UNIVERSITÄT
BERN

FluentCrypto

A Fluent Wrapper for the NodeJS Crypto API

Bachelor Thesis

Simon Kafader
from
Belp BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

31. July 2020

Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari

Software Composition Group
Institut für Informatik

University of Bern, Switzerland

Abstract

NodeJS Developers often struggle with cryptographic tasks. This not only leads to bugs but also to security issues. In this work, we aim to mitigate these issues by providing a wrapper around the NodeJS Crypto API. This wrapper is based on the design principles of fluent APIs and has built-in validation, which is based on CryRule rule files, a domain-specific language we explicitly create for this purpose.

Contents

1 Introduction

Since the introduction of NodeJS in 2009 which opened the possibility of using JavaScript in server environments, JavaScript has grown steadily in popularity and has become one of the most used programming languages for server-side applications. Meanwhile, the development of the web has led to applications and their underlying devices storing increasing amounts of data, often of sensitive nature. To protect this data, cryptography is used. This also means that developers operating under these conditions not only have to implement the business logic but also have to know and select the correct cryptographic algorithms and apply these algorithms in a correct and secure way.

Several studies such as the one from Hazhirpasand et al. have shown that developers issued with such tasks often struggle with the usage of the cryptographic APIs [5]. Consequently, these developers struggle to apply cryptographic concepts, leading to non-working or insecure applications. To mitigate such issues in NodeJS, we introduce FluentCrypto, an easy to use fluent interface with built-in usage validations that forms a wrapper around the NodeJS Crypto API.

1.1 Motivation

Consider the task “Encrypt all persistent data in our system with a private key”. Code that solves this task could look like in figure 1.1.

On line 3, a certain cipher algorithm that we want to use is specified. A cipher is an algorithm to encrypt and decrypt data that follows a well-defined sequence of steps. The chosen cipher algorithm will need an initialization vector of length 8 (a string with 8

Figure 1.1: Insecure usage of NodeJS crypto (encryption)

```
1 const crypto = require('crypto');
2
3 const algorithm = 'des';
4 const key = crypto.scryptSync(process.env.PRIVATE_KEY, 'salt', 8);
5 const iv = Buffer.alloc(8,0);
6
7 const cipher = crypto.createCipheriv(algorithm, key, iv);
8
9 let encrypted = cipher.update('first part to encrypt', 'utf8', 'hex');
10 encrypted += cipher.update('second part to encrypt' , 'utf8', 'hex');
11 encrypted += cipher.final('hex');
12
13 storeEncryptedResultSomehow(encrypted, key, iv);
```

characters, where each character is 1 byte with 7 bits reserved for the character and 1 parity bit, or a buffer of 8 bytes) and a key of length 8 (a string with 8 characters or a buffer of 8 bytes) to work.

On Line 4, a key of length 8 is created from a password stored in a process variable and a hardcoded salt, using the *scryptSync* function. A (cryptographic) key is a piece of information (e.g in form of a string or a buffer) that is used to encrypt and decrypt data and can either be stand-alone (for symmetric encryption, meaning both the encryption and decryption are performed with the same key) or part of a key-pair where the private key is used to encrypt data and the corresponding public key is used to decrypt this data. A salt is a unique, randomly generated piece of data that is added to the input (mostly passwords) before hashing it to ensure that the same input does not always result in the same hash value, thus reducing the risk of pre-computed dictionary attacks (a form of brute force attack where you have a large database of common, hashed passwords and their original value to compare) and making it harder to crack large numbers of hashes since there are more unique hash values. It should be of at least length 16 bytes[1]. In our example, the *scryptSync* function takes a salt as argument to variate keys that are built using the same password. This way, you can derive multiple distinct keys from one password (assuming that your salt is randomly generated for each key).

On the next line, an initialization vector of length 8 which consists only of zeros is created. An initialization vector (IV) is a randomly generated piece of information (e.g in form of a buffer) that is used in iterating cryptographic processes, such as cipher algorithms. It ensures that similar messages encrypted with the same key result in different, unique encrypted values, thus hiding patterns in the encrypted data. How this is achieved depends on the cryptographic process. E.g. block ciphers (except cipher modes

[1]https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf

Figure 1.2: Insecure usage of NodeJS crypto (decryption)

```
1 const crypto = require('crypto');
2
3 const algorithm = 'des';
4 const key = getStoredKey()
5 const iv = getStoredIV();
6
7 const decrypter = crypto.createDecipheriv(algorithm, key, iv);
8
9 const encrypted = somehowGetEncryptedData();
10
11 let decrypted = decrypter.update(encrypted, 'hex', 'utf8');
12 decrypted += decrypter.final('utf8');
```

operating in ECB-mode which do not operate on IVs) add (via XOR) the initialization vector to the first block of the input data. An IV does not have to be secret, but it must not be reused.

On line 7, a cipher object with the algorithm *des* and the created key and initialization vector is initialized. This cipher object then receives data to encrypt with the *update* function on line 9. This function encrypts the received data and returns the encrypted result. The result is stored in a variable. The second argument of the *update* function, "utf8", tells the NodeJS Crypto API that it can expect the input data to be a string encoded in "utf8". The third argument, "hex", tells the NodeJS Crypto API that it should encrypt and return the data in hex encoding. If the first encoding was not set, the NodeJS Crypto API would by default assume that the input data is not a string but a buffer. If the second encoding was not set, the NodeJS Crypto API would by default assume that it should return a buffer. This process is repeated to encrypt a second string on line 10. The encodings must be set here too, otherwise the NodeJS Crypto API would by default assume that the input is not a string but a buffer. The result is added (concatenated) to the same variable as on line 9.

It is possible that not all encrypted data has been returned by the *update* functions. Therefore, on line 11, *final* is called. This function returns any remaining encrypted data on the cipher object and closes it for further calls. It is again necessary to specify the output encoding, otherwise by default a buffer would be returned. The result of this function is added (concatenated) to the same variable again.

Figure 1.2 shows the decryption process that decrypts the data encrypted in figure 1.1. To decrypt data encrypted with a cipher, a decipher object must be created with the same algorithm, key and initialization vector as for the cipher object. On line 3, the same algorithm as for the cipher object is chosen. Line 4 shows that the same, stored key that was used for the encryption is retrieved. On line 4, the initialization vector is retrieved in the same way.

On line 7, the decipher object that can decrypt data encrypted with the cipher object in figure 1.1 is initialized. This decipher object then receives the encrypted data on line 11 with the *update* function. This function decrypts the input data and returns the decrypted result. The second argument tells the NodeJS Crypto API that it can expect the received data to be a string in "hex" encoding. The third argument tells the NodeJS Crypto API to return the decrypted result as a string in "utf8" encoding. The decrypted result is stored in a variable. It is possible that not all decrypted data has been returned by the *update* function. Therefore, on line 12, the *final* function is called. This function returns any remaining encrypted data on the decipher object and closes it for further calls. It is again necessary to specify the output encoding, otherwise by default a buffer would be returned. The result of this function is added (concatenated) to the same variable.

While this example is working fine and looks simple and straightforward, small changes could render it non-functional. For example, if the encodings were not set, the code would result in the following error message:

```
error:0606506D:digital envelope routines:EVP_DecryptFinal_ex:
wrong final block length
```

In our example, this error happens because by default the input would be treated as a buffer and not as a string, but in this case the input is a string in the "utf8" encoding. For the same reason, this code would also result in an error if we left out a single specification of an encoding in lines 10 and 11.

Another potential issue lies with the key. We could use the password that is stored in the process variable directly as our key. But if this password of type string would have any other length than 8, (e.g. "VerySecretKey", a string of length 13) this would result in an error because it would be of the wrong length for the chosen algorithm. In this example, this issue is already mitigated as the key used is derived via *scryptSync*.

Our examples can also not be considered secure. Neither the initialization vector nor the salt are random but a hardcoded string or buffer. The chosen algorithm "des" can also not be considered secure.

An issue of the API design is the separation of the key generation from the algorithm that uses the key. The algorithm choice defines the characteristics of the key that can be used. The key generation process, however, is completely separated from the chosen algorithm and the encryption process. In our example, additional knowledge is required. The user has to know that she/he needs a key and an initialization vector for the chosen algorithm. Then the user needs to know the required length of the key or the initialization vector (both 8 in the example). There is no direct way to generate an initialization vector or a matching key from a password that fit the chosen algorithm.

Another issue with the NodeJS crypto API is that of the cryptic error messages. The error message that result if the encodings were left out like in example 1.3 tells the user neither what exactly went wrong nor does it suggest a possible solution.

Figure 1.3: Left out encodings

```
let encrypted = cipher.update('first part to encrypt');
encrypted += cipher.update('second part to encrypt');
encrypted += cipher.final();
```

We will use this code example throughout the thesis to motivate the features and design decisions behind FluentCrypto.

1.2 Structure

In Chapter 2 we will first explore various existing studies and approaches on which we want to build. In chapter 3 we then present CryRule, a domain-specific language (DSL), very similiar to the already existing CrySL but compilable to JavaScript, with the goal of building a bridge between cryptography experts and NodeJS developers. Based on this language we then present FluentCrypto, a wrapper for the NodeJS Crypto API that should provide an easier to use interface with helpful error messages and secure default configurations. To achieve this, FluentCrypto provides its users with a fluent interface and uses the rules written in CryRule to safe check the used configurations and set thoughtful defaults. Chapter 4 presents an evaluation of this prototype where we carried out a survey in which participating developers solved different tasks using either the NodeJS Crypto API or FluentCrypto. We then discuss the results of this evaluation and come to a conclusion in chapter 5.

2
Related Work

2.1 Insecure Usage of cryptographic APIs

A study by Egele et al. on 11,748 applications that use cryptographic APIs found that 88% (10,327) of these applications make at least one mistake when it comes to applying cryptographic concepts [3]. Lazar et al. found in their case study that 83% of 269 cryptographic vulnerabilities have been caused by misuses of cryptographic libraries by developers [8].

2.2 API design

In their work, Green and Smith state that many misuses of cryptographic libraries originate in the developer having trouble understanding the API [4]. They criticize that while it has become accepted that systems should be user-friendly for the end user, a different attitude where the end-user is expected to be an expert prevails amongst cryptographic libraries. They then propose a set of characteristics that should lead to more secure cryptographic APIs, including:

- The API should be easy to use, even without documentation
- Incorrect use should lead to visible errors
- Defaults should be safe and never ambiguous

- Code that uses the API should be easy to read and update

Das et al. selected the most popular cryptographic libraries from C, C++, Java, Python and Go and examined them for properties that encourage cryptographic misuse by developers [2]. They used a set of common potential issues (such as initialization vector reuse, library defaults or incomplete features) for which they examined the libraries. Their work highlights the disconnect between the actual user of such a library and the user for whom it is designed. For each potential issue that they examine they also present best practices. Furthermore they introduced PyCrypto Linter, a proof of concept tool that is built into the library and logs any misuses in the same way as the standard library.

2.3 Static analysis

Egele et al. defined six rules for working with cryptographic APIs that must be satisfied to accomplish secure code [3]. Based on these rules they then developed a static analysis tool named CryptoLint. They then ran CryptoLint against 15'134 applications. The tool terminated successfully on 11'748 applications with 88% of them violating at least one of the six rules.

Krüger et al. built on previous studies that found misuses of cryptographic libraries [7]. To define secure usage of cryptographic libraries they presented CrySL, a domain specific language that serves as a bridge between cryptography experts and developers. CrySL enables cryptography experts to specify secure usage of libraries as rules. Based on this language, they implemented a compiler that parses these rules and builds meta objects from them to be used for static analysis. Based on this compiler they then introduced CogniCrypt, a static analysis tool for the Java Cryptography Architecture.

Hazhirpasand et al. developed a web platform named CryptoExplorer that provides real-world case studies with 3263 secure and 5897 insecure uses of the Java Cryptography Architecture [6]. They have integrated a pipeline that regularly mines more projects and uses CogniCrypt for the security analysis. In another study, they analysed 2324 open-source Java projects that rely on the Java Cryptography Architecture. They found that over 70% of these projects suffer from at least one misuse of the Cryptography API and that 40% of developers have always misused cryptographic APIs. [5]

These studies show that developers need support when working with cryptographic APIs. While many of the existing solutions use static analysis to detect misuses, few actually aim to prevent developers from these misuses while they are solving the cryptographic tasks. This is especially true when it comes to the design of cryptographic APIs. FluentCrypto aims to fill this gap by wrapping the existing NodeJS Crypto API and prohibiting insecure usage.

3
The Solution

In this chapter, we present FluentCrypto, a wrapper around the NodeJS Crypto API that aims to make cryptography in NodeJS easier to use and more secure. In subsection 3.1 we discuss the scope of FluentCrypto which results in a task-based interface that is demonstrated in subsection 3.2.2. We then go into further discussion about how FluentCrypto handles validation, defaults and error messages in the subsequent subsections. Section 3.3 presents CryRule, a domain specific language in which rule files for FluentCrypto can be written and that serves as a bridge between developers and cryptographic experts.

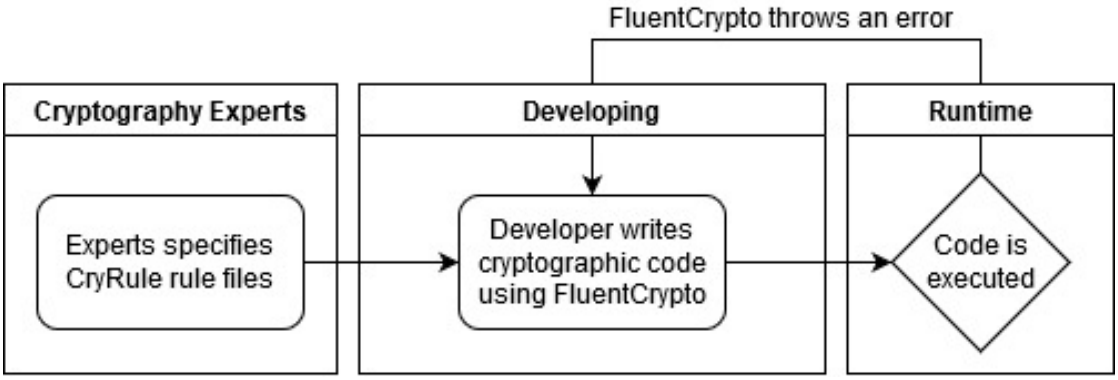

Figure 3.1: FluentCrypto Flow

Figure 3.2: Hash code example

```
1 const crypto = require('crypto');
2
3 const hash = crypto.createHash('sha512')
4 hash.update('text to hash', 'utf8');
5
6 const hashed = hash.digest('hex');
```

3.1 Scope

We wrote example code using the NodeJS Crypto API to explore the API. We followed the official documentation and its examples as closely as possible. While writing this code, we noted the obstacles we had and searched on Stack Overflow and in blog posts for potential security issues. To verify these potential security issues, we double checked with the CrySL rulefiles [1]. We also noted the steps we took while writing this code.

3.1.1 Hash

We chose to start with the smallest class (by number of exposed API calls) in the NodeJS Crypto API that can encrypt or hash data. This proved to be the Hash class. We then wrote example code that uses this hash class to hash data. An example for such code can be found in figure 3.2. For the hash class, the *update* function does not return anything and the *digest* function returns the hashed result. The steps we took that led to the code in figure 3.2 are:

- Create hash object (line 3)
- Select hash algorithm (line 3)
- Add data to hash to the hash object (line 4)
- Set the encoding of the input data (line 4)
- Perform the hashing (line 6)
- Set the encoding of the hashed result (line 6)

The security issues that we found for the hash class were related to the algorithm choice. Certain algorithms such as “md5” and “sha1” are considered broken in a cryptographic context and should not be used. A better algorithm choice, like the choice in the example, could be “sha512”.

[1] 31.03.2020 https://github.com/CROSSINGTUD/Crypto-API-Rules

Figure 3.3: Cipher code example

```
1 const crypto = require('crypto');
2
3 const algorithm = 'aes-256-cbc';
4 const salt = crypto.randomBytes(23);
5 const key = crypto.scryptSync(process.env.PRIVATE_KEY, salt, 32);
6 const iv = crypto.randomBytes(16);
7
8 const cipher = crypto.createCipheriv(algorithm, key, iv);
9
10 let encrypted = cipher.update('first part to encrypt', 'utf8', 'hex');
11 encrypted += cipher.update('second part to encrypt' , 'utf8', 'hex');
12 encrypted += cipher.final('hex');
13
14 storeResultWithKeyAndIV(encrypted, key, iv); //store data
```

3.1.2 Symmetric Encryption

After we covered the hash class, we wanted to cover a class that can encrypt and decrypt encrypted data. We chose the cipher class since it is the smallest encryption class (by number of exposed API calls) whose results can also be decrypted. Again we wrote example code that uses this cipher class to find potential obstacles and security issues. At first we wanted to use the *createCipher* call to create the cipher. We then noted that this function is marked as **deprecated** in the documentation with the hint to use *createCipheriv* instead, so we adapted our code accordingly. A code example can be found in figure 3.3.

In this example, we use a salt to derive a key of length 32 from a password stored in an environment variable (lines 4-5). A (cryptographic) key is a piece of information (e.g. in form of a string or a buffer) that is used to encrypt and decrypt data and can either be stand-alone (for symmetric encryption, meaning both the encryption and decryption are performed with the same key) or part of a key-pair where the public key is used to encrypt data and the corresponding private key is used to decrypt this data. A salt is a unique, randomly generate piece of data that is added to the input (mostly passwords) before hashing it to ensure that the same input does not always result in the same hash value, thus reducing the risk of pre-computed dictionary attacks (a form of brute force attack where you have a large database of common, hashed passwords and their original value to compare) and making it harder to crack large numbers of hashes since there are more unique hash values.

Subsequent, an initialization vector of length 16 for the chosen cipher algorithm is created on line 6. An initialization vector (IV) is a randomly generated piece of information (e.g. in form of a buffer) that is used in iterating cryptographic processes,

such as cipher algorithms. It ensures that similar messages encrypted with the same key result in different, unique encrypted values, thus hiding patterns in the encrypted data. How this is achieved depends on the cryptographic process. E.g. block ciphers (except cipher modes operating in ECB-mode which do not operate on IVs) add (via XOR) the initialization vector to the first block of the input data.

