APPLICATION OF FAULT ANALYSIS TO SOME CRYPTOGRAPHIC STANDARDS

Onur Duman

A thesis

in

The Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science in Information Systems
Security
Concordia University
Montréal, Québec, Canada

June 2016

© Onur Duman, 2016

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Onur Duman**

Entitled: **Application of Fault Analysis to Some Cryptographic Standards**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Information Systems Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. C. Assi ______________________________ Chair

Dr. A. Youssef ______________________________ Supervisor

Dr. C. Assi ______________________________ CIISE Examiner

Dr. W. Hamouda ______________________________ External Examiner (ECE)

Approved ____

Chair of Department or Graduate Program Director

______________ 20 ______ ______________________________

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

ABSTRACT

Application of Fault Analysis to Some Cryptographic Standards

Onur Duman

Cryptanalysis methods can be classified as pure mathematical attacks, such as linear and differential cryptanalysis, and implementation dependent attacks such as power analysis and fault analysis. Pure mathematical attacks exploit the mathematical structure of the cipher to reveal the secret key inside the cipher. On the other hand, implementation dependent attacks assume that the attacker has access to the cryptographic device to launch the attack. Fault analysis is an example of a side channel attack in which the attacker is assumed to be able to induce faults in the cryptographic device and observe the faulty output. Then, the attacker tries to recover the secret key by combining the information obtained from the faulty and the correct outputs. Even though fault analysis attacks may require access to some specialized equipment to be able to insert faults at specific locations or at specific times during the computation, the resulting attacks usually have time and memory complexities which are far more practical as compared to pure mathematical attacks.

Recently, several AES-based primitives were approved as new cryptographic standards throughout the world. For example, Kuznyechik was approved as the standard block cipher in Russian Federation, and Kalyna and Kupyna were approved as the standard block cipher and the hash function, respectively, in Ukraine. Given the importance of these three new primitives, in this thesis, we analyze their resistance against fault analysis attacks.

Firstly, we modified a differential fault analysis (DFA) attack that was applied on AES and applied it on Kuzneychik. Application of DFA on Kuznyechik was not a trivial task because of the linear transformation layer used in the last round of Kuznyechik. In order to bypass the effect of this linear transformation operation, we had to use an equivalent representation of the last round which allowed us to recover the last two round keys using a total of four faults and break the cipher.

Secondly, we modified the attack we applied on Kuzneychik and applied it on Kalyna. Kalyna has a complicated key scheduling and it uses modulo 2^{64} addition operation for applying the first and last round keys. This makes Kalyna more resistant to DFA as compared to AES and Kuznyechik but it is still practically breakable because the number of key candidates that can be recovered by DFA can be brute-forced in a reasonable time. We also considered the case where the SBox entries of Kalyna are not known and showed how to recover a set of candidates for the SBox entries.

Lastly, we applied two fault analysis attacks on Kupyna hash function. In the first case, we assumed that the SBoxes and all the other function parameters are known, and in the second case we assumed that the SBoxes were kept secret and attacked the hash function accordingly. Kupyna can be used as the underlying hash function for the construction of MAC schemes such as secret IV, secret prefix, HMAC or NMAC. In our analysis, we showed that secret inputs of Kupyna can be recovered using fault analysis.

To conclude, we analyzed two newly accepted standard ciphers (Kuznyechik, Kalyna) and one newly approved standard hash function (Kupyna) for their resistance against fault attacks. We also analyzed Kalyna and Kupyna with the assumption that these ciphers can be deployed with secret user defined SBoxes in order to increase their security.

Acknowledgments

Firstly, I would like to thank my supervisor, Dr. Amr Youssef. He has always been caring and encouraging during my graduate studies. He showed an excellent guidance and he was always available whenever I needed to discuss anything about my research or even things not related to my research. He also showed amazing leadership skills which made working under his supervision both a pleasure and a great learning experience.

Secondly, I would like to thank my labmates, Riham AlTawy, Ahmed Abdelkhalek, Mohamed Nabil, Abdullah Al-Barakati and Afaf Moussa, for their friendship and sharing their experiences with me generously.

Thirdly, I would like to thank all my instructors in CIISE. This list includes Dr. Ayda Basyouni, Dr. Makan Pourzandi, Dr. Jeremy Clark, Dr. Lingyu Wang and Dr. Mohammad Mannan. They all provided me with the opportunity to learn in a positive learning environment and made me more and more interested in systems security in general.

Last but not least, I would like to thank my parents for always encouraging me to follow my passions and their endless support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

According to Ron Rivest, “Cryptography is about communication in the presence of an adversary” [75]. In this communication, two honest parties, usually referred to as Alice and Bob, communicate over an insecure channel which can be, for example, a telephone line, a wireless channel, the internet, or a company network. Messages can be English text, numerical data or anything as long as they can be understood by both Alice and Bob. Since the channel is insecure, messages sent between Alice and Bob can also be received by other parties. Those other parties may include an adversary whose goal may be listening to the communication between Alice and Bob without their approval. The adversary is able to read, modify and delete those messages. In order to prevent the adversary from reading the messages and understanding the communication between Alice and Bob, encryption is used. If the messages are encrypted, the adversaries can still read messages between Alice and Bob, but they cannot understand them since they do not know the secret key used in the encryption; in order to perform encryption, ciphers, which map a given plaintext into a ciphertext under a secret key, are used.

There are two kinds of ciphers, namely, block ciphers and stream ciphers. If the block cipher uses the same key for encryption and decryption, it is called symmetric-key block cipher. In this case, the key is called private key since the key has to be only known by the communicating parties. If the block cipher uses different keys for encryption and decryption, it is called asymmetric key block cipher. In asymmetric key block ciphers, each party has a public key and a private key. The ciphertext is generated using the public key of the receiving party and that ciphertext is decrypted by the receiver using the receiver's private key. The public key is known by all participants but the private key has to be kept secret by the receiver. Stream ciphers encrypt individual characters of the input plaintext one at a time using a keystream. If the keystream is generated independently from the plaintext messages and of the ciphertext, it is called synchronous stream cipher. If the keystream is generated as a function of the key and a fixed number of previous ciphertext characters, it is called self-synchronizing (asynchronous) stream cipher [57].

Even though ciphers prevent the adversary from understanding the communication between Alice and Bob, they do not prevent the adversary from modifying or deleting messages without being detected. In order to achieve that, integrity mechanisms are needed. In order to provide message integrity, hash functions are used as building blocks. Hash functions are mappings which take a message as input and generate a fixed size output called hash code [57]. In the case of communication between Alice and Bob, if Alice wants to send a message M to Bob, Alice calculates the hash of the message $H(M)$ and sends $(M, H(M))$ to Bob. After that, if Bob wants to make sure the message has not been modified, Bob calculates $H(M)$ from M and compares this calculated value with the hash value sent by Alice. If these values match, Bob is sure that the message has not been (accidentally) modified. However, an attacker may intercept the communication, modify M to M' and $H(M)$ to $H(M')$ and send $(M', H(M'))$ to Bob. In that case, Bob is not able to detect that the message has been modified. Also, Bob cannot be sure that the message comes from Alice. This shows that hash functions alone are not enough to detect modifications by attackers and they provide no message authenticity. Integrity mechanisms, such as MAC (Message Authentication Code) schemes or digital signature algorithms are needed for message integrity and authenticity. MAC schemes can be built using a hash function with a secret key. In order to verify integrity and authenticity of a message using a MAC scheme, the receiving party has to know the key used in the MAC scheme. In addition to MAC schemes, digital signature algorithms can also be used for message integrity and authenticity. Signatures are generated using a private signature key and verification of the signature is performed using a public verification key. So, in the case of digital signature algorithms, the public verification key is known by all participants but the private signature key has to be only known by the entity signing the message. In most attacks, the adversary may only be interested in reading the messages between two or more communicating parties. In this case, the adversary is called passive adversary. However, the adversary may also be interested in changing messages or impersonating the identity of an honest party.

In this case, the adversary is called an active adversary. A passive adversary threatens confidentiality of the data. An active adversary threatens integrity and authenticity of the data in addition to its confidentiality [57].

If the adversaries succeed in their goal after performing an attack on a cryptographic system, we say that the cryptographic system is broken. According to [57], a block cipher is totally broken if the secret key can be found, and partially broken if an adversary is able to recover part of the plaintext (but not the key) from the ciphertext. In order to totally break a block cipher, the adversary may try all possible keys in the key space of the cipher. This is called brute force key search [11]. However, in some cases, brute force key search can have a very large time complexity. Time complexity is measured by the number of operations required to apply the algorithm. As an example, in order to recover a private key with size 128 bits, the average number of operations required to perform brute force key search is $\frac{2^{128}}{2} = 2^{127}$. Considering that total number of electrons in the universe is in the order of 8.37×2^{77} [57], 2^{127} is clearly a very large number. Also, time complexity of the attack grows exponentially as the key size grows. So, brute force key search is usually not a practical approach unless the key space of the block cipher is small. Cryptanalysis can simply be defined as finding ways to totally or partially break a cipher without performing brute force key search. Attacks on block ciphers can be classified as ciphertext only attacks, known plaintext attacks, chosen plaintext attacks or chosen ciphertext attacks [72]. In ciphertext only attacks, the adversary only knows a set of ciphertexts generated by the cryptosystem. In known plaintext attacks, the adversary knows a set of

$(plaintext, ciphertext)$ pairs. In chosen plaintext attacks, the adversary chooses a plaintext and asks for its corresponding ciphertext. In chosen ciphertext attacks, the adversary chooses a ciphertext and asks for its corresponding plaintext. In chosen plaintext/ciphertext attacks, we assume that the adversaries have access to an encryption/decryption oracle and they can query it with plaintext/ciphertext of their choice.

Cryptanalysis techniques in the literature can be classified as mathematical attacks and implementation dependent attacks. Examples of mathematical attacks include linear cryptanalysis [54], differential cryptanalysis [15], integral attacks [48] and meet-in-the-middle attacks [28]. Mathematical attacks aim to exploit the mathematical structure of the cipher. Implementation dependent attacks include side channel analysis (SCA) [45] and fault analysis [16]. In side channel analysis, the adversary aims to retrieve the secret key using side channel information. Side Channel Analysis (SCA) exploits information which is leaked by default while the device is running, such as power usage [42], time taken to perform the cryptographic operations [12], and acoustics of noise generated during the calculations [35]. In fault analysis, the adversary disrupts the computation by injecting faults and aims to retrieve the secret information by comparing the correct and faulty results. For example, a smartcard which contains an embedded processor and presumably stores some secret key information can be subjected to high temperature outside its operating range, unsupported supply voltage, or strong magnetic field, to influence the operation of the processor. In this case, the processor may begin to output incorrect computation results due to physical data corruption, which may help the attacker to deduce the instructions that the processor is executing, or the internal data state, including the secret key. Another example is where the fault attack is applied on DVB Common Scrambling algorithm (CSA). Many Pay-TV devices in Europe rely on this algorithm to secure the media broadcasting. So, if an attacker retrieves the secret key used in this algorithm, the attacker is able to listen to channels without paying [77].

Fault analysis was first introduced by Boneh *et al.* in order to retrieve the private signature key in the RSA-CRT algorithm [17]. Later, this idea was combined with differential cryptanalysis by Biham and Shamir where Differential Fault Analysis (DFA) was introduced in [16]. In DFA, the attacker retrieves information about the secret key by utilizing the difference between the correct and faulty computation results.

While fault analysis attacks may require access to some specialized equipment in order to be able to insert faults at specific locations/time, the resulting attack time and memory complexities are usually far more practical as compared to pure mathematical attacks. Consequently, when a new cipher is proposed, it is important to examine its resistance against fault analysis. This is particularly important for the case of standardized algorithms given the potential of wide deployment in different operating environments. The Russian Federation has been using GOST 28147-89 [38] as the national standard cipher. This standard was also being used by many other CIS (Commonwealth of Independent States) countries such as Ukraine. Recently, the Russian Federation decided to change their standard cipher and created a new standard cipher called Kuznyechik [30]. Ukraine decided to change their standard cipher as well since there were theoretical attacks presented against GOST 28147-89 and its software implementation was significantly slower on modern platforms. Ukraine chose Kalyna [61], which was the winner of the Ukrainian National Public Cryptographic

Competition (2007-2010), as their standard block cipher. After Kalyna had been accepted, its modified version was made the new standard cipher in Ukraine. Given the importance of Kuznyechik and Kalyna, in this thesis, we decided to analyze their resistance against fault analysis attacks.

In addition to block ciphers, it is also important to analyze hash functions for their resistance against fault analysis attacks when these functions are used as the underlying primitive for message authentication code (MAC) schemes using secret IV, secret prefix, NMAC or HMAC constructions. The Kupyna hash function [60], is based on the cipher Kalyna, and has been recently chosen as a standard hash function in Ukraine. In the last part of this thesis, we investigate the resistance of Kupyna against fault analysis attacks.

1.2 Contributions

This thesis has three main contributions:

- We show how to apply differential fault analysis on Kuznyechik if everything about the cipher is known except for the secret key. In particular, we show that the DFA technique that is applied on AES [62] can also be applied on Kuznyechik, with only a small modification to count for the fact that, unlike AES, the linear transformation layer in the last round of Kuznyechik is not omitted. According to our analysis, the key of Kuznyechik can be recovered using an average of four faults. This work was included in [2].

- We apply differential fault analysis on Kalyna and show that it is more resistant to the same attack compared to AES and Kuznyechik. We also show how to apply fault analysis on Kalyna with secret SBoxes and show how the attacker can recover the SBox entries.

- We apply DFA on the Kupyna hash function when it is used as the underlying primitive for MAC schemes using secret IV, secret prefix, NMAC or HMAC constructions. We also analyze it when it is used with secret SBoxes. According to our analysis, recovering each state byte requires an average of 2.21-2.42 faults depending on the used fault model.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we first present general definitions of block ciphers and hash functions. After that, we present a brief literature review of mathematical attacks and implementation dependent attacks. In Chapter 3, we present our DFA on Kuznyechik and in Chapter 4, we present our attacks on Kalyna. In Chapter 5, we present our fault analysis on Kupyna. Finally, in Chapter 6, we present a conclusion and some suggestions for future research directions.

Chapter 2

Background and Related Work

In this chapter, we review the definition of block ciphers and hash functions. We also briefly explain some of the most commonly used mathematical attacks against these primitives, including linear cryptanalysis, differential cryptanalysis, integral cryptanalysis and meet-in-the-middle attacks. After that, implementation dependent attacks, which include side channel analysis and fault analysis, are described.

