case tool, it comes with sub-par compression performances, and no security feature. The biggest advantage of this solution is all the tools that have been built around it. For example, there exists **BCFTools**, which can use GZip compressed VCF files just like raw VCF files. And with the usage of **tbi** files, which are index files for **.vcf.gz** files, **BCFTools** is even capable of random access.

Research into the topic of compression algorithm for VCF files in the last few years has spawned a few specialized solutions: GQT [10], BGT [11] SeqArray [14], GTRAC [12], GTC [15], GTShark [16], and GVC [23]. Those solutions focus on the compression of only GT information, (the 0|1 in a sample) and leave most (if not all) of the metadata behind, all the other columns, and all the other formats.

GQT, BGT and SeqArray, while they do provide some good compression, are works more geared towards indexing and advanced querying of VCF files, they will not be discussed further.

GTShark provides, by a significant margin, the best compression ratio but does not offer random access. GVC and GTC can be considered the 2 best algorithms if there is a need for random access. GVC offers a slightly better compression ratio while GTC allows for slightly faster random access. None of these however have any kind of security layer incorporated.

GTRAC, one of the earliest compression algorithm, aimed at providing genotype random access both record and sample wise. It achieved this through a customized Lempel-Ziv compressor and specialized data-structure (most notably, succinct bitvectors, which are compressed bit vectors that allow for random access on any bit of the sequence)

GTShark makes use of the Positional Burrows–Wheeler Transform (PBWT) to compress the genotypes. Its most important contribution is its capability to efficiently compress single-sample VCF file based on the knowledge of previously compressed files.

In high level, GVC works by first dividing all the records into blocks of a specific,

predetermined size, it encodes the genotype and phasing information into binary matrices, then sorts the rows and/or columns of these matrices according to some metric, to finally compress them using JBIG (or another entropy based binary image compressor). Since GVC plays an important part in SVC, it will be explained in greater detail in section 2.1.1.

GTC, despite showing some similarities, works a bit differently. It also starts by dividing the whole file into blocks of records. Once this is done, it encodes every single genotype value with two bits (the only accepted value are 0, 1, 2, and N/A) in a matrix, where the rows are the different records and the columns are the samples. The columns are then sorted (based on Hamming distance), and this final version of the matrix is encoded using Run Length encoding.

There are only a couple of algorithms working to compress the entire VCF files beyond GZip; VCFShark [19] and GenoZip [20].

The most notable algorithm is Genozip. It is a universal compressor, meaning that it can compress genomic data not limited to the VCF format. For VCF files, unlike most of the algorithm presented previously but just like VCFShark, it also compresses the metadata alongside the genotype information. Genozip also comes with a rudimentary security layer (it encrypts the whole file with a single key) that allows (premium) users to add a password when compressing a file, this however does not allow for limited access, if the password is brute-forced, the whole file can be decrypted by a bad actor. Genozip is not free for all use (commercial usage is prohibited without a license) and has a premium version with costs that, as of writing this, can go from 3,500\$/y to 20,000\$/y. The most expensive license is necessary for the encryption.

VCFShark provides very good compression ratios, at times better than Genozip's, but treats the entire file at once and thus doesn't allow for any random access. You can see in Figure 1.2 the results of VCF shark when put up against 4 other algorithms, being GenoZip, BCF (which is simple GZip), 7z (another general use case compression algorithm) and pigz (which is just a parallel implementation of GZip, with slight modifications). This highlights, as just explained, the lack of specialized tools for the compression of the entirety of VCF files. At the moment, there exist **no** free open-source specialized compression algorithm for whole VCF files capable of random access. Table 1.1 gives an overview of different characteristics of algorithms discussed in this Section as well as the solution we propose: **SVC**.

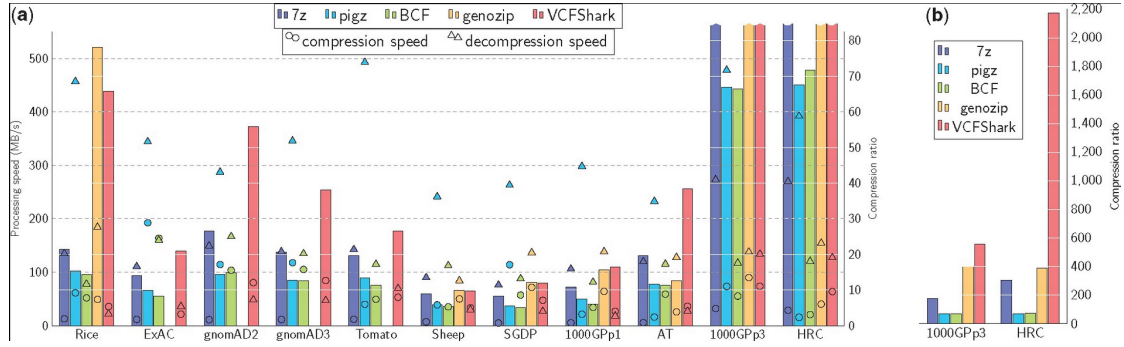


Figure 1.2: VCFShark results [19]. Verbatim caption: "Results of compression of 11 sets of VCF files using 5 different tools. (a) Compression ratio (vertical bars), compression and decompression speed (circle and triangle markers) for each tool and each dataset. To highlight the differences between the ratios on all sets, the compression ratios larger than 85 (achieved for 1000GPp3 and HRC sets) are cut off [see plot (b) for complete ratio results]. Genozip does not support the compression of some datasets (Tomato, ExAC, gnomAD2, gnomAD3]. (b) Compression ratios for 1000GPp3 and HRC datasets"

1.4 Contribution

What has been developed is a solution, the Secure VCF Compressor (SVC), the first free and open source, random access capable, whole VCF compressor, that also offers a security layer that divides the file into regions encrypted with different keys and thus easily control any person's access. These regions are defined by the user, making it easy to only communicate the keys necessary for a specific purpose. The default encryption scheme provided, being AES-GCM, is an authenticated encryption scheme, adding a check for data integrity. On datasets where most of the file is filled with genotype information, it offers much better compression than the de-facto solution of GZip.

	GZip	GTRAC	GTC	GTShark	GVC	GenoZip	VCFShark	SVC
Free	✓	✓	✓	✓	✓		✓	✓
Open-Source	✓	✓	✓	✓	✓	✓	✓	✓
Whole VCF	✓					✓	✓	✓
Random Access	~*	✓	✓		✓	✓		✓
Encryption						✓		✓
Controlled Access								✓

Table 1.1: Characteristics comparison of different solutions

*GZip is capable of doing random access with compressed VCF files but requires an index file

Chapter 2

Secure VCF Compressor

2.1 Methods

We propose SVC, an algorithm that uses existing compression techniques and adds on top of them a security layer designed to have a user-definable regions that will be encrypted using different keys. Thanks to this, SVC allows for decrypting only small parts of the file at once, and that provides both confidentiality and integrity to the data by using an authenticated encryption scheme.

The *user-definability* comes from SVC's expectation of a *region index file* provided and created by the user of the tool who can adapt it to their needs and wants. This is a simple comma-separated-value file, containing the (one per line) regions of interest in the genome encoded in the VCF file. These regions of interest could be genes or any other bigger/smaller genomic structures. Table 2.1 shows a snippet of such a file in a tabular format. The program will only look at the **chrom**, **start_pos** and **end_pos** columns, which together define a region that the program will encrypt, independently from other regions, using a key created for this one only.

Since, in addition to encryption, we also want to compress the whole VCF file and have random access capabilities, there is a need to implement a brand new codec (read: compression process) on the grounds that, as explained in the State Of The Art (SOTA) Section (1.3), currently no open-source project does this.

As a first step towards this, the VCF file's records will be divided up into the above-mentioned regions, and these regions will be further divided up into blocks, whose size can be chosen.

Once we have those blocks, we use two different algorithm for different parts

GeneID	Symbol	chrom	start_pos	end_pos
Inter-gene block		22	1	7
100996316	LOC100996316	22	8	32864
Inter-gene block		22	32865	48209
100287610	LOC100287610	22	48210	56399
Inter-gene block		22	56400	65296
102724197	LOC102724197	22	65297	86953

Table 2.1: Snippet of a region index file for chromosome 22

of the blocks. GVC, which was briefly introduced in SOTA (Section 1.3), was re-implemented from the ground up to compress the genotype and the phase information. A much more precise explanation of GVC’s inner workings, as well as a presentation of its performances will be given in Section 2.1.1. A description of the re-implementation, the choices, differences, and simplifications that were made for SVC are all discussed in Section 2.1.2. A complete re-implementation was necessary because:

1. GVC’s source code is in python, and since speed is paramount to make SVC a viable tool it was decided that it would be written in C++.
2. The code is quite convoluted and way more complex than necessary, so in the re-implementation it was stripped down to its essentials.

The rest of the blocks (the CHROM, POS, INFO... columns as well as the format values) is currently handled by GZip, although in the future we wish to move away from this and create/use a compression algorithm better suited to this data, see Section 2.5.3 for more about this.

#CHROM	POS	ID	REF	ALT	QUAL	FILTER	INFO	FORMAT	NA000001	NA000002	NA000003
20	14370	1	G	A	29	PASS	NS=3;DP=14;AF=0.5;DB;H2	GT:GQ:DP:HQ	0 0 48:1:51,51	1 0 48:8:51,51	1 1 43:5:.,.
20	17330	.	T	A	3	q10	NS=3;DP=11;AF=0.017	GT:GQ:DP:HQ	0 0 49:3:58,50	0 1 3:5:65,3	0 0 41:3
20	24445	3	A	G,T	67	PASS	NS=2;DP=10;AF=0.333;DB	GT:GQ:DP:HQ	1 2 21:6:23,27	2 1 2:0:18,2	2 2 35:4
20	26765	.	T	.	47	PASS	NS=3;DP=13;AA=T	GT:GQ:DP:HQ	0 0 54:7:56,60	0 0 48:4:51,51	0 0 61:2
20	30012	4	GTC	G,GTCT	50	PASS	NS=3;DP=9;AA=G	GT:GQ:DP	0 1 35:4	0 2 17:2	1 1 40:3

 Compressed with simple gzip
 Compressed using GVC

Figure 2.1: Division of data between compression techniques

2.1.1 The Original GVC Algorithm

This section aims at providing a high level understanding of the GVC algorithm a[23], the next section (2.1.2) will explain what simplification/shortcut were taken

to implement it in SVC and why.

In its original form, GVC works on blocks of records (whose size can be chosen) and focuses only on the GT field for all the samples. It represents all the genotype values and phase information for a block in a matrix, called the *Genotype Matrix*.

$$\mathcal{G} = \begin{pmatrix} 0|0 & 1|0 & 1/1 \\ 0|0 & 0|1 & 0/0 \\ 1|2 & 2|1 & 2/2 \end{pmatrix} \quad (2.1)$$

GVC processes and compresses this matrix in 5 keys steps: Genotype Pre-Processing, Splitting, Binarization, Sorting, and Entropy Coding. The authors proposed multiple binarization techniques that would result in slightly different binary matrices.

The goal of the first step, *Genotype Pre-Processing* is to change special values that some genotypes can have. This happens in two cases:

1. When the allele for a sample at a specific loci is not know, this is represented with a dot in the VCF file (for example 1|.), and VCF parsing libraries associate the value -1 with this case.
2. When a sample's ploidy (read: number of chromosome) is smaller than that of other samples for the same records. This case is associated with the value -2

The later stage of *Binarization* expects only positive numbers. In order to fix this, GVC takes the highest genotype value that can be found in the current block (a^{max}) and then changes the special values: $-1 \rightarrow a^{max} + 1$ and $-2 \rightarrow a^{max} + 2$. These two special values will later need to be stored since they are necessary to retrieve the original matrix. Note that, in the lower ploidy case, phase has no meaning but a value has to be generated to fill the matrix. By default it is set to *unphased* (oblique bar), but for potentially better compression it could be chosen based on the rest of the matrix. Lower ploidy being a rare case in the VCF files used, this was not explored in either GVC or SVC.

$$\mathcal{G} = \begin{pmatrix} 0|. & 1|0 & 1/1 \\ 0|0 & - & 0/. \\ 1 & 2|1 & 2/2 \end{pmatrix} \xrightarrow{\text{Pre-Processing}} \begin{pmatrix} 0|3 & 1|0 & 1/1 \\ 0|0 & 4/4 & 0/3 \\ 1/4 & 2|1 & 2/2 \end{pmatrix} \quad (2.2)$$

The algorithm then performs *Splitting* to transform the genotype and phase information into 2 matrices, an integer one for the genotype (called the *allele matrix*), and a binary one for the phase. This can be seen in Equation 2.3, where the matrix

$\mathcal{G}$ contains the genotype and phase information, which is then divided up between $\mathcal{A}$ and $\mathcal{P}$.

$$\mathcal{G} = \begin{pmatrix} 0|0 & 1|0 & 1/1 \\ 0|0 & 0|1 & 0/0 \\ 1|2 & 2|1 & 2/2 \end{pmatrix} \xrightarrow{\text{Splitting}} \mathcal{A} = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 2 & 2 & 1 & 2 & 2 \end{pmatrix}, \mathcal{P} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} \quad (2.3)$$

Once the matrices are split, it is necessary to binarize the allele matrix, not the phase matrix since it is already just binary data (phase can only be *true* or *false*). The authors proposed 2 different binarization techniques called *Bit-Plane Binarization* and *Row Binarization*. The bit-plane binarization works by creating n binary matrices (bit-planes) of the same size as the allele matrix. If we call a_{max} the maximum value found in the allele matrix, then $n = \lceil \log_2(a_{max} + 1) \rceil$. These bit planes are then filled such that, at any position in the plane, the q -th bit plane ($\mathcal{B}_q$) contains the q -th least significant bit of the value at the same position in the allele matrix. Once this is done, the bit-planes can be concatenated row or column wise to get one singular binary matrix. We can see that in Equation 2.4, because $\mathcal{A}_{1,0} = 3 = 11_2$ we have $\mathcal{B}_{0,1,0} = 1$ and $\mathcal{B}_{1,1,0} = 1$

$$\mathcal{A} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 3 & 0 & 2 & 1 \end{pmatrix} \xrightarrow[\text{Binarization}]{\text{Bit-plane}} \mathcal{B}_0 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}, \mathcal{B}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad (2.4)$$

The row binarization works slightly differently, it will encode the allele matrix one row at a time. Each row i in the allele matrix is encoded as n binary rows (using naive binary encoding) where $n = \lceil \log_2(a_i^{max} + 1) \rceil$ and a_i^{max} is the maximum value found in the i -th row of the allele matrix. All these binary rows for all the rows of the allele matrix are concatenated row-wise. Note that, while the bit plane technique only necessitated the number of planes in order to decode the data, the row-binarization needs to know how many binary rows are needed to reconstruct a row from the allele matrix. This means that an additional vector of integers, that we call the *Rows For Record* vector, has to be stored and passed to the decoder.