We noted the first potential obstacles while using this class. As there is no default choice of algorithm for the cipher, you have to choose the algorithm and its mode yourself. In example 3.3, we chose the algorithm "aes-256-cbc" which means that our algorithm "aes-256" performs in "cbc" mode. This means that the cipher algorithm works on fixed-length groups of bits (a block) and the mode tells the cipher algorithm how to apply the operation to the block. We then had to use a key of a certain length that fits the algorithm we chose to use. For our chosen algorithm "aes-256-cbc", this length is 32 bytes. The initialization vector also had to be of a certain length (16 in our example), depending on the chosen algorithm. Otherwise, we would not have been able to construct the cipher object on line 8. In our use cases, we only worked with string inputs. This resulted in the next obstacle we faced, we had to specify the input encodings when we wanted to provide string input to the *update* function. Otherwise, the NodeJS Crypto API would by default assume our input to be a buffer and would fail. A problem is also that we would get a rather cryptic error message:

```
error:0606506D:digital envelope routines:EVP_DecryptFinal_ex:
wrong final block length
```

We also had to specify the output encodings, otherwise the *update* function would have returned buffers, while we expected the results to be strings. Note that we have to store the initialization vector together with the encrypted data and the symmetric key. We need this information to later construct a correct decryption object.

Note that generating the key with *scryptSync* is only one of two possibilities to generate a key using the NodeJS Crypto API. The other possibility is using the *pbkdf2* function as is demonstrated in figure 3.4.

Both examples 3.3 and 3.4 show how a key can be created with the NodeJS Crypto API. In both examples, the user provides a password, from which the key should be generated. Also in both examples, the user has to create a salt value. The user also has to provide the length that the key should have in both examples. This length depends on the purpose of the key (here the cipher algorithm with which the key will be used). While this is enough for the scryptSync variant, the user also has to provide the number of iterations (how many times the algorithm should run to generate the key) and with which algorithm (in this example "sha512") the symmetric key should be generated to the pbkdf2Sync variant. This algorithm can be any hash algorithm. The iterations argument must be a number that should be set as high as possible and be at least 10'000.

Whenever we got a code example with a cipher to work, we also implemented the corresponding code to decrypt the result from the cipher encryption. Figure 3.5 shows

Figure 3.4: Creating a symmetric key with pbkdf2

```
const crypto = require('crypto');

const algorithm = 'aes-256-cbc';
const salt = crypto.randomBytes(19);
const iterations = 100000;
const digestAlgorithm = 'sha512';
const key = crypto.pbkdf2Sync(process.env.PRIVATE_KEY, salt,
    iterations, 32, digestAlgorithm);
const iv = crypto.randomBytes(16);

const cipher = crypto.createCipheriv(algorithm, key, iv);

let encrypted = cipher.update('first part to encrypt', 'utf8', 'hex');
encrypted += cipher.update('second part to encrypt' , 'utf8', 'hex');
encrypted += cipher.final('hex');

storeResultWithKeyAndIV(encrypted, key, iv); //store data
```

Figure 3.5: Decipher code example

```
const crypto = require('crypto');

const algorithm = 'aes-256-cbc';
const iv = getStoredIV();
const encrypted = getStoredEncryptedData();
const key = getStoredKey();

const decrypter = crypto.createDecipheriv(algorithm, key, iv);

let decrypted = decrypter.update(encrypted, 'hex', 'utf8');
decrypted += decrypter.final('utf8');
```

such a code example that encrypts data with a cipher like in figure 3.3 and then decrypts the encrypted result. We had to use the same algorithm, key, initialization vector and encodings as for the cipher object. We stored the initialization vector together with the encrypted data. We also stored the salt so we could use it again to construct the same key as in figure 3.3 Since this information were already set, we did not run into any obstacles with these configurations. However, at first we did not concatenate the results from the *update* function of the cipher object. This led to an error when decrypting the encrypted data. Therefore, the obstacle here was that we had to concatenate the results from the *update* and *final* functions on the cipher object, otherwise we would not get a valid result. We noted that we also had to do this with the results from the *update* and *final* function on the decipher object. We again noted the steps we took while writing the code examples. The list of steps for the example in figure 3.3 would be:

- Choose a cipher algorithm (line 3)
- Create a salt (line 4)
- Derive a key of correct length for the chosen algorithm from a password and a salt (line 5)
- Create an initialization vector of correct length for this algorithm (line 6)
- Create a cipher object (line 8)
- Add data to encrypt to the cipher object (lines 10-11)
- Set the input encoding of the input data (lines 10-11)
- Set the output encoding of the encrypted result (lines 10-12)
- Perform the cipher encryption process (lines 10-12)
- Concatenate the results from the encryption processes (lines 10-12)
- Store the key and initialization vector together with the encrypted data for later usage (line 14)

The list of steps for the example in figure 3.5 would be:

- Choose the same cipher algorithm as for the encryption (line 3)
- Get the stored initialization vector (line 4)
- Get the stored key used for the encryption (line 6)
- Get the encrypted data (line 5)

- Create the decipher object (line 8)
- Add data to decrypt to the decipher object (line 10)
- Set the input encoding of the encrypted input data (line 10)
- Set the output encoding of the decrypted result (lines 10-11)
- Perform the decipher decryption process (lines 10-11)

One of the security issues we found for the cipher and decipher classes was again the algorithm choice. There are algorithms such as “des” that are considered broken in a cryptographic context, yet they are still widely used. A more secure algorithm choice could be a “aes”-variant such as “aes-256-cbc”. Another security issue is that the initialization vector should be created randomly instead of a hardcoded value. Our code example takes care of this by using the randomBytes function that creates a random buffer value of the specified length. The remaining security issues we found concern the key generation and storage. To narrow the scope of this thesis, we decided not to cover the topic of key management and only cover key generation. The symmetric key for a cipher should be generated with a strong cryptographic algorithm and should at least be the size of the block-size of the chosen cipher algorithm. The key in our example is the same size of the block-size (32 bytes). We can also trust the scryptSync function to use a strong cryptographic algorithm to generate the key. Since we are using a salt, it is important in a security context that the salt is generated randomly. In our code example, we use the randomBytes function to achieve this.

3.1.3 Asymmetric Encryption

We now had a class to hash and a class to encrypt and decrypt with a symmetric key using a cipher. We also wanted to cover the possibility to encrypt and decrypt using a key pair. The way to do this with the NodeJS Crypto API is to use the *publicEncrypt* and *privateDecrypt* functions. Again, we wrote code examples to find potential obstacles and security issues. We decided that while the NodeJS Crypto API covers keys in the form of Object, string, Buffer and KeyObject, we would only cover keys that come in the form of a string. We wrote an example that encrypts data with a public key (figure 3.6) and an example that decrypts data with the private key that matches this public key (figure 3.7).

The obstacle here was the key generation since we needed a matching key pair. To achieve this, we used the *generateKeyPair* function from the NodeJS Crypto API. To generate these keys we used the “rsa” algorithm and set the length of the keys to 4096 bits. We also specified the encoding of both the public and private key. The public key encoding was set to “spki” and the private key encoding to “pkcs8”. Both keys are created in “pem” format. Format and encoding define what the key should look

Figure 3.6: Public encryption code example

```
1 const crypto = require('crypto');
2
3 const publicKey = getPublicKey(); //load public key
4 const data = Buffer.from('toEncrypt');
5
6 const encrypted = crypto.publicEncrypt(publicKey,
7    data);
8
9 storeOrSendData(encrypted); //store data
```

Figure 3.7: Decrypt the encrypted data with the private key

```
1 const crypto = require('crypto');
2
3 const encrypted = getEncryptedData();
4 const privateKey = getPrivateKey(); //load private key
5
6 const decrypted = crypto.privateDecrypt(
7    {
8      key: privateKey,
9      passphrase: process.env.PRIVATE_KEY_PASSPHRASE
10   },
11   encrypted
12 ).toString();
```

like. To strengthen the security we encrypted the private key with the cipher algorithm "aes-256-cbc" and a password that we load from an environment variable. By doing this, we make sure that even if somehow some attacker gets the key, it still cannot be used without also knowing the password. Since we encrypt our private key, we have to provide the passphrase whenever we want to use this private key as can be seen in example 3.7. An obstacle was that we could not use string inputs since these functions only operate on buffers. Therefore we had to create buffers from our strings and also transform the returned buffer from the decryption process into a string.

We again noted all steps we took while writing such code examples. The following list shows the steps we took while writing the key generation code snippet (figure 3.8):

- Choose an algorithm to generate the key pair (line 3)
- Set the key length (line 4)
- Set the public key encoding (lines 5-7)
- Set the private key encoding (lines 9 - 11)

Figure 3.8: Generate a key pair

```
1  const crypto = require('crypto');
2
3  const { publicKey, privateKey } = crypto.generateKeyPairSync('rsa', {
4      modulusLength: 4096,
5      publicKeyEncoding: {
6          type: 'spki',
7          format: 'pem'
8      },
9      privateKeyEncoding: {
10         type: 'pkcs8',
11         format: 'pem',
12         cipher: 'aes-256-cbc',
13         passphrase: process.env.PRIVATE_KEY_PASSPHRASE
14     }
15 });
16
17 storeOrDistributeKeys(publicKey, privateKey);
```

- Encrypt the private key with an algorithm and a passphrase (lines 12-13)
- Store or distribute the keys (line 17)

The steps to encrypt data with the public key (figure 3.6) were:

- Get the public key (line 3)
- Create a buffer from the unencrypted string data (line 4)
- Perform the public encryption process with data in form of a buffer (lines 6-7)

The steps to decrypt data with the private key (figure 3.7) were:

- Get the encrypted data (line 3)
- Get the private key (line 4)
- Provide the key together with the passphrase (line 7)
- Perform the public decryption process with data in form of a buffer (lines 6-8)
- Transform the decrypted result to a string (line 8)

The security issues when working with asymmetric encryption concern the key pair generation and storage. We again decided not to cover the topic of key storage and only cover key pair generation. In a security context, it is important to generate the key pair using a strong cryptographic algorithm such as “rsa”, the algorithm we also use in our code example. It is also important that the keys are long enough so that an attacker needs a long time in a brute force attack. NIST recommends the keys generated with “rsa” to be at least 2048 bits long. [1]. In our example we chose a 4096 bit length. NIST also recommends that the private key should be secured with a passphrase. In our example 3.6 we provide a passphrase that is stored in an environment variable (line 13).

3.2 Fluent Crypto

Fluent Interface

A fluent interface is a design principle for APIs. It is based heavily on method chaining and intelligent naming of the methods with the goal of making the readability of code using the API as close as possible to human written text. To achieve method chaining, a function of an object returns the object itself. On a grammar-less fluent interface, the order of the calls does not matter.

The APIs of all FluentCrypto classes follow this design principle. All configuration and data calls on the encryption or decryption class return the object to allow method chaining. The methods are named after the tasks they solve and the order of the calls does not matter.

Figure 3.9: Performing an encryption

```
1 const encryptionObject = FluentCrypto.Encryption()
2    .data(data)
3    .run();
4
5
6 const encrypted = encryptionObject
7    .getResult();
```

To ensure that code written with FluentCrpyto is secure, we developed a small domain-specific language named CryRule. The integrated CryRuleParser parses all .cryrule files that lie in the rules directory and passes the results to FluentCrypto. FluentCrypto is of course only secure when the CryRule rule files are securely configured. A more in-depth description of CryRule can be found in section 3.3. On initialization of the encryption process via the FluentCrypto API, the corresponding rules are passed to the relevant classes. Each time one of the API functions is called, all rules for this function are validated. If any rule is violated, an exception is thrown.

Figure 3.10: Insecure algorithm

```
const algorithm = 'des';

const encrypted = FluentCrypto.Encryption()
    .data(data)
    .withCipher(algorithm)
    .run()
    .getResult();
```

Since the decryption process must be performed with the same configurations (e.g. the same key or the same initialization vector) as the encryption, all validation is done at the level of encryption.

Providing secure defaults

To make a cryptographic task as easy as possible, default values for all possible and needed configurations are used so that the user does not have to set them. This frees the user of otherwise necessary knowledge and thereby reduces the risk of errors and insecure configurations. To ensure that the defaults are actually secure, values from the parsed CryRule files are used. If there are multiple values listed for a configuration (e.g. algorithms), then the first entry is used as default. These files are considered secure configurations. The user is then free to overwrite every configuration (as long as the rules are satisfied), but is not obliged to.

Meaningful error messages

An issue with the NodeJS Crypto API is that of the rather cryptic error messages that often confuse inexperienced users. With FluentCrypto, the user calls the FluentCrypto API, which ensures with the provided defaults and validation of configurations together with the correct handling of the NodeJS Crypto API that no error from the wrapped API is thrown. The errors that do get thrown are insecure usage errors thrown by the FluentCrypto API itself. Such an error happens when a user uses an argument that is not allowed by the rules parsed from the CryRule files. These error messages are designed to tell the user not only what went wrong and why the desired configuration ended in an error but also how a possible secure configuration could look like.

Consider the example in figure 3.10. In this example we try to use the cipher algorithm “des”. This algorithm is considered insecure as a cipher algorithm. The rule files used in this project therefore do not list this algorithm as secure and when running this code, FluentCrypto will throw the following error:

```
AlgorithmNotAllowed: des is not allowed here. Use one of
aes-128-cbc, aes-128-gcm, aes-192-cbc, aes-192-gcm,
aes-256-cbc, aes-256-gcm
```

This error message not only tells the user what the problem with this piece of code is

(the algorithm “des” is not allowed) but also how the user can fix the problem (by using one of the listed algorithms).

3.2.1 Hashing

Figure 3.11: Hashing code example with FluentCrypto (corresponds to 3.2)

```
1 const FluentCrypto = require('./FluentCrypto');
2
3 const hashed = FluentCrypto.Hashing('sha512')
4    .data('text to hash')
5    .run()
6    .getResult();
```

Figure 3.11 shows how our example 3.2 would look like with FluentCrypto. On line 3, we tell the FluentCrypto API that we want to perform hashing with the algorithm “ sha512”. On line 4 we add the data, perform the hashing on line 5 and get the hashed result on line 6.

3.2.2 Symmetric Encryption

Consider our example 3.3. This example solves the task from the introduction, “Encrypt all data in our system with a private key”. In this example, we had to do this list of steps:

- Step 1: Choose a cipher algorithm (line 3)
- Step 2: Create a salt (line 4)
- Step 3: Derive a key of correct length for the chosen algorithm from a password and a salt (line 5)
- Step 4: Create an initialization vector of correct length for this algorithm (line 6)
- Step 5: Create a cipher object (line 8)
- Step 6: Add data to encrypt to the cipher object (lines 10-11)
- Step 7: Set the input encoding of the input data (lines 10-11)
- Step 8: Set the output encoding of the encrypted result (lines 10-12)
- Step 9: Perform the cipher encryption process (lines 10-12)

Figure 3.12: Cipher example rewritten with FluentCrypto

```
1  const FluentCrypto = require('./FluentCrypto');
2
3  const encrypter = FluentCrypto.Encryption()
4     .withCipher('aes-256-cbc')
5     .data('first part to encrypt')
6     .data('second part to encrypt')
7     .run();
8
9  let encrypted = encrypter
10    .getResult();
11
12 storeResuts(encrypter);
```

- Step 10: Concatenate the results from the encryption processes (lines 10-12)
- Step 11: Store the key and initialization vector together with the encrypted data for later usage (line 14)

Each of these steps required a certain knowledge until we could solve it. Take step 3, deriving a key of correct length. For this step, we had to create a cryptographic secure salt, find the correct key length for our algorithm and finally find a cryptographic secure way to generate the key given the salt and the length.

Figure 3.12 shows how this example could look like written with FluentCrypto. For the first step, nothing changes. We still choose the cipher algorithm (“aes-256-cbc”). However, with FluentCrypto, a lot that we had to do ourselves with the NodeJS Crypto API is now taken care of. Step 3 is done by FluentCrypto: Since we do not specify a key ourselves, FluentCrypto derives a fitting key from a random passphrase and a cryptographic secure salt once we choose the cipher algorithm on line 4. Note that we don’t need to know the salt and passphrase used to derive this key as long as we store the key. At this point, FluentCrypto also automatically creates a fitting initialization vector for our chosen algorithm, therefore performing step 4 for us too. On line 5 and 6, we add data that should be encrypted to our encryption object. FluentCrypto detects that the input is a string and assumes by default the input encoding “utf8”, doing step 6. It also sets the output encoding by default to the string encoding “hex”, doing step 7. On line 7, we perform the actual encryption and get the result on lines 9-10. FluentCrypto automatically concatenates the data correctly and returns the correct result, doing step 10 for us. Finally, the used key and initialization vector are stored on the encryption object together with the data and we can store this object.

