Faculty of Fundamental Problems of Technology
Wroclaw University of Science and Technology

Format-Preserving Encryption

Adam Budziak
album number: 229747

Engineer Diploma Thesis
under supervision of
Ph. D. Filip Zagórski

Wrocław
University
of Science
and Technology

Wrocław 2018

Contents

Introduction

Format-Preserving Encryption is a field of cryptography that has been meeting with growing interest over time. Its primary purpose is to develop methods for encrypting data of various formats, for example, email addresses, in a way such that the resulting ciphertexts hold the format of their corresponding plaintexts.

There are many widely-used solutions for this problem, and NIST in its Special Publication 800-38G recommended two of them, namely FF1 and FF3 [8]. However, in the light of the recently discovered attacks on FF3 [7] and on Feistel-networks in general [2][12], the demand for different solutions grows.

FPE's possibilities can be further expanded by utilizing methods like *domain targeting*, *domain extension*, and *Rank-then-Encipher*. They provide solutions for narrowing, expanding, and transforming the original format, and allow a single FPE-scheme to be used in various contexts.

This work aims to provide an efficient and easy-to-use library for constructing custom FPE-schemes, and it contains an implementation of a new FPE scheme called GraFPE, which was introduced by Lorek, Słowik and Zagórski [15]. It serves as a base for all schemes constructed later, and for this reason we devote an entire chapter to its description.

The first chapter introduces the key concepts regarding FPE, such as its formal definition, notions of security, current standards, and limitations. Furthermore, it includes a description of *Rank-then-Encipher*, which allows encryption of every efficiently *rankable* and *unrankable* set.

The second chapter concerns the GraFPE protocol and includes both its detailed description and a proof that it is an encryption scheme. Later, we explain all the key concepts on which GraFPE builds upon such as *inverse riffle shuffle* and *simple random walks* over *regular graphs*.

In the third chapter we describe in detail the aforementioned techniques, which extend functionality of FPE schemes. We present three algorithms for *domain targeting*, namely: *Cycle Walker*, *Reverse Cycle Walker*, and *Cycle Slicer*; and show how to specialize *Cycle Slicer* for *domain extension*. Finally, by following [3], we construct an *RtE* scheme that is able to perform encryption on regular languages.

The implementation of the library is presented in the fourth chapter. It includes detailed pseudocodes of the most important algorithms, some with proofs of correctness, as well as examples of usage, and benchmarks.

Format-Preserving Encryption

Classical encryption schemes operate on binary data, and it is a natural approach since in most applications the data is represented by objects stored in physical memory. Usually, after encryption, the messages no longer resemble their original content, which is indeed a behavior we expect. The ciphertexts are still binary, but there is little chance that an encrypted email address still passes validity checks.

The only difference FPE introduces is that we can specify the data format by imposing additional restrictions which must be met both by the plaintext and the ciphertext. For example, $\mathcal{X}$ may be defined as the set of these binary sequences which are alphanumeric when ASCII-encoded, or as the set of even numbers. If such $x \in \mathcal{X}$ is encrypted into y by using an FPE scheme, then also $y \in \mathcal{X}$.

Informally, an encryption scheme E is not secure if there exists any adversary $\mathcal{A}$ (a probabilistic polynomial-time algorithm) that can efficiently distinguish ciphertexts generated by E from truly random data. Finding such $\mathcal{A}$ is a strong argument against using E in real-world applications.

1.1 Motivation

There are many reasons for using FPE, and one of the most important is cost reduction when attempting to encrypt the data stored in an already deployed system. Assume there is a company which holds users' sensitive information as plaintext. Probably, the database expects that the datatype of a CCN (credit card number) is indeed a number, and, similarly, the birth date should be represented by its corresponding date-type if available. There could exist an additional abstraction layer on top of the database that validated the data passed through.

In this situation, attempting to encipher the stored data would require a reconstruction of many components of the already existing environment, and therefore would drastically increase the total migration cost. However, utilizing FPE can give the advantage of not keeping unencrypted information, and without the need of adjusting the other parts of the system. [21]

1.2 Formal definition

To formally define FPE schemes we use definitions and notations from [3].

Definition 1.1 ***of FPE encryption scheme***

Define

- *format space* $\mathcal{N}$*,*
- *domain* $\mathcal{X} = \bigcup_{N \in \mathcal{N}} \mathcal{X}_N$ *and call each* $\mathcal{X}_N$ *a slice,*
- *key space* $\mathcal{K}$*,*
- *tweak space* $\mathcal{T}$*,*

then a scheme for FPE is function:

$$E : \mathcal{K} \times \mathcal{N} \times \mathcal{T} \times \mathcal{X} \to \mathcal{X} \cup \{\perp\},$$

where $\mathcal{K}, \mathcal{N}, \mathcal{T}, \mathcal{X}$ *are non-empty, and* $\perp \notin \mathcal{X}$*.*

Definition 1.2 *For $K \in \mathcal{K}, N \in \mathcal{N}, T \in \mathcal{T}$, and $X \in \mathcal{X}$ the encryption function is $E_K^{N,T}(X) = E(K,N,T,X)$*

Whether E returns $y \in \mathcal{X}_N$, or $\perp$, depends only on the format N and input X. A natural approach is that $E_K^{N,T}(X) = \perp$ if $X \notin \mathcal{X}_N$, and $E_K^{N,T}(X) \in \mathcal{X}_N$ otherwise.

Remark With the definition above, a block cipher such as AES also can be seen as an FPE encryption scheme. In this case $\mathcal{N} = \{128^{k+1} : k \in \mathbb{N}\}$, $N \in \mathcal{N}$ is the length of the message, and $\mathcal{X}_N = \{0,1\}^N$.

Examples below present some possible use-cases of FPE schemes.

Example 1.1 *Credit card numbers (CCNs) can have from 12 to 19 digits and the number is validated by Luhn checksum. In case of CCN encryption, the format space is $\mathcal{N} = \{12, 13, ..., 19\}$, and for each $N \in \mathcal{N}$, $\mathcal{X}_N$ is a set of CCNs of length N which pass the Luhn test.*

Example 1.2 *Assume in some corporation every employee has an unique email address formatted as **name.surname@company-name.com**. The task is to create an FPE scheme that will perform encryption of employees email addresses within the company. Let A be the set of all names, and B be the set of all surnames. Then $\mathcal{N}$ could be some set containing many different company names, and $\mathcal{X}_N$ would be the set of all possible combinations of names and surnames for the given company name N.*

Remark on informal notation

Sometimes, in this work, we may say that X has a different format than Y. Usually, what we mean is that the *slices* are different, not necessarily the *formats*. For example, let $\mathcal{X}_N$ be the set of numbers having N digits in decimal, and let $\mathcal{Y}_N$ be the set of numbers having N digits in binary. Although in both cases $\mathcal{N} = \mathbb{N}$, and formats N may be the same, the slices are different, meaning that their corresponding FPE schemes are not interchangeable.

1.3 Current standards

There are many FPE schemes but it's rare that one is both provably secure and efficient. The problem is in a domain, which is usually non-binary and can be small. As a result, NIST decided to standardize two schemes for FPE that have been only validated through cryptanalysis - FF1 and FF3 [8]. In [2] a few message recovery attacks on Feistel-based FPE protocols have been shown, and in [12] the results have been further improved. These attacks are not specific only for FF1 or FF3, but for all FPE schemes based on Feistel networks and it is a reason why schemes based on other mechanisms meet with growing interest.

1.4 Limitations

As everything, FPE comes along with a set of limitations. Some of them are only practical and may be mitigated by further improvements in the field, but others are inherent to FPE due to the generality it provides.

1.4.1 Restrictions on the format space

Not every set can be a proper format. The encryption needs an efficient way to check if given X is in $\mathcal{X}_N$. For example, if each slice $\mathcal{X}_N$ represents the set of natural numbers that have exactly N distinct prime divisors, then deciding if a particular X is in $\mathcal{X}_N$ may be, for $N > 2$, as hard as factorizing X itself. Another, more theoretical example, may be $\mathcal{X}_N$ representing the set of all inputs of length N, for which a given Turing machine T returns some desired value. It may appear a little forced, but consider a situation where one attempts to encrypt an executable program into a different one, with the same output.

1.4.2 Variety of formats

In many practical applications, the data we attempt to encrypt may consist of multiple different formats. For example, during encryption of databases storing user information, a row could contain a name, surname, home address, phone number, email address, or CCN. It is unlikely that a single FPE scheme will be sufficient for the encryption of all these pieces of data. If we tailored our FPE schemes for each particular format, then we would need to develop multiple distinct schemes, one for each cell. Moreover, all of these schemes should be proven secure and implemented separately, which drastically increases the effort of using such a solution.

Instead, we may try a more general approach, which allows us to reuse a single FPE scheme for multiple domains, with the transition effort minimized. One such method is called *Rank-then-Encipher (RtE)*. It uses an FPE on $\mathcal{X}$ and creates a mapping between $\mathcal{X}$ and some other set $\mathcal{U}$, for instance, a set of words matched by a given regular expression. Then the problem described above reduces to adjusting mappings.

Another aspect of this problem is that the domains in FPE may have various sizes and, usually, the security of an encryption scheme is a function of the size of the domain it operates on. Therefore, a good general FPE scheme should be both secure for small domains and efficient for the bigger ones.

1.4.3 Risk of domain changes

In example 1.2, we create two distinct sets N, S of names and surnames, respectively, and perform encryption of employees' email addresses. However, if there appears an employee whose name e_N is not in N, his data cannot be encrypted. It requires an expansion of the initial $\mathcal{X} = N \times S$. If a new encryption scheme is devised for the expanded slice $\mathcal{X}' = (N \cup \{e_N\}) \times S$, it usually is incompatible with previously enciphered points, i.e. it would need re-encrypting all of already stored data with the new expanded scheme.

The task of *domain extension* which constructs an FPE scheme on an extended domain while preserving old mappings, was studied in [10] by Grubbs and Ristenpart, and resulted in devising *Zig-Zag* algorithm. Later, Miracle and Yilek developed another method, called *Cycle Slicer*, and presented it in [19].

1.5 Rank-then-Encipher

A simple general method for encrypting a finite set A is to define a total ordering on its elements and to perform encryption on their corresponding indexes instead. Let slice $\mathcal{X}_N$ be a finite set that can be ordered as $\mathcal{X}_N = \{X_0, X_1, ..., X_{n-1}\}$, where $n = |\mathcal{X}_N|$. The enciphering process of any X_i consists of three steps. First, a *Rank* function is used to get i from X_i. Next, this index is passed to the internal FPE scheme E over $\{0, 1, ..., n-1\}$ to obtain $j = E(i)$. Finally, the resulting X_j is retrieved by using the $Unrank$ function on j. This approach is called *Rank-then-Encipher* [3].

1.5.1 Formal definition

Definition 1.3 *Let $\mathcal{E}$ be an FPE scheme with $\mathcal{K}$ as a key space, $\mathcal{N}$ as a format space, $\mathcal{T}$ as a tweak space, and $\mathcal{X}$ as a domain. Let $X \in \mathcal{X}$, $K \in \mathcal{K}$, $T \in \mathcal{T}$, $N \in \mathcal{N}$, and $E_K^{N,T}$ be the corresponding encryption function. Let U be a finite set and assume there exists a bijection $f : U \to X$. Then, the RtE encryption function E_{RtE} on $E_K^{N,T}$ is defined as follows:*

$$E_{RtE}(u) = f^{-1}(E_K^{N,T}(f(u)))$$

if $u \in U$, and $\perp$ otherwise.

The *Rank* and *Unrank* functions are defined by f, and f^{-1}, respectively.

1.5.2 Examples of usage

One advantage of *RtE* is that it makes it possible to reuse a single FPE-scheme on any $\mathcal{X}_N$, and encrypt every finite set that can be efficiently mapped into it. The main asset of this is that the security should rely on the internal FPE-scheme, and usually the particular ranking/unranking functions do not matter in this regard. As a result, the process of creating encryption schemes for new domains becomes safer and easier.

FPE over regular languages

Regular languages can define many formats like, for instance, email addresses, and therefore an efficient scheme for encrypting them would be useful. As it is shown in chapter 3, there exist ranking/unranking functions for DFAs, and consequently the *RtE* approach is feasible also for regular languages.

Definition 1.4 *Let Σ be an alphabet and $L \subset \Sigma^*$ be a language over it. An FPE scheme $E : \mathcal{K} \times \mathcal{N} \times \mathcal{T} \times \mathcal{X} \to \mathcal{X} \cup \{\perp\}$ is an FPE scheme on L if $\mathcal{X} = L$, $\mathcal{N} = \mathbb{N}$ and the slices are $\mathcal{X}_N = L \cap \Sigma^*$. [3]*

There are many approaches for performing *RtE* on a regular expression r. The simplest way is to enumerate all words matched by r in their lexicographical order, and then rank v to its corresponding index. However, this solution requires space linear in the number of words matched by r, and in practice that may quickly consume all available resources.

Example 1.3 *Let L be a language of arbitrary strings consisting of letters a, b, or c. Then $|\mathcal{X}_N| = 3^N$. For $N = 20$, $|\mathcal{X}_N| > 10^9$, and it grows exponentially with respect to N.*

An on-line method, being able to rank an element without explicitly listing all the previous ones would be much better, and this work contains an implementation of one such solution, based on Brzozowski's algorithm. The idea behind it is to create an efficiently rankable finite automaton that is equivalent to the given regular expression. Then it is used to perform ranking and unranking of words $X \in L$ such that $|X| = k$ for some constant k. [3][5][20]

Using DFAs as the ranking "engines" has its own advantages. Sometimes it is easier to represent a slice $\mathcal{X}_N$ directly as a DFA. This happens, for example, for CCNs, as there exists a 20-state finite automaton that recognizes the language $Luhn^R$. It is defined as $M = (Q, \Sigma, \delta, q_0, F)$ with states $Q = \mathbb{Z}_{10} \times \mathbb{Z}_2$, final states $F = \{0\} \times \mathbb{Z}_2$, initial state $q_0 = (0, 0)$, and transition rule $\delta((a, b), d) = (a + 2d + a\lceil d/5 \rceil \mod 10, 1 - b)$. [3]

However, finding a DFA for a given regular expression is, in general, a difficult problem, and therefore it is broadly described in chapter 3. The forthcoming solution yields results that are satisfactory in most situations, as the generated DFAs are usually minimal, and the construction time is negligible.

FPE over context-free grammars

A natural thing to consider after having an efficient *RtE* scheme for regular languages is its analog for *context-free grammars*.

In [14] it is shown that a universally efficient method of ranking context-free languages[1] implies $P = NP$. Fortunately, most typical applications do not require such a strong general method. Usually, one is interested in *unambiguous context-free languages* or languages for which exists an unambiguous context-free grammar. It is known [9] that there is a polynomial-time ranking algorithm for languages given by such grammars.

