Format Preserving Encryption

Analysis and implementation of FF1 and FF3-1

Gustav Grønvold s184205
Tobias H. Møller s184217
Anton K. Kirk s184191

Supervised by Lars Ramkilde Knudsen

Danmarks
Tekniske
Universitet

DTU Compute
Danmarks Tekniske Universitet
29-11-2021

DTU Compute
Department of Applied Mathematics and Computer Science

Abstract

This thesis has been written with the goal to make FF1 and FF3-1 more accessible. To realize this goal, a library to encrypt/decrypt with FF1 and FF3-1 has been designed and implemented. Before designing the library, we thoroughly analyzed FF1 and FF3-1 as well as related topics. This was done to acquire the knowledge base needed to ensure a secure and optimal design. We designed the library with a focus on user-friendliness and functionality. We also defined milestones for our project to help with project management. After designing the project, the library was implemented in Python. This implementation has been thoroughly examined and is explained in detail in this thesis. Some performance-critical sections of the implementation have been rewritten as Cython code. These sections have been directly compared to their Python counterparts. The library was then evaluated through tests in usability, security, and performance. This was done to document if, and to what extent, we had realized the goal.

Contents

1 Introduction

Format Preserving Encryption (FPE) is a relatively new topic within Cybersecurity. It allows for encryption of data without changing the format, e.g. encrypting a valid Credit Card Number (CCN) into another random valid CCN. Being a new topic within Cybersecurity, FPE was lacking in standards and generalization. M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers provided this in 2009 in the book *Selected Areas in Cryptography*, where they formally define FPE and investigate how to achieve FPE on complex domains[11]. Following this, the FFX and BPS modes of operation were submitted to NIST[2][3]. Both use Feistel-based encryption with a pseudorandom function (PRF) such as AES to achieve security. In 2016, NIST then released a publication with recommendations for FPE algorithms. The publication contains two algorithms: FF1 based on FFX and FF3 based on BPS. In response to an analysis on FF3 by Durak and Vaudenay[5], the publication was updated and a revised version of FF3 called FF3-1 was released in 2019[6].

FF1 and FF3-1 are both defined as block-cipher modes of operation, meaning an input is divided into "blocks" of a specific size, and then each block is encrypted. How each block is encrypted depends on the mode of operation, and is specified in FF1 and FF3-1 in this case. Common for each block-cipher mode of operation is the use of a PRF, where AES is considered to be the standard. This also means AES can be hardware accelerated on most CPUs for extra performance.

The digitization of personal data led to the introduction of GDPR in 2016, which sparked a growing interest in online privacy. The COVID-19 outbreak has even further increased digitization, and put a spotlight on privacy within healthcare. Personally Identifiable Information (PII) is a term used for data that can be used to identify a person, and can be E-mails, CCNs, SSN, Names, etc. Encrypting PII through normal means would provide privacy. However, the data would lose all information. FPE is used for situations, where privacy is needed, but some information must be left intact, e.g. a CCN is still identifiable as a CCN. For this reason, FPE is a tool that has a lot of potential, but is still relatively unknown.

The goal of this project is to make FF1 and FF3-1 accessible to a wide range of developers and in all kinds of projects. To realize this goal, we will implement an FPE library. This library will contain functions to encrypt and decrypt inputs of certain formats. Additional functionality like encrypting databases will also be considered. Furthermore, the library will be tested to ensure security and usability.

2 Research Questions

Multiple questions need to be investigated before we can realize the goal of the project. The main question being:

> *How do we implement a library that makes FF1 and FF3-1 accessible to a wide range of developers and in all kinds of projects?*

This question raises several new questions. The first question we will need to answer is:

> *What language should we choose to implement the library in?*

The programming language we choose will have a big impact on the final product. The programming language chosen should reflect the goal. This means we naturally should not choose a language that barely anyone uses. We should also avoid niche languages only used in applications, where FPE is irrelevant. It is also important that we consider languages we are proficient with. The limited time frame means we will not have time to learn a new language before starting. The next question we need to answer is:

> *How do we make the library user-friendly and functional?*

If we want FF1 and FF3-1 to be available for a wide range of developers, the library needs to be user-friendly. If the library is too difficult to install or use, then we have not reached the goal. If we want the library to be used in many different projects, it needs to be functional. This means the library needs to have some general functions that are usable in many scenarios. The next question to investigate is:

> *How do we confirm the security of the library?*

We generally rely on FF1 and FF3-1 being secure. However, this does not necessarily mean our implementation is secure. That entirely depends on us being capable of implementing it in conformity with NIST's requirements. Furthermore, any functionality we add to the library must also not break the security of the library. We need to confirm, to the extent possible, the security of our program. The last question to answer is:

> *How do we finish the library within the time frame of the project?*

We must ensure we have a finished product before the deadline of the project. This means we need to define the scope of the project and a plan for reaching the goal. We need to be able to follow the progress of our project so that we can revise the

plan in case we are behind or ahead.

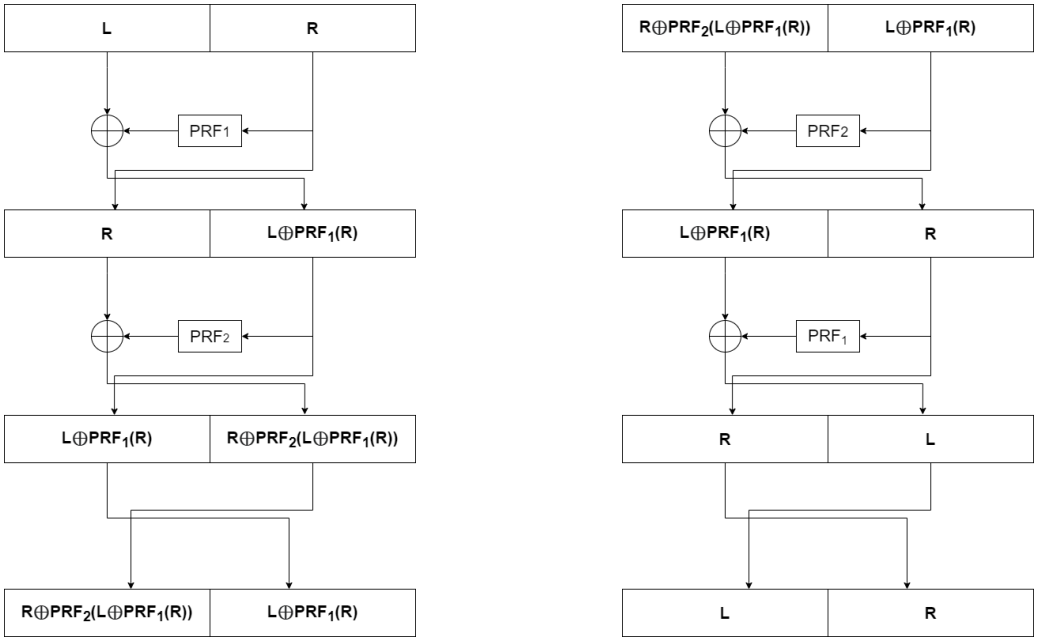
If we can answer all of these questions, we have confidence that the goal of the project can be realized.

3 Analysis

The goal of the analysis is to lay the groundwork for investigating our research questions. Therefore, we will in the following chapter explain and connect concepts in the domain of format preserving encryption algorithms. Specifically, we will discuss Feistel Networks, choice of programming language, different types of plaintext formats, pseudorandom functions, and, most importantly, the format preserving algorithms FF1 and FF3-1.

3.1 Feistel Network

Feistel networks serve as the framework upon which FF1 and FF3-1 are built. A Feistel network starts by splitting the input into left and right parts. Both parts are encrypted through several rounds, where each round consists of a PRF and a combining function. This framework for encryption is unique in the sense that the PRF used in each round can be irreversible such as a hashing function, yet the ciphertext can still be decrypted with the same PRF. This is due to the combining function, which combines the left and right parts in each round. The combining function must have an inverse for decryption, e.g. you can combine the two parts by multiplying them in encryption, and by dividing them in decryption. A typical method used for combining the two parts is by XORing them, since XOR can be reversed by XORing with the same input. An example Feistel Network with 3 rounds is illustrated in Figure 1 below, with any PRF, and XOR as the combining function.



(a) Encryption (b) Decryption

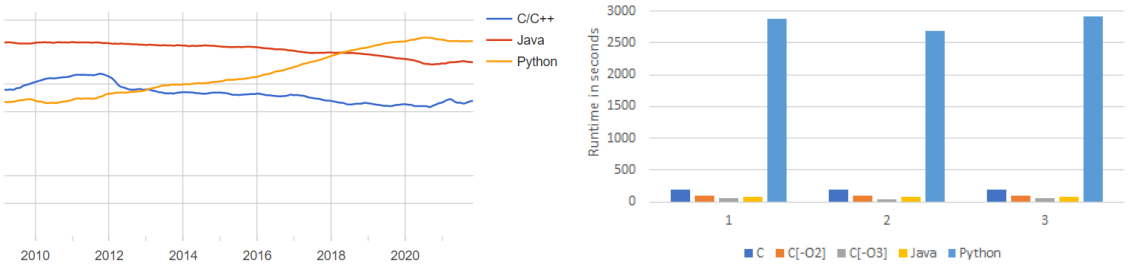
Figure 1: Feistel Network

Figure 1 shows how the left (L) and right (R) part is encrypted with a PRF and then combined with XOR. To decrypt this, the Feistel network is flipped upside down, and both parts are again encrypted with PRF and combined with XOR. However, this time the first XOR would be $(R \bigoplus PRF_2(L \bigoplus PRF_1(R))) \bigoplus PRF_2(L \bigoplus PRF_1(R))$, which means the $PRF_2(L \bigoplus PRF_1(R))$ part will cancel out, and the only thing left is R. The same happens with $L \bigoplus PRF_1(R)$ leaving only L, and then the message is decrypted and just needs to be flipped.

3.2 Programming Language

The programming language chosen should reflect the goal of making FPE more accessible, so only widely used languages will be considered. Among the most popular languages are Python, Java, and C at top 3, according to the TIOBE and PYPL index [16][17]. Just considering popularity, Python would be the obvious choice, since it is not only the most popular language, but it is rising in popularity, while C and Java are descending, as seen in Figure 2a. However, another key factor to consider is the relative runtime of the code, since cryptology is reliant on fast runtimes. A

general rule is that C is slightly faster than Java, while Python is many times slower than both. Figure 2b shows the runtimes for 3 iterations of a matrix multiplication program, with a 2048×2048 matrix in C, Java, and Python. While using the base GCC compiler, C runs slower than Java, however, when you add optimization flags (-O2, -O3) to the compiler the runtimes improve. As seen in the figure, the -O3 flag makes C slightly faster than Java, for this specific application. These results will vary depending on the application, but can be used as a general model for the runtimes.



(a) Popularity of Java, Python and C

(b) Runtime of Java, Python and C

Figure 2: Comparison of Java, Python and C

For cryptology purposes, you generally want C-like performance, which is why Python rarely is the language of choice when implementing cryptology applications. C or Java would provide the performance needed, but suffer from a lack of ease of programming. Python is a dynamically written language. This makes it an easy programming language to learn and write, which is part of the reason for Python's popularity compared to C and Java. One way to overcome this is by using Cython, which lets you define variables as in C. This can improve the runtime of Python, but the ease of programming suffers as a result.

3.3 Formats

Since FF1 and FF3-1 use sequences of numbers as data input and output, it is necessary to devise a way to represent nonnumerical data to use the ciphers to encrypt and decrypt them. Both numerical and nonnumerical data can be encoded in different formats. These formats may be as simple as a set of allowed symbols, or they may involve complicated syntactical rules. In the following sections, we will describe a handful of such types of data, first by introducing the notion of an

alphabet, and then discussing the most important formats that we have decided to support in our library.

3.4 Alphabet

An alphabet is defined as a non-empty, finite set, consisting of a number of symbols[6]. An example of an alphabet is the set of lowercase English letters,

$$\{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}, \mathrm{g}, \mathrm{h}, \mathrm{i}, \mathrm{j}, \mathrm{k}, \mathrm{l}, \mathrm{m}, \mathrm{n}, \mathrm{o}, \mathrm{p}, \mathrm{r}, \mathrm{s}, \mathrm{t}, \mathrm{u}, \mathrm{v}, \mathrm{w}, \mathrm{x}, \mathrm{y}, \mathrm{z}\},$$

or the first ten natural numbers, e.g. $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Using the first alphabet, the character strings `hello` and `will` can be expressed, but not `Hello` and `won't`, since the symbols "`H`" and "`'`" are not contained in the alphabet. Similarly, the set of all natural numbers is not an alphabet, since there is an infinite amount of natural numbers.

3.5 Strings

A string is a sequence of characters. Since the symbols used to represent characters are a finite set, a natural way to represent strings follows directly from the specification given by NIST. We simply define an alphabet with radix n, where n is the number of characters that we wish to support. The characters are then encoded using a mapping between two sets. One being the alphabet Σ that contains the numbers 0-9, lowercase and uppercase letters, as well as any other symbols, we wish to support (spaces, punctuation, etc.)

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, \ldots\}.$$

The other being the numerals from 0 to $n - 1$. The alphabet can then be mapped directly to numerals.

$$0 \to 0, 1 \to 1, 2 \to 2, \ldots, a \to 10, b \to 11, c \to 12, \ldots$$

This is similar to how characters are encoded using ASCII[7].

3.5.1 Credit Card Numbers

A credit card number (CCN) is a type of payment card number and functions as a card identifier. The first digit of a credit card is the major industry identifier

number that identifies the industry of the issuer of the card (denoted with a **1** on fig. 3). The first 6 digits (**2**) (including the major industry identifier) is the Issuer identifier number (INN). The INN is much more specific than the major industry identifier[8]. It is originally 6 digits but as of 2015, it has gradually been moving to 8 digits[18]. The next nine digits are associated with the cardholder, as well as the cardholder's bank account (**3**). The last digit (**4**) is the checksum, which is used to validate the credit card number.

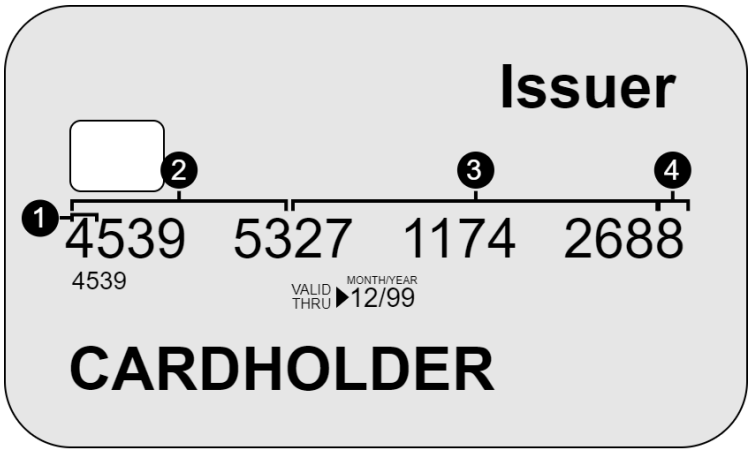


Figure 3: Illustration of a credit card.

The checksum is calculated using the Luhn algorithm[8]. The Luhn algorithm is commonly used to validate credit card numbers, but is also used to validate other identification numbers. In table 1, an example is given of how steps 1 and 2 of the algorithm would work, given the credit card number "4539 5327 1174 2688".

1. Take the number, excluding the last digit which is the checksum (for 16-digit credit cards, take the first 15). Starting from the rightmost digit moving left, double every other digit, and keep the other digit as they are. If a given digit exceeds 9, subtract 9.
2. Add the digits together to a sum s
3. The checksum digit is then $10 - s \mod 10$.

Card #	4	5	3	9	5	3	2	7	1	1	7	4	2	6	8	-
x2	8	5	6	9	10	3	4	7	2	1	14	4	4	6	16	-
x2 mod 9	8	5	6	9	1	3	4	7	2	1	5	4	4	6	7	-
Sum	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	72

Table 1: The Luhn Algorithm example

The sum we found in step 2 is shown to be 72. The calculation for step 3 is shown below:

$$10 - 72 = -62 \implies -62 \equiv 8 \bmod 10$$

As seen above, the checksum is 8, which corresponds with the last digit of the given credit card number.

3.5.2 CPR numbers

A CPR number in Denmark is a 10-digit civil registration number used to identify a person in the Danish Civil Registration System (Danish: Det centrale personregister). The first 6 digits record the date of birth of the person (DD/MM/YY), and the four last digits (SSSS) are a sequence number that also encodes sex (odd for biological males, and even for biological females). The first digit of the sequence number also determines which millennium the person was born in.

`DDMMYY-SSSS`

CPR numbers also have a control digit. The last digit in the number is calculated (and verified) by using a "modulo 11 check": each of the 9 first digits is multiplied by a specific number, and the resulting sum is reduced modulo 11. The control digit is then found by subtracting this number from 11. In the case of 11 - 1 = 10, the control digit is invalid and will not be used. If an invalid CPR number is encountered, you should simply look for another set of sequence numbers, so that it becomes a valid CPR number[4].

An example calculation is given below.

d	d	m	m	y	y	s_1	s_2	s_3
0	7	0	7	6	1	4	2	8
$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
4	3	2	7	6	5	4	3	2
$=$								
0	+21	+0	+49	+36	+5	+16	+6	+16

The sum of the last row of the matrix is 149, which is 6 modulo 11. The last digit is then given by

$$s_4 = 11 - 6 = 5.$$

Hence, the control digit for the CPR number 070761-428 is 5.

Today, there is a problem with the modulo 11 check that forced us to leave it behind a little. The space of valid CPR numbers is very small, and because of this, there are not enough CPR numbers for our entire population. There are between 360 and 540 valid CPR numbers that are valid per day depending on the year you look at. This became a problem when the population started to grow, and we had to give out more CPR numbers than the space allowed. It was decided that any CPR number issued after the entire space is used up, simply won't have any validation. The new CPR numbers are issued such that (for males) the first sequence number issued is '0001', and then every followed number is incremented by 6. Once we can't go any higher, we start again at '0003' and increment by 6. and then at last we start again at '0005' and increment by 6. The same goes for females, just with '0000', '0002' and '0004', still incremented by 6. This expands the space of CPR numbers by just more than 10 times, but the rule that the first number of the sequence numbers states what millennium you are born in still stands. This new addition to the CPR numbers expands the space, giving a total of 4000-6000 CPR numbers per day, depending on the year you look at.

