POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Format Preserving Encryption for databases

Supervisor
prof. Antonio Lioy
prof. Andrea Atzeni

Candidate
Francesco Vaccaro

Internship Tutor
ing. Marco Mangiulli, ing. Luca Castello

DECEMBER 2024

A mio padre

A mia madre

A mia sorella

Summary

Companies migration to the cloud implies protection of their sensitive and private data. Encryption is the key tool for business's confidential data protection against cyber security threats. However, storing the data in an encrypted format requires to address critical issues: performance, format and ordering of data, protection of encryption keys. Format-preserving encryption (FPE) allows to encrypt the data in such a way that the output (the ciphertext) is in the same format as the input (the plaintext). Thus, FPE schemes are revealed to be exceptionally useful since they permit to encrypt existing databases without changing their format. In this thesis, firstly, the most important FPE techniques are presented and analysed. Then the robustness of these techniques is evaluated, using various attacks which were implemented in the recent years. Afterwards, a proof-of-concept implementation, working on a relational database, is provided. Finally, starting from the PoC implementation, some performance comparison on the FPE techniques are provided.

Contents

Chapter 1

Introduction

1.1 Motivation

In the recent years cloud migration has become a crucial phenomenon both in private and public sectors. Cloud migration is the process of moving applications, data, infrastructure, security and other objects to a cloud computing environment. The major reasons for cloud migration are to reduce costs and improve performances. A recent example of the growing importance of this topic is the allocation of funds for the cloud migration of the Italian public administration, as part of the National Recovery and Resilience Plan [1].

The increased digitalization has made the data access easier, consequently the security of the users' sensitive information should be the priority. In this context data breaches can lead to serious consequences: a breach in the bank account of a user can reveal personal information which can be used to implement various types of attack (i.e. ransomware attack, malware injections, ...), similarly a breach in a healthcare system that contains medical information can put patients' lives in danger. In order to enhance the protection of users' sensitive data and to thwart the danger of a cyber attack, some new regulations were created by governments or supranational entities.

The two major regulations are the General Data Protection Regulation (GDPR), adopted by the European Union on 14 April 2016, and the California Consumer Privacy Act (CCPA), signed into law by the California Government on 28 June 2018. One of the main differences between GDPR and CCPA is that the CCPA is applied only to companies with a gross annual turnover greater than 25 million US dollars, while the GDPR is applied to all companies which have a legal basis. In the case of GDPR the data controller, which is the organisation that collects personal data and determines how to process them, must report data breaches to national supervisory authority within 72 hours if they have an harmful effect on user privacy. GDPR foresee fines up to 4% of the annual worldwide turnover of the preceding financial year, with a maximum fee not exceeding EUR 20 million. It is important to underline that organizations are subject to the regulation if they elaborate and control user data of EU residents, regardless of where they are located.

The key aspect of GDPR, that is also the main motivation of this work of thesis, is contained in Article 33-34. In these articles, in fact, it is stated that the data processor (the entity that process the personal data on behalf of the data controller), often a cloud service provider, has to notify the data controller without undue delay after becoming aware of a personal data breach. Instead, the notice to data subjects (the physical person, the final user) is not required if the data controller has implemented appropriate technical and organisational protection measures that render personal data unintelligible to not authorised accesses, such as encryption. However, adding encryption to data leads to changes in length and type. For example, if we take a social security number (which is composed only by digits and it is 9 bytes long) and we encrypt it using a block cipher like AES-128-CBC, we will obtain an hexadecimal value that is 128 bits long. This formatting problems will break any existing application expecting a social security number 9-digit long.

Furthermore, data encrypted with an encryption algorithm in CBC mode changes its value at every new encryption; this happens because the random seed value is different for each encryption operation. As a consequence, we cannot use data encrypted using the CBC mode as a unique key in order to identify a row in a database. A solution to all of these problem is represented by Format Preserving Encryption (FPE).

1.2 Intro FPE

Format preserving encryption refers to a set of techniques for encrypting data such that the ciphertext has the same format as the plaintext. The form of the text can vary according to use and the application. For example, encrypting a 16-digit credit card number produces a ciphertext which is another 16-digit number. Similarly, encrypting an English word produces a ciphertext having the same number of English characters.

FPE is a powerful data protection technology and is currently becoming the *de-facto* standard across the industry. The *format-preserving* property of FPE has several benefits, especially for legacy systems. A legacy system is an old technology, application or computer system that is still in use by an organisation. The choice to keep an out of date system may be driven by many reasons such as: backward compatibility, economic return or simply because the system works well.

FPE can enable a simpler migration when encryption is added to legacy systems and databases, avoiding violations in existing format constraints, as well as extensive redesign and refactoring of applications and business processes. Thus, the power of FPE can be summarized in two principal advantages:

- **minimal database schema impact**: FPE facilitates retrofitting encryption technology to existing devices or software where conventional encryption modes would not be feasible;
- **minimal data storage impact**: Since length preservation is a direct consequence of the FPE definition, enterprises do not have to worry about additional storage usage, unlike conventional encryption methods, which typically expand the data size.

The widespread interest in this new cryptography tool is witnessed by several proposal of FPE scheme. In particular, practical FPE has been used and deployed by various companies, such as: *Voltage Security*, *Verifone*, *Ingenico*, *Cisco* and major credit-card payment organizations. On the other hand, the need for standardized methods leads to the first draft publication by NIST in 2013 [2]. The National Institute of Standards and Technology (NIST) was founded in 1901 (named National Bureau of Standards until 1988) and is an agency of the U.S. Department of Commerce. NIST's mission is to promote U.S. innovation and industrial competitiveness by advancing standards and technology in ways that enhance economic security and improve quality of life.

1.3 Outline of this Thesis

This work of thesis was born from the collaboration with the researches of *Aruba Software Factory SRL* and it is organized as follows.

In Chapter 2 the State-of-the-Art literature is given as an overview of the FPE methods. The main approaches to FPE are discussed, covering NIST standards, practical implementations and other schemes. In particular, the mathematical and cryptographic foundations are highlighted. Then, Chapter 3 analyses the existing attacks against FPE schemes, focusing the attention on the types of techniques used and on the efficiency of the attack algorithms. In Chapter 4 a proof-of-concept is presented, covering the various implementation details. Finally, the main conclusions and practical implications are summarized in Chapter 5.

Chapter 2

State of the art

2.1 Modes of operation

A block cipher mode of operation is an algorithm that is used in the scope of a specific symmetric key block cipher algorithm in order to provide an information service, such as confidentiality or authentication.

The reason behind the definition of a block cipher mode of operation comes from the need to have a working block cipher even if the input data of the block cipher is different from the algorithm's block size.

The very first modes of operation were published in FIPS PUB 81 [3] in 1980 by the National Institute of Standards and Technology (NIST). This publication included four modes of operation (ECB, CBC, OFB and CFB), which were all originally suited for Data Encryption Standard (DES) block cipher, that was later withdrawn in 2005.

After this initial publication, NIST starts considering proposals for new modes of operation. Proposals are evaluated by the NIST and when a mode of operation is approved it is published in the 800-38 series of Special Publication (SP 800-38). Currently NIST approved eight confidentiality modes (ECB, CBC, OCB, CFB, CTR, XTS-AES, FF1 and FF3), one authentication mode (CMAC) and five combined modes for confidentiality and authentication (CCM, GCM, KW, KWP and TKW), for a total of fourteen modes.

2.2 Format Preserving Encryption mode of operation

Methods for Format Preserving Encryption were published by NIST in the seventh part of the 800-38 series [4]. The modes for encryption defined in the previous six parts are all transformations on binary data, that is, the inputs and the outputs of the modes are bit strings. For sequences of non-binary symbols there is no natural way for these modes to produce encrypted data that has the same format.

A Format Preserving Encryption, given any finite set of symbols, transforms data that is formatted as a sequence of the symbols in such a way that the encrypted form of the data has the same format, including the length, as the original data. A typical example is a Social Security Number (SSN), that consists of nine decimal numbers, consequently the SSN is an integer less then one billion (1,000,000,000). If we use a non-FPE mode to encrypt an SSN number, we have to convert it to a bit string as input for that mode; then we apply the mode and we obtain an output that is again a bit string. When the bit string is converted back to an integer, it can be the case that the integer is greater than one billion, which is too long for an SSN and breaks the format defined.

FPE is useful especially for data at rest in database applications, where changes to the length or format of data fields are not supported. In fact a lot of companies, working in the finance

word, as well as in the healthcare or government, have legacy applications (old-fashioned and expensive applications) requiring a certain format of data. In order to account for the new format the application should be redone from scratch, spending time and money. FPE allows a drop-in replacement of plaintext with the respective ciphertext in legacy applications.

Another advantage of FPE is that it helps in recognizing data encrypted. As an example we can take a credit card number (CCN), typically composed of 16 integer; the number obtained after encryption using FPE will consist again of 16 integer, so in the contest of a database we will know that we are dealing with a CCN. This aspect can be useful if we have sensitive data protected by the GDPR legislation and we want to perform some statistical researches on these data. Furthermore FPE, as already mentioned, in contrast with other modes of operation, gives the possibility to use encrypted data as a unique key to identify a row in a database.

2.3 FPE history

The origins of the FPE problem go back in 1981, when the US National Bureau of Standards (which later became NIST) published FIPS PUB 74 [5], an appendix describing an approach for enciphering arbitrary strings over an arbitrary alphabet. Afterwards, in 1997, Brightwell and Smith were the first authors to describe more generally the FPE problem, calling it *datatype-preserving encryption* [6]. Specifically, they wanted to encrypt database entries of some particular data-type without disrupting that data-type.

In 2002 the cryptographers John Black and Phillip Rogaway published *"Cipher with Arbitrary Finite Domains"* [7], which can be considered, without any doubt, the first cornerstone for FPE. In this publication they formalized three different FPE methods, providing a provable security investigation. They have shown that each of these techniques is as secure as the block cipher that is used to construct it. The authors give no general definition for FPE, but they clearly point out that ciphers with domain $\mathbb{Z}_N$ can be used to construct schemes with other domains, like the set of valid CCNs of a given length. In 2008 Terence Spies, at that time *Voltage Security*'s CTO (later acquired by *Hewlett-Packard*), proposed an encryption mode called FFSEM [8], that has been accepted for considerations by NIST. FFSEM combines the *cycle-walking* (proposed by Black and Rogaway) with an AES-based balanced Feistel network. In 2009 Spies coined the term *Format-Preserving Encryption*, present for the first time within a personal communication. In those years *Voltage Security* and *Semtek* (later acquired by *Verifone*) have been two of the most active companies in advertising FPE and explaining its utility.

On February 2010, the FFX mode was submitted to NIST by Bellare, Rogaway and Spies [9]. On September 2010 this mode was further expanded with an addendum defining the scheme FFX[Radix] [10]. In the revision done by NIST, the FFX[Radix] scheme was renamed **FF1**. On April 2010, the BPS mechanism for FPE, named after its designers Brier, Peyrin and Stern was submitted to NIST [11]. NIST took the internal block cipher of BPS, called BC, to create the standard of the **FF3** algorithm (FF3 is the equivalent to a BC algorithm instantiated with a 128-bit block cipher). Instead, the full BPS mode was not approved in NIST SP 800-38G. On May 2011, the VAES3 FPE scheme was submitted by Joachim Vance (Verifone Systems Inc.) [12]; in this publication VAES3 is proposed as a set of parameters for FFX. NIST has renamed the VAES3 scheme into **FF2**.

On July 2013 NIST published these three methods in the Draft SP 800-38G and started a public review period. During this routine consultation, NIST was advised by the National Security Agency (NSA) that the FF2 mode in the draft did not provide the expected 128 bits of security strength for some use cases. On April 2015, the NIST cryptographers Dworkin and Perlner confirmed the assessment of the NSA with an analysis of FF2 [13], describing a theoretical chosen-plaintext attack that, by the way, was not feasible in practice. The company indicated the intention to submit a revised version of FF2, in order to meet NIST's security requirements for other potential applications. An extension of VAES3 was submitted for NIST's consideration on November 2015 by Vance and Bellare and called DFF (Delegatable Feistel-based Format-preserving encryption mode) [14].

On March 2016 NIST published a new version of SP 800-38G, where FF1 and FF3 were specified and approved as methods for FPE, while FF2 was removed. Since the release of this publication, many researchers have identified vulnerabilities in FF1 and FF3 when the number of possible inputs, the domain size, is sufficiently small. An important analysis was published in 2017 by Durak and Vaudenay, describing an attack on FF3 [15]. In response to the attack NIST announced to either modify the FF3 specification, reducing, as suggested by the two researchers, the size of the tweak from 64 bits to 48 bits, or to completely withdraw FF3. On February 2019 the Draft SP 800-38G Revision 1 was published [16]. In this new draft Durak and Vaudenay cooperated with NIST's researchers; together they decided to reduce the tweak parameter to 56 bits. The revised FF3 was recalled FF3-1.

In SP 800-38G (2016) the domain size for FF1 and FF3 was required to be at least one hundred (100) and recommended to be at least one million (1,000,000). However, a new attack was published in *"The Curse of Small Domains: New Attacks on Format-Preserving Encryption"* (2018) by Hoang, Tessaro and Trieu [17], inspired by an earlier work of Hoang and Tessaro with Mihir Bellare: *"Message-Recovery Attacks on Feistel-Based FPE"* (2016) [18]. In the former paper some experiments confirmed the correctness of the attack for tiny domains. Consequently, the recommendation was strengthened to a new requirement: the minimum domain size for FF1 and FF3-1 in Draft 800-38G Revision 1 (2019) was set to one million (1,000,000).

2.4 Proposed FPE schemes

2.4.1 Black and Rogaway methods

The work *"Cipher with Arbitrary Finite Domains"* [7] by Black and Rogaway describes three methods:

- Prefix Cipher;
- Cycle-Walking Cipher;
- Generalized-Feistel Cipher.

Black and Rogaway proved that each of these three methods is as secure as the block cipher used to construct them; thus, if the AES is used to create the FPE algorithm, an adversary can break the FPE algorithm if and only if he can break the AES algorithm too.

The *Prefix Cipher* method fixes some integer k and works on M, the set $[0, k-1]$. His goal is to build a cipher with domain M. It assigns a pseudorandom weight to each integer, then sort by weight. The weights are defined by applying an existing block cipher to each integer. This method is useful only for small values of k, because the cost in time and space due to the initialization step is $O(k)$, while generally enciphering and deciphering are constant-time operations. A significant drawback is that, although the initialization is a one-time cost operation, it results in a table of sensitive data which must be stored somewhere. The ciphering and deciphering algorithms are given in Figure 2.1.

The *Cycle-Walking Cipher* method uses a block cipher whose domain is larger than M, where the points out-of-range are handled by repeatedly applying the block cipher until the result is within M. More precisely let N be the smallest power of 2 larger or equal to k and n be $log(N)$, the underlying cipher works on blocks of n-bit. The recursion is guaranteed to terminate, because the block cipher is supposed to be ideal, which is in fact a random permutation. If we apply the block cipher enough times we must eventually arrive back at some point in M, even at the initial point itself. This method is quite feasible if k is just smaller than some power of 2, because in this case the number of points we have to traverse during any encipherment is correspondingly small. Instead, in the worst case scenario where k is one larger than a power of 2, the algorithm might require k calls to the underlying block cipher to encipher just one point. There is also another drawback: if the block cipher is of a fixed size, such as AES, this is a severe restriction on the

Algorithm Init_Px_K
for $j \leftarrow 0$ **to** $k-1$ **do**
  $I_j \leftarrow E_K(j)$
for $j \leftarrow 0$ **to** $k-1$ **do**
  $J_j \leftarrow \mathrm{Ord}(I_j, \{I_j\}_{j\in[0,k-1]})$
for $j \leftarrow 0$ **to** $k-1$ **do**
  $L_{J_j} \leftarrow j$

Algorithm $\mathrm{Px}_K(m)$ **return** J_m

Algorithm $\mathrm{Px}_K^{-1}(m)$ **return** L_m

Figure 2.1: Algorithms for the Prefix Cipher. On the left the initialization algorithms. On the right ciphering/deciphering algorithms.

Algorithm $\mathrm{Cy}_K(m)$
$c \leftarrow \mathrm{E}_K(m)$
if $c \in M$ **then return** c
else return $\mathrm{Cy}_K(c)$

Algorithm $\mathrm{Cy}_K^{-1}(m)$
$c \leftarrow \mathrm{E}_K^{-1}(m)$
if $c \in M$ **then return** c
else return $\mathrm{Cy}_K^{-1}(c)$

Figure 2.2: Algorithms for the Cycle-Walking Cipher.

sizes of M for which this method is practical. The ciphering and deciphering algorithms are given in Figure 2.2.

The *Generalized-Feistel Cipher* method consists in decomposing all the numbers in M into pairs of "similarly sized" numbers and then apply the well-known Feistel construction to produce a cipher. The cipher $Fe[r, a, b]$ takes as input r, the number of round used in the Feistel network and two positive numbers a and b such that $ab \geq k$. The two values a and b are used to decompose any $m \in M$ into two numbers to give as inputs into the network. Within the network r random function $F_1, \ldots, F_r$ are used, whose ranges contain M. In the case that using the Feistel construction results in a number not in M, it is possible to continue to iterate in the same way of the *Cycle-Walking Cipher*. This method is an adaptation of Luby-Rackoff construction (with the related security proof) and it shows that, when the attacker is limited to access less than $Q = 2^{min\{L,R\}/2}$ plaintext/ciphertext pairs, he has not enough information to distinguish this construction from a random permutation with domain M. The Generalized-Feistel Cipher can be quite efficient, even if the proven bounds are weak when the message space M is small. The ciphering and deciphering algorithms are given in Figure 2.3.

2.4.2 FFSEM

The *Generalized-Feistel* method mentioned in Section 2.4.1 combines the Luby-Rackoff construction with the technique of *"cycle-following"*, allowing the encryption in any domain, independently of the size. In their work Black and Rogaway showed that the security of the FPE scheme can be reduced to the security of the base block cipher (i.e. AES) when an attacker has less than $2^{m/2}$ plaintext/ciphertext pairs. This reduction is adequate for the encryption of large sets (because the number of pairs would be too large), but provides an insufficient level of confidence for some cases, such as credit card numbers where $2^{2m} \approx 10^{16}$. Jacques Patarin proved in [19] that an extension of Black and Rogaway's method (with a increased number of rounds) reduces to the underlying block cipher for a computationally unbounded attacker with less than 2^m plaintext/ciphertext pairs. This extension allows an encryption of sets around the size of the CCN domain.

The Feistel Finite Set Encryption Mode (FFSEM) [8], proposed by Terence Spies of *Voltage Security*, is a concrete instantiation of the Black and Rogaway method with an increased round count, using AES as the underlying block cipher. This mode is used to encrypt items smaller than the block size of AES to ciphertext of the same size.

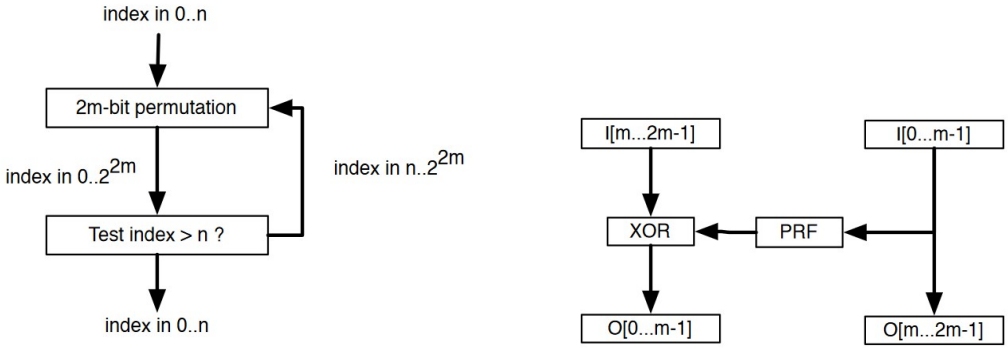
Algorithm $\mathrm{Fe}[r,a,b]_K(m)$
$c \leftarrow \mathrm{fe}[r,a,b]_K(m)$
if $c \in M$ **then return** c
else return $\mathrm{Fe}[r,a,b]_K(c)$

Algorithm $\mathrm{fe}[r,a,b]_K(m)$
$L \leftarrow m \bmod a$; $R \leftarrow \lfloor m/a \rfloor$
for $j \leftarrow 1$ **to** r **do**
  if (j is odd) **then**
    $tmp \leftarrow (L + F_j(R)) \bmod a$
  else
    $tmp \leftarrow (L + F_j(R)) \bmod b$
  $L \leftarrow R$; $R \leftarrow tmp$
if (r is odd) **then return** $aL + R$
else return $aR + L$

Algorithm $\mathrm{Fe}[r,a,b]_K^{-1}(m)$
$c \leftarrow \mathrm{fe}[r,a,b]_K^{-1}(m)$
if $c \in M$ **then return** c
else return $\mathrm{Fe}[r,a,b]_K^{-1}(c)$

Algorithm $\mathrm{fe}[r,a,b]_K^{-1}(m)$
if (r is odd) **then**
  $R \leftarrow m \bmod a$; $L \leftarrow \lfloor m/a \rfloor$
else
  $L \leftarrow m \bmod a$; $R \leftarrow \lfloor m/a \rfloor$
for $j \leftarrow r$ **to** 1 **do**
  if (j is odd) **then**
    $tmp \leftarrow (R - F_j(L)) \bmod a$
  else
    $tmp \leftarrow (R - F_j(L)) \bmod b$
  $R \leftarrow L$; $L \leftarrow tmp$
return $aR + L$

Figure 2.3: Algorithms for the Generalized-Feistel Cipher.



(a) Overall cycle structure

(b) A single Feistel round

Figure 2.4: Two components of FFSEM method, as shown in [8].

FFSEM consists of two basic components, which are represented in Figure 2.4:

- **Cycle Following** used to encrypt sets of approximately the same size as a given cipher's block size;
- **Feistel Method** used to produce a block cipher of approximately the right size.

The cycle following structure (Figure 2.4a) uses a q-bit block cipher to encrypt/decrypt sets of size n where $n < 2^q$. It is important to notice that since the underlying block cipher is a pseudo-random permutation (PRP) over the space 2^q, the probability that any given cycle will produce a valid value is $\frac{n}{2^q}$. If $n << 2^q$, the cycle following algorithm will have a poor average behaviour, thus it is necessary to start with a block cipher of approximately the right size. The Luby-Rackoff construction can be utilized to produce such a block cipher.