Figure 3.13 shows a different way to solve the same task if we already have an existing (fitting) key. On line 5, we load a key that is stored somewhere. Note that in this example,

Figure 3.13: Performing an encryption with a symmetric key

```
const Fluentcrypto = require('./FluentCrypto');

const key = loadStoredKey();
const encrypted = FluentCrypto.Encryption()
   .data('Data to be encrypted')
   .withCipher('aes-256-cbc')
   .withSymmetricKey(key)
   .run()
   .getResult();
```

Figure 3.14: Configure a symmetric key with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');
const password = process.env.PASSPHRASE;

const encryptionObject = FluentCrypto.Encryption()
   .keyGenerationPassword(password)
   .data('data to encrypt')
   .run();

const encrypted = encryptionObject.getResult();
const key = encryptionObject.getKey();
```

we assume that this key is fitting for the chosen cipher algorithm (otherwise FluentCrypto would tell us by throwing an error). This key is then set manually on the encryption object on line 9 instead of using the key that FluentCrypto would have generated for us by default. Note that in this example, we again do not create an initialization vector ourselves. FluentCrypto does this for us, analogous to example 3.12. This example demonstrates how FluentCrypto creates all defaults (like the initialization vector) but lets the user overwrite a default value (in this case the symmetric key).

With FluentCrypto, keys are generated directly on the encryption object. FluentCrypto uses *pbkdf2* with secure default values (specified in cryrule files) for the salt, amount of iterations, the key generation algorithm and sets the length of the key fitting for the chosen (or default) algorithm. This is demonstrated in figure 3.14. Here, the user chooses to overwrite the default password the key is derived from. It is also possible to not use this function like in example 3.15. Then, a secure, random password is used. Similar to the password, a user can also configure the salt, amount of iterations, the digest algorithm and the key length, see 3.20 for an example for salt, amount of iterations and digest algorithm. Note that these configuration calls overwrite the defaults set by FluentCrypto. Note also that the purpose of the keys generated on the FluentEncrypt

Figure 3.15: Minimal configuration

```
const FluentCrypto = require('./FluentCrypto');

const encryptionObject = FluentCrypto.Encryption()
    .data('data')
    .run();
```

Figure 3.16: Insecure example rewritten with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');

const key = getStoredKey();
const iv = getStoredIV();
const algorithm = 'aes-256-cbc';
const encrypted = getEncryptedData();

const decrypted = FluentCrypto.Decryption()
   .fromCipher(algorithm)
    .data(encrypted)
    .key(key)
    .iv(iv)
    .run()
    .getResult();
```

class is to be used in a encryption or decryption scenario with FluentCrypto.

Example 3.16 shows how the data encrypted in 3.12 can then be decrypted. On lines 3-6, the stored key, initialization vector and data are retrieved and the algorithm choice is set to the same algorithm as for the encryption. A decryption object is then instantiated on line 8. The algorithm is set to the cipher algorithm choice we made on line 9. Then the encrypted data is added (line 10) and the correct configurations are then set with the *key* and *iv* calls on lines 11 and 12. Finally, on line 13, the decryption process is performed and the result ins returned on line 14.

Example 3.17 shows how the decryption can be done when the encryption object is still available. Here, a configuration object of the encryption object is passed to the *from* function on line 21. This function takes a configuration object as an argument and pulls all set or default configurations from it (like the encryption method “cipher”, the algorithm choice “aes-128-cbc” and the initialization vector , secrets like the key have to be set manually) and sets it on the decryption object. Note that this is an equivalent solution to example 3.16.

While these solutions do not save us from many lines of code, they are far more readable than the original solution written with the NodeJS Crypto API. They also free

Figure 3.17: Insecure example rewritten with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');

const encrypter = FluentCrypto.Encryption()
   .keyGenerationPassword(proccess.env.PASSPHRASE)
   .data('first part to encrypt')
    .withCipher('aes-128-cbc');

encrypter.data('second part to encrypt');

encrypter.run();

let encrypted = encrypter
    .getResult();

storeEncryptedData(encrypted);

//Decryption with the encrypter object still available
const encryptedData = getStoredEncryptedData();
const decrypted = FluentCrypto.Decryption()
    .data(encryptedData)
    .from(encrypter.configurations())
    .run()
    .getResult();
```

Figure 3.18: Distributed example with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');

const key = loadAgreedKey();

const encryptionObject = FluentCrypto.Encryption()
   .key(key)
   .data('encrypted data for bob')
   .run();

const encrypted = encryptionObject
    .getResult();

const configurations = encryptionObject.configurations();

sendToBob(configurations);
```

the user from a lot of otherwise required knowledge. When using NodeJS Crypto API, it was up to the user to know that when using *aes-128-cbc*, key and IV length have to be 16 bytes and that the encodings must be set. With FluentCrypto, we can trust the module with deriving a key of correct length for our encryption method and algorithm choice. We also do not have to construct an IV ourselves. FluentCrypto derives both the correct key length and the correct IV length from CryRule files. The solution written in FluentCrypto can also be considered more secure. While the original solution already had no insecure algorithm choices, there were still some bad choices like hardcoded, non-random salts or IV. With FluentCrypto, we can now trust that all choices are secure since no error telling otherwise is thrown when executing this code. We also have no more hardcoded salts and the IV is securely derived by FluentCrypto from the CryRule files.

3.2.2.1 Distributed symmetric encryption

Consider this example. Alice wants to encrypt data with a symmetric key and send this encrypted data to Bob, who then decrypts the data. They have already agreed on a symmetric key that they want to use. Since both of them are using FluentCrypto, sharing the other non-confidential information (such as the initialization vector or the encrypted data) and setting up the encryption and decryption process is straight-forward. FluentCrypto provides an API function that returns a representation of the encryption object and the encrypted data that contains no secret informations and can be sent over a network. This representation can then be sent to Bob, who can use the *from* call to set up the decryption process. This is demonstrated in figure 3.18.

Figure 3.19: Distributed example with FluentCrypto, Bob

```
const FluentCrypto = require('./FluentCrypto');

const key = loadAgreedKey();

const configurations = getFromAlice();

const decrypted = FluentCrypto.Decryption()
   .from(configurations)
   .key(key)
   .run()
   .getResult();
```

Now, when Bob gets the configurations from Alice, he can create a Decryption object and provide the configurations to this object. This will set all configurations (such as the initialization vector, the encrypted data or the input encoding) correctly. All that is left to Bob is to set the symmetric key and run the decryption process. This is demonstrated in figure 3.19.

If Alice and Bob do not already have a symmetric key, Alice can generate one using FluentCrypto. A minimal example is shown in figure 3.20. Figure 3.21 shows how it is possible to provide key generation parameters to overwrite the defaults.

Figure 3.20: Symmetric key generation with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');

const key = FluentCrypto.Encryption().getKey();

distributeKey(key); //share the key securely
```

Figure 3.21: Symmetric key generation with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');

const key = FluentCrypto.Encryption()
   .keyGenerationSalt(crypto.randomBytes(16))
   .keyGenerationIterations(100000)
   .symmetricKeyGenerationAlgorithm('sha512')
   .getKey();

distributeKey(key); //share the key securely
```

Figure 3.22: Distributed example with FluentCrypto and a passphrase

```
const FluentCrypto = require('./FluentCrypto');

const encryptionObject = FluentCrypto.Encryption()
   .fromPassphrase(getAgreedPassphrase())
   .keyGenerationIterations(100000)
   .data('encrypted data for bob')
   .run();

const encrypted = encryptionObject
    .getResult();

const configurations = encryptionObject.configurations();

sendToBob(configurations);
```

Figure 3.23: Distributed example with FluentCrypto, Bob

```
const FluentCrypto = require('./FluentCrypto');

const configurations = getFromAlice();

const decrypted = FluentCrypto.Decryption()
   .from(configurations)
   .fromPassphrase(getAgreedPassphrase())
   .run()
   .getResult();
```

Note that it is necessary that Alice and Bob have to share or agree on the key with other means than FluentCrypto, since the topic of exchanging or distributing keys is not covered by FluentCrypto.

Alice and Bob can also perform encryption and decryption using a shared passphrase. This is demonstrated in figures 3.22 and 3.23. Since this passphrase is secure and must be shared accordingly, it is not part of the configurations object and has to be shared with other means than FluentCrypto. Because of this, Bob and Alice have to set the passphrase using the *fromPassphrase* function. This function sets the passphrase the symmetric key is derived from and triggers then the key generation. FluentCrypto then uses this generated key to perform the encryption and decryption. Is is still possible to overwrite any defaults. In this example, this is demonstrated by overwriting the number of iterations. Note that this number of iterations is then stored in the configurations object. Therefore we don't have to configure the amount of iterations manually in the decryption example since this is already done by the *from* call.

3.2.3 Asymmetric Encryption

Consider the case where Alice and Bob want to encrypt and decrypt data using a key pair. Alice can then encrypt data with the public key and send the configuration object to Bob who can then decrypt the data using the corresponding key pair. This is demonstrated in figures 3.24 and 3.25.

Figure 3.24: Public encryption code example with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');

const publicKey = getPublicKey();

const encrypted = FluentCrypto.Encryption()
   .withPublicKey(publicKey)
   .data('toEncrypt')
   .run()
   .getResult();

storeOrSendData(encrypted);
```

Figure 3.25: Private decryption code example with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');

const privateKey = getPrivateKey();

const decrypted = FluentCrypto.Decryption()
   .withPrivateKey(privateKey)
   .privateKeyPassphrase(getPrivateKeyPassphrase())
   .data('text to hash')
   .run()
   .getResult();
```

If they do not already have an existing key pair, it is possible to generate one using FluentCrypto as is demonstrated in figure 3.26. Here, it is also demonstrated that key generation properties like the modulus can be set. If these are not set, a secure default is used. This generation has to be done by the person doing the decryption (in our example this is Bob) since the private key should not be shared. The public key can then be sent to Alice who can then perform the encryption. Note that key pairs generated with FluentCrypto always have the private key secured with a passphrase. This passphrase has to be provided to the decryption object like in figure 3.25.

Figure 3.26: Creating a key pair code example with FluentCrypto

```
const FluentCrypto = require('./FluentCrypto');

const encryptionObject = FluentCrypto.Encryption()
   .keyGenerationModulus(4096);

const privateKey = encryptionObject.getPrivateKey();
const publicKey = encryptionObject.getPublicKey();
const privateKeyPassphrase = encryptionObject.
    getPrivateKeyPassphrase();
```

3.3 CryRule

CryRule is a small, domain-specific language designed to be a bridge between developers and cryptography experts that is built for the NodeJS Crypto API and lets cryptography experts declare rules by putting constraints on parameters. It is heavily inspired by CrySL, a domain-specific language built by Krüger et al. for the Java Cryptographic Architecture that lets cryptography experts specify rules.

CryRule builds on the idea that rules are written by experts and separated from the actual code implementation. With CryRule, cryptography experts are able to declare cryptographic rules in a simple, comprehensive language. CryRule also comes with a parser that compiles CryRule files into plain JavaScript objects. These objects hold the constraints from the rule files and can then be used to validate cryptographic code.

FluentCrypto uses the CryRule parser to parse the rule files at run-time. The resulting objects are then used to validate the arguments passed to the FluentCrypto API. This also means that FluentCrypto does not perform static analysis, but only run-time analysis. The choice between static analysis and run-time analysis leads to different possible features. With run-time analysis it is possible to perform analysis on the value that a variable has at run-time. Therefore it is also possible to perform analysis on function arguments. Static analysis cannot access the value that a variable has at run-time and therefore cannot perform analysis on these values, which is only possible when performing run-time analysis. But static analysis makes it possible to check how the value of a variable was created. E.g. static analysis can check whether the value in a certain variable (e.g. a salt) was created by a certain (e.g. random) cryptographic function. FluentCrypto uses secure defaults and wraps around the NodeJS Crypto API in a way that is safe to use. Therefore, all code that is written with FluentCrypto can be considered secure and static analysis is not needed, but all values that are passed to the FluentCrypto API must be (and are) validated.

3.3.1 Design Decisions

- Scope: CryRule supports the same same scope as FluentCrypto does. It is possible to specify rules for the classes *Hash*, *Cipher* and *Public Encryption*. The CryRule rule files are structured around these classes and provide the possibility to specify rules for the arguments passed to the functions of theses classes.
- White listing:

  CryRule is designed to use white listing. A secure usage is explicitly written down while every other usage is considered insecure. This ensures that the rule files only specify secure configurations.
- Rule Possibilities:

  Rules can be specified for all parameters that are used in an API call of FluentCrypto. A Rule is a constraint on what values a parameter can take.
- Syntax:

  Cryrule is not whitespace sensitive. Neither line breaks nor indentations change the meaning.

3.3.2 Elements of a CryRule Rule

A rule written in CryRule consists of one or multiple task sections. A task section describes the rules for a certain encryption/decryption task. The hashing/encryption tasks supported are *Hash*, *Cipher* and *Public encryption*. Such a section always starts with the name of the hashing/encryption task it describes. These tasks are

HASH,
CIPHER,
PUBLICENCRYPTION

None of these sections is mandatory, but the order stated above must be fulfilled. This is due to a problem with the Backus-Naur-Form where it is not possible to both state that each event can occur zero or one time and have these events occur in any order at the same time without listing each and every possibility. The priority was set on having all sections optional.

3.3.2.1 Algorithm constraints

ALGORITHM defines a constraint on the algorithm(s) that can be used for a task and means that every algorithm listed is considered secure. It also means that every algorithm

Figure 3.27: Whitelist algorithms for hashing

```
Hash
   algorithm IN [sha256, sha512]

Cipher
   ALGORITHM aes-128-cbc
```

Figure 3.28: Constraining symmetric key length

```
Cipher
    algorithm
        IN [aes-192-cbc, aes-192-gcm, aes-256-cbc]
    symmetricKey
        LENGTH 24 IF algorithm IN [aes-192-cbc, aes-192-gcm]
        LENGTH 32 IF algorithm aes-256-cbc
```

not listed is considered insecure. This keyword is followed by either directly stating a single algorithm or by using the **IN** keyword, which is used to list different algorithms in typical JavaScript array notation. In Figure 3.27, the algorithms *sha256* and *sha512* are white-listed and therefore considered secure for hashing whereas for the cipher in this example only the algorithm *aes-128-cbc* is considered secure. Capitalisation of the keyword itself does not matter. Only one ALGORITHM keyword can be present per task.

The ALGORITHM constraint can be used to constraint HASH and CIPHER algorithms. It can also be used to constraint the key generation algorithm for a symmetric key (see 3.3.2.3) and the key generation algorithm for a key pair (see 3.3.2.5).

3.3.2.2 Length constraints

LENGTH defines a constraint on an object that has a length. This is either a symmetric key or an initialization vector. This keyword is followed by either a number or the keyword IN and an array of numbers. These number(s) are the allowed length(s) that the object is allowed to have. The number (or array of numbers) can be followed by an **IF** statement that states that this length constraint is only to apply if the algorithm of the described task is one of a certain list or a specific algorithm (compare 3.3.2.1).

In Figure 3.28 the key length is constrained to 24, if the algorithm of the cipher is one of *aes-192-cbc*, *aes-192-gcm* and constrained to 32 if the algorithm of the cipher is *aes-256-cbc*.

Figure 3.29: Symmetric key constraints

```
Cipher
    symmetricKey
        LENGTH 24 IF algorithm IN [aes-192-cbc, aes-192-gcm]
        ITERATIONS >= 10000
        SALTLENGTH >= 20
```

Figure 3.30: IV constraints

```
Cipher
    iv
        LENGTH 16 IF algorithm aes-128-cbc
        LENGTH 96 IF algorithm IN [aes-256-gcm]
```

3.3.2.3 Symmetric Key constraints

SYMMETRICKEY defines constraints for a symmetric key that can be set by the user or generated by FluentCrypto. This keyword can only appear in the cipher task section. It can contain multiple LENGTH constraints, one ALGORITHM constraint, which concerns the algorithm used in the the key generation process(compare 3.3.2.1), a SALTLENGTH >= constraint, which sets the minimum length for the salt that is to be used to generate a symmetric key and finally an ITERATIONS >= constraint which sets a minimum of times the key generation algorithm must iterate to generate a symmetric key.

In the symmetricKey section, the LENGTH, ALGORITHM, SALTLENGTH and ITERATIONS keywords have to appear in this order.

Figure 3.29 demonstrates such a symmetric key rule. In this example, the length of the symmetric key is constrained to 24 if the algorithm used for the cipher task is either “aes-192-cbc” or “aes-192-gcm”. The minimum amount of iterations is set to 10000 and the salt is constrained to a length of at least 20.

3.3.2.4 Initialization vector constraints

IV defines constraints for the initialization vector and consists solely of LENGTH constraints. This keyword can only appear in the cipher task section. Figure 3.30 demonstrates how an initialization vector can be constrained to a certain length for certain algorithms.

Figure 3.31: Key constraints

```
PublicEncryption
    keyPair
        ALGORITHM IN [rsa]
        MODULUSLENGTH [4096] IF algorithm rsa
```

3.3.2.5 Key Pair constraints

KEYPAIR defines constraints for the private key and public key of a key pair that can be set by the user or generated by FluentCrypto. ALGORITHM keyword (see 3.3.2.1) can be used to constrain the algorithm that is used to generated a key pair. Further can the modulus that is to be used to generate a key pair be constrained using **MODULUSLENGTH**. Figure 3.31 shows how constraints for a key pair (generation) can be set.

The KEYPAIR keyword can only appear in the public encryption task section. The ALGORITHM and MODULUSLENGTH keywords have to appear in this order.

3.3.3 Current state

The current cryrule rule set that is included in FluentCrypto constrains the algorithms to “aes-128-cbc”, “aes-128-gcm”, “aes-192-cbc”, “aes-192-gcm”, “aes-256-cbc”, “aes-256-gcm” for cipher tasks and “sha256”, “sha384” and “sha512” for hashing tasks. For each of these cipher algorithms, there is one constraint rule for the length of a symmetric key used with the algorithm and one constraint rule for for the length of an initialization vector used with the algorithm.

In this rule set, the symmetric key generation algorithm is constrained to “sha256”, with the amount of iterations constrained to at least 10000 and the salt length constrained to at least 20.

There are also constraints for the key pair generations. In these, the key pair generation algorithm is constrained to “rsa” and the key generation modulus length to 4096.

Any user of FluentCrypto and CryRule can adapt these rules according to section 3.3.2 but we advise to leave this to security experts.

3.4 Implementation

We implemented CryRule using chevrotain[2], an open-source parser building toolkit. The language parser is split into two parts. The first part defines the tokens and the grammar

[2]https://github.com/SAP/chevrotain

of CryRule, while the second gives the language its semantics.

FluentCrypto is built without a library and wraps the NodeJS Crypto API.

3.4.1 Overview

The entry point is the FluentCrypto class which is described in figure 3.32 and table 3.1.

As a general rule, we chose following conventions when naming our API functions:

A function name to choose or change a specific mode of encryption or hashing (e.g. symmetric encryption with a cipher) starts with “with”, followed by the method of encryption or hashing (e.g. “withCipher”). If the function takes a key or password as parameter to construct the encryption mode from, this is added to the function name (e.g. “withCipherFromPassword”).