3.5.3 Email addresses

An email address is used as an identifier to an email box. The format of an email address is *local-part@domain* (illustrated in fig. 4), where domain is the domain of the host to which the email is to be delivered (either an Internet domain name, or a literal Internet address), and local-part is a domain-dependant string identifying

a particular mailbox on that host. The format for an email address is specified in RFC 5322 (sections 3.2.3 and 3.4.1)[15] and RFC 5321[10].

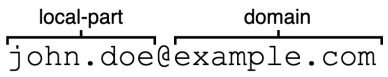


Figure 4

Local-parts may be up to 64 octets (characters) long and can contain a subset of the ASCII character encoding. Specifically, they can contain alphanumeric characters, or any of the special characters

```
! # $ % & ' * + - / = ?  ^ _ ` { | } ~
```

as well as a period (”.”). Periods may not be consecutive, nor may they be placed at the start or end of the local-part[9]. The local-part in figure 4 is `john.doe`. It should be noted that many local-parts live up to the requirements set by the RFC specification, but not all are accepted by every organization or mail server.

The email address must contain a domain after the '`@`'-sign, and it must be a valid hostname. In other words, the domain must comply with the following rules set by the RFC 1034[12]:

- It must be composed of a sequence of labels concatenated with dots.
- Each label must be from 1 to 63 characters long.
- The entire hostname (including dots) must have a maximum length of 255 characters,
- and the characters must come from the subset of ASCII characters that consists of the letters `a-z`, the digits 0-9, and the hyphen '`-`'.

In the email in figure 4, the hostname consists of two labels: a domain name (example), and a top-level domain (.com). Top-level domains are restricted to an approved list of domains maintained by ICAAN[1].

3.6 Pseudorandom function

The PRF is used both in FF1 and FF3-1. The PRF can be any cryptographically secure 128-bit block cipher. NIST recommends using AES since it is a standard

[1]Internet Corporation for Assigned Names and Numbers

created by NIST, but an alternative to AES could be ChaCha, both of which will be described in the following sections. Note that certain details from the ciphers will be omitted to provide a simple and general explanation.

3.6.1 AES

The Advanced Encryption Standard (AES), also known as Rijndael, is one of the most used encryptions. AES is a block cipher, and it takes 128 bits of information, and encrypts them using a key with a length of either 128, 192, or 256 bits. AES then uses transformation rounds to obscure the output, and there are either 10, 12, or 14 rounds, depending on the key size. AES operates with 4-by-4 grids, thus operating on 16 bytes at a time (though multiple blocks can be calculated in parallel). The grids are arranged in a way such that the bytes are put in order from the top-left corner, downwards, continuing on the next column, also shown in the tables below.

B0	B1	B2	B3	...	B12	B13	B14	B15

B0	B4	B8	B12
B1	B5	B9	B13
B2	B6	B10	B14
B3	B7	B11	B15

One typical round of transformations includes the following:

1. Substitution of bytes, where every byte is changed according to a lookup table.
2. Each row is shifted a different number of times, so the table is more mixed.
3. Each column is mixed, using the cross product of the given row and a carefully chosen matrix with special properties.
4. Adding round-key by X-or.

The exception to this is that the first round starts by adding an initial key just like with the round-keys, and the last round skips step 3, because it doesn't improve the security, and thus is wasted run-time.

AES is considered to be very fast. The fast runtime of AES due to it being part of the instruction set in most processors, makes it optimal for FPE.

3.6.2 ChaCha

ChaCha, also known as the ChaCha-20 cipher, is a refinement of the cipher 'Salsa20'. ChaCha is the primary competitor to AES, as it is almost as fast, and like AES, ChaCha is deemed secure by researchers worldwide. The only reason AES is faster is that most processors today have built-in AES instructions, to optimize the runtime of AES. On machines that are not optimized to run AES, primarily older machines, ChaCha would generally be faster. ChaCha is a stream cipher, which uses a 256-bit key, and it uses its round function 20 times. It also uses a nonce and a block number along with the key to generate a stream of keys to encrypt a stream of blocks. One key encrypts one byte worth of information, and the way ChaCha encrypts is simply by XORing the key and the message, so to decrypt it, you would just have to go the other way, and XOR the ciphertext with the key again. This process is also illustrated in the picture below.

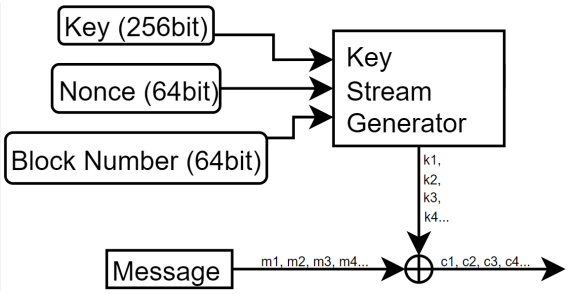


Figure 5: The Flow of ChaCha

Just like AES, ChaCha also uses a 4-by-4 grid, but AES uses it to encrypt a block of information, and ChaCha uses it to generate the keystream. The 4-by-4 grids used in ChaCha has a structure as shown below, where the Cs are a known fixed string of characters, that are always the same, to help mix everything around to make it extra secure in some edge-cases. The Ks are the key, the Bs are the Block Numbers, and the Ns are the nonce. It is important to note that each field in this 4-by-4 grid is 32 bits, that way it also shows that the key is $8 \times 32 = 256$ bits, the nonce is $2 \times 32 = 64$ bits and the block number is $2 \times 32 = 64$ bits.

C	C	C	C
K	K	K	K
K	K	K	K
B	B	N	N

ChaCha does not use confusion and diffusion to mix up the grid, as AES does. ChaCha only uses diffusion which if described simply means obscuring the grid by mixing the fields. ChaCha uses a round function to mix up the grid, and it uses this round function for 20 rounds. The round function consists of 4 rounds itself, where it mixes the fields at A, B, C and D. Each of the 4 sub-rounds, it shifts which fields are the A, B, C and D. This concept is illustrated in figure 6.

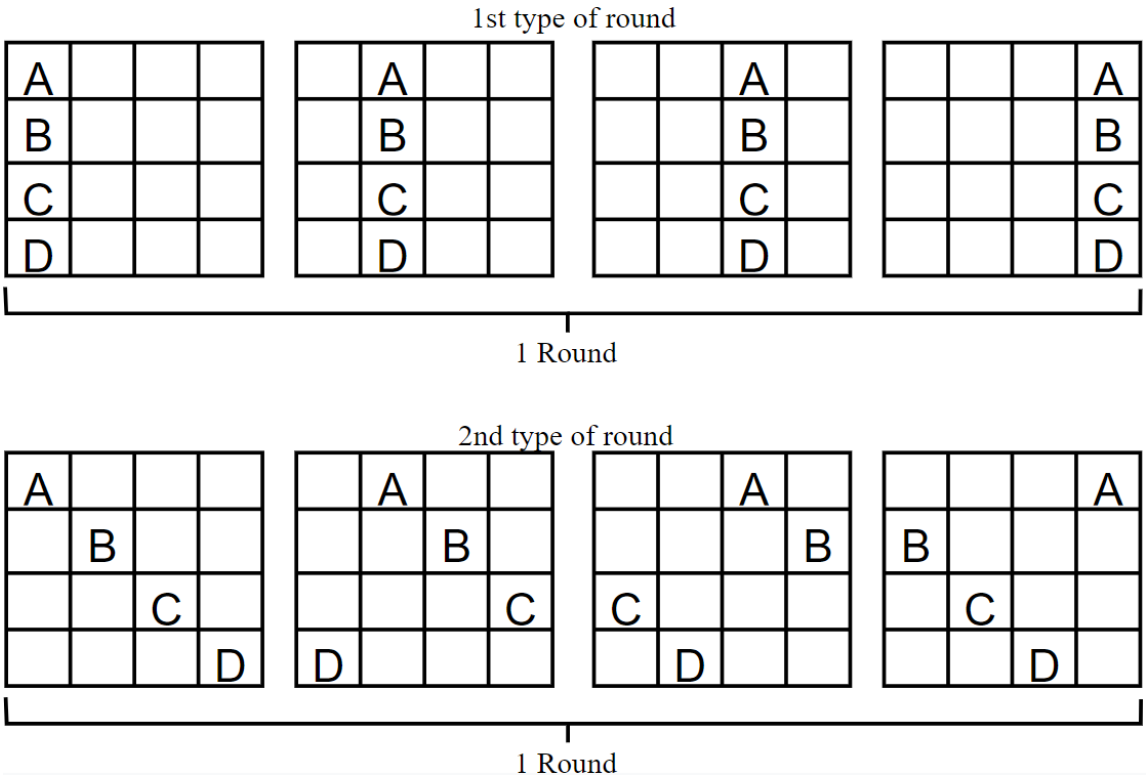


Figure 6: Illustration of the two types of round-functions used in ChaCha

As the figures above illustrate, there are two different types of rounds. One, where it mixes the 4 columns individually, and one where it is mixed diagonally. This is done because of the desired effect of a change in one field creating a change in the entire grid.

As stated earlier, ChaCha uses 20 rounds. Every other round is type 1 from the figure above, and the other is type 2, so ChaCha runs 10 of each type of round. No matter what type of round, the computation is still carried out on the letters A, B, C and D.

The computation that is done at every sub-round is illustrated below.

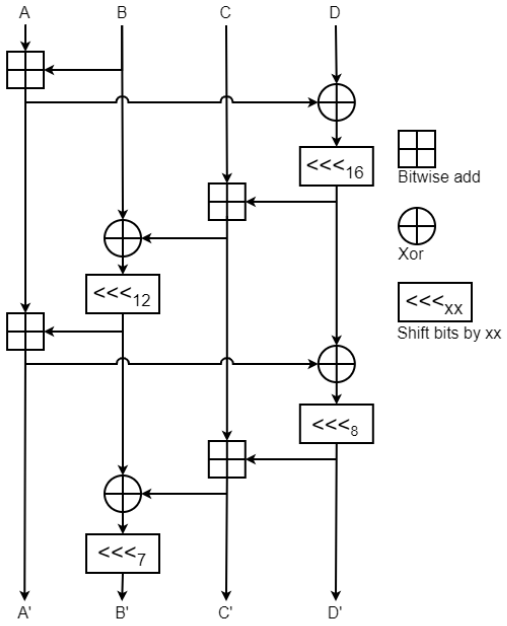


Figure 7: Illustration of the sub-round function used in ChaCha

3.7 Numerals

Numerals are used to represent any kind of input of any format in FF1 and FF3-1. Encryption and decryption require the input to be mapped to numerals, according to the alphabet of the format. An example input could be "hello", with the format being lowercase letters, meaning a-z is mapped to 0-25. The corresponding numeral list would then be [7,4,11,11,14], and after encryption the new random numeral list can be mapped back to letters. However, the numeral list must be represented as bytes for the PRF, and as an integer for addition, subtraction, and modulo. Translating integers to bytes is no issue, but translating a numeral list to an integer and back is harder. This is done through two functions: NUM_{radix} and STR_{radix}, seen in Algorithm 1 and 2. NUM_{radix} works by taking the first numeral as an integer. For each numeral after the first the integer is multiplied by radix and then the numeral is added. In the case of the sample numeral list, the integer would be $(((7 \times 26 + 4) \times 26 + 11) \times 26 + 11) \times 26 + 14 = 3,276,972$. This can also be seen as a specific combination of a total of 26^5 combinations, which means "aaaaa" is the

first combination $0^5 = 0$, and "zzzzz" is the last combination $26^5 - 1 = 11,881,375$. Modulo and integer division is used to convert from an integer back to a numeral list. The integer from NUM_{radix} can be written as $x \cdot radix + numeral$, so by using modulo the numeral can be retrieved. The integer can also be written as $(x \cdot radix + numeral_1) \cdot radix + numeral_2$, and since $numeral < radix$ the $radix + numeral_2$ part can be removed with floor division, leaving $x \cdot radix + numeral_1$. The last numeral in the example is 14 so $3,276,972 \bmod 26 = 14$, and then by floor dividing with 26 the number will be 126.033 which is $x \cdot 26 + 11$. This process is repeated until only the first numeral in the list is left.

Algorithm 1 Pseudcode for NUM_{radix}

```
Let x = 0
for i from 1 to LEN(X) do
    Let x = x × radix + X[i]
end for
return x
```

Algorithm 2 Pseudcode for STR_{radix}

```
for i from 1 to m do
    X[m + 1 − i] = x mod radix
    x = ⌊x/radix⌋
end for
return X
```

3.8 FF1

FF1 encrypts a numeral list keeping each numeral within the radix. To ensure the security of the algorithm there are certain requirements to the input.

- $radix \in [2..2^{16}]$
- $radix^{minlen} \geq 1,000,000$, and
- $2 \leq minlen \leq maxlen \leq 2^{32}$.

The full pseudocode for FF1 can be seen in appendix A, but will be examined step for step in this section. The code starts with calculating the length of u and v in step 1, which is used to determine the size of the left and right parts of the Feistel structure. u is defined as $\lfloor n/2 \rfloor$, where n is the length of the input. This definition

means that in case n is uneven, u will be smaller than v. In step 2, the input X is then split into two parts A and B, according to the sizes calculated in step 1. An example input could be "12345", in which case u would be 2 and v is 3. The input would then be split into $A =$ "12" and $B =$ "345". Note that this example does not fulfill the requirement of $radix^{minlen} \geq 1,000,000$. This is done to keep calculations simple, but will not work for the actual implementation.

Algorithm 3 FF1 Encryption, Step 1 & 2

Let $u = \lfloor n/2 \rfloor; v = n - u$
Let $A = X[2..u]; B = X[u + 1..n]$

In step 3 b is defined as the byte length of B, and is calculated as $\lceil \lceil v \cdot \log_2(radix) \rceil / 8 \rceil$. $\log_2(radix)$ computes the number of bits needed for each numeral. By multiplying this with v, the total number of bits needed to contain B is computed. This needs to be an integer value, but it has to be ceiled since flooring could make it one bit too small. This is then divided by 8 to get the size in bytes, and ceiled again for the same reason as before. In the example of the input of "12345" if the radix is 10, this would be $\lceil \log_2(10) \cdot v \rceil = 10$. With 10 bits the number of combinations are $2^{10} = 1024$, and with 3 digits the combinations are $10^3 = 1000$, so 10 bits is enough to represent 3 digits. This is then divided by 8 to get bytes $\lceil 10/8 \rceil = 2$, i.e. 2 bytes are needed to represent the 3 digits.

Algorithm 4 FF1 Encryption, Step 3

Let $b = 4\lceil \lceil v \times \text{LOG}(radix) \rceil / 8 \rceil$

In step 4 d is defined as a byte length that is at least 4 bytes larger than b. d is used in step 6.iii. to eliminate modulo bias. It is computed by ceil dividing b with 4, then multiplying by 4 and adding 4. This definition ensures that d is always a multiple of 4, which is important in step 6.iv. and 6.vi. In the example input of "12345", where b was 2, d would be $\lceil 2/4 \rceil \cdot 4 + 4 = 8$.

Algorithm 5 FF1 Encryption, Step 4

Let $d = 4\lceil b/4 \rceil + 4$

In step 5 P is created to use in the PRF function. P simply concatenates bytes from $radix$, u, n, t, and selected numbers to create a 16-byte string to use as initial input

for the PRF. The purpose of P is to use as the initial block in the PRF, so Q may be more securely encrypted.

Algorithm 6 FF1 Encryption, Step 5

Let $P = [1]^1||[2]^1||[1]^1||[radix]^3||[10]^1||[u \bmod 256]^1||[n]^4||[t]^4$

Step 6 begins the Feistel rounds, which code-wise is just a 10 round loop. The following steps are executed 10 times for each encryption and decryption and are therefore the most critical steps performance-wise. Step 6.i. is where the bytes to encrypt are computed, and is made using the *tweak*, i, some 0-bytes, and B, which is first converted to an integer in NUM_{radix} and then to bytes. The 0-bytes are used as padding to ensure Q has a multiple of 16 bytes. The number of 0-bytes are dependent on the size of T and B, and is calculated as $(-t-b-1) \mod 16$. In the example with $b = 2$ and a 4-byte tweak, the number of 0-bytes would be $(-4-2-1) \mod 16 = 9$. In step 6.ii. P and Q are then concatenated and used in PRF, seen in Figure 8.

Algorithm 7 FF1 Encryption, Step 6, 6.i & 6.ii

for i from 0 to 9: **do**
  Let $Q = T||[0]^{(-t-b-1) \bmod 26}||[i]^1||[\text{NUM}_{radix}(B)]^b$
  Let $R = \text{PRF}(P||Q)$
end for

The PRF starts by dividing the bytes into 128-bit blocks, and then "feeds" each block to the block cipher (AES, ChaCha, etc.). When encrypting a block, the previous cipher is used in the encryption, except for the first block where a default initial value is used. The initial value is 128-bits of zeros, which is why P is used as the first block instead of Q.

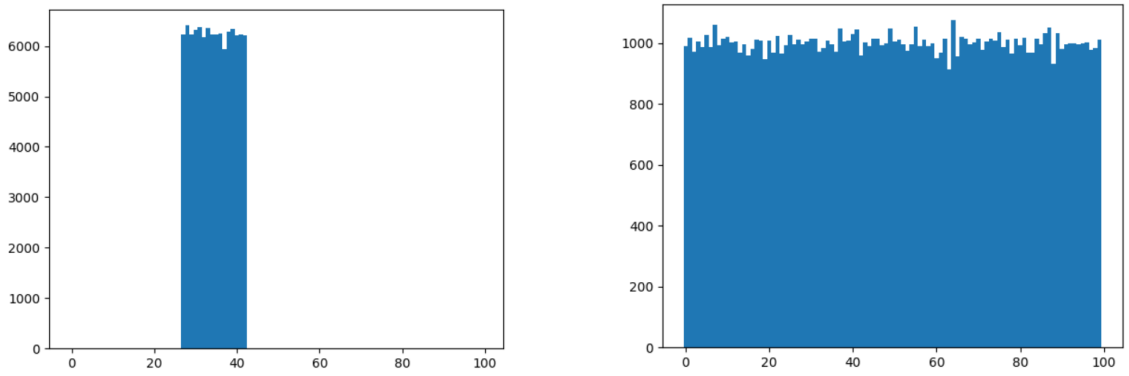
Algorithm 8 PRF Pseudocode

Let $m = \text{LEN}(C)/128$
Let $X_1, ..., X_m$ be the blocks for which $X = X_1||...||X_m$
Let $Y_0 = 0^{128}$, and for j to m let $Y_j = \text{CIPH}_g(Y_j - 1 \oplus X_j)$
Return Y_m

The PRF returns a 128-bit cipher, which is converted to an integer in step 6.iv, and then combined with A in step 6.vi. The combination of A and the cipher is done by adding them together modulo $radix^m$. This method of combining the inputs ensures the length of the input is preserved, but is vulnerable to modulo bias which can lead to attacks. To eliminate modulo bias d is used in step 6.iii. before the cipher is converted to an integer. Modulo bias happens in FF1 when the input is larger than the cipher. Figure 8a illustrates an example, where $A = 27$, $radix = 10$, and the cipher is a 4-bit random number. The combination is simulated 100,000 times, and should be evenly distributed over all 10^2 possible combinations. However, due to the small cipher, all combinations lay within $27 - 42$. To solve this, the cipher is changed to a random 16-bit number, and another 100,000 combinations are simulated as seen in figure 8b.

