CLEAR Cryptosystem ™
User Manual ©

Quantum Knight, Inc.

Contents

Preface

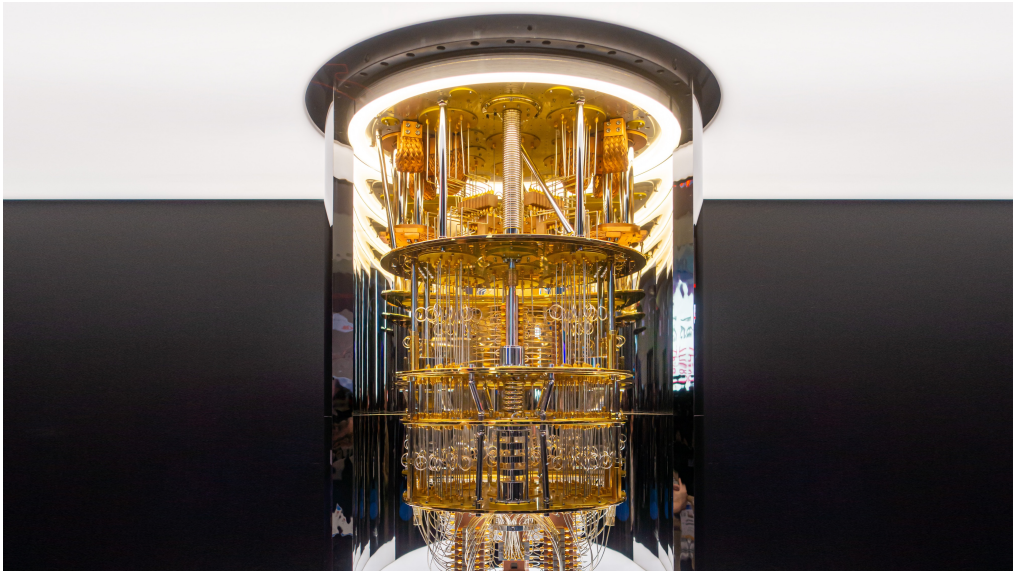
This document provides detailed instructions for using the CLEAR Cryptosystem ("CLEAR").

In a world where advacnces in quantum computing will soon render traditional cryptographic algorithms like AES and RSA obsolete, CLEAR provides post-quantum security starting at 512-bit strength, going all the way up to 10,240-bit key sizes. CLEAR has received NIST certification and is both FIPS-140-2 and GOOSE compliant. This means that you can trust the peer-reviewed mathematics underpinning CLEAR with absolute confidence.

CLEAR fundamentally is a symmetric cipher, allowing for the encryption and decryption of strings, files and streams (abritrary binary data). While historically developers have had to choose between strength and performance, CLEAR operates in single milliseconds with a low 400 kB footprint, making it suitable even for devices, IoT and industrial usage. We have demonstrated the ability to stream 8K video with minimal loss of frames per second even at key sizes of up to 5,120-bit.

To leverage CLEAR, the bundle you have downloaded contains both a command-line interface, useful for scripting and learning about the various modes available for encryption and decryption, as well as a full software development kit (SDK) and an intuitive Java-based API for integrating with your systems, devices and services.

Quantum Knight, the developers of CLEAR, believe in a world where security and performance do not have to be traded-off against one another, in a world where crygography is both easy and fun to use, and in a world that remains secure even as quantum computers grow ever more powerful and commonplace. Stay safe, and stay secure!



How long do you want these messages to remain secret?
I want them to remain secret for as long as men are capable of evil.
Neal Stephenson, Cryptonomicon

1 Quick Start Guide

This guide will quickly introduce you to the command-line interface options for installation, as well as encryption and decryption. For those wishing to integrate with the Java SDK, string-based, file-based and stream-based examples are provided. Please read the remainder of the documentation for additional details.

CLI: Validate Installation/Show Help

```
%java -jar clear.jar
```

CLI: Register for Trial/Activate License

```
%java -jar clear.jar -license
```

CLI: Encrypt a File

```
%java -jar clear.jar -encrypt <cleartext_file> {options}
```

CLI: Decrypt a File

```
%java -jar clear.jar -decrypt <encrypted_file> <keyfile> {options}
```

CLI: Additional Options for Encryption

```
-hmac           (Uses AEAD Encryption Authentication with HMAC)
-compliance     (Runs with FIPS 140-2 Compliance Mode)
-rng            (Use pluggable random number generator provider)

-bit            <strength in bits (defaults to 512)>
```

SDK: Encrypt and Decrypt a String

```
CLEARResult result = CLEAR.clearString().encrypt(
    someString,
    512,
    SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY
);

CLEAR.clearString().decrypt(
    result.getCipherTextString(),
    result.getKeyMaterial(),
    SingleJobDecryptionMode.SINGLE_JOB_KEY
);
```

SDK: Encrypt and Decrypt a File

```
CLEARResult encrypted = CLEAR.clearFile().encrypt(
    someFile.getAbsolutePath(),
    512,
    StreamingEncryptionSysGenKeyMode.STREAMING_SYSTEM_GEN_KEY
);

CLEARResult decrypted = CLEAR.clearFile().decrypt(
    encrypted.getCipherTextFile(),
    encrypted.getKeyFile(),
    StreamingDecryptionMode.STREAMING_KEY
);
```

SDK: Encrypt and Decrypt a Strean

```
CLEARResult encrypted = stream.encryptStart(
    someString.getBytes(),
    512,
    StreamingEncryptionSysGenKeyMode. STREAMING_SYSTEM_GEN_KEY,
    true
);

CLEARResult decrypted = stream.decryptStart(
    encrypted.getCipherText(),
    encrypted.getKeyMaterial(),
    StreamingDecryptionMode.STREAMING_KEY,
    true
);
```

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files.

Bruce Schneier

2 Command-Line Interface

CLEAR is Quantum Knight's next generation cipher, protecting data at rest and in transit on any device with a Java Runtime or Virtual Machine. CLEAR is most easily approached starting with the command-line interface (CLI). Here, you may encrypt and decrypt local filesystem resources. For each command invocation, a number of operating modes are available. This chapter will briefly cover the basic CLI operations. To learn in detail about available additional modes, please refer to "Advanced Topics: Operating Modes".

Validating the Installation

As a prerequisite, ensure that your system contains a Java runtime of version 8 or higher. You can validate that you have Java on your path via:

```
% java -version
java version "17.0.6" 2023-01-17 LTS
Java(TM) SE Runtime Environment (build 17.0.6+9-LTS-190)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.6+9-LTS-190, mixed mode, sharing)
```

If your environment does not have a working version of Java, download a Java Runtime Environment for your platform at:
https://www.java.com/en/download/.

Navigate to the directory where you installed CLEAR. You should see a version of the application (for this document, we will use 1.2.1.5). When typing your own commands into the shell, be sure to use the name of version installed (e.g., clear_1.2.1.5.jar). To validate that the application is working:

```
% java -jar clear.jar
```

Simply running CLEAR with no parameters will bring upa help page and indicate the status of your license. CLEAR will always perform decryption. A license is only required to perform encryption once the 30-day trial period has ended.