Later, in [17] Mäkinen showed that if $O(n^2)$-time and $O(n^2)$-space preprocessing phase is allowed, then ranking of left Szilard languages on context free grammars can be performed in linear time, and unranking in $O(n \log n)$. As he stated in the paper, this implies similar algorithms for CFLs generated by arbitrary unambiguous context-free grammars.

1.6 Security notions

As for classical encryption schemes, the notions used for FPE are based on bounding the advantage that can be obtained by a PPT adversary[2] $\mathcal{A}$. Usually, when some E is called indistinguishable from R, it means that for any $\mathcal{A}$ the probability that it distinguishes E from R is negligible. Authors of [3] define multiple notions of security for FPE schemes, and also adapt to FPE some of those used for conventional encryption schemes. A few of them, ordered from the strongest to the weakest, are listed below:

Let E be an FPE scheme on $\mathcal{X}$ and $\mathcal{A}$ be any PPT adversary.

[1] hereinafter CFLs

[2] *Probabilistic Polynomial Time* adversary - an algorithm that can run only in polynomial time and has access to a source of randomness.

1. *Pseudo-Random Permutation* - E is PRP-secure if it indistinguishable from a randomly selected permutation π on $\mathcal{X}$. PRP is a strong notion as it implies all of the following ones, and furthermore, in many cases, an attack against the PRP notion poses no practical threat.

2. *Single-Point Indistinguishability* - E is SPI-secure if the encryption of a single point is indistinguishable from choosing a random one in the range. This applies even when $\mathcal{A}$ has access to a true encryption oracle. SPI is a variation of PRP, and they are equivalent, meaning that SPI-security implies PRP-security and vice versa. In some situations it is easier to prove one of them through the other; however, the bounds obtained in this manner may be weaker than a direct proof's results.

3. *Message Recovery* - a deterministic encryption scheme E is MR-secure if $\mathcal{A}$ is unable to recover the plaintext X^* from a ciphertext Y^* even with access to an encryption oracle and when plaintexts, formats and tweaks are from a favorable distribution. For such E, the best message recovery attack is to encrypt various $X_1, X_2, ..., X_q$ to $Y_1, Y_2, ..., Y_q$ until some Y_i is equal to Y^*. E is a deterministic encryption scheme, hence X^* must be equal to X_i.

GraFPE

GraFPE[1] is an FPE scheme that allows encryption of arbitrary sets formed as $\mathbb{Z}_N = \{0, 1, 2, ..., N-1\}$. In contrast to many popular FPE schemes, it is not based on Feistel networks, but instead, it builds upon random walks over regular graphs and sponge construction.

We can divide the protocol into two separate parts: the *build* phase, and *encryption* phase. The first is an algorithm that, given a key and IV, generates a pseudo-random regular graph G. Then, during the second phase, the encryption is done by performing a random walk over vertices of G.

With respect to the notation from chapter 1, GraFPE can be thought of as an FPE scheme $E : \mathcal{K} \times \mathcal{N} \times \mathcal{T} \times \mathcal{X} \to \mathcal{X} \cup \{\perp\}$, where $\mathcal{N} = \mathbb{N}$, and $\mathcal{X}_N = \mathbb{Z}_N^t$ represents the set of all sequences of length t with elements from $\mathbb{Z}_N$. $\mathcal{K}$, as well as $\mathcal{T}$, rely on the particular implementation.

GraFPE's primary advantage is that it is provably *PRP*-secure and resistant for message recovery attacks.

2.1 Build phase

To encrypt numbers from $\mathbb{Z}_N = \{0, 1, ..., N-1\}$ GraFPE needs to construct an internal graph $G = (V, E)$ of precisely N vertices. The authors of [15] describe two approaches for this purpose. The first of them is called the *Configuration model*, but it has not been implemented in this work due to its long running time and technical difficulties that it arises.

The another algorithm is the *Permutation model*, and it consists of two steps. At first, it creates a **valid** set of $d/2$ permutations. Then it transforms them into a d-regular[2] graph.

2.1.1 Building graph from permutations

Definition 2.1 *The set of permutations $\mathcal{P}_m = \{P_1, P_2, ..., P_m\}$ is **valid** if it meets the following conditions: For all $i \in \{1, 2, ..., m\}$*

1. *if $P_i(x) = y$ then $\forall_j P_j(y) \neq x$ (no loops or cycles of length 2)*
2. *if $P_i(x) = y$ then $\forall_j P_j(x) \neq y$ (no multiple edges)*

The reason for disallowing loops and multiple edges is to make the resulting graph G simple. The desired regularity does not, by itself, impose any restrictions on the permutations. Cycles of length 2 are forbidden to improve the chance of G being an *expander*.

Definition 2.2 *Given a valid set $\mathcal{P}_{d/2}$ of permutations on $\{0, 1, ..., n-1\}$, the undirected d-regular graph $\gamma(\mathcal{P}_{d/2}) = (V,E)$ is defined as follows.*

- $V = \{0, 1, ..., n-1\}$
- $E = \{(x, y) : x, y \in V \wedge (P_i(x) = y \vee P_i(y) = x)$ *for some* $i \in \{1, 2, ..., d/2\}\}$.

The algorithms for checking if the given set of permutations is valid, and for building the corresponding graph, are both presented in chapter 4. An example of a valid set of permutations along with its corresponding graph is shown on Figure 2.1

[1] Most of the definitions and algorithms in this chapter are based on [15].
[2] i.e. for any vertex v of the graph, its degree $deg(v) = d$.

$$P_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 7 & 0 & 2 & 1 & 6 & 4 & 5 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 6 & 4 & 5 & 7 & 2 & 0 & 3 \end{pmatrix}$$

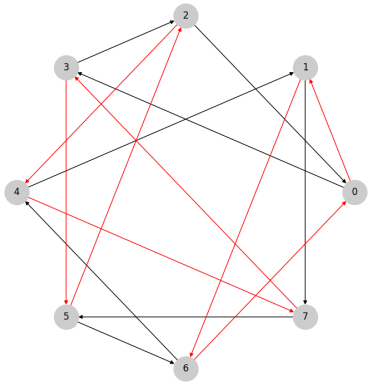

Figure 2.1: A valid set $\mathcal{P}_2 = \{P_1, P_2\}$ of permutations on $\{0, 1, ..., 7\}$, and their corresponding graph $\gamma(\{P_1, P_2\})$. For construction clarity the depicted graph is directed, but in general it is not. The black arrows represent edges constructed from P_1, and the red ones, from P_2.

Lemma 2.1 *The undirected graph G constructed in a way described in definition 2.2, from a valid set of m permutations, is simple and $2m$-regular.*

Proof *(by induction)*

1. Let $m = 0$, $P_m = \emptyset$, and $G = \gamma(P_m) = (V, \emptyset)$.

   Then, it follows that $\forall_{v \in V}\ deg(v) = 0 = 2m$, and hence G is simple and $2m$-regular.

2. Assume for all $m \leqslant M$ graphs generated from $\mathcal{P}_m$ are simple and $2m$-regular.

   Let G be an undirected graph generated from a valid set of $M+1$ permutations $\mathcal{P}_{M+1} = \{P_1, P_2, ..., P_{M+1}\}$. Any two vertices u and v of the constructed graph are connected by an edge if, and only if $P_i(u) = v$ or $P_i(v) = u$ for some $i \in [M + 1]$.

   A permutation, considered as a function, is a bijection. This means it "affects" every $v \in V$ twice - once as an argument, and once as a value. Since P_{M+1} is a *valid* permutation, it cannot add any loop to the graph, because this would imply $P_{M+1}(x) = x$ for some x. Therefore, every valid permutation increases the degree of each vertex v in V by 2.

   If we ignore P_{M+1} during the construction process, we obtain a graph $G' = \gamma(\mathcal{P}_M)$, which is, by the inductive foundation, simple and $2M$-regular. Adding the edges appropriate for P_{M+1} back, increases the degree of every vertex by 2, and results in a $2M + 2 = 2(M + 1)$-regular graph.

   If G contained any multiple edges, this would contradict the assumption that $\mathcal{P}_{M+1}$ is a valid set of permutations; therefore G is simple.

   ♦

In practice, the generated graph is not stored explicitly in memory. It remains represented by its corresponding set of permutations $\mathcal{P}_{d/2}$. For efficiency improvement, a set of inverse permutations $\mathcal{P}^{-1}_{d/2} = \{P_1^{-1}, P_2^{-1}, ..., P_{d/2}^{-1}\}$ is computed and stored alongside $\mathcal{P}_{d/2}$.

2.1.2 Inverse Riffle Shuffle

As stated before, to create a d-regular graph, a set of $d/2$ pseudo-random permutations is required. Any method of obtaining them would be sufficient, but in [15] the *Inverse Riffle Shuffle (IRS)* algorithm was selected as it is asymptotically the fastest and gives a predictable bound on the number of iterations needed to properly mix the numbers.

Shuffling cards

A *riffle shuffle* is a popular method of shuffling cards in which the deck is divided into two halves. The dealer takes both of them to his hands and interleaves the cards in an unspecified order. The resulting deck is then again split into two, and the process repeats a few times to ensure a proper level of randomness.

The *inverse riffle shuffle* can be seen just like this method, but reversed in time. Each card in the resulting deck is randomly assigned to either left or right half. Then the deck is divided into two, according to the assignment, and finally, one of the halves is placed on top of the another.

Remark In many contexts the algorithm described here as *riffle shuffle* is known as *Faro shuffle*. An important distinction is that in riffle shuffle the halves are combined randomly while Faro shuffle is usually assumed to interlace the cards perfectly.

Generating a permutation using Inverse Riffle Shuffle

As described in [15], to shuffle a vector N of n numbers, an additional vector B is needed. The algorithm can be described as follows:

- For each $n \in N$ flip a coin and store the resulting bit $0 = Heads$, $1 = Tails$ in the corresponding slot in B. After the first iteration, B holds only ones and zeros.
- Take all slots in B that have a zero on the least significant bit, and move them to the top without changing their relative order. Do a copy of this move on N.
- Repeat the steps above. New bits in B should be appended as the new least-significant bits. During the process, numbers stored in B are growing, for example, after the fifth iteration, all slots in B should hold a five-digit binary number.
- When all elements of B are pair-wise distinct, stop.

On average, it is enough to perform $2\log_2|N|$ steps to have the vector N sufficiently shuffled, but the result, mixed this number of times, is only approximately random. Later, we describe how can we use *Inverse Riffle Shuffle* to obtain a perfectly mixed permutation. The following pseudo-code shows how to construct a valid $\mathcal{P}_{d/2}$ using *inverse riffle shuffle*:

Algorithm 2.1: Generate P_{D2}

Input : $d/2$ – desired size of the set of permutations
Output: $\mathcal{P}_{d/2}$ – a valid set of permutations

```
1 P_D2 ← ∅
2 while |P_D2| < d/2 do
3 |   P ← InverseRiffleShuffle()
4 |   if ValidSet(P_D2 ∪ P) then
5 |   |   P_D2 ←∪ P
6 return P_D2
```

The *ValidSet(P)* function checks if P meets the conditions from definition 2.1, and *InverseRiffleShuffle* generates a new pseudo-random permutation. In practice, algorithm 2.1 can take a long time to generate $\mathcal{P}_{d/2}$ even if d is small. With every new permutation added to $\mathcal{P}_{d/2}$, the chance that the next one, generated randomly, will form a valid set with $\mathcal{P}_{d/2}$, drops exponentially.

2.1.3 Practical considerations

Since the graph should always be constructed in a deterministic way, the previously described coin flips cannot be truly random. Instead, GraFPE uses an internal block cipher such as AES as a *Pseudorandom*

Number Generator (PRNG) and treats its outputs as the random bits. Since it is the only role of AES during the *build phase*, the process of embedding it is straightforward. The algorithm takes a single key and IV and passes them directly to AES.

One may consider storing the graph in the memory a security risk, but GraFPE does not require G to be unknown to the adversary for its security; conversely, G may be public. The real problem is the size of the graph when n is large. From a user's standpoint, there is not much difference between the usability of a graph with $n = 10^6$ vertices or $n = 10^9$, but the former requires a few megabytes of memory, while the size of the latter is in order of gigabytes. Additionally, this may introduce a vulnerability against side-channel attacks such as cache attacks or timing attacks.

2.2 Encryption phase

Having a graph $G = (V,E)$ with exactly N vertices constructed, one can identify it with $\mathbb{Z}_N = \{0, 1, ..., N-1\}$ which is the set of the points that GraFPE operates on. The encryption process is done by performing a *Simple Random Walk (SRW)* over V. The input value $v \in V$ is the vertex where the random walk begins, and, in a simplified context, the encrypted value of v the vertex u entered last during the walk. For security reasons, GraFPE introduces further modifications such as the *sponge construction*, which are described later.

2.2.1 Simple Random Walks

A Simple Random Walk of length t on graph $G = (V, E)$ can be defined as a function $W : V \to V$, described by the following algorithm:

Algorithm 2.2: Random walk

Input : $v \in V$ – the starting vertex, t – the walk length, $G = (V, E)$ – a valid d-regular graph
Output: $u \in V$ – the final vertex

```
1 u ← v
2 for i from 1 to t do
3 |   u ← SelectNeighbor(G, u)
4 return u
```

Where the *SelectNeighbor* function chooses randomly, with probability $1/d$ each, one of the neighbors of the vertex u.

Remark GraFPE can be constructed to perform a *Non-backtracking Random Walk (NBRW)* instead of a *SRW*. The difference is that when in *NBRW* the algorithm steps from v to u, then in the next step it cannot step back to v. It results in better mixing times for the encryption phase, and thus achieves the same level of randomness with shorter random walks.

Since GraFPE must be deterministic to make the decryption possible, the neighbor to step into is not selected truly randomly, but pseudo-randomly. Assume that $i \in \mathbb{Z}_N$ is the starting vertex and let $\mathcal{P}_{d/2} = \{P_1, P_2, ..., P_{d/2}\}$. Then, by construction of G, the neighbors of i are $P_1(i), P_2(i), ..., P_{d/2}(i)$. The *SelectNeighbor* function chooses pseudo-randomly a permutation index $t \in \{1, 2, ..., d/2\}$ and returns either $u = P_t(i)$ or $u = P_t^{-1}(i)$, depending on a pseudo-random coin flip.

2.2.2 Sequence inputs

While GraFPE can be used for encrypting messages consisting of a single vertex, it is better suited for enciphering longer sequences. In this case, GraFPE performs a separate *SRW* for each element of the input, so, by construction, the encryption preserves the length of the input sequence. Furthermore, to make itself invulnerable for message-recovery attacks, GraFPE introduces a mutual dependency between these walks by influencing the internal *PRNG*.

Assume the input vector $\mathbf{x} = [x_0, x_1, ..., x_{t-1}]$ is being encrypted to some, currently unknown, $\mathbf{c}$ of equal length. The encryption goes from left to right, so it starts with x_0 and obtains its corresponding c_0. Meanwhile, the remaining points $x_1, ..., x_{t-1}$ form a *digest* which is passed to the *PRNG* as a seed, and hence, make the walk for x_0 dependent of their values. Generally, the digest for any point $x_i \in \mathbf{x}$ is formed as a vector $d = [c_0, ..., c_{i-1}, x_{i+1}, ..., x_{t-1}]$ of length $t-1$. As a result, each walk depends on all elements from $\mathbf{x}$.