A function name to choose or change a specific mode of decryption (e.g. symmetric decryption of data encrypted with a cipher) starts with “from”, followed by the method with which the encryption happened (e.g. “fromCipher”). Alternatively, if the encryption/decryption process is asymmetric (e.g. public encryption/private decryption), the name can start with “with” followed by the decryption method. This can be followed by a “from” and a key name if the function can generate the decryption from a provided key (e.g. “withPrivateDecryptFromPrivateKey”).

A function name to run any encryption, decryption or hashing should be named “run”.

A function name that sets a certain configuration (e.g. an algorithm or an encoding) starts with “set”, followed by the configuration that should be set (e.g. “setAlgorithm” or “setInputEncoding”).

A function name that gets a certain configuration (e.g. an algorithm or an encoding) starts with “get”, followed by the configuration that should be retrieved (e.g. “getAlgorithm” or “getInputEncoding”).

A function name that adds something (e.g. raw data to encrypt) starts with “add” followed by what it adds (e.g “addData”).

The emphasised words in the following tables are parameters

Table 3.1: FluentCrypto API

Function	Description
Encryption	Returns a FluentEncrypt object that uses by default cipher as encryption method
Decryption	Returns a FluentDecrypt object with no defaults preconfigured
Hashing	Returns a FluentHash object with preconfigured defaults

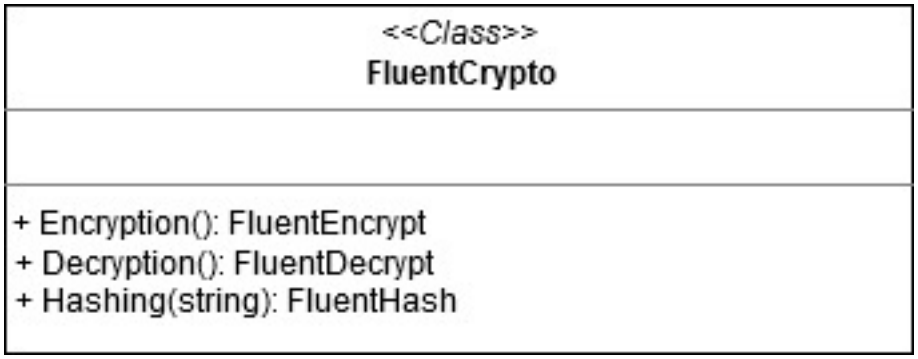

Figure 3.32: FluentCrypto

From here, the user can start with encryption, decryption or hashing by using the respective calls. These calls return an object of the respective class. Since there are two ways (symmetric and asymmetric) to perform encryption/decryption, there are also two classes for both encryption and decryption. These classes hold the configurations necessary to perform the encryption/decryption as well as the capability to perform the encryption/decryption process. The interfaces of these classes are unified under the FluentEncrypt/FluentDecrypt classes. Image 3.33 and table 3.2 illustrate this relationship for FluentEncrypt.

Function	Description	Errors
withPublicEncrypt	Uses the encryption type “encryption with public key” and initializes the defaults	
withPublicEncryptFromPublicKey	Uses the encryption type “encryption with public key”, initializes the defaults and if provided sets the *publicKey*	InvalidKeyLengthError
withCipher	Uses the encryption type cipher and initializes the defaults. If *algorithm* is provided, this algorithm is set.	AlgorithmNotAllowed, InvalidKeyLengthError, InvalidIvLengthError
withCipherFromSymmetricKey	Uses the encryption type cipher, initializes the defaults for the provided *symmetricKey* and sets the provided *symmetricKey*	AlgorithmNotAllowed, InvalidKeyLengthError, InvalidIvLengthError

withCipherfromPassword	Uses the encryption type cipher, initializes the defaults and generates a key from the provided *password*	
setKey	Sets the symmetric key to be used with a cipher to the provided *symmetricKey*. Overwrites the default	InvalidKeyLengthError
setPublicKey	Sets the public key to be used with a public key encryption to the provided *publicKey*. Overwrites the default	InvalidKeyLengthError
setIv	Sets the initialization vector to be used with a cipher to the provided *iv*. Overwrites the default	InvalidIvLengthError
setInputEncoding	Sets the encoding of the data that is to be encrypted to the provided *inputEncoding*. Overwrites the default	
setOutputEncoding	Sets the encoding of the encrypted data that will be returned to the provided *outputEncoding*. Overwrites the default	
setKeyGenerationPassword	Sets and overwrites the password the default symmetric key is generated from to the provided *password*. Triggers the key generation and overwrites the set or default key.	
setKeyGenerationSalt	Sets and overwrites the salt the default symmetric key is generated with to the provided *salt*. Triggers the key generation and overwrites the set or default key.	SaltTooShort

setKeyGenerationModulus	Sets and overwrites the modulus the default symmetric key is generated with to the provided *modulus*. Triggers the key generation and overwrites the set or default key.	InvalidModulusLengthError
setKeyGenerationIterations	Sets and overwrites the amount of iterations the default symmetric key is generated with to the provided *iterations*. Triggers the key generation and overwrites the set or default key.	
setKeyGenerationLength	Sets and overwrites the length of the generated default symmetric key. Triggers the generation and overwrites the set or default key.	
setSymmetricKeyGenerationAlgorithm	Sets and overwrites the digest algorithm with which the default symmetric key is generated to the provided *algorithm*. Triggers the key generation and overwrites the set or default key.	AlgorithmNotAllowed
setKeyPairGenerationAlgorithm	Sets and overwrites the algorithm with which the default key pair is generated to the provided *algorithm*. Triggers the key generation and overwrites the set or default key.	AlgorithmNotAllowed
data	Adds *data* to encrypt	
run	Perform the encryption	
getResult	Return the encrypted result	

getPublicKey	Returns the public key generated by default or set by the user	NoPublicKeyError
getPrivateKey	Returns the private key generated by default or set by the user	NoPrivateKeyError
getPrivateKeyPassphrase	Returns the passphrase for the private key generated by default or set by the user	NoPassphraseError
getSymmetricKey	Returns the symmetric key generated by default or set by the user	NoSymmetricKeyError
getAlgorithm	Returns the algorithm set by default or by the user	NoAlgorithmError
getIV	Returns the initialization vector generated by default or set by the user	NoIVError
getInputEncoding	Returns the input encoding set by default or set the user	NoInputEncodingError
getOutputEncoding	Returns the output encoding set by default or set the user	NoOutputEncodingError
getConfigurations	Returns a FluentEncrypt configurations object from which a FluentDecrypt object can be created	

Table 3.2: FluentEncrypt API

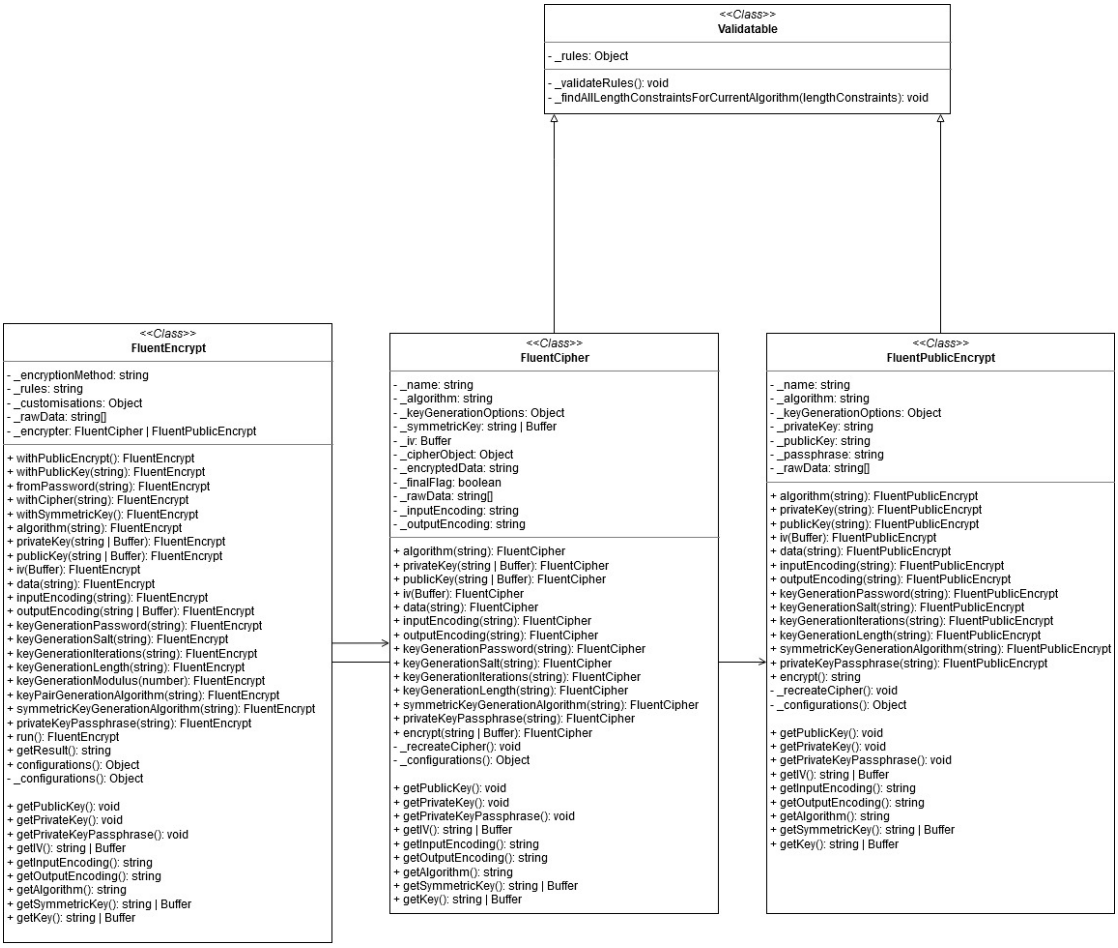

Figure 3.33: FluentEncrypt

Every time the user changes a configuration on a FluentEncrypt object (e.g. *iv* or *inputEncoding*), this configuration is stored on the _customisations object and then applied to an object of the FluentCipher or FluentPublicEncrypt class that is stored in the _encrypter variable, depending on which encryption method is currently selected (by default this is FluentCipher). To make this clearer, we will explain what happens in the first part of example 3.34. On line 5, the user starts the encryption process by calling the *Encryption* function. This function now returns a FluentEncrypt object with secure defaults. These defaults are set using the CryRule rule files. By default, FluentEncrypt uses symmetric encryption, therefore a FluentCipher object is created and stored in the *_encrypter* class variable. Also, on instantiation of a FluentEncrypt object, an empty object is stored in the *_cusotmisations* class variable. Now, on line 6, the user calls the *setPublicKey* function with a public key as argument. Now the public key is added to the *_customisations* object as a property named “publicKey” (note that this property

name is the same as the function name of the FluentEncrypt API). Then, the public key is set on the FluentCipher object stored in the *_encrypter* variable using its *publicKey* function. This call again returns the FluentEncrypt object. On line 7, data to be encrypted is added. This data is added to the *_rawData* class variable that is an array of raw data that should be encrypted. Then, the data is added to the FluentCipher object stored in the *_encrypter* object using its *data* function. Finally on line 8, the encryption method is changed from cipher to public encryption by using the *withPublicEncryption* call. Now, on the FluentEncrypt object a FluentPublicEncrypt object is instantiated and stored in the *_encrypter* variable. FluentEncrypt now uses the *_customisations* object which stores all configurations made by the user (in this case only the public key) and sets this configurations on the newly instantiated object of the FluentPublicEncrypt class. The same is then done for all data stored in the *_rawData* array. The public key set on line 6 and the data set on line 7 are now also set on the FluentPublicEncrypt object.

This allows the user to set configurations and change the encryption methods in any order (see 3.34).

Figure 3.34: Different order - same effect

```
1  const FluentCrypto = require('./FluentCrypto');
2
3  const publicKey = getPublicKey();
4
5  const encrypter = FluentCrypto.Encryption()
6     .publicKey(publicKey)
7     .data('first')
8     .withPublicEncryption();
9
10 const equalEncrypter = FluentCrypto.Encryption()
11    .data('first')
12    .withPublicEncryption()
13    .publicKey(publicKey);
```

The following table 3.3 shows the exposed functions of the FluentDecrypt class. Note that this class does not perform validations on the input since the configurations have to match the configurations of the encrypting object and are therefore validated there. The only validation is made for the private key passphrase at the decrypt stage.

Function	Description	Errors
fromCipher	Uses the decryption type cipher and initializes the defaults with the provided *algorithm*	
fromPublicEncrypt	Uses the decryption type “decryption with private key” and initializes the defaults	

withPrivateDecryptFromPrivateKey	Uses the decryption type “decryption with private key” and initializes the defaults and sets the provided *privateKey*	
fromCipherWithPassword	Uses the decryption type cipher and generates a key from the provided *password* to decrypt with. Needs the configurations from the encryption object	
fromFluentEncrypt	Uses the correct decryption configurations derived from the provided FluentEncrypt configurations object *fluentEncrypt*	
data	Adds *data* to be decrypted	
setKey	Sets the key to be used with a cipher decryption to the provided *key*. This is a symmetric key in case of a cipher and a private key in case of a private decryption	
setPrivateKeyPassphrase	Sets the passphrase for the private key to be used in a private key decryption to the provided *passphrase*	
setIv	Sets the initialization vector to be used with a cipher decryption to the provided *iv*	
setInputEncoding	Sets the encoding of the data that is to be decrypted to the provided *inputEncoding*	
setOutputEncoding	Sets the encoding of the decrypted data that will be returned to the provided *outputEncoding*	
run	Perform the decryption	BadPassphraseError, NotProvidedError
getResult	Get the decrypted result	

Table 3.3: FluentDecrypt API

Image 3.35 and the following 3.4 illustrate the FluentHash class.

Function	Description	Errors
setAlgorithm	Sets the algorithm with which the data should be hashed to the provided *algorithm*	AlgorithmNotAllowed
setInputEncoding	Sets the encoding of the data that is to be hashed to the provided *inputEncoding*. Overwrites the default	

setOutputEncoding	Sets the encoding of the hashed data that will be returned to the provided *outputEncoding*. Overwrites the default	
data	Adds *data* to hash	
run	Perform the hashing	
getResult	Get the hashed result	
getAlgorithm	Returns the algorithm set by default or by the user	
getInputEncoding	Returns the input encoding set by default or set the user	
getOutputEncoding	Returns the output encoding set by default or set the user	

Table 3.4: FluentHash API

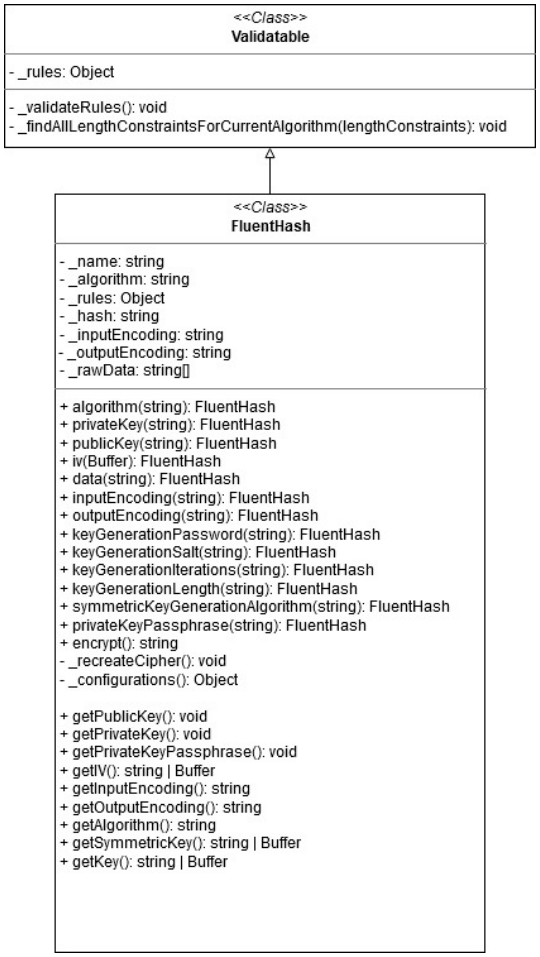

Figure 3.35: FluentHash

To generate keys, salts or initialization vectors that conform to the constraints specified in the rule files, the classes depend on the FluentHelpers class which is a collection of helper functions that can generate rule-conform objects. FluentHelper is illustrated in image 3.36.

When an object of the hashing, encryption or decryption class is instantiated, the integrated CryRule parser parses the CryRule files in the rules directory and passes the result to the newly instantiated class. During runtime, each time any configuration (such as the key, initialization vector, etc.) changes on a Validatable object, the *_validateRules* function is called and the rules are validated. If there is any bad configuration, an error with an expressive error message is thrown. An overview of the classes can be seen in image 3.37. The code with the documentation and the examples can be found on GitHub[3].