Obtaining a License

We will now activate our 30-day trial period. During this time, you may freely encrypt and decrypt via both the CLI and the SDK.

```
%java -jar clear.jar -license
```

Now follow the on-screen prompts to activate your trial period. Once the 30-day trial period where you may freely use application (both CLI and the Java SDK) ends, you will need to purchase a license to continue encrypting resources. Place the downloaded license in the same directory as the CLEAR executable. Then, you can activate the license via:

```
%java -jar clear.jar -license
...
------------------------------------------------------------
LICENSED TO:  John Doe
              john.doe@foo.bar

LICENSE:      MONTHLY - CLEAR FULL-FEATURED - 10240 BIT
EXPIRES:      07-08-2023
------------------------------------------------------------
```

As the text above indicates, CLEAR is now licensed and fully functional. If you encounter issues with the registration process or use of your license, please contact help@quantumknight.io.

File-Based Encryption
We will start with encrypting a file. However, CLEAR supports the normal range of operating system redirects (for standard out) and pipes as well. CLEAR will generate two artifacts: the encrypted result and the system-generated key that will be used for decryption. The general format used for encryption is:

```
%java -jar clear.jar -encrypt <cleartext_file> {options}
```

To start with a simple encryption of a file readme.txt using CLEAR's default key-size of 512 bits:

```
% java -jar clear.jar -encrypt readme.txt
Strength Level: 512
% ls -l
total 16640
-rw-r--r--@     11184 Apr 10 13:44 CLEAR.LIC
-rw-rw-r--@    404057 Mar 17 10:32 clear_1.2.1.5.jar
-rw-r--r--         13 Apr 24 08:36 readme.txt
-rw-r--r--       6164 Apr 24 08:36 readme.txt.ckey
-rw-r--r--         21 Apr 24 08:36 readme.txt.clear
```

The application performed the encryption and output two files: readme.txt.ckey (the system-generated key) and readme.txt.clear (the encrypted file). Note that the ciphertext is 8 bytes longer than the plaintext, with the additional data residing in a header stored in the ciphertext.

```
%java -jar clear.jar -decrypt readme.txt.clear readme.txt.ckey
%cat readme.txt
Hello World!
```

As can be seen from the two examples above:

> Encryption requires at least a single parameter: the plaintext to encrypt.
>
> Decryption requires at least two parameters: the ciphertext to decrypt and the key to use.

If privacy is outlawed, only outlaws will have privacy.
Philip Zimmermann

3 SDK Integration

CLEAR provides a robused Software Development Kit (SDK) to integrate cryptography with your Java services, applications and devices. The SDK provides three separate higher-level abstractions to work with keys and ciphers:

- **String API:** encrypts and decrypts UTF-8 data while leveraging base-64 encoding to ensure ciphertext remains a valid string.
- **File API:** encrypts and decrypts individual files. Since files may contain binary data, the encrypted result will not have base-64 encoding performed.
- **Stream API:** encrypts and decrypts binary data from any source representable as a byte array. As with the files API, no base-64 encoding occurs.

All three APIs will utilize CLEAR's HyperKey technology. In order to understand the commmon principles between all three interfaces, we will first explore how HyperKeys, CLEAR's patented symmetric keys, operate and the different ways in which you will generate a key.

We will then proceed to dive into the `CLEARResponse` object to understand which fields to use to extract the return value from an encryption or decryption. Finally, we will briefly explore the various operating modes, which control what additional features will be used for a given operation.

With that foundational understanding in hand, we will demonstrate encryption and decryption for all three interfaces: Strings, Files and Streams. (This is when we will get to the fun part: coding!)

3.1 License Configuration

In order to encrypt, you must have a valid CLEAR license (either trial or paid). Decryption is always allowed regardless of license status. CLEAR will look for your license in the following locations:

1. In the same directory as the clear.jar file
2. At the location specified by the `CLEAR_LIC` environment variable

The preferred method to fix the location of the license is to specify the `CLEAR_LIC` environement variable. However, if you package CLEAR within a WAR file or similar methodology, then place the license in the same directory as clear.jar. When debugging or running in your IDE, you can normally create a run configuration that passes the `CLEAR_LIC` variable directly if you do not have permissions to modify the global environment variables on your system.

You must specify the actual license file and not a directory when using the environment variable approach.

> Note: If you inspect the result of encryption or decryption, and the `CLEARResult` that is returned contains all null values, then your license file could not be located. You can easily debug this by adding a call to `System.getenv()` to see if or where it has been set.

3.2 Understanding Keys

CLEAR's patented technology for symmetric keys is known as a HyperKey. Traditional cryptographic keys solely contain the data required to encrypt or decrypt a message. However, CLEAR keys also may contain additional metadata beyond the initialization vector, salt and entropy required for encryption and decryption:

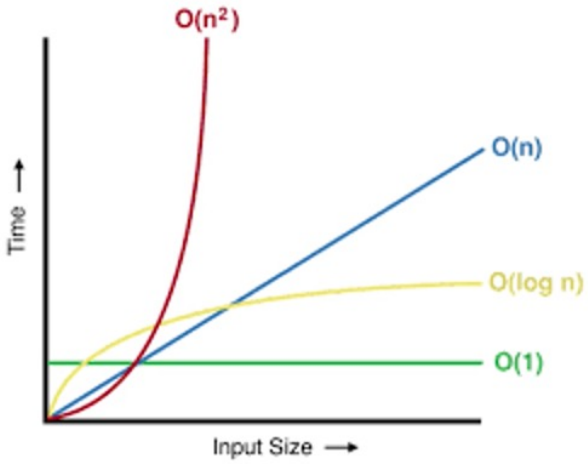
- Multi-factor authentication data, from a hardware token, a biometric, etc.
- HMAC data utilized to authenticate a given message.
- Access control lists (paired with MFA) to allow different principals to decrypt different portions of the ciphertext.

> Note: some features listed above require additional options enabled on your CLEAR license. Please contact your administrator or Quantum Knight to inquire.

Encryption will normally generate both the ciphertext from the specified plaintext as well as a key to later utilize for decryption.

Key Strengths

Higher key strengths typically correspond to both stronger security and longer processing time. A traditional algorithm exhibits $\mathcal{O}(n^2)$ performance, meaning doubling the key size leads to an expoential increase in the time required for encryption or decryption. CLEAR, in contrast, operates in $\mathcal{O}(n \log n)$ performance. As key sizes increase, the total time required for a given operation still increases, but the performance improves relative to the larger key size.



When selecting a given key strength, higher is not always better. You should analyze the following characteristics when choosing a size, as performance often will be a consideration as well as security:

- How long will the data need to be secure? Data in transit normally has a lower need for protection than data at rest. The longer the data needs to remain encrypted, the higher the key size you should use.

- What type(s) of adversary must you defend against? All sorts of attackers from script kiddies, commercial competitors, organized crime syndicates to nation states will present different levels of sophistication and resources. The greater the threat, the higher the key size.
- What damage would occur if the information was disclosed? Forms of damage include reputational, financial, criminal and others. As the level of consequences increases, so should the your key strength.