Row binarization is exemplified in Equation 2.5, where $a_0^{max} = 1$, $a_1^{max} = 0$ and $a_2^{max} = 3$. This means that encoding the first row of $\mathcal{A}$ will take $\lceil \log_2(1 + 1) \rceil = 1$ line, the second line will take $\lceil \log_2(0 + 1) \rceil = 0$ and the third one will take $\lceil \log_2(3 + 1) \rceil = 2$. From this, the encoding of the 3rd line of $\mathcal{A}$ starts at the second line of $\mathcal{R}$.

$$\mathcal{A} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 2 & 1 \end{pmatrix} \xrightarrow[\text{Binarization}]{\text{Row}} \mathcal{R} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}, V = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \quad (2.5)$$

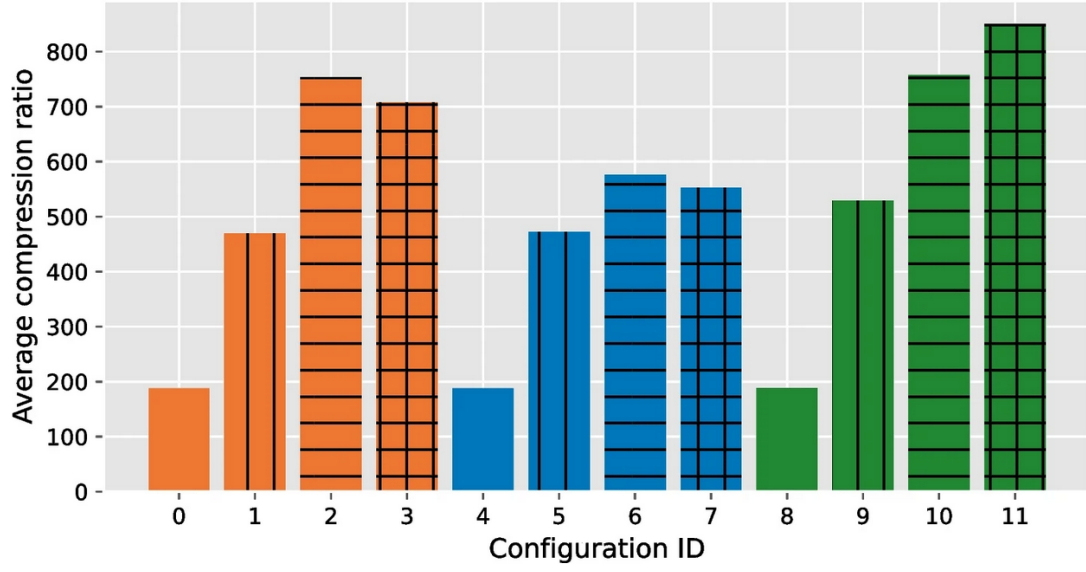
Now that we only have binary matrices, we could already proceed to compression using some binary image compressor. However GTC [15] proposed and showed the advantage of column-wise (i.e sample-wise) sorting on genotype based binary matrices. The reasoning behind this idea is that some samples, belonging to similar groups, are expected to have similar genotype variations. The main goal behind sorting binary matrices is to maximize run lengths which should help entropy coders achieve better compression ratios.

The team behind GVC, who iterated on GTC’s sorting idea, then also experimented with row-wise (record-wise) sorting. This sorting is done by first computing the (Hamming, or other) distance between every pair of vectors (the vectors being the rows or the columns depending on which axis the sorting is being done). The result of this is a distance matrix, which can be represented as a complete weighted graph, where the nodes are the vectors, in which we are looking for the shortest path that visits all nodes once. The visiting order gives us the new order to arrange the vectors. This is the definition of the Travelling Salesman Problem. Since this problem is NP-Hard, GVC finds an approximate solution via nearest-neighbor heuristic, but other heuristics could be used.

The last step, entropy-coding, can be done using any existing entropy-coder capable of handling binary images. GVC can be linked to any library available on the computer with simple settings, but by default it gives a guide to help the user to setup JBIG (more precisely, GVC’s python code calls an executable compiled from the open-source implementation of the JBIG standard called JBIG-KIT [1]). In the end, GVC got the results shown in Figure 2.2 and 2.3, please note that the compression ratios are somewhat misleading since they were obtained by comparing the total size of the VCF file to the compressed file that only contains genotype and phase information:

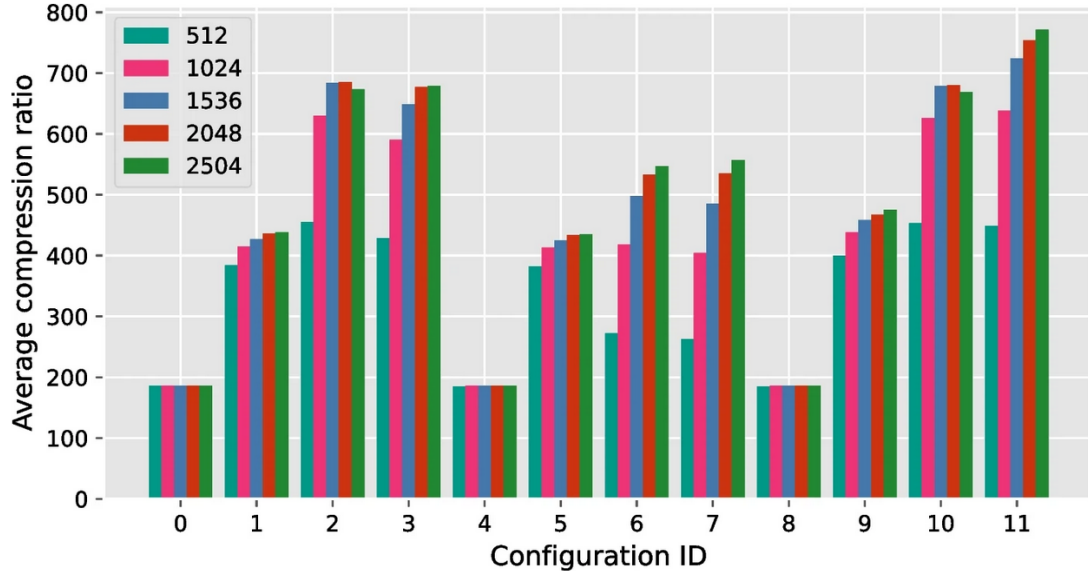
2.1.2 GVC’s re-implementation for SVC

As said previously, GVC was written in python. Since an important performance factor for these types of algorithm is speed and memory efficiency, we wanted to write SVC in C++. This means that GVC would need to be re-implemented from scratch, we decided to only re-create a single one of the proposed configuration (you can see these configuration on the X-axis of Figure 2.2). Since better compression



Average compression ratio (averaged over all chromosomes) achieved by each GVC configuration. The colors indicate the employed binarization: orange—bit plane binarization (concatenated), blue—bit plane binarization, green—row binarization. The patterns indicate the employed sorting: no pattern—no sorting, vertical lines—column sorting, horizontal lines—row sorting, both horizontal and vertical lines—sorting in both directions

Figure 2.2: From [23]



Compression ratio achieved by each GVC configuration with different block sizes. For all configurations with the sorting enabled, increasing the block size increases the compression ratio

Figure 2.3: From [23]

ratio are a good thing, the selection had to be done between configuration 2, 3, 10 and 11.

Configurations 3 and 11, which are characterized by sorting the binary matrices on both axes, were ruled out due to implementation reasons: The most expensive step of the whole SVC process complexity-wise (not just the GVC part) is computing the hamming distance matrix. Let's define n as the number of rows in our matrices, which is proportional to the block size, and m as the number of columns, which is proportional to the number of samples. Then the complexity of computing the row-wise hamming distance matrix is $\mathcal{O}(n^2m)$, and the complexity for the column-wise distance matrix is $\mathcal{O}(m^2n)$. The test files used to obtain the results of Figure 2.2 and 2.3 are from the 1000 genome project [4] and contain a number of samples in the order of thousands. And, from Figure 2.3, GVC achieves its best possible compression ratios when the block size is also in the order of thousands.

From this, we see that the number of operations needed to compute a single distance matrix is in the order of 10^9 . As a rule of thumb, consumer machines are usually capable of doing 10^8 operations per second. So at first sight it would take about 10 seconds to compute a single hamming matrix for a binary 1000×1000 matrix, this is later confirmed experimentally, see Figure 2.4. Since a complete VCF file can easily contain enough records for many thousands of blocks, processing a single file could easily take multiple hours (10^3 blocks $\times 10$ s = 2h40). Fortunately, the use of two simple tricks can greatly reduce this:

1. Parallelization, which is quite simple in this case since there is no dependency of results, the distance between row 1 and 2 is not needed to compute the distance between row 2 and 3. As long as the number of threads stays small compared to the matrix size and there are enough cores to handle the threads, the speedup is expected to grow linearly with the number of threads.
2. It is also possible to do some significant constant/implementation optimization. The hamming distance between two binary vectors is equal to the number of different bits between the two vectors. So, assuming the binary vectors are stored as an int8 (byte) array in the C++ code, the naive way to compute it would be to isolate every bit of each byte in both vector and compare them using XOR. Here is the sketch of such an algorithm in C trying to compare two 4000 bit-long bit-vectors.

```

1| uint8_t a[500], b[500];
2| int dist = 0;
3| for(int i = 0; i < 500; i++){
4|     for(int bit = 1; bit < 256; bit*=2){ // 8 iterations
5|         dist += ((a[i] | bit) ^ (b[i] | bit)) != 0;
6|     }
7| }

```

The `bit` variable from line 4 is always a power of two (i.e. a single bit), and it is used on line 5, if the $\log_2(\text{bit})$ th bit of the i -th element of a and b differ, then the distance is incremented by one.

Performance can be gained by simply removing the inner loop. Because the hamming distance between 2 bytes is simply the number of bit that differ between them (i.e. the number of bits in their XOR), and because there is only 256 different values for a byte, it's easy to pre-compute all the possible values. This gives the following algorithm:

```

uint8_t bitsInByte[256] = {0, 1, 1, 2, ..., 7, 7, 8};
uint8_t a[500], b[500];
int dist = 0;
for(int i = 0; i < 500; i++){
    dist += bitsInByte[a[i] ^ b[i]];
}

```

Thanks to this we shave off almost an order of magnitude of all the operations necessary, making the algorithm run in reasonable time even on consumer-level laptops. Because this optimization is only possible on one axis (over rows if the array is stored row-major and over column otherwise), it isn't possible to benefit from this optimization for both sorting axes. See Figure 2.4, which plots the time taken by both the optimized and unoptimized versions of the algorithm to compute the hamming matrix, from a binary matrix of variable size, as well as the speedup between the two.

With this, the choice was limited to configuration 2 and 10, which differ only by their binarization technique. We chose the latter as row-binarization seemed simpler to implement since only a single structure had to be allocated, meaning less memory movement and management.