The Luby and Rackoff approach constructs a block cipher using a specified pseudo-random function (PRF) in a repeated Feistel network. The Feistel round, when repeated a certain number of times, yields a pseudo-random permutation. As shown in Figure 2.4b, a single Feistel round divides the input bit-vector into a right half and a left half, runs the right half through the PRF, XORs it with the left half and lastly swaps the right with the left. Since the L-R PRF operates

on half of the block size, it must be m bits wide for a cipher $2m$ bits wide. It was shown that is possible to use a truncated version of the base block cipher as PRF, provided that the block size of the L-R cipher is smaller than the base cipher. The round count is employed as a *tweak* in order to have a different PRF at each round, obtaining a "tweakable" PRF. The tweaking method will be explored in depth in the following sub-Section.

Summing up, FFSEM although enables encryption from a given domain back into the same one, it has many disadvantages. First of all it has a performance issue, because it requires multiple invocations of the block cipher to encrypt a single data item. Furthermore, due to the cycling construction, FFSEM gives also a non-deterministic performance; in fact different data items can take more or less time to encrypt/decrypt, exposing the scheme to possible timing attacks. Finally, FFSEM provides only encryption, not integrity or authentication.

As a general guideline, Spies gave the following suggestions for parameters:

- for a message domain $n > 40$, six rounds should be sufficient (which provide security bounds shown by Patarin);
- for a message domain in the bit range $32 < n < 40$, additional rounds should be used to compensate for the small number of plaintext/ciphertext pairs required for the theoretic attack.

Tweakable block cipher

The concept of *tweak* was formalized for the first time by Liskov, Rivest and Wagner in [20].

A conventional block cipher takes two inputs, a *key* $K \in \{0,1\}^k$ and a *plaintext* (or *message*) $M \in \{0,1\}^n$, and produces a *ciphertext* $C \in \{0,1\}^n$. The signature for a conventional block cipher is thus: $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. Block ciphers are inherently deterministic: every encryption of a given message with a given key will be the same. However, many applications require different instances of the block cipher in order to prevent attacks that permute blocks of the input. The goal is to keep the same key for an efficiency reason, while achieving variability of the output. An elegant solution is to redefine the primitive of the basic block cipher. This revised primitive, which already contains internally a notion of variability, is called a *tweakable block cipher* and it introduces a new second input, the "tweak". Consequently a tweakable block cipher takes three input, a *key* $K \in \{0,1\}^k$, a *tweak* $T \in \{0,1\}^t$ and a *plaintext* (or *message*) $M \in \{0,1\}^n$, and produces a *ciphertext* $C \in \{0,1\}^n$. The signature of a tweakable block cipher is: $\tilde{E} : \{0,1\}^k \times \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^n$.

The tweak serves much the same purpose that an initialization vector does for the CBC mode or that a nonce does for the OCB mode. In particular, changing the tweak should be less expensive than changing the encryption key, because in the last case a "key setup" operation is performed. Another crucial feature is that the tweak can be public, because the tweakable block cipher remains secure even if the tweak is available to an adversary. This concept of security can be expressed with the fact that each fixed setting of the tweak gives a different, apparently independent, family of standard block cipher encryption operators. Furthermore, there must be a clear distinction between the function of the key, which is to provide *uncertainty*, and the function of the tweak, which is to provide *variability*. The tweak is not employed to provide additional uncertainty.

With the introduction of the "tweak" input, various modes of operation are enabled, such as: Tweak Block Chaining (TBC), Tweak Chain Hash (TCH), Tweakable Authenticated Encryption (TAE). These modes are, in practice, the only "payoff" for using tweakable block ciphers.

2.4.3 FF1

FFX

The FFX mode of operation [9] was published in 2010 after a collaboration between researchers of the University of California, the *Semtek Innovative Solutions Corporation* and *Voltage Security*.

parameter	description
radix	The *radix*, a number radix ≥ 2 that determines the alphabet Chars = {0, ..., radix-1}. Plaintexts and ciphertexts are strings of characters from Chars.
Lengths	The set of *permitted message lengths*. For a plaintext to be encrypted, or for a ciphertext to be decrypted, its length must be in this set.
Keys	The *key space*, a finite non-empty set of binary strings.
Tweaks	The *tweak space*, a non-empty set of strings. Conceptually, different tweaks name unrelated encryption mappings.
addition	The *addition operation*, either 0 (characterwise addition) or 1 (blockwise addition). Determines the meaning of the operators $X \boxplus Y$ and $X \boxminus Y$ that add or subtract equal-length strings over the alphabet Chars = {0, ..., radix-1}.
method	The *Feistel method*, either 1 or 2. The value determines which of the two prominent Feistel variants will be used.
split(n)	The *imbalance*, a function that takes a permitted length $n \in$ Lengths and returns a number $1 \leq$ split(n) $\leq n/2$.
rnds(n)	The *number of rounds*, a function that takes a permitted length $n \in$ Lengths and returns an even number rnds(n).
F	The *round function*, a function that takes in a key $K \in$ Keys, a permitted length $n \in$ Lengths, a tweak $T \in$ Tweaks, a round number $i \in \{0, \ldots,$ rnds(n) $- 1\}$, and a string $B \in$ Chars*. It returns a string $\mathsf{F}_K(n, T, i, B) \in$ Chars*. If method $= 1$ or i is even then $\lvert B\rvert = n -$ split(n) and $\lvert\mathsf{F}_K(n, T, i, B)\rvert =$ split(n). If method $= 2$ and i is odd then $\lvert B\rvert =$ split(n) and $\lvert\mathsf{F}_K(n, T, i, B)\rvert = n -$ split(n)

Figure 2.5: Parameters of FFX.

Specifically the main contribution from Voltage Security was done done by Terence Spies, that is one of the author of this draft and of the FFSEM scheme discussed earlier.

The name FFX is meant to suggest *Format-preserving Feistel-based encryption*; the X reflects the possibility of having multiple instantiations, when different parameters are chosen. The "double F" also suggest that FFX is a direct extension of the FFSEM specification and replaces it. FFX is more general compared to FFSEM, adding in support for tweaks, non-binary alphabets and non-balanced splits.

Various cryptographic results have shown that FFX achieves goals including:

- non-adaptive message-recovery security;
- chosen-plaintext security;
- PRP-security against an adaptive chosen-ciphertext attack.

The encryption algorithm of FFX takes as input a key K, a plaintext X and a tweak T. The plaintext is taken over an arbitrary alphabets Chars and the supported length is $n = |X|$. The encryption function will produce deterministically a ciphertext $Y = \text{FFX.Encrypt}_K^T(X) \in \text{Chars}^n$. The decryption function instead recovers X from Y, $X = \text{FFX.Decrypt}_K^T(Y)$. The FFX mode is a customizable function, in fact it depends on a certain number of *parameters* which, once chosen, are held fixed for the lifetime of a given user-generated key.

Examples of FFX instantiations are used for binary strings of 8-128 bits and decimal strings of 4-36 digits, denoted respectively FFX-A2 and FFX-A10. Both the instantiations employ a round function derived from AES (the round function for FFX must be a good PRF).

The description of the parameters on which FFX depends is presented in the Figure 2.5.

The method parameter indicates if it is used an **unbalanced** (*method* $= 1$) Feistel network or an **alternating** (*method* $= 2$) Feistel network.

An illustration of the first four rounds of the FFX encryption, when the unbalanced method is employed, is reported in Figure 2.6.

The function $\boxplus$, present in the cipher above, takes a pair of equal-length strings and returns a string of the same length. The $\boxplus$ symbol can represent two different operations: a *characterwise addition* or a *blockwise addition*. For the characterwise addition the following property is observed: $a_1, \ldots, a_n \boxplus b_1, \ldots, b_n = c_1, \ldots, c_n$, where $c_i = (a_i + b_i) \bmod radix$. Instead, for the blockwise

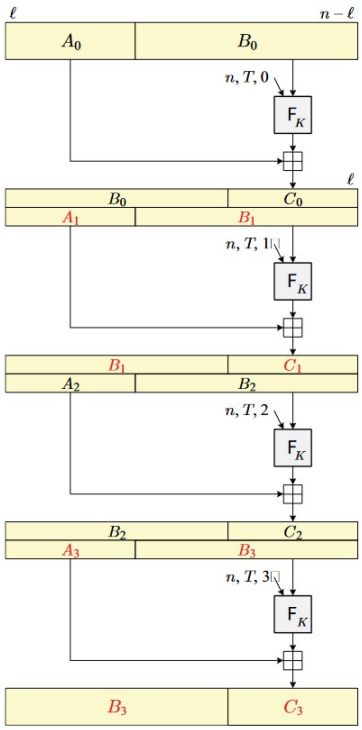


Figure 2.6: Graphical representation of the FFX encryption cipher, as shown in [9].

```
10 algorithm FFX.Encrypt(K, T, X)
11 if K ∉ Keys or T ∉ Tweaks or X ∉ Chars* or |X| ∉ Lengths then return ⊥
12 n ← |X|;  ℓ ← split(n);  r ← rnds(n)

20 if method = 1 then                    30 if method = 2 then
21 for i ← 0 to r − 1 do                 31 A ← X[1 .. ℓ];  B ← X[ℓ + 1 .. n]
22     A ← X[1 .. ℓ]; B ← X[ℓ + 1 .. n]  32 for i ← 0 to r − 1 do
23     C ← A ⊞ FK(n, T, i, B)            33     C ← A ⊞ FK(n, T, i, B)
24     X ← B ‖ C                         34     A ← B;  B ← C
25 return X                              35 return A ‖ B
26 end if                                36 end if
```

```
50 algorithm FFX.Decrypt(K, T, Y)
51 if K ∉ Keys or T ∉ Tweaks or Y ∉ Chars* or |Y| ∉ Lengths then return ⊥
52 n ← |Y|;  ℓ ← split(n);  r ← rnds(n)

60 if method = 1 then                    70 if method = 2 then
61 for i ← r − 1 downto 0 do             71 A ← Y[1 .. ℓ];  B ← Y[ℓ + 1 .. n]
62     B ← Y[1 .. n − ℓ];  C ← Y[n − ℓ + 1 .. n]  72 for i ← r − 1 downto 0 do
63     A ← C ⊟ FK(n, T, i, B)            73     C ← B;  B ← A
64     Y ← A ‖ B                         74     A ← C ⊟ FK(n, T, i, B)
65 return Y                              75 return A ‖ B
66 end if                                76 end if
```

Figure 2.7: FFX encryption and decryption pseudo-code.

addition we have that $c_1, \ldots, c_n$ is the unique string such that $\sum c_i \text{radix}^{n-i} = (\sum a_i \text{radix}^{n-i} + \sum b_i \text{radix}^n) \bmod radix$. The round function F_K must be constructed from a block cipher E or a hash function H. For the block cipher the AES algorithm is recommended, together with some options like CBC MAC or CMAC. When an hash function is used, the PRF construction could be based on HMAC.

The encryption and decryption function are defined in the pseudo-code shown below in Figure 2.7, where Chars = {0, 1, ..., radix} is the underlying alphabet and Lengths = {minlen, ..., maxlen} are the permitted message lengths.

The sets of parameters for the instantiations FFX-A2 and FFX-A10 presented before are available respectively in Figure 2.8 and 2.9. FFX-A2 works with a value of radix equal to 2, while FFX-A10 use radix = 10. Furthermore, FFX-A2 uses the characterwise addition, while

parameter	value	comment
radix	2	alphabet is Chars $= \{0,1\}$
Lengths	[minlen .. maxlen] where minlen $= 8$, maxlen $= 128$	permissible message lengths
Keys	$\{0,1\}^{128}$	128-bit AES keys
Tweaks	$\text{BYTE}^{\leq M}$ where $M = 2^{64}-1$	tweaks are arbitrary byte strings
addition	0	characterwise addition (xor)
method	2	alternating Feistel
split (n)	$\lfloor n/2 \rfloor$	maximally balanced Feistel
rnds (n)	$\begin{cases} 12 & \text{if } 32 \leq n \leq 128, \\ 18 & \text{if } 20 \leq n \leq 31, \\ 24 & \text{if } 14 \leq n \leq 19, \\ 30 & \text{if } 10 \leq n \leq 13, \text{ and} \\ 36 & \text{if } 8 \leq n \leq 9 \end{cases}$	from entropy-based heuristic
F	defined below	AES-based round function

100 **algorithm** $\mathsf{F}_K(n,T,i,B)$
101 $\text{VERS} \leftarrow 1$; $t \leftarrow |T|_8$
102 $P \leftarrow [\text{VERS}]^2 \parallel [\text{method}]^1 \parallel [\text{addition}]^1 \parallel [\text{radix}]^1 \parallel [n]^1 \parallel [\text{split}(n)]^1 \parallel [\text{rnds}(n)]^1 \parallel [t]^8$
103 $Q \leftarrow T \parallel [0]^{-t-9 \bmod 16} \parallel [i]^1 \parallel 0^{64-|B|} \parallel B$
104 $Y \leftarrow \text{CBC-MAC}_K(P \parallel Q)$
105 **if** $\text{EVEN}(i)$ **then** $m \leftarrow \text{split}(n)$ **else** $m \leftarrow n - \text{split}(n)$
106 **return** $Y[129-m \,..\, 128]$

Figure 2.8: FFX-A2 parameters.

parameter	value	comment
radix	10	alphabet is Chars $= \{0,1,2,\ldots,8,9\}$
Lengths	[minlen .. maxlen] where minlen $= 4$, maxlen $= 36$	permitted message lengths
Keys	$\{0,1\}^{128}$	128-bit AES keys
Tweaks	$\text{BYTE}^{\leq M}$ where $M = 2^{64}-1$	tweaks are arbitrary byte strings
addition	1	blockwise addition
method	2	alternating Feistel
split (n)	$\lfloor n/2 \rfloor$	maximally balanced Feistel
rnds (n)	$\begin{cases} 12 & \text{if } 10 \leq n \leq 36, \\ 18 & \text{if } 6 \leq n \leq 9, \text{ and} \\ 24 & \text{if } 4 \leq n \leq 5 \end{cases}$	from entropy-based heuristic
F	given below	AES-based round function

200 **algorithm** $\mathsf{F}_K(n,T,i,B)$
201 $\text{VERS} \leftarrow 1$; $t \leftarrow |T|_8$
202 $P \leftarrow [\text{VERS}]^2 \parallel [\text{method}]^1 \parallel [\text{addition}]^1 \parallel [\text{radix}]^1 \parallel [n]^1 \parallel [\text{split}(n)]^1 \parallel [\text{rnds}(n)]^1 \parallel [t]^8$
203 $Q \leftarrow T \parallel [0]^{-t-9 \bmod 16} \parallel [i]^1 \parallel [\text{NUM}_{10}(B)]^8$
204 $Y \leftarrow \text{CBC-MAC}_K(P \parallel Q)$
205 $Y' \leftarrow Y[1 \,..\, 64]$; $Y'' \leftarrow Y[65 \,..\, 128]$
206 $y' \leftarrow \text{NUM}_2(Y')$; $y'' \leftarrow \text{NUM}_2(Y'')$
207 **if** $\text{EVEN}(i)$ **then** $m \leftarrow \text{split}(n)$ **else** $m \leftarrow n - \text{split}(n)$
208 **if** $m \leq 9$ **then** $z \leftarrow y'' \bmod 10^m$
209 **else** $z \leftarrow (y' \bmod 10^{m-9}) \cdot 10^9 + (y'' \bmod 10^9)$
210 **return** $\text{STR}_{10}^m(z)$

Figure 2.9: FFX-A10 parameters.

FFX-A10 uses the blockwise addition. The number of rounds recommended is different in the two instantiation and it depends on the message length. This number comes from some entropy-based heuristics.

FFX[radix]

The FF1 mode, formalized by NIST in 2013, is nothing but a shortcut for FFX[radix]. FFX[radix] [10] was published on September 2010 as an addendum to the previous FFX mode by the same authors. The FFX scheme is expanded, allowing to use any possible value for the parameter radix; FFX[radix], in fact, can be seen as an instantiated version of FFX. It also enlarges the allowed message lengths, permitting arbitrary strings to be enciphered. The bracketed value compactly names an FFX parameter collection. Now the number of rounds is made constant, rather than depending on the message length n. It is essential to underline that FFX[2] and FFX[10] modes do not coincide with FFX-A2 and FFX-A10 modes.

FFX[radix] takes advantage of an AES-based balanced Feistel network. If the message length is odd, an alternating, maximally-balanced Feistel scheme is used instead.

radix	a number radix $\in [2 \,..\, 2^{16}]$	alphabet is Chars $= \{0, 1, \ldots,$ radix $- 1\}$
Lengths	[minlen .. maxlen] where minlen $= 2$ if radix ≥ 10 and minlen $= 8$ otherwise; and maxlen $= 2^{32} - 1$.	permitted message lengths
Keys	$\{0,1\}^{128}$	128-bit AES keys
Tweaks	$\text{BYTE}^{\leq \text{maxlen}}$ where maxlen $= 2^{32} - 1$	tweaks are arbitrary byte strings
addition	1	blockwise addition
method	2	alternating Feistel
split (n)	$\lfloor n/2 \rfloor$	maximally balanced Feistel
rnds (n)	10	number of rounds
F	given below	AES-based round function

algorithm $\mathsf{F}_K(n, T, i, B)$
vers $\leftarrow 1$; $t \leftarrow |T|_8$; $\beta \leftarrow \lceil n/2 \rceil$; $b \leftarrow \lceil \lceil \beta \log_2(\text{radix}) \rceil / 8 \rceil$; $d \leftarrow 4\lceil b/4 \rceil$
if EVEN(i) **then** $m \leftarrow \lfloor n/2 \rfloor$ **else** $m \leftarrow \lceil n/2 \rceil$
$P \leftarrow [\text{vers}]^1 \,\|\, [\text{method}]^1 \,\|\, [\text{addition}]^1 \,\|\, [\text{radix}]^3 \,\|\, [\text{rnds}(n)]^1 \,\|\, [\text{split}(n)]^1 \,\|\, [n]^4 \,\|\, [t]^4$
$Q \leftarrow T \,\|\, [0]^{(-t-b-1) \bmod 16} \,\|\, [i]^1 \,\|\, [\text{NUM}_{\text{radix}}(B)]^b$
$Y \leftarrow \text{CBC-MAC}_K(P \,\|\, Q)$
$Y \leftarrow$ first $d+4$ bytes of $(Y \,\|\, \text{AES}_K(Y \oplus [1]^{16}) \,\|\, \text{AES}_K(Y \oplus [2]^{16}) \,\|\, \text{AES}_K(Y \oplus [3]^{16}) \cdots)$
$y \leftarrow \text{NUM}_2(Y)$
$z \leftarrow y \bmod \text{radix}^m$
return $\text{STR}^m_{\text{radix}}(z)$

Figure 2.10: FFX[radix] parameters.

algorithm FFX.Encrypt(K, T, X)
if $K \notin$ Keys **or** $T \notin$ Tweaks **or**
  $X \notin$ Chars* **or** $|X| \notin$ Lengths
  then return $\perp$
$n \leftarrow |X|$; $\ell \leftarrow \text{split}(n)$; $r \leftarrow \text{rnds}(n)$
$A \leftarrow X[1 \,..\, \ell]$; $B \leftarrow X[\ell + 1 \,..\, n]$
for $i \leftarrow 0$ **to** $r - 1$ **do**
  $C \leftarrow A \boxplus \mathsf{F}_K(n, T, i, B)$
  $A \leftarrow B$; $B \leftarrow C$
return $A \,\|\, B$

algorithm FFX.Decrypt(K, T, Y)
if $K \notin$ Keys **or** $T \notin$ Tweaks **or**
  $Y \notin$ Chars* **or** $|Y| \notin$ Lengths
  then return $\perp$
$n \leftarrow |Y|$; $\ell \leftarrow \text{split}(n)$; $r \leftarrow \text{rnds}(n)$
$A \leftarrow Y[1 \,..\, \ell]$; $B \leftarrow Y[\ell + 1 \,..\, n]$
for $i \leftarrow r - 1$ **downto** 0 **do**
  $C \leftarrow B$; $B \leftarrow A$
  $A \leftarrow C \boxminus \mathsf{F}_K(n, T, i, B)$
return $A \,\|\, B$

Figure 2.11: FFX[radix] encryption and decryption functions.

FF1 is defined with the parameters present in Figure 2.10. It can be noted that the number of *rounds* is fixed to 10, while in FFX-A2 and FFX-A10 the number of rounds increases as messages get shorter. The cost for using so many rounds was so high, so the authors decided to fix it. This simplified choice leave some margin of safety as well.

Below, in Figure 2.11, the functions for encryption and decryption.

2.4.4 FF3

The BPS mode of operation [11] is built upon two basic components:

- an internal length-limited block cipher (which itself uses an internal function such as AES);
- a mode of operation in order to handle long strings.

BPS is a Feistel based design and consists of 8 rounds, so it is faster than FF1 which has 10 rounds. The internal cipher is called BC and it is instantiated according to the cardinality s of the character set and the block length b of the cipher we are building. The expression

$$Y = BC_{F,s,b,w}(X, K, T)$$

denote the w-round encryption of a s-integer string X of length b, with key K and the 64-bit tweak value T.