To assist in choosing your key size, the following table roughly illustrates the typical use case based on a simple rubric:

Method	**Purpose**
512	Post-quantum strength data at rest
1,024	Commerial post-quantum strength for data at rest
2,048	Military-grade post-quantum strength
5,120	Intelligence agency strength
10,240	Maximum strength suitable for protecting data for decades

The above examples show the relative strength and provide some guidance for a given key size. However, when possible, analyze the time the data will need to remain secure, the types of adversary you are guarding against and the scope of damage you are attempting to mitigate.

Traditional symmetric ciphers utilize 256-bit keys (sometimes up to 512 but with significant performance penalties). Asymmetric ciphers have higher key-sizes (2,048 or 4,096) but cannot encrypt large amounts of data and cannot be directly compared to an asymmetric key size. For example, AES 256-bit will typically secure data better than RSA 4,096.

CLEAR's 512-bit strength will guarantee quantum-resilience while having better performance than a traditional cipher.

CLEAR allows several ways to utilize and generate keys. These fall into three principal categories: system generated keys, user-supplied keys and user-generated keys. The enum `com.clear.cryptosystem.CLEARMode` specifies the allowable operations and key modes that one may utiliize.

System-Generated Keys

System-generated keys occur at encryption time. CLEAER will generate a new random key, use that to encipher the plaintext and then return both the ciphertext and the generated key in the `CLEARResult`. A typical use case would be to then store the ciphertext along with the encrypted data, while putting the generated key in separate storage (or ideally a key vault) to be used for subsequent decryption. CLEAR does not offer a key vaulting solution interally, as so many good options already exist in the market.

The CLEAR interface is designed for ease of use and minimal required coding. The generation of entropy (randomness) in Java is frequently accomplished using the `SecureRandom` class for what is deemed to be cryptographically secure random number generation, capable of producing random integers as well as binary byte-array sequences. In the default configuration, the Java implementation of CLEAR also utilizes Java `SecureRandom` for its system-generated encryption keys.

System-generated keys do not require a user-defined key or password to function. CLEAR will automatically generate strong entropy with the aforementioned cryptographically secure random number generator (CSRNG) as the starting place for all encryptions. Like other symmetric algorithms, CLEAR's keys are internally constructed in way that incorporates an initialization vector, a salt, and a secret key vector. Each of these precursory artifacts are made directly from a series of 64-bit long integers, produced directly by CSRNG as their source of entropy. For additional detail on the inner-workings and mechanics of CLEAR encryption, please see our white paper available at https://quantumknight.io.

User-Generated Keys

User-generated key encryption will take a user's password and blend it with pseudo-randomly generated numbers to create sufficient entropy to generate 512 bits of key space. While the minimum amount of data required to generate a UGK is one single character, the Quantum Knight team recommends providing MFA data that is at least as long as the bit strength of the encryption operation. Like system-generated keys (above), CLEAR will return both the ciphertext and the key derived from the password in a CLEARResult.

Further, CLEAR does not prevent large amounts of UGK data as an input parameter, however; the use of very large UGK Material may be an unnecessary waste of system memory resources.

As a best practice in symmetric cryptography, the Quantum Knight team recommends against password reuse in encryption keys. As a form of "abstract" non-transactional key exchange, the CLEAR UGK interface could be used to produce CLEAR-compatible key material on two sides of an encryption sharing-schema if Alice and Bob were both aware of a password and create their own private keys via UGK modes.

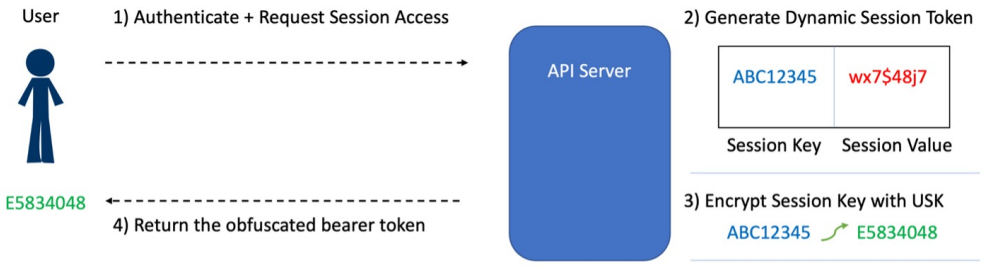
Regardless of the amount of entropy supplied (i.e., length of the password) or the key size specified, user-generated keys are limited to 512-bit encryption strength.

User-Supplied Keys

Real-world symmetric cryptography often implies the need for a User Supplied Key ("USK"). Simply, this means that we can begin encryption (or decryption) with a CLEAR HyperKey compliant format that has been produced earlier, and at a different time than a specific encipherment operation.

Here's an example:

As a CLEAR SDK developer, you are creating some code within a website or internet-facing API that is used to tokenize strings whereby key-exchange or key-rotation is not feasible or unnecessary to the security of your use-case.



As shown in the example above, there are scenarios whereby the use of a single encryption key may be used to "blind" dynamic data so that it can be shared externally without revealing the actual system state. The Quantum Knight team would like to convey that proper key management and handling is foundational to the security of systems and infrastructure.

The exhibit shown above in Figure-6 is only intended to illustrate a data transformation with USK in the simplest terms and is not being presented as a strategy or security recommendation. Tokenization of secure information in a well-managed environment is a best practice.

In the case of SGK, the default onboard RNG can be used, or a plug-n-play random number generators (RNG) can be used to provide cryptographic key entropy.

There are use-cases whereby creating a CLEAR encryption key (without performing an encryption job) may be convenient, necessary, or specifically required to satisfy a solution. For this reason, CLEAR provides "Key Tool" as a convenient utility feature for generating stand-alone keys without performing an encryption operation. Keys can be stored, used later, or applied specifically to new encryption jobs in lieu of generating keys during the encryption process.

Additional detail and suggested use of User Supplied Keys (USK) is provided in ["Advanced Topics - KeyTool"].

3.3 Inspecting Results

CLEAR will output the results of encryption or decryption–regardless of string, stream or file-based encryption–in a `CLEARResult` instance. As an example:

```
// Set to string-based encryption (thread-safe and may be reused)
CLEARString cs = CLEAR.clearString();

// Encrypt a string at 512-bit with a system-generated key
CLEARResult encrypted = cs.encrypt(
    "Hello world",
    CLEAR.STRENGTH_512_BIT,
    SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY
);
```

We have encrypted a string using a system-generated key at 512-bit strength. Every call to an encrypt() or a decrypt() method will return a CLEARResult. Let's inspect the details of what might be returned.