The digest is the only thing that influences the choice between permutations during the encryption and decryption. Hence, for a known digest, a t-step encryption function can be seen as

$$E_K^{N,T}(x) = p_t(p_{t-1}(...p_2(p_1(x))...)),$$

where each p_i is either a permutation from $\mathcal{P}_{d/2}$, or its inverse.

The decryption function $D_K^{N,T}$ is precisely a time-reversed copy of the encryption process. Hence, the permutations are swapped with their inverses, and selected in reverse order. For example, if the encryption algorithm chooses P_1, P_5^{-1}, P_3^{-1}, then the decryption selects P_3, P_5, P_1^{-1}. To simplify the notation, a t-step decryption process is represented as

$$D_K^{N,T}(c) = p_1^{-1}(p_2^{-1}(...p_{t-1}^{-1}(p_t^{-1}(c))...)).$$

Let us note that each input may generate a different digest, thereby changing the permutations stored in each p_i. Therefore, to avoid ambiguity, when we use this notation, we will always specify the digest from which come the permutations.

Theorem 2.1 *GraFPE, as described above, is a valid FPE scheme, meaning that for any $K \in \mathcal{K}$, $N \in \mathcal{N}$, $T \in \mathcal{T}$ it meets the following conditions:*

1. $\forall_{x \in \mathcal{X}_N}\ E_K^{N,T}(x) \in \mathcal{X}_N$
2. $\forall_{x \in \mathcal{X}_N}\ D_K^{N,T}(E_K^{N,T}(x)) = x$
3. $\forall_{x,y \in \mathcal{X}_N}\ x \neq y \implies E_K^{N,T}(x) \neq E_K^{N,T}(y)$

Proof

Let $G = (V, E)$ be a graph that GraFPE operates on, and let $N = |V|$.

Ad. 1

Assume $\mathbf{x} = [x_0]$.

The slice $\mathcal{X}_N$ is defined as $\{0, 1, 2, ..., N-1\}$ and identified with the set V of vertices. Since the encryption of $\mathbf{x}$ is performed as a random walk over V, the resulting $E(\mathbf{x})$ must be some $\mathbf{y} \in V$ and hence it preserves the format of $\mathbf{x}$.

If $\mathbf{x} = [x_0, x_1, ..., x_{n-1}]$ for $n > 1$, then the slice is defined as $\mathcal{X}_N = \{0, 1, ..., N-1\}^n$. The resulting $\mathbf{c} = E(x)$ must also be a vector of n elements because each of them is generated by encrypting one component from $\mathbf{x}$. Since each $c_i \in \mathbf{c}$ is also G's vertex, they all preserve formats of their corresponding points in $\mathbf{x}$, and therefore $\mathbf{c}$ also has the same format as $\mathbf{x}$.

□

Ad. 2

Assume $\mathbf{x} = [x_0]$. In this case the digest is empty, so the *PRNG* returns a constant value. Let $c = E_K^{N,T}(x)$. Then, by following the earlier description, we have:

$$D_K^{N,T}(E_K^{N,T}(x)) = p_1^{-1}(...p_{t-1}^{-1}(p_t^{-1}(p_t(...p_2(p_1(x_0)))...))).$$

For all y, $p_t^{-1}(p_t(y)) = y$, and hence these permutations cancel out. Next, p_{t-1}^{-1} cancels out with p_{t-1}, and this repeats for all the subsequent pairs of permutations, until finally

$$D_K^{N,T}(E_K^{N,T}(x)) = p_1^{-1}(p_1(x_0)) = x_0.$$

To prove for $\mathbf{x} = [x_0, ..., x_{n-1}]$ it is enough to show that if we reverse the sequence of digests generated during decryption, then it is the same as the sequence generated during encryption. Then the result follows from the paragraph above.

Let $\mathbf{c} = [c_0, ..., c_{n-1}] = E_K^{N,T}(\mathbf{x})$. The decryption process is performed from right to left, so the first decrypted point is c_{n-1}. The digest used is $[c_0, ..., c_{n-2}]$, which is the same as during enciphering x_{n-1}. From this it can be deduced that c_{n-1} will be decrypted back to x_{n-1}.

The next decrypted point, c_{n-2} will use $[c_0, ..., c_{n-3}, x_{n-1}]$ as digest, which is again the same as during encrypting x_{n-2}.

The same argument applies to the remaining points $c_0, ..., c_{n-3}$, from which it follows that:

$$D_K^{N,T}(E_K^{N,T}(\mathbf{x})) = \mathbf{x}$$

□

Ad. 3

This follows directly from 2. If there were any two points $\mathbf{x}, \mathbf{x}', \mathbf{x} \neq \mathbf{x}'$, such that $E_K^{N,T}(\mathbf{x}) = E_K^{N,T}(\mathbf{x}') = \mathbf{c}$ then either $D_K^{N,T}(\mathbf{c}) = \mathbf{x}$ which contradicts 2 when considering $\mathbf{x}'$ or vice versa.

Hence $E_K^{N,T}(\mathbf{x}) \neq E_K^{N,T}(\mathbf{x}')$

□

♦

2.2.3 Sponge construction

To further increase security, GraFPE employs a *sponge construction*. This is a common technique that divides a process into an *absorbing phase* and *squeezing phase*. In GraFPE, the result of the pseudo-random walk is not yet the ciphertext, but it forms the *absorbed* layer *(absorbing phase)*. After the first series of random walks, the *absorbed* data is treated as the new plaintext, and the whole process repeats *(squeezing phase)*. The final results are treated as the ciphertext.

Lemma 2.2 *The sponge construction does not affect the validity of the proof of Theorem 2.1.*

To see why this is true, notice that the sponge construction is basically the encryption algorithm repeated a second time.

2.2.4 Practical considerations

The *encryption* algorithm selects the permutation pseudo-randomly by fetching an index of one of them from the internal *PRNG*. However, since the *PRNG* returns nothing but binary strings, this can be easily done only for d being a power of 2. To select a permutation we generate a vector of $f = \log_2 d$ bits. The least significant bit marks if eventually we take the inverse permutation, and the remaining $f - 1$ bits compose a number $i \in \{0, 1, ..., 2^{f-1} - 1\} = \{0, 1, ..., \frac{d}{2} - 1\}$, which is the index of $P_i \in \mathcal{P}_{d/2}$.

The *PRNG* is constructed by a block cipher such as AES, and thus, it is better suited for generating pseudo-random vectors instead of single numbers. GraFPE takes an advantage of that by upfront generating all the randomness needed for a single *SRW* before starting the walk itself. Let l denote the length of the walk. Then, since each step of *SRW* needs exactly f bits of randomness, the total size of the vector of bits $\mathbb{B}$ is lf.

It is not necessarily true for a *NBRW* if G is undirected. In situation when bits stored in $\mathbb{B}$ result in backtracking, *NBRW* needs to take the decision using some additional information. The problem does not occur for a directed graph, since, by construction, every random walk on it is a *NBRW*.

Algorithm 2.3 presents a simplified approach to encryption using GraFPE; practical solutions may also pass i or additional tags to the *RandomBuffer* function to make $\mathbb{B}$ differ even for the same digests.

Algorithm 2.3: GraFPE Encrypt

Input : $\mathbf{x} = \{x_0, x_1, ..., x_{n-1}\}$
Output: $\mathbf{c} = \{c_0, c_1, ..., c_{n-1}\} = E_K^{N,T}(\mathbf{x})$

$a \leftarrow \{0, 0, ..., 0\}$ – n elements
$c \leftarrow \{0, 0, ..., 0\}$ – n elements
for i *from* 0 *to* $n-1$ **do**
    $d =$ **GetDigest(x, a, i)**
    $\mathbb{B} =$ **RandomBuffer(**fl**,** d**)**
    $a_i =$ **RandomWalk(**x_i**,** $\mathbb{B}$**)**
for i *from* 0 *to* $n-1$ **do**
    $d =$ **GetDigest(a, c, i)**
    $\mathbb{B} =$ **RandomBuffer(**fl**,** d**)**
    $c_i =$ **RandomWalk(**a_i**,** $\mathbb{B}$**)**
return c

2.3 Mixing times

Both of GraFPE's phases are random processes with input parameters specifying the number of iterations they should perform. Finding the optimal number of rounds is a crucial task since if it is too small, then the scheme will be insecure; conversely, any excess iteration can result in a drastic increase of the processing time.

GraFPE uses results from [1] to bound the mixing time in the *Permutation Model*, and builds upon [16] to do the same for the encryption process. Authors of [15] utilize a notion of *cutoffs*, which can be informally defined as the moment when the *total variation distance* between a Markov chain $\mathcal{M}$ and its stationary distribution drops abruptly from almost 1 to nearly 0.

2.3.1 Build phase

In [1] Aldous and Diaconis showed that shuffling n cards can be seen as a random walk over elements of S_n. To bound the number of shuffles needed to mix the cards properly they use *strong uniform times* and show that when one occurs, there is no reason to continue shuffling. The following definitions and a lemma come from [1] and formalize what a *strong uniform time* is and why is it useful.

Definition 2.3 *Let G be a finite group, and define G^∞ as the set of all sequences of elements from G. Let $\mathbf{g} = (g_1, g_2, ...) \in G^\infty$. A function $\hat{T} : G^\infty \to \mathbb{N}$ is called a stopping rule if $\hat{T}(g) = j$ implies that $\hat{T}(g') = j$ for all g' such that $g'_i = g_i$ for $i \leqslant j$.*

In other words, $\hat{\mathrm{T}}$ is a stopping time if its value is defined only by these events which happen before or exactly at $\hat{\mathrm{T}}$.

Definition 2.4 *Let Q be a distribution on G, and (X_k) be a random walk associated with it. T is called a stopping time if $T = \hat{T}(X_1, X_2, ...)$, where $\hat{T}$ is a stopping rule. T is a strong uniform time, if for each $k \in \mathbb{N}$*

$$P(X_k = g | T \leqslant k) = 1/|G|$$

for all $g \in G$.

The reason why strong uniform times are particularly useful is stated by the following lemma from [1].

Lemma 2.3 *Let Q be a probability distribution on a finite group G, and let μ be the uniform distribution. Then*

$$d_{TV}(Q^{k^*}, \mu) = ||Q^{k^*} - \mu|| \leqslant P(T \geqslant k),$$

where T is a strong uniform time.

Now, to analyze the number of rounds needed by *Inverse Riffle Shuffle* to properly shuffle n cards it is enough to find its corresponding *strong uniform time*. Let T be the first time when all bit slots as described in section 2.1.2 are distinct. According to [1], T is a *strong uniform time*, and $2\log_2 n$ shuffles are needed for T to occur.

2.3.2 Encryption phase

As shown in [11], a *simple random walk (SRW)* on a graph G yields a reversible Markov chain. For such chain $\mathcal{M}$, there exists a reversible distribution equivalent to $\mathcal{M}$'s equilibrium, and it is defined as

$$\pi = \left(\frac{d_1}{d}, \frac{d_2}{d}, ..., \frac{d_n}{d}\right)$$

where $d = \sum_{i=1}^{n} d_i$, and d_i is the degree of the $i-$th vertex.

If G is regular, then π is a uniform distribution, and therefore the results obtained by such infinitely-long *SRW* can be considered random. For this reason GraFPE requires the internal graph to be regular.

However, to yield a ciphertext, GraFPE must perform only a finite number of steps, and therefore the results are only approximately random. However, as shown in [16], *SRWs* over regular graphs are processes for which the *cutoff* occurs at an easy-to-compute time. The bound is presented in the following theorem from [16]:

Theorem 2.2 ***(from [16])***
Let $G \sim \mathcal{G}(n,d)$ be a random $d-$regular graph for $d \geqslant 3$. Then, with high probability, the SRW on G exhibits cutoff at

$$t_c = \frac{d}{d-2}\log_{d-1} n,$$

with a window of order $\sqrt{\log n}$. Furthermore, for any fixed $0 < s < 1$, the worst case total-variation mixing time, with high probability, is

$$t_{\text{MIX}}(s) = \frac{d}{d-2}\log_{d-1}(n) - (\Lambda + o(1))\Phi^{-1}(s)\sqrt{\log_{d-1}(n)},$$

where $\Lambda = \frac{2\sqrt{d(d-1)}}{(d-2)^{3/2}}$ and Φ is the CDF[3] of standard normal.

Example 2.1 *Let $G = (V,E) = \gamma(\mathcal{P}_{d/2})$ be a $d = 8$-regular graph with $|V| = 10^6$ vertices. Then, the cutoff occurs after $t_c \approx 9.466$ steps, and if $d_2 = 4$, then $t_c \approx 25.151$. This shows a complexity trade-off between the build and encryption phases.*

The theorem above not only allows to approximate when the cutoff occurs, but also gives means to bound the total variation distance between $\mathcal{M}$ and the uniform distribution μ. Let $\mathcal{L}_i$ be the distribution of $\mathcal{M}$ after i steps and let $i = \frac{d}{d-2}\log_{d-1}(n) + \gamma(s)\sqrt{\log_{d-1}(n)}$, where $\gamma(s) = (\Lambda + o(1))\Phi^{-1}(s)$. According to [15], there is

$$d_{TV}(\mathcal{L}_i, \mu) \leqslant s.$$

Remark Formally, Theorem 2.2 does not apply directly to graphs constructed during the build phase, because they are not necessarily uniformly distributed. However, authors of [15] prove that *Theorem 1* indeed holds for graphs generated by GraFPE as well.

By definition, *SRW* assumes that each neighbor has an equal probability of being chosen and that the starting vertex v cannot choose itself. Having G simple makes the construction of *SRW* in GraFPE as easy as just selecting any vertex that has an adjacent edge with v. Since there are no loops in G, v must step into $u \neq v$, and because there are no multiple edges each such u is equally likely.

[3] Cumulative Distribution Function

2.4 Examples of applications

GraFPE, without any extensions, can still be used in many contexts.

Example 2.2 *GraFPE can be used for encryption of employees email addresses from example 1.2. The set of all names is finite and relatively small so that it can be easily indexed. The same applies to the surnames. To perform encryption of an email address* $e =$ ***name.surname@company.com****, two graphs* $G_{Name}, G_{Surname}$ *would be created. The resulting ciphertext would consist of separately encrypted name, and surname.*

Example 2.3 ***Encryption of numbers of arbitrary size.***

Each natural number n *can be treated as a sequence of its digits, and since GraFPE is better suited for encryption of whole vectors instead of single points, this approach also carries a security improvement.*

Let $n \in \mathbb{N}$ *be represented as a number in the base-*b *positional system. In this case* $n = a_t b^t + a_{t-1} b^{t-1} + ... + a_1 b + a_0$*, and it can be treated as a vector* $n = [a_t, a_{t-1}, ..., a_1, a_0]$*, where each* $a_i \in \{0, 1, ..., b-1\}$*.*

Therefore, to encrypt such n*, one can use GraFPE with the internal graph* $G = (\{0, 1, ..., b-1\}, E)$*. The resulting* $\mathbf{c} = E(n)$ *will also be a vector of* $t+1$ *elements with each* $c_i \in \{0, 1, ..., b-1\}$*, and represent a* $(t+1)$*-digit number in base* b*.*

Extension protocols

GraFPE by itself only allows encryption in $\mathbb{Z}_N$, which is insufficient in many situations where the data format is under additional restrictions. In case of CCNs, the slices consist only of $n-$digit numbers which pass the Luhn checksum test. Other formats may be defined, for example, by a regular expression.