Algorithm 9 FF1 Encryption, Step 6iii - 6.vi

Let S be the first d bytes of the following string of $\lceil d/16 \rceil$ blocks:
$R||\text{CIPH}_K(R \oplus [1]^{16})||\text{CIPH}_K(R \oplus [2]^{16})...\text{CIPH}_K(R \oplus [\lceil d/16 \rceil - 1]^{16}$
Let $y =$ NUM(S)
if $i \bmod 2 = 0$ **then**
    Let $m = u$
else
    Let $m = v$
end if
Let $c = (\text{NUM}_{radix}(A) + y) \bmod radix^m$



(a) Biased graph

(b) Non-biased graph

Figure 8: Modulo bias

In step 6.iii., S is defined to be the first d bytes of the cipher, and in case d is more than 16 bytes, the cipher will be concatenated with 16 random bytes. This is done until the cipher is larger than d, and then the first d bytes of the cipher can be converted to an integer in 6.iv. This also ensures that the integer is not unnecessarily large, i.e. if d is smaller than the cipher, there is no need to use the entire cipher. This is also where it is important that d is a multiple of 4, since most programming languages do not support infinitely large integers without external libraries. For example, in C the largest integer is 8 bytes, which is too small, as soon as the input exceeds 8 bytes in size. To solve this, the cipher can be converted into multiple integers, each representing 4 bytes of the cipher. In that way, the integers never have to exceed 4 bytes, since the actual large integer is not needed for the modulo addition in step 6.vi. This is because modulo has the property that $(a+b) \mod n = ((a \mod n)+(b \mod n))mod$, which means that if y is represented as an array of 4 byte integers, instead of one large integer, then $y \mod radix^m$ can be computed as $(arr[i-1] \cdot 2^{32\cdot(i-1)} \mod radix^m) + (arr[i] \cdot 2^{32\cdot i} \mod radix^m) \mod radix^m$ for $i \in [1; (d/4) - 1]$. That means the numbers will not exceed $radix^m$, however, $radix$ has a max value of 2^16 and m of 2^31, which is $2^{16\cdot 2^31} = 2^{34359738368}$. Even if that is smaller than y would be, it is still a very large number that is hard to work with. This is why $radix^m$ can be divided into 4-byte integers as well. b can be used to compute the number of 4-byte integers needed for $radix^m$, by dividing b with 4 and ceiling it. This also means that y is represented as one 4-byte integer more than $radix^m$, which is important to keep in mind when computing the modulo. To avoid large numbers, the integers representing $radix^m$ can be computed as seen in algorithm10. The algorithm only requires p to be larger than 4 bytes to compute ints[j] $\cdot radix$, e.g. in C p would be a 'long long integer'.

Algorithm 10 $\text{SPLT}^{\text{m}}_{\text{radix}}$

```
Let n = ⌈b/4⌉
Let ints[0] = radix
for i from 0 to m do
    for j from n to 0 do
        p = ints[j] · radix
        if p ≥ 2^32 then
            ints[j+1] = ints[j+1]
            ints[j] = p mod 2^32
        end if
    end for
end for
```

The modulo of y can then be computed by subtracting the 4-byte integers of $radix^m$ from the 4-byte integers of y. However, this would require a large number of operations for large numbers. To reduce the number of operations required the integers of $radix^m$ can be left-shifted and then subtracted as in algorithm 11. The $\text{SPLT}^{\text{m}}_{\text{radix}}$ algorithm can also be used in $\text{NUM}_{\text{radix}}$ to avoid large numbers. Using these algorithms, operating on large numbers can be avoided entirely. The amount of bytes needed is still the same, so the memory required to encrypt large inputs is unchanged. The number of operations used for the 'Shift and Subtract' algorithm with any y, $radix$, and m is $d-b$, and since $d \leq b+7$ the runtime is O(7), which is constant time and can be discarded. For $\text{SPLT}^{\text{m}}_{\text{radix}}$ the number of operations are $b/4 \cdot m$, where $b \leq 16 \cdot 2^{31}$ and $m < 2^{32}$ which would be considered constant time. However, it is likely noticeable for large numbers. To help understand the 'Shift and Subtract' algorithm, an example with small numbers is given in appendix D.

Algorithm 11 Shift and Subtract

Input:
Integer, X, Y, represented as array of 4 byte integers
Integer, x, y, byte size of X and Y

Output:
Integer, m, result of X mod Y as array of 4 byte integers
Integer, d, result of X/Y

Let d = 0
if X < Y **then**
    return X, d
end if
Let u = LEN(X)
Let v = LEN(Y)
Let Z = Y $\parallel$ $[0]^{u-v}$
Let n = ((x-y) mod 4)-1
if n > 0 **then**
    Z << n·8
else
    Z >> 8
end if
Let a = 0
while Z $\leq$ Y **do**

```
    X = X-Z
    d = d + 1
    while X[a] < Z[a] and Z < Y do
        Z >> 1
        d = d · 2
        if Z[0] is 0 then
            a = a + 1
        end if
    end while
end while
return X, d
```

At Step 6.vii., c is converted back to numerals in the $\text{STR}^{\text{m}}_{\text{radix}}$ function. If c is represented as 4 byte integers, then $\text{STR}^{\text{m}}_{\text{radix}}$ can use the 'Shift and Subtract' algortihm as well. $\text{STR}^{\text{m}}_{\text{radix}}$ computes the mod of c and then divides c with $radix$. The 'Shift and Subtract' algorithm can compute both at once by counting each subtraction and multiplying by the rightshifts. This is what the d variable represents in the 'Shift and Subtract' algorithm.

Algorithm 12 FF1 Encryption, Step 6vii

Let $C = \text{STR}^{m}_{radix}(c)$

Step 6.viii. and ix. is swapping the variables so that both parts of the input get encrypted. This is the end of the 10 round loop that represents the Feistel structure, and then the result of the encryption is returned.

Algorithm 13 FF1 Encryption, Step 6.viii, ix & 7

Let $A = B$
Let $B = C$
Return $A||B$

The difference in encryption and decryption is minimal, and is mostly just reversing the encryption. Step 1-5 is unchanged from encryption to decryption, but in step 6 the loop is reversed so i goes from 9 to 0 instead. The decryption also swaps A and B in all steps. This is because i starts as 9 (uneven) in decryption, compared

to encryption, where i starts at 0 (even). The last important part is the combining method in step 6.vi. This combining method must have opposites for encryption and decryption, which is why addition is used in encryption and subtraction in decryption. The rule of addition in modulo also applies to subtraction, so the modulo can be computed as $((B \mod radix^m) - (y \mod radix^m)) \mod radix^m$. This means that the number that is used in the modulo might be negative, which means the shift and subtract algorithm must be able to handle negative numbers. However, in this case it is enough to simply check if the number is negative and add $radix^m$. This is because the negative number will never be less than $-radix^m$.

3.9 FF3-1

As with FF1, the format-preserving algorithm FF3-1 is also based on Feistel networks. In many regards, FF3-1 is similar to FF1: it uses a 128-bit block cipher as a round function, and encrypts and decrypts numeral strings. However, there are some differences: it uses fewer rounds (8), uses a different order of significance, and is more stringent with regards to the length of both message and tweak. These differences should lead to a performance gain in comparison to FF1.

FF3-1 has a number of requirements with regards to the parameters $radix$, $minlen$, and $maxlen$. Specifically, NIST prescribes that

- $radix \in [2..2^{16}]$
- $radix^{minlen} \geq 1,000,000$, and
- $2 \leq minlen \leq maxlen \leq 2\lfloor \log_{radix}(2^{96}) \rfloor$.

In the following, we will discuss the algorithm step-by-step. In algorithm 14, pseudocode for the FF3-1 encryption algorithm is shown.

There are a number of prerequisites for the function:

- A designated cipher function CIPH of an approved 128-bit block cipher
- Key, K, for the block cipher
- Base, $radix$
- Range of supported message lengths $[minlen..maxlen]$

The input to the function is a numeral string X in base $radix$ of length n, where $n \in [minlen..maxlen]$, a tweak bit string T, where $\text{LEN}(T) = 56$, and the output is a numeral string Y, where $\text{LEN}(Y) = n$.

Algorithm 14 Pseudocode for FF3-1 encrypt

1: Let $u = \lceil n/2 \rceil$
2: Let $v = n - u$
3: Let $A = X[1..u]$
4: Let $B = X[u+1..n]$
5: Let $T_L = T[0..27]||0^4$
6: Let $T_R = T[32..55]||T[28..31]||o^4$
7: **for** i from 0 to 7 **do**
8: **if** i is even **then**
9: Let $m = u$
10: Let $W = T_R$
11: **else**
12: Let $m = V$
13: Let $W = T_L$
14: **end if**
15: Let $P = W \oplus [i]^4||[\text{NUM}_{radix}(\text{REV}(B))]^{12}$
16: Let $S = \text{REVB}(\text{CIPH}_{\text{REVB}(K)}\text{REVB}(P))$
17: Let $y = \text{NUM}(S)$
18: Let $c = (\text{NUM}_{radix}(\text{REV}(A)) + y) \bmod radix^m$
19: Let $C = \text{REV}(\text{STR}^m_{radix}(c))$
20: Let $A = B$
21: Let $B = C$
22: **end for**
23: **return** $A||B$

The first part of the algorithm is very similar to FF1 encrypt: The input numeral string X is split into two substrings A and B in steps 1-4. The substrings are of equal length in the case where n is even, otherwise $\text{LEN}(A) = \text{LEN}(B) + 1$ (opposite of FF1 encrypt, where A is one numeral longer than B in the case of n being odd). The next preparatory steps (steps 5-6) consist of partitioning the tweak T into a 32-bit left and right tweak called T_L and T_R, respectively. An example calculation for step 1-4 is shown below (alg. 15) with an input of $X =$ "12345" and $radix = 10$[2]:

[2]As with the example in FF1, this input is chosen for sake of simplicity to illustrate how the algorithm works, but it does not fulfill the requirement of $radix^{minlen} \geq 1,000,000$.

Algorithm 15 FF3-1.encrypt("12345")

1: Let $u = \lceil n/2 \rceil$	$\rightarrow u = \lceil 5/2 \rceil = 3$
2: Let $v = n - u$	$\rightarrow v = 5 - 3 = 2$
3: Let $A = X[1..u]$	$\rightarrow A = X[1..3] =$ "123"
4: Let $B = X[u + 1..n]$	$\rightarrow B = X[4..5] =$ "45"

Step 7-22 is composed of eight Feistel rounds, and constitutes the main bulk of the algorithm. Each round starts in steps 8-14 by determining the parity of the round counter i: if i is even, we use the right tweak T_R for the value of W, else we use T_L. The value m is also set to correspond with the length of B, since the two substrings A and B alternate positions between rounds (steps 20 and 21). In the next step, a block P is produced by XORing a 32-bit encoding of i with W (which is either T_L or T_L) and concatenating the result with a 96-bit encoding of B[3]. Next, S is produced in step 16 by applying the round function CIPH under the reversed key REV(K) on REV(P).

¡In the next few steps (17-19), c is produced by adding the integer value of S to the integer value of A and reducing the result modulo $radix^m$. We then produce C as the numeral string representation of c. In the case of FF3-1, modulo bias is already taken into account by two factors: the tweak length and the upper limit for $maxlength$. These parameter requirements ensure that y is at least 4 bytes longer than A, which corresponds to the modulo bias correction of FF1. Lastly, the two sides are swapped (step 20-21), so that for the next round $A = B$, and $B = C$. After eight rounds, A concatenated with B is returned (step 23).

The reverse functions REV(X) and REVB(X) are used multiple times in FF3-1, since the algorithm uses the opposite order of significance of FF1, i.e. an increasing order of significance. The pseudocode for the reverse functions is shown in algorithm 16 and 17. The reverse function REV(X) takes a numeral string X as input, and copies its contents in reverse order in a for-loop to an output numeral string Y. REVB(X) is an analogous function for byte strings.

FF3-1 is the second version of the algorithm proposed by NIST. The only change from FF3 to FF3-1 is in the length of the tweak (64-bit to 56-bit tweak), and how

[3]It should be noted that since both the left and right tweak are zero-padded with 4 bits, and the maximum value of i is 8, i could simply be added to W instead of using XOR. The likely reason for using XOR is that this line was not changed between FF3 and FF3-1, since FF3 did not use zero-padding on the left and right tweaks.

the left and right tweaks are constructed in step 3. In FF3, the input tweak was simply split in the middle to produce a 32-bit left and right tweak, whereas FF3-1 both rearranges the bits in the right tweak and pads each tweak with 4 zero bits. This was done to prevent an adversary from controlling what is XORed with i in step 15, which could otherwise be used in a chosen-plaintext attack against FF3 over small domains, as shown by Durak and Vaudenay[5].

Since FF3-1, like FF1, is based on Feistel networks, the encryption and decryption algorithms are very similar. The only differences are

1. in step 7, where the for-loop indices are reversed,
2. in step 15-21, where the roles of A and B are flipped, and
3. in step 18, where modular subtraction is used instead of modular addition.

The decryption algorithm is provided in appendix C.

Algorithm 16 Pseudocode for REV(X)

for i from 1 to LEN(X) **do**
    Let Y[i] = X[LEN(X)+1-i]
end for
return Y[1..LEN(X)]

Algorithm 17 Pseudocode for REVB(X)

for i from 0 to BYTELEN(X)-1 and j from 1 to 8 **do**
    Let Y[$8i$+j] = X[$8 \cdot (\text{BYTELEN(X)} - 1 - i) + j$]
end for
return Y[1..8*BYTELEN(X)]

4 Design

The design should reflect the goal of the project. To realize the goal, we have stated several issues that we need to address. The design should cover these issues with help from the knowledge gained from the analysis. Considering the knowledge from the analysis, as well as our proficiencies, we have chosen to write the implementation in Python. User-friendliness and functionality will be the main focus throughout the design phase. To address this, the structure of the library should be designed to

reflect these goals. We will also design milestones to set the scope of the project. These milestones will also help us keep track of our progress throughout the project. Furthermore, the overall security of our design will be considered. Lastly, we will also consider our design of the FF1 and FF3-1 algorithms in Python.

4.1 Structure

The structure of the program has two major factors to consider. One being the usability of the program. The program should be easy to implement and use in different projects. The other being expandability. It should be possible to add new features, formats, etc. without changing the entire program.

4.1.1 Usability

The basic feature is two algorithms capable of encrypting and decrypting an input while preserving its format. The algorithms only encrypt a list of integers, keeping the integers within the range of 0 to $radix$. This would mean that if the user wished to encrypt letters, the user would have to define the $radix$ and map the letters to a list of integers. This would not only be a lot of work, but would also require the user to know how the algorithms work. Instead, we use a `formatter`, which takes a string and some pre-specified format such as `CCN` as input, and translates the string into a list of integers. To allow the user to select formats, the program needs an interface that the user can interact with. This interface can then call the `formatter` and encrypt it with either FF1 or FF3-1. The interface should also have functions for generating a valid key and tweak to use for encryption. This means the user would only have to interact with a single "encrypt" and "decrypt" function where the string, key, tweak, format, and mode are given as input. We also expect that users want to encrypt files such as CSV files with multiple formats and strings. To free the user from having to digest the file into formats and strings and then encrypt/decrypt them, a "encrypt file" function will be added. This function should only take a source file path, destination file path, key, tweak, mode, and list of formats. It will then create a file at the destination file path containing all the encrypted data. The program also generally needs to be robust to wrong inputs, and different errors. The user should always know exactly what went wrong, and eventually get some help to solve the issue. This requires adding different checks throughout the program, and in particular checking the user input is important. For example, if the user input contains symbols not contained in the format, then an error should inform that the symbol is unsupported in the format.

4.1.2 Expandability

There are many ways the program can be expanded, and the structure of the program needs to reflect that. This generally means that the program should be as modular as possible. The amount of hard-coded variables should also be kept to a minimum. This allows for quick identification of where any changes or add-ons should be made. It also helps to prevent the need for a single change/add-on to be implemented in several places. This means that FF1 and FF3-1 need to be in separate files. The cipher used in FF1 and FF3-1 should also be separated and changeable to support different ciphers such as AES and ChaCha. A `mode_selector` can then choose which algorithm to use, and can be expanded with new algorithms easily. The `mode_selector` should not handle any formatting, this is done in advance by the `formatter`. In this way, any format changes/add-ons only need to be implemented in the `formatter`. A `file_encrypter` should also be in a separate file, that reads the file and sends all strings to the `formatter`. The `formatter` and `file_encrypter` is called through an `FPE` interface. This allows the user easy access to everything. In case more features like the `file_encrypter` are added, these can then be called through `FPE`. This structure can be seen in figure 9. The colored segments represent the layers of the program. For example, any changes/add-ons to formats should be contained in the Format layer. Some layers can be linked, such as modes and algorithms. If an algorithm is added, a new mode corresponding to the algorithm needs to be added. This structure allows for features, formats, and algorithms to be added without changing the entire program. It also allows for easy identification of where changes should be added.