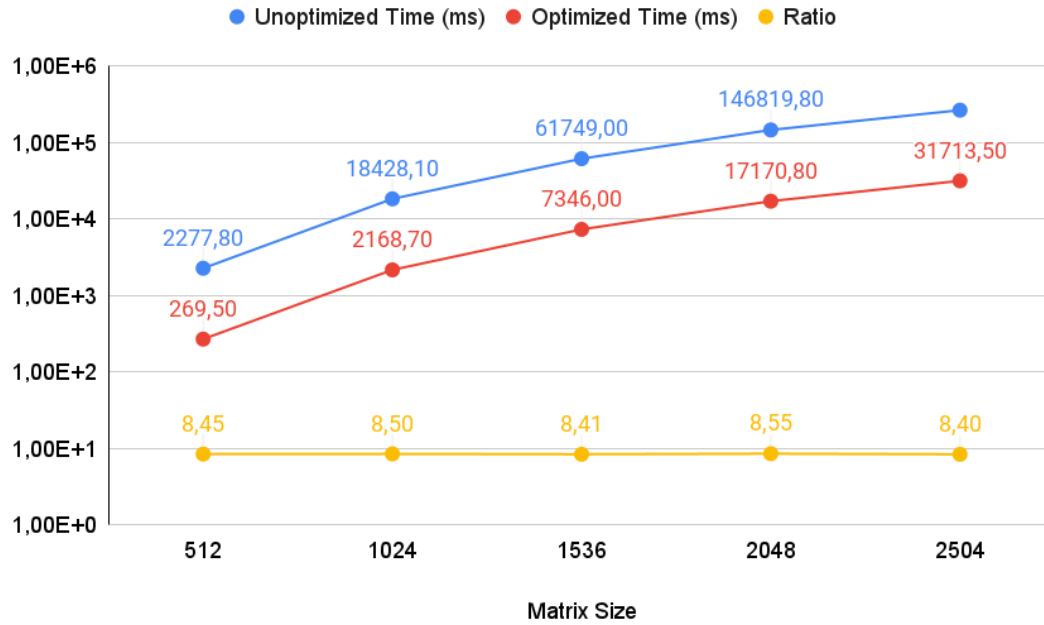


Figure 2.4: Speed comparison for hamming matrix computation between optimized and un-optimized loops (single thread, square binary matrix)

2.1.3 SVC Pipeline

2.1.4 VCF to SVC

Figure 2.5 represents a simplified version of the data pipeline for SVC when compressing and encrypting a VCF file, this will first be explained, then the inverse process (decryption and decompression) and its necessary additional steps will also be explained. This section will provide insight on every step of the procedure.

0. The program expects several items as arguments:

- -f A path to the VCF file to handle
- -i A path to an index file, defining the regions of interest the user wishes to compress.
- -j The number of parallel "processing threads" for the processing of blocks.
- -o A filename under which to store the compressed data
- -k A filename under which to save the generated encryption keys

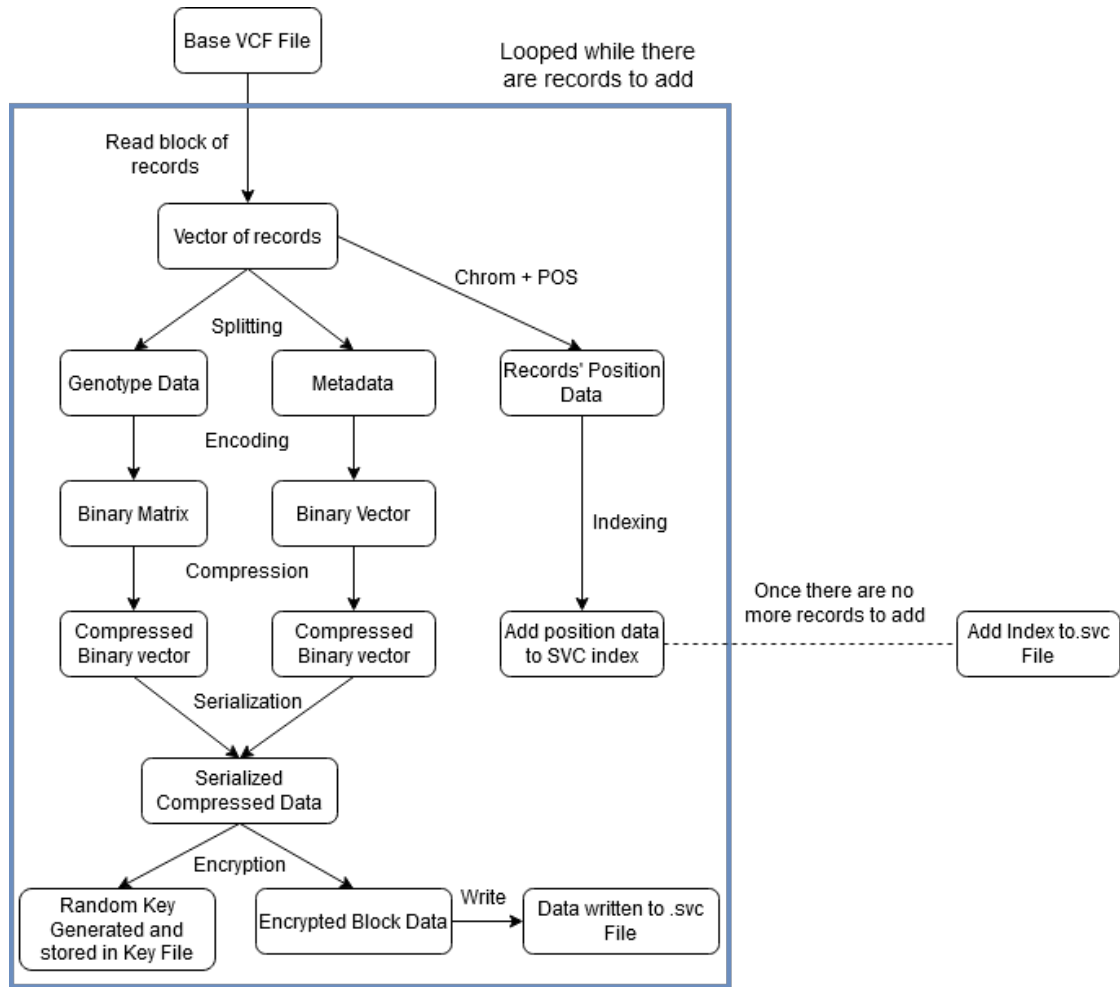


Figure 2.5: Simplified SVC Pipeline

- -b A target block size
1. The program then creates 1 input thread, to read the VCF file and generate the raw blocks, 1 output thread to write the final compressed and encrypted data to the SVC file and j process threads. These threads do most of the heavy work: encoding, sorting, compression, and encryption. Note that over the lifetime of the program, more than just $j + 2$ threads might be alive at once. This is because the input thread and the process threads will divide their work to sub-threads (e.g. for the hamming matrix computation of the processing threads).
 2. The index file is parsed, stored and the first region defined is set as current region of interest.

3. The input thread reads the records from the VCF file using the `htslib` [18] C Library. It reads as many records as necessary to reach the block size limit or until the last read record does not belong to the current region of interest, in which case the record will be set aside for the next block and the thread will move on to the next region of interest. All these records are then put together in a single block and pushed to a queue (the '`input_queue`') alongside some additional data; most notably, the ID of the section this block belongs to and the block's ID (every block get assigned a unique ID when it is created).
4. The process threads monitor the `input_queue` so that when a block is available for processing, it takes it out of the queue and starts working on it. The steps taken by these threads are as follow:
 - (a) (from GVC) Encode the genotype and phase data as binary matrices
 - (b) (from GVC) Sort these binary matrices
 - (c) (from GVC) Compress these sorted binary matrices using JBIG
 - (d) Compress the metadata of the records in the block using GZIP
 - (e) Serialize the different compressed structures into a single array of bytes
 - (f) Encrypt this array. The keys for the different sections are all pre-computed at the very beginning of the program (before the spawning of the different threads) and thus can simply be retrieved. Since the encryption algorithm used (AES-GCM) requires an Initial Vector (IV) and for security reason, no two blocks can be encrypted using the same key AND the same IV, we use the (unique) ID of the block as IV [2].
 - (g) Once all of the above steps are completed, the encrypted data is passed to the `output_queue` alongside the genomic interval the encoded data spans, as well as the block's ID
5. The output thread monitors the `output_queue` for blocks. When one is available, it writes the encrypted array to the `.svc` file. The class responsible for writing all the blocks' data to the disk also keeps in memory an *index* of where all the different blocks were stored in the file (i.e. the byte offset at which every block's data start in the `.svc` file, this index will be explained more thoroughly in Section 2.1.5). This index also records what genomic region this block spans and what the associated blocks' IDs are. Note that the block ID being only used as IV does not need to be secret and can be stored plainly [2]. This index is what later allows for random access.

6. The different threads can communicate when their job is done to each other (the input thread can tell the process thread that no more blocks are coming, same thing for the process thread to the output thread). Once all of the threads have finished, the main thread will do the encoding, compression and encryption of the VCF's header and sample list (with 2 different keys). This compression is also done using GZip. The main thread will then finalize the `.svc` file by adding the index at the end. It will also create a new file containing all the keys used.
7. Note: A small utility is also available in SVC's source code, that allows for simple retrieval, from the key file, of only a subset of the keys based on the section IDs (creating a new key file).

2.1.5 SVC to VCF

The encryption, compression and encoding steps simply need to be reversed, so that will not be explained in too much details, but some points in the process remain interesting and will be developed upon. When trying to decompress a SVC file, a few arguments can be given to the program.

- `-f` Should give the SVC file to work on
- `-o` Should give the name of the file to store the result to (the result will be a VCF file).
- `-c` Gives a start and end chromosome for random access (in the format `s:e`)
- `-p` Gives a start and end position for random access (in the format `s:e`)
- `-s` The name of a file containing the coma separated names of samples the user wishes to keep. NOTE: this is only a front-end filtering, as you might have realized from the previous section, there is no indexing over the samples or any special encoding that allows for a by-sample encryption or decryption. This means that, even with this option, the data of all the samples will be decrypted and appear in memory, they simply will not all appear in the resulting VCF file.
- `-k` The name of a file containing AT LEAST all the keys necessary to decrypt all the sections overlapping with the region described by the `-c` and `-p` parameters. If the key of any one of the overlapping section is missing, the program will terminate and inform the user that some keys are missing or wrong.

Random Access in SVC

Let's first look at how random access works in SVC. For this the most important parameters are `-c` and `-p` that together define a *genomic span*, we will denote the start and end chromosome defined by `-c` as C_s and C_e , and the positions as P_s and P_e . Let's imagine that $C_s = 1, C_e = 3, P_s = 1000, P_e = 3000$, then the region starts at position 1000 on the chromosome 1 and fully englobes it after that, it spans the whole of chromosome 2, and contains only positions 1 to 3000 on chromosome 3.

Note that while it is possible to have $C_s \neq C_e$ as just shown, it is not the expected and recommended way to do it:

1. It usually does not make a lot of sense when doing analysis to see different chromosomes as one continuous strand.
2. When `htslib` parses the VCF file, different chromosomes are associated to a numerical ID, while these IDs can be set in the header of the VCF file, it is not always done. Because SVC translates the chromosomes' names to their ID using `htslib` functions, there might be some unexpected behaviours if the user did not specify the chromosomes' IDs themselves and the (numerical) order of the automatic ones don't match the order the user expects. If the order of the ID's doesn't match what the user has in mind.

This technical feature's main goal is for entire chromosomes retrieval. In that case the `-p` parameter should be dropped, and if the whole file is needed, then the `-c` parameter should be dropped as well.

The next item of interest for random access in the *block index* that exists at the end of the file. It is obviously stored in simple format but it can be represented in a table (as for example in Table 2.2, there is one line per block).

Chr.Start	Chr.End	Pos.Start	Pos.End	File Offset	Block ID	Section ID
1	1	1	2000	33000	0	0
1	1	2033	5904	0	1	0
1	1	6049	18567	16330	2	0
1	1	555009	559005	75378	3	1
2	2	3045	8765	51049	4	2
2	2	8766	15643	91113	5	2

Table 2.2: Example of a simple *block index*

The Chr.Start/End and Pos.Start/End come from the records contained in the block. Note that, here, the chromosomes are actually the chromosome IDs, not their name. The Block ID is assigned to the block in the input thread and is simply

incremented at every block read. From Galois Counter Mode (GCM) [2] we know that the IV used in the encryption of AES-GCM needs only be unique, it being predictable does not cause any security issue, this method is thus acceptable. The file offset represents at which byte the data of the block starts in the `.svc` file. In this index, we can see that the blocks are not stored in the same order as they were read, this can happen because of the multi-threading. Some other things that can be observed on this table:

1. No region (Chr.Stt/End + Pos.Stt/End) overlap, this should never happen since no two records in the VCF file should have the same CHROM POS combination.
2. Although all the blocks should have the same number of records (except the last one in each section, that can be smaller), this doesn't mean that the blocks span the same amount of bases since adjacent records can have very close/distant positions.
3. The table is sorted according to the regions' chromosome and position. With the current block ID generation and because records are read sequentially, this will be the same order as the block ID.

This sorting happens at the very end, once all the blocks have been added to the index and is very important. With the index already sorted, it is easy to find the blocks needed to cover a random access query. We can simply run 2 binary searches to find the first and last block overlapping with the C_s, C_e, P_s, P_e span, and then iterate over this reduced list since all the blocks in-between will also overlap with the span.

If the list was in a somewhat random order (as it could be because of the multi-threading) then retrieving the list of blocks to decrypt for a certain random access region would be quite cumbersome: It would be necessary to traverse the whole list to find all the blocks belonging to the region, and then sort it to have the order in which the blocks need to be processed.