Type	Target	Method	Returns	Description
Encrypt	Strings	getCipherTextString()	String	Encrypted string
	Files	getCipherTextFile()	String	URI of the encrypted file
	Streams	getCipherText()	byte[]	Encrypted binary data
	All	getKeyMaterial()	String	Key used
Decrypt	Strings	getClearTextString()	String	Encrypted string
	Files	getClearTextFile()	String	URI of the decrypted file
	Streams	getClearText()	byte[]	Encrypted binary data
	All	getKeyMaterial()	String	Key used

To complete the example above, let's decrypt the result that we earlier encrypted:

```
// Decrypt the string
CLEARResult decrypted = cs.decrypt(
    encrypted.getCipherTextString(),
    encrypted.getKeyMaterial(),
    SingleJobDecryptionMode.SINGLE_JOB_KEY
);

// Outputs "hello world"
System.out.println(decrypted.getClearTextString());
```

Note how we simply used the encrypted result from before when performing decryption. In most nontrivial instances, the decryption does not occur in memory right after encryption. In that case, you would pass the appropriate encrypted data and key material directly to the function. The above illustrates how to inspect `CLEARResult` from an encryption operation and a decryption operation. You would typically call the `getCipherText()`, `getCipherTextFile()` or `getCipherTextString()` and store the encrypted result somewhere. In addition, the key must be stored for subsequent decryption. Please see "Advanced Topics - Key Storage" for advice on properly safeguarding your keys.

The curious may have noticed two additional methods availble in CLEARResult, namely `isHmacSegment()` and `getHmacSignature()`. These methods are used when operating in HMAC mode to verify the authenticity of an encrypted message. Please see section "Advanced Topics - Operating Modes" for more details on HMAC.

3.4 Operating Modes

Keys may be generated and used in a variety of different operaitng modes. All keys perform encryption and decryption. However, keys may also incude second-factor (2FA or biometric) data when operating in HMAC mode, guaranteeing the authenticity of a given message. Keys may also operate in FIPS-140-2 compliance mode (see "Advanced Topics - Operating Modes" for more details). And finally, keys may include both HMAC and FIPS compliance operation.

A given key will normally contain a header. Headers are required for HMAC and FIPS compliance use. However, when these additional features are not required, and the encryption is occuring in a resource-constrained environment (for example. a camera, an industrial sensor or a commercial IoT device, etc.), CLEAR can operate in "Headerless" mode, whereby the cipehrtext will not contain a header. When utilizing the File and Streaming interfaces, the ciphertext will have the same size as the plaintext. (Strings will have Base-64 encoding applied and hence change the ciphertext length regardless of Headerless operation). Another use case for this feature is encrypting database columns, as there is no need to expand the column size to accomodate the additional 8kB a header will normally add to the encrypted result.

Mode	**Description**
Key	Standard key for encryption and decryption
Headerless	Key without any header information
Compliance Mode	Key operating in FIPS-140-2 compliance mode.
HMAC	Key with HMAC embedded
HMAC and Compliance	Key with HMAC embedded in FIPS compliance mode

CLEAR supports the above operating modes for three separate key modes: system-generated keys, user-generated keys and user-supplied keys. Note that all the key modes that follow pertain to encryption. Decryption either uses the header to determine the appropriate mode to use or operates in Headerless mode (where HMAC and FIPS compliance do not pertain).

We will start with basic encryption and decryption. For those interested in the additional modes of Headerless, compliance and HMAC (or their permutations), please see "Advanced Topics - Operating Modes".

3.5 String-Based Encryption

The String interface is a convenience method built on top of the root binary interface and is designed to make encrypting textual data more convenient. Additional convenience (for Strings) is achieved via the incorporation of UTF-16 encoding of cleartext binary input and Base64 encoding of binary ciphertext output. UTF-16 Unicode character encoding widens all characters to 2 bytes to support multiple languages and advanced character-sets. In this way, data encrypted using the String interface will result in a cyphertext that is larger (in byte size) than the original. If looking to achieve encryption that is the same size as the original, please utilize the CLEAR streaming interface whereby you may directly encrypt byte arrays or the File interface for filesystem resources.

```
// Set to string-based encryption (thread-safe and may be reused)
CLEARString cs = CLEAR.clearString();

// Encrypt a string at 512-bit with a system-generated key
CLEARResult encrypted = cs.encrypt(
    "Hello world",
    CLEAR.STRENGTH_512_BIT,
    SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY
);
```

As illustrated by the example above, CLEAR String encryption requires only two lines of code in all cases. In the following sections we will explore the various input parameters, response outputs, and unique modes of operation available with CLEAR String encryption. String encryption is intended for use within online applications, messaging systems, and anywhere that unique and strong protection of textual data or tokenization is required. String Encryption provides various methods for applying security to text in a highly performant manner.

Now, let's decrypt the result above. The code below will only focus on the decryption portion to demonstrate that again only a single line of code is required to implement:

```
CLEARResult decrypted = cs.decrypt(
    encrypted.getCipherTextString(),
    encrypted.getKeyMaterial(),
    SingleJobDecryptionMode.SINGLE_JOB_KEY
);
```

When operating in "Headerless "mode, CLEAR omits the 8 bytes normally prefixed to a given ciphertext (see "Advanced Topics - Headerless Operation" for details). This mode would be suitable for operating in an IoT, device, camera or other resource-constrainted environnment. Normally, Headerless also makes an ideal use case for database column encryption. However, since the String API encodes encrypted results as Base-64, the ciphertext will always be larger than the plaintext.

```
CLEARResult encrypted = cs.encrypt(
    "Foo bar",
    CLEAR.STRENGTH_2048_BIT,
    SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_HEADERLESS
);

CLEARResult decrypted = cs.decrypt(
    encrypted.getCipherTextString(),
    encrypted.getKeyMaterial(),
    SingleJobDecryptionMode.SINGLE_JOB_KEY_HEADERLESS
);
```

Note above the new mode passed into the encryption method to avoid writing a header. Since CLEAR expects a header by default, the decryption step also includes a mode indicating no header will be present.

CLEAR has an operating mode that embeds an HMAC. The HMAC will be directly embedded int the hyperkey and would normally be utilized to verify the authenticity of a message. For example, to confirm that an email had not been tampered with. Here is an example of encrypting and decrypting with an HMAC:

```
CLEARString cs = CLEAR.clearString();
CLEARResult encrypted = cs.encrypt(
    "Hello world",
    CLEAR.STRENGTH_2048_BIT,
    SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_WITH_HMAC
);
CLEARResult decrypted = cr = cs.decrypt(
    decrypted.getCipherTextString(),
    decrypted.getKeyMaterial(),
    SingleJobDecryptionMode.SINGLE_JOB_KEY
);
```

Note how the new mode `SINGLE_JOB_SYSTEM_GEN_KEY_WITH_HMAC` was utilized during encryption. HMAC requires a header to be present, so this mode (and compliance mode) cannot be used in Headerless operation.

CLEAR may also operating a mode enforcing FIPS-140-2 compliance, as well as a mode including both an HMAC and FIPS. Please see "Advanced Topics - Operating Modes" for details.