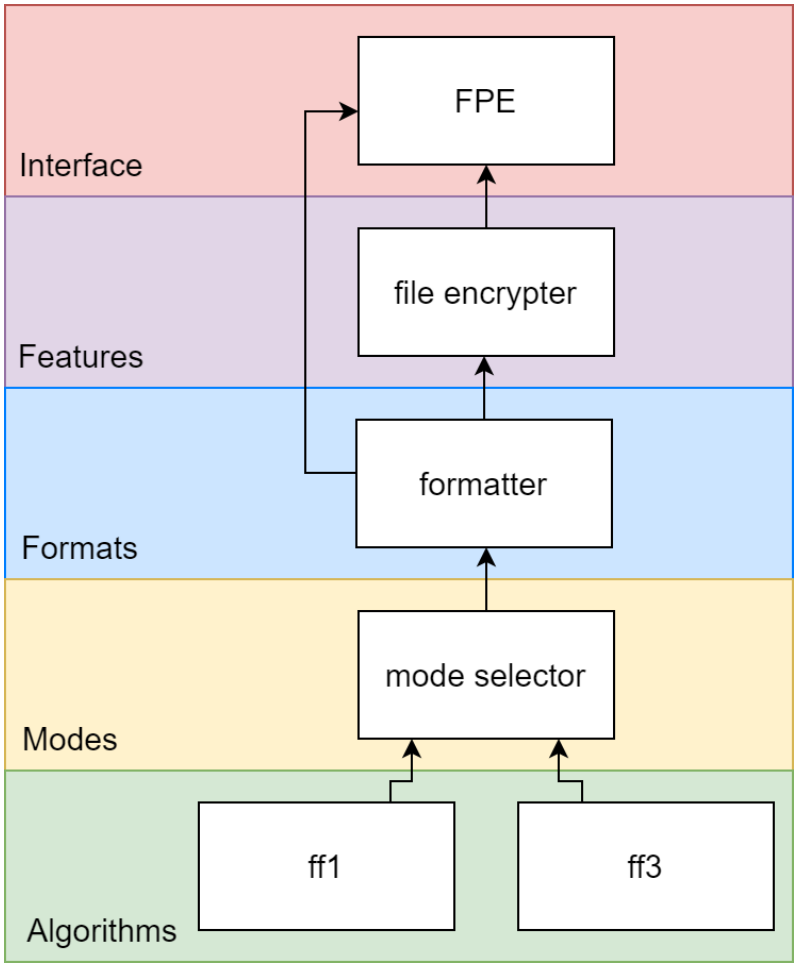


Figure 9: Overview of the structure of the program

4.2 Milestones

4.2.1 First prototype

As a first step, we want the prototype to be the "Algorithm-layer" as seen in figure 9. This means the base functionality of FF1 and FF3-1 should be functional. This first step is an essential component of the library, and also functions as a proof-of-concept. It also constitutes the first basis for comparison, where we can evaluate whether or not it is feasible to finish the project within the deadline. For this milestone, we will not focus on usability and expandability. Since this is the first prototype, our focus will be kept on implementing the ciphers correctly. Usability will play a greater role in the

higher layers, which will be developed in later milestones. The expandability of the design arises from following the structure specified earlier. The security of FF1 and FF3-1 is highly dependant on correct implementation and conformance. Our main focus should be avoiding mistakes in the implementation. The knowledge acquired in the analysis will be key in making sure we avoid mistakes. Testing the prototype will mainly involve conformance testing using test vectors from NIST[13][14].

4.2.2 Alpha build

In the alpha build, all the formats we wish to include in the final product should be implemented. At this point, we will focus on following the planned structure of the program to ensure expandability. Our main focus for this milestone will be the "Formats" and "Modes" layers. At this point, it should also be easy to add new formats and modes. The usability should also have improved significantly at this point. We should now be able to encrypt by specifying a string and a format. Furthermore, we should be able to easily switch between FF1 and FF3-1. This also allows us to test both FF1 and FF3-1 quickly. Furthermore, we should also be able to make some general testing framework to quickly test that all our formats work as intended. The security of the formats is difficult to test and is mostly just dependant on the security of FF1/FF3-1. However, as stated earlier, some formats may require different methods beyond the NIST specification and could be insecure if implemented incorrectly. This is difficult to test and verify, but can be evaluated theoretically.

4.2.3 Stable version

The stable version is planned to be the second-to-last version, where we have every feature finished and optimized. We want this version to be distributable, so that the final build only contains small changes from this version. At this point, we should have created the contents of the "Features" and "Interface" layers. The "Features" layer should be expandable with new features. Our "Interface" layer is meant to be as clean as possible - it is the highest level of our structure, and it will be the layer that users interact with. This layer should tie all of the features together, so we have to consider how we handle potential new features in the future. Currently, we plan to add a feature to encrypt files. At this point, we also want to test the runtime of our code, to optimize it. The optimization will be done mostly by translating sections of Python code into Cython code, but other ways to optimize it will also be considered. To test these additions, we plan on adding a feature to generate test data.

4.2.4 End-of-build

This is our final product. Any small changes to the stable version should be finished and tested. Our code should have a structure that is similar to the designed structure, and the library should satisfy the end goal. The program then needs to be packaged and uploaded to the Python Package Index *pypi.org*. The program has been structured to be a package, which means the program should already be ready for packaging. To upload it, we need to add a license, a readme, and setup. The license and readme are added to allow everyone to use our package. We plan on distributing it as an open-source and free-to-use package. The MIT license reflects those goals and generally seems to be a good fit for our package. The readme should explain how to install and use our program. It is important that the package can be installed and used only using the help found in readme. The setup is used to build the package. It tells Python which files to include, how to compile them, and how the program is structured. It also contains a lot of info about the package such as name, version, author, dependencies, etc. We expect this setup file to be tricky, since we have both Python and Cython code in our project.

4.3 Formats

Our library supports encryption and decryption of different formats. A format is defined by NIST as a mapping between numerals and an alphabet. This definition is sufficient for some formats, e.g. strings and numbers, but for other formats, requirements can go beyond simple mappings, notably emails and credit card numbers. In the following, we will describe our design for tackling the more complicated formats.

4.3.1 Credit Card Numbers

As explained in the analysis, credit card numbers are 16-digit numbers. We chose not to validate the credibility of the prefix that identifies the issuer of the card. The Issuer Identification Number (IIN) could be relevant in some cases, but we consider this to be irrelevant in most use cases for our library. We use the checksum, also described in section 3.5.1, to check if a given credit card is valid. Because of our choice not to check the prefix, we can simply regard the different numbers as a fixed-length string of integers using the alphabet 0-9. If we were to encrypt the entire CCN, the checksum would end up being a random number. This would mean that the encrypted CCNs are invalid. Instead, we chose to only encrypt the first 15 digits, and then use the Luhn validation algorithm to generate the value of the checksum. This also works well with decryption. When we decrypt, we again use the first 15

digits and disregard the generated checksum in the ciphertext. We then get the first 15 digits of the plaintext, and because we started by checking if the credit card was valid, we know that we can just generate the right number again using the checksum.

4.3.2 CPR Numbers

As described in the analysis, there are number of rules and conventions regarding what constitutes a valid CPR number. Some are not important for our implementation, such as "the 7th number determines what century the given person is born in". We do not want to restrict CPR numbers that are out of date, or that have not been issued yet, so it would not make sense to check if, for example, the 5th and 6th number matches the 7th. We also want to design our implementation to work with the "modulo 11 check". Even though it is not used in Denmark anymore, it is the standard in other countries, so it also works as a proof-of-concept to include this validation. The checksum can be solved in the same way as with CCNs by just computing the last digit. That still leaves one more issue to solve, though. A CPR is number of the form DDMMYYNNNC where D = day, M = month, Y = year, N = number, C = checksum. For encryption, we can see Y as a number since it can be any value between 0-9. However, this still leaves DDMM that cannot be encrypted as normal digits. To solve this, we have designed a unique way to encrypt this part. If we produce a list of all 365 dates, we can identify the plaintext date in the list. Then we can compute a new date by taking (DDMM+YYNNN) mod 365. However, this method has no randomness and could be vulnerable to attacks where all possible values are pre-computed. To solve this, we can also add the key. Using the key here will be safe since we use modulo. The encrypted date would then be (DDMM+YYNNN+Key) mod 365. We can then concatenate the encrypted numbers and compute the checksum to complete the CPR number.

4.3.3 Email Addresses

The email address format poses even more challenges than the other formats. The email address is composed of some different parts, also described in section 3.5.3. The different parts of the email have different criteria for being valid, mostly criteria about which characters are allowed (but not exclusively). Therefore, we have to split the email address into three parts: the local-part, the domain name, and the top-level domain. The local-part can contain letters and a limited amount of special characters. The domain name includes the same characters as the local-part, but is expanded with a few extra special characters, but two (or more) consecutive special

characters are not allowed[4]. The top-level domain has to be one of the certified top-level domains, such as '.com' and '.dk'. The characters accepted in the local-part depend on the provider, but common for all is that all English alphabet letters are accepted. Therefore, we will only include letters in the local-part format. For the domain name, we will also include "-" and ".", since these symbols are generally accepted. We do not want to include too many symbols since we want the domain to be generally accepted. For the top-level domain, we can use the same method as the dates in CPR. We can make a list of all top-level domains and shift it by the numerals from the local-part and domain name. Naturally, the key needs to be added for extra security again.

4.4 Security

Security should be considered throughout the program. Generally, we want to follow Kerckhoff's principle stating that a cryptosystem should be secure, even though everything about the system is public knowledge. The exception to this is of course the key. The most essential part of the security is the encryption algorithms. FF1 and FF3-1 have been analyzed by NIST and experts within cryptography worldwide. This implies that FF1 and FF3-1 can be trusted to be secure. The biggest issue regarding FF1 and FF3-1 is implementing them in conformity with the requirements by NIST. It is also important to use secure, thoroughly tested ciphers for the PRF. Both ChaCha and AES are well-known and tested ciphers used around the world. Again, these are secure when implemented in conformity with the requirements. Even though the rest of the program does not directly have to do with encryption, security should still be considered. An example, where security could go wrong, would be if we were to hard-code the key to some constant variable. This would allow anyone to read the key from the source code. The key should naturally be chosen/generated by the user, and the user is responsible for storing the key. Different formats may require multiple encryptions or different methods of encryption. It is essential to consider security here, since the original input or the key may be guessed if the encryption is done in an insecure way.

4.5 FF1 & FF3-1

The design of FF1 and FF3-1 is already defined by NIST. Though some variations can still occur when translating the pseudocode into actual code. NIST has designed

[4]Technically, only consecutive periods are disallowed in the specification, but very few mail providers allow for consecutive special characters in general.

them in a way that only works with basic formats (a numeral list and a radix). This means that all inputs and outputs will be numeral strings. Naturally, this is insufficient for encryption of emails, CCNs, CPRs, etc. Any input that is not already a numeral list needs to be processed into numerals somehow. NIST does not specify any method for this, so it is up to us to design this. We could modify the algorithms to support other inputs and then process them inside FF1/FF3-1. Another way is to understand FF1/FF3-1 as a framework that only encrypts numeral strings. The formatting then happens outside of FF1/FF3-1 and then "feeds" the numerals to FF1/FF3-1. This method has several advantages, and we will use this design for our project. First of all, this means that FF1/FF3-1 should be implemented as close to the pseudocode as possible. This allows us to test the conformance of our implementation with the NIST requirements. It also means that in case of errors or insecurities we can disregard FF1/FF3-1 as being a possible issue. This design also reflects the structure we proposed previously.

The goal of making FPE accessible through a library should be able to be realized with this design. Our focus will be a well-tested project with documentation of security and usability. This documentation will be in the form of a thorough examination of the implementation. As well as a post-distribution evaluation with tests as proof of security, usability, expandability, and optimal runtimes. We will also evaluate how well the milestones tied the project together and provide a guideline for reaching our goal.

5 Implementation

This section will describe how we implemented the library in Python in accordance with the design chapter. We will provide a detailed explanation of how we translated the pseudocode into actual Python code. Our implementation is kept as close to the pseudocode as possible. This also means that variable and function names are the same as in the pseudocode. We will also give detailed explanations of all the formats and functionality of the library. All of the optimizations that we have done in Cython and Python will also be explained[5].

[5]To install our library simply run `pip install FPE` in the terminal. Note that our library requires at least Python 3.7 installed. For a list of commands and examples of usage please consult the readme in appendix E.

5.1 FF1

The FF1 pseudocode can be seen in appendix A. In this section, the Python code corresponding to the pseudocode will be examined. The code below shows the variable initialization, where `msg` is the input message to encrypt, `T` is the tweak. The rest of the variables are named as in the pseudocode. $log_2(radix)$ is the bit length for a single numeral. This is multiplied by `v`, which is the length of `B` and represents the total bit length of `msg`. Lastly, it is divided by 8 to give the size in bytes.`d` is defined to be a byte length at least 4 bytes (max 7) larger than `b`. In Python, this is solved with the native `math` library, which allows us to ceil and log. `P` is defined as a 16-byte variable that is used as the initial value for the block cipher. Python has a built-in function `.to_bytes()` that converts an integer to bytes and pads with zeros to a specific size. The only issue with this function is that it can't handle inputs larger than the specified byte size. This means that we have to ensure the number we want to be converted never exceeds the specified byte size. in `P` the tweak length must not exceed a 4-byte number, which means our max tweak is limited to ≈ 4.3 billion digits. A tweak this large should never be necessary, so it shouldn't be a problem limiting the tweak length.

```
t = len(T)
n = len(msg)

u = n // 2
v = n - u

A = msg[:u]
B = msg[u:]

b = math.ceil(math.ceil(v * math.log2(radix)) / 8)
d = 4 * math.ceil(b / 4) + 4

P = b'\x01' + b'\x02' + b'\x01' + \
    radix.to_bytes(3, 'big') + b'\n' + \
    (u % 256).to_bytes(1, 'big') + \
    n.to_bytes(4, 'big') + t.to_bytes(4, 'big')
```

The next part is the loop for the 10 Feistel rounds, where we start by creating the bytes to encrypt. To do this we convert `B` into an integer through the `num_radix` function seen below. The function works as specified in the analysis, where each numeral is added and then multiplied by radix.

```
def num_radix(radix, numbers):
    x = 0
    for numeral in numbers:
        x = x * radix + int(numeral)
    return x
```

After `B` is converted to an integer, the integer is converted to bytes with the `to_bytes` function. `B` is then concatenated with the tweak and some padding that ensures the byte length is a multiple of 16. This is done to ensure `Q` can be divided into 16-byte blocks without issues. The PRF works as a chain block cipher, where the previous output is used as input for the next block. As seen in the PRF code below, the first block we encrypt uses 16-bytes of 0's as "output" from the previous block. Since this is less random, we use `P` as the first block, so `Q` can be securely encrypted. The encryption itself is the `cipher.encrypt` function from the PyCryptodome library. The cipher is an AES cipher, with a 128-bit key.

```
key = bytes.fromhex('2B7E151628AED2A6ABF7158809CF4F3C')
cipher = AES.new(key, AES.MODE_ECB)

def PRF(X,cipher):
    m = int(len(X)/16)
    Yj = b'\x00\x00\x00\x00\x00\x00\
                x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
    for j in range(m):
        Xj = X[j * 16:(j * 16) + 16]
        Yj = cipher.encrypt(bytes(A ^ B for A, B in zip(Yj, Xj)))
    return Yj
```

After `Q` has been encrypted we make sure `R` is larger than `A` by using the variable `d`. `d` is at least 4 bytes larger than `A`, and if `d` is larger than 16 bytes, `R` is not large enough to eliminate modulo bias. This is fixed by concatenating d/16 blocks to `R`, ensuring that `R` is at least 4 bytes larger than `A`. In the end, only the first `d` bytes of `R` is used, ensuring that `S` is not unnecessarily large. As seen below the random blocks are created by encrypting `j`, which is just a number between 1 and `d`/16. S is then converted to an integer, and combined with `A`.

```
for j in range(1, int(math.ceil(d / 16))):
    S = S + cipher.encrypt(
                    bytes(A ^ B for A, B in
                    zip(R, (j).to_bytes(16, 'big'))))
```

```
S = S[:d]
y = int.from_bytes(S, 'big')
if (i % 2 == 0):
    m = u
else:
    m = v
c = (num_radix(radix, A) + y) % radix ** m
C = str_radix(radix, m, c)
A = B
B = C
```

`c` is the integer created by combining `A` and `y`, which then needs to be converted to a numeral string. This is done in the `str_radix` function, which can be seen below. `str_radix` works as described in the analysis by filling a list of numerals with number mod radix. Then floor divide number with radix and repeat until the numeral list is complete.

```
def str_radix(radix, length, number):
    if length < 1:
        raise ValueError(f"{length} is not a valid string length")

    if not (0 <= number <= radix ** length):
        raise ValueError(
            f"{number} is not in range [0;{radix}^{length}]"
            )

    numerals = [''] * length

    for i in range(length):
        numerals[length - 1 - i] = str(number % radix)
        number = number // radix
    return numerals
```

Algorithm 18 FF1 Encryption

Let $u = \lfloor n/2 \rfloor; v = n - u$
Let $A = X[2..u]; B = X[u+1..n]$
Let $b = 4\lceil \lceil v \times \text{LOG}(radix) \rceil / 8 \rceil$
Let $d = 4\lceil b/4 \rceil + 4$
Let $P = [1]^1 || [2]^1 || [1]^1 || [radix]^3 || [10]^1 || [u \bmod 256]^1 || [n]^4 || [t]^4$
for i from 0 to 9: **do**
  Let $Q = T || [0]^{(-t-b-1) \bmod 16} || [i]^1 || [\text{NUM}_{radix}(B)]^b$
  Let $R = \text{PRF}(P||Q)$
  Let S be the first d bytes of the following string of $\lceil d/16 \rceil$ blocks:
  $R||\text{CIPH}_K(R \oplus [1]^{16})||\text{CIPH}_K(R \oplus [2]^{16})...\text{CIPH}_K(R \oplus [\lceil d/16 \rceil - 1]^{16}$
  Let $y =$ NUM(S)
  if $i \bmod 2 = 0$ **then**
    Let $m = u$
  else
    Let $m = v$
  end if
  Let $c = (\text{NUM}_{radix}(A) + y) \bmod radix^m$
  Let $C = \text{STR}^m_{radix}(c)$
  Let $A = B$
  Let $B = C$
  Return $A||B$
end for

5.2 FF3

Our implementation of FF3 follows the pseudocode provided by NIST quite closely. We will in the following describe how we implemented the algorithm in Python.