2.1 Definitions of Cryptographic Primitives

2.1.1 Block Ciphers

A block cipher is a function which maps n-bit plaintext blocks to n-bit ciphertext blocks. The function is parametrized by a k-bit key K, which takes values from a subset κ(the keyspace) which includes all possible k-bit binary vectors V_k, where it is generally assumed that K is chosen at random. An $n-bit$ block cipher is a function $E : V_n \times K \rightarrow V_n$, such that for each $K \epsilon \kappa$, $E(P, K)$ is an invertible mapping, the encryption function for K from V_n to V_n, written as $E_K(P)$. The inverse mapping is the decryption function, denoted as $D_K(C)$. $C = E_K(P)$ denotes that ciphertext C results from encrypting P under K [57].

According to [76], a block cipher is secure if it behaves as a pseudorandom permutation. In other words, it must be secure against distinguishing attacks: no efficient algorithm, given interactive access to encryption and decryption black boxes, should be able to distinguish the real cipher (i.e., E_K and D_K) from a truly random permutation. In order to design a cipher which randomizes a given plaintext in a way that the ciphertext is not distinguishable from a random sequence of bits, two properties are introduced by Shannon which are confusion and diffusion [68]. According to [57], the confusion property is intended to make the relationship between the key and ciphertext as complex as possible. Diffusion refers to rearranging or spreading out the bits in the message so that any redundancy in the plaintext is spread out over the ciphertext. There is no simple function that provides both confusion and diffusion, so modern ciphers are made of composition of simple functions. The SPN (Substitution-Permutation Networks) structure is mostly used in modern ciphers, such as AES [27], where the confusion property is achieved by using substitution boxes and the diffusion property is satisfied using permutation operation and other linear transformations such as Galois Field multiplication. These operations are applied in many rounds with a round key mixing at each round to increase the security. In what follows, we define some of the building blocks that are used for constructing SPNs.

- **Substitution Layer (S):** This function is a non-linear bijective mapping between two vectors. In most ciphers such as AES and Kuznyechik, the substitution layer is built by applying byte-to-byte mapping ($V_8 \rightarrow V_8$) to all state bytes. However, in some other ciphers, the input and the output vectors in the mapping are not of the same size. As an example, in DES [22], the substitution layer uses eight mappings between 6-bit binary vectors to 4-bit binary vectors ($V_6 \rightarrow V_4$) that make up the 48-bit binary state. Since this substituting layer is usually the only non-linear component in the cipher, weaknesses in substitution boxes can be exploited in many attacks, which makes their design a very challenging task.

- **Permutation Layer (P):** This function usually operates on the whole state where it reorders the bits in the state. In AES, Kuznyechik, Kalyna and Kupyna, reordering is performed on the byte level.

- **Linear Transformation Layer (L):** This function either operates on a subset of bytes in the state or on the whole state where the output of this layer is obtained by applying a linear transformation over its input. In AES, Kalyna and Kupyna, the linear transformation is applied column by column. However, in Kuznyechik, the linear transformation layer is applied to all the state bytes.

Another commonly used block cipher structure is the Feistel network [59]. A Feistel cipher is an iterated cipher mapping of a $2t$-bit plaintext (L_0, R_0), for $t - bit$ blocks L_0 and R_0, to a ciphertext (R_r, L_r), through an r-round process. For $1 \leq i \leq r$, round i maps $(L_{(i-1)}, R_{(i-1)}) \rightarrow (L_i, R_i)$ as follows: $L_i = R_{(i-1)}, Ri = L_{(i-1)} \oplus f(R_{(i-1)}, K_i)$, where each subkey K_i is derived from the cipher key K. The f function of the Feistel cipher may be a product cipher, though f itself need not be invertible to allow inversion of the Feistel cipher. Decryption is thereby achieved by using the same r-round process but with subkeys used in the reverse order, K_r through K_1 [57]. DES, which was adopted as a national standard in the United States in 1977 [22] is an example of a block cipher which was designed as a Feistel Network.

The key addition (X) is also a part of the round operations in both SPN and Feistel structures. In order to perform key addition, firstly, the round keys for each round are calculated using the input key given by the user. This process is called key scheduling. The addition of round keys to the current state in each round is called key mixing. The key mixing is crucial to the security of the cipher since it adds randomization. If there was no key mixing, the attacker would be able to find the inverse of encryption function used. According to Kerckhoffs's Principle, the attacker knows everything about the cipher except the secret key [47].

Figure 1 shows a schematic view of the round functions in both SPN and Feistel Networks.

2.1.2 Hash Functions

Hash functions are used to construct short fingerprints of some data. If the data changes, the fingerprint also changes [72]. According to [57], a hash function H is a function which has at least the following two properties:

1. Compression: H maps an input x of an arbitrary finite bit length to an output $H(x)$ of finite bit length n.

2. Ease of computation: Given H and input x, $H(x)$ is easy to compute.

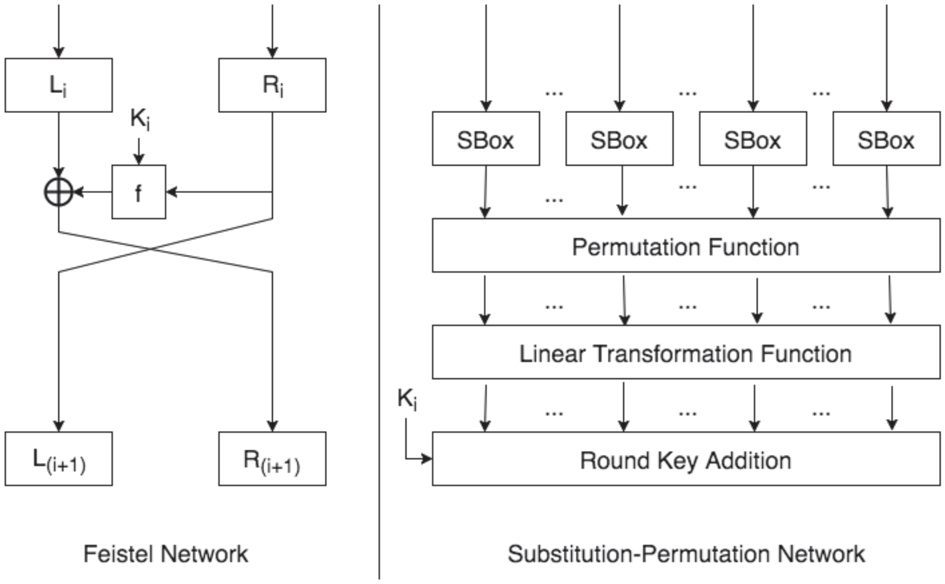


Figure 1: Round function for round i in Feistel Network and SPN

In addition to the above two basic properties, cryptographic hash functions also need to satisfy the following three properties:

- **One wayness:** Given a message M, calculating $H(M)$ should be easy but given $H(M)$, calculating M should be hard.
- **Strong Collision Resistance:** It should be computationally infeasible to find any two messages, M, M', such that $M \neq M'$ but $H(M) = H(M')$.
- **Weak Collision Resistance:** Given a message M, it is computationally infeasible to find another message M' such that $M \neq M'$ but $H(M) = H(M')$. If a hash function is not weak collision resistant, then it is not strong collision resistant but the opposite is not true.

2.2 Attacks on Cryptographic Primitives

2.2.1 Mathematical Attacks

2.2.1.1 Linear Cryptanalysis

Linear cryptanalysis is a known plaintext attack which was first applied on DES by Matsui [54]. The goal is to get a linear approximation for a given cipher algorithm. In order to achieve this, a statistical linear path between input and output bits of each SBox in the cipher is constructed. More formally, the purpose is to find an effective linear expression for a given cipher algorithm [54] such that:

$$P[i_1, i_2, ..., i_a] \;\oplus\; C[j_1, j_2, ..., j_b] \;=\; K[k_1, k_2, ..., k_c] \tag{1}$$

holds with probability $p \; \neq \frac{1}{2}$, where $i_1, i_2, \cdots, i_a, j_1, j_2, \cdots, j_b, k_1, k_2, \cdots, k_c$ denote fixed bit locations in the equation 1. The magnitude of equation 1 is calculated as $\mid \; p - \frac{1}{2} \mid$ and it defines the effectiveness of the approximation. The goal here is to find the subset of bits that maximizes the magnitude of Equation 1.

Matsui [54] showed that 8-round DES can be broken using 2^{21} known plaintexts and the full DES can be broken using 2^{47} known plaintexts. Since its introduction, linear cryptanalysis has been applied on many other ciphers (e.g., see [55], [36], [53]).

2.2.1.2 Differential Cryptanalysis

Differential cryptanalysis is a chosen plaintext attack where the attacker exploits the high probability of certain occurrences of plaintext differences and differences into the last round of the cipher [41]. In linear cryptanalysis, finding a linear approximation to the cipher algorithm was achieved by finding linear approximations to the underlying nonlinear components. However, in differential cryptanalysis, the goal is to find the probability that a ciphertext difference ($\Delta\ C = C \oplus C'$) occurs given a plaintext difference has occurred ($\Delta\ P = P \oplus P'$), where $C = E_K(P)$ and $C' = E_K(P')$. For an ideal cipher, this probability should be $\Delta\ Y = (\frac{1}{2})^n$ where n is the block length [41]. Differential cryptanalysis finds the scenarios where ΔC occurs with a high probability given that a particular ΔP has occurred. We call the pair $(\Delta\ P, \Delta\ C)$ a differential. Similar to linear cryptanalysis, differential cryptanalysis analyzes properties of individual SBoxes. As an example, consider the SBox entries used in DES (DES has many SBoxes, we chose first row of its first SBox), given in Table 1:

Input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Output	E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7

Table 1: SBoxes used in DES

Here, for the input difference to the SBox, $\Delta x = 0x0B$, the probability that $\Delta y = 2$ is $\frac{8}{16}$. This is the difference with the highest probability. Also, note that key addition before the substitution layer has no effect on the difference because of the simple mathematical property of the XOR function. Let K be the round key after the substitution layer. In that case, inputs to SBoxes become ($P \oplus K$) and ($P' \oplus K$) and the input difference is $\Delta = P \oplus K \oplus P' \oplus K = P \oplus P'$ which is equal to ΔP.

Differential cryptanalysis was first applied on DES by Biham and Shamir [15] where they showed that they are able to break any reduced round variant of DES (up to 15 rounds) using less than 2^{56} chosen plaintexts. They also showed that differential cryptanalysis can be applied on any kind of DES-like ciphers. By taking the fact that DES can be broken using linear and differential cryptanalysis, resistance against those attacks was taken as an important element among design criteria for AES [26]. On October 2000, NIST (National Institute of Standards and Technology) chose AES, which replaced DES, as the standard block cipher for the United States [32].

In [13], Biham *et al.* introduced impossible differential cryptanalysis. Unlike the classical differential cryptanalysis which aims to find differentials that occur with high probability, this attack is based on finding differences with 0 probability, i.e., differences that are impossible to occur and use them to exclude wrong key guesses during the analysis phase of the attack.

2.2.1.3 Integral Cryptanalysis

After linear and differential cryptanalysis were proposed, a new block cipher, which was resistant to those attacks, was designed by Knudsen *et al.* This block cipher was called SQUARE [25]. For resistance against linear and differential cryptanalysis, they used a design strategy called wide trail design strategy [24] for SBoxes and linear transformation layer. In this design strategy, SBoxes were chosen where the maximum difference propagation probability and the maximum input-output correlation were as small as possible. In addition to the design of the new cipher, Knudsen *et al.* also suggested an attack on that cipher. This attack was called integral cryptanalysis. It is also known as the Square attack since it was first applied on the cipher SQUARE. Integral cryptanalysis is a chosen plaintext attack where the attacker chooses a set of plaintexts which are the same in some bytes and different in some other bytes. Bytes that are same in all plaintexts are called passive bytes and bytes that differ among plaintexts are called active bytes. In this attack, the cryptanalyst changes the active bytes across all possible byte values and analyzes results. Knudsen *et al.* showed that 4-round SQUARE can be broken using 2^9 chosen plaintexts and 6-round SQUARE can be broken using 2^{32} chosen plaintexts [25]. Integral cryptanalysis was also applied on other ciphers such as HIEROCRYPT [9], IDEA [14], CAMELLIA [78], Skipjack [44], MISTY1 [73], SAFER++ [63], and KHAZAD [58].

2.2.1.4 Meet-in-the-middle Attacks

The meet-in-the-middle attack was first introduced by Diffie and Helman in [29]. In the basic form of this attack, the attacker tries to split the cipher into two parts where one is used in the encryption direction and the other is used in the decryption direction. Then, the attacker partially guesses key bits from both ends and propagates her knowledge of the internal state of the cipher until the information propagated in both directions meet in the middle for matching. The key bits are considered wrong if no match is found. Otherwise, the key bits may be a key candidate.

Many variants of meet-in-the-middle attacks were recently proposed. For example, Dunkelman *et al.* [31] presented a meet-in-the-middle attack on reduced round DES. In this attack, rather than guessing the key bits, they guess intermediate encryption values which reduced the time complexity of the attack. The attack was applied on 4-round DES, 5-round DES and 6-round DES. One of the most important meet-in-the-middle attacks in the literature is the one applied on AES by Demirci and Selcuk [28] where they showed that if the truncated differential for 4-round distinguisher has only one active byte that takes all possible values, then the output byte can be evaluated using a function of 25 parameters. Their attack was applied on AES-256 with 7 rounds. Furthermore, they were able to reduce time complexity of the attack using a high-memory to store the pre-computation table [1] which was computed offline. A similar meet-in-the-middle attack was applied on Kuznyechik by Altawy and Youssef [3]. In their attack, they used the differential enumeration approach where they proposed a distinguisher for the middle rounds and matched the sequence of state differences at the output. They showed that for 5-round reduced cipher, 256-bit master key of Kuznyechik can be recovered with a time complexity of $2^{140.3}$, a memory complexity of $2^{153.3}$, and a data complexity of 2^{113}. Variants of meet-in-the-middle attacks were also applied on hash functions (e.g., see [43], [65]).

2.2.2 Implementation Dependent Attacks

2.2.2.1 Side Channel Analysis

Side channel analysis refers to analyzing side channel information of the device during the computation. In what follows, we briefly review some examples of side channel attacks:

- **Timing Attacks:** Timing attacks were first introduced by Kocher [50]. These attacks exploit the fact that cryptosystems take different amounts of time to process different inputs. As an example, the RSA [64] cryptosystem includes modular exponentiation operations $C = M^e \ mod \ n, \ M = C^d \ mod \ n$, where e is the public key and d is the private key. When using the square and multiply algorithm to perform the exponentiation operation, if the bit in the binary representation of the exponent is 1, a multiplication operation is performed. However, if this bit is zero, no multiplication is performed. This means that, if the key bit is 1, it takes longer processing time. Thus observing the time required to perform the decryption operation allows the attacker to find how many bits in the key are equal to 1. However, learning how many bits of the key are 1 does not allow the attacker to recover the whole key directly. So, the attacker also uses statistical information based on many encryption/decryption operations in order to get more information about the key.