2.1.6 SVC file format

The file format developed for SVC will be presented in a top-down manner, with a high-level overview first making use of smaller structures that will be defined afterwards.

The highest level overview of the file format can be seen in Figure 2.6:

The 3 offset fields are used to retrieve the encrypted header, sample list and index in the file. The positions of the different encrypted block can all be found in the

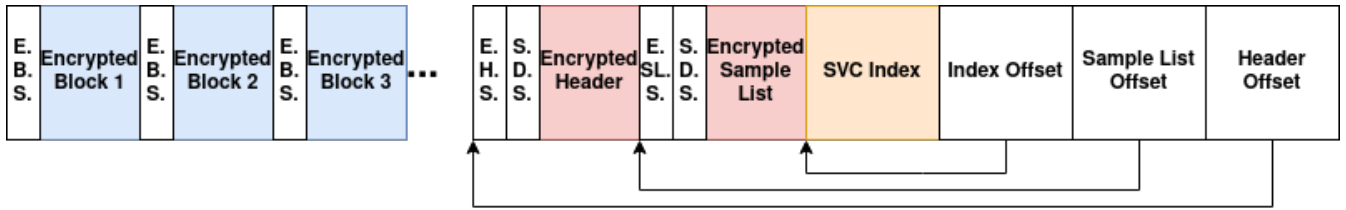


Figure 2.6: Overall SVC File Format

index. The index has no special format once in the file, it is simply a serialized version of the table format explained above. The **E.B.S.**, **E.H.S.** and **E.S.L.S.** are simply fields giving the size (in bytes) of the encrypted structure following them (**Encrypted Block Size**, **Encrypted Header Size**, **Encrypted Sample List Size**). The 2 **S.D.S.** blocks represent the *Serialized Data Size* of the block following them, that is the memory size they were taking in uncompressed form, this is an information GZip needs in order to decompress the data. The *Encrypted Sample List* is simply all the sample names (raw strings) serialized, then GZipped and encrypted.

The serialized header is represented in Figure 2.7, the different fields and values are not of major importance and are linked to htlib and the VCF format. Some of them (mainly the ints representing array sizes) are important for deserialization.

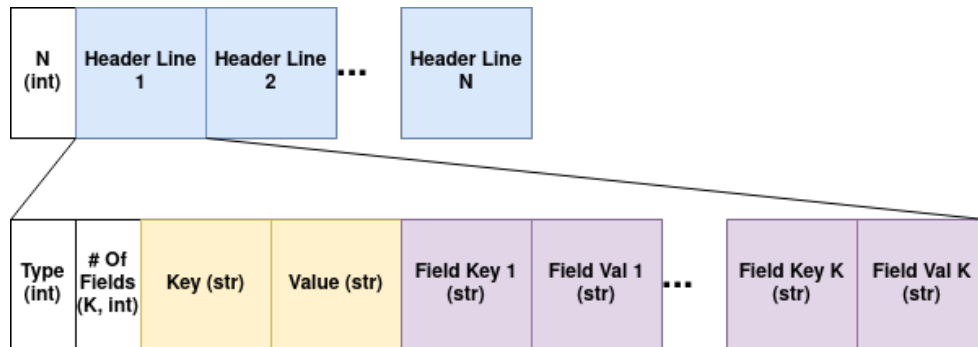


Figure 2.7: Serialized Header Format

The format of the blocks before encryption can be seen on Figure 2.8, here follows a detailed explanation of the different fields:

1. **Block Size:** The number of records in the block
2. **S.D.S.:** The uncompressed size (in bytes) of the record's fields
3. **G.T. Bit Matrix Rows:** Number of rows in the binarized version of the genotype matrix.

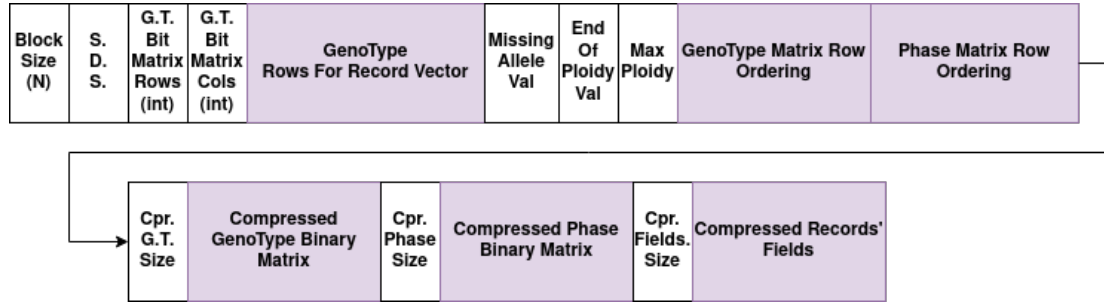


Figure 2.8: Format Of Blocks Before Encryption

4. **G.T. Bit Matrix Cols:** Number of columns in the binarized version of the genotype matrix, depends on the number of samples and the maximum ploidy found in the block.
5. **GenoType Rows For Record Vector:** The *Rows for Record* vector associated with the genotype binary matrix as explained in Section 2.1.1.
6. **Missing Allele val:** The special genotype value associated with a missing allele, as explained in Section 2.1.1.
7. **End of Ploidy val:** The special genotype value associated with an early end of ploidy, as explained in Section 2.1.1.
8. **Max Ploidy:** The maximum ploidy found in the block (currently, it is expected that all records, for all samples have all the same ploidy, but this might be useful in the future).
9. **Genotype Matrix Row Ordering:** The changed row order, resulting from the sorting. Used to undo the sorting operation. The V vector presented in Section 2.1.1 for the allele matrix.
10. **Phase Matrix Row Ordering:** The changed row order, resulting from the sorting. Used to undo the sorting operation. The V vector presented in Section 2.1.1 for the phase matrix.
11. **Cpr. G.T. size:** Size, in bytes, of the compressed genotype binary matrix.
12. **Compressed Genotype Binary Matrix:** JBIG compressed genotype binary matrix.
13. **Cpr. Phase size:** Size, in bytes, of the compressed phase binary matrix.
14. **Compressed Genotype Binary Matrix:** JBIG compressed phase binary matrix.

15. **Cpr. Fields size:** Size, in bytes, of the compressed fields data.
16. **Compressed Records' fields:** GZip compressed records metadata.

The metadata, before being compressed is simply serialized column-wise (this was chosen arbitrarily, perhaps row-wise would result in better compression, this is to be tested in future works). As of now, SVC only supports the standards VCF columns (CHROM, POS, REF, ALT, FILTER, QUAL, INFO and FORMAT).

The format of the keyfile generated by SVC can be found in Figure 2.9. This file is very simple, and parsing is quite easy as the **Expected Key Size** is always 4 bytes. All **Key Section IDs** are all 4 bytes as well, and every key should be \$(**Expected Key Size**) bytes. When the SVC program is ran in "SVC to VCF" mode, the class responsible of handling keys will parse this whole file and put every <Section ID, Key> pair in a map that can be accessed later.



Figure 2.9: SVC Key file format

2.2 Security

For the implementation of important security functions, we relied on LibSodium [7], which underwent a security audit by Cryptography Engineering LLC in 2017, funded by PIA [13].

By default, encryption in SVC is done using AES-GCM [2], a very popular AEAD (Authenticated Encryption with Additional Data). Its popularity comes from its usage in TLS and its speed. AEAD constructs are useful because of the additional (non-confidential) data they are capable of compiling into the authentication tag. Since this additional data is currently not used by SVC, it is possible to use another algorithm which is an authenticated encryption scheme but without additional data. This algorithm is called **SecretBox** in LibSodium, the encryption is done using XSalsa20 [5] (an adapted version of Salsa20 that uses longer nonces) and the authentication is done using Poly1305 MAC [3]. The usage of this algorithm is already implemented and can be chosen instead of AES-GCM by setting SVC's **-x** parameter.

There is a page on the LibSodium website ¹ which gives some warnings about the

¹https://libsodium.gitbook.io/doc/secret-key_cryptography/aead/aes-256-gcm

usage of AES-GCM, some of which do apply to SVC.

- If the same key is used multiple times, to encrypt different messages, there is a limit on how much data can be encrypted securely. For example, if messages are 16KB then no more than 350GB of data should be encrypted with a single key, note that this limit depends on the message size but this 350GB limit is usually used as a rule of thumb. It is possible that SVC only uses a single key for the encryption of the whole file (if the *region index* only defines a single region for the whole file). In that case, it would still require for the **compressed** VCF data to take up more than 350GB before problems arise. While this is unlikely, it is still possible. In the case this happens there either needs to be a split in the index file, create more than one region and thus more than one key, or simply use another AEAD.
- As said previously, two messages should **never** be encrypted using the same key **and** the same nonce. With AES-GCM, the nonces are quite short (12 bytes) from a cryptographic point of view. It is very discouraged, when dealing with "short" nonces, to use random generation.

It is instead recommended to use atomic counters, which SVC currently does. Its currently very simply implemented since the nonces for each block are generated alongside the block, in the `input_thread`, but if SVC is ever re-written to support multiple input threads, or even work in a distributed environment, then special care will have to be given to nonce generation.

The two limitations explained above, could be avoided by using AEGIS-256, another AEAD. The reason SVC uses AES-GCM by default instead of AEGIS-256 is simply because more people know of, and trust, AES-GCM, making SVC's adoption easier.

Since encryption methods in SVC are contained into a Factory class, it is very easy to implement new schemes. As of writing this, SVC currently supports 4 schemes that can be chosen via the `-x` parameter:

1. NONE: No encryption will be done and SVC will simply work as a compression tool
2. AES-GCM (default): The well know AES encryption scheme with Galois counter mode for authentication. LibSodium's implementation requires Intel AES-NI extensions or ARM crypto extension. If the platform on which SVC runs does not have either of those, it will inform the user that they cannot use AES-GCM, there is no such limitations with other constructs.
3. AEGIS-256: Another AEAD without the above-mentioned limitations, overall a better alternative to AES-GCM. Note that with LibSodium's implemen-

tation (the one used in SVC), if the code is ran on a platform without hardware AES support, the computation is much slower and is vulnerable to side-channel attacks. This algorithm was made available in a stable version of LibSodium only very recently (stable version 1.0.19, which was released on the 9th of May 2024)

4. SECRET BOX: An authenticated encryption scheme without additional data but that does not suffer from the above-mentioned limitations and runs similarly well on all machines.

2.3 Example Usage

Let's showcase how SVC can be used to only give limited access to some users. Imagine someone in possession of a VCF file containing data about chromosome 21, with positions ranging from 9,000,000 to 11,000,000. This person wants some people to only have access to the first half of the position, and some other people to only have access to the second half. This person could then setup a region index file like in Table 2.3, which creates two regions corresponding to these limited access bounds. Note that the RegionID column is not necessary for SVC, but here are interesting because they show the IDs necessary to isolate the regions' keys later, SVC will always create region IDs starting from 0 and incrementing sequentially.

RegionID	chrom	start_pos	end_pos
0	21	9,000,000	10,000,000
1	21	10,000,001	11,000,000

Table 2.3: Example region index file, cutting the file into two regions

With this, the user is ready to make use of SVC. After downloading the source code, they can simply compile it and run the `svc_exec` executable, which is the compression/encryption and decryption/decompression utility:

```
> make svc
> ./svc_exec -f chr21.vcf -i index.csv -o archive21.svc -k keys21.keys -j 4
```

This command will execute SVC, tell it to compress the VCF file (`chr21.vcf`) based on the regions defined in the index file (`index.csv`) and output the archive into `archive21.svc` and the keys into `keys21.keys`, as well as use 4 processing threads.

Once this is done, the user can then compile the key utility, whose source code is bundled into SVC's source code, and use it to create two different key files for the

different types of access they wish to grant. Note that the key utility expects the IDs of the regions whose keys need to be isolate in the standard input stream (one per line). Hence why they are simple echoed and then piped into the program.

```
> make keyhandler
>
> # Echo the ID of the first region into the key utility
> # to create access for the first half
> echo "0" | ./svc_keys -f keys21.keys -o access1.keys
>
> # Echo the ID of the second region into the key utility
> # to create access for the second half
> echo "1" | ./svc_keys -f keys21.keys -o access2.keys
```

This creates two new key files that can be shared alongside the archive, but which only allow limited access. For someone who received only the **access1.keys** key file, trying to query positions outside of the 0-th region, e.g. from position 9,000,000 to 10,050,000 which overlaps with region 1, will result in an error since the **access1.keys** file does not have the key necessary to decrypt position 10,000,0001 to 10,050,000. The resulting VCF file will still contain the decrypted and decompressed data of all the regions that appear before the first faulty one.

In the scenario explained just above, the user tries to decrypt a region for which, in their key file, there is no key corresponding to the region's ID, thus there isn't even an attempt at decryption. If a key is found but is the wrong one, then decryption will be attempted but will fail, also resulting in an error and the program will stop. This will also happen when, if using one of the authenticated encryption scheme provided, the part of archive corresponding to the region being decrypted has been tempered with or corrupted.

2.4 Performances

2.4.1 Losslessness

While the design of SVC is that of a lossless compression algorithm (both GVC and GZip being lossless), let's show that the implementation indeed does work that way.