This chapter covers a few algorithms for constructing new schemes relying on GraFPE, which can operate on various domains. The topics include *domain targeting*, *domain extension*, and methods for building *RtE* on arbitrary regular languages. All of the presented solutions are parts of the resulting implementation.

3.1 Domain targeting

This technique uses an encryption scheme E_T on the set T, to build a secure encryption scheme E_S on some set S, where $S \subset T$. In the following section three different protocols for this purpose are presented, each of which has its upsides and downsides. The decision on which of them to use depends mainly on the properties of T and S.

These methods can be used as an alternative to *Rank-then-Encipher*. For example, encrypting numbers that pass *Luhn* checksum test can be done both by *RtE* using DFA and by building a "targeted" encryption scheme with the domain being the set of these numbers that pass the test.

An important advantage of *domain targeting* over *RtE* is that it does not require any efficient way of ordering the elements, nor any order on them at all. It only needs an indicator function $\mathcal{I}(x)$ that tests membership of x in the targeted domain S. [19]

3.1.1 Cycle Walker

This technique was originally described in [4] by Black and Rogaway. In this method, they attempt to use an existing encryption scheme E_T on some set $T = 2^n$ to encipher points from an arbitrary domain $S \subset T$.

The algorithm itself is simple. Given $s \in S$ perform encryption using E_T. If the resulting $E_T(s) = s' \notin S$, encipher it again to obtain $s'' = E_T(s')$. Repeat until the first $s^{(n)} \in S$ is found.

Algorithm 3.1: Cycle Walker

Input : $T, S, E_T,\ s \in S$
Output: $c = E_S(s)$

```
1 c ← E_T(s)
2 while c ∉ S do
3 |   c ← E_T(c)
4 return c
```

If E_T is some permutation π over T, then its corresponding *Cycle Walker* is denoted as CW_π.

Under assumption that $s \in S$, it is always guaranteed that *Cycle Walker* performs a finite number of steps to find $c = E_S(s)$. Let π be a permutation on T, created by repeatedly applying E_T. CW_π can be seen as walking over cycles in $\pi's$ cycle decomposition. Since $s \in S$, CW_π can only traverse these cycles that have at least one element in S, namely s itself. CW_π stops when it finds the first element that is in S. If s is the only such element, then $s = E_S(s)$.

According to Theorem 1 from [4], if E_T is an ideal cipher on T, then E_S constructed using *Cycle Walker* is a uniform random permutation on S.

This approach has a few drawbacks, unfortunately. First, it only performs well if S is relatively big when compared with T. If $2|S| \geqslant |T|$ then the expected number of applications of E_T is at most 2. However, in cases of small subsets of T, the number of iterations can be much bigger. The worst-case running time of CW_π is $\Theta(|T|)$.

Secondly, the encryption time depends on the particular $s \in S$. Authors of [19] point out this can leak subtle timing information. On the contrary, [3] shows in *Theorem 4* that if E_T is a good *PRP*, then information about timing leaked during CW_π gives no advantage to the adversary. Later, a practical example of risk induced by leaking the timing information was shown in [18].

3.1.2 Reverse Cycle Walker

Miracle and Yilek introduced *Reverse Cycle Walker* in [18] to address the unpredictability of Cycle Walker running-times. The main idea is to modify CW_π in such a way that it is possible to "cut" the algorithm if it takes too many steps.

It is not possible to do this in plain CW_π since the algorithm loops only when it is outside the subdomain S. Cutting the execution in such a situation would not result in a permutation over S.

The idea behind RCW is easily explained with an example.

Example 3.1 *Consider permutation π over domain T with the following cycle structure*

$$\pi = (t_1\ s_1\ s_2\ s_3\ t_2\ s_4\ s_5\ t_3\ t_4)$$

As in CW_π, RCW will map s_1 to s_2, and s_2 to s_3. The difference occurs when enciphering s_3. In CW_π it would be mapped to s_4, however, here the algorithm goes to the left as long as it remains in the subdomain S. In this case, the last entered point would be s_1. In the same manner, s_4 is mapped to s_5 and s_5 back to s_4.

RCW explained as above has the same disadvantages as CW. The worst-case running time is still $\Theta(|T|)$, and occurs for the elements in "far-right" of the cycle, where $|S| \approx |T|$.

Cutting execution time

In contrast to Cycle Walker, the execution of RCW can be stopped after a specified number of steps to the left was taken. Specializations of RCW that cut the execution after n applications of π^{-1} are called *Reverse n-Cycle Walkers*, denoted as R_nCW_π.

Let $(t_i\ s_j\ s_{j+1}\ ...\ s_{j+l}\ t_{i+1})$ be a sequence that appears in some cycle in $\pi's$ cycle decomposition. If $l \leqslant n$, R_nCW_π performs as usual. On the other hand, if $l > n$, then every point $s_j, ..., s_{j+l}$ is mapped to itself.

Miracle and Yilek pay particular attention to R_2CW_π. This results in an even simpler algorithm: every sequence of more than two consecutive points from S is mapped to itself. Sequences of length 2 have their elements swapped.

To perform security analysis, an additional pseudo-random function $B : S \to \{0, 1\}$ is needed. For each pair of points that are candidates for being swapped, a fair coin is flipped. The swap occurs if, and only if B returns 1 for the input value.

R_2CW_π is also an involution, meaning that decryption can be performed without any modification to the algorithm.

Multi-round RCW

Running R_2CW_π only once does not yield a random permutation over S. Many sequences would be mapped to themselves, and these of length 2 have only 50% chance to be swapped [18]. For this reason, a multi-round R_2CW is used. It is denoted as $R_2CW^k_{(\pi_1,\pi_2,...,\pi_k),(B_1,B_2,...,B_k)}$, or R_2CW^k when the permutations π_i and functions B_i are clear from context.

It is assumed that particular π_i and B_i are all independent, and, to enable decryption, should be generated pseudo-randomly in each round.

Algorithm 3.2: Reverse 2-Cycle Walker

Input : $S, \pi, s \in S$
Output: $c = R_2CW_\pi(s)$

```
y ← π(s); z ← π^{-1}(s)
if y ∈ S ∧ z ∉ S ∧ π(y) ∉ S then
    b ← B(s)
    if b = 1 then
        return y
    else
        return s
else if y ∉ S ∧ z ∈ S ∧ π^{-1}(z) ∉ S then
    b ← B(s)
    if b = 1 then
        return z
    else
        return s
else
    return s
```

A single run of R_2CW is shown on algorithm 3.2. The algorithm checks if the input value s is an element of a pair $(s_1, s_2) \in S^2$. It also verifies that (s_1, s_2) is surrounded by points from $T-S$, meaning that $\pi^{-1}(s_1) \notin S$, and $\pi(s_2) \notin S$. If all conditions are met, and $B(s) = 1$, then the two points are swapped.

If s is not in a pair, or $B(s) = 0$, no action is performed.

Bounding mixing time

The main question of R_2CW_π is how many rounds are required to have a satisfying level of randomness. The notion used here - *total variation distance*, was already utilized before. Denote $v_{R_2CW^r}$ as the distribution after performing r rounds of R_2CW_π.

To bound the mixing time Miracle and Yilek use a technique called *delayed path coupling* introduced by Czumaj and Kutyłowski in [6] to analyze matching exchange protocols. The proofs of the bounds are technical, and therefore will be omitted. The main corollary from [18] is presented in conclusion 3.1.

Conclusion 3.1 ***(excerpt from [18])***

Let $N_S = |S|$, $N_T = |T|$, $c = N_T/N_S$, and let

$$T = \max\left(40\ln(2N_S^2), \frac{10\ln(N_S/9)}{\ln(1+0.3(c-1)^4/c^6)}\right) + \frac{36c^3\ln N_S^2}{(c-1)^2}$$

Then, under assertion that $N_S \geqslant 2^{10}$,

$$||v_{R_2CW^r} - \mu_s|| \leqslant N_S^{1-2r/T}$$

Unfortunately, these bounds as claimed in 3.1 are unsatisfactory for practical purposes. Results presented in Table 3.1 show that the number R of rounds required to set 0.1 as the upper bound of d_{TV} grows drastically when $|S|$ gets close to $|N|$.

The corollary applies only to R_2CW^k, its analog for reverse n-cycle walking remains an open problem in [18].

3.1.3 Cycle Slicer

A year after presenting *RCW*, Miracle and Yilek came up with another solution for *domain targeting* and presented it in [19] as *Cycle Slicer*. It serves as a further improvement of *RCW* and its main target is

N_S	N_T	$1/c$	$\mathcal{R}$
2^{10}	2^{11}	0.5	9,536
$2^{10}+2^{9}$	2^{11}	0.75	58,991
$2^{10}+2^{9}+2^{8}$	2^{11}	0.875	644,114
$2^{10}+2^{9}+2^{8}+2^{7}$	2^{11}	0.9375	87,960,027
2^{127}	2^{128}	0.5	128,088
$2^{127}+2^{126}$	2^{128}	0.75	741,344
$2^{127}+2^{126}+2^{125}$	2^{128}	0.875	8,143,824
$2^{127}+2^{126}+2^{125}+2^{124}$	2^{128}	0.9375	111,138,620

Table 3.1: Smallest number of rounds $\mathcal{R}$ performed by $R_2CW_\pi^R$ such that $N_S^{1-2\mathcal{R}/T} \leqslant 0.1$.

to reduce the number of rounds required to mix the subdomain S properly. The authors claim that when $|S| \approx |T|$ then the improvement is of a few orders of magnitude.

The new solution works by transforming the original permutation π on T into a matching. The decision about which points are included in it is taken by the *inclusion function* I. By manipulating I, CS can be used not only for *domain targeting* but also for *domain completion* and *domain extension*.

Generally, the inclusion function is defined as $I : T \times T \to \{0,1\}$, and to achieve domain targeting from T to S, $I(x,y)$ should return 1 if, and only if $x \in S \wedge y \in S$.

Assume E_T makes up a permutation π on T. As the name suggests, *Cycle Slicer* aims to "slice" the cycles of π's cycle decomposition into transpositions, also called swaps. To do this, CS defines a *direction* function $Dir : T \to \{0,1\}$. If for some $t \in T$ there is $Dir(t) = 1$, then t is called *forward-looking*, otherwise *backward-looking*.

If t is forward-looking and $\pi(t)$ is backward-looking, then these points are paired up and become candidates for being swapped. As an example, suppose π has a cycle $(w\ x\ y\ z)$ and Dir yields bits 0, 1, 0, 0. Only x and y are paired up, and assuming they would be finally swapped the resulting cycle is $(w)(y\ x)(z)$.

Let $t \in T$ be the input value which is going to be mapped by CS_π. First, the algorithm checks the value of $Dir(t)$. If $Dir(t) = 1$, and $Dir(\pi(t)) = 0$ then t is matched with $\pi(t)$. Otherwise, if $Dir(t) = 0 \wedge Dir(\pi^{-1}(t)) = 1$, then t is matched with $\pi^{-1}(t)$. Let (x,y) denote the resulting pair of points, equal to either $(t,\pi(t))$ or $(\pi^{-1}(t),t)$.

The decision whether swap x and y or not is taken by I, and a pseudo-random function $B : T \to \{0,1\}$. The swap occurs if, and only if, $I(x,y) = 1$ and $B(t) = 1$. In domain targeting, only points from S will be swapped.

Algorithm 3.3: Cycle Slicer

Input : $T, t \in T, \pi$
Output: $c = CS_\pi(t)$

```
if Dir(t) = 1 then
    t' ← π(t)
    if Dir(t') = 0 ∧ I(t,t') = 1 ∧ B(t) = 1 then
        t ← t'
else
    t' ← π^{-1}(t)
    if Dir(t') = 1 ∧ I(t',t) = 1 ∧ B(t) = 1 then
        t ← t'
return t
```

Similarly to *RCW*, *Cycle Slicer* must perform many rounds to properly mix all elements, and its R-round variation is denoted as CS^R. During each iteration i, CS requires independent π_i, Dir_i, and B_i. The following corollary allows estimating how many rounds are needed for CS_π^R to yield a satisfactory level of randomness.

Conclusion 3.2 ***(excerpt from [19])***

Let $N_S = |S|, N_T = |T|$, *and let*

$$T = max\left(40\ln(2N_S^2), \frac{10\ln(N_S/9)}{\ln((1+7/144)((7/9)N_S^2 - N_S)/N_T^2}\right) + \frac{144N_T\ln(2N_S^2)}{N_S},$$

then

$$||v_{CS^r} - \mu_s|| \leqslant N_S^{1-2r}.$$

The following table presents applications of corollary 3.2 for selected N_T and N_S assuming the desired total variation distance must be less or equal 0.1.

N_S	N_T	$1/c$	$\mathcal{R}$
2^{10}	2^{11}	0.5	6,149
$2^{10} + 2^9$	2^{11}	0.75	3,544
$2^{10} + 2^9 + 2^8$	2^{11}	0.875	2,900
$2^{10} + 2^9 + 2^8 + 2^7$	2^{11}	0.9375	2,656
2^{127}	2^{128}	0.5	72,930
$2^{127} + 2^{126}$	2^{128}	0.75	38,513
$2^{127} + 2^{126} + 2^{125}$	2^{128}	0.875	30,548
$2^{127} + 2^{126} + 2^{125} + 2^{124}$	2^{128}	0.9375	27,595

Table 3.2: Smallest number of rounds $\mathcal{R}$ performed by CS_π^R such that $N_S^{1-2\mathcal{R}/T} \leqslant 0.1$.

As shown, in all situations CS_π performs better than R_2CW_π, and in the case when $|S| \approx |T|$ the improvement may be of a few orders of magnitude. Reason for this is that when S is a large fraction of T, then π's cycle decomposition contains many sequences of elements from S longer than 2. For instance, consider cycle (s, s', t, s'', s''') where all elements besides t are in S. $R_2CW\pi$ can not swap any points, while CS may mix some, or even all of them, depending on Dir and B.