[3]https://github.com/Smoenybfan/FluentCrypto

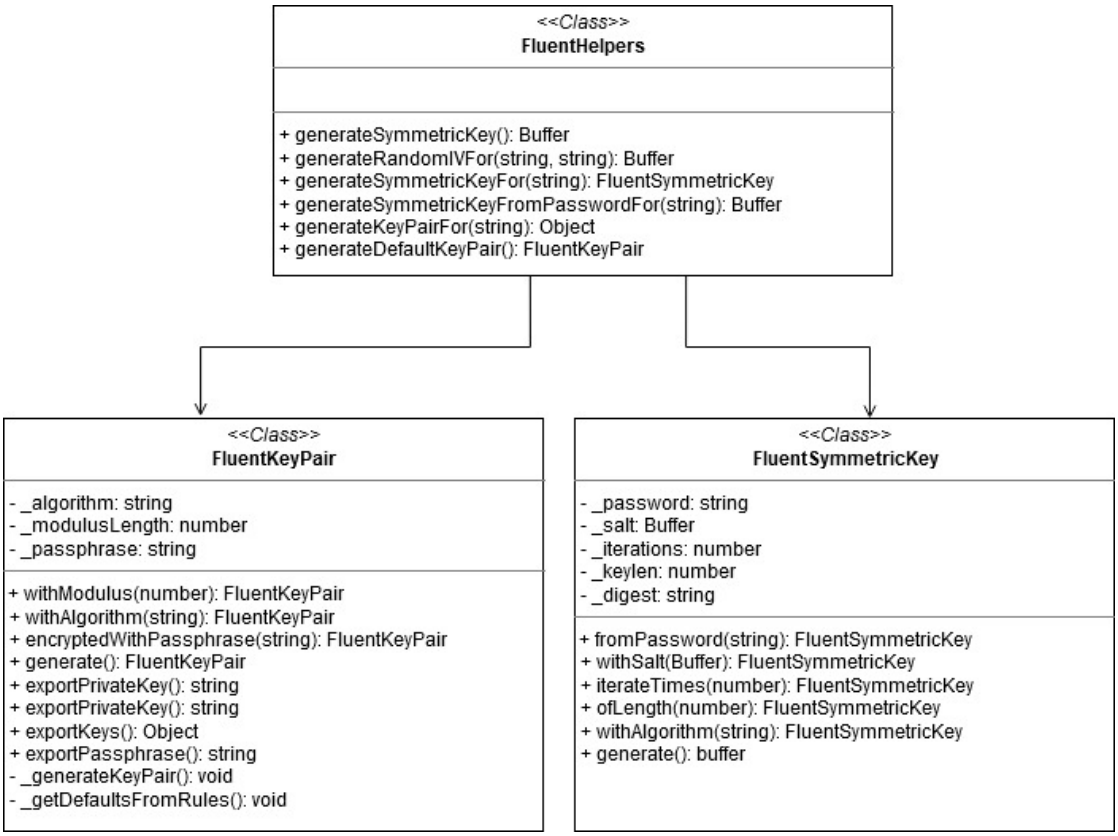

Figure 3.36: FluentHelper

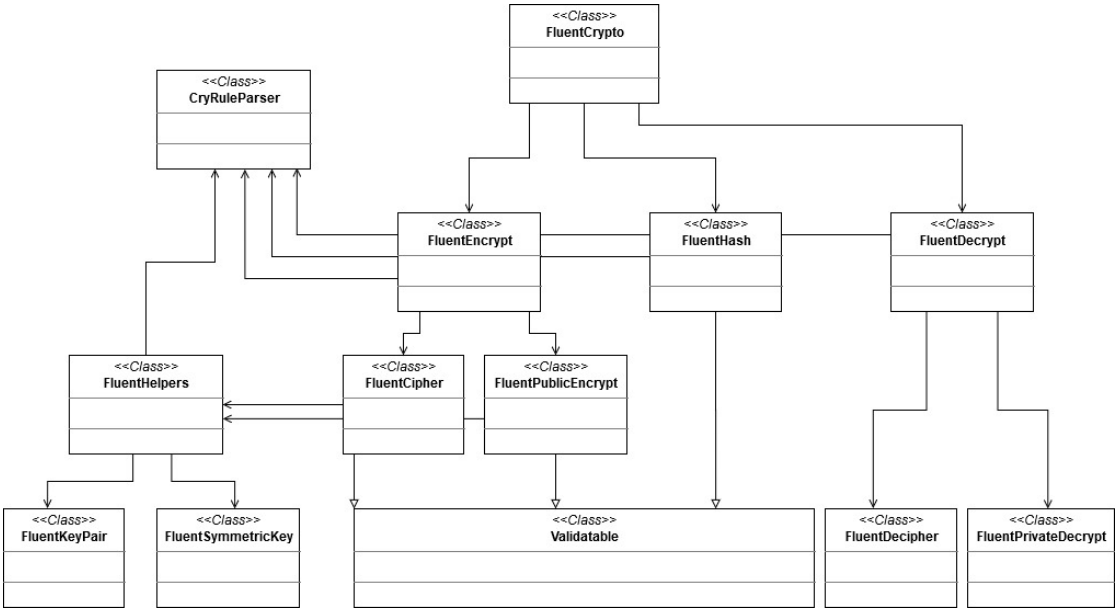

Figure 3.37: Overview

3.4.1.1 Errors

This table shows the Errors that can be thrown by the FluentCrypto API.

Table 3.5: FluentCrypto API Errors

Error	Description
AlgorithmNotAllowed	Thrown when the user tries to use an algorithm that is not allowed by the rule files
InvalidKeyLengthError	Thrown when the user tries to set a key whose length does not match the algorithm
InvalidIvLengthError	Thrown when the user tries to set an initialization vector whose length does not match the algorithm
InvalidModulusLengthError	Thrown when the user tries to generate a key pair with a modulus length that is not allowed by the rule files
NoPublicKeyError	Thrown when the user tries to get a public key from an encryption object that does not use a public key
NoPrivateKeyError	Thrown when the user tries to get a private key from an encryption object that does not use a private key
NoSymmetricKeyError	Thrown when the user tries to get a symmetric key from an encryption object that does not use a symmetric key
NoInputEncodingError	Thrown when the user tries to get the input encoding from an encryption object that does not use an input encoding
NoOutputEncodingError	Thrown when the user tries to get the output encoding from an encryption object that does not use an output encoding
NoAlgorithmError	Thrown when the user tries to get the algorithm from an encryption object that does not use a specific algorithm
NoIVError	Thrown when the user tries to get the initialization vector from an encryption object that does not use an initialization vector
NoPassphraseError	Thrown when the user tries to get the private key passphrase from an encryption object that does not use a private key
BadPassphraseError	Thrown when the passphrase provided does not match the private key
SaltTooShort	Thrown when the salt provided is too short

3.4.2 Extending FluentCrypto

This section should give you an idea on how FluentCrypto can be extended in the future by briefly explaining how the passphrase policy for private keys could be implemented (this is currently encouraged by FluentCrypto and done by default during key pair generation, but not enforced).

The first step would be to extend CryRule and the CryRuleParser. One would first add new tokens (like “PASSPHRASE” and “REQUIRED” in figure 3.38). Then these tokens would be used to write a new grammar rule that describes this new constraint (see

Figure 3.38: New tokens for CryRule

```
const passphrase = createToken({name: 'passphrase',
    pattern: /PASSPHRASE/});

const required = createToken({name: 'required', pattern: /REQUIRED/});
```

Figure 3.39: New rule for CryRule

```
  // new rule for the passphrase
  $.RULE('passphraseClause', () => {
      $.CONSUME(passphrase);
      $.CONSUME(required);
  });

  ...

  // adding it to the existing keyPair rule as a subrule
  $.RULE('keyPairClause', () => {
      ...
      $.OPTION4(() => $.SUBRULE($.passphraseClause));
  });
```

3.39). This rule would then be added as an optional subrule in the keyPair rule.

The next thing to adapt would be the semantics of the visitor that gives meaning to these parsed tokens. First, one would add a function for the "passphraseClause" we introduced. Since the relevant part about this rule is its presence, the function can simply return a truthy boolean. We then adapt the semantics of the keypairClause accordingly by adding a boolean named "passphraseRequired" to the returned object of the keyPairClause function. Whether this boolean is truthy or falsy depends on the presence of the "passphraseClause" (see figure 3.40). For more details on tokens, semantics and parsing, we refer to the documentation of chevrotain[4].

Now that CryRule supports this new constraint, one would need to adapt FluentCrypto accordingly. The next step would be to adapt the _validateRules function of the Validatable class to check this new constraint analogous to the existing constraints (see figure 3.41). The final step would then be to call the _validateRules at the correct time. One probably does not want to call it whenever a user sets the private key, because if one did this, then the user would be required to first set the passphrase and then the private key, otherwise FluentCrypto would throw an error. The better place to make this call would be whenever the encrypt function is called since the passphrase must be provided together with the key at this moment (figure 3.42).

[4]https://sap.github.io/chevrotain/docs/

Figure 3.40: Semantics of the new rule

```
// semantics rule for the passphraseClause
passphraseClause(ctx) {
    if(ctx.required) {
      return true;
    }
    return false;
  }

...

// adding it to the existing keyPair semantics
keyPairClause(ctx) {
    ...
    let passphraseConrtaint = ctx.passphraseClause?
      this.visit(ctx.passphraseClause) : false;

    return {
      ...
      passphraseConstraint
    };
}
```

Figure 3.41: Adapting the _validateRules call

```
 _validateRules() {
     ...
     if(this._rules.keyPairConstraints &&
       this._rules.keyPairConstraints.passphraseRequired) {
         if(this._privateKey && !this._passphrase) {
           throw new NoPassphraseProvidedError();
         }
     }
 }
```

Figure 3.42: Adding the _validateRules call

```
     encrypt() {
       this._validateRules()
       ...
     }
  }
```

Figure 3.43: Adding the rule to a rule file

```
PublicEncryption
    keyPair
        ALGORITHM IN [rsa]
        MODULUSLENGTH [4096] IF algorithm rsa
        PASSPHRASE REQUIRED
```

Figure 3.44: Testing with examples

```
//with passphrase, should run fine
FluentCrypto.Encryption()
    .withPrivateKey(loadPrivateKeyWithPassphrase())
    .privateKeyPassphrase(loadPrivateKeyPassphrase())
    .data('test')
    .run();

//without passphrase, should throw an error
FluentCrypto.Encryption()
    .withPrivateKey(loadPrivateKeyWithoutPassphrase())
    .data('test')
    .run();
```

One can now test this by first adapting the rule file of the CryRuleParser accordingly (see figure 3.43). Then one could write two code samples with FluentCrypto where in one, one provides a private key with a passphrase and in the other one provides a private key with no passphrase (note that this has to be done with pre-generated key pairs/a pre-generated private key since a key pair/private key generated with FluentCrypto is per default always encrypted with a passphrase). The first example should then run fine, whereas the second example should throw an error because of the rule newly added to CryRule.

With this being implemented, CryRule and FluentCrypto would be extended to state a policy on private key passphrases. Other new constraints could be added analogously.

4 The Validation

To evaluate FluentCrypto we carried out an anonymous survey. In the first part of this survey, the participants were asked questions about their background and experience. In the second part, they were asked to solve three basic cryptographic tasks with both the NodeJS Crypto API and FluentCrypto. These tasks were designed after frequent problems which we found on Stack Overflow. The participants then stated what was challenging about each task, where the challenges lay and how confident they were in their solution. The code they wrote for each task was then evaluated by hand to see if the code written with FluentCrypto is actually more secure than the code written with the NodeJS Crypto API.

4.1 The participants

To find participants, we first wrote two blog posts [1] [2] explaining FluentCrypto, when and how it can be useful and asked the readers then to participate in the evaluation survey we held. We shared these blog posts and the call for participants across programming forums, especially NodeJS, JavaScript and cryptography formus. We also sent the call for participants to colleagues we knew were working with NodeJS and developing backend systems.

[1] https://dev.to/smoenybfan/a-glimpse-into-the-challenges-of-working-with-cryptography-apis-in-nodejs-2hc7

[2] https://dev.to/smoenybfan/secure-and-easy-use-of-cryptography-with-fluentcrypto-1bno

Eight participants (of whom we know from talks at least three were colleagues) answered our call and carried out our survey. Of these eight participants, four had 2-5, three had 5-10 and one had more than 10 years of experience in programming. All of the participants had experience with NodeJS and had developed at least one server-side system with NodeJS. Exactly half of the participants felt that they had some knowledge of cryptography, meaning that they had a vague idea about some areas of cryptography and where and how they are used. The other half felt that they had a solid knowledge of cryptography, meaning they were familiar with various areas of cryptography and knew how and where to apply them. None of the participants felt that they had either no knowledge or expert knowledge.

Three participants had used the NodeJS Crypto API before, with only one participant having more than one year (2-4 years) experience with the API.

4.2 The tasks

The survey and the tasks were based on the problems that developers frequently search for on Stack Overflow. For each of the following tasks, the participants had to solve them first with NodeJS and then with FluentCrypto.

To measure the results, the participants stated after each task how long it took them to solve the task, if they could solve the task at all and how difficult they perceived the task. They were also asked what the obstacles in solving the task were and if they had to consult the documentation (and if so, if it was sufficient or if they had to search for additional resources). Finally, the participants uploaded or sent their code. We then checked if they experienced the same problems we found in the Stack Overflow questions and whether FluentCrypto helped to mitigate these problems. To find out whether FluentCrypto also made the code more secure, we analysed this code by hand for possible security issues we found in section 3.1.

Task 1

A Stack Overflow search[3] for “hash” with the tag *node.js* showed over 500 questions of people with over 30 of the questions active within the last week. We then searched for the most relevant questions since these are the questions that are more likely to appear when developers search for a solution to their problems. Since some of the questions were about different libraries but still matched our search or used the crypto module but had problems elsewhere, we skipped questions that were not directly about problems with the NodeJS Crypto API. From these questions we analysed the first ten for the problems the person asking the question had. Seven of the ten questions had the problem that the person asking did not know how to use or apply a hash. One person did not have enough knowledge about what a hash is in general and the remaining problems were

[3] 14.02.2020 https://stackoverflow.com/search?tab=active&q=%5bnode.js%5d%20hash

with either JavaScript itself or an external library in a different language. We decided that one of the tasks of the survey should be based on simply hashing a string to see whether how to hash was easier with FluentCrypto.

The participants were tasked to create two different strings, then hash them both and finally to compare them. The goal in this task was to compare how difficult it is to perform simple hashing with NodeJS and FluentCrypto.

Task 2

Analogously a Stack Overflow search [4] for "cipher" with tag *node.js* and a search [5] for "key cipher" with tag *node.js* showed also both over 500 questions with each having around 10 active questions with the last week. Having seen that developers also frequently struggle with this topic, we again searched for the most relevant questions since these are the questions that are more likely to appear when developers search for a solution to their problems. Since some of the questions were about different libraries but still matched our search or used the crypto module but had problems elsewhere, we skipped questions that were not directly about the NodeJS Crypto API. From these questions we analysed the first ten for the problems the person asking the question had. Here, five of the ten questions were problems where the person asking did not know how to use the API correctly, resulting in an error or incorrect result. Two of these persons explicitly struggled with concatenating the results from the calls, because they did not know they had to do that. Two problems were about not knowing the correct length of the initialization vector, one person did not have enough general knowledge about a cipher and tried to use it when they wanted to hash a string and the remaining questions were problems in context with other languages. Since FluentCrypto only covers NodeJS, we left these last two out in the task design and decided to simply design a task based on just using a cipher. Since we expect FluentCrypto to implicitly help developers with using the API correctly (concatenate results, use an initialization vector of correct length), we did not specify anything about these parts to see the impact FluentCrypto has.

The participants were tasked to first encrypt a given string, then add another given string to this encryption. Then, this total encrypted data should be decrypted again. Both the encryption and decryption should be performed with a symmetric key. The first goal in this task was to compare how difficult it is to find out how to encrypt and decrypt with a symmetric key (with a cipher) and perform the encryption/decryption with NodeJS and FluentCrypto. The second (hidden) goal was to see whether the participants could create or load a symmetric key matching for this task. The reasoning behind this (hidden) second goal is that using a correct key is one of the challenges when using cryptographic libraries. Giving the participant more information on how, why and where such keys can be created would have distorted the outcome of the survey.

Task 3

[4] 14.02.2020 https://stackoverflow.com/search?tab=active&q=%5bnode.js%5d%20cipher

[5] 14.02.2020 https://stackoverflow.com/search?q=%5Bnode.js%5D+key+cipher

Since cipher uses a symmetric key and a Stack Overflow search [6] for "key pair" with tag *node.js* also showed over 500 results we narrowed the search down to encryption with private key with a search [7] for "private encrypt" with tag *node.js*. This narrowed the search down but still returned 251 questions, with six having been active within the last two weeks. We could then assume from these results that NodeJS developers frequently struggle with asymmetric encryption and chose the last task to be based on private/public encryption and again searched for the most relevant questions since these are the questions that are more likely to appear when developers search for a solution to their problems. Since some of the questions were about different libraries but still matched our searches or used the crypto module but had problems elsewhere, we skipped questions that were not directly about the NodeJS Crypto API. From these questions we analysed the first ten for the problems the person asking the question had. Here we noticed that eight of these ten questions had the problem that the persons asking either did not know how to generate a key pair for asymmetric encryption or then did not know how to apply the key correctly. One person did not know how to provide a password for an encrypted private key and one person did not understand the API and JavaScript itself correctly. We decided that we wanted the participants to create the keys themselves and then just apply them to see the impact FluentCrypto has on the issue of generating and then using a key pair.

The participants were tasked to encrypt a given string. Then they should encrypt a second given string and add it to the already encrypted data. This encryption should be done either with a private or public key. The encrypted data should then be decrypted with the corresponding public or private key. The first goal of this task was to compare how difficult it is to find out how to encrypt/decrypt with a key pair and perform the encryption/decryption with NodeJS and FluentCrypto. The second (hidden) goal was to see whether the participants could create or load a key pair matching for this task. The reasoning behind this (hidden) second goal is that using a correct key pair is one of the challenges when using cryptographic libraries. Giving the participant more information on how, why and where such keys can be created would have distorted the outcome of the survey.

In the end, the survey was structured as follows:

- String hashing
  - Create two different strings and compare their hash values
- Symmetric String Encryption
  - First encrypt the string "Text that is going to be sent over an insecure channel". Then update the encrypted data with "and must be encrypted at all costs!"

[6] 14.02.2020 https://stackoverflow.com/search?q=%5Bnode.js%5D+key+pair
[7] 14.02.2020 https://stackoverflow.com/search?tab=active&q=%5bnode.js%5d%20private%20encrypt

(encrypted). Then decrypt the encrypted data. The encryption and decryption must be performed with a symmetric key. The encrypted data must be in hex encoding, the unencrypted and decrypted strings in utf8.