Sometimes the losslessness of SVC is obvious, when looking at the files obtained when compressing (and encrypting) then decompressing (and decrypting) for example data for chromosome 21, one might see the following (actual values obtained when compressing with **-b 1024** and a "Single Region" index).

SIZE (B)	NAME
11232432537	original_chr21.vcf
54419434	compressed_chr21.svc
148	chr21.keys
11232432537	retrieved_chr21.vcf

The original file and retrieved one being the same size is already a good indicator, but running the `diff` utility on those two files does show that there are zero differences between them.

It is some other times less obvious. Let's imagine that someone compressed and encrypted chromosome 22 from 1000gp phase 1, with a lot of format fields ². Then, at some point, the file is decrypted and uncompressed in its entirety, in this case one might see the following when inspecting the original and result file.

SIZE (B)	NAME
15304146564	original_chr22p1.vcf
11042658362	retrieved_chr22p1.vcf

The sizes don't match. Despite the first dreary looks, and although the file is indeed altered, no information is actually lost. This file size difference of about 35% comes from the fact that some floating values (appearing as text in VCF) might have trailing zeroes cut out in the retrieved version. When taking a random record from the original VCF file and comparing with the equivalent record in the retrieved VCF, one might see the following when looking at some sample's values (these values come from the sixth sample at POS=45830721). Note that the spaces were added to make the values appear more clearly:

```
<Original >: 1|0:0.950:-4.70,-0.00,-2.77
<Retrieved>: 1|0:0.95 :-4.7 , -0 , -2.77
```

All the values are strictly equal: no information is lost. Comparing the total size from the two records, we can see that the one from the original file is 31087 characters long, while in the retrieved version it is 22533 characters long. We see this same 35% reduction of size. Checking the value of many samples at different position and seeing the same scenario every time gives good indication that this is indeed the only kind of alteration in the file.

2.4.2 SVC's performance depending on its parameters

By its design, SVC's performances depend on many factors:

²From <http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/>

- Target block size (from `-b` argument): Higher should yield better compression, but should make the program slower (which is currently not the case, more on this later) and consume more memory.
- Number of samples in the VCF file: With a higher number of samples, the ratio between the amount of memory taken by genotype information and the amount of memory needed for the rest increases in favor of the genotype information. because currently SVC uses GVC for compression of genotype data and only GZIP for the rest, this should result in better compression. This will however make the size of blocks in active memory larger, meaning that more RAM would be necessary.
- Number of records / File size: While this has effects on the runtime (linearly) it shouldn't impact the compression ratio since a file is always divided into independent blocks.
- Regions defined in the index file: This can, theoretically, have a very large impact on the time and compression performance of the algorithm, if the region are poorly defined and blocks only contain a few records each time, this means that many more blocks will be generated than necessary, although each block will be processed much faster.
- Number of "Format" fields: (Similar argument to the number of samples) When the amount of format fields increases, the ratio of genotype data vs other lowers drastically, meaning the compression ratio will tank lower and lower the more fields are added. In these kinds of scenarios, SVC should reach, at worst, similar compression to GZip.

For these performance tests we have decided to mostly use data from 2 sets of file from the 1000 genome project. Each being a collection of 24 files (one for each chromosomes) with data for 2500 samples, because this is the dataset used by most of the algorithms described in the introduction. The difference between the two sets is that in the "simple" set, on all records, the value for the sample is simply limited to the genotype, while in "annotated" set there are a lot of format fields, so a (much) lower compression ratio is expected.

Note that on **ALL** tests, while sometime only the compression step is mentioned (because it is the most computationally expensive one), the encryption step is **ALWAYS** done. And more precisely, it is always AES-GCM.

During testing, the two different sets of files resided on different drives, with mostly similar performances, you can find details about those drive as well as the result of simple benchmarks in Section A.4.4.

The tests were also done using 2 types of index files:

1. "Single Region" index files: These indexes consider their associated VCF files as a single region, meaning that the file will always be divided up into blocks of maximum size (from `-b`) and the whole content will be encrypted using a single key, meaning limited security but better compression.
2. "Padded Genes" index file: These index files are used to show a possible use case of SVC. The regions defined in those files come from NIH's *Gene* database [24] which supplies a list of genes that can be found in humans (a gene being a cluster of alleles that together define a human characteristic, and where a variant can sometimes be linked to a disease). Additionally, regions have been added between each gene so that the whole VCF file can be retrieved afterwards. This type of index file allows owners of an SVC archive to only give the key to a single gene to a person that might need it, without compromising the rest of the data.

For the sake of time, the tests for computing resources show values obtained from testing only on a non-annotated (GT as only format field) version of chromosome 21³. The values are similar for other chromosomes.

Let's first look at how the computing resources evolve when running SVC with a growing number of process threads (`-j`). See Figure 2.10, showing results obtained when using "Single Region" index files.

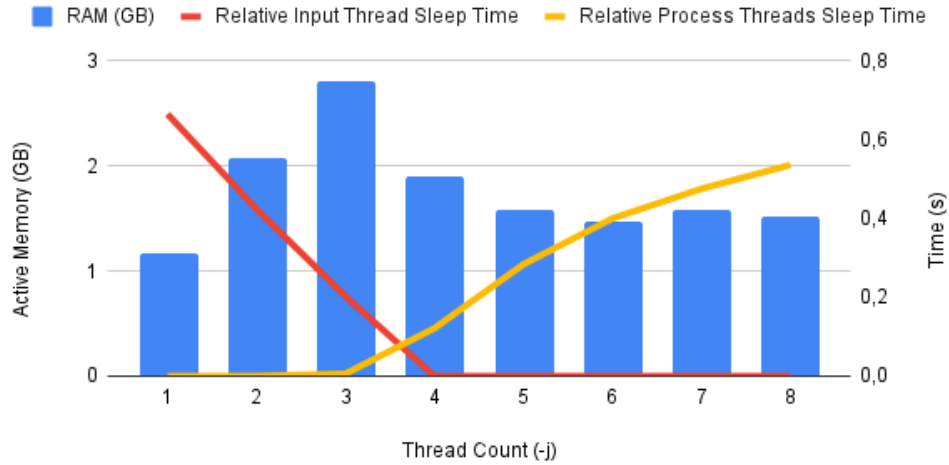
One of the things one could expect is to see the amount of active memory necessary increase linearly with the number of threads (that will now be called j). Indeed, at any one point in time, the program will only keep a maximum of $2 * j + 2$ blocks of records in memory (j being processed, a maximum of $j + 1$ waiting in the `input_queue` and 1 being read from the VCF file). In Figure 2.10 we can see that this stands true only up to 3 threads. At 4 threads and beyond, it takes more time to read a block than to process one (this value of 4 threads is only true for blocks of size 1024, different block sizes would hit this limit at a different number of threads). This means that the $2 * j + 2$ potential blocks are never alive at the same time, reducing the total memory necessary. This is a great waste of performance, as can be seen from the red and yellow lines in the graphs. As more process threads are being added, they all start spending more and more of their time sleeping. This shows that, as of now, the throughput of SVC is limited by the input step, this plateau can clearly be seen in the second graph of the figure. Ideas on how to improve this in the future are given in Section 2.5.

In Figure 2.11, we can see the same test being done as in Figure 2.10 but this time

³From <http://ftp.1000genomes.ebi.ac.uk/vol11/ftp/release/20130502/>

Effect of thread count on SVC's RAM usage

Using "Single Region" index file



Effect of thread count on SVC Run time

Using "Single Region" index file

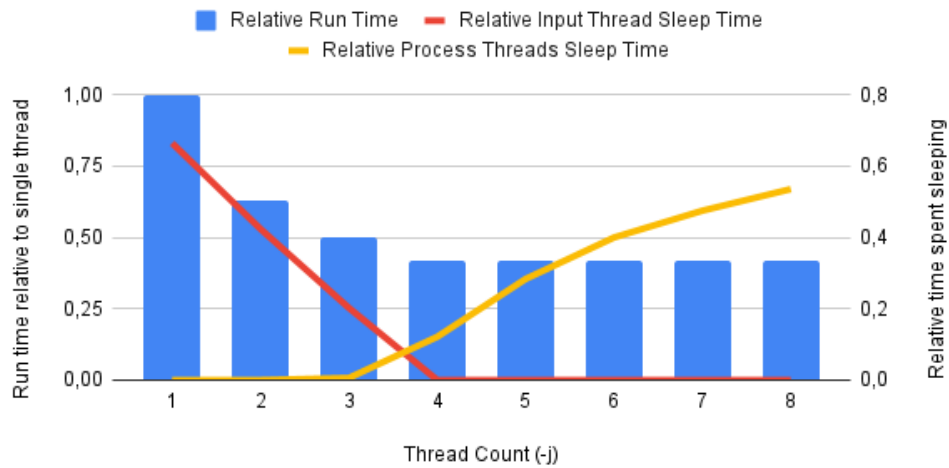
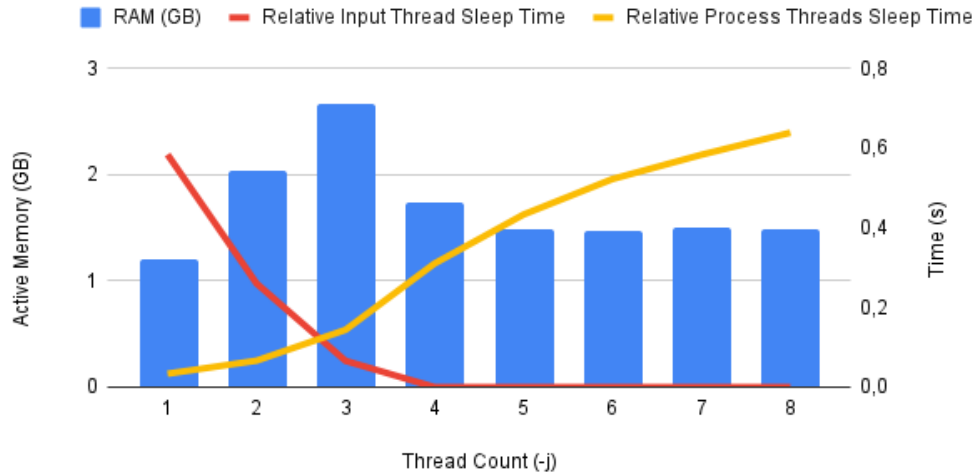


Figure 2.10: SVC computation cost in relation to thread count when using a "Single Region" index file. The target block size ('-b') is set to 1024.

using "Padded Genes" index files. The results are, as expected, similar between the two. One of the main differences between those two scenario (beyond those linked to security and key generation/usage) is that blocks are generated with more variable sizes when using "Padded Genes" indexes. A region defined in the index

Effect of thread count on SVC's RAM usage

Using "Padded Genes" index file



Effect of thread count on SVC Run time

Using "Padded Genes" index file

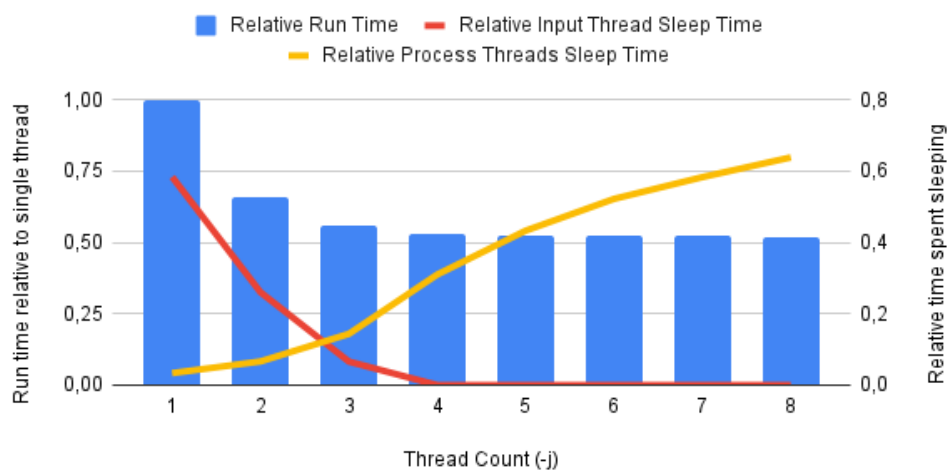


Figure 2.11: SVC computation cost in relation to thread count when using a "Padded Genes" index file. The target block size ('-b') is set to 1024.

file might create a block containing just a few records, even if the target block size is defined to be much higher, and regions rarely contain a number of records being a multiple of this target block size. This, potentially frequent depending on the number of regions, creation of smaller blocks, means that more blocks need to be

created to cover the whole file, but those blocks being smaller are also processed faster in the process threads. This leads to less clear cut and consistent results. For example, we can see that even with a single or two process threads, they will spend some time sleeping (unlike when using Padded Genes indexes). This can happen if a few very small blocks are created back to back, they will be processed very quickly and so the process threads will empty the `input_queue` very quickly and will have to wait for the input thread to fill it back up, sleeping in the meantime.