3.6 File-Based Encryption

The File interface is a convenience method built on top of the root binary interface and is designed to make encrypting filesystem resources more convenient. CLEAR can a file on any system that can host a JVM. Note that only individual files and not folders are covered by this API. You may choose to write a small amount of code to traverse a directory if you need to encrypt all the contents, each outputing an encrypted file, or you may choose to compress, ala ZIP, the folder into a single encrypted resource.

```
File file = new File("./test_file");
CLEARFile cf = CLEAR.clearFile();
CLEARResult encrypted = cf.encrypt(
    file.getAbsolutePath(),
    CLEAR.STRENGTH_2048_BIT,
    StreamingEncryptionSysGenKeyMode.STREAMING_SYSTEM_GEN_KEY
);
File encryptedFile = new File(encrypted.getCipherTextFile());
CLEARResult decrypted = cf.decrypt(
    encrypted.getCipherTextFile(),
    encrypted.getKeyFile(),
    StreamingDecryptionMode.STREAMING_KEY
);
File decryptedFile = new File(decrypted.getClearTextFile());
```

Whereas the Strings interface is obtained by a call to `CLEAR.clearString()`, accessing a reference to the File interface instead uses `CLEAR.clearFile()`. Also node that the File API does not have dedicated enums for specifying the operating mode. Rather, it leverages the Stream API under the hood to perform the encryption to achieve low-memory overhead, even when encrypting or decyrpting an arbitrarily large file.

> Note that the File API uses a String (representing the URI of the file) rather than a `java.io.File` instance.

Similarly to the Strings interface, File supports Headerless and HMAC operation. Examples of both follow:

```
// Headerless operation
File noHeader = new File("./test_file_headerless");
CLEARResult encryptedNoHeader = cf.encrypt(
    noHeader.getAbsolutePath(),
    CLEAR.STRENGTH_512_BIT,
    StreamingEncryptionSysGenKeyMode.STREAMING_SYSTEM_GEN_KEY_HEADERLESS);
CLEARResult decryptedNoHeader = cf.decrypt(
    encryptedNoHeader.getCipherTextFile(),
    encryptedNoHeader.getKeyFile(),
    StreamingDecryptionMode.STREAMING_KEY_HEADERLESS
);

// HMAC operation
File hmac = new File("./test_file_hmac");
CLEARResult encryptedHmac = cf.encrypt(
    hmac.getAbsolutePath(),
    CLEAR.STRENGTH_512_BIT,
    StreamingEncryptionSysGenKeyMode.STREAMING_SYSTEM_GEN_KEY_WITH_HMAC
);
CLEARResult decryptedHmac = cf.decrypt(
    cr.getCipherTextFile(),
    cr.getKeyFile(),
    StreamingDecryptionMode.STREAMING_KEY
);
```

In each example above, the relevant mode is passed to the encryption step. For Headerless decryption, no header exists in the ciphertext, CLEAR must be told to expect a Headerless file. In HMAC decryption, the header does exist and will automatically indicate that an HMAC is present for decryption.

3.7 Stream-Based Encryption

The Stream API allows for lower-level encryption of byte arrays (binary data). Both the File and String APIs in effect are convenience wrappers on top of the Stream SDK. A typical use case for streaming is to process large amounts of data (say, from a web socket or the filesystem) in a performant, low-memory fashion. Note that while the Stream API is extremely powerful and flexible, it also adds additional complexity to your code. As such, in this chapter, we will write some convenience wrappers that demonstrate typical stream usage with Java's InputStream and OutputStream.

When encrypting a stream, the following steps will normally occur:

1. Invoke a starting operation for the first packets
2. Invoke a continue operation for the next several packets
3. Invoke a finalize operation for the last packet

CLEAR utilizes a separate method to start encryption or decryption, and the same method for continuing and finalizing the stream. You differentiate between the two with a boolean 'terminate' flag (set to false when continuing and to true when finalizing).

If you are simply encrypting or decrypting a relatively small amount of data (that would fit comfortably in memory), you can use a simple set of operations to encrypt and decrypt using minimal code. This use case will look very similar to the Sring and File examples above:

```
byte[] bytes = "Hello World".getBytes();
CLEARStream cs = CLEAR.clearStream();
CLEARResult encrypted = cs.encryptStart(
    bytes,
    CLEAR.STRENGTH_512_BIT,
    StreamingEncryptionSysGenKeyMode.STREAMING_SYSTEM_GEN_KEY,
    true
);
CLEARResult decryptedNoHeader = cs.decryptStart(
    encrypted.getCipherText(),
    encrypted.getKeyMaterial(),
    StreamingDecryptionMode.STREAMING_KEY,
    true
);
System.out.println(new String(decrypted.getClearText()));
```

Since we passed the final termination flag as true for both `startEncryption()` and `startDecryption()`, only a single line of code is requried to encrypt or decrypt. Normally, streams will contain more data than will scale in memory, and a more complicated use of the Stream API will be needed. To explore how to handle these use cases, let's write wrappers that utilize traditional Java `InputStream` and `OutputStream`.

Encryption

Let's now explore our wrapper class:

```
import com.clear.cryptosystem.*;
import com.clear.cryptosystem.CLEARMode.*;
import java.io.*;

public class CLEAREncrypter {

    private final int keySize;

    private final StreamingEncryptionSysGenKeyMode mode;

    public CLEAREncrypter(int keySize, StreamingEncryptionSysGenKeyMode mode) {

        this.keySize = keySize;
        this.mode = mode;
    }

    public CLEARResult encrypt(InputStream in, OutputStream out)
        throws IOException {

        CLEARStream cs = CLEAR.clearStream();
        byte[] buffer = new byte[64];
        int bytesRead = in.read(buffer);
        CLEARResult header = null;
        if (bytesRead > -1) {
            header = cs.encryptStart(buffer, keySize, mode, false);
            out.write(header.getCipherText());
            bytesRead = in.read(buffer);
        }
        while (bytesRead > -1) {
            byte[] clearBuffer = new byte[bytesRead];
            System.arraycopy(buffer, 0, clearBuffer, 0, bytesRead);
            boolean terminate = bytesRead < buffer.length;
            CLEARResult result = cs.encryptContinue(clearBuffer, header.getJobNumber
            ↪  (), mode, terminate);
            out.write(result.getCipherText());
            bytesRead = in.read(buffer);
        }
        out.flush();
        return header;
    }
}
```

As can be seen above, the utility class that we have written above is initialized with a key strength and key generation mode. This type can be resused across threads and invocations. The `encrypt()` method takes an `InputStream` containing the plaintext and an `OutputStream` that will receive the ciphertext. Note for the first packet, we invoke `encryptStart()` . For all other packets except the last, we utilize `encryptContinue()` with the terminate flag set to false. Finally, we end the encryption on the last packet with `encryptContinue()` and the terminate flag set to true. The final call to `flush()` ensures the `OutputStream` receives all the encrypted bytes. (Most implementations implicitly call `flush()` on `close()` , but there is no harm being explicit).