  Another timing attack was performed by Brumley and Boneh on SSL [19]. Their attack was able to retrieve the private key used in OpenSSL based web server running on a machine on the local network.

  Another timing attack is the cache timing attack performed on AES by Bernstein [12]. In this attack, the attacker exploits the fact that different SBox entries take different amounts of time in SBox lookup operations. The attacker observes the cases where the SBox lookup operation gives maximum timing and guesses key bits accordingly. To summarize, timing attacks can still be applied on many ciphers and they need to be taken into account while designing and implementing new ciphers.

- **Differential Power Analysis:** In differential power analysis, the attacker aims to get information about secrets in a device, such as keys, using the power usage information of the device. Similar to the timing attack, this attack is also non-invasive, which means that the attacker does not change anything during the calculations performed by the device. Power analysis was first introduced by Kocher *et al.* [49] who applied it to hardware implementations of DES. Differential power analysis requires observing many encryption/decryption operations and capturing the corresponding power traces. In order to prevent differential power analysis, Kocher *et al.* [49] suggested three different techniques which are reducing signal sizes so that the attacker is not able to get information about the secret key from looking at the power usage, introducing noise to power consumption so that the attacker needs to take more samples for getting information about the secret key using power usage analysis, and the last approach is to design the device by making realistic assumptions about the hardware. In [67], Shamir proposed a solution to differential power analysis by decorrelating the external power supplied to the device from the internal power used by the chip. Since this attack may allow an attacker to get secret information from devices without being detected, it is important to check newly designed devices against their resistance to power analysis. This is especially true for devices processing critical information such as credit card readers.

- **Acoustics Cryptanalysis:** In acoustics cryptanalysis, the attacker uses the sound emitted by the device to gain information about the secret key. This idea was applied on RSA in [35] where Genkin *et al.* were able to recover 4096-bits RSA private key within an hour by placing a high sensitive microphone close to the computer performing the decryption. They distinguish operations performed by the CPU according to the sound waveforms. In order to mitigate such analysis, they suggested acoustic shielding where sound signals are absorbed by the shield. Another solution was to use noisy environment where the noise is generated in a way that it cannot be distinguished from the noise generated by the CPU. This attack shows that many different side channels can be found by the attackers.

- **Cold Boot Attack:** A cold-boot attack is a SCA that exploits the fact that data loss of a non-powered random access memory can be retarded by cooling it down [39]. In 2002, Skorobogatov performed experiments to study the temperature dependency of data retention time in static RAM devices [70]. The reported results indicated that many chips may preserve data for relatively long periods of time at temperatures above $-20°$C which contradicted the common wisdom that was widely believed at that time. Thus, one way to launch a cold-boot attack is to remove the memory module, after cooling it, from the target system and immediately plug it in another system under the adversary's control. This system is then booted to access the memory. Further analysis can then be performed against the information that is retrieved from memory in order to find sensitive information such as cryptographic keys or passwords. For example, Halderman *et al.* have developed a recovery algorithm for the 128-bit version of AES-128 that recovers keys from 30% decayed AES-128 Key Schedule images [40].

2.2.2.2 Fault Analysis

All the implementation dependent attacks we have mentioned earlier in this chapter are non-invasive since the attacker does not change anything during the computation. However, fault analysis is invasive since it requires the attacker to intervene the computation by inserting faults. Fault analysis was first introduced by Boneh *et al.* [17] where the private signing key of RSA-CRT algorithm was obtained using a correct signature and a faulty signature for the same message. Later, this idea was combined with differential cryptanalysis by Biham and Shamir where differential fault analysis was first applied on DES [16]. Biham and Shamir claim that even though smart cards are tamper-resistant, meaning that even the owner of the smart card cannot get secrets out of it, they are easy to attack due to their simplicity. In the fault model described in [16] by Biham and Shamir, one fault is introduced in each encryption/decryption and the fault changes one bit in the register either from 0 to 1 or from 1 to 0. The attacker encrypts a known plaintext twice and if resulting ciphertexts are different, the attacker knows that a fault has occurred and can also identify the round where the fault has occurred. If the fault has occurred in the last round, it gives information on bits of the last round key.

Fault injection can be performed using power glitches, tampering with the clock pulses or injecting laser beams [23, 71].

Differential fault analysis (DFA) attacks vary greatly in terms of number of faults required and the way the attacker is able to inject faults. Faults attacks were applied on many ciphers such as AES. In the fault attack applied on AES [62] by Piret and Quisquater, the attacker faults the input to substitution layer in the round before the last one and the attacker faults the state by changing one of the state bytes to a random value. Depending on the byte faulted by the attacker, the attacker is able to retrieve a set of bytes for the last round key. So, in this fault model, the attacker knows that the fault occurs in which state of the computation but the attacker does not have control over which byte to fault. So, this is called a random fault model. Also, the attacker may apply the same fault twice which will lead to gaining no additional information. In another fault attack applied on AES by Ghalaty *et al.* [37], the attacker is assumed to know which state the fault has occurred and the way the attacker applies the fault is by flipping bits from 0 to 1 or from 1 to 0. Also, the attacker is able to define the fault intensity, which is number of bits in a state byte that can be flipped. The attacker does not exactly know which byte has been faulted but the attacker can know that information by analyzing the correct and faulty ciphertexts. This fault attack requires many more faults compared to the attack presented in [62], however, it did not involve complex calculations. In both of these attacks, it was assumed that the attacker knows everything about the cipher except the secret key. This includes values of SBoxes, the permutation function and constants for the linear transformation matrix. However, this may not always be the case. A novel fault attack was applied on AES by Clavier and Wurcker [21] which assumes that the attacker does not know anything except the way the algorithm works. So, in their attack, they assumed that the attacker does not know the SBox entries, the permutation function and linear transformation constants. The attacker just knows these functions are applied but she does not know exactly how. In their fault model, faults were applied by forcing a specific SBox output to 0. So, in that case, the attacker knows which byte has been faulted and also knows the state fault has been applied. They showed that fault analysis can still be applied on AES even when the security through obscurity is used.

In [74], Tunstall *et al.* applied fault attack on AES which consisted of a single random byte fault applied on the input to eight round. After the fault, they used a two-stage algorithm. In the first stage, they reduced the possible set of keys from 2^{128} to 2^{32} and in the second stage, they were able to reduce the number of possible keys to 2^8. In some cases, they ended up with only 2 possible keys. Fuhr *et al.* applied fault attack on AES using faulty ciphertexts only [33]. Their attack was applied on AES-128 using nonuniform fault models. It targeted last 4 rounds and it was able to recover user given key using a limited number of faulty ciphertexts only. This means that the attacker does not have to know the correct plaintext and the correct ciphertext for applying the attack. Battistello and Giraud showed that it is still possible to apply fault attack on AES even when infective computation countermeasure is applied [10]. They showed that it is very difficult to design an effective countermeasure for AES against fault attacks.

Fault attacks were also applied on other cryptographic functions (ciphers and hash functions) such as APE [7], MULTI2 [8], SHACAL-1 [52], and Streebog [4]. In the fault attack applied on APE, Saha *et al.* showed how diagonal fault attack can be applied on APE [7] and how the key candidates can be reduced from 2^{160} to 2^{25}. In the fault attack applied on MULTI2, which is a cipher used in Japan to secure media broadcasting, Aumasson *et al.* [8] were able to recover the secret key uniquely for any number of rounds. In the fault attack applied on SHACAL-1, Li *et al.* [52] showed that 72 random faults are needed to recover 512-bit key successfully with probability 60%. Korkikian *et al.* applied blind fault attack on ciphers with SPN structure [51]. In their attack, they injected faults to the last round of the cipher and applied statistical method on pairs of correct and faulty ciphertexts to retrieve the key. They showed that fault attacks are possible even when the input and output messages to the cipher are not known. Their attack required so many faults which was the price to pay for blindness. As an example, 480000 faults were needed to completely recover AES key. Lastly, in the fault attack applied on the Streebog hash function, Altawy and Youssef [4] showed that secret inputs to the Streebog hash function, when used to build a MAC scheme, can be retrieved using fault analysis. They used three different fault models, namely, random byte fault model, known byte random fault model and known byte unique fault model. In the random byte fault model, the attacker does not know which byte has been faulted. In the known byte random fault model, the attacker knows which byte has been faulted but this fault may have been applied before, i.e., uniqueness of faults is not ensured by the fault injection procedure. In the known byte unique fault model, the attacker knows which byte has been faulted and she is sure that same fault is not applied twice.

2.3 Countermeasures Against Fault Attacks

Since fault attacks may allow attackers to retrieve the secret key from the cryptographic device, it is important to protect cryptographic devices against those attacks. In [46], Joye and Tunstall categorize prevention techniques under three categories which are modular-redundancy, patches, and protocol-based. Modular redundancy is basically achieved by executing the algorithm several times with same inputs to see if outputs are different. Protocol-based solutions contain randomizing the key or the plaintext. As an example, in All-Or-Nothing transform [56], a message is randomized and the random message is encrypted in order to prevent the adversary from gaining information about the original message using fault analysis. There are also general methods which can be applied on any algorithm such as detection of changes or masking.

In order to attack a device, an adversary may reduce the number of rounds [20] since this makes the cipher easier to attack. In order to detect this attack, a second invariant is used which counts backwards from the total number of rounds or a signature can be added to a loop counter which is updated at each iteration.

Another way the adversary can attack a cryptographic device is by modifying the non-volatile parts of the device such as the secret key [6] or SBox entries [66]. This can be prevented using cyclic redundancy check which checks if data has been tampered.

Masking refers to adding random delays during the calculation. This does not prevent the adversary from injecting faults but it makes the job of the adversary more difficult. However, in [18], Boscherand and Handschuh showed that masking does not totally prevent against fault attacks. They were able to recover secret keys from two masked AES implementations using a basic differential fault attack.

Chapter 3

Attacks on Kuznyechik

Kuznyechik is an SPN block cipher that has been chosen recently to be standardized by the Russian Federation as a new GOST cipher. In this chapter, we present a differential fault attack against Kuznyechik. Our attack employs the random byte fault model, where the attacker is assumed to be able to fault a random byte in rounds seven and eight. Using this fault model enables the attacker to recover the master key using an average of four faults. The presented attack has a practical complexity and aims to demonstrate the importance of protecting the hardware and software implementations of the new standard.

3.1 Specification of Kuznyechik

Kuznyechik operates on 128-bits state [$30, 69$]. It gets a 256-bits key input from the user. This key, called the master key, is used to generate 10 sub-round keys $(K_1, K_2, \cdots, K_{10})$ with 128-bits each. The encryption procedure updates the 128-bits state by iterating the round function 9 times as shown in Figure 2. The round function consists of the following operations:

- SubBytes (S): This is a nonlinear bijective mapping and it works on the byte level.
- LinearTransformation (L): This is the diffusion layer which operates on 128 bits (16 bytes). It can be seen as a row left multiplication by a 16×16 byte (MDS) matrix.
- KeyAddition(X): This operation mixes the round key with the state bytes. Key mixing is done by a bitwise XOR operation.

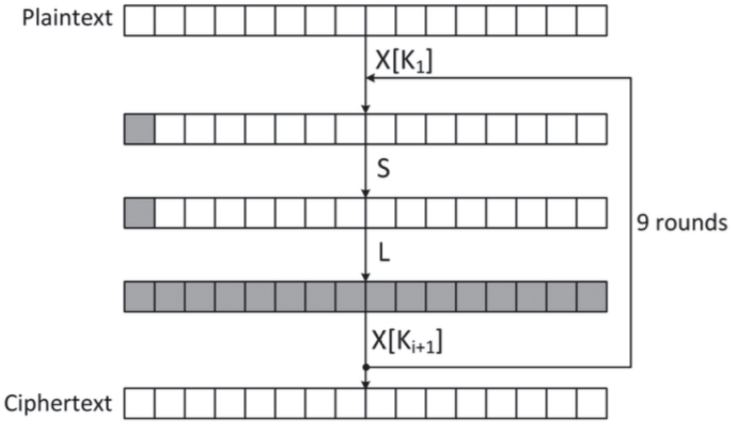


Figure 2: Encryption procedure of Kuznyechik [2]

There is an XOR operation with the round key K_1 in the beginning of the encryption. Here is the description of encryption operation where C is the ciphertext and P is the plaintext:

$$C = (X[K_{10}] \circ L \circ S) \circ \cdots \circ (X[K_2] \circ L \circ S) \circ X[K_1](P) \tag{2}$$

The first two sub-round keys, K_1 and K_2, are derived directly from the master key where K_1 is the left half of the master key and K_2 is the right half of the master key, i.e., $K = K_1 \parallel K_2$ where K is the 256-bits master key and $\parallel$ denotes the concatenation operation. The other round keys are derived according to the following transformation:

$$(K_{2i+1}, K_{2i+2}) = F[C_{8(i-1)+8}] \circ \cdots \circ F[C_{8(i-1)+1}](K_{2i-1}, K_{2i}), i = 1, 2, 3, 4, \quad (3)$$

where $C_i = L(i),\ i = 1, 2, \cdots, 32$ and $F[C](a, b)$ denotes $(LSX[C](a) \oplus b, a)$. Figure 3 shows the key schedule of Kuznyechik which is basically a Feistel Network where the round constants (C_i) are used instead of the round keys. The following notation will be used throughout the rest of the chapter to explain the fault attack we performed on Kuznyechik:

- x_i, y_i, z_i: The 16-byte state in the correct computation after the X, S, L operations, respectively, at round i.

- x'_i, y'_i, z'_i: The 16-byte state in the faulty computation after the X, S, L operation, respectively, at round i.

- $x_i[j]$: The j^{th} byte of the state x_i, where $j = 0, 1, \cdots, 15$, and the bytes are indexed from left to right.

- (C, C'): A pair of ciphertexts where C denotes the original ciphertext and C' denotes the faulty ciphertext.

- $P[j]$: The j^{th} byte of the plaintext.

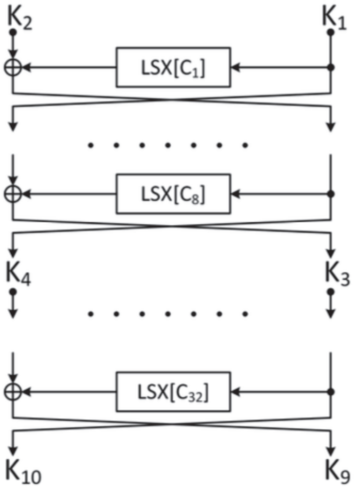


Figure 3: Key schedule of Kuznyechik [2]

3.2 Differential Fault Analysis on Kuznyechik

In this attack, we adopt the random byte fault model where the attacker is able to fault a random byte in the output of x_8 or y_8. The fault insertion is shown in Figure 4.