The w rounds of the internal cipher BC are simple Feistel-like rounds, each of them update the right or left branch in turn. The left and right branch value after application of round i are

Algorithm $\mathrm{BC}_{F,s,b,w}(X, K, T)$
$T_R = T \bmod 2^{32}$;
$T_L = (T - T_R)/2^{32}$;
$l = \lceil b/2 \rceil; r = \lfloor b/2 \rfloor$;
$L_0 = X[0].s^0 + X[1].s^1 + \ldots + X[l-1].s^{l-1}$;
$R_0 = X[l].s^0 + X[l+1].s^1 + \ldots + X[l+r-1].s^{r-1}$;
for $i = 0$ **to** $w - 1$ **do**
  if (i is even) **then** $L_{i+1} = L_i \boxplus F_K((T_R \oplus i).2^{f-32} + R_i) \bmod s^l; R_{i+1} = R_i$;
  else $R_{i+1} = R_i \boxplus F_K((T_L \oplus i).2^{f-32} + L_i) \bmod s^r; L_{i+1} = L_i$;
for $i = 0$ **to** $l - 1$ **do**
  $Y[i] = L_w \bmod s; L_w = (L_w - Y[i])/s$;
for $i = 0$ **to** $r - 1$ **do**
  $Y[i + l] = R_w \bmod s; R_w = (R_w - Y[i + l]/s$;
return Y;

Figure 2.12: $\mathrm{BC}_{F,s,b,w}(X, K, T)$ encryption.

denoted by L_i and R_i respectively, and are initialized with X_L and X_R respectively:

$$L_0 = X_L[0].s^0 + X_L[1].s^1 + \ldots + X_L[l-1].s^{l-1}$$

$$R_0 = X_R[0].s^0 + X_R[1].s^1 + \ldots + X_R[r-1].s^{r-1}$$

When the encryption process BC is instantiated with a block cipher E, for each $0 \leq i \leq w$ we apply, for the left branch, the round function:

$$L_{i+1} = L_i \boxplus E_K((T_R \oplus i).2^{f-32} + R_i) (\bmod\ s^l), \text{if } i \text{ is } \textit{even}$$

$$L_{i+1} = L_i, \text{if } i \text{ is } \textit{odd}$$

For the right branch, we apply:

$$R_{i+1} = R_i, \text{if } i \text{ is } \textit{even}$$

$$R_{i+1} = R_i \boxplus E_K((T_L \oplus i).2^{f-32} + L_i) (\bmod\ s^r), \text{if } i \text{ is } \textit{odd}$$

Finally, the output string Y is the concatenation of Y_L and Y_R, i.e. $Y = Y_L || Y_R$ with Y_L and Y_R built by decomposing L_w and R_w into the s basis:

$$Y_L[0].s^0 + Y_L[1].s^1 + \ldots + Y_L[l-1].s^{l-1} = L_w$$

$$Y_R[0].s^0 + Y_R[1].s^1 + \ldots + Y_R[r-1].s^{r-1} = R_w$$

In Figure 2.12 the encryption algorithm for the internal cipher BC. In Figure 2.13, instead, there is a graphical representation of 2 rounds of the encryption BC, where it is clear the turning mechanism at each round. The specular analysis can be made to obtain the decryption function BC^{-1} and formulas.

With the internal encryption routine BC it is possible to cipher from 2 to $max_b = 2 \times log_s(2^{f-32})$ s-integer with one call. In order to cipher larger input strings an operating mode on BC it is needed. The operating mode proposed in the BPS mode is simple and efficient, similar to the well known Cipher-Block Chaining mode (CBC mode) with an IV set to 0. A counter is incorporated on the tweak input; more precisely a 16-bit counter will be XORed on the 16 most significant bits of both the right and left 32-bit tweak words T_L and T_R.

An example of operating mode encryption with $s = 10$, meaning that it is working with digits, and $len = 3.max_b + 2$ is shown in Figure 2.14. Here $u = 2^{16} + 2^{48}$.

In the SP 800-38G NIST specified that the new mode of operation FF3 is equivalent to the BPS-BC component of BPS, instantiated with 128-bit block cipher, while the full BPS mode (with its chaining mechanism for longer input strings) was not approved in the publication.

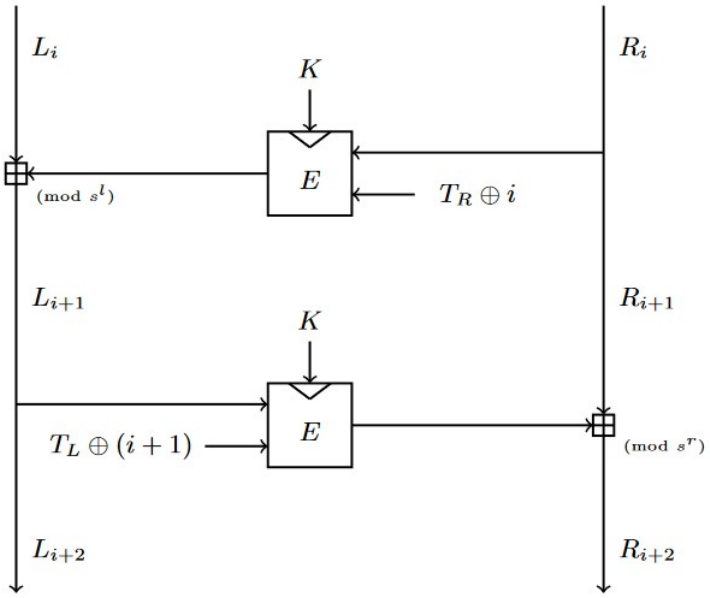


Figure 2.13: Two rounds of BC encryption, as shown in [11].

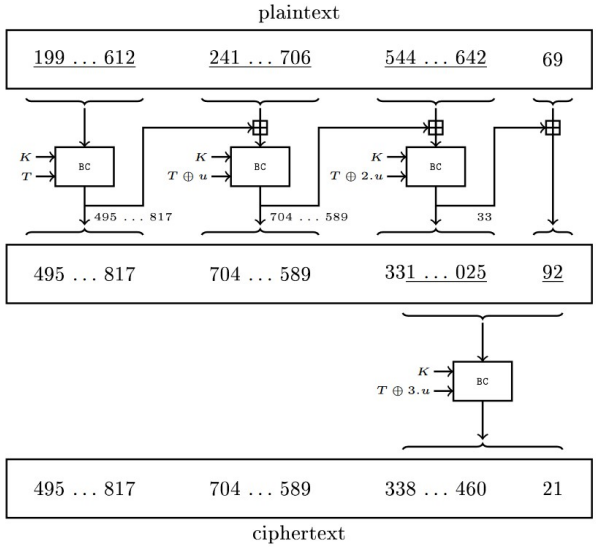


Figure 2.14: BC operating mode, as shown in [11].

2.4.5 FF2

In 2011 a new mode was proposed by Joachim Vance under the name VAES3 [12]. As for FF1, also VAES3 is a set of parameters for the general scheme of FFX. The acronym VAES stands for *variable-AES*, while the number "3" stands for *third-generation* FPE. VAES3 distinguishes itself from other schemes for its *delegation* feature: it associates to each key K and tweak T a **subkey** $J(K,T)$, and the ciphertext corresponding to K,T,X is a function of $J(K,T),X$ alone.

The subkey step *"enhances security and lengthens the lifetime of the key"*. The delegation feature is valuable because it limits the direct use of the base key. Many side-channel attacks are effective at recovering a key as they obtain more encryptions under it. With delegation, the loss is limited to the subkey; thus, even if encryption is compromised, it is only under a particular tweak, not under all tweaks as would happen if the base key is compromised. By limiting use of

Parameter	Value	Comment
radix	a number radix ∈ [2..2^8]	alphabet is Chars = ∑ = {0,1, ..., radix −1}
Lengths	[2,3, … , N(radix)] where N(radix) = 2 · ⌊120/lg (radix)⌋	Permitted message lengths. lg( ) denotes the logarithm in base 2
Keys	$\{0,1\}^{128}$	128-bit AES keys
Tweaks	a string over Chars = {0,1, ..., radix −1} of length 0 to a maximum length of ⌊104 / lg(radix)⌋	Tweaks are input as strings of radix converted to a byte string. The radix of the tweak is allowed to be different than that of the plaintext and ciphertext
addition	1	Blockwise addition
method	2	Alternating Feistel
split(n)	$\lfloor n/2 \rfloor$	Maximally balanced Feistel
rnds(n)	10	Number of rounds is fixed
F	The VAES round function is given below	AES-based round function

Figure 2.15: VAES3 parameters.

Algorithm $\mathrm{F}_K(n, T, i, B)$
$t \leftarrow |T|; i \leftarrow i + 1$
if EVEN(i) **then** $m \leftarrow \lfloor n/2 \rfloor$
else $m \leftarrow \lceil n/2 \rceil$

$P \leftarrow [\mathsf{radix}]^1 \,||\, [t]^1 \,||\, [n]^1 \,||\, [\mathrm{NUM}_{\mathsf{radix}}(T)]^{13}$
$J \leftarrow \mathrm{AES}(K, P)$

$Q \leftarrow [i]^1 || [\mathrm{NUM}_{\mathsf{radix}}(T)]^{15}$
$Y \leftarrow \mathrm{AES}(J, Q)$
$y \leftarrow \mathrm{NUM}_2(Y)$

$z \leftarrow y \bmod \mathsf{radix}^m$
return $\mathrm{STR}^m_{\mathsf{radix}}(z)$

Figure 2.16: VAES3 round function.

the base key, delegation also extends its lifetime, so that key changes, which are heavy to perform, are needed less frequently.

In Figure 2.15 there is the set of parameters of VAES3 and in Figure 2.16 there is the round function, while in Figure 2.17 the encryption/decryption algorithms of VAES3 are presented. Note that at line 34 there is the key-derivation step.

In 2015, in a paper of Dworkin and Perlner [13], it was shown that FF2 does not provide the expected 128-bits of security strength, hence removed from NIST recommended designs. The researchers indicate that FF2 is subject to an attack, more specifically a subkey attack. This subkey attack works like this: given encryptions of a single plaintext X under different tweaks $T_1, \ldots, T_Q$, the attack returns an index i and the subkey $J(K, T_i)$, in the time for around $2^{128}/Q$ evaluations of the block cipher. NIST indicates it as a theoretical attack, in fact this attack does not appear to compromise anticipated uses of FF2 for credit-card encryption and would appear unfeasible to mount in practice. By the way, it shows that FF2 with a 128-bit key cipher does not provide 128 bits of security for all use cases. NIST decided to remove FF2 from the final version of SP 800-38G, giving VeriFone the opportunity to propose a modification to FF2. The proposed modification is an FPE scheme called DFF[OFF].

```
10 algorithm FFX.Encrypt(K, T, X)
11 if K ∉ Keys or T ∉ Tweaks or
12   X ∉ Chars* or /X/ ∉ Lengths
13   then return ⊥
14 n ← |X|; ℓ ← split(n); r ← rnds(n)
15 A ← X[1 .. ℓ]; B ← X[ℓ+1 .. n]
16 for i ← 0 to r − 1 do
17   C ← A ⊞ F_K(n, T, i, B)
18   A ← B; B ← C
19 return A ‖ B
```

```
20 algorithm FFX.Decrypt(K, T, Y)
21 if K ∉ Keys or T ∉ Tweaks or
22   Y ∉ Chars* or |Y| ∉ Lengths
23   then return ⊥
24 n ← |Y|; ℓ ← split(n); r ← rnds(n)
25 A ← Y[1 .. ℓ]; B ← Y[ℓ+1 .. n]
26 for i ← r − 1 downto 0 do
27   C ← B; B ← A
28   A ← C ⊟ F_K(n, T, i, B)
29 return A ‖ B
```

Figure 2.17: VAES3 encryption/decryption algorithms.

Algorithm DFF[OFF].Enc(K, T, X)

$n \leftarrow \mathsf{len}(X)$; $t \leftarrow \mathsf{len}(T)$
$P \leftarrow [\mathsf{radix}]^1 \| [t]^1 \| [n]^1 \| [\mathsf{NUM}_{\mathsf{radix}}(T)]^{13}$
$T' \leftarrow [0]^3 \| [\mathsf{NUM}_{\mathsf{radix}}(T)]^{13}$
$J \leftarrow \mathsf{CIPH}(K, P)$; $J' \leftarrow \mathsf{OFF}(K, T)$
$Z \leftarrow \mathsf{FEISTEL}(J, J', X)$
Return Z

Algorithm $\mathsf{FEISTEL}(J, J', X)$

$u \leftarrow \lfloor n/2 \rfloor$; $v \leftarrow n - u$
$A \leftarrow X[1 \ldots u]$; $B \leftarrow X[u+1 \ldots n]$
For $i = 0, \ldots, 9$ **do**
  $Q \leftarrow [i]^1 \| [\mathsf{NUM}_{\mathsf{radix}}(B)]^{15}$
  $Y \leftarrow \mathsf{CIPH}(J, J' \oplus Q)$; $y \leftarrow \mathsf{NUM}_2(Y)$
  If i is even then $m \leftarrow u$ else $m \leftarrow v$
  $c \leftarrow (\mathsf{NUM}_{\mathsf{radix}}(A) + y) \bmod \mathsf{radix}^m$
  $C \leftarrow \mathsf{STR}^m_{\mathsf{radix}}(c)$
  $A \leftarrow B$; $B \leftarrow C$
Return $A \| B$

Algorithm DFF[OFF].Dec(K, T, Z)

$n \leftarrow \mathsf{len}(Z)$; $t \leftarrow \mathsf{len}(T)$
$P \leftarrow [\mathsf{radix}]^1 \| [t]^1 \| [n]^1 \| [\mathsf{NUM}_{\mathsf{radix}}(T)]^{13}$
$T' \leftarrow [0]^3 \| [\mathsf{NUM}_{\mathsf{radix}}(T)]^{13}$
$J \leftarrow \mathsf{CIPH}(K, P)$; $J' \leftarrow \mathsf{OFF}(K, T)$
$X \leftarrow \mathsf{FEISTEL}^{-1}(J, J', Z)$
Return X

Algorithm $\mathsf{FEISTEL}^{-1}(J, J', Z)$

$u \leftarrow \lfloor n/2 \rfloor$; $v \leftarrow n - u$
$A \leftarrow Z[1 \ldots u]$; $B \leftarrow Z[u+1 \ldots n]$
For $i = 9, \ldots, 0$ **do**
  $Q \leftarrow [i]^1 \| [\mathsf{NUM}_{\mathsf{radix}}(B)]^{15}$
  $Y \leftarrow \mathsf{CIPH}(J, J' \oplus Q)$; $y \leftarrow \mathsf{NUM}_2(Y)$
  If i is even then $m \leftarrow u$ else $m \leftarrow v$
  $c \leftarrow (\mathsf{NUM}_{\mathsf{radix}}(A) - y) \bmod \mathsf{radix}^m$
  $C \leftarrow \mathsf{STR}^m_{\mathsf{radix}}(c)$
  $A \leftarrow B$; $B \leftarrow C$
Return $A \| B$

Figure 2.18: DFF[OFF] FPE scheme.

Extension of FF2. DFF

The DFF[OFF] scheme [14] means *"delegatable FF"* parametrized by an *offset* function OFF. The function OFF takes the base key K and the tweak T to return the 128-bit binary string $\mathrm{OFF}(K, T)$. Therefore DFF specifies a family of FPE schemes, one for each choice of OFF. By making a particular choice of OFF we get a particular FPE scheme; FF2 is one of these. It corresponds to the trivial choice $\mathrm{OFF}(K, T) = 0^{128}$.

The proposed standard is obtained as DFF with a different choice of OFF, one that makes the offset depending on both K and T in an unpredictable way via CIPH, specifically $\mathrm{OFF}(K, T) = \mathrm{CIPH}(K, T')$, where T' is derived from T. DFF unifies the prior and new versions of delegatable FPE, both appearing as special cases. Regardless of the choice of OFF, the scheme DFF[OFF] maintains the delegatability feature. The choice of OFF affects the susceptibility to the subkey attack. The DFF[OFF] scheme is shown in Figure 2.18.

There are some choices of OFF for DFF[OFF]:

- OFF1, where $\mathrm{OFF}(K, T) = 0^{128}$; it is the FF2 scheme, subject to the subkey attack;
- OFF2, where $\mathrm{OFF}(K, T) = \mathrm{CIPH}(K, T')$; designed to resist to the subkey attack.

The first offset highlights the fact that the DFF can be seen as a generalization of FF2.

2.4.6 Other FPE schemes

In this section we want to briefly introduce other non-standard scheme, which will be later analysed in some of the attacks. The FEA scheme [21], by Lee et al., is a South-Korean model which has two variants, FEA-1 and FEA-2. Both algorithms are built from a family of dedicated tweakable block ciphers supporting various block bit-lengths. The FEA schemes were analysed in [22]. The FNR scheme (Flexible Naor and Reingold) [23] is a practical scheme introduced inside *Cisco Systems*, thought in the network context to cipher data formats in small domains, such as: IPv4, port numbers, MAC addresses. This scheme, as well as the DTP scheme [24], were analysed in [17] and demonstrated to be not secure.

2.4.7 FAST (2021)

The FAST construction was proposed by Durak et al. in [25]. FAST stays for *"Format-preserving Addition Substitution Transformation"*. The algorithm is a substitution-permutation network (SPN) based on random S-boxes. FAST can be used in two different modes, FPE mode or tokenization mode. Tokenization mode differs from FPE mode by having two specific inputs instead of one secret key:

- pre-generated random S-boxes as *stateless table secret*, which can be common to several domains;
- a key, which is used for domain separation.

SPN in other works

There exist some dedicated constructions based on SPN (for example DEAN 18 [26]), but they are designed only for fixed blocks of decimal digits. One difficulty with SPN-based FPE is that the internal S-boxes must be adapted to the specific format of the input data. The eSPF construction, by Somitra et al. [27], mixes the cycle walking technique with SPN based on S-boxes working on domain which is larger than the format. However this construction is not a pure SPN, is rather based on one-time-pad with a keystream generated from an SPN.

Tokenization

Tokenization introduces the notion of mapping cleartext values to substitute *token* values that retain format and structure of the original data, but not cleartext data, while logically isolating the process that performs the mapping. Tokenization typically implies that the tokenization secrets and mapping process are owned by a tokenization system, a strongly isolated single entity authenticating and auditing access to the token mapping process using tokenization secrets. ANSI X9. 119-2 defines three main approaches for tokenization:

1. On Demand Random Assignment (ODRA) which generates random tokens on demand and stores the association with the plaintext value in a dynamic mapping table which grows per new token generated.
2. Static table-driven tokenization (a.k.a. vault-less tokenization) generates tokens using a tokenization mapping process which operates using small pre-generated static random substitution tables used as the tokenization secret.
3. Encryption-based tokenization generates tokens using a suitable FPE or symmetric encryption algorithm where the key serves as tokenization secret.

The FAST design can be used as base for static table-driven tokenization as well as for encryption-based tokenization.

Security goal

The FAST construction is supposed to offer a pretty high security (e.g. 128-bit security) even though the input domain could be of very small size a^l. Security holds even when the adversary can choose the parameters, the tweak, the plaintext and the ciphertext. It provides security also when the pool of S-boxes is known, which may happen for instance when the stateless table secret leaks in tokenization. Towards this goal, we will need PRNG_1 and PRNG_2 to be secure pseudorandom generators, PRF to be a pseudorandom function and we will reduce to the assumption that CEnc_S is a super-pseudorandom permutation (keyed by a random SEQ) when S is known but randomly set.

For Quantum Security, we consider adversaries who can run quantum algorithms, such as Grover or Simon. However, we do not assume quantum access to encryption/decryption *oracles*. To face quantum adversaries, the authors recommend to change the formulas only by replacing the parameter s with $2s$, except for $L_1 = L_2 = 3s$. We obtain that the number of rounds is doubled for the low l values but remains unchanged for the large ones. PRNG_1 and PRNG_2 have to change to accommodate the quantum 128-bit security; they remain still AES-CTR, but with 256-bit key. As AES-CMAC does not offer 256-bit security, we need another algorithm or to twist CMAC with 256-bit key.

Chapter 3

Attacks on FPE

3.1 Message - recovery attacks on Feistel-based FPE (2016)

The paper by Bellare, Hoang, Tessaro [18] is the first work which contains attacks on FPE schemes that succeed in message recovery when the message space is small. Later in the discussion, the attack is shortened as BHT. For 4-bit message length, the attacks fully recover the target message using:

- 2^{21} examples for the FF3 standard;
- 2^{25} examples for the FF1 standard.

The examples include three message per tweak, which makes the attacks non-trivial even though the total number of examples exceeds the size of the domain. The attacks can be neutralized by increasing the number of Feistel rounds in the standards.

The paper has three main contributions:

1. new message recovery attacks on Feistel-based FPE that are practical for small domains;
2. a definitional framework for message recovery security, called **Message sampler**;
3. rigorous analyses establishing lower bounds on the advantages of the attacks in the framework.

The attacks proposed are the first to have all of the following properties:

- they succeed in (partial or full) recovery of the target message (not just in distinguishing outputs of the FPE from random);
- they have advantage as close to one as possible (rather than very small);
- they succeed given a number Q of examples that, for the values of r rounds in the standards, makes the attacks feasible for small n. An example is (tweak, ciphertext) pair, possessed by the adversary.

Message Recovery Framework

The framework gives a new formalization of message-recovery security, defining the goal which the attacks will violate.

A **message sampler** (Figure 3.1) is an algorithm XS that returns a tuple

$$((T_1, X_1), \ldots, (T_Q, X_Q), X, a)$$

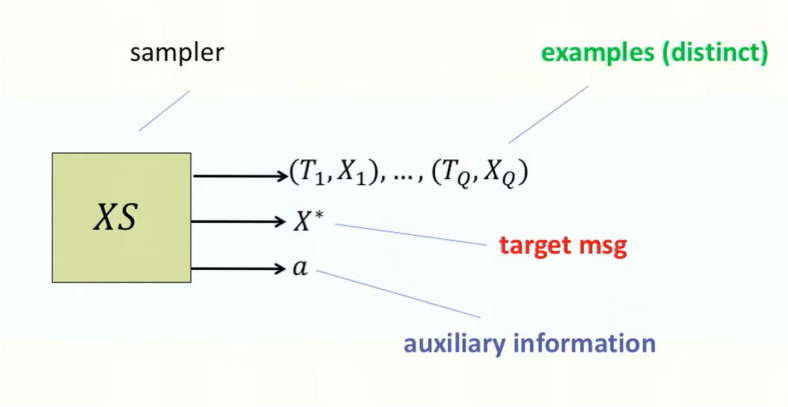


Figure 3.1: Message sampler.

consisting of Q tweak-message pairs called the *example tweak-message pairs*, a message X called the *target message* and a string a called the *auxiliary information*. The number of examples Q is a parameter of XS that is denoted $\mathsf{XS.Q}$. The Q pairs $(T_1, X_1), \ldots, (T_Q, X_Q)$ are all distinct.

In order to infer what is the unavoidable guessing probability that an adversary may have regardless of the scheme we do the following reasoning. Consider a message-guessing game where we sample the outputs of the sampler:

$$((T_1, X_1), \ldots, (T_Q, X_Q), X, a) \leftarrow \mathsf{XS}$$

and we have an adversary S which has access to the tweaks and to the auxiliary information. S attempts to guess the target message by outputting the message X'.

We can define the message-guessing advantage w.r.t. XS as:

$$\mathrm{Adv}^{\mathrm{mg}}_{\mathsf{XS}} = \max_S \Pr[X' = X]$$

It's the best guessing probability that an adversary can achieve in such a game.

Now consider the case when the adversary knows also the ciphertexts

$$C_i \leftarrow \text{FPE-Enc}(K, T_i, X_i) \text{ for } i = 1, \ldots, Q$$

If the adversary outputs X', we can define the advantage of the adversary A as

$$\mathrm{Adv}^{\mathrm{mr}}_{\mathrm{FPE},\mathsf{XS}} = \Pr[X' = X] - \mathrm{Adv}^{\mathrm{mg}}_{\mathsf{XS}}$$

This is called the message-recovery advantage.