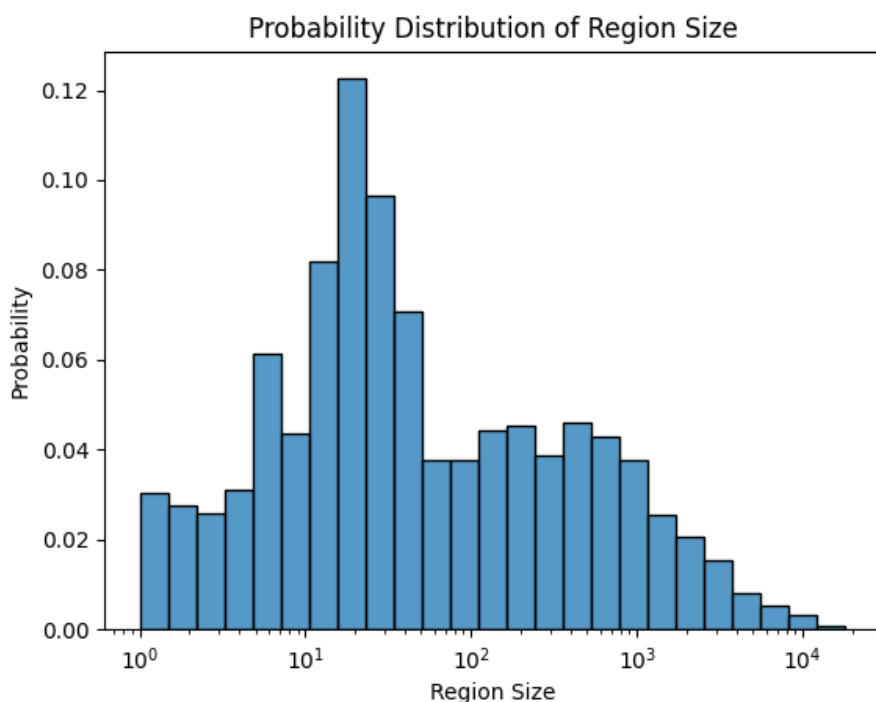


Figure 2.12: Region size probability. From "Padded Genes" index file created for chromosome 21.

We can see an example of distribution of the size of the regions in Figure 2.12. Note that this distribution is specific to our VCF file and our index, many different distribution are possible, but this shows the major difference there can be between single region indexes and more complex indexes. In this case, as we can see, most of the regions span less than 1024 records. This is why the behaviour with this kind of index file is more erratic: most blocks are of different sizes. To follow-up on this, let's look at different SVC performance metrics when varying the target size of blocks.

Figure 2.13 shows the relation between the target block size parameter and the resulting peak memory usage of SVC. For Single Gene we can clearly see the linear

Active Memory in relation to Block Size

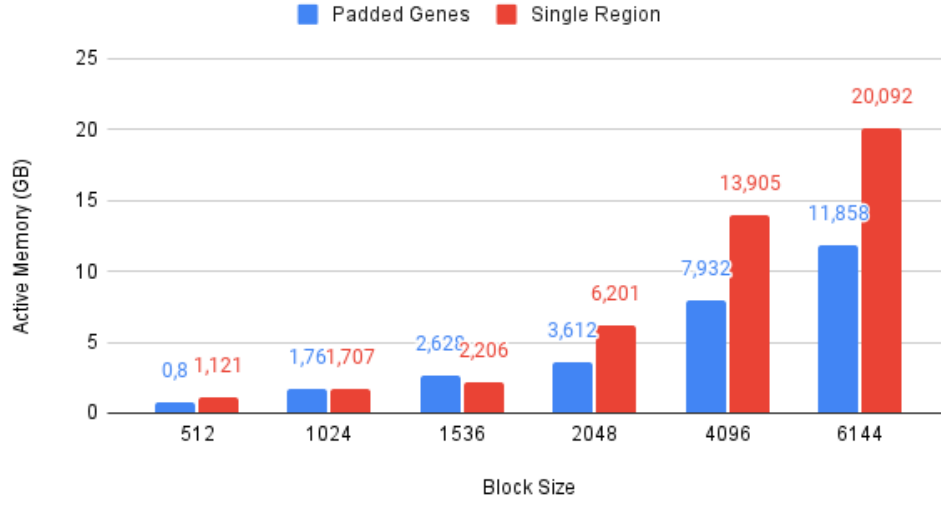


Figure 2.13: Peak Amount of Active Memory (RAM) necessary to execute SVC on a file with 2500 samples depending on the target block size ('-b'). Number of process threads is 4.

relationship between the two, especially at high block sizes. This linearity even stands true for Padded Genes despite the irregular block size it has to deal with. This shows a clear limitation: even though the amount of memory necessary can be reduced by adjusting for example the in-between-threads queue size, the amount of memory will always grow linearly with the block size. This means that if someone is uninterested in the random access capability of SVC (which depends on this block-wise division of the file), it is still not possible to consider the file as one very big block to squeeze out the best possible compression. Most likely the machine will not have the necessary memory for it.

As the size of the blocks increases, the performance bottleneck shifts from the input operations, to the processing of the blocks (This is later seen and discussed via Figure 2.17). We can clearly see this in Figure 2.14, the time SVC takes to process the file, when using a Single Gene index, stays somewhat constant for block sizes [512; 2048]. Beyond that point we see that the required time starts to increase drastically. It could be expected, from a complexity analysis, to see a polynomial time increase with the block size since the worst complexity (hamming distance matrix computation) is $\mathcal{O}(n^2m)$, with n being proportional to $-b$, and m being the number of samples. The growth however looks very linear, which probably indicates that, once again thanks to the optimizations, the hamming computation

Compression Time in relation to Block Size

All run times are relative to the one with single region and block size = 512

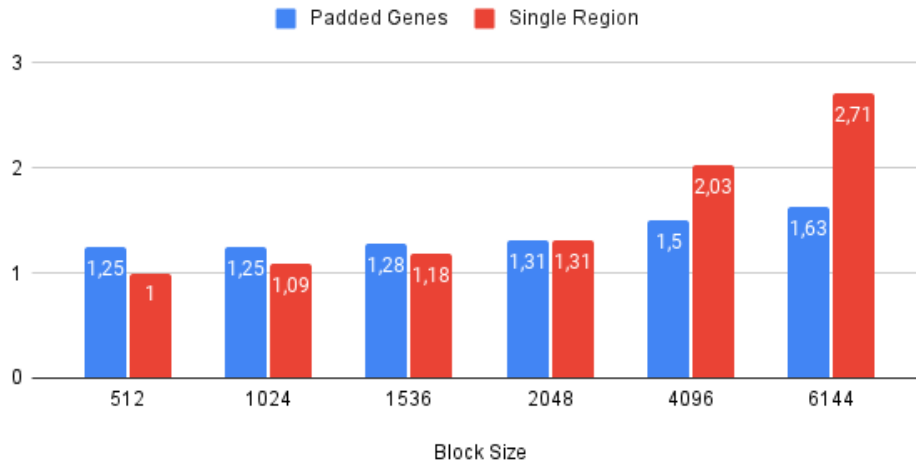


Figure 2.14: Amount of time necessary to execute SVC on a file with 2500 samples depending on the target block size('b') relative to when that size is set to 512 using a "Single Region" index. Number of threads is 4

is fast enough and the slowest operation is actually another one. A deeper analysis of the performances of the different operation in the process threads would be necessary to go further about this.

Figure 2.14 also shows that, for Padded Genes index, the time growth is much lower than for a Single Region index. This is expected, since the average block size is actually much smaller than the given target block size.

Figure 2.15 shows the time necessary to decompress a SVC file, depending on its compression parameter. Since there is no special computation when going from a SVC archive back to a VCF file, it would be expected for the total time necessary to be proportional to the number of records. And so we would expect about constant run time, which we do see when using a Single Region index. The run time is however much more erratic when using Padded Genes, which is really unexpected and for which there is currently no good explanation. A deeper analysis would be necessary to understand what is happening.

From Figure 2.16 it can be seen that increasing the block size provides better compression for both Single Region and Padded Genes indexes. While the compression ratio is consistently better for Single Region, it is not so by a large margin. We can see that the compression ratio growth flattens out for high block sizes, this is

Decompression Time in relation to Block Size

All run times are relative to the one with single region and block size = 512

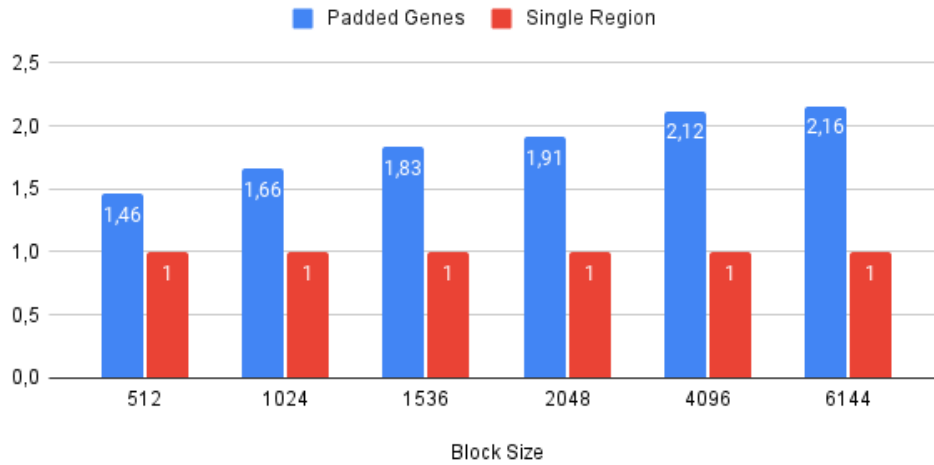


Figure 2.15: Amount of time necessary to decrypt and decompress a SVC file depending on the target block size ('-b') and the type of region index file, relative to when that size is set to 512 and using a "Single Region" index.

Compression Ratio in relation to Block Size

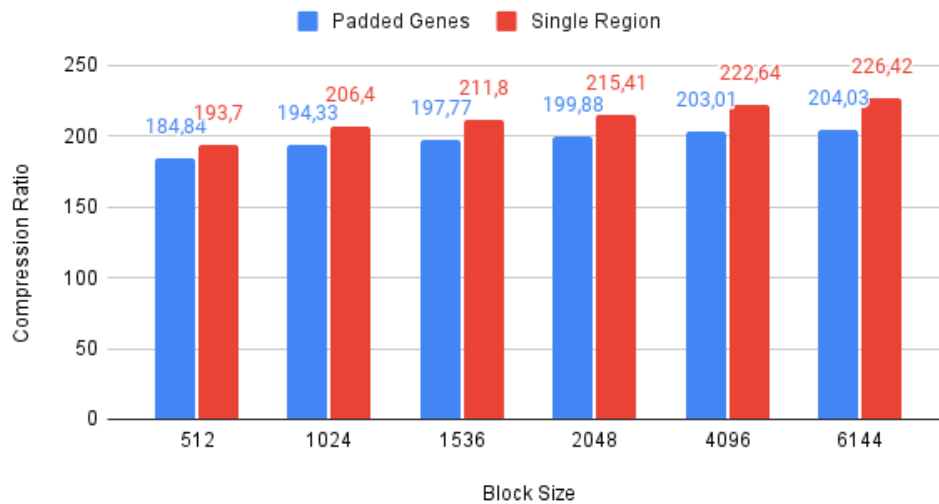


Figure 2.16: Compression Ratio achieved by SVC on a file with 2500 samples depending on the target block size ('-b')

expected since, as was shown in Figure 2.12, there are not a lot of blocks with a high number of records.

Combining the information from Figure 2.14 and 2.16, we can see that a great usage of SVC could be using a Padded Gene index with a high value for the `-b` parameter (about 4096). The P.G. index would enable the usage of SVC's security capabilities to its fullest extent and its inconsistent block size (ideally, the average size should be quite small, while still having some large contiguous regions) would allow for a great balance between speed and compression.

Threads Sleep Time in relation to Block Size

Sleep time for every thread is relative to total time alive

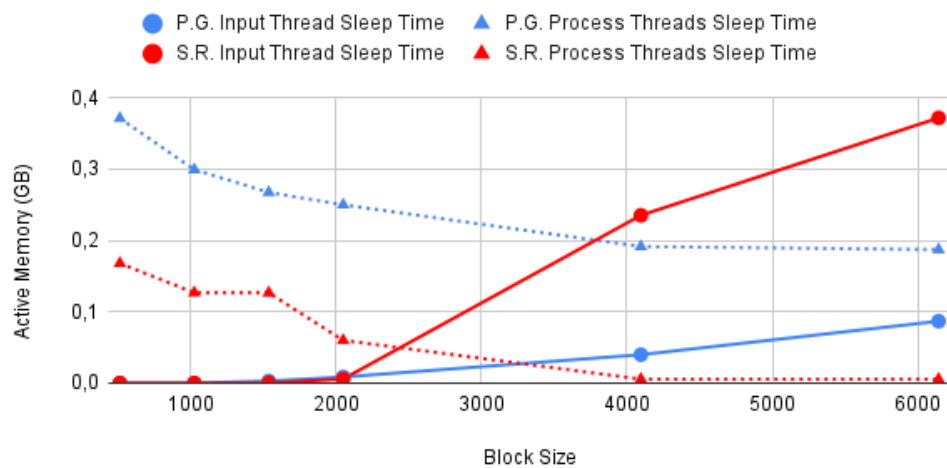


Figure 2.17: Time spent idling by the different threads when executing SVC on a file with 2500 samples depending on the target block size ('-b'). Number of threads is 4

Figure 2.17 shows the shifting performance bottleneck as the block size increases. With small block sizes, we see that the input thread never sleeps, while the process thread spend a non-negligible amount of time Idle. This is because, for small blocks, their processing is much faster than their creation. At larger blocks sizes, this trend is completely reversed, process threads sleep much less and the input thread starts sleeping more and more. This effect is much more pronounced with the S.R. index since all blocks have the (large) target block sizes. Meanwhile, a lot of blocks, with the P.G. index are still quite small (ref Figure 2.12).