3.2 Domain extension

Domain extension aims to address the issue of domains that may need including new points after the encryption scheme has been already deployed, just as it was described in section 1.4.3. Let D be the original domain with some existing mapping $G : T \to U$, where $U, T \subset D$ and T is called the preservation set. Define a new domain $\mathcal{X} \supset D$ and its corresponding encryption scheme $E_\mathcal{X}$. The essential requirement is that $E_\mathcal{X}$ is compatible with G, meaning that for every $t \in T$, $E_\mathcal{X}(t) = G(t)$. As noted in [19], it is unreasonable to expect that $E_\mathcal{X}$ constructed thus will be a PRP over $\mathcal{X}$. Consider a situation where we add a single point x_1 to the original domain D and let $\mathcal{X}_1 = T \cup \{x_1\}$. To conform the requirement of compatibility, we must have $E_{\mathcal{X}_1}(x_1) = x_1$. Now, if we add $x_2, x_3, ..., x_n$ in the same manner, then for each $i \in [n]$, $E_{\mathcal{X}_n}(x_i) = x_i$. Therefore, to make the domain extension feasible, we must assure that the new points are added in batches, not one by one.

3.2.1 Cycle Slicer

Miracle and Yilek showed in [19] that it is possible to use the *Cycle Slicer* for domain extension. To do this, they investigate a similar problem of *domain completion* and conclude that a solution for one can be as well used for the other.

: CSDC	: CSDCInv
Input : $x \in \mathcal{X}$	**Input** : $x \in \mathcal{X}$
Output: $c = E_{\mathcal{X}}(x)$	**Output:** $d = D_{\mathcal{X}}(x)$
1 **if** $x \in T$ **then**	1 **if** $x \in U$ **then**
2 $\quad x \leftarrow G(x)$	2 $\quad x \leftarrow G^{-1}(x)$
3 **else**	3 **else**
4 $\quad$ **if** $x \in U$ **then**	4 $\quad x \leftarrow CSInv^{r}(x)$
5 $\quad\quad x \leftarrow$ `first(x)`	5 $\quad$ **if** $x \in T$ **then**
6 $\quad x \leftarrow CS^{r}(x)$	6 $\quad\quad x \leftarrow$ `last(x)`
7 **return** x	7 **return** x

Figure 3.1: Domain extension/completion algorithms using *Cycle Slicer*.

Overview

To understand how the *CSDC*[1] works, consider the cycle decomposition of the partial permutation between T and U, defined by G. Notice that it can have cycles, with all of their points in $T \cap U$; lines, when for some $t \in T$, there is $G(t) \in U - T$; and single points for all remaining $x \in X - T - U$. *CSDC* needs not to affect points in cycles as they all remain in the preservation set T, and hence should be mapped using G. Lines only need to be handled at their ends, so *CSDC* "collapses" them into points and ignores their internal edges. Finally, it takes all the points that existed before, adds these created while collapsing lines, and uses *Cycle Slicer* to create a PRP over them.

Miracle and Yilek argue, that after bringing back all the cycles and expanding lines we obtain an encryption scheme on $\mathcal{X}$, which preserves the mappings defined by G.

Details

To construct *CSDC*, they use an internal r-round *Cycle Slicer* with the following inclusion function

$$\mathcal{I}_c(x, y) = \begin{cases} 1 \text{ if } x \notin U \wedge y \notin U \\ 0 \text{ otherwise} \end{cases}.$$

Such $\mathcal{I}_c$ is exactly the same as the inclusion function for domain targeting on set $X - U$. If we only used CS with $\mathcal{I}_c$, it would mix all points from $X - U$ which is what we want. On the other hand, it would also include the *first* point in the line, which should instead be mapped by G, and ignored the *last* point in the line. The idea behind `CSDC` is to map *last* to *first* during encryption and use CS to perform encryption.

Consider the `CSDC` algorithm as shown on figure 3.1. If $x \in T$, then we use the mapping G, thereby satisfying the compatibility condition. Otherwise, if $x \in U - T$, then we use the `first` algorithm, which traverses the line created by G backward until it hits the first point x' outside U[2].

: first	: last
Input : $x \in U - T$	**Input** : $x \in T - U$
Output: $c = T - U$	**Output:** $x \in U - T$
1 **while** $x \in U$ **do**	1 **while** $x \in T$ **do**
2 $\quad x \leftarrow G^{-1}(x)$	2 $\quad x \leftarrow G(x)$
3 **return** x	3 **return** x

Figure 3.2: The internal **first** and **last** algorithms used in `CSDC`.

[1] Cycle Slicer Domain Completion

[2] This makes up the "collapsing" part of the description. Notice that `first` will always end since it cannot be executed if $x \in T$, and hence, x can never be in a cycle.

In our applications, the preexisting mapping G is probably an FPE scheme which needed to be expanded by adding new points to the domain. In many such applications, there is that $T = U$, and therefore `CSDC` can be simplified. Assuming that T and U are equal, the cycle decomposition of G cannot have any lines as they imply that the sets $T - U$ and $U - T$ are non-empty, which is not true. Hence, it contains only cycles, which are ignored by `CSDC` in any case; and points, mapped by CS.

We can construct `CSDC_Eq` for domain extension when $T = U$ simply by using G for the points in the preservation set and the internal *Cycle Slicer* for all the remaining ones.

3.3 Encryption of regular languages

One general method for encryption of regular languages was proposed by Bellare et al. in [3]. Their idea is based on the *RtE* approach and can be divided into three separate parts. The first step is to convert the input regular expression r into a *Deterministic Finite Automaton* (*DFA*) M_r. Next, to make the ranking and unranking on M_r efficient, a preprocessing step is required. It constructs rank-tables, which are then used during the third step as cached intermediate values.

The domain of the resulting FPE scheme is $\mathcal{X} = L[[r]]$, and the slices $\mathcal{X}_n$ are defined as $\mathcal{X}_n = S_n$. Let $S_n = \{s \in L[[r]] : |s| = n\}$. To utilize *RtE*, the algorithm creates the set $R_n = \{Rank(s) : s \in S_n\}$ and uses an internal FPE scheme over R_n to perform the actual encryption.

Since ranking S_n is basically indexing the words of length n in the lexicographic order, R_n is equal to $\mathbb{Z}_N$, where $N = \max_{s \in S}$. Therefore, GraFPE can be used as the internal scheme.

3.3.1 Converting RE into DFA

Converting a language L given by an arbitrary regular expression into a DFA is a problem studied extensively since the last century. Currently, there is no known universally efficient way of performing this conversion, and it is believed that none exists. A premise for this is that the equivalence problem for regular languages is proven $PSPACE - complete$ [22], while for DFAs this can be done in polynomial time. [13].

A typical way of converting RE into DFAs is by using Thompson's construction algorithm to build an intermediate NFA and then Rabin-Scott powerset construction to convert it into the final DFA. The intermediate step is however undesirable since it increases the difficulty of implementation, and direct solutions exist.

One such solution uses derivatives of regular expressions introduced by Brzozowski in [5].

Derivatives of regular expressions

Let $\mathcal{L} \subset \Sigma^*$ be a language. The derivative of $\mathcal{L}$ with respect to a symbol $a \in \Sigma$ is the language of strings in $\mathcal{L}$ that have a as the first symbol, but *without* this symbol.

Example 3.2 *Let $\mathcal{L} = \{aaaa, abcd, efg\}$. Then the derivative of $\mathcal{L}$ with respect to "a" is the language $\mathcal{L}' = \{aaa, bcd\}$.*

To formalize the notion of derivatives, a few definitions from [5] and [20] must be recalled.

Definition 3.1 *A regular expression is defined recursively as follows:*

1. *The symbols $a_0, a_1, a_2, ..., a_{k-1}, \epsilon, \emptyset$ are regular expressions.*
2. *If P and Q are regular expressions, then regular expression are also: PQ (concatenation), P^* (Kleene closure), and $f(P, Q)$ where f is any Boolean function of P and Q. $P + Q$ denotes logical-or, $P\&Q$ logical-and, $\neg P$ logical negation.*

Symbols $a_0, a_1, a_2, ..., a_{k-1}$ denote characters from some finite alphabet Σ. ϵ is the "empty" character. $\emptyset$ represents a "failed" regular expression which is used to simplify the definitions of derivatives: when a derivative of an expression yields an empty language, the derivative itself is said to be $\emptyset$.

Definition 3.2 ***The nullability function.***
The regular expression r is nullable if $\epsilon \in \mathcal{L}[[r]]$. The nullability function $v(r)$ is defined as follows:

$$v(r) = \begin{cases} \epsilon \text{ if } r \text{ is nullable} \\ \emptyset \text{ otherwise} \end{cases}.$$

Definition 3.3 *Given a language $\mathcal{L}$, the derivative $D_a\mathcal{L}$ of $\mathcal{L}$ with respect to a is defined as $D_a\mathcal{L} = \{v : a \cdot v \in \mathcal{L}\}$. The derivative of a regular expression r is defined by a derivative of the language $\mathcal{L}[[r]]$ generated by r, i.e. $D_a r = D_a\mathcal{L}[[r]] = \{v : a \cdot v \in \mathcal{L}[[r]]\}$.*

The definition above is naturally extended to sequences of characters and for logical functions. The derivative of a regular expression is always a regular expression [5]. The following rules follow from the definition of the derivative and allow to compute it for Boolean functions mentioned before.

$$\begin{aligned} D_a\epsilon &= \emptyset \\ D_a a &= \epsilon \\ D_a b &= \emptyset \\ D_a\emptyset &= \emptyset \\ D_a(r+s) &= D_a(r) + D_a(s) \\ D_a(r \cdot s) &= D_a(r) \cdot s + v(r) \cdot D_a(s) \\ D_a(r^*) &= D_a(r) \cdot r^* \\ D_a(r\&s) &= D_a(r)\&D_a(s) \\ D_a(\neg r) &= \neg D_a(r) \end{aligned}$$

To compare two regular expressions, a notion of equivalence is introduced. Let r and s be regular expressions. The equivalence relation between r and s is defined as follows: $r \equiv s \iff L[[r]] = L[[s]]$. For example, $a + b \equiv b + a$, or $\epsilon \cdot a \equiv a$.

Converting RE into DFA

The construction assigns one state for each equivalence class on the set of all derivatives of the original regular expression r. The initial state q_0 represents r itself. The transition rule is following: $\delta(q_i, a) = q_{D_a q_i}$, where $q_{D_a q_i}$ holds all derivatives equivalent to $D_a q_i$. For example, each state q_j adjacent to q_0 is derivative, or equivalent to derivative D_{a_j} of r.

For simplicity, it will be assumed that each state has one corresponding regular expression up to equivalence relation. An example of a DFA constructed in the described way is shown on Figure 3.3.

The state $q_\emptyset$ is a *sink*, meaning that whenever the DFA enters $q_\emptyset$, it never can "get out". This happens for words that were not matched by r. By construction $q_\emptyset$ is never an accepting state.

The algorithm works by utilizing a few key observations from [5]:

1. Every regular expression has a finite number of distinct[3] derivatives. *(Theorem 4.3)*

2. Two states of a DFA are indistinguishable if their corresponding regular expressions are equivalent derivatives of the original regular expression r. *(Theorem 5.1)*

Algorithm 3.4 recursively finds all states that are derivable from the regular expression r corresponding to the initial state q_0 and adds the new distinct found ones to Q. To build a DFA for any r, all that is needed is initializing Q with $q_0 = r$ and later running `Build DFA States` to construct the remaining states as well as the transition function. Let r_j be the corresponding regular expression of a state q_j. The set of accepting states is defined as follows $\mathcal{F} = \{q_j \in Q : v(r_j) = \epsilon\}$.

The pseudo-code for constructing DFA from a regular expression is shown on algorithm 3.5. For simplicity, the states of the DFA are used interchangeably with their corresponding regular expressions.

[3] up to equivalency

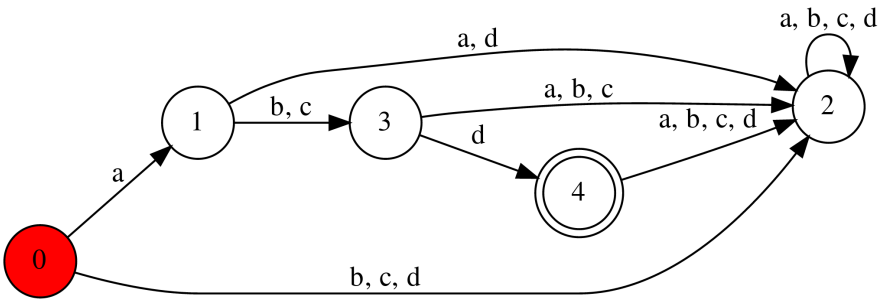

Figure 3.3: DFA equivalent to a regular expression $r = a(b + c)d$. State q_0 is equivalent to r, state q_1 is $D_a q_0 = (b + c)d$, and state $q_3 = D_b q_1 = D_c q_1 = d$. The only accepting state is $q_4 = D_d q_3 = \epsilon$. All derivatives equivalent to $\emptyset$, for example $D_b q_0$, are in q_2.

Algorithm 3.4: Build DFA States

Input : q, Q, δ
Output: Q', δ'

$Q' \leftarrow Q; \delta' \leftarrow \delta$
for $c \in \Sigma$ **do**
  $q_c \leftarrow D_c q$
  if $\exists q' \in Q'$ *such that* $q' \equiv q_c$ **then**
    $\delta(q, c) \leftarrow q'$
  else
    $Q' \leftarrow Q' \cup \{q_c\}$
    $Q', \delta' \leftarrow$ ***Build DFA States***(q_c, Q', δ')
    $\delta(q, c) \leftarrow q_c$
return Q', δ'

Algorithm 3.5: Convert RE into DFA

Input : r – regular expression
Output: $M = (Q, \Sigma, \delta, q_0, F)$ – DFA equivalent to r

$q_0 \leftarrow r$
$Q \leftarrow \{q_0\}$
$\delta \leftarrow \emptyset$
$Q, \delta \leftarrow$ ***Build DFA States***(q_0, Q, δ)
$F \leftarrow \{q \in Q : v(q) = \epsilon\}$
return $M = (Q, \Sigma, \delta, q_0, F)$

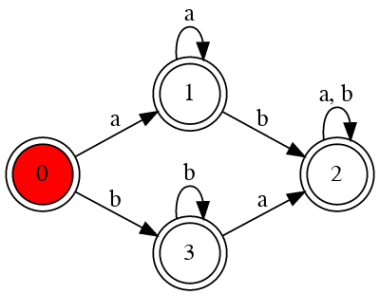

Figure 3.4: Example of DFA generated using algorithm 3.4 for $r = (ab^* + ba^*)^*$ with *similarity* relation used instead of *equivalence*. The automaton is correct, but not minimal.

Practical considerations

As noted in [20] it is expensive to check if two regular expressions are equivalent. Therefore, a second notion - *similarity* - is introduced. It serves as an easy-to-compute approximation of equivalence.