To summarize, the *mg advantage* captures the a priori probability of guessing the target message given the tweaks and the auxiliary information, while the *mr advantage* is the excess of the adversary's probability of winning the mr game over the mg advantage. A property that a good FPE scheme should at the very least satisfy is that $\mathrm{Adv}^{\mathrm{mr}}_{\mathrm{FPE},\mathsf{XS}} \approx 0$ for all samples XS with a feasible number of examples and all A with feasible time complexity. Feasible in this context means a complexity around 2^{100} or even higher. In the three attacks below a concrete sampler XS and adversary A such that $\mathrm{Adv}^{\mathrm{mr}}_{\mathrm{FPE},\mathsf{XS}} \approx 1$ are given.

Left Half Recovery - LHR

In the LHR attack the encryptions of two samples X and X' under q tweaks $T_1, \ldots, T_q$ are given. $X = (L, R)$ and $X' = (L', R)$ have equal right segment, whereas their left segments differ. No assumptions are made on the distribution of $L', R, T_1, \ldots, T_Q$, but it is assumed that L is uniform, conditioned on being distinct from L'. The attack is formalized using the message-recovery framework, which characterize under what conditions the attack works.

Sampler $\mathsf{XS}[\mathsf{D}]$
$(X', T_1, \ldots, T_q) \leftarrow\!\!\$\ \mathsf{D}\ ;\ (L', R) \leftarrow X'$
$L \leftarrow\!\!\$\ \mathbb{Z}_N \backslash \{L'\}\ ;\ X \leftarrow (L, R)\ ;\ a \leftarrow X'$
Return $((T_1, X'), (T_1, X), \ldots, (T_q, X'), (T_q, X), X, a)$

Figure 3.2: Left Half Recovery sampler.

The LHR attack specifies a class $\mathsf{SC1}_q$ of samplers and lets $\mathsf{DC1}_q$ be the class of all algorithms D that output $X' \in \mathbb{Z}_M \times \mathbb{Z}_N$ and distinct $T_1, \ldots, T_Q \in \{0,1\}^*$. To any such D it is associated the sampler in Figure 3.2.

The LHR attack against $\mathsf{SC1}_q$ is given in Figure 3.3. It can recover the left segment of X from the ciphertexts and the left segment of X'. Since the mr notion asks for full message recovery, the right segment of X is included in a, but this information is not needed for recovering the left segment of X.

Adversary $\mathsf{LHR}((T_1, C'_1), (T_1, C_1), \ldots, (T_q, C'_q), (T_q, C_q), a)$
$X' \leftarrow a\ ;\ L \leftarrow 0\ ;\ (L', R) \leftarrow X'$
For $s \in \mathbb{Z}_M$ do $V_s \leftarrow 0$
For $i = 1$ to q do
  $(A, B) \leftarrow C_i; (A', B') \leftarrow C'_i; s \leftarrow A \boxminus A' \boxplus L'; V_s \leftarrow V_s + 1$
For $s \in \mathbb{Z}_M$ do
  If $V_s > V_L$ then $L \leftarrow s$
$X \leftarrow (L, R)\ ;$ Return X

Figure 3.3: Left Half Recovery attack.

In this scenario the message-guessing advantage has $\mathrm{Adv}^{\mathrm{mg}}_{\mathsf{XS}} \leq \frac{1}{M-1}$ as lower bound.

Right Half Recovery - RHR

In the RHR, in contrast with LHR, there are no requirement on the relationship between X and X', even so the attack will recover the right segment of X. Instead of requiring the known message X' and the target message X to have the same right segment, we only consider ciphertexts C and C' of $X = (L, R)$ and $X' = (L', R')$ such that C and C' have the same left segment.

The attack fix an integer $q \geq 1$ and let $\mathsf{DC2}_q$ be the class of all algorithms D that output $X' \in \mathbb{Z}_M \times \mathbb{Z}_N$ and distinct $T_1, \ldots, T_Q \in \{0,1\}^*$, where each sampler $\mathsf{XS}[\mathsf{D}]$ in $\mathsf{DC2}_q$ is as presented in Figure 3.4.

Sampler $\mathsf{XS}[\mathsf{D}]$
$(X', T_1, \ldots, T_q) \leftarrow\!\!\$\ \mathsf{D}\ ;\ (L', R') \leftarrow X'$
$(L, R) \leftarrow X \leftarrow\!\!\$\ (\mathbb{Z}_M \times \mathbb{Z}_N) \backslash \{X'\}\ ;\ a \leftarrow (L, R')$
Return $((T_1, X'), (T_1, X), \ldots, (T_q, X'), (T_q, X), X, a)$

Figure 3.4: Right Half Recovery sampler.

The RHR attack is given in Figure 3.5.

In this scenario the message-guessing advantage has $\mathrm{Adv}^{\mathrm{mg}}_{\mathsf{XS}} \leq \frac{1}{N-1}$ as lower bound.

Full Message Recovery - FMR

By combining the LHR attack and the RHR attack one can fully recover X as follows. We require ciphertexts of three messages X, X', X^* for q tweaks, with sufficiently large q, to recover X. The

Adversary $\mathsf{RHR}((T_1, C'_1), (T_1, C_1), \ldots, (T_q, C'_q), (T_q, C_q), a)$
$(L, R') \leftarrow a$; $R \leftarrow 0$; $\ell \leftarrow 0$; $p \leftarrow \frac{1}{N-1}$; $\Delta \leftarrow \frac{1-1/(N-1)}{M^{(r-2)/2}}$
For $s \in \mathbb{Z}_N$ do $V_s \leftarrow 0$
For $i = 1$ to q do
  $(A, B) \leftarrow C_i$; $(A', B') \leftarrow C'_i$
  If $A = A'$ then $s \leftarrow B \boxminus B' \boxplus R'$; $\ell \leftarrow \ell + 1$; $V_s \leftarrow V_s + 1$
For $s \in \mathbb{Z}_N$ do
  If $V_s > V_R$ then $R \leftarrow s$
If $V_R \leq \ell(p + \Delta/2)$ then $R \leftarrow R'$
$X \leftarrow (L, R)$; Return X

Figure 3.5: Right Half Recovery attack.

message X' is fully known but has no relation with the target X; this is already enough to recover the right segment of X, according to the RHR attack. The message X^* is required to have the same right segment as the target, but it is only partially known: only the left segment of X^* is included in the auxiliary information. For example, X^* can be the default version of X, in which the left segment is 0. Although X^* is only partially known, we already recovered the right segment of X. Then the LHR attack gives the left segment of X.

The attack fix an integer $q \geq 1$ and let $\mathsf{DC3}_q$ be the class of all algorithms D that output $(L', R') \leftarrow X' \in \mathbb{Z}_M \times \mathbb{Z}_N, L^* \in \mathbb{Z}_M\{L'\}$ and distinct $T_1, \ldots, T_Q \in \mathsf{F.Twk}$. Let $\mathsf{SC3}_q = \{\mathsf{XS[D]} \text{ — } \mathsf{D} \in \mathsf{DC3}_q\}$, where each sampler $\mathsf{XS[D]}$ in $\mathsf{SC3}_q$ is as presented in Figure 3.6.

Sampler $\mathsf{XS[D]}$
$(X', L^*, T_1, \ldots, T_q) \leftarrow\!\!\$\ \mathsf{D}$; $(L', R') \leftarrow X'$
$L \leftarrow\!\!\$\ \mathbb{Z}_M \backslash \{L', L^*\}$; $R \leftarrow\!\!\$\ \mathbb{Z}_N$; $X \leftarrow (L, R)$
$X^* \leftarrow (L^*, R)$; $a \leftarrow (X', L^*)$
For $i = 1$ to q do $Z_i \leftarrow ((T_i, X^*), (T_i, X'), (T_i, X))$
Return $(Z_1, \ldots, Z_q, X, a)$

Figure 3.6: Full Message Recovery sampler.

The FMR attack is given in Figure 3.7.

Adversary $\mathsf{FMR}(U_1, \ldots, U_q, a)$
For $i = 1$ to q do $((T_i, C^*_i), (T_i, C'_i), (T_i, C_i)) \leftarrow U_i$
$(X', L^*) \leftarrow a$; $(L', R') \leftarrow X'$; $a_1 \leftarrow (L^*, R')$
$X^* \leftarrow \mathsf{RHR}((T_1, C'_1), (T_1, C_1), \ldots, (T_q, C'_q), (T_q, C_q), a_1)$
$a_2 \leftarrow X^*$
$X \leftarrow \mathsf{LHR}((T_1, C^*_1), (T_1, C_1), \ldots, (T_q, C^*_q), (T_q, C_q), a_2)$
Return X

Figure 3.7: Full Message Recovery attack.

In Figure 3.8 the proposed attacks are summarized.

3.2 Breaking the FF3 format FPE standard over small domains (2017)

In this work by Durak and Vaudenay [15] a practical total break attack to the FF3 scheme is given. The attack requires $O(N^{11/6})$ chosen plaintext with time complexity $O(N^5)$.

Attack name	Advantage	Number of tweaks	Examples per tweak
LHR	$1 - 2/2^n$	$24(n+4)2^{(r-3)n}$	2
RHR	$1 - 2/2^n$	$24(n+4)2^{(r-2)n}$	2
FMR	$1 - 2/2^n$	$24(n+4)2^{(r-2)n}$	3

Figure 3.8: Attack parameters and effectiveness

It is a slide attack (the attack is deepen in Section 3.2.1) that exploit the bad domain separation of the FF3 design. The slide attack uses another attack developed, a known-plaintext attack to 4-round Feistel network that reconstructs the entire tables for all round functions and it works with $N^{3/2}(N/2)^{1/6}$ known plaintexts and time complexity $O(N^3)$, which is a big improvement with respect to [18] where time and data complexity is $O(log(N)N^{r-3})$, which in the case of FF3 ($r = 8$ rounds) is $O(log(N)N^5)$.

Later in the discussion, the attack is shortened as DV.

In the paper there are three important sections:

- A section which defines a total practical break to 8-round Feistel network based FF3 FPE standard over small domain. The specific design choice of FF3 allow permuting the round function by changing the tweak and it leads to develop a slide attack (using only two tweaks).
- A section which defines a generic known-plaintext attack on 4-round Feistel networks. This generic attack is inserted in the slide attack mentioned above.
- A section where the 4-round FN attack is utilized to extend the round function recovery on more rounds.

Round-function recovery on 4-Round Feistel

Consider a 4-round Feistel scheme with round functions F_0, F_1, F_2, F_3. Given x and y in X, we define the following equations:

$$\begin{aligned} c &= x + F_0(y), \\ d &= y + F_1(c), \\ z &= c + F_2(d), \\ t &= d + F_3(z). \end{aligned}$$

Assume that we collected M random pairwise different plaintext messages (xy).

We collect the pairs

$$V = \{(xy, x'y') | z' = z, t' - y' = t - y, xy \neq x'y'\}$$

and,

$$V_{good} = \{(xy, x'y') | z' = z, c' = c, xy \neq x'y'\}$$

The V_{good} set is the set of *"good vertices"*.

The parameters c, d, z, t (respectively c', d', z', t') are defined from (xy) (respectively $(x'y')$) as above. We also define $Label(xy, x'y') = x - x'$. We form a directed graph $G = (V, E)$, with the vertex set V as defined in the previous slide.

We take two pair of tuples, which together form an edge of the graph:

$$(x_1, y_1, x'_1, y'_1, x_2, y_2, x'_2, y'_2) \in E, \text{if } y'_1 = y_2.$$

A pair of tuples $x_1y_1x'_1y'_1$ is connected to another pair of tuples $x_2y_2x'_2y'_2$ if the y_2 in the second message in former tuple is the same as in the first message in latter tuple, y'_1.

Furthermore, we let $E_{good} = (V_{good} \times V_{good}) \cap E$ and define the *sub-graph* $G_{good} = (V_{good}, E_{good})$.

The following four properties are observed:

1. $V_{good} \subseteq V$.
2. If $(xy, x'y') \in V$, then $y \neq y'$.
3. If $(xy, x'y') \in V_{good}$, then $F_0(y') - F_0(y) = Label(xy, x'y')$.
4. For all cycles $v_1 v_2 \dots v_L v_1$ of G_{good}, $\sum_{i=1}^{L} Label(v_i) = 0$.

The principle of the DV attack is that if we get good vertices (vertices in V_{good}), the property 3 defined above gives equations to characterize F_0. The initial problem is that we can identify vertices in V, but we cannot tell apart good and non-good vertices. To resolve this we can employ property 4, that is to find cycles with zero sum of labels. It can be proved that this is a characteristic property of good cycles, meaning that all vertices in these cycles are good vertices.

We can hypothesize the following conjecture (which is supported by experiment for L=3): if $v_1 v_2 \dots v_L v_1$ is a cycle of length L in G with zero sum of labels and the vertices use no messages in common, then $v_1 \dots v_L$ are all good with probability close to 1.

The aim of the attack is to collect as many F_0 outputs as possible to reconstruct a table of this function. Thus we are interested in vertices whose labels are defined as $Label(v_i) = F_0(y) - F_0(y'), \forall i \in \{0, 1, \dots, |V|\}$ and we generate another graph to represent the collection of many independent equations for F_0. We have a valid cycle $v_1 v_2 \dots v_L v_1$ of length L in G when $v_i \in V$,

$$\sum_{i=1}^{L} Label(v_i) = 0$$

and vertices use no messages in common.

Now we define an undirected graph $G' = (V', E')$, where:

- $V' = \{0, 1, \dots, N-1\}$;
- E' is made up by the edges of type $\{y_i, y_i'\}$, such that the vertex $v_i = (xy, x'y')$ is in a valid cycle $v_1 v_2 \dots v_L v_1$ of length L. The label of the edge $\{y_i, y_i'\}$ is set to $Label(v_i)$.

The purpose of this new graph is to put y values which are dependent on each other in a *single* connected component and put apart independent y values in *separate* connected components.

The objective is to have a large connected component in G'. To do so we can adjust the number M of pairwise different plaintext messages. If the attack recovers at least $\sqrt{N}$ points in F_0 correctly, which is the case when we have a large connected component in G', we obtain a number of samples $>> N$ that we can use to apply the attack on 3-rounds so that it recovers a good fraction of F_1, F_2, F_3. At the end it is enough to bootstrap a yoyo attack to deduce the other points (steps 9-14 in the algorithm in Figure 3.9).

The data complexity of this algorithm is $O(N^{\frac{3}{2}+\frac{1}{2L}})$, while time complexity is $O(N^{2+\frac{3}{L}})$.

Experimentally it was noted that the best value for L is $L = 3$, having data complexity of $O(N^{\frac{5}{3}})$ and time complexity of $O(N^3)$.

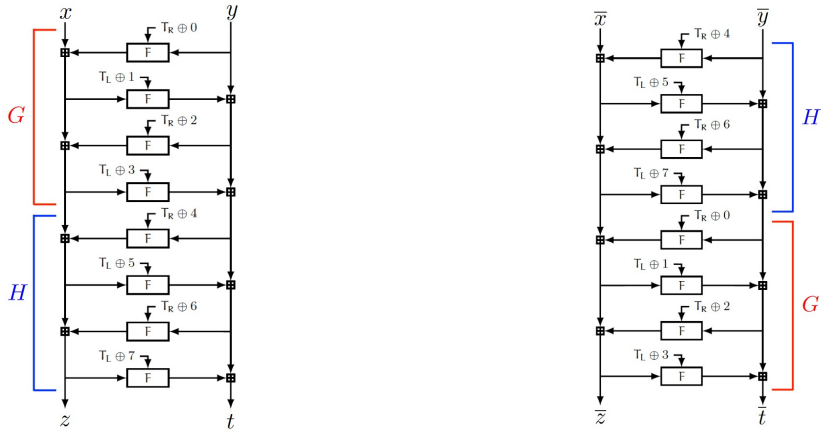
Attack on FF3

The main idea of the designed FF3 attack takes advantage of the flexibility to change the tweak to permute the round functions. In order to apply the slide attack, we consider two functions G and H, which are both 4-round Feistel scheme using as tweakable block cipher the function F. We perform two runs of the FF3 encryption, one with tweak $T = (T_L, T_R)$ and one with tweak $T' = (T_L, T_R) \oplus (4, 4)$, on two distinct plaintext. This is shown in Figure 3.10.

We can observe that the first encryption with tweak T can be expressed as the composition $H \circ G$, while the second encryption with tweak T' can be expressed as the composition $G \circ H$. Given this permuting ability by setting the tweaks XORed with round functions, we try to form

1: Pick M known plaintext and retrieve their ciphertext.
2: Create $G = (V, E)$.
3: Find valid cycles of length $2, 3, \ldots, L$ and collect the vertices in these cycles.
4: Create G' from the collected vertices.
5: Find the largest connected component in G'.
6: Assign one $F_0(y)$ value arbitrarily and deduce F_0 on the connected component.
7: For all known plaintexts using y in the connected component, evaluate and deduce a tuple for the 3-round Feistel scheme based on (F_1, F_2, F_3).
8: Apply the attack on 3-round Feistel scheme to recover a constant fraction of (F_1, F_2, F_3).
9: **while** *nothing more revealed* **do**
10: **for all** *of the M plaintext/ciphertext pairs* **do**
11: **if** F_0 *and* F_1 *are known for this plaintext* **then**
12: deduce one point for F_2 and F_3
13: **if** F_2 *and* F_3 *are known for this ciphertext* **then**
14: deduce one point for F_0 and F_1

Figure 3.9: Round-Function-Recovery Attack on 4-Round Feistel Scheme (F_0, F_1, F_2, F_3).



(a) Encryption $H \circ G$, tweak is $T = (T_L, T_R)$. (b) Encryption $G \circ H$, tweak is $T' = (T_L, T_R) \oplus (4,4)$.

Figure 3.10: Two different runs of FF3, as shown in [28].

a "cyclic" behaviour of plaintext/ciphertext pairs under two FF3 encryption with sliding G and H.

We now define a chosen-plaintext codebook-recovery attack on FF3, as shown in Figure 3.11. This can also be viewed as a slide attack. We pick at random two sets of messages $X = \{xy_0^1, \ldots, xy_0^i, \ldots, xy_0^A\}$ and $\overline{X} = \{\overline{xy}_0^1, \ldots, \overline{xy}_0^i, \ldots, \overline{xy}_0^A\}$ of size A. For each message xy_0^i in X, set $xy_{j+1}^i = \mathrm{Enc}(K, T, xy_j^i)$ with a fixed tweak $\mathrm{T} \in T$ and a fixed key $\mathrm{K} \in K$. We repeat the chain encryption of outputs B times for each message in X. Let XC be the set of chain encryption of elements of X. It contains segments of length B of cycles of $H \circ G$.

Similarly, for each message $\overline{xy}_0^i$ in $\overline{X}$, set $\overline{xy}_{j+1}^i = \mathrm{Enc}(K, T', \overline{xy}_j^i)$ with a fixed tweak $\mathrm{T}' \in T$, under the same key K used previously. Let $\overline{XC}$ be the set of chain encryption of elements $\overline{X}$. Apparently, we have $|XC| = AB$ and $|\overline{XC}| = AB$. Given these two sets XC and $\overline{XC}$, we attempt to find a collision between XC and $\overline{XC}$ such that $G(xy_j^i) = \overline{xy}_0^{i'}$ or $G(xy_0^i) = \overline{xy}_{j'}^{i'}$, for $i \leq 1, i' \leq A$ and $1 \leq j, j' \leq B$.

Upon having a table with inputs G and H, we can apply the known-plaintext recovery attack on 4-round Feistel networks presented in Section 3.2.

The algorithm for the slide attack on FF3 is reported in Figure 3.12. This is a total practical break to FF3 standard when the message domain is small. We can observe that in lines 14 and 19 we try to find collisions for the two sets.

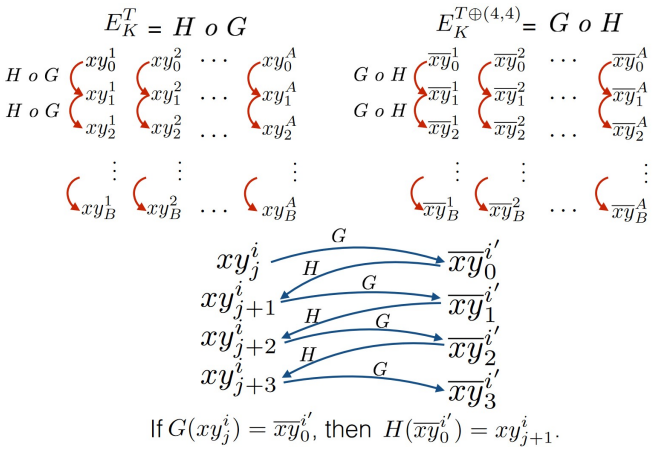


Figure 3.11: Representation of the chosen-plaintext attack on FF3, as shown in [28].

The slide attack has time complexity as $2N^2$ times the complexity of 4-round recovery attack. Since the 4-round recovery attack has running time $O(N^{2+3/L})$, with $L = 3$ we have $O(N^3)$. The total time complexity of the slide attack is:

$$O(2N^2 * N^3) = O(N^5).$$

The data complexity of the slide attack is $4N\sqrt{M}$. For the 4-round recovery attack we have that the data complexity is $M = O(N^{3/2+1/2L})$, with $L = 3$ we obtain $M = O(N^{3/2+1/6})$. The total data complexity is:

$$O(4N\sqrt{M}) = O(N\sqrt{M}) = O(N(N^{10/6})^{1/2}) = O(N * N^{10/12}) = O(N^{(12+10)/12}) = O(N^{11/6}).$$

Input : a tweak bit string T such that $|T| = 64$, a key K
$T_L \| T_R \leftarrow T$
$T' \leftarrow T_L \oplus 4 \| T_R \oplus 4$
foreach $i = 1 \cdots A$ **do**
  pick xy_0^i and $\overline{xy}_0^i$
  foreach $j = 1 \cdots B$ **do**
    $xy_j^i = \text{FF3.E}(K, T, xy_{j-1}^i)$
    $\overline{xy}_j^i = \text{FF3.E}(K, T', \overline{xy}_{j-1}^i)$
  end
end
foreach $i, i' = 1 \cdots A$ **do**
  foreach $j = 0 \cdots B - M - 1$ **do**
    // assume that $G(xy_j^i) = \overline{xy}_0^{i'}$
    run attack on G with samples $G(xy_{j+k}^i) = \overline{xy}_k^{i'}$ for $k = 0 \cdots B - j$
    if succeeded, run attack on H with samples $H(G(xy_k^i)) = xy_{k+1}^i$ for $k = 0 \cdots B - 1$
  end
  foreach $j = 0 \cdots B - M - 1$ **do**
    // assume that $G(xy_0^i) = \overline{xy}_j^{i'}$
    run attack on G with samples $G(xy_k^i) = \overline{xy}_{j+k}^{i'}$ for $k = 0 \cdots B - j$
    if succeeded, run attack on H with samples $H(G(xy_k^i)) = xy_{k+1}^i$ for $k = 0 \cdots B - 1$
  end
end

Figure 3.12: FF3 Slide Attack.