- Asymmetric String Encryption
  - First encrypt the string “Text that is going to be sent over an insecure channel”. Then update the encrypted data with “and must be encrypted at all costs!” (encrypted). The encryption must be performed with a private or public key. The encrypted data must then be decrypted with corresponding public or private key

4.3 Survey Results

We grouped the participants according to their years of experience, their perceived level of cryptographic knowledge and whether they had previous experience with the NodeJS Crypto API to find out, which groups benefited from FluentCrypo (and how much they benefited), if there were groups FluentCrypto had no effect on and if there were groups which were hindered by FluentCrypto. We did not analyse the results of the participants individually since we were rather interested in how a group of a certain level of knowledge performed.

4.3.1 Task 1

Table 4.1: Survey Results Task 1 broken down on years of experience

Years of experience	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
2-5 years	9 min 15 sec	3.5	5 min	3.5
5-10 years	4 min 20 sec	2	4 min 20 sec	2
$>$ 10 years	2 min	2	6 min	2

Table 4.2: Survey Results Task 1 broken down on perceived level of cryptographic knowledge

Perceived level of cryptographic knowledge	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
No knowledge	NA	NA	NA	NA
Some knowledge	9 min 15 sec	3.5	5 min	3.5
Solid knowledge	3 min 45 sec	2	4 min 45 sec	2
Expertise knowledge	NA	NA	NA	NA

Table 4.3: Survey Results Task 1 broken down on experience with NodeJS Crypto API

Previous experience with NodeJS Crypto	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
No Experience	8 min 25 sec	3	5 min	3
Experience	3 min 20 sec	2.5	4 min 40 sec	2.5

As can be seen in table 4.1, the results from the survey show that while the experienced difficulty stays the same, the time needed for the task decreases for less experienced programmers but increases for very experienced participants. Tables 4.2 and 4.3 show a similar result when broken down on the perceived level of cryptographic knowledge and experience with the NodeJS Crypto API.

All participants managed to solve the task with both the NodeJS Crypto API and FluentCrypto.

4.3.2 Task 2

Table 4.4: Survey Results Task 2 broken down on years of experience

Years of experience	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
2-5 years	17 min	6.5	10 min	5.5
5-10 years	8 min 20 sec	4	11 min	7
$>$ 10 years	12 min	5	1 min	3

Results from one participant were excluded for the results of task 2 since the participant encountered a bug that was then fixed for the other participants.

Task 2 was more challenging to the participants than the first part. This is visible in the overall increased perceived difficulty and the tasks taking more time to solve (tables 4.5 and 4.6) but also in three of the (seven) participants not being able to complete the task when using the NodeJS Crypto API. These participants all managed to complete the task when using FluentCrypto. Overall, FluentCrypto also managed to decrease the perceived difficulty for less experienced participants while increasing it a bit for more experienced participants. The time it took to complete the task when using FluentCrypto decreased for all participant groups except the group "5-10 years of experience" as can be seen in table 4.4.

Table 4.5: Survey Results Task 2 broken down on perceived level of cryptographic knowledge

Perceived level of cryptographic knowledge	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
No knowledge	NA	NA	NA	NA
Some knowledge	27 min 20 sec	9	11 min 30 sec	5.5
Solid knowledge	9 min 30 sec	4	7 min 40 sec	5.5
Expertise knowledge	NA	NA	NA	NA

Table 4.6: Survey Results Task 2 broken down on experience with NodeJS Crypto API

Previous experience with NodeJS Crypto	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
No Experience	23 min	8	11 min	5
Experience	9 min	4.5	6 min 30 sec	6.5

4.3.3 Task 3

Table 4.7: Survey Results Task 3 broken down on years of experience

Years of experience	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
2-5 years	20 min	9	9 min	5
5-10 years	6 min 20 sec	5	5 min 20 sec	4
$>$ 10 years	20 min	9	NA	2

Table 4.8: Survey Results Task 3 broken down on perceived level of cryptographic knowledge

Perceived level of cryptographic knowledge	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
No knowledge	NA	NA	NA	NA
Some knowledge	20 min	9	9 min	5
Solid knowledge	9 min 45 sec	5.5	5 min 20 sec	4.5
Expertise knowledge	NA	NA	NA	NA

Table 4.9: Survey Results Task 3 broken down on experience with NodeJS Crypto API

Previous experience with NodeJS Crypto	**NodeJS time (min)**	**NodeJS difficulty (1-10)**	**FluentCrypto time (min)**	**FluentCrypto difficulty (1-10)**
No Experience	17 min 30 sec	7.5	7 min 25 sec	4.5
Experience	9 min 40 sec	7	7 min 30 sec	5.5

Task 3 was the task with the highest perceived difficulty when being solved with the NodeJS Crypto API, as can be seen in table 4.7. It also results in being the task that took the most time to solve on average. FluentCrypto managed to bring down the perceived difficulty and the time the task took to solve for all groups, with the less experienced groups being the most significant. Table 4.8 shows that the complexity and time to solve go down for both the groups with “Some knowledge” and “Solid knowledge”. The same can be seen in table 4.9.

Of the eight participants, five did not manage to complete the task using the NodeJS Crypto API. When using FluentCrypto, all eight participants could complete the task.

4.4 Survey Security Analysis

The participants of the survey were required to send in their solutions to all tasks. We then compared the solutions written with the NodeJS Crypto API and FluentCrypto and evaluated them in their algorithm choice and key generation/retrieval process. The main focus on this evaluation lay on secure choices and processes.

4.4.1 Hash

When using the NodeJS Crypto API, all participants chose an algorithm that is considered secure with the most frequent choice being *sha256*.

When using FluentCrypto, the participants either used the secure default algorithm set by the CryRule rule file (*sha256*) or their own choice that was considered secure by FluentCrypto. All participants who did not use the default algorithm chose *sha256*. One participant originally wanted to use the *md5* algorithm but was not allowed to by FluentCrypto.

4.4.2 Cipher

When using NodeJS Crypto API, all participants chose an algorithm that is considered secure with the choice being one of *aes256* or *aes-256-cbc*.

The choice of key differed more. Only one participant used a key that was not hardcoded. Most of the keys were generated with one of the key generation calls from the NodeJS Crypto API but still used a hardcoded secret to generate the key from.

When using FluentCrypto, the participants either used the secure default algorithm set by the CryRule rule file (*aes-128-cbs*) or chose the algorithm *aes-128-cbs*.

All participants used the provided call of FluentCrypto to generate a key to use. The default key choice was not used by any participant. No hardcoded values were used by any participant.

4.4.3 Asymmetric encryption/decryption

The main subtask containing security risks is creating/retrieving key pair. When using the NodeJS Crypto API, all participants chose the *generateKeyPair* call and used the *rsa* algorithm with sufficient length. The participants also all used hardcoded values to generate the key pairs from.

When using FluentCrypto, all participants used the default key pair provided by the FluentEncrypt module.

4.5 Obstacles and Solutions

At the end of the survey, the participants were asked what obstacles they had to overcome and what helped them find solutions. We also check here, if the participants experienced the problems we found in the Stack Overflow questions and if FluentCrypto helped to mitigate these problems.

4.5.1 Hash

The participants had few issues using the NodeJS Crypto API. The problems were solved by reading the documentation. Only one participants had specific issues with having to concatenate the hashing results and had to do additional research on Stack Overflow. The problems we identified in our Stack Overflow searches, which were almost all basic questions on how to hash, seemed not to appear for our participants.

When using FluentCrypto API, the only obstacle the participants had to overcome was finding the correct part of the documentation. No participant had to do additional research.

4.5.2 Cipher

When using the NodeJS Crypto API, the participants mentioned having several difficulties with concatenating the results and using the provided call correctly. All participants found the documentation on this part either incomplete or confusing and had to do additional research. They used Stack Overflow and various Wikipedia pages to solve the task. Some participants could not solve the task.

This matches with the problems we found in our Stack Overflow searches, where the most frequent problem was not understanding how to use the API correctly. Five of the eight participants mentioned such struggles. Two of the participants also had the problem of generating an appropriate initialization vector, a problem we also found in our searches.

When using FluentCrypto, the biggest issue was concatenating the results correctly. One participant tried to generate an initialization vector for an unsupported cipher algorithm and was not satisfied with the thrown error message. The concatenating issue could mostly be solved by reading the documentation. Only two participants had to do additional research, they looked at the source code. Therefore we can assume that FluentCrypto managed to mitigate the problems the participants were having with the NodeJS Crypto API.

4.5.3 Asymmetric encryption/decryption

When using the NodeJS Crypto API, most participants could not finish the task. These participants complained about the lack of examples and documentation and having to do lots of research on Stack Overflow. The participants who could solve the task also complained about lacking documentation and had to do research on Stack Overflow to complete it.

Again, this matches with the problems we found in our Stack Overflow searches, where the most frequent problem was not knowing how to generate a key pair or how to then apply these keys.

When using the FluentCrypto API, one participant was not able to complete the task due to a bug in FluentCrypto. Otherwise, no participant mentioned any obstacles. Every subtask could be solved by reading the documentation. No additional research had to be done. We can therefore assume that FluentCrypto managed to mitigate the problems the participants were having with the NodeJS Crypto API.

5
Conclusion and Future Work

One of the two main goals of FluentCrypto was to provide an API for cryptography in NodeJS that is easier to use than the standard NodeJS Crypto API. The results from the survey show that more participants were able to complete the tasks handed to them than when using the NodeJS Crypto API. The less experienced participants needed overall much less time to finish the task and they rated the task less difficult than with the NodeJS Crypto API. When we look at the more experienced participants, we see that for some tasks the time and complexity was higher with FluentCrypto than with the NodeJS Crypto API, but overall there is still a gain in speed and ease to solve, though smaller. We can therefore assume that the task-based design with a fluent API and run-time validation eases the process of solving cryptographic tasks in NodeJS.

The other main goal of FluentCrypto was to make the process of solving these tasks more secure. However, we did not experience much more secure solutions with FluentCrypto than we did with the NodeJS Crypto API. While no algorithm choices with FluentCrypto are considered insecure, only one choice of a participant had to be corrected from the NodeJS Crypto API solutions. An improvement from the NodeJS Crypto API solutions are less hardcoded sensitive values in favour of the FluentCrypto defaults. Overall, we can document an increase in security, but only a small one.

We can also confirm that one of the main struggles of the participants was incomplete or lacking documentation with few examples. This led to all participants having to do additional research with only some of them succeeding. With FluentCrypto, these participants were mostly able to complete the task with the API and documentation alone. This strengthens our arguments for more excessive documentations with examples.

None of the participants of the surveys could be considered beginners nor experts,

therefore not allowing much insight into how well these groups of developers would perform with FluentCrypto. The group of participants is also quite small. Future resources could be invested in widening the scope of participants of the study.

Code analysis showed that there is a problem with participants storing the key or a secret in plain sight. This is not something that FluentCrypto can currently mitigate and has to be left open for future work.

While being successful, FluentCrypto could only prove the usefulness of its design for two (main) sections of cryptography. The tasks it covers are also all synchronous. Future work could be invested into expanding the concept to cover more of the original NodeJS Crypto API and discover if asynchronous tasks are solvable with the design of FluentCrypto.

CryRule is a basic language with only a few features. Should FluentCrypto cover more concepts, then so must CryRule. The design of the language makes future improvements feasible. Future work should also be invested by professional cryptographers to improve both the language and existing rule files.

6
Anleitung zu wissenschaftlichen Arbeiten

To make the NodeJS Crypto API easier to use we chose to implement a wrapper around it. We narrowed the API down to a size that was feasible to cover in this thesis but still includes important parts of the API widely used in the field. Our choice fell on the subject of encrypting and decrypting data. Being the two most used encryption methods, the hash and cipher classes were chosen as a basis.

6.1 Moving to a task-based implementation

As the first action, a class was introduced that should be the class the user is talking with, the main API of FluentCrypto. To begin with, this class handled and stored everything. The user could choose between hash and cipher, but she/he would have had to instantiate the object, configure it and put the data on it. Both the Hash and the Cipher were then refactored into their own classes. The main class would then have a variable for each a cipher and a hash. This resulted in a main flaw in the tool at this stage. As can be seen in figure 6.1, the API does not much more than providing defaults. The user would still have to know how to use a cipher and a hash.

During the next iteration, FluentCrypto moved from a mere wrapper to a tasked-based implementation, meaning that the API calls provide the user with a solution to solve tasks instead of providing tools to find these solutions. The main tasks that should be supported are **Encryption** and **Decryption**. To provide this, a class for each task was introduced. The purpose of these classes is to provide an API to the user that is not dependent on which of the encryption methods is used to fulfil the task. Under the hood,

Figure 6.1: First FluentCrypto API

```
createHash(algorithm = 'sha256') {
    ...
}

withHashAlgorithm(algorithm){
    ...
}

hash(data, encoding = 'hex'){
    ...
}

createSignature(algorithm = 'sha256'){
    ...
}

generateKeyPair(){
    ...

}

generateRandomIV(size = 16){
    ...
}

createCipher(iv, privateKey, algorithm = 'aes-192-cbc'){
    ...
}

createDecipher(iv, privateKey, algorithm = 'aes-192-cbc'){
    ...
}

generateSymmetricKey(){
    ...
}
```

an encrypter object is stored that is either an object of type FluentHash or FluentCipher. These classes expose the same interfaces that can be used by the FluentEncryption class that itself is exposed to the user.

To provide such an interface, common sub-tasks had to be identified. These resulted to be **adding data to encrypt**, **setting the algorithm**, **setting the key**, **setting the initialization vector**, **setting the encodings** and **performing the encryption**. Figure 6.2 shows the API after the refactoring.

The user now did not have to know anymore which specific functions she/he has to call to create an object that can perform hash (or cipher) encryption. The FluentEncrypt class would use a sensible default. All the user had to do to get a basic encryption going is to call the *data* function to add as much data as wanted and finally perform the encryption with the *encrypt* call. This relieved the user from some required knowledge, but left the possibility to configure the encryption process manually.

Problematic here is that a hash and a cipher are different in what they need to run. Both need an algorithm, but while this is sufficient for a hash, the cipher needs a key and an initialization vector. These properties are not present on a hash. Having a coherent API is an important property of the encryption class. Therefore, the choice was made to add these unneeded calls to the hash class too. These calls would do nothing as is demonstrated in Figure 6.3. In a later stage of development, these calls would throw an error that would inform the user that this configuration is not applicable to the chosen encryption method (Figure 6.4).

By default, the FluentEncrypt class chooses the Hash as default encryption method due to it being the simplest.

6.2 Adding the decryption

The next step was to add the decryption. This happened the same way as the encryption. First, a FluentDecrypt class was introduced that serves as an abstraction for the whole decryption process. At this point, since a hash is per definition irreversible, only one decryption option was possible, the decipher. The same configuration options that are used for encryption are also used for decryption. This also means that the same key and initialization vector are used for both the decipher and the cipher. In the optimal case, the encryption object is still available at the time of decryption. In NodeJS, this is not impossible. To make the user's life as easy as possible, FluentCrypto should be able to construct the decryption object from the encryption object. A new call on the FluentDecrypt class was introduced, the *from* call that takes a FluentEncrypt object and uses its configurations to set the configurations for the FluentDecrypt object accordingly. Figure 6.5 shows how this had been implemented.

Figure 6.2: FluentEncrypt API

```
class FluentEncrypt {


    withHash(algorithm) {
        ...
    }

    withCipher(algorithm) {
        ...
    }

    key(symmetricKey){
        ...
    }

    iv(iv) {
        ...
    }

    inputEncoding(encoding) {
        ...
    }

    outputEncoding(encoding) {
        ...
    }

    data(data) {
      ...
    }

    encrypt() {
        ...
    }
}
```

Figure 6.3: Not Applicable Configuration

```
class FluentHash {

   ...

   iv(iv){
      return this;
   }
}
```

Figure 6.4: Not Applicable Configuration throws Error

```
class FluentHash {

   ...

   iv(iv){
      return this;
   }
}
```

6.3 Key-pair encryption and decryption

The task that was still missing was performing an encryption and decryption with a key pair consisting of a public and a private key. With the NodeJS Crypto API, this is done with publicEncrypt (using the public key to encrypt and the private key to decrypt) or privateEncrypt (using the private key to encrypt and the public key to decrypt) and their corresponding decryption methods. These methods were implemented via four new classes, FluentPrivateEncrypt and FluentPublicEncrypt and their corresponding decryption classes. Since these methods introduced the new concepts **private key** and **public key**, new configuration calls had to be added to the exposed FluentDecrypt API as well as to the already existing internal classes.

To ensure this independence of the order of calls, every configuration call that is being called at any time is tracked internally and added to a configuration object. When the encryption or decryption method is changed, these calls are all replayed on the new object. The same happens with the raw, unencrypted data.

Figure 6.5: Construct FluentDecrypt object from FluentEncrypt

```
class FluentCipher {

   ...

   _configurations() {
        return {
            algorithm: this._algorithm,
            iv: this._iv,
            key: this._symmetricKey,
            inputEncoding: this._inputEncoding,
            outputEncoding: this._outputEncoding,
        };
    }
}

class FluentDecrypt {

   ...

   from(fluentEncrypt){
        let generalConfigs = fluentEncrypt._configurations();

        let encrypter = generalConfigs.encrypter;
        switch(generalConfigs.encryptionMethod){
            case 'hash': {
                throw new NotDecryptable('A hash is not decryptable.
                Use another algorithm if you need to decrypt.');
            }

            case 'cipher': {
                let cipherConfigs = encrypter._configurations();
                this._decrypter =
                  new FluentDecipher(cipherConfigs.algorithm,
                  cipherConfigs.key, cipherConfigs.iv)
                    .inputEncoding(cipherConfigs.outputEncoding)
                    .outputEncoding(cipherConfigs.inputEncoding);

                break;
            }
        }
    }
}
```

Figure 6.6: First Implementation of FluentPublicEncrypt

```
class FluentPublicEncrypt {

    constructor() {
        this._name = 'Public Encrypt';
        let {privateKey, publicKey} =
        FluentHelpers.generateKeyPairFor('publicEncryption');
        this._privateKey = privateKey;
        this._publicKey = publicKey;
        this._rawData = [];
    }

    privateKey(key) {
        this._privateKey = key;

        return this;
    }

    publicKey(key) {
        this._publicKey = key;

        return this;
    }

    keyPair(privateKey, pubKey) {
        this._privateKey = privateKey;
        this._publicKey = pubKey;
        return this;
    }

    data(data) {
        this._rawData.push(data);

        return this;
    }

    encrypt() {
        return crypto.publicEncrypt(this._publicKey,
        Buffer.from(this._rawData.join('')));
    }

}

class FluentPrivateEncrypt() {
   ...
}
```

Figure 6.7: Hash rule example

```
Hash
    algorithm
         IN [sha256, sha384, sha512]
```

6.4 Implementing CryRule

At this point, the FluentCrypto API was a wrapper around the NodeJS Crypto API, providing an interface to the user that is task based and more fluent to use than the original API. What was missing was a way to validate or enforce secure usage of the API. The idea was to have a way of providing configurations for such usage independent from the code implementation. A small domain-specific language that is compiled and evaluated during runtime and that can be written by experts got chosen as the provider for these configurations. Since no such language existed, the decision was made to implement it using the *chevrotain* JavaScript library. *Chevrotain* makes it possible to define languages by defining tokens and grammar rules and then add semantics.