Definition 3.4 *Let r and s be regular expressions. The similarity relation $\approx$ is the smallest relation that satisfies the following conditions:*

$$
\begin{array}{rcl@{\qquad}rcl}
r\&r & \approx & r & r+r & \approx & r \\
r\&s & \approx & s\&r & r+s & \approx & s+r \\
\emptyset\&r & \approx & \emptyset & (r+s)+t & \approx & r+(s+t) \\
\neg\emptyset\&r & \approx & r & \neg\emptyset+r & \approx & \neg\emptyset \\
 & & & \emptyset+r & \approx & r \\
(r\cdot s)\cdot t & \approx & r\cdot(s\cdot t) & & & \\
\emptyset\cdot r & \approx & \emptyset & (r*)* & \approx & r* \\
r\cdot\emptyset & \approx & \emptyset & \epsilon* & \approx & \epsilon \\
r\cdot\epsilon & \approx & r & \emptyset^* & \approx & \epsilon \\
\epsilon\cdot r & \approx & r & \neg(\neg r) & \approx & r
\end{array}
$$

Both [5] and [20] prove that similarity implies equivalence, and therefore it is sufficient to just change the line 4 in algorithm 3.4 to assure it checks for $q' \approx q_c$. The new algorithm will be correct for the same reasons as before, but it will be much more efficient as the similarity is easily verifiable.

Using similarity instead of equivalence does no longer guarantee the minimality of the resulting DFA, however, in many applications, the number of states is optimal, or close. Experiments performed in [20] show that in all cases but two, the resulting DFA was indeed minimal.

3.3.2 DFA ranking

The solution used for efficient ranking and unranking of DFAs was shown by Bellare et al. in [3]. Its efficiency relies on a rank table which is built by *dynamic programming* during a preprocessing step. To bound the size of the domain, this algorithm can be used only for ranking words of some previously specified length n.

The rank table T assigns for each state $q \in Q$ and $n \in \mathbb{N}$ the number of distinct words of length n that would be accepted by a DFA $\mathcal{M}$ if q was its initial state. As an initial step, each pair $(q_F, 0)$, where $q_F \in F$ is initialized by 1, which represents ϵ. Let $\mathcal{A}_{q,n}$ denote the set of words of length n accepted by $\mathcal{M}$ when q is the new initial state. The algorithm continues by observing that

$$\mathcal{A}_{q,n} = \bigcup_{a \in \Sigma} \mathcal{A}_{\delta(q,a),n-1}.$$

: Build Table

Input : n, Q, F, δ, Σ
Output: T

for $q \in Q$ **do**
  if $q \in F$ **then**
    $T[q, 0] \leftarrow 1$
for $i \leftarrow 1, ..., n$ **do**
  for $q \in Q$ **do**
    for $a \in \Sigma$ **do**
      $T[q, i] \xleftarrow{+} T[\delta(q, a), i - 1]$
return T

: Rank

Input : $s \in L$
Output: $c \in \mathbb{N}$

$q \leftarrow q_0; c \leftarrow 0; n \leftarrow |s|$
for $i \leftarrow 1,...,n$ **do**
  for $j \leftarrow 1, ..., ord(s_i) - 1$ **do**
    $c \xleftarrow{+} T[\delta(q, a_j), n - i]$
  $q \leftarrow \delta(q, s_i)$

return c

: Unrank

Input : $c \in \mathbb{N}$
Output: $s \in L$

$s \leftarrow \epsilon; q \leftarrow q_0; j \leftarrow 1$
for $i \leftarrow 1, ..., n$ **do**
  while $c \geqslant T[\delta(q, a_j), n - i]$ **do**
    $c \leftarrow c - T[\delta(q, a_j), n - i]$
    $j \xleftarrow{+} 1$
  $s_i \leftarrow a_j; q \leftarrow \delta(q, s_i)$
  $j \leftarrow 1$
return s

Figure 3.5: Triplet of algorithms that are used for handling *RtE* on DFAs. **Left**: the preprocessing step that builds the Rank Table. **Middle**: Computing the rank of a word s accepted by the DFA. **Right**: Unranking algorithm that maps the input number c to a word s.

Therefore, to find the final value of $T[q, n + 1]$ it is enough to compute the corresponding values of all of q's descendants.

By following the pseudo-codes presented in figure 3.5, it is easy to see that the time complexity of `Build Table` algorithm is $\mathcal{O}(n|Q||\Sigma|)$, where n is the length of the ranked words. The ranking and unranking functions can be computed in $\mathcal{O}(n|\Sigma|)$ time.

Example 3.3 *Consider the graph presented on figure 3.3. It accepts only two words, namely "abd" and "acd". Therefore, its rank table is following:* $T[0, 3] = 2$, $T[1, 2] = 2$, $T[3, 1] = 1$, *and* $T[4, 0] = 1$. *In all other cases,* T *is equal to* 0.

Since GraFPE constructs a graph of the same size as the domain, combining it with the currently described construction requires knowing how many words of length n are accepted by $\mathcal{M}$. Fortunately, the rank table already contains this information as $T[q_0, n]$, where q_0 is the initial state of $\mathcal{M}$.

Furthermore, one must impose a total order on characters in Σ. Assume $\Sigma = (a_0, a_1, a_2, ..., a_k)$. Define a function $ord : \Sigma \to \mathbb{N}$ as $ord(a_j) = j$.

Implementation

In this chapter, we present the implementation of our FPE library. It consists of all previously described schemes and protocols along with many additional utilities to ease the usage of the library. The primary target of the implementation is to show the variety of possibilities when combining multiple FPE-related algorithms, and therefore, it is biased towards simplicity and performance rather than security.

Our language of choice for the implementation is `C++` mainly because of its superior performance when compared with other popular languages such as `Python` or `Java`, and much greater verbosity than `C`.

For the best portability the number of dependencies is minimal. There is only one library requirement - `OpenSSL`, and we need `CMake` for the build process. For unit tests and integration tests we used a header-only lightweight library named `Catch2.hpp`. Since it is shipped in a single file, we included it into our implementation, and hence, no further installation is required.

The last dependency, however optional, is `Boost`. When available, it enables some additional functionality in *Cycle Slicer* and *Reverse Cycle Walker*.

4.1 Overview

The library consists of three main modules:

- `LibFPE` - the library's core. It contains the implementation of GraFPE itself, alongside three algorithms for domain targeting and one algorithm for domain extension. Internally, all algorithms use PRNGs based on `AES-128-CTR` or `AES-256-CTR`. Apart from performing encryption, the module provides an easy to use `API` for building $d-$regular graphs from a permutation set, and implementation of the *Inverse Riffle Shuffle*.

  The primary target of `LibFPE` is to make the process of creating encryption schemes for any domain as comfortable as possible. Since it is infeasible to support a large number of various domains, we chose a different approach. The library gives access to a few helper classes that can be treated as building blocks and nested one in another. This allows the user to construct the desired encryption scheme in the bottom-up approach.

- `DFARE` - a module that encapsulates the process of creating `Ranked DFAs` from a given regular expression. It consists of a custom regular expression parser using syntax from [5], and a hierarchy of classes representing the fundamental regular expressions such as sums and concatenations with support for computing derivatives and easily translatable to a DFA by algorithms described in 3.

- `PurLib` - a utility library containing helper functions for manipulating program parameters and parsing arguments. Its description is not included in this work.

4.2 LibFPE

To provide extensibility, LibFPE loosely couples its internal components. For each kind of a task, such as creating pseudo-random permutations or combining them into a valid set, the user can define a custom type, which provides a new solution. It is similar to class hierarchies typical for object-oriented programming, but we rejected them to improve performance; dynamic polymorphism can introduce a significant efficiency

deterioration. Instead, we use a compile-time polymorphism by *CRTP*[1] and utilize *SFINAE*[2] for rejecting types that do not match the required specification[3]. Although these solutions are much more verbose and harder to apply, they are a better approach for constructing *LibFPE* since the abstractions they provide are zero cost in the runtime.

4.2.1 GraFPE

The module contains two implementations of GraFPE: one for encryption of vectors, implemented according to the previous description, and another one for enciphering single points. The second one is particularly useful with *RtE* or domain-targeting techniques. The following snippet shows how to construct a fresh instance of GraFPE containing a new graph.

```
auto grafpe = GraFPE {
    key,    // for constructing graph and for encryption
    iv,     // for constructing graph
    N, d,   // parameters of the graph
    walk_length // length of the RCW
};
```

It is also possible to create the graph elsewhere and pass a pointer to it to GraFPE.

Graph builders

The process of building the graph is the most time and space consuming part of the entire implementation. Therefore, it has been highly optimized by improving data locality and thereby avoiding undesirable cache misses. The module provides a class named `PermGen_Compat`, which is responsible for generating pseudo-random permutations using *Inverse Riffle Shuffle (IRS)*[4]. Classes responsible for creating pseudo-random permutations are called *permutation generators* and can be passed to *graph builders*, which are algorithms for combining the permutations into valid permutations sets.

There is currently only one graph builder available, and it constructs the graph precisely as described in chapter 2. It is a greedy algorithm that always takes the first permutation that keeps P valid when added to it. This may not be the best approach, and thus, it is easy to define and use a custom graph builder instead.

Algorithm 4.1: GraphBuilder::build

Input :
Output:

```
1 P = ∅ · n // vector containing n empty sets
2 for i from 0 to d/2 − 1 do
3 |   P[i] = generator.create_permutation(n)
4 |   bad = find_bad_vertex(P, i)
5 |   while bad.has_value() do
6 |   |   generator.repermute(P.last, bad)
7 |   |   bad = find_bad_vertex(P, i)
8 return Graph{n, d, P}
```

The `generator.create_permutation` function creates a new pseudo-random permutation which is then added to P, and passed along to the `find_bad_vertex` function. This function assumes that P was previously valid and checks if the newest permutation does not invalidate it. Therefore, it only goes through the last

[1] CRTP - Curiously Recurring Template Pattern.
[2] SFINAE - Substitution Failure Is Not An Error.
[3] In OOP, we would say they do not implement a required interface.
[4] There is also a `PermGen` class which originated as a failed implementation of *IRS* that worked by swapping each element with a one selected pseudo-randomly. It has not been removed from the implementation as it may serve as a comparison for empirical tests. However, using it is discouraged.

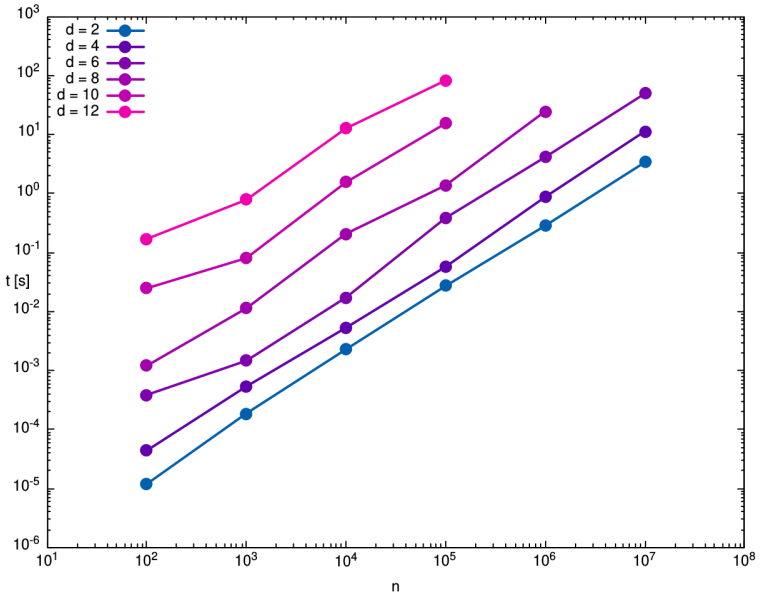

Figure 4.1: An average time of graph generation with n and d given. The average is calculated from ten samples. Notice that the current implementation does not allow some of the configurations showed on the figure, since it requires that $d = 2^{f-1}$ for some $f \in \mathbb{N}$.

permutation once, thereby reducing the complexity of validation from an $\mathcal{O}(nd^2)$ check to $\mathcal{O}(nd)$. If it finds any vertex with a loop or multiple edges, it returns its index, and, otherwise, an empty value.

If the new permutation p is not "compatible" with the ones already stored in P, it is mixed again by `generator.repermute`. This function takes advantage of the fact that p is already random, and therefore, there is no need to keep shuffling it until an *SUT* occurs. Consequently, it performs only a single iteration of *IRS* to create p', the validity of which is then rechecked by `find_bad_vertex`.

The figure 4.1 presents running times of `GraphBuilder::build` for various n and d. It shows that the time is linear with regard to the number of vertices, but grows exponentially when considering their degree - increasing d from 2 to 12 results in almost 10^3 times longer generation time.

Comparing the precise results from table 4.1 with these obtained in [15] shows that our implementation is around 10 times faster[5]. When d is small, it is the memory usage, not time, that becomes a problem. Indeed, to compute a valid set of permutations for $n = 10^9$ and $d = 4$ one needs approximately $32GBs$ of memory[6], while extrapolating results from figure 4.1 suggests that the computation needs around twenty minutes to finish.

Inverse Riffle Shuffle

The most time-consuming part of the *IRS* algorithm as presented in chapter 2 is checking whether all bit slots are distinct. In the first implementation, to find any duplicated elements, we used a *set* data structure. However, this resulted in long running times due to the cost of creating a new object for each check. Fortunately, the bit slots are constructed in such a way, that a check in $\mathcal{O}(n)$-time and $\mathcal{O}(1)$-space is possible.

Lemma 4.1 *Let $\mathcal{B}$ be the bit slots vector constructed by the IFS algorithm. Assuming that the bit slots are read backward, then, after each iteration, $\mathcal{B}$ is sorted.*

Proof (by induction)

[5]Nevertheless, our implementation was tested on a different machine than the one from [15].

[6]We store each permutation as a vector of 64-bit values, and for each of them we compute its inverse with the same size. Hence, the total memory we use, in bytes, is $8 \cdot n \cdot \frac{d}{2} \cdot 2 = 8nd$. For $n = 10^9, d = 4$ this results in around $32GBs$ of space.

d	n	time
2	10^2	$0.1188 \cdot 10^{-4}$
	10^3	$0.1875 \cdot 10^{-3}$
	10^4	$0.2287 \cdot 10^{-2}$
	10^5	$0.2715 \cdot 10^{-1}$
	10^6	0.2853
	10^7	$0.3395 \cdot 10^{1}$
4	10^2	$0.4383 \cdot 10^{-4}$
	10^3	$0.5356 \cdot 10^{-3}$
	10^4	$0.5294 \cdot 10^{-2}$
	10^5	$0.5753 \cdot 10^{-1}$
	10^6	0.8747
	10^7	$0.1138 \cdot 10^{2}$

d	n	time
6	10^2	$0.385 \cdot 10^{-3}$
	10^3	$0.1463 \cdot 10^{-2}$
	10^4	$0.1725 \cdot 10^{-1}$
	10^5	0.3852
	10^6	$0.4147 \cdot 10^{1}$
	10^7	$0.4939 \cdot 10^{2}$
8	10^2	$0.12 \cdot 10^{-2}$
	10^3	$0.1132 \cdot 10^{-1}$
	10^4	0.2084
	10^5	$0.1380 \cdot 10^{1}$
	10^6	$0.2473 \cdot 10^{2}$
	10^7	–

d	n	time
10	10^2	0.1323
	10^3	0.2061
	10^4	$0.1119 \cdot 10^{2}$
	10^5	$0.3243 \cdot 10^{2}$
	10^6	$0.3330 \cdot 10^{3}$
	10^7	–
12	10^2	0.1696
	10^3	0.7816
	10^4	$0.1273 \cdot 10^{2}$
	10^5	$0.8270 \cdot 10^{2}$
	10^6	–
	10^7	–

Table 4.1: Precise running times of `GraphBuilder::build` for various n and d. Obtained on Intel Core i7-8750H @ 2.20GHz. For d equal $8, 10$, or 12, some of the results took too much time to compute, and therefore they were omitted.