3.2.1 Slide attacks

The slide attack is a form of cryptanalysis that contrasts the idea that even weak ciphers can become very strong by increasing the number of rounds, which can ward off a differential attack. Consequently, the slide attack makes the number of rounds in a cipher irrelevant. The slide attack

works by analysing the key schedule and exploiting weaknesses in it to break the cipher. The most common weakness is when the keys are repeated in a cyclic manner. This type of attack was first described by David Wagner and Alex Biryukov [29]. The slide attack is closely related to another form of cryptanalysis, the related-key attack.

The requirements for a slide attack to work on a cipher are only two:

- That the cipher can be broken into multiple rounds of an identical F function. This probably means that it has a cyclic key schedule.
- The F function must be vulnerable to a known-plaintext attack.

We now give a brief explanation of the slide attack.

Assume that the cipher takes n bit blocks and has a key-schedule $K_1, \ldots, K_m$ as keys of any length. Firstly, the slide attack breaks the cipher into identical permutation functions F. This F function may consist of more than one round of the cipher; it is defined by the key schedule. For example, if a cipher uses an alternating key schedule where it switches between a K_1 and K_2 for each round, the F function would consist of two rounds. Each of the K_i will appear at least once in F.

The next step is to collect $2^{n/2}$ plaintext-ciphertext pairs. Depending on the characteristics of the cipher fewer may suffice (anyway the birthday problem gives as upper bound $2^{n/2}$). These pairs, denoted as (P, C), are then used to find a *slid pair* which is denoted as $(P_0, C_0), (P_1, C_1)$.

A slid pair has the property that $P_0 = F(P_1)$ and that $C_0 = F(C_1)$. Once a slid pair is identified, the cipher is broken because of the vulnerability to known-plaintext attacks. The key can easily be extracted from this pairing.

The slid pair can be seen as the message after one application of the function F, it is a *"slid"* over one encryption round. In Figure 3.13 there is a graphical representation of the slid.

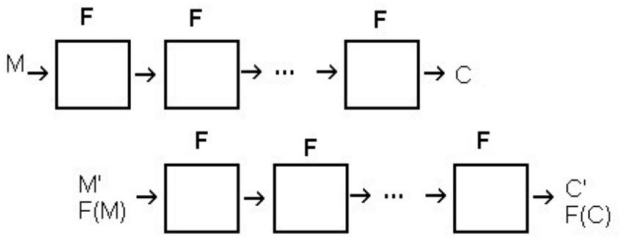


Figure 3.13: Slid pair.

The process of finding a slid pair is different for each cipher, but follows the same basic scheme. In fact it is easy to extract the key from just one iteration of F. Chose any pair of plaintext/ciphertext pairs, $(P_0, C_0), (P_1, C_1)$ and check to see what are the keys corresponding to $P_0 = F(P_1)$ and $C_0 = F(C_1)$. If these keys match, this is a slid pair; otherwise move on to the next pair.

With $2^{n/2}$ plaintext/ciphertext pairs, one slid pair is expected. A small number of false-positive are by the way expected, depending on the structure of the cipher. The false positive can be eliminated by using the keys on a different message-ciphertext pair, to see if the encryption is correct. The probability that the wrong key will correctly encipher two or more messages is very low for a good cipher.

Sometimes the structure of the cipher greatly reduces the number of plaintext/ciphertext needed. The most relevant example is the Feistel cipher using a cyclic key schedule, where the round function $F((l, r)) = (r \oplus f(l), l)$ modifies only half of the input. This reduces the possible paired messages from 2^n to $2^{n/2}$ and so at most $2^{n/4}$ plaintext/ciphertext pairs are needed in order to find a slid pair.

3.3 The curse of small domains: new attacks on FPE (2018)

As shown in Section 3.1, the BHT attacks can be thwart by increasing the number of rounds of the constructions. The Feistel construction is not the only approach used in practice for FPE; *Cisco* presented a variant of Feistel, called FNR [23], which appears to bypass the BHT attacks. *Protegrity* uses a different construction, called DTP [24], based on Brightwell and Smith's construction [6].

This paper by Hoang, Tessaro and Trieu [17] (later in the discussion shortened as HTT) contains three main contributions:

- new attacks against Feistel-based FPE that improve upon BHT in settings where multiple messages can be recovered;
- an attack against FNR;
- a strong ciphertext-only attack against DTP.

The first contribution is a message-recovery attack on a generic Feistel based FPE $F = \text{Feistel}[r, M, N, \boxplus, \text{PL}]$, where $\text{PL} = (T, K, F_1, \ldots, F_r)$. The Feistel structure is represented in Figure 3.14. The approach of the attack is based on the BHT Left-half Differential attack and we consider only the case where r is even, as NIST standards only use $r = 8$ (for FF3) or $r = 10$ (for FF1). There are τ known messages $X_1, \ldots, X_\tau$ and p targets $Z_1, \ldots, Z_p$. The adversary is given the encryption of those $\tau + p$ distinct messages under q tweaks $T_1, \ldots, T_q$, for an appropriately large q. Due to the distinctness requirement $X_1, \ldots, X_\tau, Z_1, \ldots, Z_p$ must be distinct. The auxiliary information is $X_1, \ldots, X_\tau, p, q$. The attack is formalized via the message-recovery framework, introduced in Section 3.1.

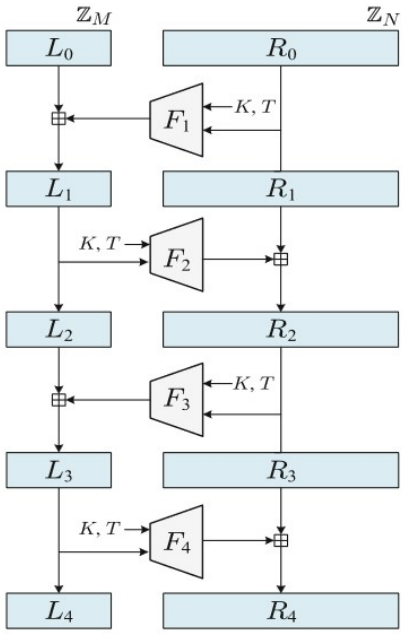


Figure 3.14: Illustration of encryption of the generic Feistel FPE with r = 4 rounds.

The attack needs to specify a class $\mathsf{SC1}_{p,q,\delta,\theta}$ of samplers and a lower bound on the *mr*-advantage of the attack for any sampler in this class. First, let $\mathsf{DC1}_{p,q,\delta,\theta}$ be the class of all algorithms D that outputs q distinct tweaks $T_1, \ldots, T_q \in \{0,1\}^*$ and distinct $X_1, \ldots, X_\tau, Z_1, \ldots, Z_p \in \mathbb{Z}_M \times \mathbb{Z}_N$ such that:

1. With probability at least $1 - \delta$, there are d or more indices k such that:

$$Z_k \in \{\text{Right}(X_1), \ldots, \text{Right}(X_\tau)\}.$$

2. Given $X_1, \ldots, X_\tau, Z_1, \ldots, Z_p$ for any subset $\{r_1, \ldots, r_d\} \subseteq \{1, \ldots, \tau\}$, for any $Z_1^*, \ldots, Z_d^* \in \mathbb{Z}_M \times \mathbb{Z}_N \{X_1, \ldots, X_\tau\}$, the conditional probability that $Z_{r_1} = Z_1^*, \ldots, Z_{r_d} = Z_d^*$ is at most $2^{-\theta}$.

For any such D, we associate the sampler in Figure 3.15.

$\underline{\text{Sampler } \mathsf{XS}[\mathsf{D}]}$
$(T_1, \ldots, T_q, X_1, \ldots, X_\tau, Z_1, \ldots, Z_p) \leftarrow\!\!\$\ \mathsf{D}$
$a \leftarrow (X_1, \ldots, X_\tau, p, q)$
Return $\big(\{(T_i, X_j), (T_i, Z_k) \mid i \le q, j \le \tau, k \le p\}, Z_1, \ldots, Z_p, a\big)$

Figure 3.15: Sampler of the message-recovery attack on a generic Feistel FPE

The sampler returns the pair (T_i, X_j) and (T_i, Z_k) for every $i \le q, j \le \tau, k \le p$, where the targets are $Z_1, \ldots, Z_p$. The Left-half Differential attack can recover d targets out of $Z_1, \ldots, Z_p$ in $O(pqN)$ time.

The attack (Figure 3.16) is based on an observation by BHT that for any two messages X and X' of the same right half, if we encrypt them under the same tweak we obtain ciphertexts C and C' respectively, the $\text{Left}(C) \boxminus \text{Left}(C')$ is most likely to be $\text{Left}(X) \boxminus \text{Left}(X')$.

$\underline{\text{Adversary } \mathsf{LD}(\{(T_i, C_{i,j}), (T_i, C'_{i,k})\}_{i,j,k}, a)}$
// $1 \le i \le q, 1 \le j \le \tau, 1 \le k \le p$
$(X_1, \ldots, X_\tau, p, q) \leftarrow a$; $S, \text{Dom} \leftarrow \emptyset$
For $j = 1, \ldots, \tau$ do
  If $\mathbf{Right}(X_j) \notin \text{Dom}$ then $S \leftarrow S \cup \{j\}$; $\text{Dom} \leftarrow \text{Dom} \cup \{\mathbf{Right}(X_j)\}$
For $k \leftarrow 1$ to p do // Recover target Z_k
  For $j \in S$, $s \in \mathbb{Z}_M$ do $V_{j,s} \leftarrow 0$
  For $i \leftarrow 1$ to q, $j \in S$ do
    $s \leftarrow \mathbf{Left}(C'_{i,k}) \boxminus \mathbf{Left}(C_{i,j}) \boxplus \mathbf{Left}(X_j)$; $V_{j,s} \leftarrow V_{j,s} + 1$
  Let $V_{j^*,s^*} = \max\{V_{j,s} \mid j \in S, s \in \mathbb{Z}_M\}$; $Z_k \leftarrow (s^*, \mathbf{Right}(X_{j^*}))$
Return $(Z_1, \ldots, Z_p)$

Figure 3.16: The Left-half Differential attack in the HTT version

We can now do some final considerations. In a Feistel-based scenario when the domain length is odd, FF1 and FF3 have different ways to interpret what are M and N, i.e. the domain dimensions of the FPE scheme. In those odd domains the attack on Feistel-based FPE proposed in the paper does not improve BHT's attack for FF1, but significantly improves BHT's attack for FF3. This attack shows:

1. That FF3's way of partitioning odd domains is inferior to that of FF1.
2. That for tiny domains, the round counts used in FF1 and FF3 are not enough (as BHT already pointed out).

The attack underlines weaknesses which might have eliminated FF1 and FF3 from consideration during standardization. To thwart the attack proposed in this paper or BHT's attack, for tiny domains one has to add a few more rounds, which is a drawback for performance-hungry applications.

3.4 Attacks only get better: how to break FF3 on large domains (2019)

In this work by Hoang, Miller and Trieu (HMT attack later in the discussion) the attack of Durak and Vaudenay on FF3 is improved. The new attack [30] reduces the running time from $O(N^5)$ to $O(N^{17/6})$ for domain $\mathbb{Z}_N \times \mathbb{Z}_N$. As an example, DV's attack needs about 2^{50} operations to

recover 6-digit PINs encrypted, whereas this new attack uses only about 2^{30} operations. The new considerations of this paper improved also the running time of DV's known-plaintext attack on 4-round Feistel of domain $\mathbb{Z}_N \times \mathbb{Z}_N$ from $O(N^3)$ to $O(N^{5/3})$. Furthermore the attacks are generalized to a domain $\mathbb{Z}_M \times \mathbb{Z}_N$, allowing to recover encrypted SSNs using about 2^{50} operations.

The efficiency declared above is achieved by involving an elegant paradigm which combines distinguishing attack with Slide attacks (as presented by Biryukov and Wagner in 1999 [29] and in 2000 [31]) and improved cryptanalysis of 4-round Feistel. The attack performs the same queries as DV's attack, thus the two attacks have the same scenario and asymptotic data/space complexity $\Theta(N^{11/6})$ for domain $\mathbb{Z}_M \times \mathbb{Z}_N$. However HMT does less aggressive choices of the parameters with respect to DV, so it has a lower recovery rate.

In the HMT attack there two main contributions proposed that improve the efficiency:

1. Eliminating false instances. In the DV attack it can be observed that it is an overkill to use an expensive codebook-recovery attack on false instances. A better solution is to find a cheap test to tell whether an instance is true or false. A natural choice for such a test is a distinguishing attack on 4-round Feistel. The new attack designed is called **Left-Half Differential** attack, having the following features:

   - In the ideal world it returns 1 with probability $\frac{1}{\sqrt{N}}$.
   - In the real world it returns 1 with probability $1 - \frac{1}{8\sqrt{N}} - \frac{10}{N} - \frac{1}{N^{3/4}}$.

   The LHD attack uses $O(N^{5/6})$ pairs of plaintext/ciphertext and runs in $O(N^{5/6})$ time. In the test the LHD algorithm runs twice: first on the plaintext/ciphertext pairs of f and then on those of g.

   Thus, given a false instance, the chance that we fail to eliminate it is at most $\frac{1}{N}$. Whereas, given a true instance, the chance that we accept it is at least $(1 - \frac{1}{8\sqrt{N}} - \frac{10}{N} - \frac{1}{N^{3/4}})^2$.

   Since nobody before this work has explored the idea of using distinguishing attacks to eliminate false instances in slide attacks, the analyses on FF3 provide a pedagogical example of this paradigm.

2. Use a better attack on 4-round Feistel. Thanks to the LHD test presented before, we are left with $O(N)$ false instances, instead of $O(MN)$ initial instances, and a few true instances. If we use DV's codebook recovery attack on 4-round Feistel (which has time complexity $O(N^3)$), one would end up with $O(N^4)$ expected time. The core part of DV attack needs to find all directed 3-cycles of zero weight in a random directed graph $G = (V, E)$.

   DV's approach is to enumerate all directed 3-cycles via some sparse *matrix multiplication* and pick those of zero weight, spending $O(|V||E|)$ time. This paper gives an algorithm which uses $O(|V| + |E|)$ expected time.

Break FF3 with HMT's approach

The attack presented is a chosen-plaintext codebook recovery, called **Slide-then-Differential** (SD) attack. The SD attack is himself based on the **Triangle-Finding** (TF) attack, a known-plaintext codebook-recovery attack on 4-round Feistel.

The running time of TF is $O(M^{5/3})$ and it actually recovers the round functions of the Feistel network, using $p = \max\{\lfloor 2^{1/3}M^{2/3}N \rfloor, \lceil M(ln(N) + 5) \rceil\}$ known plaintext/ciphertext pairs. It can be noted that TF is used in a modular way: one does not need to know its technical details to understand SD.

In Figure 3.17 is given an alternative code of the DV attack, where it is used a number of plaintext/ciphertext pairs p, such that $p = \max\{\lfloor 2^{1/3}M^{2/3}N \rfloor, \lceil M(ln(N) + 5) \rceil\}$, and a number of samples s, such that $s = \lfloor \sqrt{MN/2p} \rfloor$. In practice we are using the parameters of the HMT; in fact, the code of DV is also the main procedure of the SD attack.

The procedure **Slide(U,V)** takes as input two chains $\mathbf{U} = (U_0, \ldots, U_{2p})$ and $\mathbf{V} = (V_0, \ldots, V_{2p})$, tries to find a *slid pair* (U_i, V_0) and then uses TF to recover the codebook. The difference between

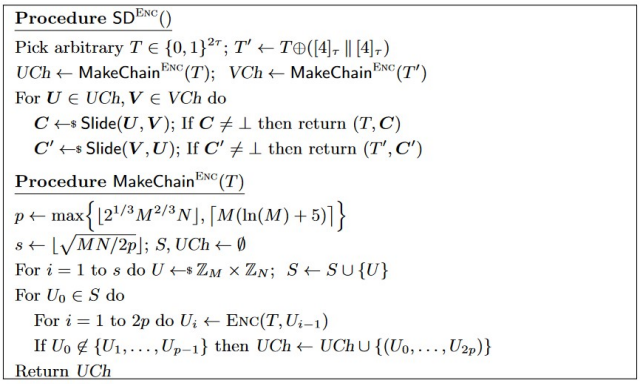


Figure 3.17: Procedure of the SD attack.

the DV attack and the SD attack is in how they implement the procedure **Slide** for finding a slid pair, among $2s^2p \approx MN$ candidates.

DV simply tries every every possible candidate, by running a slow version of the TF algorithm to recover the codebook of f and g. However there are often very few slid pairs, thus DV attack has to run TF for about $\Theta(MN)$ times, which is very expensive. The key idea of the SD attack is to use some differential analysis to quickly eliminate false candidates.

Left Half Differential attack

We now give the LHD attack in the $\mathbb{Z}_M \times \mathbb{Z}_N$ domain. It is a distinguishing attack on 4-round Feistel such that:

- in the ideal world it returns 1 with probability $\frac{N^{5/6}}{M^{4/3}}$;
- in the real world it returns 1 with probability $1 - \frac{\sqrt{N}}{8M} - \frac{9.7}{M} - \frac{0.88N^{3/4}}{M^{3/2}}$.

For each slid-pair candidate, we can run the LHD on the plaintext/ciphertext pairs of f and on those of g. We will accept the candidate only if LHD returns 1 in both cases.

For each false candidate, the chance that we incorrectly accept it is at most $\frac{N^{5/3}}{M^{8/3}}$. Since we have at most MN false candidates, on average we will have at most $\frac{N^{8/3}}{M^{5/3}}$ false candidates that survived the test.

For each true candidate, the chance that we incorrectly reject it is at most $(1 - \frac{\sqrt{N}}{8M} - \frac{9.7}{M} - \frac{0.88N^{3/4}}{M^{3/2}})^2$ for $M, N \geq 64$.

The LHD algorithm is based on a *Lemma*, which basically tells that if we encrypt two messages X and X' of the same right segment under a 4-round Feistel network, then there will be some **bias** in the distribution of the ciphertext C and C':

- the chance that $LH(C) \boxminus LH(C') = LH(X) \boxminus LH(X')$ is $\frac{M+N-1}{MN}$;
- the probability that we instead sampled C and C' uniformly without replacement from $\mathbb{Z}_M \times \mathbb{Z}_N$ is just $\frac{N}{MN-1}$.

The LHD attack wants to amplify this bias, by using several messages of the same right segments. In Figure 3.18 is given an efficient implementation of LHD with proper parallelization and hash table usage. The total running time of this implementation is $O(m) = O(M^{2/3}N^{1/6})$.

Procedure $\mathsf{LHD}(X_1, \ldots, X_m, C_1, \ldots, C_m)$
// $X_1, \ldots, X_m$ are already grouped according to their right segments
Partition $X_1, \ldots, X_m$ by the right segments into groups $P_1, \ldots, P_d$
$count \leftarrow 0$; $\Delta \leftarrow \frac{1}{5} \cdot \frac{M+N-1}{MN} + \frac{4}{5} \cdot \frac{N}{MN-1}$; $size \leftarrow \sum_{\ell=1}^{d} \frac{|P_\ell|(|P_\ell|-1)}{2}$
For $\ell \leftarrow 1$ to d do
  $count_\ell \leftarrow 0$; Initialize a hash table H_ℓ
  For $X_k \in P_\ell$ do
    $Z \leftarrow \mathsf{LH}(C_k) \boxminus \mathsf{LH}(X_k)$; $v \leftarrow H_\ell[Z]$
    If $v = \perp$ then $H_\ell[Z] \leftarrow 1$ else $H_\ell[Z] \leftarrow v + 1$
  For each key Z in H_ℓ do $v \leftarrow H_\ell[Z]$; $count_\ell \leftarrow count_\ell + \frac{v(v-1)}{2}$
$count \leftarrow count_1 + \cdots + count_d$
If $count \geq \Delta \cdot size$ then return 1 else return 0

Figure 3.18: LHD implementation.

Slide-then-Differential attack

We now describe how to combine DV's slide attack with the LHD attack discussed before, resulting in the SD attack.

The first aspect that we have to focus on is how we can speed-up the pre-processing step. We recall that we have $\Theta(MN)$ slid-pair candidates, for each of them we have to process $\Theta(p)$ pairs of plaintext/ciphertext.

Since $p \approx M^{2/3}N$ for big values of M and N, normally we should have $\Omega(pMN) = \Omega(M^{2/3}MN) = \Omega(M^{5/3}N^2)$. However we will perform an efficient one-time pre-processing step using only $O(MN)$ time. After this pre-processing we can extract m right-matching messages for f in $O(m) = O(M^{2/3}N^{1/6})$ time and these messages are already grouped according to their right segments. The same time is required to extract messages for g. We then can run LHD to eliminate most of the false slid-pair candidates.

In Figure 3.19 is given an implementation of the procedure **Slide**, obtained by combining the LHD attack and the pre-processing step.