This language was named CryRule with its files being called “rule files”. To be treated as a listing of sensible and secure choices, CryRule follows a whitelisting approach. Whitelisting in this context means that the language tells only what is sensible and secure, but not what is forbidden. On the other side, every configuration that is not listed can be considered as not sensible and secure and as forbidden. As a proof of concept, the grammar for the hash was implemented. A hash rule starts with the *Hash* token, followed by the rule for the hash algorithm where a list of allowed algorithms can be specified. Since the algorithm is the only sensible choice to be made for a hash, the rule was already complete. An example rule can be seen in Figure 6.7.

Next came the cipher. Since the initialization vector and a symmetric key have to be provided for a cipher and can be used in a wrong way, experts must be able to specify rules for these parts. The randomness of the initialization vector cannot be tracked, therefore it is also not covered by a rule. What can be tracked is the length of both the key and the initialization vector. To do this, the *LENGTH* token was introduced. It is used together with a following integer value that is the allowed length of the key or the vector. Multiple *LENGTH* statements for the same concept are allowed. To save a user from having to write multiple statements to allow multiple lengths, the lengths can also be provided as an array following the *LENGTH* token. This array has to be preceded by a *IN* keyword. Some lengths may be allowed for only certain algorithms, therefore an *IF* statement following the length or an array of lengths can be stated. This *IF* statement is either followed by an algorithm name or an array of algorithm names for which this length rule should apply. These rules are sufficient for the cipher class. Figure 6.8 shows what such a rule file could look like.

Figure 6.8: Cipher rule example

```
Cipher
    algorithm
        IN [aes-128-cbc, aes-128-gcm, aes-192-cbc,
        aes-192-gcm, aes-256-cbc, aes-256-gcm]
    symmetricKey
        LENGTH 16 IF algorithm IN [aes-128-cbc, aes-128-gcm]
        LENGTH 24 IF algorithm IN [aes-192-cbc, aes-192-gcm]
        LENGTH 32 IF algorithm IN [aes-256-cbc, aes-256-gcm]
    iv
        LENGTH 16 IF algorithm IN [aes-128-cbc,
        aes-192-cbc, aes-256-cbc]
        LENGTH 96 IF algorithm IN [aes-128-gcm,
        aes-192-gcm, aes-256-gcm]
```

For the private and public encryption, the possibility to provide rules for both the private key and the public key was introduced. Similar to the symmetric key of the cipher, a length constraint can be specified. Another addition is the ability to provide rules for the key pair generation. The end-user can configure the generation algorithm as well as the modulus length. These are also the possibilities to provide rules for. A demonstration of such a rule file can be seen in Figure 6.9.

Chevrotain grammar follows the Backus-Naur-Form. This means that it is not possible in an efficient way to say that multiple terms A,B and C can all be used once or never and that their order does not matter. The way to solve this would be to make a giant OR statement that includes every possibility in every order. While it would be possible, it would be very impractical. That is why the decision was made to sacrifice the trade of the order not mattering, meaning that you do not have to provide rules for all of *Hash*, *Cipher*, *Private Encryption* or *Public Encryption*, but you have to follow this order. The implementation behind this logic can be seen in Figure 6.10.

After the definition of the tokens and the grammar came the semantics. *chevrotain* provides a simple, yet effective way to implement a visitor for the concrete syntax tree. For every grammar rule defined, a visitor function must be created that gives the visited node the semantics. For CryRule, it was enough to construct an object that holds all written rules in a consistent way. This means that the single algorithm and length constraints had to be put in an array so that the FluentCrypto tool implementation does not have to make unnecessary checks. Otherwise it was a straight-forward, bottom-up stitching together of the rule parts following the grammar.

The grammar and visitor are then wrapped by a main class that has a single public function that is exposed and starts parsing the rule files in a certain directory.

Figure 6.9: Private and public encryption rule example

```
PrivateEncryption
    keyPair
        ALGORITHM rsa
        MODULUSLENGTH [4096] IF algorithm rsa
        LENGTH IN [509, 800, 3268, 3272, 3276]
    privateKey
        LENGTH IN [509, 800, 3268, 3272, 3276]

PublicEncryption
    keyPair
        ALGORITHM IN [rsa]
        MODULUSLENGTH [4096] IF algorithm rsa
        LENGTH IN [509, 800, 3268, 3272, 3276]
    publicKey
        LENGTH IN [509, 800, 3268, 3272, 3276]
```

6.5 Integrating CryRule into FluentCrypto

With the tool in place to parse rule files, it had to be implemented into FluentCrypto. The main class that provides the first interface to the end user also starts the CryRuleParser and gets the parsed rules in return. When a FluentEncrypt object gets instantiated, the rules for encryption are provided as parameter. Now each time any configuration call is being made, the rules are validated to ensure the correct usage. To make this process easier to adopt and extend, a base class responsible to validate itself against the rule was introduced, from which each class that follows rules (such as the FluentCipher or FluentHash) derives.

Most of the rules are simple checks: Is the algorithm that the user wants to use whitelisted? Is the key the user just provided of valid length for the current algorithm? If a user now makes any configuration that is not allowed, FluentCrypto will throw an error with an error message that tells the user both why it went wrong and how she/he could do it correctly, followed by providing the rules.

Another usage of the rules is providing sensible defaults. Instead of any hardcoded values, FluentCrypto extracts the defaults from the rules. Since CryRule follows a whitelisting approach, a value from a rule can be treated as a sensible and secure choice.

6.6 Adding Symmetric Encryption From Password

During the testing, we realised that the simplest way for two different users would be to agree on some passphrase, exchange this and then create all cipher configurations from this passphrase. So we added a call. This call takes a passphrase as parameter,

Figure 6.10: CryRule Grammar

```
const chevrotain = require('chevrotain');
const createToken = chevrotain.createToken;

const Hash = createToken({name: 'Hash', pattern: /Hash/});

const Cipher = createToken({name: 'Cipher', pattern: /Cipher/});

...

const KeyPair = createToken({name: 'KeyPair', pattern: /KeyPair/});

const WhiteSpace = createToken({
    name: "WhiteSpace",
    pattern: /\s+/,
    group: chevrotain.Lexer.SKIPPED
});

...

class CryRuleParser extends chevrotain.Parser {
    constructor() {
        super(allTokens);

        const $ = this;

        $.RULE('CryRule', () => {
            $.OPTION(() => $.SUBRULE($.hashStatement));
            $.OPTION1(() => $.SUBRULE1($.cipherStatement));
            $.OPTION2(() => $.SUBRULE2($.privateEncryptionClause));
            $.OPTION3(() => $.SUBRULE2($.publicEncryptionClause));
            $.OPTION4(() => $.SUBRULE3($.keyPairClause));
        });

        ...

        $.RULE('InClause', () => {
            $.CONSUME(IN);
            $.CONSUME1(arrayStart);
            //Until last Identifier
            $.MANY(() => {
                $.CONSUME2(Identifier);
                $.CONSUME3(Comma)
            });
            //Last identifier without trailing comma
            $.CONSUME4(Identifier);
            $.CONSUME5(arrayEnd);
        });

        ...

    }
}
```

Figure 6.11: Cryrule visitor

```
class Visitor extends BaseVistor {

    constructor() {
        super();
        this.validateVisitor();
    }

    InClause(ctx) {
        let listedIns = ctx.Identifier.map(identToken => {
            return identToken.image;
        });

        return {
            type: 'InClause',
            listed: listedIns
        }
    }

     algorithmIsClause(ctx) {
        return {
            type: 'algorithmIsClause',
            allowedAlgorithms: [ctx.Identifier[0].image]
        }
    }

    //This is the place where all constraints come together
    hashStatement(ctx) {
        let hashConstraints = this.visit(ctx.hashClause);

        return {
            type: 'hashStatement',
            allowedAlgorithms: hashConstraints.algorithmConstraints
        }
    }

    algorithmClause(ctx) {
        let listedAlgorithms = this.visit(ctx.InClause);

        return {
            type: 'algorithmClause',
            allowedAlgorithms: listedAlgorithms.listed
        }
    }

    ...
}
```

Figure 6.12: The validatable class

```
1
2 class Validatable {
3     constructor(rules) {
4         this._rules = rules;
5     }
6
7     //Called after every change
8     _validateRules() {
9         ...
10    }
11
12    _findAllLengthConstraintsForCurrentAlgorithm(lengthConstraints) {
13         ...
14    }
15 }
```

sets the appropriate defaults and constructs a cipher object with these defaults. The user decrypting can do the same thing to achieve a matching decryption object.

6.7 Cleaning up FluentCrypto

Now that the user can only use secure configurations and the implementation is using sensible defaults, the remaining thing to do was to make the API as easy to use as possible. A documentation covering all exposed functions and classes was written. This documentation was then also used to write JavaDocs-like comments, the extensive documentation of all exposed function on code-level. IDEs use these comments to display tooltips to the developer using an API. Tools like JSDoc can extract these comments to provide an external documentation.

To complete the task-based orientation, the API got expanded with calls that use a concept that is not an encryption method itself but part of it, e.g. a symmetric key that is used for a cipher.

We then decided to move all key generation configurations directly to the FluentEncrypt object. So we added these calls to the FluentEncrypt class and removed the helper functions from the main FluentCrypto class. We also refactored the FluentHash class out of the FluentEncrypt since hashing and encrypting are different concepts. We then also exposed a configurations call that can be used to extract all configurations from a FluentEncrypt object. From this configuration object, a FluentDecrypt object can be created using the *from* call.

Figure 6.13: Task-based API calls

```
class FluentEncrypt {

   ...

    withPrivateKey(key){
      ...
    }
    withPublicKey(key) {
      ...
    }
    withSymmetricKey(key) {
      ...
    }

    ...
}
```

7 NodeJS Crypto API Guidelines

7.1 Hash

A cryptographic hash function is a function with the following properties:

- It is deterministic (the same message results always in the same hash)
- It is quick to compute
- It is infeasible in practice to revert the function or get the original message from the hash function
- Small changes to the message lead to extensive changes in the hash, so there is no visible correlation of the original messages
- It is infeasible to find two different messages with the same hash value

It takes a message as an input and results in an output of fixed length, called the digest. Often, what is meant with the term Hash is the result of this function, but it can also refer to the function itself. Hashing is usually part of the process of creating digital signatures or forms of authentication. It can also be used to index data in hash tables or as a checksum.

Figure 7.1: createHash example

```
1 const crypto = require('crypto');
2 const hash = crypto.createHash('sha256');
```

7.1.1 crypto.createHash(algorithm *[,options]*)

This function creates and returns a Hash object. You can then either use the hash object as a stream or with the update and digest functions. Either way, the hash object calculates the hash digest using the given algorithm. The options argument is optional and controls the stream behaviour. An example can be seen in figure 7.1.

7.1.1.1 Crypto-related dangers

The dangers depend on your use case. Using weak and fast algorithms such as MD5 or SHA1 for password encryption is dangerous since brute-force and dictionary attacks can find a matching hash in relatively short time even for salted hashes[1]. Also, such algorithms have been proven to be attackable with collision attacks [2] [3] . For this reason, Google [4] and Mozilla [5] have officially deprecated sha1 and discourage the usage of these algorithms.

- Brute-force attack: A brute-force attack on a cryptographic hash executes the hash function with different inputs until the output value matches a desired value.
- Dictionary attacks: In a dictionary attack, a list of pre-computed values is used to speed up the process of brute-forcing an attack. Usually such lists contain commonly used passwords (taken from previous data breaches), standard English words and so on. These lists are usually presented as easy-to-search tables called rainbow tables.
- Collision attack: A collision attack on a cryptographic hash tries to find two inputs that result in the same hash value. A possible exploit from this would be to present a trustworthy document to sign. For that, the signer would calculate the hash value of the document. The attacker would then take the signature and append it to a malicious document with the same hash value and make the malicious document look trustworthy.

[1] https://security.stackexchange.com/questions/19906/is-md5-considered-insecure
[2] https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html
[3] https://shattered.io/static/infographic.pdf
[4] https://security.googleblog.com/2014/09/gradually-sunsetting-sha-1.html
[5] https://blog.mozilla.org/security/2017/02/23/the-end-of-sha-1-on-the-public-web/

7.1.1.2 Rules

- Don't use a variant of the SHA1 or MD5 algorithms, since they have been proven broken in a cryptographic context. Instead use a strong hashing algorithm such as Whirlpool, SHA384 or SHA3. The algorithms you can use depend on the version of OpenSSL. You can find the possible algorithms with the crypto.getHashes() function or with the command openssl list -digest-algorithms.
- If you are using the hash during the process of user authentication/password management, do not use a fast hash algorithm since they also lead to faster brute-force attacks. For this use case also check the general rules applying for hashes in general.

7.1.1.3 Using the hash object as a stream

You can use the hash object as a stream that is both writeable and readable. For example, you can listen to the "readable" event that is emitted when there is data to read (when data was written on the stream) and then read from the stream.

7.1.2 hash.update(data *[,inputEncoding]*)

Update the hash with the given data that can be of type string, Buffer, TypedArray or DataView. If the data is of type string, then you can set the encoding of this data string by providing the inputEncding argument. Possible values are utf8, ascii, or latin1 with the default value of utf8. You can call this function as many times as needed until the digest function is called and the hash is locked. An example can be found in figure 7.2

7.1.3 hash.digest(*[inputEcoding]*)

Calculates the output value (the digest) of the data that have been passed to the hash via the update function. If encoding is provided, a string in this encoding will be returned, otherwise a buffer is returned. A call of this function locks the hash, further calls to the update function will result in errors. An example can be found in figure 7.2

7.1.4 General rules

If you are looking to use a hash for password storage, hashing alone is not sufficient. This is due to their nature of being computationally fast and deterministic what leads to faster brute-force and dictionary attacks. You should therefore always use salted hashes. In general, if you are just looking for a way to store your passwords securely, security

Figure 7.2: update and digest example

```
1 const crypto = require('crypto');
2 const hash = crypto.createHash('sha256');
3
4 hash.update('some data to hash');
5 console.log(hash.digest('hex'));
```

experts recommend that you do not roll your own cryptography[6] but use a hardened and tested algorithm such as the current industry standard bcrypt[7] , respectively its node implementation[8].

7.2 HMAC

Keyed-Hashed Message Authentication Code (HMAC) is a way to verify data integrity and authenticity of a message. It works based on a hash function and a secret key. To start, it derives two keys from the secret key. Then, on a first round, it appends the first derived key to the message and hashes it. On the second round, it appends the other derived key to the hashed value and hashes this again.

7.2.1 crypto.createHMAC(algorithm, key *[,options]*)

Creates and returns an HMAC object. This object can be used as a stream or by calling the update and digest methods. Either way, the object calculates the HMAC digest based on the given algorithm and the secret key. The optional options argument controls stream behaviour.

7.2.1.1 Crypto-related dangers

The dangers to this algorithm depend on its two big components: the secret key and the underlying hash algorithm. If the chosen underlying hash algorithm is not strong enough, it might lead to attacks mentioned in the hash chapter. It should be mentioned that there have not been any feasible attacks on sha1-HMAC and md5-HMAC but this is assumed to be only a matter of time since they are considered “broken” in a cryptographic context. More dangerous for a HMAC algorithm is to have its key leaked. This would reduce the HMAC to a simple hash algorithm and weaken the security strongly. A leaked key would also enable attackers to sign malicious documents with a legit signature.

[6]https://motherboard.vice.com/en_us/article/wnx8nq/why-you-dont-roll-your-own-crypto
[7]https://auth0.com/blog/hashing-in-action-understanding-bcrypt/
[8]https://github.com/kelektiv/node.bcrypt.js

7.2.1.2 Rules

- Do not use a variant of the SHA1 or MD5 algorithms, since they have been proven broken in a cryptographic context. There have not been any feasible attacks on a hmac using these algorithms yet, but this could change. Also, these algorithms have been marked as deprecated. Instead use a strong hashing algorithm such as Whirlpool, SHA384 or SHA3. The algorithms you can use depend on the version of OpenSSL. You can find the possible algorithms with the crypto.getHashes() function or with the command openssl list -digest-algorithms.
- The choice of the secret key length depends on the underlying hash function. It is recommended that the length is at least the size of the blocks of the underlying hash function [9].
- The secret key must be generated using a strong cryptographic pseudo-random function.
- Never store the key in plaintext anywhere.
- Do not use the same secret key in another place of your application. A single secret key should be used only for one purpose.[10]

7.2.2 hmac.update(data *[, inputEncoding]*)

Updates the hmac with the given data. If the data is a string, then the optional inputEncoding argument sets the encoding of this string. An example can be found in figure 7.3.

7.2.3 hmac.digest(*[encoding]*)

Calculates the digest value of all the data that has been passed to the hmac object. If the optional encoding argument is provided, then a string in this encoding is returned, otherwise a Buffer. An example can be found in figure 7.3.