Decryption

Now, let's see how decryption works:

```java
import com.clear.cryptosystem.*;
import com.clear.cryptosystem.CLEARMode.*;
import java.io.*;

public class CLEARDecrypter {

    private final String keyMaterial;

    private final StreamingDecryptionMode mode;

    private final long jobNumber;

    public CLEARDecrypter(final String keyMaterial, StreamingDecryptionMode mode,
    ↪  final long jobNumber) {

        this.keyMaterial = keyMaterial;
        this.mode = mode;
        this.jobNumber = jobNumber;
    }

    public void decrypt(InputStream in, OutputStream out)
        throws IOException {

        CLEARStream cs = CLEAR.clearStream();
        byte[] buffer = new byte[64];
        int bytesRead = in.read(buffer);
        CLEARResult header = null;
        if (bytesRead > -1) {
            header = cs.decryptStart(buffer, keyMaterial, mode, false);
            out.write(header.getClearText());
            bytesRead = in.read(buffer);
        }
        while (bytesRead > -1) {
            byte[] clearBuffer = new byte[bytesRead];
            System.arraycopy(buffer, 0, clearBuffer, 0, bytesRead);
            boolean terminate = bytesRead < buffer.length;
            CLEARResult result = cs.decryptContinue(clearBuffer, header.getJobNumber
            ↪  (), mode, terminate);
            out.write(result.getClearText());
            bytesRead = in.read(buffer);
        }
        out.flush();
    }
}
```

Decryption works in a nearly identical fashion to encryption with calls to `decryptStart()`, `decryptContinue()` with terminate set to false and finally `decryptContinue()` with finalize set to true. Whereas we passed in the key size and mode to for encryption, decryption instead requires the key material to use and a mode.

Job Numbers and Key Materials One subtelty that may not be apparent in the code examples above is the use of job number and key material and when to retrieve them. CLEAR will populate both the initial job number (one for encryption and one for decryption) during the calls to `encryptStart()` and `decryptStart()`. Your code will in the decryption phase need to resupply the correct job number during invocation. You can retrieve the job number from `getJobNumber()`.

In addition, the very first `CLEARResult` from `encryptStart()` will contain the generated key to use for decryption (if you are using system-generated key mode). You will need to store that key for later use when decrypting. Here is an example putting are helper classes to use to encrypt and then decrypt a file while streaming:

```
public static void main(String... args) throws IOException {

     CLEARResult result = null;
     File source = new File("clear-java.iml");
     File encrypted = new File("clear-java.iml.clear");
     File decrypted = new File("clear-java.iml.copy");
     try(FileInputStream in = new FileInputStream(source);
         FileOutputStream out = new FileOutputStream(encrypted)) {
         CLEAREncrypter encrypter = new CLEAREncrypter(
             512,
             StreamingEncryptionSysGenKeyMode.STREAMING_SYSTEM_GEN_KEY
         );
         result = encrypter.encrypt(in, out);
     }
     try(FileInputStream in = new FileInputStream(encrypted);
         FileOutputStream out = new FileOutputStream(decrypted)) {
         CLEARDecrypter decrypter = new CLEARDecrypter(
             result.getKeyMaterial(),
             StreamingDecryptionMode.STREAMING_KEY,
             result.getJobNumber()
         );
         decrypter.decrypt(in, out);
     }
 }
```

The code above happens to encrypt the VS Code configuration file in the same environment as the helper classes, but this could be any arbitrary file (or stream, for that matter). Note how the result returned from the encryption step contains both the key material and the job number to pass to the decryption step.

Note that the Stream API also supports Headerless, HMAC, FIPS Compliance and HMAC with FIPS Compliance modes, just like the String and File APIs (see "Advanced Topics - Operating Modes" for details).

Congratulations! You can now use some of the strongest, fastest, and (with streaming) lowest memory overhead cryptography on the planet.

Random numbers should not be generated with a method chosen at random.
Donald Knuth

4 Advanced Topics

4.1 KeyTool and Access Control Lists

KeyTool generates standalone keys outside of an encryption operation. This use case occurs frequently in systems that need to generate keys that will subsequently be used for multiple encyrption and decryption operations. KeyTool instances, which are thread-safe, are obtained from the CLEAR factory, just as the String, File and Streaming interfaces are obtained. Below is a demonstration of common KeyTool operations:

```
String password = ...
byte[] material = ...

// Obtain a reference to the KeyTool generator
CLEARKeyTool keyTool = CLEAR.clearKeyTool();

// Create a user-supplied key directly
String usk = keyTool.genKey(512);

// Create a user-supplied key with multi-factor material
String uskMFA = keyTool.genKeyWithMFA(512, material);

// Create a user-generated key from a password
String ugk = keyTool.genUGK(password);

// Create a user-generated key from a password and MFA
String ugkMFA = keyTool.genUGKWithMFA(password, material);
```

> Note that KeyTool also contains a number of convenience methods to generate keys and access control lists from files rather than programatically. This allows for scripting key and ACL generation in bulk. Please refer to the Javadocs for detailed information.

4.2 Operating Modes

CLEAR allows for several additional operating modes beyond standard encryptoin and decryption. Your license may or may not allow for the use of these additional features. If you would like to upgrade to leverage these additional modes, please contact Quantum Knight sales for details.

Headerless Operation

The CLEAR default mode of operation will produce a ciphertext that is only eight [8] bytes larger than the original data. When running in HMAC mode, the total additional encumbrance to ciphertext is eighty [80] bytes.

Adding 8 bytes of data to something might not seem like much. If you're encrypting a 3GB file, would you really notice an additional 8 bytes?

If your data comes from a small IoT sensor and is only sending packets of 16 bytes, then adding 8 more bytes could be a really impactful matter. This is why we have "CLEAR Headerless Mode."

Encrypting data in CLEAR Headerless Mode adds exactly zero [0] additional bytes of overhead to ciphertext, making it ideal for use with sensors and IoT/IIoT devices. As a caveat, Headerless Mode encryptions cannot be combined with HMAC/AEAD signed encryptions (see below).

Encrypting in Headerless Mode Encrypting a String in Headerless mode is almost identical to encrypting in standard (basic) mode. To use Headerless mode during encryption, select `SINGLE_JOB_SYSTEM_GEN_KEY_HEADERLESS` for the mode selection.

```
CLEARResult encrypted = cs.encrypt(
    "Hello world",
    CLEAR.STRENGTH_512_BIT,
    SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_HEADERLESS
);
```

Encryption in Headerless mode is intended for environments (like IoT) where eight additional bytes of data encumbrance is problematic or material to a use case. Even in large database environments where one might not necessarily view eight bytes as a material concern; strategies that use CLEAR to encrypt every VARCHAR in a database may elect to use Headerless mode when every character matters.

> Important note: In cases (like IoT) where each bit of data is counted and padding is infeasible, the Quantum Knight team recommends using CLEAR-Streaming SDK (also in Headerless mode). CLEAR-String SDK will add space to ciphertext by way of UTF-16/Unicode and Base-64 encoding of String data. The byte[] binary interface in CLEAR-Streaming SDK allows for ciphertext that exactly equals the original plaintext length.