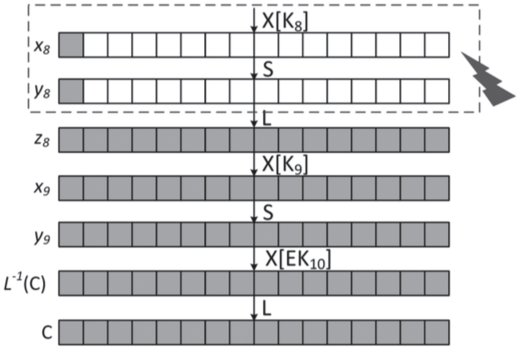


Figure 4: Fault injection in round 8 of Kuznyechik [2]

In order to launch the attack, the attacker builds a table of all possible differences in z_8 as the first step. This table contains 16×255 entries since there are 16 bytes in the state and 255 possible differences for a given byte. After that, using the observed pair (C, C'), the attacker guesses set of bytes for the last round key and evaluates the state difference between the correct and the faulty computation as $(x_9 \oplus x'_9)$. Then, the attacker checks the evaluated difference with the set of differences stored in the table. If there is a match, it means that set of bytes guessed by the attacker are candidate bytes. In our experiment, we used an array of 16 linked lists for candidate bytes where each linked list contains candidate bytes for each byte position. According to [62], the expected number of remaining candidate keys after analyzing N candidate pairs of (C, C') is given by $256^n(n \times 255^{1-n})^N$, where n is number of bytes in the state. So, for the case of Kuzneychik, the number of candidate pairs after applying N faults is $256^{16}(16 \times 255^{-15})^N$. From this equation, we are able to claim that two pairs are required to guess the last round key uniquely. However, this implementation of the attack requires guessing the 128-bits of the last round key when testing the first correct and faulty ciphertext pair. This fact makes this attack not practical. If the last round did not contain linear transformation layer (L), the attacker would be able to guess key bytes independently. This reduces the complexity as in AES and Khazad [62]. In Kuznyechik, the linear transformation operation is not omitted from the last round. This fact makes attacking Kuznyechik a bit trickier. In our attack, we used an equivalent representation of the last round by replacing the order of the key addition operation and the linear transformation operation. Since they are both linear, their order can be replaced. Figure 4 shows our attack and the equivalent representation where the order of linear transformation and key mixing have changed. We also use an equivalent key in this equivalent representation which is $EK_{10} = L^{-1}(K_{10})$. This equivalent representation makes the fault attack practical. Here is the algorithmic description of the attack:

Algorithm 1 Differential fault analysis on Kuzneychik

```
Store all possible differences at the output of $z_8$ which result from a change in one byte value into a table $T$. This table contains $16 \times 255$ entries.
Initialize $Y$ as an array of linked lists which contains candidates for each byte index. We refer to candidates for byte $i$ as $Y[i]$, $i = 0 \cdots 15$
Consider two correct and faulty ciphertext pairs $(C_1, C'_1)$, and $(C_2, C'_2)$. For each pair, compute $EC_i = L^{-1}(C_i)$, and $EC'_i = L^{-1}(C'_i)$, for $i = 1, 2$.
for all $2^{16}$ values of $EK_{10}[0]||EK_{10}[1]$ do
    Calculate the following two differences $\Delta_0$ and $\Delta_1$:
    $\Delta_0 = S^{-1}(X[EK_{10}[0]||EK_{10}[1]](EC_1[0]||EC_1[1])) \oplus S^{-1}(X[EK_{10}[0]||EK_{10}[1]](EC'_1[0]||EC'_1[1]))$
    $\Delta_1 = S^{-1}(X[EK_{10}[0]||EK_{10}[1]](EC_2[0]||EC_2[1])) \oplus S^{-1}(X[EK_{10}[0]||EK_{10}[1]](EC'_2[0]||EC'_2[1]))$
    Match these two differences with two leftmost bytes of differences in set $T$.
    if There is a match then
        Add candidate $EK_{10}[0]$ to $Y[0]$
        Add candidate $EK_{10}[1]$ to $Y[1]$
    end if
end for
for all Elements in $Y[1]$ do
    Remove $EK_{10}[1]$ from $Y[1]$ and extend it by one byte $EK_{10}[2]$.
    for all $2^8$ values of $EK_{10}[2]$ do
        Compute the following two differences $\Delta_1$ and $\Delta_2$:
        $\Delta_1 = S^{-1}(X[EK_{10}[1]||EK_{10}[2]](EC_1[1]||EC_1[2])) \oplus S^{-1}(X[EK_{10}[1]||EK_{10}[2]](EC'_1[1]||EC'_1[2]))$
        $\Delta_2 = S^{-1}(X[EK_{10}[1]||EK_{10}[2]](EC_2[1]||EC_2[2])) \oplus S^{-1}(X[EK_{10}[1]||EK_{10}[2]](EC'_2[1]||EC'_2[2]))$
        Match these two differences with bytes at indices 1 and 2 in set $T$.
        if There is a match then
            Add candidate $EK_{10}[1]$ to $Y[1]$
            Add candidate $EK_{10}[2]$ to $Y[2]$
        end if
```

```
25:     **end for**
26: **end for**
27: Repeat the above procedure starting in line 14 two bytes at a time until bytes 14 and 15.
28: $Y$ contains candidates for each byte index for $EK_{10}$. Find candidates for $K_{10}$ using candidates for $EK_{10}$ by calculating $L(EK_{10}) = K_{10}$.
```

We have simulated this attack for the recovery of the last round key (K_{10}) using 100 randomly generated user keys. Using two faults resulted in an average of 462.86 candidates for the last round key K_{10}. After that, these remaining candidates were exhaustively tested using the same pairs for correct and faulty ciphertexts, in order to uniquely recover K_{10}. After finding K_{10}, one can peel off the last round and apply the same attack to recover K_9 by inserting two faults in either x_7 or y_7. Finally, all the round keys can be recovered using K_9 and K_{10}, which allows us to find the 256-bit key $(K_1||K_2)$.

3.3 Conclusion

In this chapter, we have presented differential fault analysis attack on the Russian encryption standard, Kuznyechik. Our attack starts by employing an equivalent representation of the last round by changing the order of the two linear operations which are key mixing and linear transformation. This is done to bypass the effect of the optimal diffusion layer in order to make the attack practical. Using this attack, we are able to recover the master key using about 4 faults where two faults are used for recovering K_{10} and two other faults are used for recovering K_9.

Chapter 4

Attacks on Kalyna

Ukraine chose Kalyna [61], which was the winner of the Ukrainian National Public Cryptographic Competition (2007-2010), as their standard block cipher. After Kalyna was accepted, its modified version was made the new standard cipher in Ukraine. In this chapter, we analyze the resistance of Kalyna against fault analysis attacks.

4.1 Specification of Kalyna

4.1.1 Encryption

The Kalyna block cipher is defined using two parameters which are the input plaintext block length and the key length. In this chapter, we use Kalyna$l$-k to denote Kalyna with a plaintext block of length l and a user key of size k bits. According to the standard [61], Kalyna has 5 possible combinations of l and k, which are Kalyna128-128, Kalyna128-256, Kalyna256-256, Kalyna256-512, and Kalyna512-512. Similar to AES, the Kalyna encryption process can be described using a state matrix. The number of rows in the state matrix is always 8 since Kalyna was designed to be compatible with 64-bit architectures. The block length l determines the number of columns c in the state matrix and the key length determines the number of rounds(r) as shown in Table 2.

Kalyna l-k	No. of Columns(c)	No. of Rounds(r)
Kalyna128-128	2	10
Kalyna128-256	2	14
Kalyna256-256	4	14
Kalyna256-512	4	18
Kalyna512-512	8	18

Table 2: Number of columns(c) and number of rounds(r) based on l, r in Kalyna

The encryption function consists of the following operations which operate on the l-bit state (see Figure 5):

- η^{K_i} : Modulo 2^{64} addition of the round key K_i to the state in little-endian.

- π: Nonlinear bijective mapping of state bytes. There are 4 different SBoxes which are S_0, S_1, S_2, S_3. If the current state is presented as ($8 \times c$) matrix M, each element of M, i.e., $M_{i,j}$ where $i = 0, 1, 2, \cdots, 7$ and $j = 0, 1, 2, \cdots (c - 1)$, is mapped to a byte value using the substitution box $S_{i \ mod \ 4}$.

- τ: In this operation, each element of M is circularly right shifted on row i by γ_i bytes where γ_i is calculated according to the formula $\gamma_i = \lfloor \frac{i \times l}{512} \rfloor$. As an example, if Kalyna128-128 is used, row number 4 is circularly right shifted by $\lfloor \frac{4 \times 128}{512} \rfloor = 1$ byte.

- ψ: Refers to liner transformation of each column over the finite field where it operates on each column. The finite field GF (2^8) is formed using the irreducible polynomial

$\gamma_x = x^8 + x^4 + x^3 + x^2 + 1$. If M_j refers to column j of current state, then each element of he new state is calculated as:

$$M^{'}{}_{i,j} = (\upsilon \ggg i) \otimes M_j$$

where $\upsilon = (0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04)$ and $\otimes$ refers to scalar product of two vectors over the finite field. $\upsilon \ggg i$ refers to circular right shift of vector υ with i positions to right.

- μ^{K_i} : Modulo 2 addition ($\oplus$) of the round key K_i to the state.

The inverses of the above functions are denoted by η^{-1}, π^{-1}, τ^{-1}, and ψ^{-1}, respectively. The encryption function is defined as:

$$E_K(M) = (\eta^{K_r} \circ \psi \circ \tau \circ \pi \circ (\prod_{i=1}^{r-1} (\mu^{K_i} \circ \psi \circ \tau \circ \pi)) \circ \eta^{K_0})(M)$$

Figure 5 shows how encryption function works for Kalyna.

4.1.2 Key Scheduling

Firstly, a temporary key (K_T) is calculated from the user key (K) according to the following formula:

$$K_T = (\psi \circ \tau \circ \pi \circ \eta^{K_\alpha} \circ \psi \circ \tau \circ \pi \circ \mu^{K_\varpi} \circ \psi \circ \tau \circ \pi \circ \eta^{K_\alpha})(K_\rho)$$

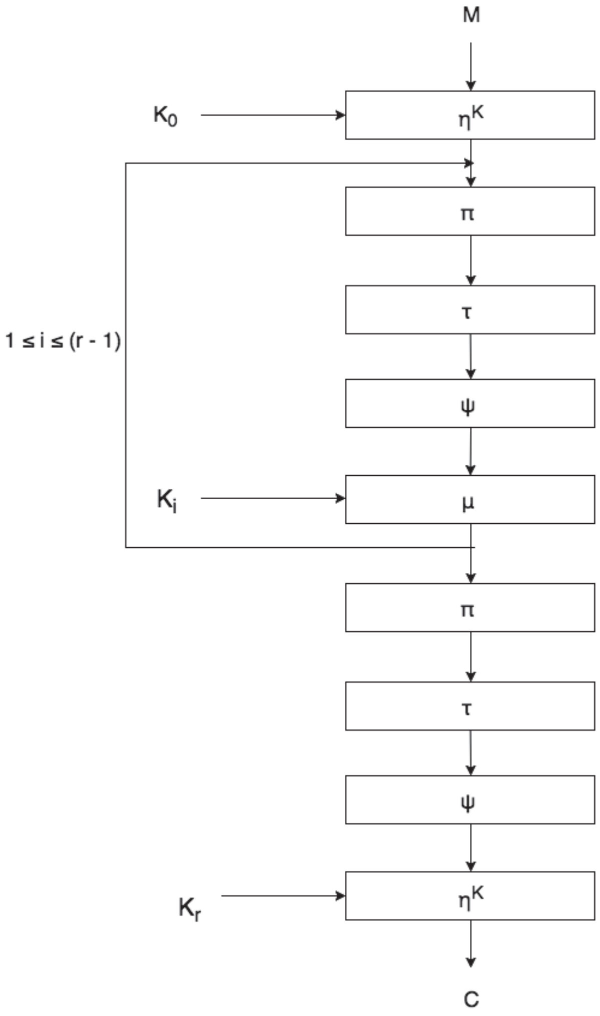


Figure 5: Kalyna encryption function

where K_ρ is l bit value which is calculated as $\frac{(l+k+64)}{64}$ in little-endian. The functions $\psi, \tau, \pi, \eta, \mu$ are the same functions as the ones used in the encryption operation.

When the key size is the same as the block size, i.e. when $k = l$, we have:

$$K_\alpha = K_\varpi = K$$

When key size is double the block size, i.e., $k = 2 \times l$, we have K_α is the left half of K

and K_ϖ is the right half of K. In other words:

$$K = K_\alpha \mid\mid K_\varpi$$

After finding K_T, the round keys with even indices ($i \ \epsilon \ 0, 2, 4, 6, \cdots$) are calculated according to the following formula:

$$\Xi(K_T, i) = (\eta^{\varphi_i^{K_T}} \ \circ \ \psi \ \circ \ \tau \ \circ \ \pi \ \circ \ \mu^{\varphi_i^{K_T}} \ \circ \ \psi \ \circ \ \tau \ \circ \ \pi \ \circ \ \eta^{\varphi_i^{K_T}})(K_\gamma)$$

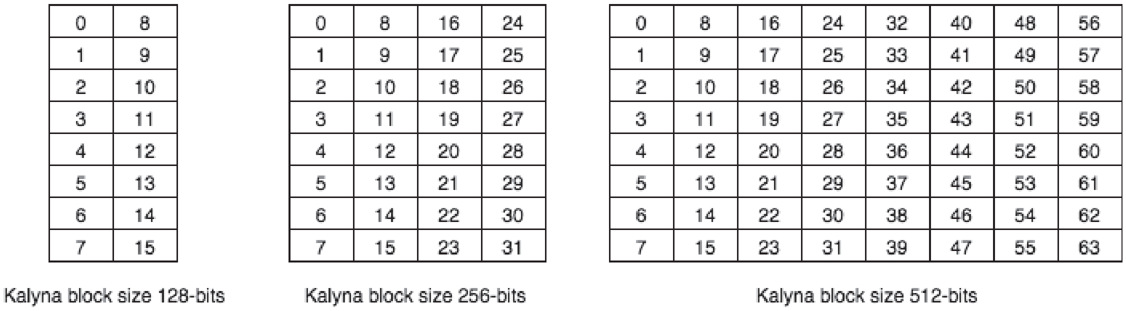
where $\varphi_i^{K_T} = \eta^{K_T}(\vartheta \ll (i/2))$ which is K_T added modulo 2^{64} with the constant ϑ shifted left by index i divided by 2. The constant ϑ is a sequence of bits of length l which is $01((0001)^{7\times(c-1)})00$. Here the notation $((0001)^{7\times(c-1)})$ means that the bit sequence 0001 is repeated $7 \times (c-1)$ times. As an example, for Kalyna128-128, we have $\vartheta = 0100010001000100010001000100100$. K_γ is calculated according to the user input key K. If the block size and key size are the same, then $K_\gamma = K \ggg (32 \times i)$. If the key size is double of the block size, then $K_\gamma = K \ggg (16 \times i)$ for even indices divisible by 4 $i\epsilon\{0, 4, 8, 12, ...\}$. For indices not divisible by 4, $K_\gamma = K \ggg (64 \times \lfloor(i/4)\rfloor)$.