The first step of our implementation is that we split both the input `numeral_string` and the `tweak` into two parts to set up for the following Feistel rounds. We use BitArrays from the Python library bitstring to simplify working with arrays of bits (specifically for the left and right tweak, and later `P`).

```
n = len(numeral_string)
u = int(ceil(n / 2))
v = n - u

# Split numeral string into two parts
```

```
A, B = numeral_string[:u], numeral_string[u:]

# Construct a left and right tweak of 32 bits
tweak_left = BitArray(tweak[:28]) + BitArray('0b0000')
tweak_right = BitArray(tweak[32:56]) + BitArray(tweak[28:32]) +
              BitArray('0b0000')
```

Next, the eight Feistel rounds are carried out in a for-loop. In the beginning of each round, the parity of the round counter `i` is checked in an if/else, and the variables `m` and `W` are set accordingly. `P` is calculated in to parts: `W` is XORed with a 4-byte representation of `i` on the left and concatenated with an array of bits representing `B` in base `radix` on the right. `B` is reversed, because FF3-1 represents numbers as strings of numerals in increasing order of significance (opposite of FF1). `P` is then encrypted using the AES cipher provided by the PyCryptodome library, and the output `S` is stored in `y` as an integer. Finally, `c` is calculated as the sum of `A` (interpreted as an integer) and `y` reduced modulo `radix ** m`.

```
for i in range(8):
    if (i % 2) == 0:
        m = u
        W = tweak_right
    else:
        m = v
        W = tweak_left

    P = (W ^ BitArray(i.to_bytes(4, 'big'))) +
        BitArray(num_radix(radix, reverse(B)).to_bytes(12, 'big'))
    S = reverse(self.cipher.encrypt(bytes(reverse(P))))
    y = int.from_bytes(S, 'big')
    c = (num_radix(radix, reverse(A)) + y) % (radix ** m)

    C = reverse(str_radix(radix, m, c))
    A = B
    B = C

return A + B
```

The final three lines of code in the for-loop is to setup for the subsequent round: `A` and `B` swap sides, and `B` takes the value of `C`, which is a numeral string representation of `c`. After 8 rounds, we exit the for-loop and return `A + B`, which is the encrypted

numeral string.

5.3 Cython optimizations

We used Cython to improve the performance of the program. Cython compiles Python code into C, which on its own does not make Python faster. However, Cython also allows you to write C code mixed in with the Python code since it is all translated into C in the end anyway. Below is a showcase of the difference between Cython and Python, where the code has the same functionality, but Cython is written as in C. It is quite clear to see why Python is gaining popularity because of the simplicity of the code, but it is exactly that which makes Python slow.

Translating into Cython is primarily just re-writing variables into C variables. The C variables do not support the same operations as in Python, so just defining the variable in C is not always enough. This can also be seen in the example below, where `A` and `B` has to use the C function `memcpy` to copy `msg`.

```
---- Cython ----
cdef int u = n / 2
cdef int v = n - u

cdef int *A = <int *> malloc(u*sizeof(int))
cdef int *B = <int *> malloc(v*sizeof(int))

memcpy(A, msg, u*sizeof(int))
memcpy(B, msg+u, v*sizeof(int))
```

```
---- Python ----
u = n // 2
v = n - u
A = msg[:u]
B = msg[u:]
```

Another difference in C and Python is the size of variables. In Python, lists can be appended, copied, iterated, etc. without knowing the size of the array. In C however, it is necessary to manually allocate the required memory for the array. In many cases, this is not a problem like when splitting the input into `A` and `B`, where the sizes `u` and `v` are already known. The problem is in the Feistel structure, where the size of `A` and `B` changes. This is partly fixed by the `m` variable, however, in the step where $A = B$ it can be different. To solve this, another variable `l` has been introduced in both FF1 and FF3. As seen below, it is the opposite of `m`, which means the length of both `A` and `B` at any point of the Feistel network is known. This allows us to do `memcpy` of `A` and `B` with the correct size.

```
        if (i % 2 == 0):
            m = u
```

```
            l = v
        else:
            m = v
            l = u
```

5.3.1 FF1

We started by testing the runtime of FF1 without any optimizations. Running 100,000 encryptions and decryptions with the input "123456" took 35.4 seconds. Just by re-writing the variables as C, the code runs in 23.4 seconds. This is about 50% faster than base Python, however, we can still improve this time. To find the parts of the code that had long runtimes, we inserted a 100,000 round loop in different places in the code and found that the PRF function seen below was taking 18.5 seconds. It makes sense that it should be the most time-demanding task. However, it is slower than it should be.

```
cdef bytes PRF(bytes X,cipher):
    cdef int m = len(X)/16
    cdef bytes Yj = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
                    x00\x00\x00\x00\x00'
    cdef int j
    cdef bytes Xj
    for j in range(m):
        Xj = X[j * 16:(j * 16) + 16]
        Yj = cipher.encrypt(bytes(A ^ B for A, B in zip(Xj,Yj)))
    return Yj
```

Some more testing showed that running the PRF 100,000 times without XORing the input took around 11 seconds. We figured it had to be the `zip` function that we used to XOR the bytes of `Xj` and `Yj`, so we made a new function `xor16bytearray` to do the XOR. The function below takes two 16 byte arrays and converts them to unsigned char arrays since `bytes` is a Python type. Then the bytes are XORed and returned as a byte array. Replacing the `zip` function with this means that the extra time spent on XORing was removed. This optimization means that the total runtime for 100,000 encryptions and decryptions is down to 17.7 seconds, which is twice as fast as the Python version. Naturally, these runtimes depend on the CPU running the program, in our case, all of the testings were done on an Intel I5 8250U.

```
cdef bytes xor16ByteArray(bytes A, bytes B):
```

```
    cdef unsigned char* Ac = <unsigned char *> malloc(16*sizeof(char))
    cdef unsigned char* Bc = <unsigned char *> malloc(16*sizeof(char))
    cdef unsigned char* xor = <unsigned char *> malloc(16*sizeof(char))

    if not Ac or not Bc or not xor:
        raise MemoryError()

    cdef int i
    for i in range(16):
        Ac[i] = A[i]
        Bc[i] = B[i]
        xor[i] = (Ac[i] ^ Bc[i])

    cdef bytes result = xor[:16]

    return result
```

5.3.2 FF3

Again, we test FF3 without optimizations first. This time 100,000 encryptions and decryptions took 192.4 seconds. Since this is way slower than FF1 even though FF3 is supposed to be faster, we realized something was wrong in our implementation. The problem was in the `bitstring` package we used to split the tweak. When casting the `BitArray` to bytes to encrypt it, each bit was cast to a byte. This means that the resulting byte array was 32 bytes instead of 4. This was the first thing we solved in Cython, as seen below. The left tweak is the first 28 bits of the tweak concatenated with 4 0-bits. In Cython, we did this by taking the first 4 bytes of the tweak and then doing bitwise AND on the last byte and the number 240. 240 is the decimal representation of "1111 0000", which means the first 4 bits will be 0, and the following 4 will be unchanged. The right tweak is bit 33-56 of the tweak concatenated with bit 28-32 and 4 0-bits. In Cython, we did this by taking the first 3 bytes of the tweak. Then we set the last byte of the left tweak to be the 4th byte of the tweak. This corresponds to the last byte of tweak left being bit 24-32 of the tweak. Then we shift the bits left 4 times, so the first 4 bits become 0, and the following 4 become bit 28-32 of the tweak.

```
memcpy(tweak_left, tweak, 4*sizeof(char))
memcpy(tweak_right, tweak+4, 3*sizeof(char))
```

```
tweak_right[3] = tweak_left[3] << 4
tweak_left[3] = tweak_left[3] & 240
```

The tweak also needs to be XORed with `i` so the `xor16bytearray` function from FF1 is also used in FF3, but changed so that it can support different lengths. As seen below the function now takes the length as input as well as the two arrays. The function is also changed to have unsigned char arrays as input instead of bytes, since the tweak is a char array. In theory, it is enough to XOR only the last byte of each array, since `i` will never be more than a byte and the rest is just zeroes. However, since we wish to convert it to bytes, we need to loop over the entire array anyway. The `xorByteArray` returns a byte array as in FF1.

```
P = xorByteArray(W, round_array, 4) + num_radix(radix,
    reverseArray(B, l), l).to_bytes(12, 'big')
```

FF3 requires the arrays to be reversed multiple times. In Python, all the reversing was handled by the simple reverse function, but that functions is not able to reverse C types. The `reverseArray` function seen below reverses an array of integers, by setting the first spot in the array to the last, and the last to the first. In this way, it only has to run for half the length of the array.

```
cdef int* reverseArray(int* arr, length):
    cdef int* numerals = <int *> malloc(length*sizeof(int))

    memcpy(numerals, arr, length*sizeof(int))

    cdef int i
    cdef int temp
    for i in range(length/2):
        temp = numerals[i];
        numerals[i] = numerals[length - 1 - i];
        numerals[length - 1 - i] = temp;

    return numerals
```

With these changes, the total time for 100,000 encryptions and decryptions went down to 10.7 seconds, which is 18 times faster. However, that is mostly due to the bug with the tweak being fixed. It does still make it about 65% faster than FF1, which makes FF3 great for inputs with length $< 2 \cdot \log_{\text{radix}}(2^{96})$.

5.4 CSV

We have one file called `fpe_csv` that created to handle CSV files. Within this file, we have several functions. We have `encrypt_csv` and `decrypt_csv` that can be called from the main function, and that encrypts or decrypts data from a CSV file. Furthermore, we have `generate_test_data` that, as the name suggests, generates test data according to its arguments. This is also meant to be called from our main file. In addition to these function, we also have some helper functions, called `encrypt`, `decrypt` and `generate_data`. Their use will be explained in the following sections.

5.4.1 encrypt_csv & decrypt_csv

The purpose of these functions is to encrypt and decrypt a csv-files. The two functions are identical, except for the line where we encrypt/decrypt the data: `encrypt_csv` calls our helper-function `encrypt`, whereas `decrypt_csv` calls the helper-function `decrypt`.

```
def encrypt_csv(csvFilePath,encryptedFilePath,formats,fpe):
```

When calling the two functions, the first step (apart from declaring our basic variables) is to declare and construct the list of data from the CSV file. This is shown below.

```
with open(csvFilePath) as csvFile:
csvReader = csv.reader(csvFile, delimiter = ';')
rowCount = 0
for row in csvReader:
    if (rowCount != 0):
        columnCount = 0
        for column in row:
            data[columnCount].append(column)
            columnCount += 1

    else:
        for column in row:
            data.append([column])

        rowCount += 1
```

To be able to handle the file (reading and writing), we use the library `csv`. It is not

possible to read and write to the file at the same time, so we start by reading from the CSV file. When reading the file, we create a for-loop for the rows, and a nested for-loop for the columns, to iterate through the data from the CSV file. All of this data is added to our variable `data` to save it, so we can operate on it and put it back into a new file.

In Python for-loops, it is possible to iterate over iterable objects (e.g. lists) instead of numbers. This is an easy way to work with the objects of a list, or in our case, a database of rows and columns. We have a for-loop looping through all the rows, and another for-loop running through the columns in the individual rows. This means that we have to create our counters where needed. First of all, we have to know if we were on the first row of the dataset, because the CSV file's first line contains the column names, and they should not be encrypted. Secondly, the current column needs to be kept track of, since we have to know which format in `dataFormats` the data should be encrypted with. Once `data` is filled with data from the CSV file, then we want to operate on the dataset. This is shown in the code snippet below.

```
with concurrent.futures.ProcessPoolExecutor() as executor:
    results = list(executor.map(encrypt,data,formats,keys,tweaks,modes))

    data = results
```

In essence, this snippet of code splits the dataset into columns, and feeds them to the function `encrypt` along with an instance of the key, tweak, and mode(FF1 or FF3-1). This is where the two functions `encrypt_csv` and `decrypt_csv` differ - one has the function `encrypt` as the input, and the other has `decrypt`.

The `write` part of the program is seen below. Just like the `read` part, two nested for-loops to go through the rows and columns of the file. A list `data2` is filled with a row of data, written to a new file, and then refilled with the next row, and so on.

```
with open(encryptedDataPath, 'w',  newline='') as encryptedCSVFile:
    csvWriter = csv.writer(encryptedCSVFile, delimiter = ';')
    for i in range(len(data[0])):
        data2 = []
        for j in range(len(data)):
            data2.append(data[j][i])
        csvWriter.writerow(data2)
```

5.4.2 generate_test_data

This function is responsible for generating our test data. The argument taken as inputs are shown below

```
def generate_test_data(csvFilePath,rows,formats,names, mode):
```

The `csvFilePath` is the path to the file, where the generated data is written, `rows` is the number of rows of data to be generated, `formats` is similar to the list we used in our encryption/decryption functions, and specifies which format to use for encryption or decryption of the different columns. `names` is a list of the column names, which is needed because the first row of CSV files contains the column names, and they should therefore be stated ahead of creating the test data. Lastly, `mode` is whether to use the mode ff1 or the mode ff3.

The way we create this test data is to have a sample string for each of the formats, and then we encrypt this string repeatedly with different keys.

```
data = [[x] for x in names]

for i in range(len(names)):
    for _ in range(rows):
        data[i].append(mapping_formats[formats[i]])
```

In the code above, we show how we create the list `data` that has similar properties to the variable with the same name in the two other functions. First, we create a new nested list inside the list, containing the column-names from `names`. Then, we put the sample string into the different rows. We do this by using a list, `mapping_formats`, containing a map from the formats to their corresponding sample text. Then we use `formats` to get the right sample strings from the map. The sample string is inserted `rows` amount of times. Once the dataset is created with a large amount of dummy data, then we want to encrypt it all, just like `encrypt_csv`. This is shown below.

```
with concurrent.futures.ProcessPoolExecutor() as executor:
    results = list(executor.map(generate_data,data,formats,modes))

    data = results
```

Here, instead of using the helper-function `encrypt`, we use a helper-function very similar to `encrypt`, called `generate_data`. The main difference is that this function does not need to take an `fpe` as input, but just a `mode`, since it does not need a key and a tweak as input. Instead, it uses a randomly generated key for each row and a

randomly generated tweak for the whole file. In the end, all we need to do is to put the data into a CSV file. This is done in the same way as in the two other functions.

5.5 Formatter

As of now, we have created 6 formats. We chose to represent them as integers, and they are shown below.

```
DIGITS = 0
CREDITCARD = 1
LETTERS = 2
STRING = 3
EMAIL = 4
CPR = 5
```

The logic tied to the formats is spread across two files `format_translator.py` and `formatter.py`. The `formatter` is responisble for deciding what happens to the input depending on the format given. The `formatter` then calls functions from the `format_translator` to map from string to numerals, get the radix, and so on. The functions in the `format_translator` will be described in detail in the next section. The `encrypt` function and the `decrypt` function are essentially the same, so we will only describe encryption in detail.

5.5.1 Email

The `formatter` is composed of several if-statements that check the format. The first format represented is `EMAIL`. First, we call `text_to_numeral_list` from the `format_translator` file. As the name states, this maps the input string to a list of numerals. Next, we obtain the radix from `get_radix_by_format`, which returns an integer corresponding to a given format. In this case, we need three different radices because emails need a different radix for each part. When we have all the radices and numeral lists, we can verify the input complies with the requirements for FF1 and FF3-1. This is just done with several if-statements and is done part of the email. Before encrypting, we need to initialize the list `cipherNumerals` to contain all the ciphertexts. At this point, we start encrypting/decrypting. We need to encrypt/decrypt twice, as the parts before and after the '@'-sign have different radices. The code below shows the encryption of emails. We have omitted the requirements checks to improve readability.

```
plainNumerals =  format_translator.text_to_numeral_list(text, dataFormat)
radixes = format_translator.get_radix_by_format(dataFormat)

cipherNumerals = []

cipherNumerals.append(
    mode_selector.encrypt(plainNumerals[0],key,tweak,radixes[0],mode))
cipherNumerals.append(
    mode_selector.encrypt(plainNumerals[1],key,tweak,radixes[1],mode))

cipherNumerals.append(
    (plainNumerals[2] +
    int(''.join([str(x) for x in plainNumerals[0]]) +
    ''.join([str(x) for x in plainNumerals[1]]) +
    str(int.from_bytes(key,'big'))))%radixes[2])

return numeral_list_to_text(cipherNumerals, FORMAT_EMAIL)
```

Above is also shown how we shift the top-level domain. This is done in the way we have described in the design. The `plainNumerals` is a list of integers, so to add them together we have to map them to strings, so we can use the join function to get them out of the list, and then map them back into integers. We use modulo `radixes[2]` to make sure the top-level domain stays within its space. As the last thing we do in this function, we call `numeral_list_to_text` and return it. `numeral_list_to_text` has the same concept as `text_to_numeral_list`, and will be described in detail later.