Decrypting in Headerless Mode Decrypting a String that was encrypted in CLEAR Headerless mode requires decryption in Headerless mode as well. Failure to match modalities in encryption and decryption will result in a failure to decrypt. To indicate this for the deciphering portion, specify `SINGLE_JOB_KEY_HEADERLESS`.

```
CLEARResult decrypted = cs.decrypt(
    encrypted.getCipherTextString(),
    encrypted.getKeyMaterial(),
    SingleJobDecryptionMode.SINGLE_JOB_KEY_HEADERLESS
);
```

Since Headerless mode, by definition, has no header in the ciphertext, HMAC and Compliance modes cannot be used. See below for details on both.

HMAC

CLEAR incorporates Authenticated Encryption with Associated Data ("AEAD") with Hash based Message Authenticate Code ("HMAC") as a mechanism that supports a form of signed encryption that is resistant to malicious injection and tampering. AEAD is a variant of Authenticated Encryption ("AE") that allows a recipient to check the integrity of both the encrypted and unencrypted information in a message. AEAD binds associated data (AD) to the ciphertext and to the context where it is supposed to appear so that attempts to "cut-and-paste" a valid ciphertext into a different context are detected and rejected.

During decryption operations, signed AEAD/HMAC authentication will compare the HMAC found near the prefix of the ciphertext with a freshly generated HMAC that is produced by decryption. If the Message Authentication Codes ("MAC") do not match exactly, then the signature is determined to be invalid and thus implies a possible tampering of the ciphertext.

```
CLEARResult encrypted = cs.encrypt(
    "Hello world",
    CLEAR.STRENGTH_512_BIT,
    SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_WITH_HMAC
);
```

Encrypting in HMAC with AEAD is almost identical to encrypting in standard (basic) mode.

To use HMAC during encryption, select `SINGLE_JOB_SYSTEM_GEN_KEY_WITH_HMAC` for the mode selection.

> Important note: Encrypting in HMAC mode requires more computational overhead than non-HMAC based encryptions. HMAC Mode will add 72 bytes of additional data to the ciphertext output.

Decrypting a String that was encrypted in HMAC mode is entirely identical to the standard decryption operations described previously. CLEAR automatically detects HMAC in the ciphertext header and will attempt to validate HMAC authentication whenever it exists.

```
CLEARResult decrypted = cs.decrypt(
    encrypted.getCipherTextString(),
    encrypted.getKeyMaterial(),
    SingleJobDecryptionMode.SINGLE_JOB_KEY
);
```

CLEAR automatically detected the need to locate and decrypt the HMAC from the header mode specified when encrypting above.

> Important note: Failed HMAC authentication (during decryption) will result in a failure to decrypt with and error message and null response output.

As can be seen from the above code examples, including an HMAC solely requires switching to a different operating mode during encryption (whether string, file or stream). Decrypting requires no changes to code, as the header will indicate that an HMAC is present. Since a header is required to utilize HMAC, one cannot use the feature in Headerless mode.

FIPS Compliance

FIPS 140-2 (Federal Information Processing Standard 140-2) is a U.S. government standard established by the National Institute of Standards and Technology (NIST) to ensure the security and strength of cryptographic modules used in hardware and software products.

The standard provides a rigorous set of requirements and guidelines for cryptographic modules and their underlying algorithms to protect sensitive information. FIPS 140-2 validation is the process of testing and certifying a cryptographic module against the FIPS 140-2 standard.

The validation is carried out by NIST-accredited Cryptographic Module Testing (CMT) laboratories. Upon successful validation, the tested product is added to the NIST's Cryptographic Module Validation Program (CMVP) list, indicating that the product is compliant with FIPS 140-2 requirements.

FIPS 140-2 module validation for CLEAR Cryptosystem can be found here:

https://csrc.nist.gov/projects/cryptographic-module-validation-program/certificate/3080

Key-wrapping is a technique used to protect cryptographic keys by encrypting them with a strong key encryption algorithm. AES-256 (Advanced Encryption Standard with 256-bit keys) is one of the approved algorithms for key-wrapping in FIPS 140-2. Using key-wrapping can help extend compliance to non-validated cryptographic modules by encapsulating sensitive key material within the secure boundaries of a FIPS 140-2 validated module.

By following this approach, the sensitive key material remains protected by the FIPS 140-2 validated module, and the non-validated module only handles encrypted (wrapped) keys, thereby extending a level of FIPS 140-2 compliance to the non-validated module. However, it is crucial to note that this strategy does not make the non-validated module fully FIPS 140-2 compliant.

CLEAR is packaged with the Crypto-Compliance CCJ JAR as an optional-use accessory to support total FIPS 140-2 compliance as a validated module. When added to the classpath of CLEAR and executed in compliance mode, AES encryption is injected into the inner core of the CLEAR encipherment process. Irrespective of the use of the CCJ JAR, CLEAR applies AES-256 key-wrapping as a post-processing step whenever executed in compliance mode.

Encrypting in Compliance Mode

Encrypting a String in Compliance mode is almost identical to encrypting in standard (basic) mode. To use Compliance mode during encryption, select `SINGLE_JOB_SYSTEM_GEN_KEY_WITH_COMPLIANCE` for the mode selection.

```
CLEARResult encrypted = cs.encrypt(
    "Hello world",
    CLEAR.STRENGTH_512_BIT,
    SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_WITH_COMPLIANCE
);
```

> Important note: Encrypting with CLEAR in compliance mode will result in the generation of a key format that is different to non-compliance modes. In CLEAR File-based encryption, the difference is most easily recognizable as being saved under a different file extension. E.g. ".ckey" versus ".ccom" versus ".ccomf". Please see the CLEAR-File SDK documentation for more detail. In this case of CLEAR-String SDK, however; compliance-mode key output is largely indistinguishable from non-compliance mode output.

The results from compliance operations cannot be directly observed. For this reason, it is very important to take care and store keys in a location and such a way that their type is remembered for future use. Keys made in compliance modes can only be decrypted using decryption operations being run (similarly) in compliance mode.

Decrypting in Compliance Mode

Decrypting a String that was encrypted in CLEAR Compliance mode requires decryption in compliance mode. Further, if CCJ JAR was used during the encryption process, it must be used in the decryption process as well. Ciphertext encrypted with CCJ may not be decrypted without CCJ, and vice versa.

```
CLEARResult decrypted = cs.decrypt(
    encrypted.getCipherTextString(),
    encrypted.getKeyMaterial(),
    SingleJobDecryptionMode.SINGLE_JOB_KEY_WITH_COMPLIANCE
);
```

In contrast to HMAC decryption, for Compliance to decipher successfully, a separate mode of `SINGLE_JOB_KEY_WITH_COMPLIANCE` must be supplied.

Note that you may combine HMAC with Compliance. Both encryption and decryption will specify this combination of both modes operating at once. Use `SINGLE_JOB_SYSTEM_GEN_KEY_WITH_HMAC_AND_COMPLIANC` for encryption and decryption.