Procedure $\mathsf{Slide}(\boldsymbol{U}, \boldsymbol{V})$
$(U_0, \ldots, U_{2p}) \leftarrow \boldsymbol{U}$; $(V_0, \ldots, V_{2p}) \leftarrow \boldsymbol{V}$; $(Z_1, \ldots, Z_{MN}) \leftarrow \mathbb{Z}_M \times \mathbb{Z}_N$
$L \leftarrow \mathsf{Process}((U_{p+1}, p+1), \ldots, (U_{2p}, 2p))$
$L' \leftarrow \mathsf{Process}((V_0, 0), \ldots, (V_{p-1}, p-1))$
For $i = 0$ to $p-1$ do // Check if (U_i, V_0) is a slid pair
  If $(\mathsf{Dist}(L, \boldsymbol{V}, -i) \wedge \mathsf{Dist}(L', \boldsymbol{U}, i+1))$ then
    $\boldsymbol{X} \leftarrow (U_p, \ldots, U_{2p-1})$; $\boldsymbol{Y} \leftarrow (V_{p-i}, \ldots, V_{2p-1-i})$
    $\boldsymbol{X}^* \leftarrow (V_0, \ldots, V_{p-1})$; $\boldsymbol{Y}^* \leftarrow (U_{i+1}, \ldots, U_{i+p})$
    $f \leftarrow\!\!\$\ \mathsf{Recover}(\boldsymbol{X}, \boldsymbol{Y})$; $g \leftarrow\!\!\$\ \mathsf{Recover}(\boldsymbol{X}^*, \boldsymbol{Y}^*)$
    If $f \neq \perp$ and $g \neq \perp$ then
      For $i = 1$ to MN do $C_i \leftarrow g(f(Z_i))$
      Return $(C_1, \ldots, C_{MN})$ // Codebook is $(Z_1, C_1), \ldots, (Z_{MN}, C_{MN})$

Procedure $\mathsf{Process}((X_1, r_1), \ldots, (X_p, r_p))$ // Preprocessing
$L \leftarrow \emptyset$; $m \leftarrow \lceil \frac{p}{N} \cdot \lceil 32N^{1/6} \rceil \rceil$
Partition $(X_1, r_1), \ldots, (X_p, r_p)$ by the right segments
Let $P_1, \ldots, P_M$ be the resulting partitions, with $|P_1| \geq \cdots \geq |P_M|$
For $i = 1$ to M, $(X, r) \in P_i$ do: If $|L| < m$ then $L \leftarrow L \cup \{(X, r)\}$
Return L

Procedure $\mathsf{Dist}(L, \boldsymbol{V}, k)$ // Running LHD with preprocessed list L
$i \leftarrow 1$, $(V_0, \ldots, V_{2p}) \leftarrow \boldsymbol{V}$; $m \leftarrow |L|$
For $(Z, r) \in L$ do $X_i \leftarrow Z$; $C_i \leftarrow V_{r+k}$; $i \leftarrow i + 1$
Return $\mathsf{LHD}(X_1, \ldots, X_m, C_1, \ldots, C_m)$

Procedure $\mathsf{Recover}(\boldsymbol{X}, \boldsymbol{Y})$
$(X_0, \ldots, X_{p-1}) \leftarrow \boldsymbol{X}$; $(Y_0, \ldots, Y_{p-1}) \leftarrow \boldsymbol{Y}$
$(F_1, F_2, F_3, F_4) \leftarrow\!\!\$\ \mathsf{TF}(X_0, \ldots, X_{p-1}, Y_0, \ldots, Y_{p-1})$
Let f be the 4-found Feistel of round functions (F_1, F_2, F_3, F_4)
// Function f will be $\perp$ if TF does not fully recover (F_1, F_2, F_3, F_4)
Return f

Figure 3.19: Implementation of procedure Slide in the SD attack.

Focusing on the time complexity, we have that the running time of the pre-processing is $O(p)$. Hence, totally, for s^2 pairs the overall running time of the pre-processing is:

$$O(s^2 p) = O((\sqrt{\frac{MN}{2p}})^2 p) = O(\frac{MN}{2p} p) = O(MN)$$

The SD attack uses

$$O(sp) = O(\sqrt{\frac{MN}{2p}}) = O(\sqrt{MNp}) = O(M^{1/2} N^{1/2} (M^{2/3} N)^{1/2}) = O(M^{5/6} N)$$

queries and space. Since $m \approx \frac{p}{N} N^{1/6} = \frac{p}{N^{5/6}}$, for the LHD algorithm the running time is

$$O(MNm) = O(MN \frac{p}{N^{5/6}}) = O(MN^{1/6} M^{2/3} N) = O(M^{5/3} N^{7/6})$$

.

For running TF $O(M^{5/3} N)$ time is needed. Hence the total running time for the SD attack is $O(M^{5/3} N^{7/6})$.

In the balanced case, where $M = N$ we have:

$$O(M^{5/3} N^{7/6}) = O(N^{5/3} N^{7/6}) = O(N^{(10+7)/6}) = O(N^{17/6}).$$

3.5 Three third generation attacks on the FPE scheme FF3 (2021)

This work [32], is co-published by the famous cryptographer Adi Shamir, known for the RSA algorithm.

Until 2021, the best known attack on FF3 had data and memory complexity of $O(N^{11/6})$ and time complexity of $O(N^{17/6})$, where the domain size is $N \times N$ (results of Hoang, Miller and Trieu, presented at Eurocrypt 2019). In this paper are presented three improved attacks on FF3. The best attack achieves the trade-off curve $D = M = \tilde{O}(N^{2-t}), T = \tilde{O}(N^{2+t})$ for all $t \leq 0.5$. In particular, the data and memory complexities are reduced to the more practical $\tilde{O}(N^{1.5})$ and the time complexity to $\tilde{O}(N^{2.5})$.

Also, another attack vector against FPE schemes is identified: the **related-domain** attack. It is shown how powerful attacks can be mounted when the adversary is given access to the encryption under the same key in different domains. Furthermore it is explained how to apply this to efficiently distinguish FF3 and FF3-1 instances.

This paper contains attacks of the third generation. The first generation is identified with the DV attack, while the second generation is identified with the attack by Hoang et al. (HTT attack). Three slide attacks on FF3 are presented:

1. A *symmetric slide attack* that follows the general strategy of Hoang et al., but simultaneously improves all its complexity measures. It can be generalized to any point along a particular time/data trade-off curve ($D = M = \tilde{O}(N^{7/4-t})$ and $T = \tilde{O}(N^{5/2+2t})$, for any $0 \leq t \leq 1/4$).
2. A new type of *asymmetric slide attack* which exploits the asymmetry of the classical distinguisher on 4-round Feistel schemes to reduce the complexity even further. The reduction in data complexity is especially important, since it pushes the amount of required data significantly farther from the entire codebook, while keeping the time complexity lower than the complexity of Hoang et al.'s attack.
3. A *slide attack* using the cycle structure which matches the second attack at the lowest overall complexity point $D = M = T = N^2$. This attack is particular interesting since is the first practical application of the slide attack using the cycle structure technique, which was previously believed to be purely academic due to its huge data complexity, but can be applied in the context of FPE schemes due to their small input domains.

The new attacks utilize an improved PRF reconstruction phase. Durak and Vaudenay presented that the actual round functions can be reconstructed given $\tilde{O}(N^{10/6})$ input/output pairs in time $O(N^3)$. The time complexity of the reconstruction attack was improved by Hoang, Miller and Trieu to $O(N^{5/3})$.

Both algorithms rely on finding cycles of length 3 in a graph (defined by the data). In this paper is presented an improved cycles detection algorithm (based on MITM approach), that allows finding longer cycles (of length 4 and 5) while reducing the data complexity of this phase to $\tilde{O}(N^{3/2})$ as well as the time complexity to $\tilde{O}(N^{3/2})$.

Clarification on FF3

The encryption algorithm of FF3 takes a 64-bit tweak $T = T_L||T_R$, where T_L and T_R are 32-bit each. An 8-round Feistel construction is used. In each round, half of the data is encoded into 96-bits (padding it with 0's if needed) using the naïve lexicographic transformation. The encoded value is appended to the XOR of the 32-bit tweak and the round constant. The resulting 128-bit string is then encrypted under AES with the key K. The AES's output is then added using modular addition to the other half.

After the DV's attack a new version, FF3-1 had been proposed. In FF3-1, the tweaks T_L and T_R are chosen such that they always have 0 in the 4 bits which accept the round counter i. This new tweak destroys the related-tweak slide property which lies in the core of the slide distinguishers, and thus prevents the DV's attack, as well as its extension by Hoang et al. and the results presented in the three new attacks of this paper. All these attacks are only applicable to the original FF3 scheme. On the other hand, the related-domain distinguishing attack applies both to FF3 and to FF3-1 (it's independent of the tweak schedule).

Similarly to Hoang et al.'s attack, this three attacks use two subroutines:

- an identification of the correct slid chains;
- a PRF reconstruction phase.

The PRF reconstruction phase is the same for all of the slid chain identification variants. It follows the same general idea suggested by DV and HMT, based on cycles. At the same time, it is introduced a *Meet In The Middle* (MITM) approach to the recovery itself, which significantly reduces its running time, allowing the use of *large cycles*.

Symmetric slide attack

In this attack the data is composed of two sets of $\tilde{O}(N^{1/4})$ chains, each containing $\tilde{O}(N^{3/2})$ plaintexts. The first set of chains is encrypted under K and T. The second set of chains is encrypted under K and T'. The attack follows these steps:

- it iterates over all $\tilde{O}(N^{1/2})$ pairs of chains created by taking a chain from each set;
- for each such pair of chains, we slid the first chain across the second for $\tilde{O}(N^{3/2})$ different offsets;
- for each of the $\tilde{O}(N^2)$ resulting offsets, we utilize a *distinguishing attack* to checking whether the candidate slid chains (with offset) corresponds to 4 round of FF3 or not.

The distinguisher used is similar to the one presented in HMT. We rely on the fact that the truncated differential characteristic $(x,0) \rightarrow (x,?)$ for 4 round FF3 has probability of about $\frac{2}{N}$ rather than $\frac{1}{N}$ for the random case.

Unlike HMT, that divided the datasets between bins and counted how many of them had "more pairs than expected in the random case", this attack argue that a single counter is sufficient. The

number of pairs that follow the truncated differential is distributed according to the Poisson distribution.

Note that we can run the distinguisher twice: once for the first 4 rounds and another time for the second 4 rounds. Hence, the probability of a wrong slid chain to pass the distinguisher is less than $\frac{1}{N}$. The attack follows the steps of HMT, but with a significantly smaller number of pairs needed for the distinguisher. The resulting algorithm takes $\tilde{O}(N^{3/2})$ data and $\tilde{O}(N^{2.5})$ time.

The algorithm is reported in Figure 3.20.

Input : $\tilde{O}(N^{1/4})$ chains C^r of $\tilde{O}(N^{3/2})$ plaintexts encrypted under K and $T = T_L||T_R$
Input : $\tilde{O}(N^{1/4})$ chains $\hat{C}^s$ of $\tilde{O}(N^{3/2})$ plaintexts encrypted under K and $T' = T_L \oplus 4||T_R \oplus 4$
Output: Slid chains $C^i, \hat{C}^j$ and their respective offset

for *all chains* C^r **do**
  Initialize a hash table H_1
  Insert all the plaintexts $(L_0^i, R_0^i) \in C^r$ into H_1 indexed by R_0^i
  Take a constant number d of bins (each with $O(\sqrt{N})$ plaintexts)
  Denote the plaintexts by $X_{i_1}, X_{i_2}, \ldots, X_{i_v}$
  for *all chains* $\hat{C}^s$ **do**
    for *all respective offsets* $u = 0, \ldots, N^{3/2}$ **do**
      Extract $(\hat{L}_0^{u+i_1}, \hat{R}_0^{u+i_1}), (\hat{L}_0^{u+i_2}, \hat{R}_0^{u+i_2}), \ldots, (\hat{L}_0^{u+i_v}, \hat{R}_0^{u+i_v})$ from $\hat{C}^s$
      Denote these values as "ciphertexts" $C_{i_1}, C_{i_2}, \ldots, C_{i_v}$
      Initialize d hash tables H_2^j
      for *all* k=1,...,v **do**
        if X_{i_k} *is from bin* j **then**
          Store in H_2^j the value $LH(C_{i_k}) - LH(X_{i_k})$
      Count the number of collisions in all H_2^j
      if *number of collisions is greater than* $\frac{1.6}{N} \cdot \Sigma_{B \in bins} \binom{|B|}{2}$ **then**
        Call the PRF-recovery procedure with C^r as inputs and $\hat{C}^s$ shifted by u as the outputs.

Figure 3.20: Improved symmetric slide distinguisher for FF3.

There is also a time-data trade-off variant which takes $\tilde{O}(N^{7/4-t})$ data and has a running time of $\tilde{O}(N^{5/2+2t})$, for $t \in [0, \frac{1}{4}]$. The attack is based on taking shorter chains as in HMT, but more of them. Given that the chains are shorter we need to collect plaintexts from N^{4t} bins to obtain enough pairs for the distinguisher. Then, when we process the second chain, we only consider a pair of outputs if they correspond to plaintexts from the same bin. This variant of the symmetric attack is presented in Figure 3.21.

Input : $\tilde{O}(N^{1/4+t})$ chains C^r of $\tilde{O}(N^{3/2-2t})$ plaintexts encrypted under K and $T = T_L||T_R$
Input : $\tilde{O}(N^{1/4+t})$ chains $\hat{C}^s$ of $\tilde{O}(N^{3/2-2t})$ plaintexts encrypted under K and $T' = T_L \oplus 4||T_R \oplus 4$
Output: Slid chains $C^i, \hat{C}^j$ and their respective offset

for *all chains* C^r **do**
  Initialize a hash table H_1
  Insert all the plaintexts $(L_0^i, R_0^i) \in C^r$ into H_1 indexed by R_0^i
  Take $O(N^{4t})$ bins (each with $O(N^{1/2-2t}$ plaintexts)
  Denote the plaintexts by $X_{i_1}, X_{i_2}, \ldots, X_{i_v}$
  for *all chains* $\hat{C}^s$ **do**
    for *all respective offsets* $u = 0, \ldots, N^{3/2-2t}$ **do**
      Extract $(\hat{L}_0^{u+i_1}, \hat{R}_0^{u+i_1}), (\hat{L}_0^{u+i_2}, \hat{R}_0^{u+i_2}), \ldots, (\hat{L}_0^{u+i_v}, \hat{R}_0^{u+i_v})$ from $\hat{C}^s$
      Denote these values as "ciphertexts" $C_{i_1}, C_{i_2}, \ldots, C_{i_v}$
      Initialize $O(N^{4t})$ hash tables H_2^j
      for *all* k=1,...,v **do**
        if X_{i_k} *is from bin* j **then**
          Store in H_2^j the value $LH(C_{i_k}) - LH(X_{i_k})$
      Count the number of collisions in all H_2^j
      if *number of collisions is greater than* $\frac{1.6}{N} \cdot \Sigma_{B \in bins} \binom{|B|}{2}$ **then**
        Ask for the extension of C^r and $\hat{C}^s$ to $\tilde{O}(N^{3/2})$ values.
        Call the PRF-recovery procedure with C^r as inputs and $\hat{C}^s$ shifted by u as the outputs.

Figure 3.21: Time-data trade-off variant of the symmetric slide for FF3.

Cycle structure attack

This attack follows the steps of the paper "Improved slide attacks" by Biham et al. [33] to find candidate slid chains. Consider a related-tweak slid pair (L_0, R_0) and $(\hat{L}_0, \hat{R}_0)$, i.e. a 4-round

FF3 with key K and T partially encrypts (L_0, R_0) into $(\hat{L}_0, \hat{R}_0)$. If we start a chain of encryption from (L_0, R_0), we are assured to reach (L_0, R_0) again after some number of encryptions $t \leq N^2$. Due to the slid property, the same is true also for $(\hat{L}_0, \hat{R}_0)$.

It is clear that this value does not repeat before t encryptions, as otherwise (L_0, R_0) would close the chain earlier. Hence, there is no point to check whether two chains can be slid chains, if their cycle length is not equal. The attack tries to find chains which are actually cycles of length $\tilde{O}(N^{3/2})$ (as this is the amount of data needed for the PRF reconstruction).

Following the Shepp and Llyod's results it is reasonable to assume that: such a cycle exists and that it is unique. If, by chance, there are two cycles in the encryption under K and T of exactly the same length of $\tilde{O}(N^{3/2})$, we can just try all pairs of chains or just take the next larger cycle. Once this pair of cycles is identified, one can run the distinguisher for all possible $\tilde{O}(N^{3/2})$ offsets. Since the distinguisher has cost of $\tilde{O}(N^{1/2})$, the total time complexity is $\tilde{O}(N^2)$. When the correct offset is identified, it is possible to run the PRF reconstruction attack as we have obtained $\tilde{O}(N^{3/2})$ input/output pairs for 4-round FF3. The data complexity of the attack is about $O(N^2)$ encryptions.

We can use two approaches to develop the attack:

- An adaptive chosen-plaintext attack. The attack would be based on picking a random plaintext, generating a cycle from it, and then check whether the cycle has the right length. If it has not the right length, an unseen plaintext need to be picked and the process is repeated. The process is expected to finish after exploring almost all plaintexts.

- A known plaintext attack. This approach consist to collect $N^2 - \sqrt{N}$ known plaintext pairs. If all the values in the cycle of length $\tilde{O}(N^{3/2})$ are not in the missing $\sqrt{N}$ ones, which happens with constant probability, then the cycle can be identified and used for the attack.

To summarize the first phase of the attack takes $O(N^2)$ data and $\tilde{O}(N^2)$ time. In Figure 3.22 the *known plaintext* variant is reported.

Input : $N^2 - N$ known plaintexts (P^i, C^i) encrypted under K and $T = T_L || T_R$
Input : $N^2 - N$ known plaintexts $(\hat{P}^i, \hat{C}^i)$ encrypted under K and $T' = T_L \oplus 4 || T_R \oplus 4$
Output: Slid chains $C, \hat{C}$

```
Initialize a bitmap B of N^2 bits to 0.
while no cycle C of size Õ(N^{3/2}) was found do
    Pick the first plaintext whose bit is not set in B — P_0.
    Set B[P_0] = 1, Set t = 0
    repeat
        Set P_{t+1} = C_t(= E_{K,T}(P_t))
        if P_{t+1} is not in the dataset then
            break (goto 2)
        Set B[P_{t+1}] = 1; Set t = t + 1
    until P_t = P_0;
    if t = Õ(N^{3/2}) then
        Set C to be P_0, P_1, ..., P_{t-1}
Initialize a bitmap B of N^2 bits to 0.
while no cycle Ĉ of size t was found do
    Pick the first plaintext whose bit is not set in B — P̂_0.
    Set B[P̂_0] = 1, Set s = 0
    repeat
        Set P̂_{s+1} = Ĉ_s(= E_{K,T}(P̂_s))
        if P̂_{s+1} is not in the dataset then
            break (goto 2)
        Set B[P̂_{s+1}] = 1; Set s = s + 1
    until P̂_s = P̂_0;
    Set Ĉ to be P̂_0, P̂_1, ..., P̂_{t-1}
for all possible offsets do
    Call the differential distinguisher for any offset between C and Ĉ
    if the distinguisher succeeds then
        Call the PRF reconstruction attack with C,Ĉ, and the offset
```

Figure 3.22: The cycle structure slide distinguisher for FF3.

For this algorithm a *bitmap* is used instead of the usual *hash table*.

Asymmetric slide attack

The Asymmetric slide attack follows the steps of the low data distinguisher presented in the Symmetric slide attack, but offers:

- an improved distinguishing algorithm;
- a trade-off curve.

The data e memory complexity is $\tilde{O}(N^{2-t})$, while time complexity is $\tilde{O}(N^{2+t})$, for $t \in [0, \frac{1}{2}]$. This related-tweak slide differential distinguisher uses the minimal amount of pairs $O(N\ logN)$, similarly to the one of the Symmetric attack. The key element in it is the algorithmic gain, coming from searching the pairs from the plaintext's side.

Consider an input chain of $\tilde{O}(N)$ values. We first pre-process the chain by computing for each of the input pairs with a common right half P_i, P_j, the value $(LH(P_j) - LH(P_i), j - i)$ and storing it in an hash table. In practice we store the difference in the left half and the location difference. From the output side, we take a chain of length $\tilde{O}(N)$. We initialize $\tilde{O}(N)$ counters to zero. Then we can compute, for each such pair (C'_i, C'_j), the value $(LH(C'_j) - LH(C'_i), j' - i')$ and find the offset it proposes in the table. We increment the $\tilde{O}(1)$ counters related to the offset. For the correct offset the amount of pairs that "succeed" is expected to be $\frac{2m}{n}$ out of m pairs, compared with $\frac{m}{n}$ for wrong offsets. If the pre-processed input chains are all keyed into the same hash table, this search can be done simultaneously against all $\tilde{O}(N^{1-t})$ of them, taking only $\tilde{O}(N^2)$ time per output chain.

In Figure 3.23 the Asymmetric slide attack is given.

Input : $\tilde{O}(N^{1-t})$ chains C^r of $q = \tilde{O}(N)$ plaintexts encrypted under K and $T = T_L || T_R$
Input : $\tilde{O}(N^t)$ chains $\hat{C}^s$ of $q = \tilde{O}(N)$ plaintexts encrypted under K and $T' = T_L \oplus 4 || T_R \oplus 4$
Output: Slid chains $C^i, \hat{C}^j$ and their respective offset

Initialize a hash table H_1
for *all chains* $C^r_k \in C^r$ **do**
  for *all* i, j *where* $R^j_k = R^i_k$ **do**
    Store in H_1 in location $(j - i, L^j_k - L^i_k)$ the value (k, i)
for *all chains* $\hat{C}^s_{k'} \in \hat{C}^s$ **do**
  Initialize $\tilde{O}(N^{2-t})$ counters
  for *all* $i' = 0, 1, \ldots, \tilde{O}(N)$ **do**
    for *all* $j' = i' + 1, i' + 2, \ldots, \tilde{O}(N)$ **do**
      for *all* $k, i \in H_1[j' - i', L^{i'}_{k'} - L^{i'}_{k'}]$ **do**
        if $i' < i$ **then**
          Increment the counter of chain k and offset $i - i'$
  Identify k, v such that $counter[k][v]$ is maximal
  if $counter[k][v] > 1.6 \cdot \binom{q}{2} \cdot \frac{1}{N^2}$ **then**
    Ask for the extension of C^r_k and $\hat{C}^s_k$ to $\tilde{O}(N^{3/2})$ values.
    Call the PRF reconstruction with the chains $C^r_k, \hat{C}^s_{k'}$, with offset v

Figure 3.23: The asymmetric slide distinguisher for FF3.

The PRF reconstruction procedure

The PRF reconstruction procedure follows the footsteps of DV and HMT. We use a **graph** where cycles are searched for. Following the HMT approach, we call the PRF reconstruction fewer times than there are candidate slid chains. However, we use cycles of larger size, we pick $L = 4$ and $L = 5$, rather than $L = 3$. This means that for finding sufficiently large connected component between all the values, it is sufficient that from any node in the graph, there will be only $\tilde{O}(N^{1/2})$ outgoing edges. Hence, we have to find cycles of length L in a graph of $\tilde{O}(N)$ nodes, with an average out degree of $\tilde{O}(N^{1/L})$.

The algorithm performs a simple Meet In The Middle procedure:

- from each node we detect all possible $\tilde{O}(N^{1/L})^{\lfloor L/2 \rfloor}$ nodes in distance $\lfloor L/2 \rfloor$,

- and then detect all the possible $\tilde{O}(N^{1/L})^{\lceil L/2 \rceil}$ nodes in distance minus $\lceil L/2 \rceil$ (walking on the reversed edges graph),
- and find a collision between these sets (which is a cycle of length L).