5.5.2 CPR

The second format is CPR, and much of the format is similar to `email`. We start by getting the `plainNumerals`, the radix. We then use if-statements to verify the input complies with the requirements. The `cipherNumerals` variable is initialized and then we encrypt the input. This time the format was only split into two parts, which means only one encryption is necessary. The date part is shifted in the same way as the top-level domain. Lastly, like the last format, we call `numeral_list_to_text` and return it. Again the code is shown below with the requirements checks omitted.

```
plainNumerals = format_translator.text_to_numeral_list(text, dataFormat)
radixes = format_translator.get_radix_by_format(dataFormat)
```

```
cipherNumerals = []

cipherNumerals.append(
    mode_selector.encrypt(plainNumerals[1][:5],key,tweak,radixes[1],mode))

cipherNumerals.append(
    (plainNumerals[0] +
    int(''.join([str(x) for x in plainNumerals[1][:5]]) +
    str(int.from_bytes(key,'big'))))%radixes[0])

return format_translator.numeral_list_to_text(cipherNumerals, dataFormat)
```

5.5.3 Simple formats

The rest of the formats are all simple enough to be handled in the same way. Just like the two other formats, we start by getting the `plainNumerals` and the radix, then followed by if-statements to check the requirements. Since there is only one part, no shifting or multiple encryptions are needed. For these formats, it is enough to simply encrypt the `plainNumerals` and map the `cipherNumerals` back to a string and return it. Again, the code is shown with the requirements check omitted.

```
plainNumerals = format_translator.text_to_numeral_list(text, dataFormat)
radix = format_translator.get_radix_by_format(dataFormat)

cipherNumerals = mode_selector.encrypt(plainNumerals,key,tweak,radix,mode)

return format_translator.numeral_list_to_text(cipherNumerals, dataFormat)
```

5.6 Format translator

`Format_translator` is the file where we handle most of our logic regarding formats. This includes mapping from plaintext to ciphertext, and creating the maps. Similar to the concept of ASCII, we have a map from characters to numerals. The domain 'characters from *a*-*å*' is shown below:

```
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5, ... ,
'x': 23, 'y': 24, 'z': 25, 'æ': 26, 'ø': 27, 'å': 28}
```

We create maps using the method `get_mapping_from_domain`. It takes a list as an input and outputs a tuple of lists. The first one is a map from the domain to its

respective number, like the map shown above. The second one from the respective number back to the domain, which would be the reverse of the map shown above (with the character and the integers places switched). The lists we give as input to make the maps are in most cases a sub-domain from our list `DOMAIN`(shown below), which covers all possible chars we support throughout our different formats. In the few cases where we do not use `DOMAIN`, we use a list of our other domains, `top-lvl-domain` or `dates`, depending on which map it creates. These two lists, `top-lvl-domain` and `dates` are structured the same way as `DOMAIN`, but instead of a list of chars, it is a list of dates or top-level domains.

```
DOMAIN = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
          'p','q','r','s','t','u','v','w','x','y','z','æ','ø','å','A',
          'B','C','D','E','F','G','H','I','J','K','L','M','N','O','P',
          'Q','R','S','T','U','V','W','X','Y','Z','Æ','Ø','Å','0','1',
          '2','3','4','5','6','7','8','9','.','-','!','#','$','£','%',
          '&',''','*','+','/','=','?','^','_','´','{','}','|','','',',',
          '(',')',':','<','>','`','~','é']
LOWER_LETTER_END = 29
UPPER_LETTER_END = 58
INTEGER_END = 68
EMAIL_SIGNS_END = 85
```

Our list `DOMAIN` is shown above, and so are the variables we use to split the big list into smaller lists, that are used for our sub-domains. The integers are, as their names state, an indicator of where in the list the different types of characters start and stop. For example, if we needed a list of all the letters, we would use the domain from the start up until `UPPER_LETTER_END`. If we needed only numbers, we would start from `UPPER_LETTER_END` to `INTEGER_END`. The code that implements this behavior, as well as the lists we made from the domains `dates` and `top_lvl_domain`, is shown below.

```
mapping_letters = get_mapping_from_domain(DOMAIN[:UPPER_LETTER_END])
mapping_upper_letters = get_mapping_from_domain(DOMAIN[
                        LOWER_LETTER_END:UPPER_LETTER_END])
mapping_lower_letters = get_mapping_from_domain(
                        DOMAIN[:LOWER_LETTER_END])
mapping_email_tail = get_mapping_from_domain(DOMAIN[:LOWER_LETTER_END]+
                     DOMAIN[UPPER_LETTER_END:INTEGER_END+2])
mapping_letters_integer = get_mapping_from_domain(DOMAIN[:INTEGER_END])
mapping_all = get_mapping_from_domain(DOMAIN)
mapping_top_lvl_domains = get_mapping_from_domain(top_lvl_domains)
```

```
mapping_dates = get_mapping_from_domain(dates)
```

5.6.1 text_to_numeral_list

This is the first helper-function called from `formatter`'s `encrypt` and `decrypt`. This function takes two inputs, a text, and a format, and outputs one or more numeral strings depending on which format is given. Just like in `formatter`'s `encrypt` and `decrypt`, we start with an if-statement for each of the formats. Within some of the if-statements, we have included a check that makes sure that the content of the string is supported by the format, e.g. by checking if `DIGITS` only contains numbers. For formats that deal with chars instead of numbers, we have included the check into the function that maps the chars to numerals.

The first, and most simple, format is `DIGITS`. All it does (besides value-checking) is take the numbers that were given as a string and separate them into a list of integers. We have also made a value-check that checks if the value contains the right characters in the given format.

```
if dataFormat == Format.DIGITS:
        return [int(x) for x in text]
```

The next format is `CREDITCARD`, and it is almost the same as `DIGITS`, but it includes a few extra features. First, it removes all spaces, as in some cases spaces are added after every fourth digit. Next, it checks if the checksum is valid. And lastly, it returns all the digits as a list of integers - omitting the last value, as the checksum digit is not encrypted or decrypted.

```
if dataFormat == Format.CREDITCARD:
      text = text.replace(' ', '')
      if (text[len(text) - 1] != validateCard(text[:len(text) - 1])):
          raise ValueError(f"{text} is not a valid credit card number")

      return [int(x) for x in text[:len(text)-1]]
```

The next format is `LETTERS`. This format is also very simple. This format uses another helper function called `map_from_numeral_string`. It takes a string and a map and returns a list of numerals corresponding to the input string.

```
if dataFormat == Format.LETTERS:
    return map_from_numeral_string(text, mapping_letters[0])
```

The next format is `STRING`. This is another version of `LETTERS`, which means that

it is the same principle, but with a different mapping. Instead of being only lower- and uppercase letters, it also includes some special characters.

```
if dataFormat == Format.STRING:
    numerals = map_from_numeral_string(text, mapping_all[0])
```

The next format is `EMAIL`, and this is one of the complicated formats. This format uses three 'plainNumerals', so we first need to identify them. We split the email up into three parts. The first part is the local-part, which is the string up to the '@'-sign. The second part is the domain name from after the '@'-sign until the last '.'-sign because there can be more than one '.'-sign in the email. Lastly, we have the top-level domain, which is the last part of the domain from the last '.'-sign until the end. We create this split by locating the '@-sign, and the last '.'-sign, and then we make three sub-strings, as seen below.

```
if dataFormat == Format.EMAIL:
    first_break_index = text.find('@')
    second_break_index = text.rfind('.')

    text1 = text[:first_break_index]
    text2 = text[first_break_index+1:second_break_index]
    text3 = text[second_break_index+1:]

    numerals1 =  map_from_numeral_string(text1,
                                    mapping_letters_integer[0])
    numerals2 =  map_from_numeral_string(text2,mapping_email_tail[0])
    numerals3 =  map_from_name(text3,mapping_top_lvl_domains[0])

    return [numerals1, numerals2, numerals3]
```

After we found the strings, we create the numerals lists by using the mapping function, described in the previous format, to map the string to their corresponding numerals.

The last format is `CPR`. This format is a mix of some of the other formats. First, it has the same form of validation as `CREDITCARD` but computes the checksum differently. After that, we split the number into two parts. First is the date, which is supposed to be mapped to another date, once we return it to `formatter`, and secondly, we have the year and the sequence numbers, which are just numbers that need to be made into a list of integers. This second part is made just like the format `DIGITS`, and the first part is made with a mapping helper-function, `map_from_name` like the

one we used in, for example, LETTERS. The main difference between these two helper functions is that map_from_name uses the entire string when checking the map, instead of splitting the string up into characters while checking the map.

```
if dataFormat == Format.CPR:
        if (text[len(text) - 1] != validateCPR(text[:len(text) - 1])):
            raise ValueError(f"{text} is not a valid CPR number")

        text1 = text[:4]

        numerals1 = map_from_name(text1,mapping_dates[0])
        numerals2 = [int(x) for x in text[4:]]

        return [numerals1, numerals2]
```

5.6.2 numeral_list_to_text

This function is the last to be called in formatter's encrypt and decrypt functions. This function serves as the reverse of text_to_numeral_list, but for some of the formats, we included some extra features, just like we did in text_to_numeral_list. Because this function is the reverse of text_to_numeral_list, it means that many of the formats will be exactly the same, but with the reverse map of the one in text_to_numeral_list.

The formats DIGITS, LETTERS and STRING are the same as text_to_numeral_list with the difference of the map used, as well as the fact that the numeral lists we get need to be made into strings, and then joined together.

```
if dataFormat == Format.DIGITS:
    return ''.join([str(x) for x in numerals])

if dataFormat == Format.LETTERS:
    return ''.join(map_from_numeral_string(numerals,
                                           mapping_letters[1]))

if dataFormat == Format.STRING:
    return ''.join(map_from_numeral_string(numerals, mapping_all[1]))
```

The format CREDITCARD has some of the same elements and it is still split into three parts, as we just described, but the main difference is that when we put it together,

we add spaces for every fourth digit for readability. We use four different variables containing four digits each, and then they are mapped just like the other formats and put back together with spaces in-between. Since we removed the checksum before encrypting (since the checksum would be very unlikely to be correct by chance), we compute the checksum and then insert it at the end.

```
if dataFormat == Format.CREDITCARD:
    text1 = ''.join([str(x) for x in numerals[:4]])
    text2 = ''.join([str(x) for x in numerals[4:8]])
    text3 = ''.join([str(x) for x in numerals[8:12]])
    text4 = ''.join([str(x) for x in numerals[12:]])

    return text1 + ' ' + text2 + ' ' + text3 + ' ' + text4 +
           validateCard(text1+text2+text3+text4)
```

`EMAIL` is very similar to `CREDITCARD`, as it also contains different parts that need to be put together. This time we need to put an '@'-sign and a '.'-sign between the local-part and top-level domain. The parts are mapped to strings in the same way as `STRING`.

```
if dataFormat == Format.EMAIL:
    text1 = ''.join(map_from_numeral_string(numerals[0],
                    mapping_letters_integer[1]))
    text2 = ''.join(map_from_numeral_string(numerals[1],
                    mapping_email_tail[1]))
    text3 = ''.join(map_from_name(numerals[2],
                                  mapping_top_lvl_domains[1]))

    return text1 + '@' + text2 + '.' + text3
```

The last format is `CPR`, and this is once again a mix of different formats. Its text-conversion from `cipherNumerals` to ciphertext is for the first part (the date) the same as for the other formats containing text instead of numbers, and the second part (the year and the sequence numbers) is converted just like in `DIGITS`. Then, in the end, we need to put the different parts together. There is nothing we need to put in between them, but we need to add a checksum digit, just like we did in the `CREDITCARD`.

```
if dataFormat == Format.CPR:
    text1 = ''.join(map_from_name(numerals[1],mapping_dates[1]))
    text2 = ''.join([str(x) for x in numerals[0]])
```

```
    return text1 + text2 + validateCPR(text1 + text2)
```

5.6.3 get_radix_by_format

This helper-function is also used by `formatter`'s `encrypt` and `decrypt` functions. As the name suggests, it takes a format as input and gives back a radix. Just like in the two other helper functions described in this chapter, this also uses if-statements to check which format's radix (or radices) it should return. The formats `DIGITS`, `CREDITCARD` are the simplest formats. These have a hard-coded radix of 10, shown below.

```
if format == Format.DIGITS:
    return 10

if format == Format.CREDITCARD:
    return 10
```

`LETTERS`, and `STRING` are also very simple, but these formats' radices are tied to the length of their map. The code is shown below.

```
if format == Format.LETTERS:
    return len(mapping_letters[0])

if format == Format.STRING:
    return len(mapping_all[0])
```

As we have described a few times, the format `EMAIL` is split into three parts. Each of these parts has its own radix. Therefore, we have chosen to output a list of radices instead of just an integer. The first index of the list contains the radix of the local-part, the second index in the list contains the radix of the domain name, and the third index contains the radix of the top-level domain. All three radices are made from the length of the given part's mapping.

```
if format == Format.EMAIL:
    radix1 = len(mapping_letters_integer[0])
    radix2 = len(mapping_email_tail[0])
    radix3 = len(mapping_top_lvl_domains[0])
    return [radix1, radix2, radix3]
```

The last format is `CPR`. As with email, this format contains more than one radix. So

once again, we chose to return a list. The first index is the radix of the dates, and its created by the length of the map used to handle the dates. The second index is just 10, just like in `DIGITS` and `CREDITCARD`, as this should also just be treated as integers.

```
    if format == Format.CPR:
        radix1 = len(mapping_dates[0])
        radix2 = 10
        return [radix1, radix2]
```

5.7 Mode selector

As of now, we only have 2 modes. Like formats, we chose to represent them as integers, and they are shown below.

```
FF1 = 0
FF3 = 1
```

The `mode_selector` is in the layer just above the algorithms layer. The `mode_selector` is responsible for running either FF1 or FF3-1 depending on the mode chosen by the user. This is done with simple if-statements and is designed to be easily expandable. Doing it in this way means we do not have to include a mode check for all the formats in the `formatter`. The code for encryption can be seen below. Decryption is almost exactly the same, but with `decrypt()` instead of `encrypt()`.

```
if mode == Mode.FF1:
        return ff1.encrypt(plainNumerals,key,tweak,radix)
elif mode == Mode.FF3:
        return ff3.encrypt(plainNumerals,key,tweak,radix)
```

6 Evaluation

In this thesis, we have investigated several research questions that have directed the development of our product. To evaluate the quality of our solution, we will in the following chapter discuss how well the structure of our final program fulfills our design goals, the security of our solution, to what degree the project process corresponded with our initial milestones, and optimizations.

6.1 Usability

We started by testing the usability of our library ourselves. This was done to ensure that all of the functionality was easy to implement in small scripts. However, we cannot expect users to have the same intricate knowledge of the library as its developers. To test the usability in a more realistic scenario, we asked fellow developers to install the library. We only asked developers with previous experience in Python, but no knowledge of our library. Furthermore, we did not provide any help except the general README that can be seen in appendix E. They were asked to run the examples given in the README and take note of the time they used from installing the library to running a script. This test showed that for Linux and Mac users, it only took a couple of minutes from running "pip install FPE" to running the example scripts. For Windows users, this process was cumbersome due to the need for a C compiler. The process of installing a C compiler on Windows is not easy, and the installation itself takes time. Generally, Windows users spent around 30-60 minutes on installing the library, unless they had already installed a C compiler. Developers within the crypto topic are likely to encounter a lot of C code for performance reasons. It is therefore not unlikely that developers using our library will already have a C compiler.

6.2 Structure

The structure of the program followed a layered design with five layers: algorithms, modes, formats, features, and interface. Throughout the implementation of the program, we followed this structure. In figure 10, the structure of the final program is illustrated. The general idea still mirrors the design detailed in section 4.1: `FPE` serves as an interface that interacts with the features and `formatter`. The `formatter` formats the input string and calls the mode selector. The mode selector then calls the corresponding algorithm. Additionally, a new layer has been added at the bottom. This layer contains all the external libraries that have been imported. Furthermore, `mode`, `format`, and `format_translator` have been added. The `mode` and `format` files contain all the modes and formats currently available for use. Having these in separate files allows other files to easily import them. Currently, they are used in the formats and modes layers, but also in the interface. This is done to let users easily access all of the formats and modes available through `FPE`. The last addition is the `format_translator`. The purpose of the `format_translator` is doing the actual mapping from a string to a list of numerals. The reason for a separate file here is to make the `formatter` more manageable. The `formatter` calls the function in `format_translator` that corresponds to the current format of the input. In

that way, when a new format is added, it can be added in the `format_translator` without changing `formatter`. This also means that all the files that are dependent on `formatter` remain unchanged. This is the case for most formats, but not for complicated formats such as email and CPR.

The design of the structure provided a great guideline to follow throughout the implementation of the program. The structure also meant all of the features could be accessed by only importing `FPE`. Only four lines of code are needed for encryption of either a single input or an entire file. The `fpe_csv` file also allows users to generate test data, which was an extra feature we added to test our program. The structure also allows for great feedback to the user, since an error can identify which layer the error happened in. For example, if the user inputs a format that does not exist, the error will be raised in the format layer. The feedback will then say that the format is invalid and provide a list of valid formats.

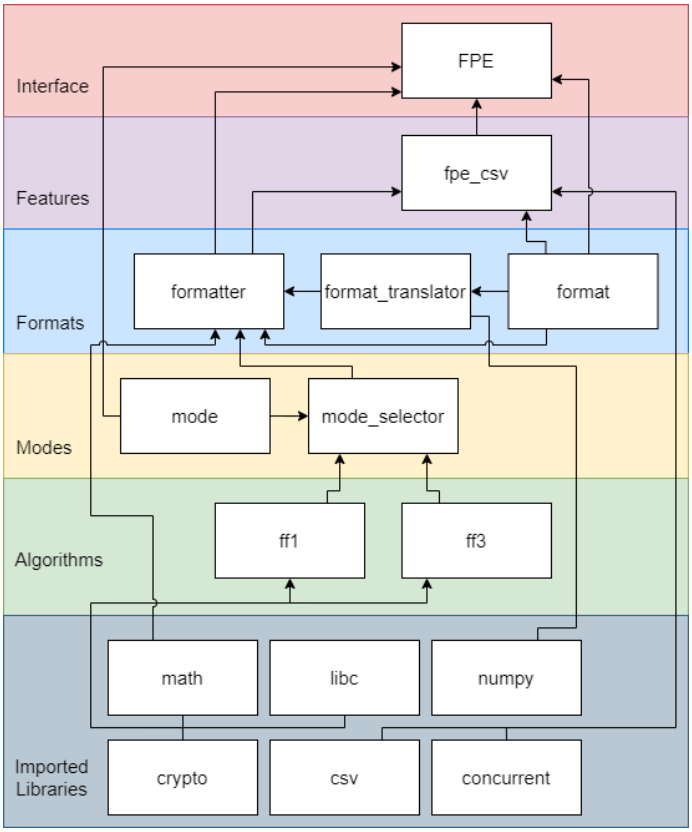


Figure 10: Structure of the final version of the program

6.3 Milestones

Before starting the process of creating our library, we defined several project milestones to guide the development of the project. The content of these milestones, as well as the thought process behind them, are described in section 4.2.

In this section, we will describe to what degree the project managed to follow the milestones.

When we developed our library, we completed it in the steps corresponding to our milestones. The steps are listed below and can be regarded as how our milestones turned out during the course of the project rather than how they were planned in the beginning. Our four stages were:

- **Initial version** - aka. First prototype: We got encrypt and decrypt to work
- **Working version** - aka. Alpha build: Implemented all the different formats, base functionality is now working.
- **Optimized version** - aka. Stable build: Optimized the algorithms and structured the library.
- **Final version** - aka. End-of-build: Finalized the refinement of our library.

6.3.1 Initial version

The initial version of the library focused on the two algorithms with the intermediary goal of supporting integers and possibly strings. This turned out to be just supporting numeral strings, which are just numbers, but no strings at this point.

The structure of the initial version of our library only consisted of two files and a few imported libraries (e.g. a math library). These two files were basic versions of the two files `FF1` and `FF3` in the "Algorithm" layer, seen in figure 10. They served as a good underlying basis to continue our work but were optimized in a later milestone.