After generating the round keys with even indices, the round keys with odd indices are generated from their predecessor round key as $K_i = K_{i-1} \lll ((l/4) + 24)$ where $i\epsilon\{1, 3, 5, 7, \cdots\}$ and l is the size of internal cipher state in bits. Thus, the round keys with even indices are linearly dependent on their successor only and the round keys with odd indices are linearly dependent on their predecessor only. For example, in Kalyna128-128, there are 11 round keys which are $K_0, K_1, \cdots, K_{10}$. K_0 and K_1 are linearly dependent on each other but K_1 and K_2 have no linear dependency. So, an attacker has to recover all the round keys with even indices or all the round keys with odd indices in order to break the cipher.

Notation : The following notation is used throughout this chapter:

- w_i: State after addition of round key K_i, this key addition can be either η or μ.
- x_i: State after the π operation at round i.
- y_i: State after the τ operation at round i.
- z_i: State after the ψ operation at round i.
- For states in faulty computation, we use w_i', x_i', y_i', z_i'
- $w_i[j][k]$: If the state is represented as a matrix with 8 rows and c columns, $w_i[j][k]$ refers to row number j and column number k of state w_i. Similar notation is used for other states.
- $W_i[j]$: If the state is considered as a sequence of bytes rather than a matrix, we use the upper case notation. $W_i[j]$ refers to byte index j where $j\epsilon\{0, 1, 2, ..., (\lfloor(l/8)\rfloor - 1)\}$ and bytes are numbered from left to right. Also, in order to convert a byte index to a matrix element, $\lfloor(\frac{j}{8})\rfloor$ gives us column number and $\lfloor\ j\ mod\ 8\ \rfloor$ gives us the row number, so $W_i[j] = w_i[j\ mod\ 8][\lfloor\frac{j}{8}\rfloor]$. Figure 6 shows how the state bytes are numbered for each block size l if the state is considered as a sequence of bytes rather than a matrix.
- W_L : Refers to the left half of the state W.

- W_R : Refers to the right half of the state W.
- Δ_{x_i,x_j} : Difference between two states x_i and x_j.
- (C, C') : A pair of ciphertexts where C is the correct ciphertext and C' is the faulty ciphertext.
- $P[j]$: Byte j of plaintext where bytes are numbered as shown in Figure 6.



0	8
1	9
2	10
3	11
4	12
5	13
6	14
7	15

Kalyna block size 128-bits

0	8	16	24
1	9	17	25
2	10	18	26
3	11	19	27
4	12	20	28
5	13	21	29
6	14	22	30
7	15	23	31

Kalyna block size 256-bits

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

Kalyna block size 512-bits

Figure 6: Kalyna state bytes with their indices

4.2 Fault Analysis on Kalyna with Known SBoxes

In this attack, the attacker is able to change a byte value in the state to a random value but the attacker does not know which byte has been changed. Attacker only knows where that change happens. In other words, the random fault model is applied here. Propagation of the injected fault depends on the operations applied to the state after the fault. Figure 7 shows how a fault applied at w_9 propagates in the last round for Kalyna128-128. As shown in the figure, if a fault appears at the input to π or τ operations, they affect only one byte of the next state. However, if a fault appears at the input of the ψ operation, it affects the whole column. During our attacks, a fault can happen in the input of π, τ or ψ, which will have the same effect in terms of propagation of the fault.

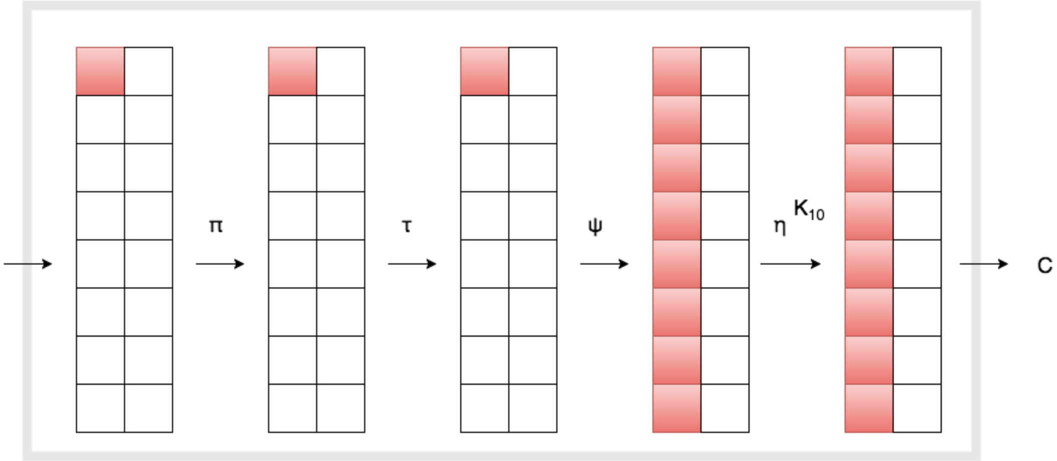


Figure 7: Fault propogation if a fault appears in input to π in the last round in Kalyna128-128

The first step in this attack is to build a lookup table with the possible set of differences after the ψ operation. This table, λ, has 8×255 entries since there are 8 possible places in one column where a change can happen and the inserted fault takes any value from $0x01$ to $0xFF$.

The second step is to recover the last round key. In order to do that, the attacker faults the computation at the input to the last π operation at round 10 (e.g., w_9 in Kalyna128-128). Based on the byte index where the fault occurs, the faulty ciphertext differs from the correct ciphertext only in one column since the fault changes only one column. Then, the attacker recovers the bytes of the key in that column by analyzing (C, C') pairs. In order to recover the last round key, the fault is applied to the last round. In order to recover the last round key fully, the attacker has to recover each column of the key. However, no matter how many faults are applied, there will be two candidates for the most significant byte in each column which differ only in one bit since the modulo 2^{64} addition may or may not generate a carry [52]. In other words, in our attack, we are not able to recover the most significant bit of the most significant byte of the key. Hence, we get two candidates for the most significant byte.

After recovering the key candidates for the last round, the attacker also needs to recover the intermediate round keys in order to break the cipher. In order to do so, all subsequent rounds have to be peeled. Also, there is a difference between recovery of the last round key and recovery of the intermediate round keys (K_i) for each round i where $i \, \epsilon \, \{2, 3, 4, 5, 6, 7, 8, \cdots (r-1)\}$. In order to recover an intermediate round key, the attacker has to fault π input to the round before that round. This means that, if the attacker wants to recover the round key at round i, the fault has to be applied at the SBox input at round $i-1$, which refers to state $w_{(i-2)}$. This is because all the intermediate key additions are Modulo 2 addition (XOR) and XOR function is also used to calculate the differences. The intermediate round keys are recovered column by column similar to the last round key. The only difference is that, for the intermediate round keys, we should end up with only one candidate if we applied enough number of faults since there is no carry propagation as in the last round key. According to the formula applied to Kuznyechik, in order to estimate the number of faults needed to recover the last round key [62], we are able to retrieve each column of the key by applying two faults.

Unlike AES [27] and Kuznyechik [30], we recover the round keys column by column rather than recovering the whole round key and we merge these candidates for each column. We are not able to recover the whole round key in one iteration since we have Modulo

2^{64} addition for the last round key which affects one column in the ciphertext and for intermediate round keys, the difference after the first ψ operation after fault is one column as well. As an example, in Kalyna128-128, if the fault occurs in one of byte with index $\in \{0, 1, 2, 3, 12, 13, 14, 15\}$, the fault propagates to the first column and the attacker is able to retrieve information about the first column of the key. If the fault occurs in one of the byte with indices in $\{8, 9, 10, 11, 4, 5, 6, 7\}$, the fault propagates to the second column and the attacker is able to retrieve information about the second column of the key. In our attack, we use the same approach we used for Kuznyechik [2], however instead of guessing two bytes at a time, we merged one byte to our previous guess at each step and we ended up guessing the whole column of the key we are trying to recover. This was an improvement to the previous attack.

The following algorithm (algorithm 2) gives step by step description of the recovery of column j of the round key for round i. For simplicity, we assume that all faults propagate to column j and we denote column j of the correct ciphertext as C_j and column j of the faulty ciphertext as C'_j. $M[i, j]$ refers to slice of state M starting from byte index i and ending at byte index j inclusive as shown in Figure 6.

Algorithm 2 Fault attack on Kalyna using known SBoxes. This algorithm recovers one column of the round key

```
$i$ : Round key index to recover. $N$ : Number of Faults to Apply $j$: Round key column to recover.
$\gamma$: Set of candidates for round key $K_i$. In order to refer to partial guess, we use the same notation as we use for slices which is $K_i[m, n]$ where $m$ is the beginning of the slice and $n$ is the end of the slice.
Compute $(255 \times 8)$ differences after the $\psi$ operation and store them in $\lambda$
if $i$ == Total Number of Rounds in Kalyna then
```

```
    Fault Round $i$ $N$ times in a way that the fault propagates to column $j$ at $\Delta_{C,C'}$ which means that the state $\Delta_{C,C'}$ is all zeros except column $j$
else
    Fault Round $(i - 1)$ $N$ times in a way that the fault propagates to column $j$ at $\Delta_{w_{i-1},w'_{i-1}}$ which means that the state $\Delta_{w_{i-1},w'_{i-1}}$ is all zeros except column j
end if
Consider N pairs of correct and faulty ciphertexts $(C, C')_n$ where $n \, \epsilon \, \{1, 2, \cdots, N\}$
// $Start\_Byte$ is byte index where we are starting our guess.
// $End\_Byte$ is byte index where we end our guess at this iteration.
// Bytes are numbered from 0 to 7 in a column where 0 is the least
// significant byte and 7 is the most significant byte
Start_Byte ← 0
End_Byte ← 1
// For intermediate rounds, peel off all subsequent
// rounds back to the state after key addition at round $i$
// Initialize pair of differences $(D, D')_n$ to $(C, C')_n$
if i != Total number of rounds in Kalyna then
    Peel off all subsequent rounds until round $i$ and analyze the states at the output of the key addition at round $i$
    $(D, D')_n = (x_i, x'_i)_n$
end if
Consider the slice of key guess $K_i[Start\_Byte, End\_Byte]$. We will append one more byte guess at each step to this slice
// $K'_i$ is the current key guess we are going to evaluate. We initialize it to 0 in the beginning
$K'_i \leftarrow 0$
while $End\_Byte! = 7$ do
    // Now we are going to generate $K'_i$ and check if it is a valid guesses
    if $End\_Byte == 1$ then
        $K'_i[Start\_Byte]$ can take any value from $0x00$ to $0xFF$
    else
        Get all bytes from previous guesses $K'_i[Start\_Byte, End\_Byte - 1] = K_i[Start\_Byte, End\_Byte - 1]$
    end if
end while
$K'_i[End\_Byte]$ can take any values from $0x00$ to $0xFF$
Now, with the guessed $K'_i$, compute the following differences:
if $i == TotalNumberofRounds$ then
    Consider the differences for every pair $(\eta^{K'_i})^{-1}(D) \oplus (\eta^{K'_i})^{-1}(D')$
else
    // For intermediate rounds, we look at the difference after $x_i$
    Consider the differences for every pair $\pi^{-1}(\tau^{-1}((\psi^{-1}(D \oplus K_i')))) \oplus \pi^{-1}(\tau^{-1}((\psi^{-1}(D' \oplus K_i'))))$
end if
if all differences analyzed are contained in $\Lambda[Start\_Byte][End\_Byte]$ then
    Add $K_i'[Start\_Byte][End\_Byte]$ into $K_i[Start\_Byte][End\_Byte]$
end if
// increment End_Byte

$End\_Byte \leftarrow (End\_Byte + 1)$
```

Algorithm 2 does not work for the first two round keys. This is because modulo 2^{64} addition of K_0 is the first operation applied to plaintext and there is no round before K_0. If we can recover K_1, we can get K_0 from K_1. However, since K_1 is an intermediate round key, we need to fault the computation at SBox input to round 0 but there is no SBox at round 0 (round 0 only contains modulo 2^{64} addition of round key K_0). In order to recover the first two round keys, we apply the following steps. First, we flip the most significant bit in P (call it $P^{'}$) and calculate $C^{'}$. Then, analyze the difference at state x_1 for C and $C^{'}$. We call this difference $\Delta_{x_1,x_1'}$. Using this difference, we recover the most significant byte of K_0. Then, we repeat the attack for the remaining bytes of the plaintext, until all the bytes of K_0 are recovered. Finally, we recover K_1 from K_0.

According to our experiment results, using 10 faults, we end up with 2 candidates for each column of the last round key. We experimented using Kalyna128-128, where we ended up having 4 candidates (2 candidates for each column) for K_{10}. This means that 20 faults (10 faults for each column) gives us 4 candidates for the last round key. If the block size is 256, we end up with 16 (2 for each column and 4 columns) candidates with 40 faults and if the block size is 512, we end up with 256 (8 columns) candidates with 80 faults.

Number of Faults	Number of candidates
10	2.06
9	2.1
8	2.6
7	3.06
6	5.44
5	24.4

Table 3: Number of candidates for one column of the last round key in Kalyna128-128

Number of Faults	Number of candidates
1	14025359.36
2	1.18
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Table 4: Number of candidates for one column of intermediate round keys between round 8 and round 2 inclusive in Kalyna128-128

Table 3 shows the number of candidates we retrieved based on the number of faults for one column of the last round key in Kalyna128-128.

For the intermediate round keys, recovery was easier since we ended up with only one candidate if we apply enough faults and our experimental results were close to our estimation. Table 4 shows our experimental results for the recovery of one column of an intermediate round key.