As in DV and HMT, once the cycles are found, all the involved nodes are assumed to be *good nodes*, and they can be used to determine values for F_0. Heuristically, it has been found out that filling in $\frac{1}{3}\ log(N)\sqrt{N}$ values for F_0 gives a high chance of success for the recovery attack on F_1, F_2, F_3 (as proposed in HMT). If the reconstruction is consistent with the slid chain, then we continue to reconstruct the missing values in F_0 and apply the recovery attack in the second half (swapping the order of the slid chains w.r.t. input/output). If the reconstruction is inconsistent with the slid chain, we try a different value to start the assignment from (a different connected component) or try a different chain (when there are other candidates). This part is similar to that of HMT.

3.6 Linear Cryptanalysis of FF3-1 and FEA (2021)

The paper [22] develops new distinguishing and message-recovery attacks on small-domain Feistel ciphers with alternating round tweaks. The attacks are based on *linear cryptanalysis* (deepen in Section 3.6.1), but go beyond standard methods in several ways. The proposed distinguishers only need access to the ciphertext of an arbitrary constant message under many half-constant tweaks.

The message-recovery attacks, instead, follow the security model introduced by Bellare et al. in 2016. For FEA-1, the message-recovery attack should be used to set up a key-recovery attack.

The attacks rely on the ability to vary the tweaks in even-numbered rounds (FF3-1 and FEA-1) or rounds numbered by a multiple of three (FEA-2), while keeping the tweaks in the other rounds fixed. Combined with the observation that the variance of the correlation of a non-trivial linear approximation over a small function is quite large, this results in strong linear trails through the cipher. The results of FF3-1 is of theoretical interest as an application of the theory of linear cryptanalysis over the group $\mathbb{Z}/N\mathbb{Z}$. The data requirements of the basic liner distinguisher were reduced using multidimensional linear cryptanalysis. Based on the same principle, efficient message-recovery attacks were obtained. For FEA-1, the message-recovery attack was in turn extended to key-recovery attack.

For many instances of FF3-1, FEA-1 and FEA-2, the data requirements of the new attacks are small enough to be a practical concern for users of these standards.

3.6.1 Linear Cryptanalysis

Linear cryptanalysis was proposed in 1993 by Misturu Mastui, who first published an attack on DES, which broke the cipher using 2^{47} known-plaintexts. This type of attack is one of the two major statistical attack techniques and design criteria for block ciphers. The other type of attack is the *differential attack* which is generally attributed to Eli Biham and Adi Shamir, who used in the late 1980s this technique in order to analyse some known block ciphers, discovering a theoretical weakness in the DES cipher. However Biham and Shamir noted that DES was surprisingly resistant to differential cryptanalysis. In 1994, a member of the IBM team which designed DES, Don Coppersmith, published a paper stating that differential cryptanalysis was known to IBM as early as 1974, and that defending against differential cryptanalysis was a design goal. This event points out that linear and differential cryptanalysis can be used in the design phase as well as by an attacker.

The main idea behind linear cryptanalysis is to take advantage of high probability occurrences of linear equations (expressed as the equality of two expressions which consists of binary variables combined with the XOR operation) involving plaintext bits, ciphertext bits (more precisely bits from the second to last round output) and subkey bits.

Every linear equation is then used as a distinguisher to recover the key. Indeed, linear cryptanalysis is a known plaintext attack: that is, the attacker has information on a set of plaintexts and the corresponding ciphertexts; however, the attacker has not the possibility to freely select the particular plaintext (this would be the case of a chosen-plaintext attack). In many applications it is reasonable to assume that the attacker has a random set of plaintext and corresponding ciphertexts.

In order to have an idea on how to approximate a cipher (that is a non linear function) by a linear equation we can start by considering the easiest example: a simple logical gate. We take the AND-gate (represented in Figure 3.24) and compare its truth table with some linear functions.

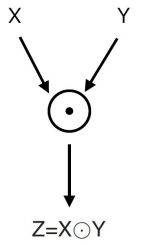


Figure 3.24: AND gate.

Input		Output	Linear functions			
x	y	x ⊙ y	0	x	y	x ⊕ y
0	0	0	0	0	0	0
0	1	0	0	0	1	1
1	0	0	0	1	0	1
1	1	1	0	0	0	0
Probability			3/4	3/4	3/4	1/4

Figure 3.25: Truth table.

As we can see from Figure 3.25 we get three suitably linear approximations for $z = x \odot y$ that are correct with probability 3/4. However in our approximations we do not limit only to linear functions, but we can also consider affine functions. Practically in this context *affine* means that i can take a linear function and XOR a constant with it; by doing so the last linear function reported in the table can be XORed with the constant 1 and we obtain the linear function $x \oplus y \oplus 1$ with probability 3/4.

In the end we observe four linear functions with probability 3/4: 0, x, y, $x \oplus y \oplus 1$.

Now we can move to a more complex scenario: we would like to approximate the S-box of the PRESENT [34] cipher. PRESENT is a lightweight block cipher, designed for scenarios where hardware optimizations is the priority. The cipher has block size of 64 bits, while the key size can be 80 bit or 128 bit. The non-linear layer is based on a single 4-bit S-box, graphically reported in Figure 3.26 and in Figure 3.27.

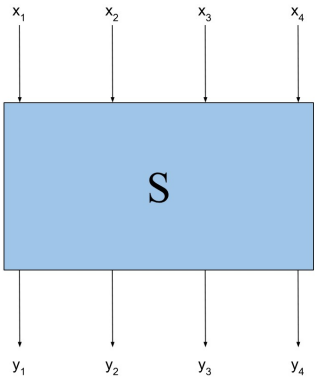


Figure 3.26: 4-bit S-box of the PRESENT cipher.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S(x)$	c	5	6	b	9	0	a	d	3	e	f	8	4	7	1	2

Figure 3.27: Lookup table of the PRESENT S-box.

In this case we do not want to known a complete linear representation of the S-box, but we want to find just some linear relations between the input and the output. We select some of the input bits (x_1, x_2, x_3, x_4) and some of the output bits (y_1, y_2, y_3, y_4) and XOR them together in order to see if the result is 0 or 1. If it is 0 it means that the approximation is correct, if it is 1 it means that the approximation is not correct.

In the example below in Figure 3.28 the bits (x_1, x_4, y_4) are selected. The definition of the S-box in binary representation is translated in Figure 3.29.

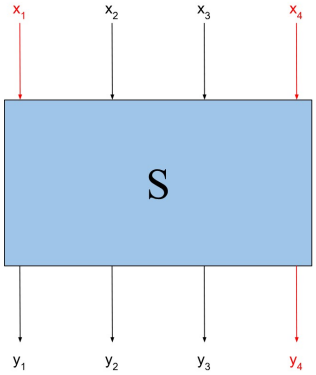


Figure 3.28: S-box with the selected bits highlighted in red.

x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	$y_4 = x_1 \oplus x_4$
0	0	0	0	1	1	0	0	✓
0	0	0	1	0	1	0	1	✓
0	0	1	0	0	1	1	0	✓
0	0	1	1	1	0	1	1	✓
0	1	0	0	1	0	0	1	×
0	1	0	1	0	0	0	0	×
0	1	1	0	1	0	1	0	✓
0	1	1	1	1	1	0	1	✓
1	0	0	0	0	0	1	1	✓
1	0	0	1	1	1	1	0	✓
1	0	1	0	1	1	1	1	✓
1	0	1	1	1	0	0	0	✓
1	1	0	0	0	1	0	0	×
1	1	0	1	0	1	1	1	×
1	1	1	0	0	0	0	1	✓
1	1	1	1	0	0	1	0	✓
Probability								12/16

Figure 3.29: S-box probability table.

From the probability table it is clear that there is a relevant tendency that the approximation $y_4 = x_1 \oplus x_4$ is a good one. Now it is easy to pass from the previous representation into a *mask* representation, where the mask is a decimal number that selects which of the bits are taken into account in the linear approximation, Figure 3.30.

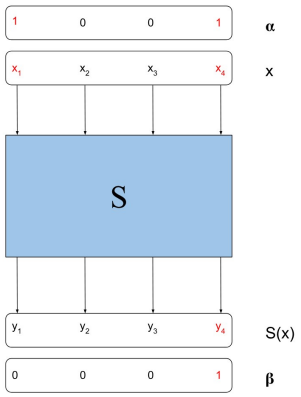


Figure 3.30: Linear masks corresponding to the tuple (x_1, x_4, y_4).

The linear approximation can be expressed by the following formula: $\alpha \cdot x = \beta \cdot S(x)$, where in the previous example $\alpha, \beta \in \mathbb{F}_2^4$, and α is the input mask, while β is the output mask. In this example we have $\alpha = 9$ and $\beta = 1$.

The quality of every approximation defined by the couple (α, β) of a generic b-bit S-box can be measured using one of the following metrics:

- Solutions. $s = |\{x \in \mathbb{F}_2^b | \alpha \cdot x = \beta \cdot S(x)\}|$, counts the number of valid solutions present. $s = 12$ in the previous example.
- Probability. $p = \mathbb{P}_x[\alpha \cdot x = \beta \cdot S(x)] = s/2^b$, probability expressed as number of solutions over number of possible input values. $p = \frac{12}{16}$ in the previous example.

- Bias. $\epsilon = p - \frac{1}{2}$, the bias works as a distinguisher. It tells how much the approximation differs from a random configuration (represented by the quantity $\frac{1}{2}$).
  $\epsilon = \frac{1}{4}$ in the previous example.
- Correlation. $cor = 2 \cdot \epsilon$, the bias is a normalized version of the bias, it varies from -1 to $+1$.
  $cor = \frac{1}{2}$ in the previous example.

Assume that there is a linear approximation $\alpha \cdot x = \beta \cdot S(x)$ that holds with bias ϵ:

- If $\epsilon = 0$ nothing can be learned, the approximation is as good as a random guess, correct half of the times.
- If $\epsilon > 0$, the approximation $\alpha \cdot x = \beta \cdot S(x)$ is good.
- If $\epsilon < 0$, the approximation $\alpha \cdot x = \beta \cdot S(x) \oplus 1$ is good.

Since there are many possible masks, $2^4\alpha$ and $2^4\beta$ in this case, a method to evaluate how good is each of these combinations is to use the so-called *linear approximation table* (LAT). The LAT lists the quality of every possible mask $\text{LAT}[\alpha, \beta] = s - 2^{b-1} = 2^b\epsilon$.

α/β	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	-4	0	-4	0	0	0	0	0	-4	0	4
2	0	0	2	2	-2	-2	0	0	2	-2	0	4	0	4	-2	2
3	0	0	2	2	2	-2	-4	0	-2	2	-4	0	0	0	-2	-2
4	0	0	-2	2	-2	-2	0	4	-2	-2	0	-4	0	0	-2	2
5	0	0	-2	2	-2	2	0	0	2	2	-4	0	4	0	2	2
6	0	0	0	-4	0	0	-4	0	0	-4	0	0	4	0	0	0
7	0	0	0	4	4	0	0	0	0	-4	0	0	0	0	4	0
8	0	0	2	-2	0	0	-2	2	-2	2	0	0	-2	2	4	4
9	0	4	-2	-2	0	0	2	-2	-2	-2	-4	0	-2	2	0	0
a	0	0	4	0	2	2	2	-2	0	0	0	-4	2	2	-2	2
b	0	-4	0	0	-2	-2	2	-2	-4	0	0	0	2	2	2	-2
c	0	0	0	0	-2	-2	-2	-2	4	0	0	-4	-2	2	2	-2
d	0	4	4	0	-2	-2	2	2	0	0	0	0	2	-2	2	-2
e	0	0	2	2	-4	4	-2	-2	-2	-2	0	0	-2	-2	0	0
f	0	4	-2	2	0	0	-2	-2	-2	2	4	0	2	2	0	0

Figure 3.31: Linear approximation table of PRESENT S-box.

Chapter 4

Proof of concept

4.1 JDBC

In this section we want to give a brief introduction to the JDBC tool, which is used for the implementation of the PoC.

The Java Database Connectivity (JDBC) is a Java application programming interface (API) that can access a tabular data. It is used especially for data stored in a relational database and it provides methods to manage activities like: connection to a database, send queries and update statements to the database, retrieve and process the results received from the database. The JDBC API supports both two-tier and three-tier processing models for database access:

- In the two-tier architecture the Java applications can connect directly to the database server. This model requires a JDBC driver in order to communicate with the particular database that is accessed. The database can be located on another machine with respect to the user one; the user is connected to the server via a network. The two-tier architecture corresponds to a client/server configuration, with the user's machine as the client and the machine housing the database as the server.
- In the three-tier architecture the commands formulated by the user are sent to a middle tier of *services*, which then sends the commands to the database. The database processes the commands and sends back the results to the services tier, which forwards them to the user. The advantages of this model are the possibility of maintain control over access (control the user activity) and the simplification of the applications' deployment.

JDBC includes four major components:

1. The JDBC API module, which constitutes the real API. It provides programmatic access to relational data from the Java programming language. The JDBC API allows applications to execute SQL statements, retrieve results, propagate changes to an eventual underlying data source. It can interact with multiple data sources in a distributed environment. The JDBC API is comprised of two packages: `java.sql` and `javax.sql` (which is an extension of the first one).
2. The JDBC Driver Manager module. It has a class called `DriverManager` that defines objects which can connect Java applications to a JDBC driver. Specifically, it loads a database-specific driver in an application to establish a connection with a database.
3. The JDBC Test Suite module. It is used to test the operations performed by specific JDBC driver, such as: *insert, delete, update*.
4. The JDBC-ODBC Bridge module, which connects the database driver to the database. It provides JDBC access via ODBC drivers, translating the JDBC method call to the ODBC

function call. It is important to say that in order to use this module the programmer needs to load ODBC binary code onto each client machine that is using this driver.

The first two components are used to connect to a database and create a Java application that uses SQL commands to communicate with the database, which is the case of implementation reported in this chapter. The last two components are designed for specific environments where the programmer wants to test web applications or communicate with a DBMS that is using ODBC.

As already said, a JDBC driver is a client-side adapter (installed on the client machine) that converts the request from Java programs to a protocol that the database can understand. In order to connect with individual databases, JDBC requires drivers for each database.

There are several implementations of JDBC drivers, that fall into four categories:

- **Type 1**, also known as JDBC-ODBC bridge, is a database driver implementation that uses the ODBC driver to connect to the database. The driver translates JDBC method calls into ODBC function calls. This type of driver is platform-dependent as it makes use of ODBC which depends on native libraries; this fact limits its portability. Furthermore it is not suitable for a high-transaction environment and it does not support the complete Java command set. When Java first came out, the JDBC-ODBC bridge was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available. It reached end of support from JDK 1.8 (Java 8). An example of type 1 driver is the JDBC-ODBC bridge implemented by the Sun company (now acquire by Oracle).
- **Type 2**, also known as Native-API driver, is a database driver implementation written partly in the Java programming language and partly in native code. This driver uses a native client library specific to the database to which it connects. Also in this case the portability is limited, because of the usage of native code. An example of type 2 driver is the JDBC OCI driver (OCI stands for Oracle Call Interface).
- **Type 3**, also known as Network-Protocol (or middleware driver), is a database driver implementation that uses a pure Java client and communicate with a middleware (a middle tier) server using a database-independent protocol. The middleware server then communicates the client's requests to the database. In this case there is the advantage that the same client-side JDBC driver can be used for multiple databases. In addition this driver is platform independent, because the platform differences are taken care of by the middleware. The use of the middleware gives additional advantages in terms of security. The main difference between the type 3 and the type 4 driver is that the protocol conversion logic stays not in the client, but in the middleware.
- **Type 4**, also known as Database-Protocol driver or Thin driver, is a database driver implementation that uses pure Java and implements the network protocol for a specific database. In this case the client interacts directly with the database. Since it is completely written in Java, the type 4 driver is platform independent. Given the fact that the database protocol is vendor specific, the JDBC client requires separate drivers, supplied by the vendor, in order to connect to different types of databases. The drivers are installed inside the Java virtual machine (JVM) of the client; this facilitates debugging, since the JVM manages the application-database connection.

4.2 Application design

The implementation of the PoC is constituted by a Java application that connects to a relational database. The Java application comes in two different version:

- a plain version for database access;
- a version using methods of Format Preserving Encryption.

With this PoC we want to demonstrate that the overhead added for the employment of FPE is acceptable with respect to the benefits introduced with it. The implementation makes use of MySQL, an open-source relational database management system (RDBMS), which gives the possibility to work with the operating system for the purpose of create a relational database locally in the computer's storage system. Furthermore MySQL helps users management, allows network access and facilitates testing the database integrity. For the MySQL driver, the version installed and used is the MySQL Connector/J 8.0.30, that is a Type 4 driver.

The application models an online shop, which manages four major components: products, customers, payments and orders. The model is reported in the Entity-Relation model in Figure 4.1. The database, called *shop_online*, was created using the MySQL tools and it is composed of eleven tables. After launching the application the user can walk through a series of menus which internally modify the tables of the database. For example, regarding the customer, the user can: insert a new customer, associate an address to a customer, associate a payment method to a customer, create a new order for a specific customer. The customer can have multiple addresses associated, which is the case of a customer which has various houses. Furthermore, the same address can be associated to multiple customers, which is the case of different customers living in the same house with the same address. The customer can also have various payment methods associated. A payment method is represented by a type, which is the name of the payment method, and by an UUID, Universally unique identifier. In general, an UUID is a 128-bit label used to uniquely identify an object in a computer system (in this case the payment method). The UUID standard is defined in [35], that was recently updated in [36]. In our implementation the UUID is represented in his canonical form, a string of 32 hexadecimal characters displayed in five groups divided by dashes, following the schema 8-4-4-4-12, for a total number of 36 characters. The UUID in an possible scenario could be created using some of the data of the payment method or of the customer associated with the payment method. Thus, in that case it is important to assure that the privacy is maintained and the application gives the possibility to encrypt or not the UUID using FPE methods. The eventual encryption of the UUID is tracked into the database using a flag inside the table that manages the payment methods. The order, that can be also called "cart", can be filled with various order items. Each order item is associated with one product, which can be part of various categories inside the shop. A single order can be associated to only one customer, while a customer can make various orders. Finally we can also try to perform a payment of a specific order with a specific payment method. The payment is successful only if the order that we want to pay and the payment method used are associated with the same customer. This means that the customer could not pay with a payment method which is not possessed by him and, at the same time, he could not pay an order which is not instantiated by him.

4.3 Developer manual

The development of the software was made using Eclipse IDE, which gives the possibility to code with pure Java as well as using external libraries. The MySQL Connector was installed in the system and imported in the `Classpath` as an external JAR (Java archive). For FPE we used an open-source implementation of the NIST approved FF3 and FF3-1 algorithms called *mysto*, by Privacy Logistics [37]. This open-source library (version 1.0) was imported as a Maven Dependency inside the `Classpath`. The library has two external dependencies, Log4j and JUnit, which were as well imported as Maven dependencies inside the `Classpath`, using version 2.24.1 for Log4j and version 3.8.1 for JUnit.

The application is composed of two principal files and other eight files which are one-to-one mappings of the tables presented in the DB (available at `src.com.shop_online.classes`). The first principal file, the `Main.java`, is available at `src.com.shop_online` and it starts the application prompting a command line interface where the user can pass through four menus and make operations with MySQL DB. This file manages the input/output operations and it does the following operations: creates the objects used to fill and update the DB, retrieves the list of objects which are stored in a table and checks the atomicity of the transactions, committing if the transaction succeeds completely and rolling back if one of the operation inside the transaction fails. In this way we ensure the consistency, preserving DB invariants, such as referential integrity.

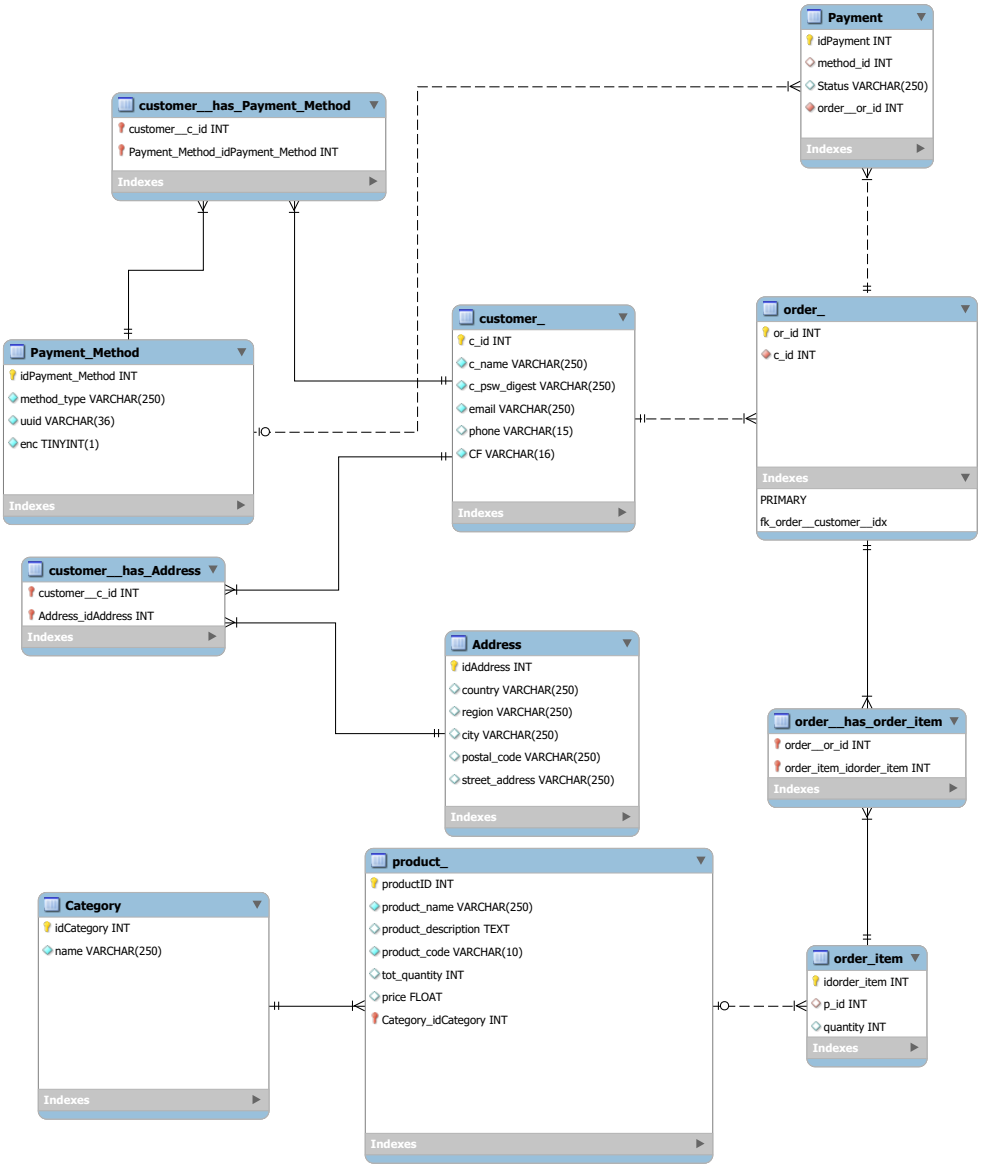


Figure 4.1: ER model.