These two files both consisted of their respective encrypt- and decrypt functions, the helper functions that were needed for encryption and decryption, and the information they needed to run, i.e. the radix, key, message, and so on. Because of this, most of the code was repeated twice. Even though this goes against our second goal of modularity, it was sufficient for a proof-of-concept, and it aided us in developing one step at a time. This version was very slow, and, as stated, it only worked with numbers as input, so we decided to implement the formats next.

6.3.2 Working version

In this version, the library was expanded with all the formats that ended up in the final version. A main file was added to tie the formats and the two algorithms together. This main-file contained the logic and computation that was later split up into `formatter` and `format_translator`. In other words, this main file was the entire "Formats" layer and also functioned as an early version of our "Mode" layer. As such, a lot of our disparate parts were kept together in a single file, which quickly became chaotic. This naturally led us to split it up into different files in the next milestone.

Our main-file contained a lot of if-statements to handle our formats in a similar fashion to `formatter` and `format_translator`. In these if-statements, we had all of the calculations and computations needed to perform on the different formats. We also made a small file called `csv_to_json` that had the task to create a Json file that we could use to create the maps used in some of the formats. This file had no impact on the execution of our encrypt and decrypt functions, but created the Json files, which are used in the later versions.

At this point, even though the library needed to be optimized and refined, the product had already reached a point where it could encrypt all the formats we had planned, which constituted a major part of the final product. On the other hand, since the "main" code was not well-structured, implementing new formats would still pose a challenge. The goal of modularity was not reached at this time.

6.3.3 Optimized version

Although this version of the library is named the "optimized version", optimization was not the only addition to the library for this version. According to the plan, this build would contain all the features of the final build except for minor changes. Much of this was fulfilled: we had begun refining, and added the features, we wanted to add, and were finished with optimization. This was very close to the goal we had set, the only thing we would have liked to have changed at this point was how much refining, we had left to do. We had split our main-function up into `formatter` and `format_translator` that we use in the final version. At this point, we had finished our "Algorithms" layer with our two optimized files FF1 and FF3-1 that contained the encryption and decryption of numerals strings with FF1 and FF3-1, and a few helper functions that were optimized together with the two algorithms. We were almost done with the "Formats" layer, where we had split our old "main" file up into `format_translator` and `formatter` that was handling the format processing

and selection logic. We also had the few Json files we created in the "Working version", that we used to create the maps.

The file `csv_to_json` got expanded to include the features `create_test_data` and `encrypt_database`. These features make up the Features layer, and the reason we chose to add these features will be described in the next section. Our optimization mainly consisted of converting our encryption and decryption code to *Cython*.

At this point we felt like we had all of our core features in place and that all we needed to do was to refine the code one last time so that everything would fit better together.

6.3.4 Final version

To get to the final version we still had some refinement to do. The main part of the refinement was to tie everything together in an interface that would make up the "Interface" layer. This file is called `FPE`, and it is where we created a constructor for an `FPE-object` that would operate as our interface. We had the idea that when our library is used, it is easier to create an object that contains all the information needed, and then you can use our library's functionalities via that object.

Our `FPE` object has five functions that cover three features. The first feature is covered by `encrypt` and `decrypt` that is calling `formatter` and they are a single encryption and decryption. `formatter`'s `encrypt` and `decrypt` ties many of the different files together. It uses our `mode_selector` to encrypt and decrypt with either FF1 or FF3-1, and `format_translator` to handle the format-logic. `Format_translator` contains some of the helper-functions that are used, mainly in `formatter`. These helper-functions take care of almost all of the format-logic, including creating the maps, and transforming the formats back and forth between the plain- and ciphertext, and numeral strings.

`FPE` also has the functions `encryptCSV`, `decryptCSV` and `generateData` that calls `fpe_csv`, which covers the two features of encrypting and decrypting an entire CSV file, and generating a CSV file with test-data. `fpe_csv` only contains logic to complete these tasks, apart from that it calls `formatter` for every encryption and decryption.

Our Json files we created in the *Working version* and used in the *Optimized version* were deleted in this version, because it was easier not to deal with the Json files when we converted our project to a library. Instead the Json objects we used from the files got put into `format_translator` as hard-coded lists.

6.4 Security

In the design, we emphasized the importance of implementing FF1 and FF3-1 according to the specification from NIST. This was to avoid any security risks that could result from an incorrect implementation. First of all, we made sure to implement the algorithms as closely to the pseudocode as possible. We also made sure to not add any extra features inside the algorithms themselves. All the added features and formatting of the library happen outside of the algorithms so that the output of the algorithms matches the expected output. To guarantee the implementations worked as intended we did some conformance testing with examples from NIST[13][14]. These examples contain the intermediate values of each step in the algorithms. In FF1, we could simply print the value in each step and compare it with the values from NIST. For FF3-1, this was more difficult, since NIST has not updated their values for FF3-1, and was still using the values for FF3. The difference between FF3 and FF3-1 is the tweak being 56 bits instead of 64. FF3 also only requires the domain size to be 100, however, all examples have a domain size of over a million, so this is no issue. To conformance test FF3-1, we changed the tweak to 64 bits purely for testing purposes. In this way, we could test our implementation was correct for everything but the tweak. For the tweak, we could only rely on ourselves implementing it correctly. Once we had confirmed that the intermediate values were correct, we assumed that the implementation was secure.

The cipher used in FF1 and FF3-1 is imported from the PyCryptodome library. This is an open-source library that is used in more than 30,000 projects. This is generally the only crypto library for Python, which also makes it quite well-tested. We took a look at the source code for AES and generally found the library trustworthy. We would have liked to make the cipher ourselves for security and optimization purposes but chose to prioritize other features. PyCryptodome supports AES with AESNI to speed up the cipher, as well as ChaCha. This meant we could relatively easily change the cipher to ChaCha for CPUs without the AESNI.

The formats we chose to support also required some attention to security. This is due to the unique way we encrypt emails and CPR numbers. Emails have a finite list of top-level domains that can be used, which means just randomly encrypting the top-level domain can be wrong. The same applies to CPR where there is a finite number of valid dates, where random digit encryption can land on a non-existent date. To solve this, we had the idea of having a list of all possible values. We then shift the current value based on the plaintext. This poses a challenge, since two similar plaintexts would get encrypted to the same ciphertext regardless of the key. This could lead to attacks where lists of all possible encryptions could be pre-computed

like in MD5. To solve this, we also use the key when shifting. This works a bit like adding a salt to MD5, negating the attack. The biggest concern for this strategy is a known-plaintext attack. If the attacker knows both plaintext and ciphertext, he can subtract plaintext shifting from the ciphertext. In this way, the ciphertext would only be shifted by the key. This is however modulo radix and would still result in an exhaustive key search of at least $2^{128}/2^{16} = 2^{112}$, which would take a long time.

The only security risk that is left. is the risk that comes with keeping the length intact. By definition, FF1 and FF3-1 keep the length of the input intact. Changing the lengths securely and randomly is difficult when the ciphertext needs to be decrypted as well. We could not find a solution to this using FF1 and FF3-1. Another solution could be making it possible to decide if the ciphertext should be decryptable. In case there is no need to decrypt the ciphertext, it is trivial to randomize the length.

6.5 Test data

To test our algorithms we needed some data to encrypt and decrypt. We were given some data by PII Guard, but the data we received was not a good fit with our FF1 and FF3-1 algorithms. The data was not a good fit for our project since most of the values cannot be encrypted using any of our current formats. Some of these fields need to have for example a valid job title. For some values, the space is way too small for it to make sense to encrypt. We could have used a lookup table containing the entire space, and instead of encrypting it, we could shift it by some value instead. However, this is not an ideal use case for our project. Just shifting on a list could be insecure, and is mainly used when decrypting is not needed. Additionally, we would not be able to use our FF1 and FF3 to encrypt these types of values, as the min-length of the message we encrypt is two numerals. With these mappings, we would only have one numeral, so we would have to use another encryption method regardless.

There are of course a few fields that we would be able to encrypt, and where it makes sense to use our FF1 and FF3 encryption, but we decided to create our own test data instead to tailor it to our project. We created our test data using our own encryption algorithms. We decided to create a dummy string for each of our formats, and encrypt it with a lot of random keys. This way we get a lot of random values for our test data. Our test data will look like gibberish, but since we created it using our encryption algorithms, they will be valid.

Username	Password	Email	PhoneNumber	Cpr-number	Creditcard
mBtFMUhRXWZa	*HRV9+é3uGBOFB2HI	CG2æ@hhF5t.consulting	95031731	411965635	0225 3450 9568 6207
ikuzxnrgÅZku	?_9)Zéå+*%sJr$mac	ÆUQl@RPs6W.imdb	6204172	1805597565	0581 4332 0159 4025
æJWDgmøwWvxh	(3s:´!npP5Ux'Å_a#	UDKQ@xvtyo.xn--2scrj9c	99799410	205521003	1614 0444 4516 8077
RsåuNyzjetÆJ	!zøXjfMÅ4<1kF_0$H	æpkb@L5ømÆ.adult	57743543	2609474591	8188 3866 2771 9015
åPvyhxXqÆSkk	R8r}Å&Vre|q UeBb~	AzXC@Xkc5Q.nl	61114896	1609326820	0444 3780 7390 3927
tqnYbøøtTvZu	O5,ÆJ6mT51'8PÅ%8u	SzvG@T325O.va	33483056	1112155968	2290 5747 9908 7611
ahqYKaPtgErk	é`BopT6A*7ØfæR:`.	Nw7Æ@ClåJh.xn--90ais	97544723	2408595994	6948 5163 8222 9413
IjJCkKXxIhAV	#*i&nT:æ|,8kUqF.o	iXus@ÅYøMs.house	58056211	1611788963	7929 7549 2945 6790
BcTCwagRStrs	:xoc8&l4jÅ6g`>´i2	TF9a@dæUDB.forex	22590994	1508182585	9166 0820 2132 0823
vAHYQÆFOwhAC	)3<M:! 4~~Kwxi~éa	båiN@ksæUt.spot	26596062	1112800073	4774 1920 3748 9788
shprMKDMDEåz	nl|kXdøqIovhØ£B7c	LL3U@fbPUÆ.sony	38292946	905966871	2059 6358 9830 7879
cihJvQicÅKyå	Ø8^u~7:l´MRMZu>7m	mjIA@OTUPE.xn--mgbah1a3hjkrd	35908812	1501524405	0905 7474 7776 3462
UeSøLQiNCcmC	9vm1d%g$g>Øp,h}-Å	XqjP@AlPWV.link	65362097	3008722195	9884 0021 9611 9122
BaXVcYVfRKøp	yså6I>VsI.Ø<m5V/a	bP3W@øe.Qd.cheap	52868450	1703333962	5098 2422 6982 2713
jNIagqåOBlKv	XQ7Lyz}6CØå=n`Æ7_	kyMJ@vZUW6.zip	5266201	2203699983	3990 3313 7733 1545
dPbuxyjEeuWw	´^zopåo?.XKhL-^q<	LUUQ@vId95.whoswho	56492662	602212971	8918 4934 0907 7485
GkøRhHstCcIV	(89ZRE~SiWWÅXC=K%	8iøF@KhUwå.aq	96070729	904570184	0639 0241 5094 3346
tuaBmÅænQOAc	AR+Tn)a4HmKæ?sY~	kovæ@FgHbM.xn--qxa6a	51351765	2603398478	0402 2072 7761 8844
XZPfSUdScjÆP	Æ!Ii:D#^pI1ÅAIq`8	XDGg@knnJæ.dm	91569805	2312636037	5178 2046 6839 5372
ÆOjEaBÆnNPSC	:sLAg£?RxNød^tDKw	79Vk@w9L.C.luxury	67125352	611983875	6871 4104 6716 0960

Figure 11: Our own test-data

We chose to include this as a feature for our library, where you can create your own structure and size of test data, though within the scope of our formats. So you can decide the number of columns and rows, the formats each column should be encrypted with, and the names at the top of each column.

6.6 Optimization

The optimization of the program primarily consisted of changing Python code to Cython. This was done with the help of the Cython compiler. The compiler keeps track of how much C code it generates for each line of Python code. Figure 12a shows the output of the compiler with pure Python code. The darker the yellow, the more C code is generated. Because the lines are almost all dark yellow, the 7 lines of Python code become several hundred lines of C code. Figure 12b shows the output of the compiler with Cython code. Most of the lines are directly compiled or compiled with few lines. This means, even though the Cython code is 28 lines, it compiles to less than a hundred lines of C code. It is important to note that not all dark yellow lines are a problem. Some lines require a lot of C code but still run fast. For example, a single line with a 100,000 round for-loop is slower than 100 lines of variable assignments. This serves as a guideline but the goal is not removing every single yellow line, as in that case you might as well write it in C.

```
+019: def PRF(X,cipher):
+020:     m = int(len(X)/16)
+021:     Yj = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
+022:     for j in range(m):
+023:         Xj = X[j * 16:(j * 16) + 16]
+024:         Yj = cipher.encrypt(bytes(A ^ B for A, B ir
+025:     return Yj
```

(a) Python code compiled to C

```
+035: cdef bytes xor16ByteArray(bytes A, bytes B):
 036:
+037:     cdef unsigned char* Ac = <unsigned char *> mal
+038:     cdef unsigned char* Bc = <unsigned char *> mal
+039:     cdef unsigned char* xor = <unsigned char *> ma
 040:
+041:     if not Ac or not Bc or not xor:
+042:         raise MemoryError()
 043:
 044:     cdef int i
+045:     for i in range(16):
+046:         Ac[i] = A[i]
+047:         Bc[i] = B[i]
+048:         xor[i] = (Ac[i] ^ Bc[i])
 049:
+050:     cdef bytes result = xor[:16]
 051:
+052:     return result
 053:
+054: cdef bytes PRF(bytes X,cipher):
+055:     cdef int m = len(X)/16
+056:     cdef bytes Yj = b'\x00\x00\x00\x00\x00\x00\x00
 057:     cdef int j
 058:     cdef bytes Xj
+059:     for j in range(m):
+060:         Xj = X[j * 16:(j * 16) + 16]
+061:         Yj = cipher.encrypt(xor16ByteArray(Xj,Yj))
+062:     return Yj
```

(b) Cython code compiled to C

Figure 12: Difference in Cython and Python

These optimizations brought the runtime of 100,000 encryptions/decryptions down to 17.7 seconds for FF1 and 10.7 seconds for FF3-1. This test shows the runtime of the algorithms but is not a great real-world example. It makes no sense to encrypt and decrypt the same input 100,000 times other than to test it. A test that reflects a real-world scenario better is encrypting a large database. An example of this could be a database with a million users. For each user, there are 10 columns of data in the database. This totals to 10 million encryptions/decryptions. If we use the runtime from before this would take 1770 seconds for FF1 and 1070 seconds for FF3-1. However, this time is only based on a simple format with 6 digits. For a more advanced format like emails, the 100,000 loop runtime becomes 38.3 seconds for FF1 and 23.5 seconds for FF3-1. With this in mind, we can assume that 10 million encryptions/decryptions without optimization would take around 1770-3830 seconds for FF1 and 1070-2350 seconds for FF3-1. In many scenarios, the database would only have to be encrypted once. In that case, these runtimes would not be a huge issue. However, some use-cases might require the entire or large parts of the database to be encrypted regularly. A runtime of more than an hour would then be a problem and it would need to be improved.

The biggest challenge was the fact that the CPU utilization was very limited. The CPU only used a single thread to encrypt/decrypt a single value at a time. Using the same Intel I5 8250U CPU with 8 threads, we only used around 25% of the CPU when encrypting. This could be solved by utilizing more threads when encrypting files. We accomplished this by using the internal Python library `concurrent`. Using `concurrent`, we assigned all available threads to a column each. In this way, our 8-thread CPU could theoretically encrypt 8 columns at a time if all 8 cores are available. Notice that all of the threads might not always be available to allow room for other programs.

We then tested it on a test database, where we generated 10 columns of both advanced and simple formats. The test database had a million rows and therefore computes 10 million encryptions/decryptions. The test used FF3-1, so without multithreading, this would likely take around 1700 seconds (about 30 minutes). The total runtime of encryption and decryption was 667.4 seconds (about 11 minutes). This is almost 3 times faster than the expected runtime without optimizations. The optimization also scales better on higher-performance CPUs. To illustrate this, we ran the same tests on an AMD Ryzen 5 5600X. The unoptimized runtime on this CPU is estimated to be around 1000 seconds. When we ran the test on the test database, the optimized runtime was 204 seconds (3.5 minutes). This means the optimized version is about 5 times faster on this CPU. Comparing this to the almost 3 times faster runtime of the Intel CPU proves it scales well with CPU performance. This scaling does have its limits, though. The scaling is bound to the number of columns in the database, so adding more rows scales worse. A better scaling would be if the threads were bound to the number of rows instead since most databases have more rows than columns. This would mean that for HPC clusters with practically infinite threads, the computation would be almost instant. We did try to make this work, but it made the computation significantly slower. This could be caused by many things and is something that could be improved upon in future work.

100,000 encryptions and decryptions with FF3-1 took 10.7 seconds. We decided to test exactly how much of this time was spent on AES. Running only the AES 200,000 times took 4.1 seconds. This can theoretically be heavily improved upon. Intel made a paper[1], where they showed a graph stating that the optimal runtime for AES on their processor is 4.20 clocks per byte or 67.21 clocks per block of 128 bits. When we run AES, we encrypt 1 block of 128 bits at a time, so in theory, when we run AES 200,000 times, it should take around 13.4 million clocks. Converting that to seconds, most processors today have at least 3 GHz, which would be 0,004 seconds. This is of course a purely theoretical time, but the statement that 4.1 seconds for 200,000

encryptions seem slow.

The most likely cause for the slow AES is the fact that the PyCryptodome library only uses C for the most performance-critical parts. When calling the encrypt function, a lot of different calls are made using Python, which can slow down the process a tiny bit. When doing this 200,000 times it could easily cost a couple of seconds. If we were to implement AES ourselves it would likely be possible to almost entirely remove the 4.1 seconds spent on AES.