Based on the experiment results we performed on Kalyna128-128 using 100 random user-defined keys and 100 random plaintexts, it can be concluded that about 3 faults can uniquely determine half of the key for each intermediate round key. However, 1 fault only eliminates a small fraction of candidates ($log_2(14025359.36) = 23.74$). 2 faults was the expected number of faults required to determine one column of an intermediate round key. However, according to our experiment results, 2 faults do not always uniquely determine one column of the key in all cases but 3 faults uniquely determine one column of the key in all cases. This means that in Kalyna128-128, we need 6 faults to retrieve round keys K_8, K_6, K_4, K_2. Note that round keys with odd indices can be retrieved using round keys with even indices. So, no faults need to be applied for recovering round keys with even indices. 10 faults are needed to recover each column of the last round key K_{10}. This means that we need about $20 + 6 + 6 + 6 + 6 = 44$ faults to retrieve round keys $10, 9, 8, 7, 6, 5, 4, 3, 2$. In that case, round key K_{10} will have 4 candidates but the other round keys will be determined uniquely. So, 44 faults give us 4 possible set of keys. For Kalyna128-256, the number of faults for each intermediate round is again 6 since it also has two columns in the internal state and there are 6 intermediate round keys to recover, which are $K_{12}, K_{10}, K_8, K_6, K_4, K_2$. In order to recover the last round key, number of faults is 20 since it has 2 columns. So, using our attack, $20 + (6 \times 6) = 56$ faults gives us 4 candidates for set of round keys except K_0 and K_1 in Kalyna128-256. Similarly, for Kalyna256-256, the number of intermediate round keys to recover is 6 since it also has 14 rounds like Kalyna128-256. However, the number of faults required to recover each intermediate round key is 12 since there are 4 columns. The number of faults required to recover the last round key is 40 since the internal state has 4 columns. So, $40 + (12 \times 6) = 112$ faults gives us 16 candidates for set of intermediate round keys except K_0 and K_1 in Kalyna256-256.

Mode	Number of Faults	Number of Key Candidates
Kalyna128-128	44	4
Kalyna128-256	56	4
Kalyna256-256	112	16
Kalyna256-512	136	16
Kalyna512-512	272	256

Table 5: Number of faults required to recover the set of round key candidates except K_0 and K_1 in Kalyna

Kalyna256-512 has the same block size as Kalyna256-256, hence, again we need to recover 4 columns of each round key. However, the number of intermediate round keys to recover is 8 since it has 18 rounds and we also need to recover K_{14}, K_{16}. So, $40+(12\times 8) = 136$ faults gives us 16 candidates for the set of round keys. For Kalyna512-512, the number of rounds is 18 so we have 8 intermediate round keys to recover. However, the number of faults for each round key is 24 since there are 8 columns to recover. For the last round key, 80 faults give us set 256 candidates for the last round key. So, applying $80 + (24 \times 8) = 272$ faults gives us 256 candidates for the set of intermediate round keys in Kalyna512-512. Table 5 summarizes the expected number of faults to recover the round keys, except the first two, for different choices of l, k.

4.3 Fault Analysis on Kalyna with Unknown SBoxes

In this section, we consider Kalyna deployed with secret SBoxes. There are 4 different SBoxes and we will refer to them as S_0, S_1, S_2, S_3. We also assume that the same SBoxes are used in the key scheduling. Deploying a cipher with secret SBoxes is not a new approach. A similar approach was used in the Russian standard GOST 28147-89 [38]. So, it is possible that Kalyna may also be deployed with user defined SBoxes in some applications. Even though Kerchoff's principle states that security should depend only on the secret key, secret SBoxes are used in some cryptographic devices and they are assumed to increase the security. As an example, secret SBoxes are used in military products, gaming systems, and Pay TV [5].

We applied ineffective fault analysis on Kalyna using stuck-at-0 fault model. In this case, the attacker applies the fault by forcing a specific SBox output to 0 and the attacker retrieves information about the secret parameters (secret round key, secret SBoxes) if an ineffective fault occurs, that is if the correct and the faulty ciphertexts are the same ($C = C'$).

In our attack, we recovered K_0 with 2^{32} candidates and we also recovered the secret SBox entries for all SBoxes with 2^8 candidate sets (1 set of candidates for each value of $S^{-1}[0]$ for each SBox). Note that the attacker is able to retrieve K_1 directly from K_0. This attack starts by iterating each byte of the plaintext until an ineffective fault occurs for that byte. Our attack is similar to the ineffective fault analysis applied to Kuznyechik [30] except that there are two differences. First, in here, we have 4 different SBoxes. Second, we have to deal with the carry propagation when recovering K_0. The SBox entries that output 0 are denoted as $S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0], S_3^{-1}[0]$.

Here is the algorithm to recover K_0 based on $S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0], S_3^{-1}[0]$:

Algorithm 3 Recovering the first round key of Kalyna based on 4 SBox inputs

```
$K_0$ : set of candidates for $K_0$
$P$ : Plaintext and P[i] refers to byte index i of plaintext $\xi$: Ineffective bytes and $\xi[i]$ refers to byte index i of ineffective byte $K_0'$: Current guess for $K_0$ $Carry$ : Set of places in modulo $2^{64}$ addition where carry occurs
for $i \leftarrow 0$ to $number\ of\ bytes\ in\ the\ state$ do
    Iteratively exhaust P[i] and fault byte $x_1[i]$ (SBox output at round 1 and at index i) until an ineffective fault occurs.
    if Ineffective fault occurs then
        $\xi[i] \leftarrow P[i]$
    end if
end for
for  Each guess for $S_0^{-1}[0]$ || $S_1^{-1}[0]$ || $S_2^{-1}[0]$ || $S_3^{-1}[0]$ do
    Fill carry[i] for each index by finding places where the carry occurs based on guesses for $S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0], S_3^{-1}[0]$ and values of $\xi$.
    Set $K_0'$ to all zeros
    for $i \leftarrow 0$ to $number\ of\ bytes\ in\ the\ state$ do
        if  i mod 4 == 0 then
            $K_0'[i] = S_0^{-1}[0] - \xi[i] - Carry[i]$
        else if  i mod 4 == 1 then
            $K_0'[i] = S_1^{-1}[0] - \xi[i] - Carry[i]$
        else if  i mod 4 == 2 then
            $K_0'[i] = S_2^{-1}[0] - \xi[i] - Carry[i]$
        else if  i mod 4 == 3 then
            $K_0'[i] = S_3^{-1}[0] - \xi[i] - Carry[i]$
        else
            This else should never be reached
        end if
        // Add current key guess to set of key candidates.
        Add $K_0'$ to $K_0$
    end for
end for
```

In order to find locations where the carry is propagated from previous index, we use the following equation:

$$carry[i] = \begin{cases} 0, & \text{if } i \,\%\, 8 \;=\; 0 \\ 1 & \text{if } (i \,\%\, 4 \;=\; 0)\; AND\; (i \,\%\, 8 \;\neq\; 0)\; AND\; S_0^{-1}[0] < \xi[i-1] \\ 0 & \text{if } (i \,\%\, 4 \;=\; 0)\; AND\; (i \,\%\, 8 \;\neq\; 0)\; AND\; S_0^{-1}[0] \geq \xi[i-1] \\ 1 & \text{if } (i \,\%\, 4 \;=\; 1)\; AND\; S_1^{-1}[0] < \xi[i-1] \\ 0 & \text{if } (i \,\%\, 4 \;=\; 1)\; AND\; S_1^{-1}[0] \geq \xi[i-1] \\ 1 & \text{if } (i \,\%\, 4 \;=\; 2)\; AND\; S_2^{-1}[0] < \xi[i-1] \\ 0 & \text{if } (i \,\%\, 4 \;=\; 2)\; AND\; S_2^{-1}[0] \geq \xi[i-1] \\ 1 & \text{if } (i \,\%\, 4 \;=\; 3)\; AND\; S_3^{-1}[0] < \xi[i-1] \\ 0 & \text{if } (i \,\%\, 4 \;=\; 3)\; AND\; S_4^{-1}[0] \geq \xi[i-1] \end{cases}$$

From the ineffective fault at byte index i, we are able to write the following equation according to encryption function and we used notation $S^{-1}_{(i\ \%\ 4)}[0]$ to refer to SBox used for index i:

$$\xi[i] + K_0[i] + Carry[i] = S^{-1}_{(i\ \%\ 4)}[0] \qquad (4)$$

If our guess for the SBox entry $S^{-1}_{(i\ \%\ 4)}[0]$ is smaller than $\xi[i]$, then equation 4 generated a carry for the subsequent index $(i+1)$. For the indices which are located in the beginning of each column where $(i \; mod \; 8 \; == 0)$, there is no carry from the previous index.

This attack gives us 2^{32} candidates for K_0 using $2^8 \times 2^4 = 2^{12}$ faults in the worst case for Kalyna128-128 since Kalyna128-128 contains 16 bytes and we need 2^8 faults for each byte. For Kalyna128-256, the number of faults is the same since the number of bytes in the state is the same. For Kalyna256-256, the number of faults required is $2^8 \times 2^5 = 2^{13}$ since Kalyna256-256 contains 32 state bytes. It is the same number of faults for Kalyna256-512. For Kalyna512-512, the number of faults is $2^8 \times 2^6 = 2^{14}$ since Kalyna512-512 contains 64 state bytes.

After guessing K_0 with 2^{32} candidates, we also have 2^{32} candidates for K_1 since K_1 can be generated directly from K_0. Our next step is to recover all SBox entries S_0, S_1, S_2, S_3 based on our guesses for $S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0], S_3^{-1}[0]$. We are going to recover each SBox entry separately by iterating two byte indices in the input to ψ operation while keeping all the other indices in the column as 0. One of these byte indices is going to contain an SBox entry we already recovered and another is going to be iterated by the attacker until an ineffective fault is observed at x_2 (SBox output at round 2 in a specific index). Figure 8 demonstrates how the state propagates if we give the input to Kalyna in a way that all bytes in x_1 are 0 except two indices. We call these two indices m and a. We have to choose these indices in the way such that they go to same SBox.

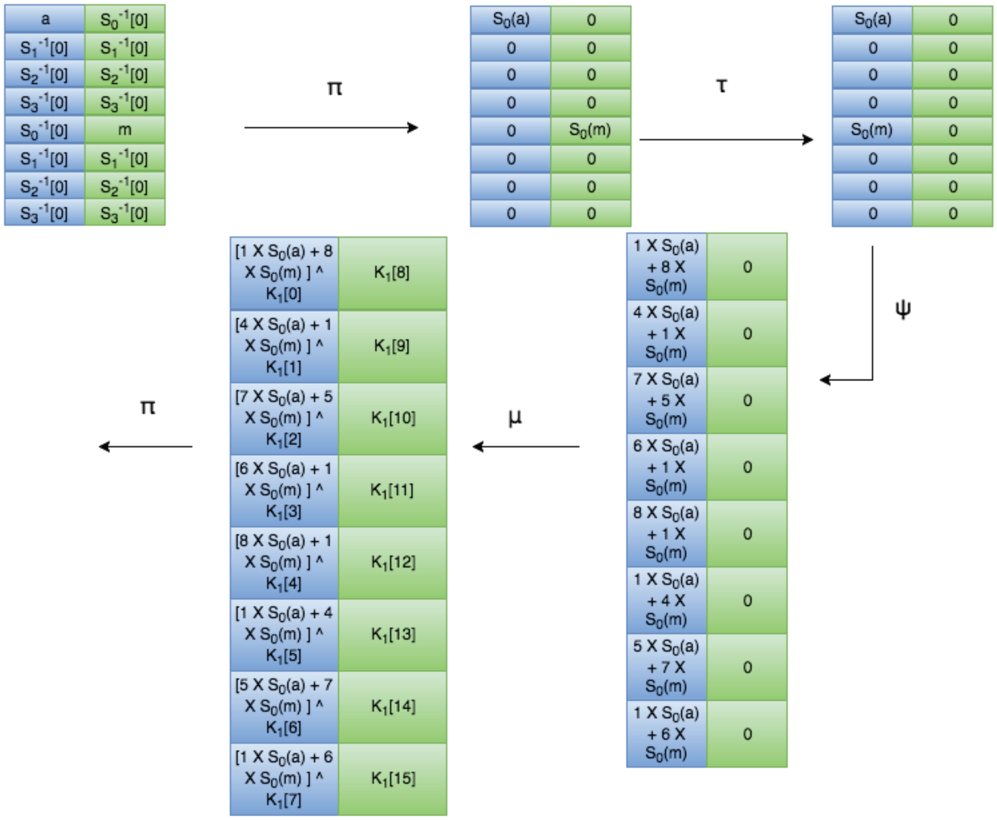


Figure 8: Calculation after forcing SBoxes at round 1 to all zeros in Kalyna

Here is the algorithm to recover SBox entries based on $S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0], S_3^{-1}[0]$. We use the MixColumns coefficients v in this recovery. The notation $S_{fault\_index \ mod \ 4}\ [a]$ means that we use SBox according to fault index, since the SBoxes are chosen according to the byte indices, and we get an entry $S(a)$ for the value a.

Algorithm 4 Recovering all SBox entries in Kalyna

```
$S_i'$ : Set of candidate entries (m, $S_i[m]$) for SBox $S_i$
$m$ : Refers to byte value iterated by the attacker. $a$: refers to byte value the attacker is using from previous recovery. $m\_index$ : index where value of m is iterated. $a\_index$ : index where value of $a$ is used. $fault\_index$: index where attacker faults SBox at round 2
```

```
for Each SBox $S_i$, $i = 1, 2, 3, 4$ do
    for Each guess for $S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0], S_3^{-1}[0]$ do
        $a \leftarrow S_i^{-1}[0]$
        for $m \leftarrow 0$ to $255$ do
            Using the guesses for $S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0], S_3^{-1}[0]$ and the guess for $K_0$, find the input plaintext which makes the state $y_1$( after round 1 shift rows operation) all zeros except for two indices, where $y_1[m\_index] = m$ and $y_1[a\_index] = a$
            Fault $x_2[fault\_index]$ by forcing the output to 0.
            if Ineffective fault occurs then
                It means $\{(v[a\_index] \otimes S_i[a]) + (v[m\_index] \otimes S_i[m])\} \oplus K_1[fault\_index] = S^{-1}_{fault\_index\ mod\ 4}[0]$
                Find $S_i[m]$ as $S_i[m] = \{(S^{-1}_{fault\_index\ mod\ 4}[0] \oplus K_1[fault\_index]) - (v[a\_index] \otimes S_i[a])\} \otimes (v[m\_index])^{-1}$
                if m is not contained in $S_i'$ then
                    Add (m , $S_i[m]$) into $S_i'$
                    Set a to m
                else
                    Find another value for a from $S_i'$.
                end if
            end if
        end for
    end for
end for
```

This algorithm iteratively recovers the SBox entries using already recovered entries until all SBox entries for an SBox are recovered. If no other entries can be added to the set of candidates, indices for m and a need to be changed. We experimented this algorithm by randomizing all 4 SBoxes and running the experiment 100 times and we were able to recover correct SBox entries each time. For the choice of indices for m and a, indices which go to same SBox have to be chosen and this SBox has to be the SBox we are trying to recover. However, the choice of $fault\_index$ does not depend on the SBox we are aiming to recover since we already have candidates for $S_i^{-1}[0]$, where $i \epsilon \{0, 1, 2, 3\}$ for each SBox. In our experiments, we used indices $0, 12$ in the first phase and faulted SBox number 0. In the second phase, a is initialized to last found m value from the first phase and SBox number 5 was faulted. This allowed us to recover all entries for S_0. The number of faults required is 256. For S_1, indices 1 and 13 were used for m and a. SBox at index 0 was faulted in the first phase of the attack and SBox 1 was faulted in the second phase. For S_2, indices 2 and 14 were used for m and a, SBox at index 1 was faulted in the first phase and SBox at index 2 was faulted in the second phase. For S_3, indices 3 and 15 were used for m and a and SBox at index 0 was faulted in the first phase and SBox at index 2 was faulted in the second phase. Since we were able to recover all SBox entries for each SBox in the two phases, the number of faults required to recover each SBox is upper bounded by $2 \times 256 \times 256$ since there are two phases, and for each of 256 entries we need to go through all possible values of m, in worst case. So, this attack requires about 2^{17} faults in the worst case.