Compression Time in relation to File Size

Simple data from 1000gp

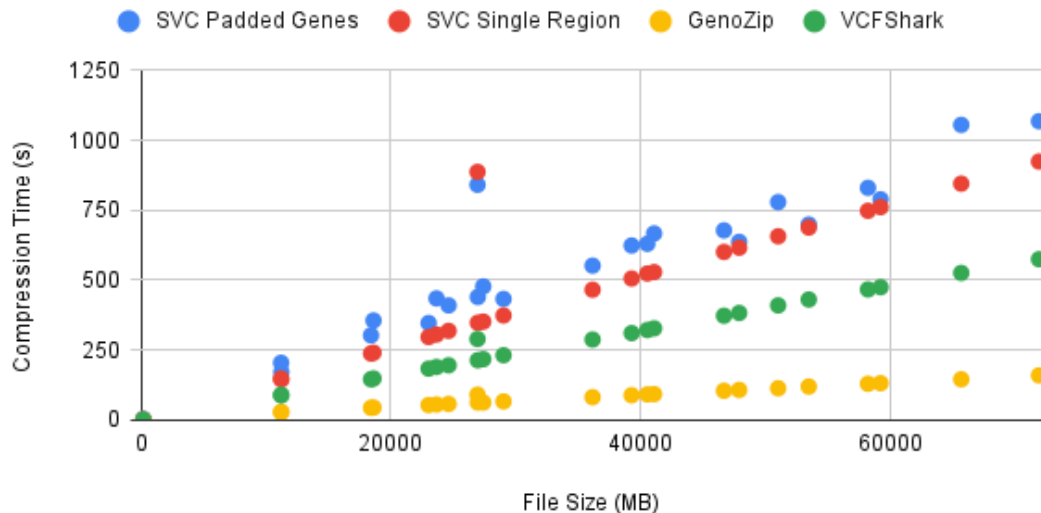


Figure 2.18: SVC Compression Time ('-b 1024 -j 4') compared to other solutions on a simple (non-annotated) dataset from the 1000gp

2.4.3 SVC compared to other solutions

Figures 2.18, 2.19 and 2.20 show the performance differences between SVC (Padded Genes and Single Region indexes), GenoZip, VCFShark and GZip when running on "simple" VCFs from the 1000 Genome Project. By simple, it is meant VCF file where, for each sample, there is only the genotype information, no other format fields.

We can see that SVC boasts about 4x better compression than the de-facto solution of GZip, but lacks behind, by a factor of about 3, VCFShark and Genozip when having to compress the whole files, ideas on how to improve SVC's compression can be found in Section 2.5.3.

We also see that both for compression and decompression, despite being 2 to 4 times slower than VCFShark or Genozip, it is still very much in the realm of reasonable run time, it is far from being prohibitively slow. And, as was already discussed, focusing on improving the input speed of SVC could probably bridge that gap.

For heavily annotated data, where there are a lot of format fields, and so where most of the size of the file does not come from genotype information anymore, we obtain the results in Table 2.4. Note that for heavily annotated data, the file

Decompression Time in relation to File Size

Simple data from 1000gp

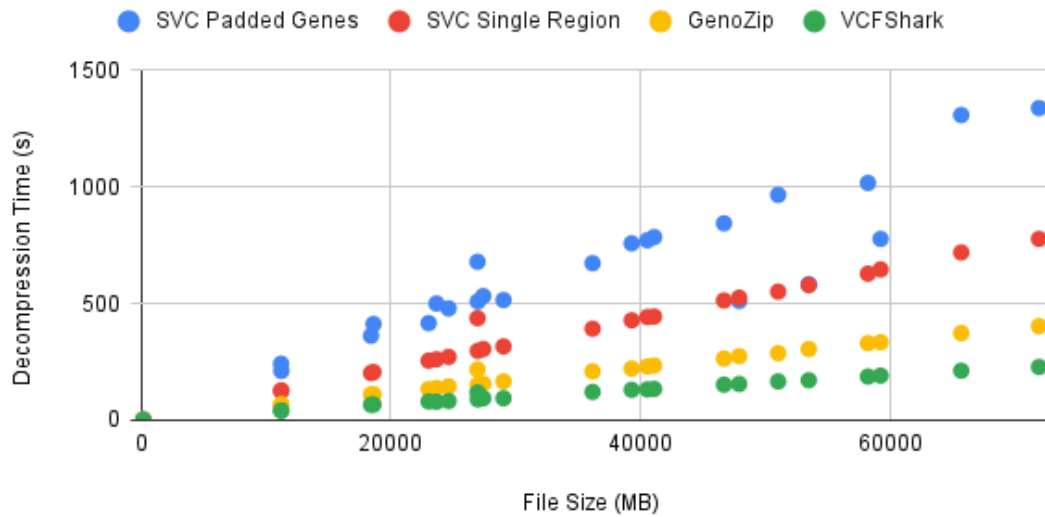


Figure 2.19: SVC Decompression Time ('-b 1024') compared to other solutions on a simple (non-annotated) dataset from the 1000gp

Compression Ratio by Chromosome

Simple data from 1000gp

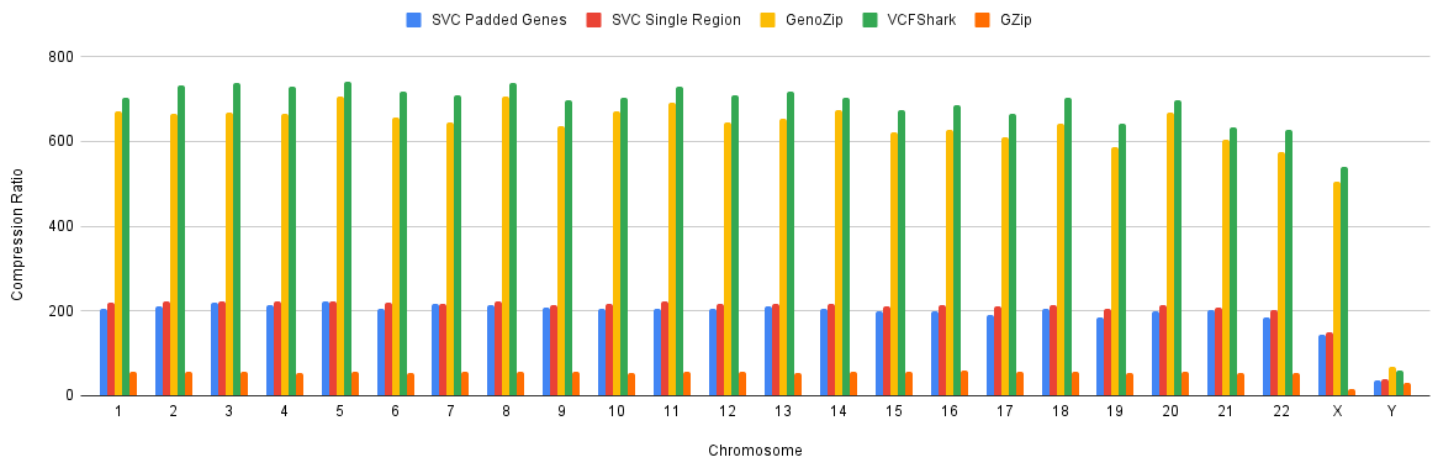


Figure 2.20: SVC Compression Ratio ('-b 1024') compared to other solutions on a simple (non-annotated) dataset from the 1000gp

sizes became so large (The largest file, for chromosome 1, alone is about 150GB) that thorough testing became impractical. The values in the table are averages from runs on the 5 smallest files (chromosomes 18 to 22) which together weigh a little bit more than 116GB. As we can see, SVC's compression is barely better than that of GZip, which is expected since in this most of the file gets compressed using GZip in SVC, this is one of its biggest flaw and ideas to improve this are given in Section 2.5.3. Despite just mostly using GZip in this case, it still shows better throughput when going from the VCF file to the archive than the raw GZip, because the UNIX utility used (**bgzip**) is single threaded, while with SVC, multiple threads are working. For the inverse process however, since the decryption of SVC is currently single threaded, and there are additional overheads to just GZip, it is comparatively slower.

	GenoZip	VCFShark	GZip	SVC S.R.
Compression Ratio	22.9	22.1	8.1	8.9
From VCF Throughput (MB/s)	295	62	49	141
To VCF Throughput (MB/s)	322	50	362	150

Table 2.4: SVC and other algorithm processing heavily annotated VCF files

Figure 2.21 shows a comparison of the random access performances of SVC and GenoZip. As can be seen, SVC is much quicker at retrieving a small amount of positions ($<$ a million), but GenoZip achieving constant time over the tested query ranges shows better performances for large requests. Since the average span of a human gene is estimated to be around 27 thousand bases [8] (i.e. positions), SVC is a much better alternative for someone for a per-gene querying service.

Random Access Time for a certain number of Positions

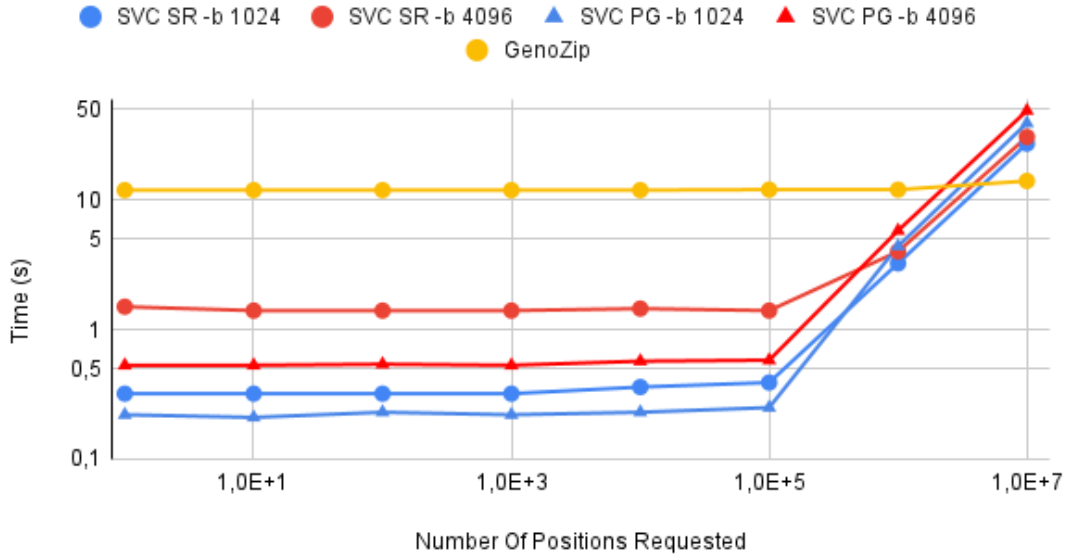


Figure 2.21: Comparison of the time taken to retrieve a certain number of **positions** from the archive for SVC and GenoZip. The number of position is not equal to the number of records in the original VCF file, but should be proportional.

2.5 Future improvements

2.5.1 Security

As of now, we know of 1 potential side-channel attack that can happen in some special cases. It is often forgotten that the size of the encrypted information can be valuable information by itself, and that a lot of other information can be deduced from this.

For example, in the famous case of Google Maps, researchers were able to retrieve the location of what someone was looking at by analyzing the SSL traffic. Since Google Maps is divided into tiles, each being a png of slightly different size, obtaining the encrypted size of those tiles gave good information on the actual size of the png and a reverse indexing would allow someone to retrieve these tiles' location from these sizes [6].

In SVC's case, since the encrypted data is already compressed, it is quite hard to link the compressed size to actual information from the block. But in the special

case were all/most of the genotype values are equal to 0, the Row-Encoded version of the allele matrix will simply be empty (see Section 2.1.1) since rows with only 0s are not added to the matrix. because of this, when the matrix is compressed and encrypted, it will be visibly smaller than other non-empty matrices. A bad actor could thus deduce the genotype value of the samples for this specific block.

To give the opportunity to the users of SVC to avoid this issue, a possibility could be to add an argument to the tool that would allow users to decide whether:

1. If they require better compression and are ready to accept this security flaw, then SVC would proceed as it currently does, not encoding anything for only zero rows and not adding any padding information.
2. Or if what is needed is excellent security, and a smaller compression is acceptable, then SVC could add some padding before encrypting the data. Simply encoding the 0s into the matrix would not be sufficient since the compression algorithm will probably be able to compress very heavily this only-zeros matrix.