Consider $\mathcal{B}$ as a column vector initialized with N zeros, $\mathcal{B} = [0, 0, ..., 0]^T$. Denote $\mathcal{B}_i$ as $\mathcal{B}$ after i iterations of *IFS*.

1. After the first iteration of *IFS*, $\mathcal{B}$ will store only ones and zeros, with zeros on the top. Therefore,

$$\mathcal{B}_1 = [0, 0, ..., 0, 1, 1, 1, ..., 1]^T,$$

and hence, $\mathcal{B}_1$ is sorted.

2. Assume $\mathcal{B}_n$ is sorted for some n. We will show that this implies that $\mathcal{B}_{n+1}$ is sorted as well. Define the operator $|$ for vectors of equal lengths as pair-wise bit-concatenation and let $b_1, ..., b_N$ denote the random bits generated during the current iteration. Then

$$B_{n+1} = \begin{pmatrix} 11...00 \\ 10...10 \\ 11...01 \\ ... \\ 10...11 \end{pmatrix} | \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ ... \\ b_N \end{pmatrix} = \begin{pmatrix} 11...00|b_1 \\ 10...10|b_2 \\ 11...01|b_3 \\ ... \\ 10...11|b_N \end{pmatrix}$$

and $\mathcal{B}_{n+1}$ is B_{n+1} with all slots with 0 as the least significant bit moved to the top with their relative order preserved.

Consider the numbers formed by strings stored in the bit slots when reading backward. In such case, the new bits $b_1, ..., b_N$ are the most significant bits, and hence their value is greater than the value provided by all bits generated previously. As a result, all slots for which 0 was drawn, are smaller than any slot with 1 as the *MSB*.

By the inductive foundation $\mathcal{B}_n$ is sorted, and therefore, if we now ignore the newly drawn bits $b_1, ..., b_N$, we get two rising sequences of numbers. The first corresponds to the numbers from which a zero was removed, and they are on the top. Later come all slots corresponding to ones. Hence, putting $b_1, ..., b_N$ back makes the resulting $\mathcal{B}_{n+1}$ sorted.

♦

As a result, to check if $\mathcal{B}$ contains any duplicates, it is enough to compare all pairs of consecutive slots. Figure 4.2 shows how much does our solution improve the time efficiency of *IRS*. With the naive approach we were unable to construct a graph for $n = 10^6$ and $d = 2$.

Remark The way *IFS* manipulates $\mathcal{B}$ is basically the same as sorting it using *Radix Sort*, and hence, *IFS* may be seen as inverse-sorting.

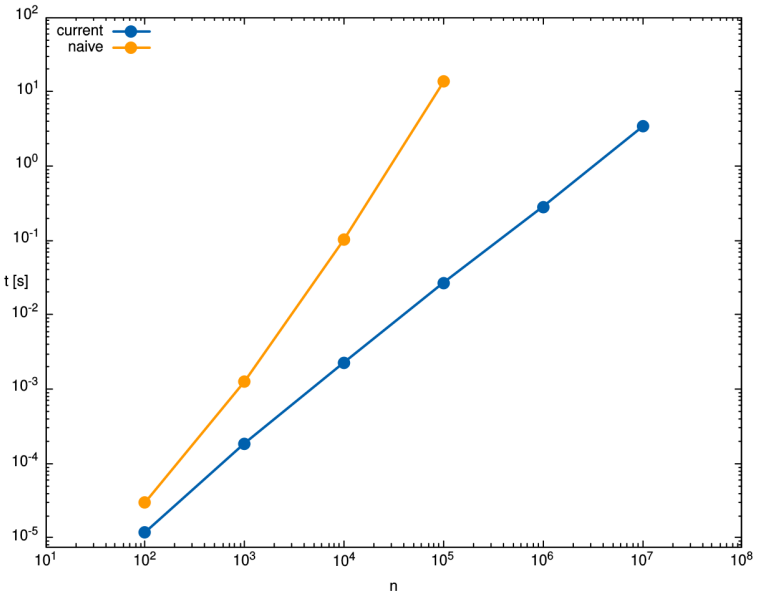

Figure 4.2: Comparison of running times of an optimized version of *IRS* with a solution which does not assume that $\mathcal{B}$ is sorted and compares all pairs of slots in $\mathcal{B}$. For $n = 10^5$, the current solution is almost 500 times faster.

Random Walks

The family of classes responsible for pseudo-random walks over graphs is called the *graph walkers*. Currently, it consists of the implementation of the *simple random walk* algorithm, done precisely as it was described in chapter 2, and supports only undirected graphs.

The encryption process can use as a *PRNG* any of the available block ciphers, for instance, AES-128-CTR or AES-256-CTR.

4.2.2 Domain targeting

As stated before, the library contains three different algorithms for *domain targeting*, namely, *Cycle Walker*, *Reverse Cycle Walker*, and *Cycle Slicer*. In all contexts, their implementations may be used interchangeably, but one must be aware in which cases each of them performs better or worse. This subject is described in detail in chapter 3.

For *Cycle Walker* the implementation is trivial and very efficient. All that is needed is a predicate function $P : \mathcal{X} \to \{0, 1\}$ and the internal encryption scheme. The implementation of *Reverse Cycle Walker* as well as *Cycle Slicer* is much more complicated as both of these algorithms require a separate pseudo-random function for each round[7].

The *PRFs* need to be both time and memory efficient and, by definition, deterministic. The first attempt resulted in creating the `RangeBoolPRF` class which takes $N \in \mathbb{N}$ as input and throws a coin for each element in $\mathbb{Z}_N$. The results obtained thus are stored in the internal `bitset`[8] and returned on demand. However, this class performs poorly for bigger domains as it flips the coin for each of its points. Moreover, in case of *CS* or *RCW* only one or two points of the whole domain are needed; therefore, `RangeBoolPRF` is not particularly suitable for this problem.

The next attempt resulted in creating the `LazyBoolPRF` class which encrypts points on demand using a block cipher and returns the first bit from the obtained ciphertext. As an optimization of cases when the client asks multiple times for the same value, the implementation stores the already collected mappings in an `std::unordered_map`. However, in `C++`, a map must have all keys of the same type, and hence, we need to

[7] In fact, *Cycle Slicer* needs **two** independent *PRFs* for each round.
[8] A data structure similar to a vector, but taking advantage of the fact that a Boolean value can be stored in a single bit.

transform all input values in a way such that they can be later retrieved when passed again. For this purpose, we selected `boost::hash` which indeed met our expectations, but the implementation, as a whole, turned out too slow.

4.2.3 Rank-then-Encipher

At the moment the implementation contains two types of *RtE*: one for regular languages and the other for ordered containers of small size.

4.2.4 Custom domains

As stated before, the strength of the library relies on the building blocks that ease the process of creating custom FPE schemes. Currently, the implementation provides the following classes:

- `ContainerCrypter` - given an ordered container C, it performs the most basic *RtE*, by using GraFPE to encrypt indexes of elements from C. It can be used to encrypt small sets of data hard to define otherwise such as names or surnames.
- `DFACrypter` - given a regular expression R or a DFA $\mathcal{M}$, equivalent to R, it constructs the Rank Table as described in chapter 3, and uses GraFPE to encrypt indexes of words accepted by $\mathcal{M}$.
- `TupleCrypter` - given a tuple of crypters it wraps the encryption of their corresponding data formats with a single call. Can be used for combining multiple crypters into one.
- `CastCrypter` - Given data in one format and a crypter suitable for a different one, it uses the predefined *cast* and *uncast* methods to transform the input back and forth. This class may be treated as a generalization of the *RtE* approach.
- `TargetedCrypter` - applies the specified *domain targeting* method over its internal crypter such as any described before, or GraFPE itself.

These types constitute the family of *crypters*. Their biggest advantage is that they can be nested one in another, and therefore one can consider splitting domain to independent parts that can be encrypted separately.

4.3 DFARE

This module contains all functions and classes needed to build *Ranked DFAs* from regular expressions. Example usage is shown in the snippet below:

```
auto dfa = RankedDFA::from_string{"(ab*+ba*)*", 10};

dfa.rank("ababbabaab");      // == 361
dfa.unrank(361);             // == "ababbabaab"
```

To achieve this, we first convert the input string into an instance of our custom regular expression class. Then, we use algorithm 3.5 from chapter 3 to convert it into a DFA, and finally we run algorithms presented on figure 3.5 to compute its Rank Table. The resulting `RankedDFA` object can be further passed to a `DFACrypter` and used to build an *RtE* scheme.

We define three operators: $^*, \cdot, +$ for a regular expression, sorted by their precedence, from the highest to the lowest. The order of operations may be changed by using parentheses as usual.

- `XY` - `X` concatenated with `Y`. Concatenation is implicitly defined by operator $\cdot$.
- `X+Y` - `X` *or* `Y`.
- `[X-Y]` - all the characters between `X` and `Y`, inclusive.
- `X`* - `X` any number of times, including zero.

4.3.1 Building RegExp from string

The process of converting the input string s into a `RegExp` consists of a few stages, which include:

1. the preprocessing step, which translates all syntax sugar in s to the canonical operators. For example, it transforms `[a-d]` to `(a+b+c+d)`. Additionally, if s has "." in it, the algorithm inserts a backslash before it in order to allow the future steps to recognize it as an input character instead of an operator.

2. the operator-inserting step, responsible for adding the concatenation operator wherever it is needed. For example, it changes every `ab` into `a.b`.

3. the infix-to-postfix step, needed to evaluate and remove all the parentheses. For instance, it converts `(a+b)*` to `ab+*`.

4. construction step, which builds the nested structure of a regular expression and returns the final object.

RegExp classes

Regular expressions are used in DFARE mostly for computing their derivatives, and hence their implementation is directed at optimizing such use-case. We defined the `RegExp` class for the most basic regular expression such as `a`. Then, we added a new type for each of the operators; therefore, we have `Concat` for the `.` operator, `Sum` for `+`, and `Closure` for `*`. To properly define `*` we also needed `Eps` to represent a regular expression of zero length, and, last but not least, `Empty` meaning a "failed" regular expression, or one for which the regular language that is generated by it is empty.

Each of these classes performs reductions upon instance construction to simplify the regular expressions they contain. For example, when one has $r = a^*$, then passing r to the `Closure` constructor results in $r' = a^*$ instead of $(a^*)^*$. Similarly, `Sum{Sum{a, b}, b}` == `Sum{a, b}` and `Sum{a, a}` == `RegExp{a}`.

These reductions prove essential as omitting any of them may result in an infinite number of derivatives, thereby making the DFA construction algorithm loop.

For example, consider parsing a regular expression given by a string $s = (ab^* + ba^*)^*$. The preprocessing step leaves the input unchanged, and after the operator-inserting step, we get $s' = (a.b^* + b.a^*)^*$. After transforming s' to the postfix notation, we obtain $s'' = ab^*.ba^*.+^*$ and from this we construct the final `RegExp` object r.

The process of transforming r into a DFA m is performed by first computing the derivative of r with respect to "a" which is the first character in its alphabet[9]. The result is $D_a r = b^*(ab^* + ba^*)^*$ which is disctinct from r, and hence creates a new state in m. Next, we consider derivatives of $D_a r$.

To compute the derivative of $D_a r$ with respect to a we treat it as a concatenation of $\rho = b^*$ and r. By following rules from chapter 3 we get for ρ that $D_a\rho = D_a(b^*) = D_a(b) \cdot b^* = \emptyset \cdot b^* = \emptyset$. Now, the derivative of $D_a r$ with respect to a is

$$D_a(D_a r) = D_a(\rho \cdot r) = D_a(\rho) \cdot r + v(r) \cdot D_a r = \epsilon \cdot r + D_a r = \epsilon + D_a r = D_a r.$$

Since this derivative is equivalent to an already existing state in m, we ignore it and proceed to the next character in the alphabet. We compute $D_b(D_a r)$ in the same manner and obtain a new state $D_{ba} r$.

While evaluating $D_{aba} r$ we get the following intermediate result:

$$D_{aba} r = ... = (b^*((ab^* + ba^*))^* + (a^*((ab^* + ba^*))^* + b^*((ab^* + ba^*))^*)),$$

which is a sum of three elements where two of them, namely the first and the last, are the same. In the first implementation, we omitted one of the reductions, which resulted in this example looping indefinitely as it creates even bigger derivatives over time and we never reduced them properly.

[9] In the implementation, we assumed, that all the characters are ASCII and ordered as such.

4.4 Examples

We will demonstrate the possibilities of the library by constructing a custom encryption scheme that will be able to encipher a single data row consisting of a name, surname, email address, and credit card number. The input will be passed as a four-element tuple `(name, surname, email address, CCN)`, where all elements are of the string type, except for `CCN`, which is a 64-bit unsigned integer.

4.4.1 Names and surnames encryption

The simplest approach for encrypting names would be to use the `ContainerCrypter` to perform the encryption on some predefined set N of names. The algorithm simply finds the index i_n of the input name $n \in N$ and enciphers i_n using GraFPE to obtain $j = i_{E(n)}$. Finally, $c = E(n)$ is found by dereferencing the $j-$th position in N.

The data structure used for storing names in the implementation needs to preserve ordering and to have a random-access iterator. Therefore a `std::vector`, or `std::set` are sufficient.

The following snippet shows the process of building an FPE scheme for names.

```
std::vector<std::string> names { "Asia", "Basia", "Kasia", ..., "Tomek" };

auto grafpe = ScalarGraFPE{
        key,
        iv,
        names.size(),  // the size of the graph
        deg,           // the degree of the graph
        walk_length    // the length of the random walk during encryption and decryption
};

auto name_crypter = ContainerCrypter { names, grafpe };

// example of usage:
assert(name_crypter.decrypt(name_crypter.encrypt("Basia")) == std::string{"Basia"});
```

The `ScalarGraFPE` class is the same as GraFPE, but for messages consisting of a single point. It is particularly useful for domain targeting algorithms. However, if one attempts to encrypt whole vectors of names, then the `GraFPE` class should be chosen.

We could encrypt surnames in the very same manner, but also more sophisticated solutions are available. For example, it is possible to build a regular expression that matches only some predefined surnames or to create an encryption scheme over all strings of some length, and utilize domain targeting to narrow it down. However, there is no noticeable advantage of both the alternatives, so the simplest solution is chosen.