Using the above attack, an attacker is able to retrieve 2^{32} candidate pairs for the multiset where all other entries depend on $S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0]$ and $S_3^{-1}[0]$:

$$(S_0^{-1}[0], S_1^{-1}[0], S_2^{-1}[0], S_3^{-1}[0], K_0, K_1, EntriesforS_0, EntriesforS_1,$$
$$EntriesforS_2, EntriesforS_3)$$

This list is still small enough to be brute forced since it contains 2^{32} elements.

4.4 Conclusion

In this chapter, we showed how to apply differential fault analysis on Kalyna. We also showed how Kalyna with secret SBoxes can be attacked using ineffective fault analysis. Our results show that importance of protecting the implementations of Kalyna against different forms of fault attacks.

Chapter 5

Attacks on Kupyna

Kupyna is a recently selected Ukrainian hash function standard. It was designed based on Kalyna block cipher. In this chapter, we apply differential fault analysis to Kupyna when it is used as the underlying function in different MAC schemes and show how the secret inputs can be retrieved.

5.1 Specification of Kupyna

Kupyna hash function was originally proposed in 2015 as the standard DSTU 7564:2014 [60]. It uses the Davies-Meyer compression function based on the Even-Mansour construction and it supports digest size of n bits such that:

$$n = (8 \times s)\, bits, s \,\epsilon\, \{1, 2, \cdots, 64\}$$

In this chapter, we use Kupyna$-n$ to refer to Kupyna hash function with n output bits. The parameter n is chosen by the user and it determines three parameters which are the internal state size l, the number of rounds of the compression function t, and the number of columns in the state matrix c. The internal state size is defined according to the following formula:

$$l = \begin{cases} 512\, bits & \text{if } 8 \leq \; n \; \leq 256. \\ 1024\, bits & \text{if } 256 \leq \; n \; \leq 512. \end{cases}$$

The compression function consists of two t-round ciphers which are very similar to Kalyna. The number of rounds (t) is defined according to the following formula:

$$t \; = \begin{cases} 10, & \text{if } 8 \leq \; n \leq 256. \\ 14, & \text{if } 256 \leq \; n \leq 512. \end{cases}$$

The size of the state matrix defines the number of columns c because the number of rows is always 8 since Kupyna was designed to be compatible with 64-bit platforms (8 bytes makes 64-bits), similar to Kalyna. The number of columns is determined as follows:

$$c \; = \begin{cases} 8, & \text{if } l = 512. \\ 16, & \text{if } l = 1024. \end{cases}$$

Figure 9 shows how Kupyna works. IV is a constant vector which is calculated according to the following formula:

$$\mathbf{IV} = \begin{cases} 1 \ll 510, & \text{if } l = 512. \\ 1 \ll 1023, & \text{if } l = 1024. \end{cases}$$

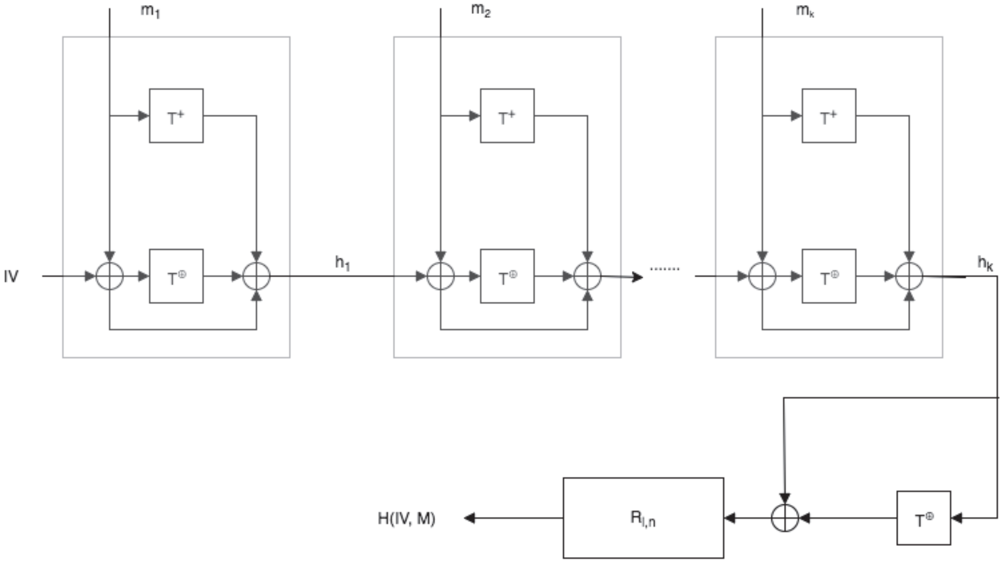


Figure 9: Kupyna block diagram

Kupyna can process messages of length up to $2^{96} - 1$ bits. After a message input with size N bits where $0 \leq N \leq (2^{96} - 1)$ is given by the user, the message (M) is padded regardless of its length. The padding follows the message and contains a single 1 bit followed by d 0 bits, where d is calculated based on the internal state size (l) such that:

$$d = (-N - 97)\, mod\, l$$

After the padding, the message M is divided into k blocks with size l bits each, which are:

$$m_1, m_2, \cdots, m_k$$

More formally, For a given message M, the hash code for Kupyna-n ($H(IV, M)$) is calculated as:

$$h_0 = IV$$

$$h_i = \tau^{\oplus}(h_{i-1} \oplus m_i) \oplus \tau^{+}(h_{i-1}) \oplus h_{i-1},\ for\ i = 1, 2, .., k$$

$$H(IV, M) = R_{l,n}(\tau^{\oplus}(h_k) \oplus h_k)$$

where $R_{l,n}$ is the truncation function which retrieves n most significant bits of input of l bits, and h_i is the hash value calculated by the compression function processing the message block m_i.

One compression function consists of two functions which are $\tau^{\oplus}$ and τ^{+}. These functions are defined as:

$$\tau^{\oplus} = \prod_{0}^{t-1}(\psi \circ \alpha \circ \sigma \circ \kappa)$$

$$\tau^{+} = \prod_{0}^{t-1}(\psi \circ \alpha \circ \sigma \circ \theta)$$

Figure 10 shows how these functions work for t rounds. As it can be seen, the only difference between them is the fact that $\tau^{\oplus}$ uses XOR addition for key mixing and τ^{+} uses modulo 2^{64} addition for key mixing.

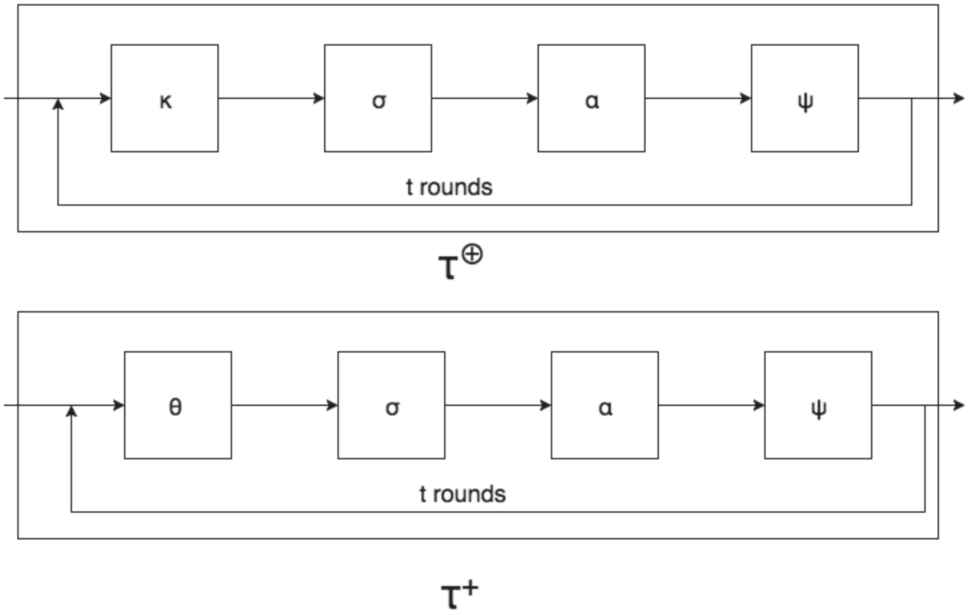


Figure 10: $\tau^{\oplus}$ and τ^{+} functions in Kupyna which work similar to Kalyna

We use the following notation throughout this chapter for referring to the state bytes: $G_{i,j}$ refers to row i and column j of the current state.

ψ is a linear transformation layer which uses the same vector υ as Kalyna:

$$\upsilon = (0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04)$$

The element at row *i* and column *j* of the new state matrix ($G^{'}_{i,j}$) is calculated as:

$$G^{'}_{i,j} = (\upsilon \ggg i) \otimes G_j,$$

where G_j is column j of the current state matrix and $\otimes$ is Galois field multiplication in the finite field GF(2^8) with the irreducible polynomial $\gamma(x) = x^8 + x^4 + x^3 + x^2 + 1$. This operation is also referred as MixColumns.

α denotes the ShiftRows operation which circularly right shifts each row. There are 8 rows in the state matrix numbered as $0, 1, \cdots, 7$, and each row i is circularly right shifted by i bytes except row number 7 which is circularly right shifted by 7 bytes if the block size(l) is 512 bits and by 11 bytes if the block size(l) is 1024 bits.

σ is a non-linear transformation of bytes, also referred as SubBytes, where each byte value is mapped to another byte value using the Kupyna SBoxes. Kupyna uses 4 different SBoxes like Kalyna (S_0, S_1, S_2, S_3). For each byte in the current state $G_{i,j}$, the SBox to apply is calculated as $S_{i\ mod\ 4}$. So, the new state byte ($G^{'}_{i,j}$) is calculated as:

$$G^{'}_{i,j} = S_{i\ mod\ 4}[G_{i,j}]$$

θ is modulo 2^{64} addition of the round constant to the current state and κ denotes the XOR addition of the round constant to the current state. The round constants are known and are calculated as follows:

For κ where the round constant($\varpi(j)$) is added to current state using XOR operation (modulo 2 addition), $\varpi(j) = ((j \ll 4) \oplus v, 0, 0, 0, 0, 0, 0, 0)^T$ where v is the round number and j is the column of the current state $G_{i,j}$.

For θ where the round constant($\varkappa(j)$) is added to current state using modulo 2^{64} addition operation, $\varkappa(j) = ((0xF3, 0xF0, 0xF0, 0xF0, 0xF0, 0xF0, 0xF0, (c-1-j) \ll 4))^T$ where c is the total number of columns in the state which is 8 for $l = 512$ and 16 for $l = 1024$ and j is the column of current state $G_{i,j}$.

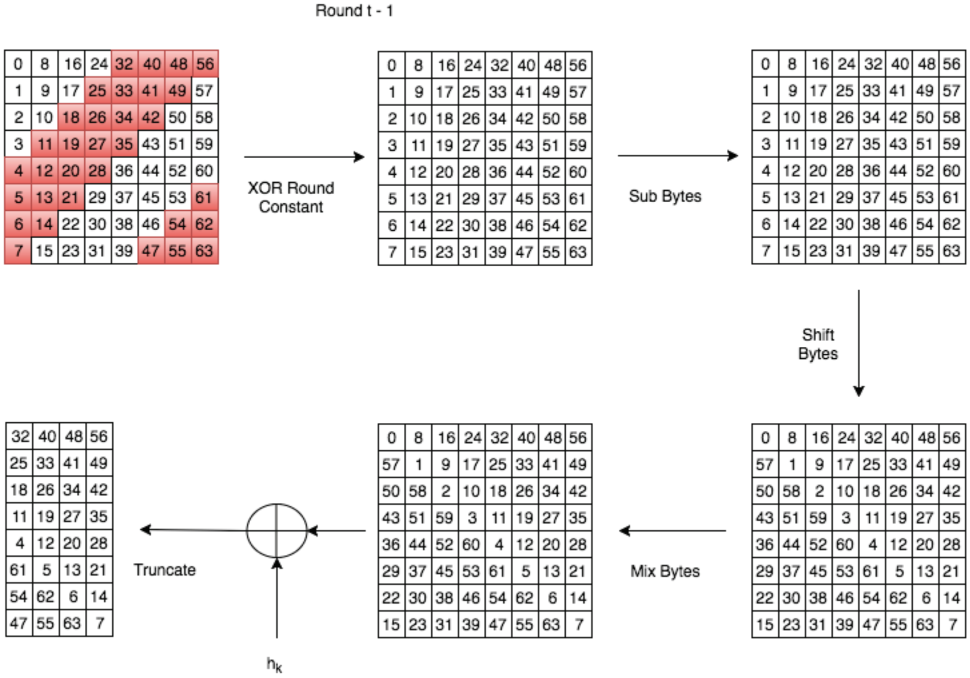
For the description of our attacks, we use the same notation to refer to the state inputs except that we will have two subindices. The first one refers to the block number, depending on which m_i is the input to the block and block numbers go from $1 \cdots k$, and the last $\tau^{\oplus}$ block before truncation is a special block which will be referred to as block number *k+1*. The second index refers to the round number where the round number goes from 0 to $(t-1)$. For example, if we refer to block number k and the state input to substitution layer in round $t-1$ in $\tau^{\oplus}$ operation of the block, we will refer that state as $\tau^{\oplus}[\sigma_{k,t-1}]$ where t is the total number of rounds. In order to refer to a specific byte in the state located at row *i* and column *j*, we use $\tau^{\oplus}[\sigma_{k,t-1}][i][j]$. In order to refer to a specific function ($\tau^{\oplus}$ or τ^{+}) within a block without referring to a specific state, we use the notation which includes the name of the function and the block number. As an example, $\tau^{\oplus}_{k+1}$ refers to $\tau^{\oplus}$ function just before the truncation step.

5.2 Differential Fault Analysis on Kupyna with Known SBoxes

Kupyna can be used to build MAC schemes in secret-IV or secret prefix mode. In the secret IV mode, the IV is secret and it needs to be recovered in order to break the MAC scheme. In the secret prefix mode, a secret input is appended to the input message such that the message input to the hash function becomes $Prefix \mid\mid M$. If the attacker recovers the secret prefix, then the MAC scheme is broken. In this attack we are going to recover all the message blocks and the value of IV which allows us to break the MAC scheme if the MAC scheme is used in secret-IV or secret prefix settings.