7.3 Sign

A digital signature is a way to present the authenticity of a digital message or document. It gives a recipient a reason to believe that the document was created by a known sender. The algorithm consists of two keys, where one of them is private and one is public. The

[9] https://tools.ietf.org/html/rfc4868#ref-HMAC

[10] https://www.owasp.org/index.php/Key_Management_Cheat_Sheet#Storage

Figure 7.3: HMAC update and digest example

```
const crypto = require('crypto');
const hmac = crypto.createHmac('sha256', 'a secret');

hmac.update('some data to hash');
console.log(hmac.digest('hex'));
```

algorithm then generates the signature using the private key. Then, the receiver can use the public key of the sender and see if the message can be trusted. Using this technique, the three properties of authentication (Check if the sender is who she/he claims to be), integrity (check that the message has not been tempered with since being sent) and non-repudiation (the sender cannot deny having sent the message) are satisfied. These properties are only valid under the condition that the system has not been breached.

7.3.1 crypto.createSign(algorithm, *[, options]*)

Creates and returns a sign object that is using the given hash algorithm. You can use the *crypto.getHashes()* function to get the available hash functions. The sign object can either be used as a stream or by using the *sign.update* and *sign.sign* functions. The optional options argument controls the behaviour of the writeable part of the stream. An example can be found in figure 7.4.

7.3.1.1 Crypto-related dangers

Since hash functions are used, it is important to choose a strong hash algorithm. Especially important for digital signatures is the aspect of collision resistance (compare the chapter on hashes for more details) that says that it is unfeasibly hard to find two messages with the same hash value. MD5 and sha1 are considered “broken” considering this property.

7.3.1.2 Rules

- Don’t use a variant of the SHA1 or MD5 algorithms, since they have been proven broken in a cryptographic context. Instead use a strong hashing algorithm such as Whirlpool, SHA384 or SHA3. The algorithms you can use depend on the version of OpenSSL. You can find the possible algorithms with the *crypto.getHashes()* function or with the command openssl list -digest-algorithms.

7.3.1.3 Using the sign object as a stream

You can use the sign object as a stream that is both writeable and readable. For example, you can listen to the "readable" event that is emitted when there is data to read (when data was written on the stream) and then read from the stream.

7.3.2 sign.update(data, *[, inputEncoding]*)

Updates the sign with the given data. If data is a string, then the optional inputEncoding argument can set the encoding of this string. This method can be called as many times as needed until the sign function is called. An example can be found in figure 7.4.

7.3.3 sign.sign(privateKey *[, inputEncoding*)

Calculates the signature of all the data that has been passed to the sign object via the *sign.update* function. The *privateKey* can be either a string or an object. If it is an object, it must contain one or more of these properties:

- key (required): the private key itself, PEM-encoded
- passphrase: a string, used to protect the key further in case it gets leaked. It should be hard to guess and reasonably long [11].
- padding: An optional value for the RSA padding that can be one of the constants crypto.constants.RSA_PKCS1_PADDING (default) or crypto.constants.RSA_PKCS1_PSS_PADDING
- saltLength: In case, the padding value of crypto.constants.RSA_PKCS1_PSS_PADDING is chosen, this optional salt length can be set. crypto.constants.RSA_PSS_SALTLEN_DIGEST sets the salt length to the digest size, crypto.constants.RSA_PSS_SALTLEN_MAX_SIGN sets it to the maximal size.

  An example can be found in figure 7.4.

If the optional outputFormat argument that can be either latin1, hex or base64. Provided, the sign function returns a string in the given encoding, otherwise a buffer is returned. Once the sign function has been called, the sign object is closed. Calls to the update or the sign function will result in errors being thrown.

[11] https://www.ssh.com/ssh/passphrase

Figure 7.4: Sign update and sign example

```
const crypto = require('crypto');

const { privateKey, publicKey } = crypto.generateKeyPairSync('rsa', {
  modulusLength: 2048,
});

const sign = crypto.createSign('SHA256');
sign.update('some data to sign');
sign.end();
const signature = sign.sign(privateKey);
```

7.3.3.1 Crypto-related dangers

The big danger for every algorithm dealing with key is having the key leaked. A leaked key would in this case enable attackers to sign malicious documents with a legitimate signature.

- The choice of the secret key length depends on the underlying hash function. It is recommended that the length be at least the size of the blocks of the underlying hash function[12].
- The secret key must be generated using a strong cryptographic pseudo-random function.
- Never secret store the key in plaintext anywhere.
- Don't use the same secret key in another place of your application. A single secret key should be used only for one purpose[13].

7.4 Verify

After a signature of some data has been created, it must be verified by some way. This is where the Verify class comes in. It takes the data, the public key (pairing the private key that created the signature) and checks if the signature can be verified.

7.4.1 crypto.createVerify(algorithm , *options]*)

Creates and returns a verify object that uses the given algorithm. There is no choice over the algorithm here since it must be the same as the algorithm used to create the

[12]https://tools.ietf.org/html/rfc4868#ref-HMAC
[13]https://www.owasp.org/index.php/Key_Management_Cheat_Sheet#Storage

signature. The optional options parameter controls the behaviour of the *stream.Writeable*. An example can be found in figure 7.5.

7.4.1.1 Using the verify object as stream

The verify object can be used as a writeable stream, where data is written on the stream to be validated against a signature.

7.4.2 verify.update(data *[, inputEncoding]*)

Updates the verify object with the given data. If data is of type string, an optional *inputEncoding* argument can be provided that sets the encoding of this string. If data is not a string, inputEncoding is ignored. This function can be called as many times as needed until *verify.verify* is called. An example can be found in figure 7.5.

7.4.3 verify.verify(object, signature *[, signatureEncoding]*)

object represents the public key. It can be either a string containing a PEM-encoded object (RSA public key, DSA public key or X.509 certificate), or it can be an object with one or more of the following properties:

- key (required): a PEM encoded public key
- padding: An optional value for the RSA padding that can be one of the constants crypto.constants.RSA_PKCS1_PADDING (default) or crypto.constants.RSA_PKCS1_PSS_PADDING
- saltLength: In case, the padding value of crypto.constants.RSA_PKCS1_PSS_PADDING is chosen, this optional salt length can be set. crypto.constants.RSA_PSS_SALTLEN_DIGEST sets the salt length to the digest size, crypto.constants.RSA_PSS_SALTLEN_MAX_SIGN sets it to the maximal size.

The *signature* argument is the signature that has previously been calculated for the data that has been put into the *verify* object. If *signatureEncoding* is provided, *signature* is expected to be a string, otherwise a buffer.

The function returns *true* if the signature can be verified with the data put on the verify object and *false* otherwise. The *verify* object cannot be used again after this function has been called. Multiple calls to it will result in an error being thrown. An example can be found in figure 7.5.

Figure 7.5: Verify update and verify example

```
const crypto = require('crypto');

const { privateKey, publicKey } = crypto.generateKeyPairSync('rsa', {
  modulusLength: 2048,
});

const sign = crypto.createSign('SHA256');
sign.update('some data to sign');
sign.end();
const signature = sign.sign(privateKey);

const verify = crypto.createVerify('SHA256');
verify.update('some data to sign');
verify.end();
console.log(verify.verify(publicKey, signature));
```

7.5 Certificate

A public key certificate, also known as digital certificate is used to prove the ownership of a public key. It contains some information about the key, the owner and a digital signature of some entity that has verified the certificates content. In a public key infrastructure, this entity is called a certificate authority.

Usually, this works by a subject creating a key pair (private/public). Then a certificate signing request (a message to a certificate authority to apply for a digital certificate) Is sent. This usually contains the public key for which a certificate should be issued, identifying information and integrity protection (e.g. a digital signature). The most common format for such a request is the PKCS#10 specification. Another format is SPKAC which contains a challenge (a primitive form of proof of possession of the private key) and the public key.

The Node Certificate class is provided for working with such data in the SPKAC data. The most common usage is handling data generated by the HTML5 <keygen> element. This element is **deprecated**!

7.5.1 Certificate.exportChallenge(spkac)

spkac is the above-mentioned request in the spkac format. This function exports the challenge from the request and returns it as a buffer. An example can be seen in the figure 7.6.

Figure 7.6: Certificate exportChallenge example

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
const challenge = cert.exportChallenge(spkac);
console.log(challenge.toString('utf8'));
// Prints: the challenge as a UTF8 string
```

Figure 7.7: Certificate exportPublicKey example

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
const challenge = cert.exportPublicKey(spkac);
```

7.5.2 Certificate.exportPublicKey(spkac *[, encoding]*)

Takes *spkac* data as an input. If it is of type string, the optional encoding argument sets the encoding. This function exports the public key from the request and returns it as a buffer. An example can be found in figure 7.7

7.5.3 Certificate.verifySpkac(spkac)

Takes *spkac* data as input and verifies the data structure. Returns true if the data structure is valid and false otherwise. An example can be found in figure 7.8

7.5.4 General rules

Historically, SPKACs is a mechanism implemented to work as part of the HTML5 $keygen$[14] element. This feature has been deprecated. Therefore, you should avoid using it and update existing code. An alternative is to use the crypto.generateKeyPair function.

There are also some general rules about keys you should follow. In this case you do not worry about key generation since you just export them from the spkac, therefore the

[14]https://developer.mozilla.org/en-US/docs/Web/HTML/Element/keygen

Figure 7.8: Certificate verifySpkac example

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
console.log(cert.verifySpkac(Buffer.from(spkac)));
// Prints: true or false
```

important aspects here are safe distribution and safe storage.

- Never secret store the key in plaintext anywhere.
- Do not use the same secret key in another place of your application. A single secret key should be used only for one purpose[15].

7.6 Cipher

In cryptography, a cipher (sometimes also written cypher) refers to an algorithm for encryption or decryption through a series of well-defined steps that can be followed. The term itself is also used for the encrypted information that results from the process.

In Node, there are two functions to create a cipher object: *crypto.createCipher* and *crypto.createCipheriv*. The first function is **deprecated** and should not be used. Therefore, it is also not covered by this guide.

7.6.1 crypto.createCipher(algorithm, key, iv *[, options]*)

Creates and returns a cipher object. The algorithm is dependent on the OpenSSL version. You can get the available algorithms by using the *crypto.getCiphers()* function or by running the openssl list -cipher-algorithms command. The key is the raw key that is used. Iv stands for the initialization vector and is a fixed-size pseudo-random input. Both the *key* and the *iv* must be utf-8 encoded. The optional *options* argument controls the stream behavior. If a cipher in CCM or OCB mode is used, options is required and must have the authTagLength option set.

7.6.1.1 Using the cipher as a stream

The cipher object can also be used as a stream that is both readable and writeable. Here, plain unencrypted data is written on the writeable part and the encrypted part can then be read from the readable part.

7.6.1.2 Crypto-related danger

Not all algorithms are a suitable choice for a certain application. While it is mostly dependent on the range of the application, it is recommended not to use the original Data encryption standard (DES). Weak keys or wrongly stored keys are attack surfaces just like having a predictable initialization vector.

[15] https://www.owasp.org/index.php/Key_Management_Cheat_Sheet#Storage

7.6.1.3 Rules

- Choose a strong and suitable algorithm. Recommended algorithms are Triple DES, RSA, Blowfish, Twofish or AES. [16]
- It is recommended that the length of the secret key be at least the size of the blocks of the underlying hash function. [17]
- The secret key must be generated using a strong cryptographic pseudo-random function.
- Never secret store the key in plaintext anywhere.
- Don't use the same secret key in another place of your application. A single secret key should be used only for one purpose. [18]

7.6.2 cipher.getAuthTag()

When using an authenticated mode (one of GCM, CCM and OCB) is used, returns the authentication tag that has been computed from the given data in a Buffer. This function should only be called after the encryption has been completed by the *cipher.final* function.

7.6.3 cipher.setAAD(buffer *[, options]*)

When using an authenticated mode (one of GCM, CCM and OCB) is used, this function is used to set the additional authentication data (AAD). The *options* parameter is optional for GCM and OCB, but must be provided for CCM and must have the *plaintextLength* option set. Returns the cipher for method chaining.

7.6.4 cipher.setAutoPadding(*[autoPadding]*

When a block encryption algorithm is chosen, the cipher class automatically adds padding to the input data to reach the block size. This can be disabled by using this function and setting *autoPadding* to false. The input data to the cipher must then be a multiple of the block size or *cipher.final* will throw an error.

[16] https://blog.storagecraft.com/5-common-encryption-algorithms/
[17] https://tools.ietf.org/html/rfc4868#ref-HMAC
[18] https://www.owasp.org/index.php/Key_Management_Cheat_Sheet#Storage

Figure 7.9: Cipher example

```
const crypto = require('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';
// Use the async `crypto.scrypt()` instead.
const key = crypto.scryptSync(password, 'salt', 24);
// Use `crypto.randomBytes` to generate a random iv instead  iv
// of the static shown here.
const iv = Buffer.alloc(16, 0); // Initialization vector.

const cipher = crypto.createCipheriv(algorithm, key, iv);

let encrypted = cipher.update('some clear text data', 'utf8', 'hex');
encrypted += cipher.final('hex');
```

7.6.5 cipher.update(data *[, inputEncoding][, outputEncoding]*)

Updates the *cipher* with the given data. If *data* is of type string, the optional *inputEncoding* argument sets its encoding. Otherwise, *data* is expected to be a buffer. If the optional *outputEncoding* parameter is given, a string of the given data encrypted in the given encoding is returned. Otherwise, a buffer is returned. The *cipher.update* function can be called as many times as needed until *cipher.final* is called. An example can be found in figure 7.9.

7.6.6 cipher.final(*[outputEncoding]*)

Returnes any remaining contents enciphered and closes the cipher object for future calls. If the optional *outputEncoding* attribute is given, a string in the given encoding is returned. Otherwise, a buffer is returned. Future calls of the *cipher.final* or *cipher.update* will result in an error being thrown. An example can be found in figure 7.9.

7.7 Decipher

The decipher class is used to decrypt data that has been encrypted by the cipher class.

7.7.1 crypto.createDecipherIV(algorithm, key, iv *[, options]*)

Creates and returns a decipher object. The algorithm is the same that has been used for the cipher instance. You can get the available algorithms by using the *crypto.getCiphers()* function or by running the openssl list -cipher-algorithms command. The key is the raw

key that has been used to encrypt the data. Iv stands for the initialization vector that has been used for the encryption and is a fixed-size pseudo-random input. Both the key and the iv must be utf-8 encoded. The optional options argument controls the stream behavior. If a cipher in CCM or OCB mode is used, options is required and must have the *authTagLength* option set. An example can be found in figure 7.10.

7.7.1.1 Using the decipher object as a stream

The decipher object can also be used as a stream that is both readable and writeable. Plain encrypted data is written on the writeable side and the decrypted data can then be read from the readable side.

7.7.2 decipher.setAAD(buffer *[, options]*)

When using an authenticated mode (one of GCM, CCM and OCB) is used, this function is used to set the additional authentication data (AAD). The *options* parameter is optional for GCM and OCB, but must be provided for CCM and must have the *plaintextLength* option set. Returns the decipher for method chaining.

7.7.3 decipher.setAuthTag(buffer)

When using an authenticated mode (one of GCM, CCM and OCB) is used, *decipher.setAuthTag* is used to pass in the received authorization tag. If not tag is provided or if the cipher tag has been tampered with, the *decipher.final* function will throw an error, indicating that the text should be discarded since authentication has failed. If the tag length is invalid[19] or does not match the value of the *authTagLength* option, this function will throw an error.

7.7.4 decipher.setAutoPadding(*([autoPadding]*)

When the data has been encrypted without standard block padding (*cipher.setAutoPadding(false)* has been called), calling *decipher.setAutoPadding(false)* will prevent the *decipher.final* function from checking and removing padding. This works only if the input length is a multiple of the block size of the cipher algorithm.

7.7.5 decipher.update(data *[, inputEncoding][, outputEncoding]*)

Updates the decipher with the data encrypted by the *cipher*. The optional *inputEncoding* argument should match the *outputEncoding* argument from the *cipher.update* call. The

[19] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

Figure 7.10: Decipher example

```
const crypto = require('crypto');

const algorithm = 'aes-192-cbc';
const password = 'Password used to generate key';
// Use the async `crypto.scrypt()` instead.
const key = crypto.scryptSync(password, 'salt', 24);
// The IV is usually passed along with the ciphertext.
const iv = Buffer.alloc(16, 0); // Initialization vector.

const decipher = crypto.createDecipheriv(algorithm, key, iv);

// Encrypted using same algorithm, key and iv.
const encrypted =
  'e5f79c5915c02171eec6b212d5520d44480993d7d622a7c4c2da32f6efda0ffa';
let decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
console.log(decrypted);
// Prints: some clear text data
```

optional *outputEncoding* argument can then be whatever encoding is desired. If the encoding is not provided, *data* is treated as a buffer, respectively the return value of the function is a buffer. An example can be found in figure 7.10.

7.7.6 decipher.final(*[, outputEncoding]*)

Locks the decipher object and returns any remaining deciphered contents. If the optional *outputEncoding* argument is provided, a string in this encoding is returned. Otherwise, a buffer is returned. Once this function has been called, the object can no longer be used to decrypt data. An example can be found in figure 7.10.

Bibliography

[1] Elaine Barker, William Burr, Alicia Jones, Timothy Polk, Scott Rose, Miles Smid, and Quynh Dang. Recommendation for key management part 3: Application-specific key management guidance. *NIST special publication*, 800:57, 2009.

[2] Somak Das, Vineet Gopal, Kevin King, and Amruth Venkatraman. IV= 0 security: Cryptographic misuse of libraries. *Massachusetts Institute of Technology, Final Rep*, 6, 2014.

[3] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84, 2013.

[4] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security APIs. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[5] Mohammadreza Hazhirpasand, Mohammad Ghafari, Stefan Krüger, Eric Bodden, and Oscar Nierstrasz. The impact of developer experience in using Java cryptography. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6. IEEE, 2019.

[6] Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. Crypto-explorer: An interactive web platform supporting secure use of cryptography APIs. *arXiv preprint arXiv:2001.00773*, 2020.

[7] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic APIs. *IEEE Transactions on Software Engineering*, 2019.

[8] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail? A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, pages 1–7, 2014.

[9] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946, 2016.