As an example, it is not possible to insert an Order_item in the corresponding table if the Order specified does not exist, even if the Order_item is a valid one (referring to a Product which exists in the DB). The second principal file is the `DataSource.java`, available at `src.model`. This file is imported as class in the `Main` and receives input data from it. The `DataSource` works as a "middle-layer" between the `Main` and the DB. It uses as a starting point the class `MysqlDataSource`, imported from `com.mysql.cj.jdbc.MysqlDataSource`, and it is responsible of opening and closing the connection with the DB. In Figure 4.2 it is shown the method used to open the connection. `DataSource` performs also the operations of inserting, updating, deleting and reading data to/from the DB, using the SQL language. For every method we prepared the query that we want to provide to the DB and we employ a `Statement`. In a particular, we defined

```java
public Connection openConnection() throws IOException {
    try {
        MysqlDataSource ds = new MysqlDataSource();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        ds.setServerName("localhost");
        ds.setPort(3306);
        System.out.println("Insert the mysql user: ");
        String user = br.readLine();
        ds.setUser(user);
        ds.setUser("root");
        System.out.println("Insert the mysql password: ");
        String psw = br.readLine();
        ds.setPassword(psw);
        ds.setPassword("laga");
        ds.setDatabaseName("shop_online");
        ds.setUseSSL(false);
        ds.setAllowPublicKeyRetrieval(true);

        con = ds.getConnection();
        if(con != null) {
            System.out.println("Connection successfully opened");
            return con;
        }
    } catch (SQLException e) {
        System.out.println("Error: " + e.getMessage());
    }
    return null;
}
```

Figure 4.2: Method to open the connection with the DB.

```java
public void updateCustomer(Customer c, int id) {
    try {
        int cnt = 0;
        String query = "UPDATE " + TABLE_CUSTOMER + " SET";
        if(c.getName() != null) { cnt++; query += " " + COLUMN_CUSTOMER_NAME + " = ?";}
        if(c.getPsw_digest() != null) { cnt++; if(cnt==1) {query += " " + COLUMN_CUSTOMER_PSW + " = ?";} else {query += ", " + COLUMN_CUSTOMER_PSW + " = ?";}}
        if(c.getEmail() != null) { cnt++; if(cnt==1) {query += " " + COLUMN_CUSTOMER_EMAIL + " = ?";} else {query += ", " + COLUMN_CUSTOMER_EMAIL + " = ?";}}
        if(c.getPhone() != null)  { cnt++; if(cnt==1) {query += " " + COLUMN_CUSTOMER_PHONE + " = ?";} else {query += ", " + COLUMN_CUSTOMER_PHONE + " = ?";}}
        if(c.getCf() != null) { cnt++; if(cnt==1) {query += " " + COLUMN_CUSTOMER_CF + " = ?";} else {query += ", " + COLUMN_CUSTOMER_CF + " = ?";}}

        query += " WHERE " + COLUMN_CUSTOMER_ID + " = ?";
        if(cnt==0) {
            System.out.println("No parameters to modify");
            return;
        }

        PreparedStatement ps = con.prepareStatement(query);
        cnt = 0;
        if(c.getName()  != null) { cnt ++; ps.setString(cnt, c.getName()); }
        if(c.getPsw_digest()    != null) { cnt ++; ps.setString(cnt, c.getPsw_digest());  }
        if(c.getEmail() != null) { cnt ++; ps.setString(cnt, c.getEmail());}
        if(c.getPhone() != null) { cnt ++; ps.setString(cnt, c.getPhone());}
        if(c.getCf() != null) { cnt++; ps.setString(cnt, c.getCf());}

        cnt++;
        ps.setInt(cnt, id);


        int affRows = ps.executeUpdate();
        if (affRows > 0)
            System.out.println("Modification completed");
        else
            System.out.println("Modification not carried out");
    } catch (SQLException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

Figure 4.3: Example of `PreparedStatement`.

a `PreparedStatement`, which is an interface that accepts input parameters at runtime. In this way we can generalize the methods in order to accept various type of input data. This is clear from Figure 4.3, where we construct the query and the statement on the basis of the input data. Finally we execute an update of the DB in the cases of "insert", "update" and "delete", while in the case of the "read" we save the results in a `ResultSet` object, that is easily mapped into the corresponding table-class.

4.4 User manual

The application was ran and tested in a Windows system, but similar steps are valid for Linux and macOS. First of all, in order to use the application we need to download MySQL from his site. Here we can select the version, the operating system and the type of file to download. For Windows the MSI installer can be selected. After the download of the file, we can start the installing procedure, which redirects us to a setup page where we have to choose the type of installation. At this point we can either go with a default installation (that installs all products

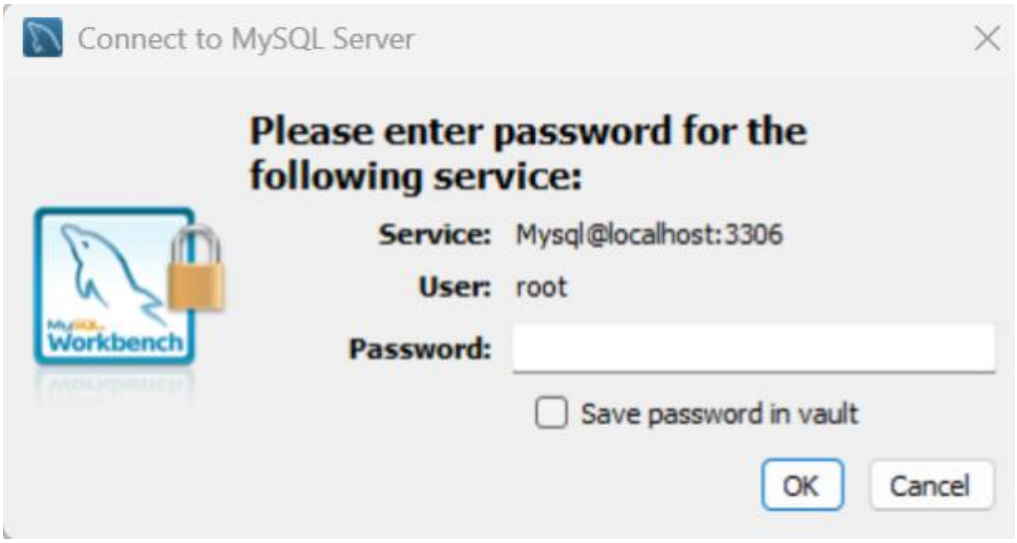


Figure 4.4: Connection to the MySQL server using root credentials.

of MySQL) or with a custom installation, where we can select the installation of MySQL Server and MySQL Workbench that together constitute the minimum software required. Subsequently we have to setup the networking, where we must be sure that the connection to the Server is done with TCP/IP on port 3306 (that is the default option). Afterwards we have to create a password for the root user. This password is very important because, together with the "root" user name, is required by the *shop_online* application for the connection to the DB. After finishing the installation, MySQL Workbench, a visual database design tool, should open up. Here we can see the default connection instance, running on the localhost on port 3306, having the root user as default. If we double click on this instance, a window similar to the one in Figure 4.4 opens and we have to insert the password previously created for the root. Once entered the connection instance we have on the left a panel which reports some default DB created together with the MySQL Workbench installation, while on the right we have an SQL editor. In order to create our DB we can open an SQL script inside this editor and run it. The script is given together with the application and it is called `generate_shop_online_from_EER.sql`. After the creation of the DB we can now populate it with some data given in `populate_shop_online.sql` or directly pass to the *shop_online* application.

In order to run the application, Java must be installed. We can choose to run it either with the executable JAR file provided, called `shop_online_app.jar` or by importing directly the source files of the application in an IDE (i.e. Eclipse or IntelliJ IDEA). In Windows, if we decide to run the JAR file we have to open to command prompt, move to the directory where the file is present and type `java -jar shop_online_app.jar` to start the application. If we instead want to run the application from an IDE we have to import the project folder and we have to install all the Maven dependencies previously discussed in 4.3. Then we have to run, with the procedure depending on the IDE, the file `Main.java` which contains the entry point of the application. When the application starts, it asks for username and password, that must be the same credentials used in the instance of MySQL server. If the credential are correct, the connection will be opened and we will enter the main menu of the program.

4.5 Test

To conclude, we implemented some tests in order to verify the performances of the application when the UUID is encrypted with FPE before the insertion in the DB and when it is not encrypted. The test was done using JUnit4 and it consists in a simple test of the method `payment_method_insert_case_1` contained in `Main.java`. In particular, in the set up phase we measured the initial time with the method `System.nanoTime()` (which returns the current value

	insertions	100	1 000	10 000
trial #1	FF3-1	1 372	4 843	33 203
	plain	1 159	4 275	27 565
trial #2	FF3-1	1 400	4 708	34 296
	plain	1 002	3 871	26 702
trial #3	FF3-1	1 354	4 851	31 804
	plain	1 025	4 253	26 447
trial #4	FF3-1	1 331	4 801	34 617
	plain	1 119	3 975	26 537
trial #5	FF3-1	1 343	4 835	32 509
	plain	1 144	4 139	26 571
Difference on average		270	705	6 521

Figure 4.5: Time in milliseconds (ms) for the tests on `testPayment_method_insert_case_1`.

of the running JVM's high-resolution time source, in nanoseconds) and we opened the connection. Then, in the test, we execute a fixed number of insertions in the DB, using the method cited previously and a boolean flag that tracks if the test must be done with or without encryption. Finally, in the tear down phase we closed the connection and we computed the total time duration. The tests were run in a Huawei Matebook 14 machine with the following specifications:

- CPU: AMD Ryzen 5 4600;
- GPU: AMD Radeon(TM) Graphics;
- RAM: 8GB DDR4;
- Storage: 512 GB SSD;
- OS: Windows 11 Home, version 24H2.

We decided to replicate the test using different number of insertions, repeating it in 5 trials. The results of this approach are synthesised in Figure 4.5 (here the time is expressed in ms).

To wrap up we can make the following considerations on the results obtained:

1. The difference in terms of performance between the two versions, although constant in the various trials, can be considered acceptable.
2. The trials done with the same number of insertions and with the same version give similar results.
3. The difference becomes more consistent when the number of insertions increases, but it has still the same order of magnitude w.r.t. a single test.
4. For all of the trials, the time for a single insertion becomes smaller when the total number of insertions increases. As an example, in trial #1 with FF3-1 the time for a single insertion is: 13.7 ms when we employ 100 insertions, 4.8 ms when we employ 1.000 insertions and 3.3 ms when we employ 10.000 insertions. The initial overhead, noticed when the number of insertions is limited, is due to opening and closing of the connection with the DB.

From the test done we can finally state that the trade-off between the costs, in terms of performances, and the advantages, given by the encryption (security) and by the preserving property (reusing of existing systems), leads to consider FPE as the preferred choice.

Chapter 5

Conclusions

In this thesis, Format-preserving encryption techniques were described, analysed and applied in the context of databases. In particular, it was highlighted the importance of applying these techniques for companies working with legacy databases. FPE permits companies to implement protection of data using the encryption, so as to respect national or supranational regulations such as GDPR, and at the same time avoids violations in existing format constraints, which could lead to extensive redesign and refactoring of a business application. The interest for this new cryptographic tool is supported by the great commitment towards the FPE study, of several companies such as: *Voltage Security*, *Verifone*, *Ingenico*, *Cisco*. Through this thesis, we tried to compare the benefits, minimal schema impact and minimal storage impact, and the drawbacks, performance overhead (w.r.t. a scenario when no encryption is applied) and security (w.r.t. to standard encryption techniques), of FPE.

We started from an analysis on the state of the art of the main FPE schemes, covering the ones standardized by the NIST in SP800-38G, where FF1 and FF3 are presented as modes of operation for an underlying approved symmetric-key block cipher algorithm. Then we tried to describe the historical evolution of FPE from the first investigation by Black and Rogaway which gives proves on the security of some schemes and it is considered a cornerstone in this research scope. After that we went through all of the proposals taken into consideration by the NIST for a new standard. The first proposal was the FFSEM scheme, which evolved in the FFX scheme, which similarly was expanded, becoming the first NIST standard with the name FF1. NIST standardized also other two schemes, the BPS-BC scheme with the name FF3 and the VAES3 with the name FF2. In a second moment NIST decided to remove FF2 from the final version of SP800-38G, because it was shown that FF2 did not provide 128-bits of security strength. A counter-proposal of VAES3, called DFF, was submitted to NIST, but it was not standardized.

After the design presentation of the schemes, we passed to evaluate the robustness. Several attacks on FPE schemes, developed in the recent years, were discussed and inspected, as well as the various attack methodology. For each attack algorithm, the attack complexity and applicability were given in order to make a critical comparison. Among the various attack, the two most relevant were the Message-Recovery Attack, which forced NIST to recommend a larger lower limit on the number of inputs for FF1 and FF3, and the DV attack, a total break attack on FF3 which caused a revision of the standard by the NIST, resulting in a new scheme called FF3-1.

Lastly we implemented a proof-of-concept of an application which works with a relational database, modelling an online shop. This part of the thesis was designed and discussed in collaboration with *Aruba Software Factory SRL*. For the development we chose the Java programming language, together with the JDBC API and MySQL. From the application it was clear that the trade-off between the costs, in terms of performances, and the advantages, given by the encryption (security) and by the preserving property (reusing of existing systems), leads to consider FPE as the best alternative.

Even though the results given by the tests are consistent, the work could be expanded and improved with a larger test platform. This would give the possibility to evaluate the results in a more systematic way. This work can be considered as a starting point to try to propose new FPE

schemes and to improve existing ones. Furthermore, the verification of the security of existing FPE schemes, with statistical attack techniques (i.e. linear and differential cryptanalysis) and other frameworks, can be used to improve the actual standards.

Bibliography

[1] "PA digitale 2026", https://padigitale2026.gov.it/misure/

[2] M. Dworkin, "DRAFT Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption", SP 800-38G (DRAFT), July 2013

[3] NIST, "FIPS PUB 81. DES modes of operation", 1980, https://csrc.nist.gov/files/pubs/fips/81/final/docs/fips81.pdf

[4] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption", SP 800-38G, March 2016, DOI 10.6028/NIST.SP.800-38G

[5] NIST, "FIPS PUB 74. Guidelines for implementing and using the NBS Data Encryption Standard", 1981, https://nvlpubs.nist.gov/nistpubs/legacy/fips/fipspub74.pdf

[6] M. Brightwell and H. Smith, "Using datatype-preserving encryption to enhance data warehouse security", NISSC-1997: 20th National Information Systems Security Conference, Baltimore (MD, USA), October 7-10, 1997. https://csrc.nist.gov/files/pubs/conference/1997/10/10/proceedings-of-the-20th-nissc-1997/final/docs/141.pdf

[7] J. Black and P. Rogaway, "Ciphers with arbitrary finite domains", Topics in Cryptology - CT-RSA 2002, San Jose (CA, USA), February 18-22, 2002, pp. 114–130, DOI 10.1007/3-540-45760-7_9

[8] T. Spies, "Feistel Finite Set Encryption Mode", 2008, https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/ffsem/ffsem-spec.pdf

[9] M. Bellare, P. Rogaway, and T. Spies, "The FFX Mode of Operation for Format-Preserving Encryption", February 2010, https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/ffx/ffx-spec.pdf

[10] M. Bellare, P. Rogaway, and T. Spies, "Addendum to "The FFX Mode of Operation for Format-Preserving Encryption"", September 2010, https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/ffx/ffx-spec2.pdf

[11] E. Brier, T. Peyrin, and J. Stern, "BPS: a Format-Preserving Encryption Proposal", April 2010, https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/bps/bps-spec.pdf

[12] J. Vance, "VAES3 scheme for FFX An addendum to "The FFX Mode of Operation for Format-Preserving Encryption"", May 2011, https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/ffx/ffx-ad-vaes3.pdf

[13] M. Dworkin and R. Perlner, "Analysis of VAES3 (FF2)", Cryptology ePrint Archive, Paper 2015/306, April 2015, https://eprint.iacr.org/2015/306

[14] J. Vance and M. Bellare, "An Extension of the FF2 FPE Scheme", July 2014, https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/dff/dff-ff2-fpe-scheme-update.pdf

[15] F. B. Durak and S. Vaudenay, "Breaking the ff3 format-preserving encryption standard over small domains", Advances in Cryptology - CRYPTO 2017, Santa Barbara (CA, USA), August 20-24, 2017, pp. 679–707, DOI 10.1007/978-3-319-63715-0_23

[16] M. Dworkin, "Rev.1-DRAFT Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption", SP 800-38G REV. 1 (DRAFT), February 2019, DOI 10.6028/NIST.SP.800-38Gr1-draft

[17] V. T. Hoang, S. Tessaro, and N. Trieu, "The curse of small domains: New attacks on format-preserving encryption", Advances in Cryptology - CRYPTO 2018, Santa Barbara (CA, USA), August 19-23, 2018, pp. 221–251, DOI 10.1007/978-3-319-96884-1_8

[18] M. Bellare, V. T. Hoang, and S. Tessaro, "Message-recovery attacks on feistel-based format preserving encryption", CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna (Austria), October 24-28, 2016, pp. 444–455, DOI 10.1145/2976749.2978390

[19] J. Patarin, "Security of random feistel schemes with 5 or more rounds", Advances in Cryptology - CRYPTO 2004, Santa Barbara (CA, USA), August 15-19, 2004, pp. 106–122, DOI 10.1007/978-3-540-28628-8_7

[20] M. Liskov, R. L. Rivest, and D. Wagner, "Tweakable block ciphers", Advances in Cryptology - CRYPTO 2002, Santa Barbara (CA, USA), August 18-22, 2002, pp. 31–46, DOI 10.1007/3-540-45708-9_3

[21] J.-K. Lee, B. Koo, D. Roh, W.-H. Kim, and D. Kwon, "Format-preserving encryption algorithms using families of tweakable blockciphers", Information Security and Cryptology - ICISC 2014, Seoul (South Korea), December 3-5, 2014, pp. 132–159, DOI 10.1007/978-3-319-15943-0_9

[22] T. Beyne, "Linear cryptanalysis of ff3-1 and fea", Advances in Cryptology - CRYPTO 2021, virtual event, August 16-20, 2021, pp. 41–69, DOI 10.1007/978-3-030-84242-0_3

[23] S. Dara and S. Fluhrer, "Fnr: Arbitrary length small domain block cipher proposal", Security, Privacy, and Applied Cryptography Engineering, Pune (India), October 18-22, 2014, pp. 146–154, DOI 10.1007/978-3-319-12060-7_10

[24] U. T. Mattsson, "Format-controlling encryption using datatype-preserving encryption", Cryptology ePrint Archive, Paper 2009/257, 2009, https://eprint.iacr.org/2009/257

[25] F. B. Durak, H. Horst, M. Horst, and S. Vaudenay, "Fast: Secure and high performance format-preserving encryption and tokenization", Advances in Cryptology - ASIACRYPT 2021, Singapore, December 6-10, 2021, pp. 465–489, DOI 10.1007/978-3-030-92078-4_16

[26] T. Baignères, J. Stern, and S. Vaudenay, "Linear cryptanalysis of non binary ciphers", Selected Areas in Cryptography, Ottawa, Canada, August 16-17, 2007, pp. 184–211, DOI 10.1007/978-3-540-77360-3_13

[27] D. Chang, M. Ghosh, A. Jati, A. Kumar, and S. K. Sanadhya, "espf: A family of format-preserving encryption algorithms using mds matrices", Security, Privacy, and Applied Cryptography Engineering, Goa (India), December 13-17, 2017, pp. 133–150, DOI 10.1007/978-3-319-71501-8_8

[28] F. B. Durak and S. Vaudenay, "Breaking The FF3 Format-Preserving Encryption Standard Over Small Domains", rwc.iacr-2018-slides-Durak, 2018, https://rwc.iacr.org/2018/Slides/Durak.pdf

[29] A. Biryukov and D. Wagner, "Slide attacks", Fast Software Encryption, Rome (Italy), March 24-26, 1999, pp. 245–259, DOI 10.1007/3-540-48519-8_18

[30] V. T. Hoang, D. Miller, and N. Trieu, "Attacks only get better: How to break ff3 on large domains", Advances in Cryptology - EUROCRYPT 2019, Darmstadt (Germany), May 19-23, 2019, pp. 85–116, DOI 10.1007/978-3-030-17656-3_4

[31] A. Biryukov and D. Wagner, "Advanced slide attacks", Advances in Cryptology - EUROCRYPT 2000, Bruges (Belgium), May 14-18, 2000, pp. 589–606, DOI 10.1007/3-540-45539-6_41

[32] O. Amon, O. Dunkelman, N. Keller, E. Ronen, and A. Shamir, "Three third generation attacks on the format preserving encryption scheme ff3", Advances in Cryptology - EUROCRYPT 2021, Zagreb (Croatia), October 17-21, 2021, pp. 127–154, DOI 10.1007/978-3-030-77886-6_5

[33] E. Biham, O. Dunkelman, and N. Keller, "Improved slide attacks", Fast Software Encryption, Luxembourg (Luxembourg), March 26-28, 2007, pp. 153–166, DOI 10.1007/978-3-540-74619-5_10

[34] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher", Cryptographic Hardware and Embedded Systems - CHES 2007, Vienna (Austria), September 10-13, 2007, pp. 450–466, DOI 10.1007/978-3-540-74735-2_31

[35] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC-4122, July 2005, DOI 10.17487/RFC4122

[36] K. Davis, B. Peabody, and P. Leach, "Universally Unique IDentifiers (UUIDs)", RFC-9562, May 2024, DOI 10.17487/RFC9562

[37] Privacy Logistics, `https://github.com/mysto/java-fpe`