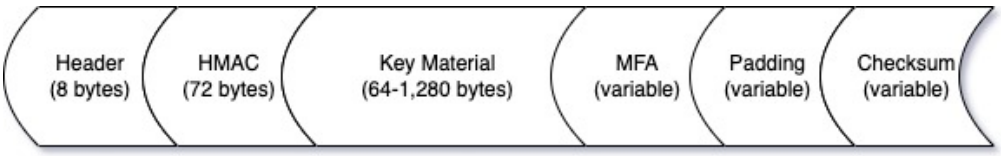
4.3 Hyperkeys

HyperKey ™ is CLEAR's patented symmetric encryption and decryption key. But so much more! While traditional keys solely contain the secret(s) needed for a given cryptographic algorithm, a HyperKey allows the user to perform a variety of additional, innovative functions **directly within the key itself**! These currently include:

- Headers indicating the type of encryption performed to reduce the amount and brittleness of code used for subsequent decryption.
- Storing a second-factor for multi-factor authentication (biometric, hardware or other).
- Including access control lists, which combined with the MFA, allow for selective encryption and decryption of portions of a resource or message.
- Randomizing the key layout to prevent tampering and even guessing the size of the key in use.
- Validating the authenticity of a message with the inclusion of an embedded HMAC.

Key Layout

The following diagram conceptually depicts the areas of data allocated within a given HyperKey. Note that this is not a formal specification of the fields within a hierarchy (or even their actual ordering). Rather, the illustration below shows the types of data stored within a given key. Since CLEAR will randomize the layout of a given key and include both padding and checksum data, even generating a key with the same entropy would lead to different result from invocation to invocation.



The various Operating Modes dictate which portions will be included in a given key. For example, one of the HMAC modes must be selected during encryption to have that segment included. Similarly, MFA must be specified at key creation to have that element reside in a given head. Operating in Headerless mode will result in a key with solely key material, padding and checksum elements. The default HyperKey will have the same but also include the standard 8-byte header.

MFA and Access Controls

Multi-factor authentication pairs with access control lists (ACLs) in a HyperKey. A given ACL specifies which portions of a message can be decrypted by which actors/principals. These users or systems are identified by the MFA supplied in a given key.

For example, Alice has an ACL denoting that she can decrypt the portion of the document relevant to Finance. Bob has an ACL indicating he can access Human Resources. Still other portions of the same document would decrypt for anyone with the key. Alice and Bob are identified by their MFA material passed in during decryption. CLEAR will mix

both the key material and the MFA to ensure that only the correct enttiy can decrypt a given portion protected by access control.

Another use case for this capability would be to turn an untrusted third-party provider, such as a Cloud SaaS provider, into a zero-trust environment. One could upload both the HyperKey and encrypted data to AWS or Azure. As long as the data is protected by ACLs and requires MFA, which the cloud provider would not pssess, then decryption cannot occur.

HMAC and Message Authenticity

4.4 Pluggable Random Number Generators

There are only relatively few publicly available Symmetric encryption ciphers in the world today. The ones that can be consumed in Java are typically already included within the Java Runtime Environment itself, made available for use by developers as part of the Java Cryptography Extension (JCE) interface. One may also find encryption ciphers online at "the legion of the bouncy castle", via https://www.bouncycastle.org.

A common thread amongst the open-source symmetric ciphers is the way in which they are instantiated programmatically. In many cases, user-level encryptions begin when an operator generates some form of password or passphrase (and optional salt) that will ultimately become their encryption key. The initialization sequence for most symmetric ciphers (at least in the JCE) is one that requires an initialization vector (IV), a salt, and a secret key. These implementations vary subtly by cipher; however, they typically involve the programmatic generation of entropy of 256 bits or less.

As it happens, mishandling of the initialization-phase entropy (as mentioned above) is one of the larger contributing factors to weakness in encryption implementations. For example, it wouldn't be AES-256 itself that provides a weakness (algorithmically); rather, it is more frequently poor user implementation or hard-coding of a salt or IV that introduces weakness into security systems. Key storage, secure vaulting, and proper key-handling in symmetric encryption is extremely important.

Perhaps you're now asking yourself the question, "How can RNG produced by CSRNG being considered 'Quantum Strength' in the CLEAR Cryptosystem?" If that is the case, we're going to get to the answer to that important question on the very next page.

Professional cryptanalysis and academic peer review applied to CLEAR over multiple rounds and multiple years have conclusively come back with the same consistent results.

> "The length of the key material ('key-space') is mathematically equivalent to the bit-strength of the encipherment."

In other words, 512 bits of key material means definitive production of a 512-bit encipherment.

Quantum-Computing and Keys
Grover's Algorithm was developed in 1996 by Lov Grover as a mechanism for intended for quadratic performance boost of searching through an unsorted database. Ostensibly, an unintended consequence of Grover's Algorithm was its ability to reduce the time complexity of brute force attacks on round-reduced block ciphers.

With the utmost respect for the Advanced Encryption Standard (AES) and 256-bit security (drafted in the late 1990's and ratified by NIST in 2001), we are now looking at the need for new solutions that can contend with computational factors that did not exist when AES was created.

When CLEAR produces 512 bits of CSRNG as the "fuel" (and starting place) for its encryption operation; the net result is encryption that can be measured at 512 bits of security protection. As we together enter the next generation of cyber security challenges, we are facing the power of qubit computers as well as the advancement of AI algorithms that capable of round-reducing encryption ciphers.

Plug-n-Play Random Number Generator

OK – so the fate of your security rests in the hands of the Java Secure Random number generator? No, definitely not. Certainly, the easiest way to use CLEAR is to run it with as few parameters as possible and let the default onboard CSRNG do the work of producing your initial entropy, the "fuel" for encryption. However, this alone does not allow for the ultimate in flexibility.

We understand that enabling plug-and-play RNG is the best way to build trust and give you full control over the input into your encryption jobs. An essential feature of CLEAR is the ability to plug-in your own entropy source (random number generator). The "... if you don't like ours, use your own." ethos is core to the way we see CLEAR being used to guarantee absolute data privacy and true cybersecurity.

CLEAR has been constructed with one "built-in" plugin for external RNG, as well as an open-SDK interface for adding RNG input from external sources. The built-in plugin allows CLEAR to source entropy from a reputable online and open-source provider: https://www.random.org.

CLEAR is also designed to work with external hardware-based random number generators. Often referred to as "quantum random number generators" or "QRNG", products such as ID-Quantique4 may be directly integrated into CLEAR via localized USB connection or on-a-chip (motherboard) type connection.

Programmatically, changing the random number generator in CLEAR can be done with one simple line of code:

```
CLEAR.setPluggableRNG(
    RandomNumberGenerator.RNG_CSPRING_SECURE_RANDOM,
    apiKey
);
```

CLEAR supports the following options for plugging in an RNG implementation:

Method	Purpose
RNG_CSPRNG_SECURE_RANDOM	CSpring RNG implementation
RNG_QRNG_ID_QUANTIQUE_USB_BASED	USB based quantum RNG
RNG_QRNG_ID_QUANTIQUE_ONBOARD_CHIP_BASED	Chip based quantum RNG
RNG_TRNG_EXTERNAL_PROVIDER	External Traditional RNG provider