Large amounts of encryptions and decryptions are not the only thing slowing down computation a lot. Large inputs can also slow down the computation significantly. In FF3-1 this issue is solved simply by not supporting large inputs, but FF1 supports very large inputs. We tried the same 100,000 loop test on an input of 1000 digits, and the runtime was 814.1 sec. The PRF part of this takes around 75.6 seconds, which means purely operating on large numbers adds more than 700 seconds. This could be optimized by the "Shift and subtract" algorithm. We are unsure of exactly how much faster this would be, but theoretically, it should almost entirely remove the 700 extra seconds. We choose not to prioritize optimizing long inputs, since inputs longer than 20 characters seem unlikely.

7 Future Work

During the work on the thesis, a number of potential improvements of the solution were identified, but not investigated further due to constraints of either time or scope of the project. In the following chapter, these potential improvements are discussed and analyzed with regards to their impact to the project.

7.1 Improve and expand our formats

One thing that can always be improved upon is the number of supported formats. Some examples include a specific format for usernames, which would be letters and a few chosen characters, or maybe a CPR number format without validation so that it is more broadly used in a Danish context. Apart from these, there are several small tweaks you could apply to the existing formats that might fit into some niche use-cases.

This is not, however, the first thing we would improve. As stated earlier, we have not implemented a method to create varying lengths for e-mails (nor for other formats). We wrote about the fact that this feature is important for the security of the format,

but did not touch upon how one would create this feature. This feature is a difficult problem to solve, but not an impossible one. One difficulty is that if we remove characters to randomize the length of outputs, then we lose the information we removed, so it might not be possible to decrypt it again. On the other hand, if we add the possibility to add characters to the string, then if we decrypt it again, then how do we know if the string had added some padding, or if it was a part of the string in the first place.

We thought about adding some padding that was outside the space, but it adds new problems to be able to map something from outside the space to something inside the space because it means that you could now also accidentally go the other way and map from inside the space to a character outside the space, which would compromise the given formats. Although none of this is a problem if we design it as a feature without the possibility of decryption, in this case, it would be very easy to just add padding or remove characters before encrypting it.

The problem about the length of the string is not only a problem for e-mails, but for every format including strings or numbers of no particular length, which is most of our formats. So this safety measure would be important for us to add as the next feature.

An additional format to add could be a `custom` format, where the user of our library can use his own format without creating a format from scratch. We imagined that this format would create a format from the information you give to the program. We had several ideas to how this custom format could be implemented. One idea is that the program uses all unique characters used in the plaintext to create a domain from that, but this would not be very secure, as the ciphertext reveals too much about the plaintext. Another option, we thought about, is to give the program an extra optional parameter that would only be used if you set the format to `custom`. This parameter would be a list of the domain, just like the one we use in our `format_tanslator` file, to create the domains `dates` and `top-lvl-domains`. This list would be used to create a map, again just like we do in `format_translator`. This map would only be used for one execution, so it would be a lot slower than the other formats, as it would have to create a map every time it should encrypt and decrypt something.

This one `custom` format would be very generic, and with no special feature inside it, so it would not be possible to create more complex formats such as *email* or *CPR* with it. Maybe we could create different modes of `custom` formats, with some different generic templates. This could for example be some `costom_creditcard_numbers` format so that it could be specified how many numbers are in the card number,

as this varies in different countries and by different providers. Or maybe create a `custom_CPR` that could work with the other countries' social security numbers, either by adapting to their number of digits or maybe also by adapting their validation techniques.

7.2 Add new FPEs

As stated in 6.5, when we received the test data from PII Guard, we decided to make our own test data instead, since the test data was not a good showcase for FF1 and FF3. As an additional feature of our library, we could add an FPE that worked well with the inputs from PII Guards test data. This could be an FPE that works similarly to the way we handle dates in our CPR format. This small part of our encryption uses a map of all the dates, where we shift the date by some amount according to the values of the rest of the message and the key.

We could take a similar approach, and develop an FPE that encrypts one *numeral*, e.g. with AES, maps this new numeral to a new object in the space. This could for example be a name, where we use a list of names and get a new name out of the encryption instead of what seems to be a random string. This FPE could either be specific with pre-defined lists of spaces that would work as a format. It could also be very general and take a list of the entire space as input together with the message, and it would automatically create the radix, encrypt it with AES, and map it to a new value in the space.

Since we designed our code to be expandable, it would be a straightforward task to add new FPEs. It would only require changes to the bottom layers of our library. Only a few changes in `mode_selecter` and in `modes` are needed.

7.3 Implement internal cipher

As a future expansion to our library, we would like to implement a feature we wrote about in section 6.6. More precisely, the feature about creating our cipher function. We already mentioned that using PyCryptodomes cipher was slow in the evaluation. Implementing our own cipher would mean we could optimize it and make the library faster.

7.4 Implement shift and subtract algorithm

In our project, we did not work extensively with large numbers. Neither in the form of long messages nor large radices. Therefore, it has not proven necessary to improve the performance of modulo on large numbers. We have described an algorithm in section 3.8 for reducing numbers using modulo with constant time complexity. In other words, no matter the size of the inputs, the runtime of the algorithm is fixed. This would be a useful addition to the library for work on very large numbers. This algorithm would also theoretically be faster for small numbers, however, the difference would likely be unnoticeable.

7.5 Choosing compiler

Some developers may use our library for small applications, where performance is not critical. For Windows users, it could be an issue having to install a C compiler just to encrypt a few things. For these reasons, we could add the ability to run the encryption purely in Python. This would make installing the library a lot easier for Windows users, and the only downside is slower performance.

8 Conclusion

The goal of the project was to make FF1 and FF3-1 accessible to a wide range of developers in all kinds of projects. To help realize this goal, we stated several research questions to address throughout the project. To investigate these questions, we started by acquiring the necessary knowledge base. This was done by thoroughly analyzing the related topics. This knowledge helped us design a library in conformity with the requirements for FF1 and FF3-1. It also provided a base for improving the quality and performance of the implementation.

One of the questions, we had, was how to make the library user-friendly and functional. This was solved by designing a layered structure for the program. The top layer contains an interface that the user interacts with. This allows the user easy access to all the functionality of our library. Furthermore, the layered structure served as a guideline for us when we implemented the library. We also wanted to address how to ensure that we met the deadline for the project. For this purpose, we defined the scope of the project and created project milestones. These milestones defined the different stages of the program, which was also complemented by the modularity of the layer structure. Each milestone was defined by one or more layers. We reached a milestone once the functionality of the corresponding layer(s) was implemented successfully. This meant we also had an overview of the progress of our library throughout the project.

To validate that our library addressed the issues, we thoroughly tested it. Part of the testing was done to evaluate the runtime of the program. This was necessary to optimize our program since we chose Python as the programming language. We tested the library's conformance to the security requirements set by NIST. These security requirements are based on recommendations from FPE researchers around the world. We also tested the usability of the program by asking Python developers to install and use our library. Based on the conducted tests, we are confident that the project has realized the initial goal.

9 Appendix

A FF1 encryption pseudocode

Algorithm 19 FF1 Encryption

Let $u = \lfloor n/2 \rfloor; v = n - u$
Let $A = X[2..u]; B = X[u+1..n]$
Let $b = 4\lceil \lceil v \times \text{LOG}(radix) \rceil / 8 \rceil$
Let $d = 4\lceil b/4 \rceil + 4$
Let $P = [1]^1 || [2]^1 || [1]^1 || [radix]^3 || [10]^1 || [u \bmod 256]^1 || [n]^4 || [t]^4$
for i from 0 to 9: **do**
    Let $Q = T || [0]^{(-t-b-1) \bmod 16} || [i]^1 || [\text{NUM}_{radix}(B)]^b$
    Let $R = \text{PRF}(P||Q)$
    Let S be the first d bytes of the following string of $\lceil d/16 \rceil$ blocks:
    $R||\text{CIPH}_K(R \oplus [1]^{16})||\text{CIPH}_K(R \oplus [2]^{16})...\text{CIPH}_K(R \oplus [\lceil d/16 \rceil - 1]^{16}$
    Let $y =$ NUM(S)
    if $i \bmod 2 = 0$ **then**
        Let $m = u$
    else
        Let $m = v$
    end if
    Let $c = (\text{NUM}_{radix}(A) + y) \bmod radix^m$
    Let $C = \text{STR}^m_{radix}(c)$
    Let $A = B$
    Let $B = C$
    Return $A||B$
end for

B FF1 decryption pseudocode

Algorithm 20 FF1 Decryption

Let $u = \lfloor n/2 \rfloor; v = n - u$
Let $A = X[2..u]; B = X[u+1..n]$
Let $b = 4\lceil \lceil v \times \text{LOG}(radix) \rceil / 8 \rceil$
Let $d = 4\lceil b/4 \rceil + 4$
Let $P = [1]^1 || [2]^1 || [1]^1 || [radix]^3 || [10]^1 || [u \bmod 256]^1 || [n]^4 || [t]^4$
for i from 9 to 0: **do**
    Let $Q = T || [0]^{(-t-b-1) \bmod 16} || [i]^1 || [\text{NUM}_{radix}(A)]^b$
    Let $R = \text{PRF}(P||Q)$
    Let S be the first d bytes of the following string of $\lceil d/16 \rceil$ blocks:
    $R||\text{CIPH}_K(R \oplus [1]^{16})||\text{CIPH}_K(R \oplus [2]^{16})...\text{CIPH}_K(R \oplus [\lceil d/16 \rceil - 1]^{16}$
    Let $y =$ NUM(S)
    if $i \bmod 2 = 0$ **then**
        Let $m = u$
    else
        Let $m = v$
    end if
    Let $c = (\text{NUM}_{radix}(B) + y) \bmod radix^m$
    Let $C = \text{STR}^m_{radix}(c)$
    Let $B = A$
    Let $A = C$
    Return $A||B$
end for

C FF3 decryption pseudocode

Algorithm 21 Pseudocode for FF3-1 decrypt

Let $u = \lceil n/2 \rceil$
Let $v = n - u$
Let $A = X[1..u]$
Let $B = X[u + 1..n]$
Let $T_L = T[0..27]||0^4$
Let $T_R = T[32..55]||T[28..31]||o^4$
for i from 7 to 0 **do**
    if i is even **then**
        Let $m = u$
        Let $W = T_R$
    else
        Let $m = V$
        Let $W = T_L$
    end if
    Let $P = W \oplus [i]^4||[\text{NUM}_{radix}(\text{REV}(A))]^{12}$
    Let $S = \text{REVB}(\text{CIPH}_{\text{REVB}(K)}\text{REVB}(p))$
    Let $y = \text{NUM}(S)$
    Let $c = (\text{NUM}_{radix}(\text{REV}(B)) - y) \bmod radix^m$
    Let $C = \text{REV}(\text{STR}^m_{radix}(c))$
    Let $B = A$
    Let $A = C$
end for
return $A||B$

D Shift and Subtract example

Example of Shift and Subtract with bits instead of bytes
b = 7
d = 12
div = 0
y = 2519 = [9] [13] [7]
radix$^{\text{m}}$ = 114 = [7] [2]

#1 leftshift radix$^{\text{m}}$ d-b-1 = 4 times

$\text{radix}^{\text{m}} = [7]\ [2]\ [0]$

#2 subtract radix from y

$[9]\ [13]\ [7]\ \text{y}$
$-[7]\ \ [2]\ [0]\ \text{radix}^{\text{m}}$

$[2]\ [11]\ [7]\ \text{y}$
$\text{div} = \text{div} + 1 = 1$

#3 rightshift radix$^{\text{m}}$ until radix$^{\text{m}} \geq$ y

$\text{radix}^{\text{m}} = [3]\ [9]\ [0]$
$\text{radix}^{\text{m}} = [1]\ [12]\ [8]$
$\text{div} = \text{div} \cdot 2^2 = 4$

#4 repeat step 2 and 3 until radix$^{\text{m}} = [7]\ [2]$

$\overset{-1}{[2]}\ \overset{-1}{[11]}\ [7]\ \text{y}$
$-[1]\ [12]\ [8]\ \text{radix}^{\text{m}}$

$[0]\ [14]\ [15]\ \text{y}$
$\text{div} = \text{div} + 1 = 5$

$\text{radix}^{\text{m}} = [0]\ [14]\ [4]$
$\text{div} = \text{div} \cdot 2^1 = 10$

$[0]\ [14]\ [15]\ \text{y}$
$-[0]\ [14]\ \ [4]\ \text{radix}^{\text{m}}$

$[0]\ \ [0]\ [11]\ \text{y}$
$\text{div} = \text{div} + 1 = 11$

$\text{radix}^{\text{m}} = [7]\ [2]$
$\text{div} = \text{div} \cdot 2^1 = 22$

$\text{y mod radix}^{\text{m}} = 11$
$\text{y/radix}^{\text{m}} = 22$

E Readme

Installation

This library uses PyCryptodome which means a C compiler is required. For Mac and Linux users this is already installed. For Windows users please follow this guide from PyCryptoDome.

https://pycryptodome.readthedocs.io/en/latest/src/installation.html#windows-from-sources

Usage

The libray is imported as `from FPE import FPE`.

To generate a tweak use `FPE.generate_tweak(tweak_length)` where tweak_len is the length of the tweak in bytes. Note that for FF3-1 the tweak_length must be 7 bytes.

To generate a key use `FPE.generate_key()`, this will generate a 16 byte key.

To make a cipher object use `cipher = FPE.New(key,tweak,mode)`

Currently supported modes are `Mode.FF1` and `Mode.FF3-1`

To encrypt use `cipher.encrypt(plaintext,format)`

Currently supported formats are:

`Format.DIGITS`
`Format.CREDITCARD`
`Format.LETTERS`
`Format.STRING`
`Format.EMAIL`
`Format.CPR`

to decrypt use `cipher.decrypt(ciphertext,format)`

The library also supports CSV files

To encrypt a csv file use `cipher.encryptCSV(InputFilePath,OutputFilePath,formats)` where formats is a list of formats sorted by the columns

To decrypt a csv file use `cipher.decryptCSV(InputFilePath,OutputFilePath,formats)`

to generate a random CSV file with certain formats use `cipher.generateData(OutputFilePath,rows,formats,variables)` where rows define the number of rows in the CSV file. Formats is the lists of formats to use. Variables is the list of variable names used for the first row.

Examples

Example of encrypting and decrypting "12345" as `DIGITS` and printing the output

```
from FPE import FPE

if __name__ == '__main__':

        T = FPE.generate_tweak(8)

        key = FPE.generate_key()

        cipher = FPE.New(key,T,FPE.Mode.FF1)

        ciphertext = cipher.encrypt('12345',FPE.Format.DIGITS)

        print(ciphertext)

        plaintext = cipher.decrypt(ciphertext,FPE.Format.DIGITS)

        print(plaintext)
```

Example of generating, encrypting and decrypting a 1000 row CSV file

```
from FPE import FPE, Format

variables = [
        'Username','Password','Email','PhoneNumber','Cpr-number',
        'Creditcard','adress','city','zip','country'
]


formats = [
        Format.LETTERS, Format.STRING, Format.EMAIL, Format.DIGITS,
        Format.CPR, Format.CREDITCARD,Format.STRING,Format.LETTERS,
        Format.DIGITS,Format.LETTERS
]

if __name__ == '__main__':

        T = FPE.generate_tweak(7)
        key = FPE.generate_key()
        cipher = FPE.New(key,T,FPE.Mode.FF3)
        cipher.generateData('testData.csv',1000,formats,variables)
        cipher.encryptCSV('testData.csv','encryptedData.csv',formats)
        cipher.decryptCSV('encryptedData.csv','decryptedData.csv',formats)
```

References

[1] Kahraman Akdemir et al. *White Paper Breakthrough AES Performance with Intel® AES New Instructions*. 2010.

[2] Mihir Bellare and Phillip Rogaway. *The FFX Mode of Operation for Format-Preserving Encryption Draft 1.1*. Tech. rep. 2010. URL: http://cseweb.ucsd.edu/.

[3] Eric Brier, Thomas Peyrin, and Jacques Stern. *BPS: a Format-Preserving Encryption Proposal*. Tech. rep. 2010.

[4] CPR-Kontoret. "Personnummeret i CPR-systemet". In: (2008). URL: https://cpr.dk/cpr-systemet/opbygning-af-cpr-nummeret.

[5] F. Betül Durak and Serge Vaudenay. "Breaking the FF3 Format-Preserving Encryption Standard over Small Domains". In: (2017). DOI: 10.1007/978-3-319-63715-0.

[6] Morris Dworkin. "Draft NIST Special Publication 800-38G Revision 1 2 Recommendation for Block Cipher Modes of Operation 4 Methods for Format-Preserving Encryption". In: (2019). DOI: 10.6028/NIST.SP.800-38Gr1-draft. URL: https://doi.org/10.6028/NIST.SP.800-38Gr1-draft.

[7] ISO. *ISO/IEC 646:1991 Information technology — ISO 7-bit coded character set for information interchange*. Dec. 1991. URL: https://www.iso.org/standard/4777.html.

[8] ISO. *ISO/IEC 7812-1:2017 Identification cards - Identification of issuers - Part 1: Numbering system*. Jan. 2017. URL: https://www.iso.org/standard/70484.html.

[9] J. Klensin. *RFC 3696, Section 3 "Restrictions on email addresses"*. Feb. 2004. URL: https://datatracker.ietf.org/doc/html/rfc3696#section-31.

[10] J. Klensin. *RFC 5321*. Oct. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5321.

[11] Bellare Mihir et al. "Format-Preserving Encryption". In: *Selected Areas in Cryptography*. Ed. by Jacobson Michael J. and Vincentand Safavi-Naini Reihaneh Rijmen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 295–312. ISBN: 978-3-642-05445-7.

[12] P. Mockapetris. *RFC 1034, Section 3.1 "Name space specifications and terminology*. Nov. 1987. URL: https://datatracker.ietf.org/doc/html/rfc1034#section-3.1.

[13] NIST. *Block Cipher Modes of Operation FF1 Method for Format-Preserving Encryption*. Tech. rep.

[14] NIST. *Block Cipher Modes of Operation FF3 Method for Format-Preserving Encryption*. Tech. rep.

[15] Ed P. Resnick. *RFC 5322*. Aug. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5322.

[16] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. Nov. 2021. URL: https://pypl.github.io/PYPL.html.

[17] TIOBE. *TIOBE Index for October 2021*. Oct. 2021. URL: https://pypl.github.io/PYPL.html.

[18] Jana Zabinski and Beth Goodbaum. *Announcing Major Changes to the Issuer Identification Number (IIN) Standard*. July 2016. URL: https://www.ansi.org/news/standards-news/all-news/2016/07/announcing-major-changes-to-the-issuer-identification-number-iin-standard-28.