4.4.2 Email addresses encryption

Successful encryption of email addresses is much more complicated than names because there are infinitely many valid ones, and they may be as short as a few letters or have dozens of characters. Moreover, a valid email address can have letters, numbers, and a few other signs.

Single regular expression

Arguably the simplest solution would be to take any widely used regular expression for matching email addresses and pass it to the `DFACrypter`. This approach has a few flaws, however. First, the resulting domain N_1 would be enormous, and GraFPE needs to construct a graph with the number of vertices equal to $|N_1|$[10]. Second, the results would be highly unsatisfactory. The probability that the enciphered point c would contain any meaningful substring, such as a real domain name, is negligible.

[10] Unless a solution for encrypting numbers of arbitrary size is used.

We can mitigate these problems to some extent by explicitly specifying the legal domain names, as shown in the following snippet.

```
auto email_regex = "[a-z][a-z]*@(wp + onet).(com + pl + eu + net)";
auto dfa = RankedDFA::from_string(email_regex, length);
auto grafpe = ScalarGrafpe{
        ...
        dfa.max_rank                // the size of the graph
        ...
};
auto email_crypter = rte::DFACrypter { dfa, grafpe, 10 };
```

Assuming the domain consists of strings of length 10, the size of $N = \mathcal{X}_{10}$ is over 24.7 million, what makes the encryption process both time and memory consuming.

Furthermore, when we use a single regular expression for the whole email address, then the necessary length restriction introduces a bias. The reason for this is that there are many more sequences of length 5 that will match the left part of the regular expression than these of length 1. The encryption scheme will, therefore, prefer long local-parts with short domains. This *is* a PRP over N, but N itself is not a good representation of the real-world situation.

Tuple solution

A solution for both of these issues is to make the local-part independent from the domain. A way for doing this is to treat the email address as a tuple `(local-part, domain)` and employ two separate crypters to perform the actual encryption. To make the usage simpler, we will pass these two objects to a `TupleCrypter`, which will wrap the encryption up with a single call. Finally, a `CastCrypter` will be used. It will transform the input email address passed as a string into a tuple, and use the internal `TupleCrypter` to encipher it.

```
std::vector<std::string> domains {"gmail.com", "onet.pl", ...};
auto domain_crypter = ContainerCrypter {
        domains,
        ScalarGrafpe {..., domains.size(), ...}
};

auto local_part_dfa = RankedDFA::from_string("[a-z][a-z]*", length);
auto local_part_crypter = DFACrypter {
        local_part_dfa,
        ScalarGrafpe {..., local_part_dfa.max_rank, ...},
        10   // length of the local-part
};

auto email_tuple_crypter = TupleCrypter { local_part_crypter, domain_crypter };
```

Creating the final `CastCrypter` is a little bit more complicated. Currently, the implementation requires the user to create a custom class that will inherit from the `CastCrypter` using static polymorphism (CRTP). The class needs to implement two methods: `cast_impl` and `uncast_impl`.

For clarity, the code in the following snippet will not be a correct `C++`. The full working example is included in the implementation.

```
class EmailCrypter: public CastCrypter { // as CRTP
    tuple cast_impl(std::string email) {
        return tuple { get_local_part(email), get_domain(email) };
    }

    std::string uncast_impl(tuple email) {
        return tuple.join('@');
```

```
    }
};
```

Now, we can perform the encryption as follows:

```
auto email_crypter = EmailCrypter { email_tuple_crypter };
assert(email_crypter.decrypt(email_crypter.encrypt("tomek@gmail.com"))
        == "tomek@gmail.com"s);
```

Splitting further

The one last problem in the presented example is that the size of the domain of local-parts grows exponentially with regard to their length. Assume the regular expression for the local-part is "$[a-z][a-z]^*$", and the desired length is 20 characters. The idea is to treat the input string as a vector of characters and use `GraFPE` for vector encryption.

As a result, the local-part crypter could be implemented similarly to `EmailCrypter`:

```
class LocalPartCrypter: public CastCrypter {  // as CRTP
    RankedDFA dfa;  // Needed, since only this class knows the conversion details.
    std::vector<uint64_t> cast_impl(std::string email) {
        return email.transform([](char c) { return dfa.rank(c); });
    }

    std::string uncast_impl(std::vector<uint64_t> email) {
        return email.transform([](char c) { return dfa.unrank(c); });
    }
};
```

Finally, the resulting construction is following.

```
auto dfa = RankedDFA { "[a-z]", 1};
auto local_part_crypter = LocalPartCrypter {
        dfa,
        Grafpe { ..., dfa.max_rank, ...}
};
```

Remark

As noted in chapter 2, GraFPE is not secure if the domain is small. Here, the internal graph has only 26 vertices, which is too little for practical purposes. A better solution would be to split the input string into substrings of length four. The resulting graph would have exactly 456976 vertices which is enough to provide a satisfactory level of randomness while keeping short running times and small memory consumption.

4.4.3 CCNs encryption

The last part of the tuple is the credit card number. In this example, we will define its corresponding format N as the set of 16−digit numbers that pass the Luhn checksum test.

One attempt could be to use `ScalarGrafpe` to perform encryption over the whole set of 16−digit numbers and then use domain targeting to find these which pass the test. This would, however, result in a domain of size around $0.9 \cdot 10^{16}$ which is too much for practical applications.

Another solution is to create a set X_N of numbers having format N, and use `ScalarGrafpe` to perform encryption directly on their indexes. Since [3] shows a construction of a *DFA* that recognizes numbers passing Luhn checksum test, the *RtE* approach may be sufficient. The automaton's input needs to be reversed, so an additional `CastCrypter` on top of the `DFACrypter` would be needed.

However, there is a more straightforward approach. Instead of treating the numbers as separate vertices, it is better to split them into vectors of digits, and then use `GraFPE` to encrypt whole vectors. To improve

the security of this solution, the base of the numeral system will be chosen greater than 10, for example, 10 000. The resulting graph would then have 10^4 vertices, and the vector would have four elements, all having a single number in $\{0, ..., 9\,999\}$.

To narrow the domain down to these numbers which pass the Luhn checksum test, we need an additional domain-targeting layer. We show the final construction on the snippet below.

```
class Digit16Crypter: public CastCrypter {
    std::vector cast_impl (uint64_t number) {
        return std::vector {four(number, 0), four(number, 1),
                            four(number, 2), four(number, 3)};
    }

    uint64_t uncast_impl (std::vetor number) {
        return to_integer(concat(number));
    }
};

auto dgt16_crypter = Digit16Crypter {
    Grafpe {
        key, iv,
        10000,          // base == size of the graph
        D,
        walk_length
    }
};

auto ccn_crypter = TargetedCrypter {
    CycleWalker {&luhn_checksum_and_16_dgt},
    dgt16_crypter
};

// Example of usage:
ccn_crypter.encrypt(1234'1234'1234'1238)  // == 4028 1219 1827 8742
```

Instead of `CycleWalker` one can use `ReverseCycleWalker` or `CycleSlicer` in exactly same manner.

4.4.4 Data-row encryption

The original aim of this example was to build an encryption scheme for whole rows of data. Having all the helper-objects implemented, it is now enough to pass them to a `TupleCrypter`.

```
auto row_crypter = TupleCrypter { name_crypter, surname_crypter,
                                  email_crypter, ccn_crypter };
// Example of usage:
row_crypter.encrypt({"Basia", "Nowak", "basia.nowak@gmail.com", 1234123412341238});
```

4.5 Limitations

The library includes in its scope various algorithms that make it seem like a matured and complex product. On the contrary, there are many aspects in which the implementation is lacking, insecure and/or inefficient. This section describes some of the most important problems it faces and how they may be addressed in the future.

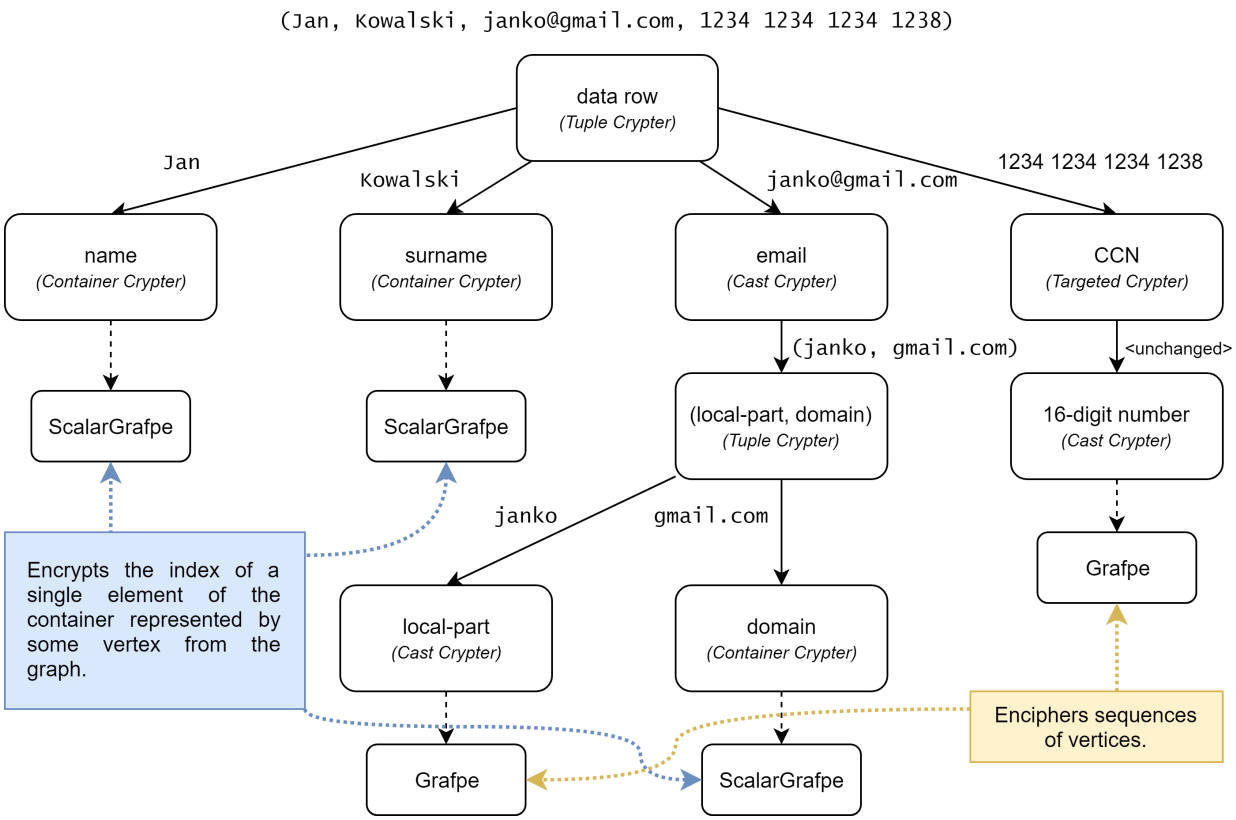

Figure 4.3: A diagram presenting how the data is divided in parts and passed to the internal helper classes. At the end of each branch there is a `ScalarGrafpe` or `Grafpe` object, which performs the actual encryption.

Statistical analysis

In the previous chapters we emphasized that GraFPE's security relies heavily on the parameters of its internal graph as well as the length of the *SRW* during the encryption phase. In this chapter we perform a few experiments, which allow to see how exactly is the security affected, by observing the distributions formed by ciphertexts created with GraFPE.

Summary

In this work we presented the general idea behind *Format-Preserving Encryption* and showed various techniques such as *Domain Targeting*, *Domain Extension*, and *Rank-then-Encipher*, which are often used alongside with it.

We also described and proved some of the properties of a currently unpublished *FPE* scheme named *GraFPE* that was introduced by Lorek, Słowik, and Zagórski in [15]. Our resulting implementation achieves better running times when considering the build phase, which is the most time and memory consuming part of the protocol.

Next, we used results from [3] to construct an *RtE* scheme for regular languages. To do this, we transformed them into *deterministic finite automata* by methods described in [20], which rely on derivatives of regular expressions introduced by Brzozowski in [5].

The final implementation allows to construct custom FPE schemes by using our predefined family of *crypter* classes. It also includes methods for converting regular expressions into *DFAs* what can be used as a separate library. We used it to show how one can implement an encryption scheme for email addresses.

Our library is still open for future improvements. We strongly consider adding ranking and unranking methods for unambiguous context-free grammars as this may allow encrypting valid programs written in simple programming languages.

Bibliography

[1] David Aldous and Persi Diaconis. "Shuffling cards and stopping times". In: *The American Mathematical Monthly* 93.5 (1986), pp. 333–348.

[2] *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 444–455.

[3] *International Workshop on Selected Areas in Cryptography*. Springer. 2009, pp. 295–312.

[4] *Cryptographers' Track at the RSA Conference*. Springer. 2002, pp. 114–130.

[5] Janusz A Brzozowski. "Derivatives of regular expressions". In: *Journal of the ACM (JACM)* 11.4 (1964), pp. 481–494.

[6] Artur Czumaj and Mirosław Kutyłowski. "Delayed path coupling and generating random permutations". In: *Random Structures & Algorithms* 17.3-4 (2000), pp. 238–259.

[7] *Annual International Cryptology Conference*. Springer. 2017, pp. 679–707.

[8] Morris Dworkin. "Recommendation for block cipher modes of operation: methods for formatpreserving encryption". In: *NIST Special Publication* 800 (2016), 38G.

[9] Andrew V Goldberg and Michael Sipser. "Compression and ranking". In: *SIAM Journal on Computing* 20.3 (1991), pp. 524–536.

[10] *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2017, pp. 499–527.

[11] Olle Häggström. *Finite Markov chains and algorithmic applications*. Vol. 52. Cambridge University Press, 2002.

[12] *Annual International Cryptology Conference*. Springer. 2018, pp. 221–251.

[13] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. "Introduction to automata theory, languages, and computation". In: *Acm Sigact News* 32.1 (2001), pp. 60–65.

[14] *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1992, pp. 24–36.

[15] Paweł Lorek, Marcin Słowik, and Filip Zagórski. *GraFPE - PRP-secure format preserving encryption scheme*. 2018.

[16] Eyal Lubetzky, Allan Sly, et al. "Cutoff phenomena for random walks on random regular graphs". In: *Duke Mathematical Journal* 153.3 (2010), pp. 475–510.

[17] Erkki Mäkinen. "Ranking and unranking left Szilard languages". In: *International journal of computer mathematics* 68.1-2 (1998), pp. 29–38.

[18] *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, pp. 679–700.

[19] *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 392–416.

[20] Scott Owens, John Reppy, and Aaron Turon. "Regular-expression derivatives re-examined". In: *Journal of Functional Programming* 19.2 (2009), pp. 173–190.

[21] Terence Spies. "Format preserving encryption". In: *Unpublished white paper, www. voltage. com Database and Network Journal (December 2008), Format preserving encryption: www. voltage. com* (2008).

[22] *Proceedings of the fifth annual ACM symposium on Theory of computing*. ACM. 1973, pp. 1–9.