2.5.2 Features

SVC's current design offers an unmatched capability for data confidentiality and integrity coupled with a near-void additional encumbrance for data analysis. Only allowing some blocks to be decrypted at once means that most of the samples' data are kept secret at any time. Unfortunately this is not perfect, the need for aggregation of records and the lack of separation between the data of multiple samples means that when decrypting it is very likely that a lot more data than necessary will be decrypted and decompressed at some point, even if it might not make it to the final VCF file. A major improvement for SVC could come from the invention of a codec that would be able to separate the different samples, meaning that when only a subset of samples is requested, then keys associated with those samples are given, no other samples than those necessary are processed.

2.5.3 Compression Ratio

As of now, the genotype information is compressed by a ratio of several hundreds thanks to GVC. The rest of the data is what causes some file to have a compression ratio orders of magnitude lower. This can be seen when comparing different type of VCF contents. One way to improve this could be to try and optimize the way the metadata is organized before compressing it using gzip, but this will probably only sprout marginal improvements. Another possible option could be to try and implement a random-access friendly version of the VCFSHARK compression method,

which as seen in the introduction and in SVC's performance comparison, yields very impressive compression results.

2.5.4 Time Performance

Although it's the computation of the hamming distance matrix that boasts the worst complexity in the whole pipeline, this part of the code has been sufficiently optimized to not be the current bottleneck of run time performances. Measuring the time spent by each thread sleeping actually shows that, as of now, the input thread is not capable of producing blocks fast enough for process threads when the number of those threads reaches a certain threshold (the actual value depending on the machine's capabilities). This is due to a few factors. Firstly, `htslib` only allows for reading VCF records sequentially (i.e. it is not possible to jump over `n` records) and isn't really written for multi-threading. Additionally, a part of the input process is copying the data from `htslib` defined structure to SVC defined ones that will be used throughout the code, and this is quite an expensive process. This was done for 2 reasons:

1. Decoupling: This way, everything in SVC apart from the functions related from I/O would not have to know anything about how `htslib` works, and if a new, better library appears, only some parts of the code would have to change instead of everything
2. Clarity and simplicity: the way `htslib` defines and uses those structures is quite convoluted, by having custom data structures with clear names and simple meaning it was easier to handle them for the rest of the development process.

Since there is basically no dependence between different process threads, if the input code (and potentially the I/O library, `htslib`) could be re-written/re-designed to be way more efficient, then SVC could enjoy almost unlimited parallelization, greatly improving its runtime efficiency, especially so for high core-count systems.

The decompression time could also easily be improved since, as it currently stands, it is single threaded. By adding some parallelism. There would need to be some synchronization mechanism for the output of blocks since they need to be in the correct order in the result VCF file, but the run time could probably be reduced by a few factors making it competitive with `GenoZip`.

Chapter 3

Conclusion

As it stands, SVC offers neither the best compression ratio, although still much better than de-facto solutions, nor the best time taken to compress the data. Despite that, because of the lack of tools for the handling of massive VCF files, SVC fills a very important need: A free, open source, random access capable, specialized VCF compressor. This was already something lacking, but on top of this SVC brings brand-new security features that allow for easy and controlled communication of information. SVC has the potential of becoming a widely used solution for the handling of VCF files. Its current shortcoming, of run time performance and sub-optimal compression ratios are not impassable walls and can easily be overcome with some work and ingenuity.

Appendix A

Appendices

A.1 Code Availability

The source code of SVC is available on my personal github account, and is licensed under GNU GPLv3: <https://github.com/tGautot/SVC>

A.2 Common INFO and FORMAT fields

The Wikipedia page for the VCF file format [25] gives the content of Table A.1 as common INFO fields, and Table A.2 as common FORMAT fields. Note that for both of these, arbitrary fields can be added but then will be defined in the header of the VCF file.

Name	Brief description
AA	ancestral allele
AC	allele count in genotypes, for each ALT allele, in the same order as listed
AF	allele frequency for each ALT allele in the same order as listed (use this when estimated from primary data, not called genotypes)
AN	total number of alleles in called genotypes
BQ	RMS base quality at this position
CIGAR	cigar string describing how to align an alternate allele to the reference allele
DB	dbSNP membership
DP	combined depth across samples, e.g. DP=154
END	end position of the variant described in this record (for use with symbolic alleles)
H2	membership in hapmap2
H3	membership in hapmap3
MQ	RMS mapping quality, e.g. MQ=52
MQ0	Number of MAPQ == 0 reads covering this record
NS	Number of samples with data
SB	strand bias at this position
SOMATIC	indicates that the record is a somatic mutation, for cancer genomics
VALIDATED	validated by follow-up experiment
1000G	membership in 1000 Genomes

Table A.1: Common INFO fields, from Wikipedia

Name	Brief description
AD	Read depth for each allele
ADF	Read depth for each allele on the forward strand
ADR	Read depth for each allele on the reverse strand
DP	Read depth
EC	Expected alternate allele counts
FT	Filter indicating if this genotype was “called”
GL	Genotype likelihoods
GP	Genotype posterior probabilities
GQ	Conditional genotype quality
GT	Genotype
HQ	Haplotype quality
MQ	RMS mapping quality
PL	Phred-scaled genotype likelihoods rounded to the closest integer
PQ	Phasing quality
PS	Phase set

Table A.2: Common INFO fields, from Wikipedia

A.3 Test Data

"Simple" dataset from the 1000 genome project: <http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/>

"Annotated" dataset from the 1000 genome project: <http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20110521/>

A.4 Test Machine Specification

A.4.1 Operating System

The machine is running Ubuntu 22.04.4 LTS with a desktop environment.

A.4.2 CPU

No overclocking.

Short form: Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz

Long form (from `lscpu`):

```

Architecture:          x86_64
  CPU op-mode(s):      32-bit, 64-bit
  Address sizes:       39 bits physical, 48 bits virtual
  Byte Order:          Little Endian
CPU(s):                16
  On-line CPU(s) list: 0-15
Vendor ID:             GenuineIntel
  Model name:          Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
    CPU family:        6
    Model:              158
    Thread(s) per core: 2
    Core(s) per socket: 8
    Socket(s):          1
    Stepping:           13
    CPU max MHz:        5000,0000
    CPU min MHz:        800,0000
    Bogomips:           7200.00
    Flags:              <Skipped>
Virtualization features:
  Virtualization:      VT-x
Caches (sum of all):
  L1d:                 256 KiB (8 instances)
  L1i:                 256 KiB (8 instances)
  L2:                  2 MiB (8 instances)
  L3:                  16 MiB (1 instance)
NUMA:
  NUMA node(s):        1
  NUMA node0 CPU(s):   0-15
Vulnerabilities:
  <Skipped>

```

A.4.3 RAM

Short form: 32GB DDR4

Long form (from dmidecode --type 17, uninstalled modules skipped):

```

Handle 0x003B, DMI type 17, 40 bytes
Memory Device
Array Handle: 0x003A
Error Information Handle: Not Provided

```

Total Width: 64 bits
Data Width: 64 bits
Size: 16 GB
Form Factor: DIMM
Set: None
Locator: ChannelA-DIMM0
Bank Locator: BANK 0
Type: DDR4
Type Detail: Synchronous
Speed: 3200 MT/s
Manufacturer: 029E
Serial Number: 00000000
Asset Tag: 9876543210
Part Number: CMK32GX4M2E3200C16
Rank: 1
Configured Memory Speed: 3200 MT/s
Minimum Voltage: 1.2 V
Maximum Voltage: 1.2 V
Configured Voltage: 1.2 V

Handle 0x003D, DMI type 17, 40 bytes
Memory Device
Array Handle: 0x003A
Error Information Handle: Not Provided
Total Width: 64 bits
Data Width: 64 bits
Size: 16 GB
Form Factor: DIMM
Set: None
Locator: ChannelB-DIMM0
Bank Locator: BANK 2
Type: DDR4
Type Detail: Synchronous
Speed: 3200 MT/s
Manufacturer: 029E
Serial Number: 00000000
Asset Tag: 9876543210
Part Number: CMK32GX4M2E3200C16
Rank: 1

Configured Memory Speed: 3200 MT/s
Minimum Voltage: 1.2 V
Maximum Voltage: 1.2 V
Configured Voltage: 1.2 V

A.4.4 Disk

The disk being used for phase 1 files is a "Samsung Portable SSD T5" of 1 Terbyte. Being an external drive, it was connected via USB. A simple benchmark (from Ubuntu's *Disk* program) showed that the disk performed reads at a speed of 450 MB/s and Writes at 420 MB/s.

The second disk is a CT4000P3 SSD8 of 4TB, the same benchmark showed an average read speed of 450MB/s and an average write speed of 366MB/s.

References

- [1] Markus Kuhn. *Jbig-kit*. 1995.
- [2] David McGrew and John Viega. “The Galois/counter mode of operation (GCM)”. In: *submission to NIST Modes of Operation Process* 20 (2004), pp. 0278–0070.
- [3] Daniel J Bernstein. “The Poly1305-AES message-authentication code”. In: *International workshop on fast software encryption*. Springer. 2005, pp. 32–49.
- [4] Nayanah Siva. “1000 Genomes project.” In: *Nature biotechnology* 26.3 (2008), pp. 256–257.
- [5] Daniel J Bernstein. “Extending the Salsa20 nonce”. In: *Workshop record of Symmetric Key Encryption Workshop*. Vol. 2011. 2011.
- [6] Vincent Berg Berg. *Google Maps Side-Channel Attack*. Via IOActive, Inc. 2012.
- [7] Frank Denis. *The Sodium cryptography library*. June 2013. URL: <https://download.libsodium.org/doc/>.
- [8] J Craig Venter, Hamilton O Smith, and Mark D Adams. “The sequence of the human genome”. In: *Clinical chemistry* 61.9 (2015), pp. 1207–1208.
- [9] Hao Ye et al. “Alignment of Short Reads: A Crucial Step for Application of Next-Generation Sequencing Data in Precision Medicine”. In: *Pharmaceutics* 7 (Nov. 2015), pp. 523–541. DOI: 10.3390/pharmaceutics7040523.
- [10] Ryan M Layer et al. “Efficient genotype compression and analysis of large genetic-variation data sets”. In: *Nature methods* 13.1 (2016), pp. 63–65.
- [11] Heng Li. “BGT: efficient and flexible genotype query across many samples”. In: *Bioinformatics* 32.4 (2016), pp. 590–592.
- [12] Kedar Tatwawadi et al. “GTRAC: fast retrieval from compressed collections of genomic variants”. In: *Bioinformatics* 32.17 (2016), pp. i479–i486.
- [13] Cryptography Engineering LLC. *LibSodium Security Audit*. Aug. 2017. URL: <https://www.privateinternetaccess.com/blog/libodium-v1-0-12-and-v1-0-13-security-assessment/>.

- [14] Xiuwen Zheng et al. “SeqArray—a storage-efficient high-performance data format for WGS variant calls”. In: *Bioinformatics* 33.15 (2017), pp. 2251–2257.
- [15] Agnieszka Danek and Sebastian Deorowicz. “GTC: how to maintain huge genotype collections in a compressed form”. In: *Bioinformatics* 34.11 (2018), pp. 1834–1840.
- [16] Sebastian Deorowicz and Agnieszka Danek. “GTShark: genotype compression in large projects”. In: *Bioinformatics* 35.22 (2019), pp. 4791–4793.
- [17] Mary Mallappallil et al. “A review of big data and medical research”. In: *SAGE open medicine* 8 (2020), p. 2050312120934839.
- [18] James K Bonfield et al. “HTSlib: C library for reading/writing high-throughput sequencing data”. In: *GigaScience* 10.2 (Feb. 2021). giab007. ISSN: 2047-217X. DOI: 10.1093/gigascience/giab007. eprint: <https://academic.oup.com/gigascience/article-pdf/10/2/giab007/36332285/giab007.pdf>. URL: <https://doi.org/10.1093/gigascience/giab007>.
- [19] Sebastian Deorowicz, Agnieszka Danek, and Marek Kokot. “VCFShark: how to squeeze a VCF file”. In: *Bioinformatics* 37.19 (2021), pp. 3358–3360.
- [20] Divon Lan et al. “Genozip: a universal extensible genomic data compressor”. In: *Bioinformatics* 37.16 (2021), pp. 2225–2230.
- [21] Fatemeh Behrad and Mohammad Saniee Abadeh. “An overview of deep learning methods for multimodal medical data mining”. In: *Expert Systems with Applications* 200 (2022), p. 117006.
- [22] Sören Richard Stahlshmidt, Benjamin Ulfenborg, and Jane Synnergren. “Multimodal deep learning for biomedical data fusion: a review”. In: *Briefings in Bioinformatics* 23.2 (2022), bbab569.
- [23] Yeremia Gunawan Adhisantoso et al. “GVC: efficient random access compression for gene sequence variations”. In: *BMC bioinformatics* 24.1 (2023), p. 121.
- [24] *Gene* [Internet]. Bethesda (MD): National Library of Medicine (US), National Center for Biotechnology Information; 2004. <https://www.ncbi.nlm.nih.gov/gene/>. Accessed: 05/05/2024 (DMY).
- [25] *Variant Call Format*. https://en.wikipedia.org/wiki/Variant_Call_Format. Accessed: 19/05/2024 (DMY) 14:59.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl