Architecture Decision Record, ADR

This document contains specific choices regarding the Data Delivery System and the motivations behind them. The sections should, to as high degree as possible, be structured in the following way.

Topic

Mention the topic and, if possible, clarify why there needs to be a decision regarding it.

Date: Add the date here

Decision

Specify the final decision so that those that only want a quick answer do not need to sift through the rest of the text.

Decision Procedure

Describe how the decision was made. Were there alternatives? Make comparisons.

Framework

Date: 2019 / 2020-?-?

Decision: *Flask*

Decision Procedure

	Flask	Tornado
Pros	• The top growing Python framework • Plenty of online resources available • Flexible • Simple • REST Support via extensions • Integrated testing system – improved stability and quality of web applications	• Asynchronous – enables developers to handle many (thousands) concurrent connections • Fast • Flexible • Simple • Built-in support for user authentication • Possible to implement third-party authentication and authorization schemes
Cons	• Many extensions no longer supported • No build-in support for user authentication, but can be included using extensions • Efficiency can be effected by extensions	• Online resources not as many as for Flask • No built in support for REST API and the user needs to implement it manually

Since Flask is flexible and simple, extensions provide a large variety of functionalities including REST API support, it has an integrated testing system and there is more online support than for Tornado, Flask was chosen as the better option for the Data Delivery System framework. The built-in asynchronicity in Tornado is not an important feature since the system will not be used by thousands of users at a given time.

Database

Date: 2020-?-?

Decision: *MariaDB – Relational Database*

The initial database used under development was the non-relational CouchDB. As development progressed, it became of interest to investigate whether this was the best approach or whether a relational database was better for the systems purposes.

Decision Procedure

Relational Databases	**Non-relational databases**

• Structured – represent and store data in tables
• Use SQL – Structured Querying Language
• Allows you to link information from different tables
• Can handle a lot of complex queries and database transactionsAre good:
  o If the data structure is not changing often and you have structured data
  o At keeping data transactions secure
• Highly scalable
	• Unstructured – represent data in collections of JSON documents
	• Easy to use and flexible
	• Efficient
	• Scalable
	• Ideal for organizations and applications with a lot of data

The main motivation behind choosing a relational database is that we are forced to keep it structured and that it, when used in the correct way, can be more efficient than a non-relational database.

MariaDB was chosen (rather than e.g. MySQL) as the relational database because of it’s query performance and that it provides many useful features not available in other relational databases.

Compression Algorithm

Date: 2020-?-?

Decision: *Zstandard*

Decision procedure

GNU zip (Gzip) is one of the most popular compression algorithm and is suitable for compression of data streams, is supported by all browsers and comes as a standard in all major web servers. However, while gzip provides a good compression ratio (original/compressed size), it is very slow compared to other algorithms.

The encryption speed using ChaCha20-Poly1305 (in this case tested on a 109 MB file) is around 600 MB/s, but when adding compression as a preceding step, the speed was less than 3 MB/s and the compression ratio 3,25. Since the delivery system will be dealing with huge files, it’s important that the processing is efficient, and therefore that the chosen algorithms are fast. Due to this, Zstandard was tested with the same chunk size, resulting in a speed of 119 MB/s and a compression ratio of 3,1. Since Zstandard gave approximately the same compression ratio in a fraction of the time, Zstandard was chosen as the algorithm to be implemented within the Data Delivery System.

Encryption Location

Date: 2020-?-?

Decision: *Local encryption prior to upload*

Decision procedure

The most efficient way of delivering the data to the owners would be to perform encryption- and decryption in-transit, thereby decreasing the amount of memory required locally and possibly the total delivery time. However, there have been some difficulties finding a working solution for this, including that the crypt4gh Python package used in the beginning of the development did not support it.

On further investigation and contact with Safespring, we learned:

- Server-side encryption (and server-side stored keys) is technically possible on Safespring S3 storage BUT Safespring has chosen to not activate that function.
- Most of the S3- and Boto clients that Safespring uses, e.g. the bash cli s3cmd, goes through GPG (Gnu Privacy Guard, based off of OpenPGP where PGP stands for Pretty Good Privacy) which performs the encryption before uploading the files. GPG/PGP makes it possible to encrypt using one key and decrypt using one or more other keys. This enables a more automated process but does not simplify for us or contribute with anything useful to the delivery system.
- All users of the Safespring backup service perform encryption on their own and handle the keys themselves.

Due to this, the encryption will be performed locally before upload to the S3 storage, whether Crypt4GH is used or not.

Encryption Algorithm

Date: 2020-?-?

Decision: *ChaCha20-Poly1305*

Possible AES-GCM for small files if ChaCha20-Poly1305 is not possible in web interface security.

Decision Procedure

The new encryption format standard for genomics and health related data is Crypt4GH, developed by the Global Alliance for Genomics and Health and first released in 2019. The general encryption standard, however, is AES of which AES-GCM is an authenticated encryption mode. AES is currently (since 2001) the block cipher Rijndael. ChaCha20 is a stream cipher and is used within the Crypt4GH format. The most secure, efficient and generally appropriate format and algorithm should be implemented within the Data Delivery System.

Crypt4GH

Crypt4GH is a file format, not an algorithm as the other two alternatives.

The files are read in 64 KiB blocks. Each block is encrypted with the algorithm ChaCha20 and a 16 byte message authentication code (MAC) is generated from the ciphertext using the authentication algorithm Poly1305. These are saved together with a 12 byte nonce, a randomly generated number used only once. Each data block thus looks like this:

The data encryption is performed using a randomly generated, 32 byte, key. The data encryption key is itself encrypted with ChaCha20and saved in the files header. This encryption is performed using a 32 byte key generated from the Elliptic-Curve Diffie-Hellman key exchange algorithm, where the following combinations (R = recipient, S = sender) produce identical keys – a shared key:

- R **public key** + R **private key** + S **public key**
- S **public key** + S **private key** + R **public key**

The senders public key is saved within the files header, which is later used by the recipient to generate the shared key, decrypt the data key, and finally decrypt the data blocks. More detailed information on the Crypt4GH file structure and algorithms can be found.

The header can include multiple header packets, thereby enabling access to the file by multiple users.

Pros

- Ready-to-use file encryption format developed by Global Alliance for Genomics and Health
- Standard for genomics- and health related data
- Provides *confidentiality* – The files can only be decrypted by holders of the correct secret key

- Guarantees *integrity* – Each data block includes a message authentication code, MAC. A change in the data results in an invalid MAC
- The message size limit of the encryption algorithm (see ChaCha20-Poly1305) is not exceeded and there is a new key- nonce pair for each data block.

Cons

- Does not protect against insertions, deletions or reordering. The MACs represent a block each, not the entire file.
- Does not provide any way of authenticating the files – the sending and receiving parties identities cannot be proven. The reader cannot know who has encrypted and sent the file, and the writer cannot know who decrypts and reads the file.
- Although the possibility of file access by multiple users and random access is useful in many cases, for delivery purposes these features are not important and do not help.
- For encryption to begin, all previous operations such as compression, need to be finished. This means that the files need to be processed from start to end multiple times, leading to a large increase in delivery time. This will be specially evident for large or even huge files, which will be common in the delivery system.
- It may be possible to alter the crypt4gh package code to enable streaming (tested without success), however since the file format does not provide simplifying properties for our purposes, this was decided against.

For the reasons listed above (mainly point 1, 2 and 4 in the cons section), Crypt4GH was NOT chosen as the encryption format within the Data Delivery System.

AES-256-GCM

The Advanced Encryption Standard, AES, is a block cipher which combines the core algorithm with a mode of operation - techniques on how to process sequences of data blocks. In this case the mode is the Galois Counter Mode (GCM), an Authenticated Encryption with Associated Data (AEAD) mode. Block ciphers perform operations on blocks of bits, not individual bits. The block size for AES is 128 bits (16 bytes).

The message size limit for AES-GCM - the amount of data that can be securely encrypted using the same key- and nonce pair - is ~64 GiB.

AES is very fast on dedicated hardware.

Pros

- Most widely used authenticated cipher
- Essentially an *encrypt-then-MAC* construction – strongest security
- Protects against insertions, deletions, reordering – the MAC represent the entire file
- Parallelizable – Possible to encrypt and decrypt different blocks independently of other blocks
- Streamable – the MAC is computed from each block as it’s encrypted. No need to store all blocks before computing, or process the file a second time.
- Available in practically all crypto packages for all languagues. This would mean that client side encryption in the browser using JavaScript would be possible, and therefore a single algorithm could be used throughout the Data Delivery System.
- Part of the Internet Engineering Task Force (IETF) for secure network protocols IPSec, SSH, TLS

Cons

- Files larger than 64 GiB need to be partitioned. Alternatively multiple new key-nonce pairs would need to be saved and kept track of in another way.
- Nonce repetition reduces the security drastically – tags can be forged
- Not as fast in software implementations
- Easy to get wrong
- Vulnerable to cache-timing attacks

For the reasons listed above (mainly point 1, 2 and 4) and that the pros for ChaCha20 over-weigh the pros for AES, AES-256-GCM was NOT chosen as the encryption algorithm within the Data Delivery System.

ChaCha20-Poly1305

ChaCha20-Poly1305 is an Authenticated Encryption with Associated Data (AEAD) algorithm, and combines the encryption algorithm ChaCha20 with the authentication algorithm Poly1305. Note: This is the algorithm used within the Crypt4GH file format.

ChaCha20 is a stream cipher and therefore processes the files bit-wise, not block-wise as in block ciphers. Stream ciphers generate pseudorandom bits from the key and encrypts the plaintext by XORing it with the pseudorandom bits.

The message size limit for ChaCha20-Poly1305 – the amount of data that can be securely encrypted using the same key- and nonce pair - is ~256 GiB.

Pros

- Is based on the ARX design – does not use substitution boxes as AES does and therefore does not produce cache footprints. It is therefore not vulnerable to cache-timing attacks.
- Fast in software implementations. Multiple times faster than AES.
- The security is currently considered to be at least as secure as AES-GCM
- Intrinsically simpler than AES
- Easier to implement and not as easy to get wrong
- The message size limit is larger than for AES
- Used in Crypt4GH file format – the encryption format standard for genomics and health related data
- Part of the Internet Engineering Task Force (IETF) for secure network protocols IPSec, SSH, TLS

Cons

- Not the general standard
- Has not undergone as much cryptanalysis as AES and may therefore be vulnerable to attacks which are currently unknown
- To our knowledge, it is not available within the WebCrypto API, used for client side encryption in the browser. Therefore, AES may still need to be implemented within the web interface.

Since files delivered within the system can be huge, it is important to choose a format or algorithm which has a high message size limit, is as fast and secure as possible for all file sizes, and has the least possible risk of implementation error. ChaCha20 fulfills this, and is in addition supported by

both the Global Alliance for Genomics and Health, and IETF secure network protocols. Although ChaCha20 is not the general encryption standard, it is becoming increasingly more used, becoming favored over AES in certain aspects and so far cryptanalysis have not (to our knowledge) found any vulnerabilities that are deal-breakers. The issue regarding client-side encryption in the browser for the web interface may be solved in one of the following ways:

- HTTPS will be used, which is secured via TLS. If TLS is deemed to give enough protection to the uploaded data from the browser to the server, ChaCha20 encrypts the files when they have reached the server and uploads them to Safespring S3 in a similar way as the CLI.
- If an extra layer of security is needed, encryption can be performed in the browser using AES-GCM. In this case, files smaller than the web interface size cap will also be encrypted using AES-GCM in the CLI. Information on the algorithm used per specific file will be added to the database. Since only smaller files will be able to be uploaded and downloaded through the web interface, the files encrypted with ChaCha20 will not be a problem since they would have needed to be downloaded through the CLI regardless of used encryption algorithm.

Files larger than 256 GiB will need to be partitioned, however the number of partitions will be far less than if AES-GCM was used. Finally, the algorithm description does not include how the data encryption key will be distributed in a secure manner – this will be handled in a similar way to the Crypt4GH format, but the public keys will be signed and uploaded to the database and not saved within the files.

Due to this, ChaCha20-Poly1305 was chosen as the encryption algorithm within the Data Delivery System.

File Chunk Size

Date: 2020-?-?

Decision: *64 KiB (65 536 bytes)*

Decision Procedure

Compression and encryption is performed in a streamed manner to avoid reading large files completely in memory. This is done by reading the file in chunks, and the size of the chunks affect the speed of the algorithms, their memory usage, and the final size of the compressed and encrypted files.

To find the optimal chunk size, a 33 GB file was compressed and encrypted using the chosen algorithms (Zstandard and ChaCha20-Poly1305) after reading the file in different sized chunks ranging from 4 KiB to 500 KiB. The image below shows why 64 KiB was chosen - the memory required for the operations increases after 64 KiB, without any significant gain in speed or compression ratio.

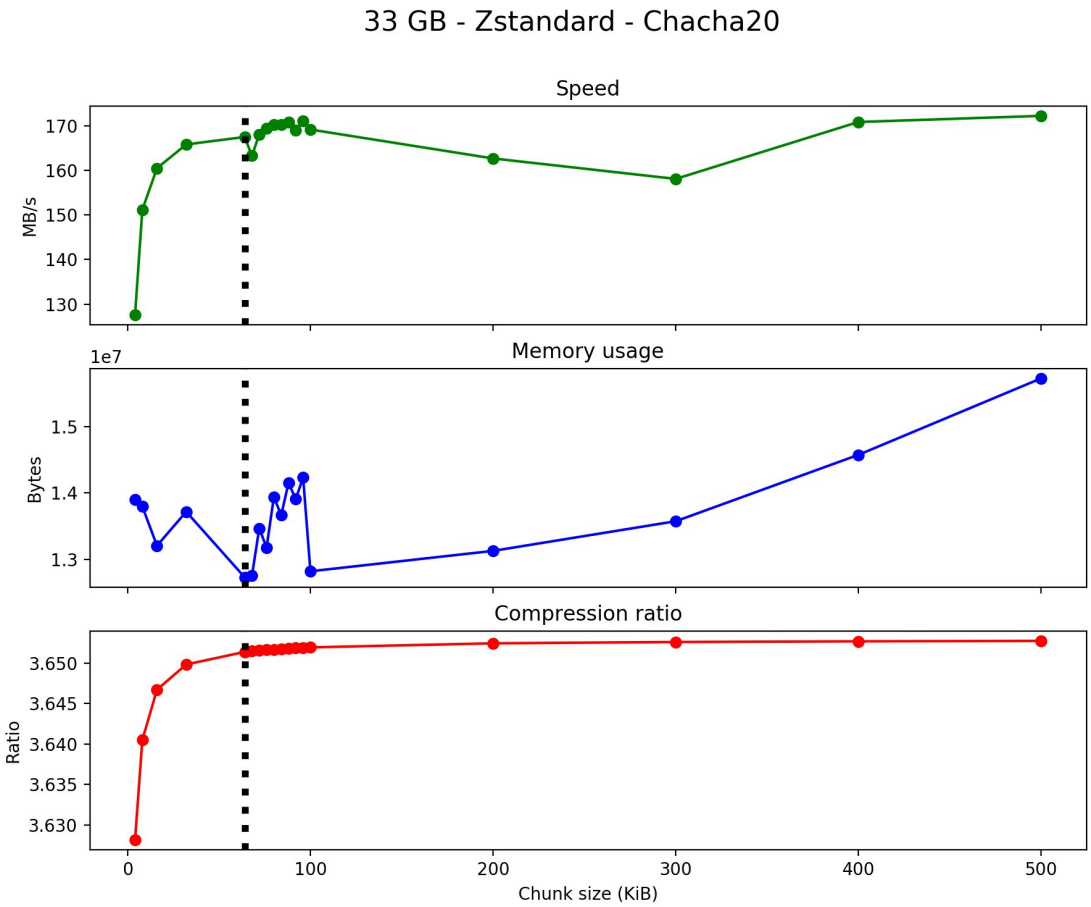

Figure 1. Compression ratio is defined as the uncompressed size divided by the compressed size, however, in this case the overhead of the encryption algorithm is also included. Thus here the compression ratio is calculated as the uncompressed size divided by the final size (compressed and encrypted).

File Integrity Guarantee

Date: 2020-?-?

Decision: *Nonce incrementation and checksum verification after upload*

Decision procedure

As described in the *Encryption Algorithm* decision section, the Crypt4GH format encrypts the files in blocks of 64 KiB, after which each data blocks unique nonce, ciphertext and MAC are saved to the c4gh file. This guarantees the integrity of the data blocks, however it does not guarantee the integrity of the entire file, and it is therefore possible that some blocks are rearranged, duplicated or missing, without the recipient knowing. Although we have chosen to not use the Crypt4GH format within the delivery system, we do use the same encryption algorithm – ChaCha20-Poly1305 – and (since we cannot read huge files in memory) we have chosen to read the files in equally sized chunks. Therefore the integrity issue can potentially give huge problems for the delivery system.

One solution to the integrity guarantee issue is to generate an additional MAC, representing the entire file, but for security reasons it is recommended to let the MAC generation be handled by (and in connection to) the encryption algorithm. Another solution (as mentioned here) is to generate a random nonce, and increment it for each block – decryption only needs to know the first blocks nonce, and then increments it in the same way as the encryption until the end of the file. If the decryption of a block fails - the file has been altered.

Since the nonces are regarded as public knowledge, the incrementation of the nonces do not decrease security - it may on the other hand increase the security: unique nonces for each data block is vital to the cryptographic security and generating a random nonce for each data block increases the risk of nonce reuse. Incrementing the nonces however mean that 296 (for 96 bit nonces as used in ChaCha20-Poly1305) data blocks can be encrypted before nonce reuse – 296 * 64 * 1024 * 8 bits (5.19229686 × 1021 terabytes). This number may even be higher, since ChaCha20 itself adds and increments 4 bytes to the user-specified nonce. None the less, it’s safe to say that files of this size will not exist. An extra advantage to the nonce incrementation option is that only the first nonce needs to be saved, reducing the encryption overhead by 128 bits per data block.

One problem with this solution is that, although we guarantee the order of the data blocks, we don’t know if all the data blocks are present – blocks at the end of the file may be missing without us knowing. Due to this, we also save the last nonce to the file. If decryption reaches the last nonce, the files integrity is proven, if not something has gone wrong.

After Upload

Initially, a file md5 checksum was generated during file processing. After upload, the checksum was compared to the ETag for the S3 object, indicating if the complete files had been uploaded. However, this method did not work for multipart uploads - files larger than 5 MB. On investigation of this, this GitHub issue was found, which addresses the issue:

> *[…] the md5 is only equal to the ETag under certain circumstances (i.e. non multipart upload).*

> *Boto3 does not integrity check the md5 of the entire file for multipart uploads, but it does send an md5 header for each part that is uploaded when doing a multipart upload such that if there is an md5 mismatch in any of the parts, it will retry the request until it is correct.*
>
> *That is probably the best we can do while still doing the transfer efficiently because determining the MD5 and doing integrity checking would require streaming the entire file upfront into memory.*

Due to this, no checksum verification is used during the upload. However, the file integrity checks described here make sure that the files are complete after delivery.

Password Authentication KDF

Since the Data Delivery System is aimed at handing large amounts of sensitive information, it is important that the access to the system is controlled and validated in a secure manner. Simply hashing the passwords and saving the results is certainly not enough to guarantee this level of security, nor is salting the passwords before hashing. The cryptographic key-derivation functions (KDFs) Scrypt, Bcrypt and Argon2 were chosen as the alternatives for password management within the DDS. These also include steps such as salting etc., however, they include additional steps, making them more resistant to multiple types of attacks.

Date: 2021-?-?

Decision: *Argon2id*

Python package: argon2-cffi

Decision Procedure

See motivation in decision below.

Date: 2020-?-?

Decision: *Scrypt*

Scrypt is memory-intensive, meaning that the cost is high to perform the key derivation. This is not an issue when performing the operation once (as in normal login procedures), however in the case of hackers attempting to crack the password, the high amount of memory (and time depending on the n-parameter value) required results in the hacking becoming very costly. Thus the goal of the algorithm is to make hacking the passwords as inconvenient and impractical as possible.

Decision Procedure

Scrypt	Bcrypt	Argon2
• Strong cryptographic key-derivation function (KDF) • Memory-intensive • Designed to prevent GPU, ASIC and FPGA attacks	• Secure KDF • Older than Scrypt • Less resistant to ASIC and GPU attacks • Uses constant memory → easier to crack	• Highly secure KDF function • ASIC- and GPU-resistant • Better password cracking resistance than Bcrypt and Scrypt
“It is considered that Scrypt is more secure than Bcrypt, so modern applications should prefer Scrypt (or Argon2) instead of Bcrypt.”		“In the general case Argon2 is recommended over Scrypt [and] Bcrypt […]”

In the first pre-release version of the Data Delivery System, Scrypt was used via the Python cryptography package. The plan was to change to Argon2, provided that the package mentioned in the link below is deemed to be appropriate, or that an alternative Python package is found. This has not been investigated yet since Scrypt is considered to provide a high level of security and focus has been on developing other key parts of the system.

All parameters used affect the memory- and CPU usage and all apart from the password are saved to the database. This does not compromise security. During user authentication, the parameters are retrieved from the database and used to test the specified password. If the resulting password hash matches, the password is correct.