Our approach contains two stages. First, we aim to recover h_k which is the input to $\tau^{\oplus}[\kappa_{k+1,0}]$ as shown in Figure 9. So, $\tau^{\oplus}_{k+1}$ is the first block we are going to apply faults. After recovering h_k, our next goal is to recover inputs to each block $(m_1, m_2, \cdots, m_k)$. During this process, the IV is also recovered. It should be noted that the size of the hash output is not the same as the size of the state since there is truncation step in Kupyna. Hence, we first recover some bytes of the state $\tau^{\oplus}[\sigma_{k+1,t-1}]$, we will refer to that partial state as $\tau^{\oplus}[\sigma_{k+1,t-1}]^{\cdot}$. After that, we recover the full state of $\tau^{\oplus}[\sigma_{k+1,t-2}]$ which allows us to recover h_k by going backward. In order to recover the inputs to a message block i, which are h_{i-1} and m_i, we first recover the input to $\tau^{+}[\sigma_{i,t-1}]$, then going backward allows us to recover m_i. After that we recover the input to $\tau^{\oplus}[\sigma_{i,t-1}]$ and going backward allows us to recover the value of $m_i \oplus h_{i-1}$. By knowing m_i and $m_i \oplus h_{i-1}$, we can recover h_{i-1}. Then the same approach is recursively applied to all the blocks until all inputs are recovered.

In order to recover the partial state $\tau^{\oplus}[\sigma_{k+1,t-1}]^{\cdot}$, we first need to find bytes indices in the state $\tau^{\oplus}[\sigma_{k+1,t-1}]$ that propagate to the output (see Figure 11 which shows indices of bytes that survive after the truncation step). Faults are applied in the input to $\tau^{\oplus}[\sigma_{k+1,t-1}]$. Some faults give us information about state bytes in the input to last round SubBytes operation. However, some faults do not give any information about the input state to the last round SubBytes. As shown in Figure 11, if a fault is applied to the byte index 0 among the indices, then this fault does not give any information about the partial state $\tau^{\oplus}[\sigma_{k+1,t-1}]^{\cdot}$. However, if the fault is applied to the byte index 32, this fault gives us information about the byte index 32 (row 0, column 4) of the partial state $\tau^{\oplus}[\sigma_{k+1,t-1}]^{\cdot}$. Using the same approach, we recover all state bytes in the partial state that are propagated to the output.



0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

0	8	16	24	32	40	48	56
57	1	9	17	25	33	41	49
50	58	2	10	18	26	34	42
43	51	59	3	11	19	27	35
36	44	52	60	4	12	20	28
29	37	45	53	61	5	13	21
22	30	38	46	54	62	6	14
15	23	31	39	47	55	63	7

0	8	16	24	32	40	48	56
57	1	9	17	25	33	41	49
50	58	2	10	18	26	34	42
43	51	59	3	11	19	27	35
36	44	52	60	4	12	20	28
29	37	45	53	61	5	13	21
22	30	38	46	54	62	6	14
15	23	31	39	47	55	63	7

32	40	48	56
25	33	41	49
18	26	34	42
11	19	27	35
4	12	20	28
61	5	13	21
54	62	6	14
47	55	63	7

Figure 11: Propagation of bytes to the output in Kupyna

In order to find the value of byte input in the partial state $\tau^{\oplus}[\sigma_{k+1,t-1}]^{\cdot}$ for a specific index among indices that survive the truncation step, we use the following property of the SBoxes:

If x is a random input byte, and if Δ_i is chosen from the set:

$$\Delta = \{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80\}$$

where $1 < i \leq n$, and n is the number of faults. Then, x is uniquely defined only using the values for Λ_i in the following equation [4]:

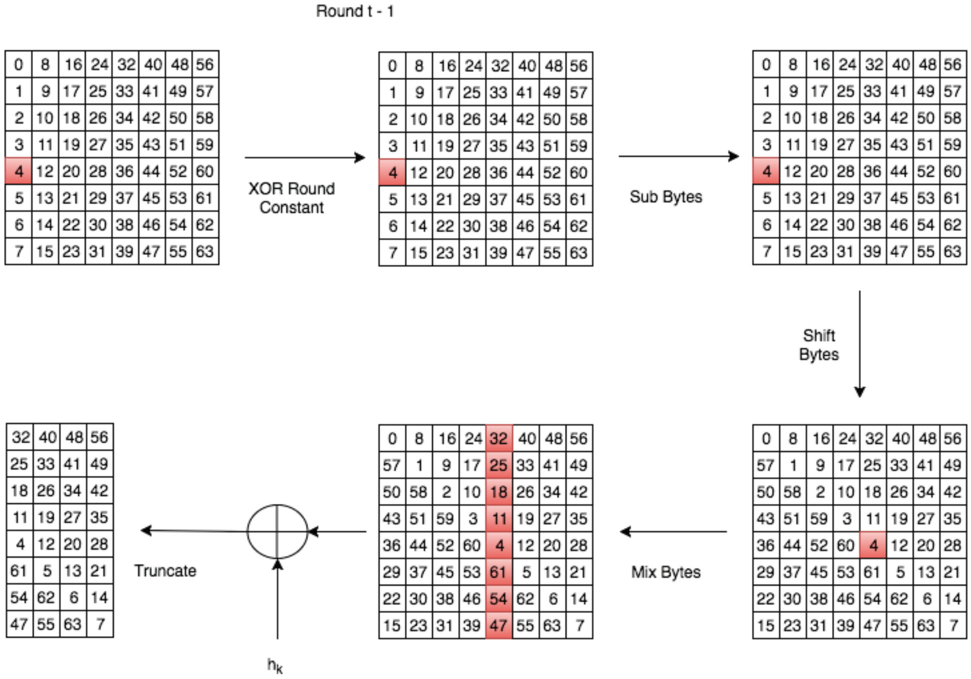
$$S(x) \oplus S(x \oplus \ \Delta_i) = \Lambda_i \tag{5}$$

This basically means that for x can be recovered using n output differences Λ_i corresponding to n one-bit distinct faults Δ_i. According to our experiments, the average number of faults required to recover x uniquely was 2.42 if known byte random fault model was used. If known byte unique fault model was used, the average number of faults required to recover x uniquely was 2.21.

In order to retrieve the state bytes in the partial state $\tau^{\oplus}[\sigma_{k+1,t-1}]^{\cdot}$, first the attacker calculates the correct hash value for a given message. After that, the attacker faults the computation at $\tau^{\oplus}[\sigma_{k+1,t-1}]$ by applying one of the values among Δ to a byte value. The attacker knows which byte was faulted if it is a chosen fault model. However, in the random fault model, the attacker does not know which byte was faulted. After the attacker applies the fault, she receives a faulty result. We call the correct hash value as H and faulty hash value as $H^{'}$. After getting H and $H^{'}$, the attacker calculates the difference between H and $H^{'}$ backward by calculating the following equation:

$$\Omega = \alpha^{-1} \circ \psi^{-1}(H \, \oplus \, H^{'}) \tag{6}$$

Figure 12 shows the propagation of a fault applied at byte index 4 in the input to SubBytes operation in the last round.



0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

0	8	16	24	32	40	48	56
57	1	9	17	25	33	41	49
50	58	2	10	18	26	34	42
43	51	59	3	11	19	27	35
36	44	52	60	4	12	20	28
29	37	45	53	61	5	13	21
22	30	38	46	54	62	6	14
15	23	31	39	47	55	63	7

0	8	16	24	32	40	48	56
57	1	9	17	25	33	41	49
50	58	2	10	18	26	34	42
43	51	59	3	11	19	27	35
36	44	52	60	4	12	20	28
29	37	45	53	61	5	13	21
22	30	38	46	54	62	6	14
15	23	31	39	47	55	63	7

32	40	48	56
25	33	41	49
18	26	34	42
11	19	27	35
4	12	20	28
61	5	13	21
54	62	6	14
47	55	63	7

Figure 12: Propagation of a fault applied at byte index 4 in Kupyna

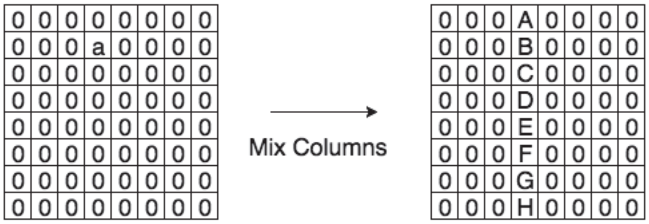
Ω is calculated as the inverse MixBytes and the inverse ShiftBytes operations applied to the difference between the correct state and the faulty state. The result of this calculation will be all zeros except in one byte position and that byte position gives us which byte was faulted and the nonzero value in that byte position gives us one value for Λ. Using this Λ, we find candidates for the byte using equation 5. After that, we find other values for Λ and narrow down the list of candidates until there is only one candidate remaining. This allows us to recover the partial state $\tau^{\oplus}[\sigma_{k+1,t-1}]^{\cdot}$.

After recovering the partial state of $\tau^{\oplus}[\sigma_{k+1,t-1}]$, the next step is to recover the full state just before that partial state. This state, $\tau^{\oplus}[\kappa_{k+1,t-1}]$, can be recovered directly since the round constant is known. However, recovering the input to the MixColumns operation just before the addition of the round constant in the last round ($\tau^{\oplus}[\psi_{k+1,t-2}]$) is a bit more difficult.

In order to recover the input to MixColumns operation in the round $(t-2)$, we are going to use the following property of MixColumns operation used in Grostl [34]:

Let $s \,\epsilon\, G$ be a state matrix with only one non-zero entry in row i and column j such that $G_{i,j} = s^{'}$ and $G_{i,j} = 0$ for all other entries, then the value of $s^{'}$ can be determined by knowing two entries in MixColumns operation applied to G.

This basically means that if we give an input to MixColumns operation a state matrix which has only one nonzero entry, we can get this nonzero entry uniquely if we know at least two bytes in the result of the MixColumns operation. Note that, this MixColumns operation changes byte values in the same column as that nonzero entry. As shown in Figure 13, if we give a state matrix as input which is all zeros except byte a, then we can find a uniquely if we know at least two entries among $A, \cdots, H$. In order to use this property, we fault the SBox input to round $(t-2)$, which is $\tau^{\oplus}[\sigma_{k+1,t-2}]$. After faulting the SBox input at round $(t-2)$, the attacker retrieves the correct and the faulty hash results which are H and $H^{'}$. Then, the attacker calculates the difference between these as $H \oplus H^{'}$



0	0	0	0	0	0	0	0
0	0	0	a	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

0	0	0	A	0	0	0	0
0	0	0	B	0	0	0	0
0	0	0	C	0	0	0	0
0	0	0	D	0	0	0	0
0	0	0	E	0	0	0	0
0	0	0	F	0	0	0	0
0	0	0	G	0	0	0	0
0	0	0	H	0	0	0	0

Figure 13: Kupyna MixColumns with input which has only one nonzero byte

and calculates this difference backward by calculating the following equation:

$$\Omega^{'} = \alpha^{-1} \circ \psi^{-1}(H \ \oplus \ H^{'})$$

However, since the SBox operation is not linear, in order to calculate the difference between the correct state and the faulty state just before SBox operation, we will use the bytes of partial state $\tau^{\oplus}[\sigma_{k+1,t-1}]^{\cdot}$ we recovered in the previous step which corresponds to actual bytes in the correct state. Let x_i be one of the bytes recovered in the previous step, i be the byte index for x_i and let $G^{'}[i]$ be the byte at index i at the faulty state , then

$$G^{'}[i] = S(x_i) \ \oplus \Omega^{'}[i]$$

gives us the byte value of the faulty state after the SubBytes operation and before ShiftRows operation at round $(t-1)$, $\tau^{\oplus}[\alpha_{k+1,t-1}]$. We do the same for all values of x_i we recovered in the previous step and we get the partial bytes of the faulty state after the first SubBytes operation in the last round. After that, if we go backward, by applying the inverse of SubBytes (σ^{-1}) operation to the bytes of the faulty state, we get the state difference after addition of the round constant in the last round.

$$G^{'}[i] = \sigma^{-1}(G^{'}[i])$$

Since we know the bytes just before SubBytes operation in the correct state and in the faulty state, we can then calculate their state difference as:

$$\Omega^{''}[i] = G^{'}[i] \oplus x_i$$

This difference will have only one nonzero bytes in one column and using the property of MixColumns operation, we are able to retrieve the location of the byte and the value of the byte at the input to MixColumns operation at round $(t-2)$, $\tau^{\oplus}[\psi_{k+1,t-2}]$. This is the state matrix which has one nonzero byte value and applying the inverse of ShiftRows to this state matrix ($\alpha^{-1}(G^{'})$), we get the difference after the SubBytes operation at round $(t-2)$ and using equation 5, we are able to get SBox input to round $(t-2)$, so we are able to recover the full state $\tau^{\oplus}[\sigma_{k+1,t-2}]$. Then we are able to recover h_k by going backwards from the state $\tau^{\oplus}[\sigma_{k+1,t-2}]$.

After recovering h_k, our goal is to recover the inputs to each block in order to break the cipher. Since the attacker is only able to observe the correct and the faulty results, we need to create a lookup table for each possible case when we apply fault analysis to recover inputs of each block. This lookup table is created by applying possible differences that can come from faults to h_k before applying h_k to the truncation function. In order to recover m_k and h_{k-1} for the block k, first the attacker faults $\tau^{+}[\sigma_{k,t-1}]$ and recovers the full state at the input to the SBox operation at τ^{+} block of round $(t-1)$. After that, the attacker inverts that full state and recovers m_k.

The next step is to recover h_{k-1}. In order to do so, the attacker faults $\tau^{\oplus}[\sigma_{k,t-1}]$ and recovers the full state at the input to the SBox operation at $\tau^{\oplus}[\sigma_{k,t-1}]$ block of round $(t-1)$ and inverts this state to recover $h_{k-1} \oplus m_k$ as shown in Figure 9. Using this known input and the recovered m_k from the previous step, the attacker is able to recover h_{k-1}. The same procedure is applied for all other blocks from $1 \cdots (k-1)$.

For the known byte random and known byte unique fault models, each fault gives us information about the state we are aiming to recover. However, in the random fault model, some faults may end up giving no information about the state we are aiming to recover since the difference between the correct result and the faulty result may be 0 if the faulted byte was truncated by the truncation step.

So far, we have shown how to break Kupyna if it is used in the secret-IV or secret prefix mode. We basically recover all inputs to the hash function including the value of IV and m_1. If Kupyna is used as the underlying block for HMAC, then the output is calculated as:

$$HMAC(K, m) = H((K \oplus opad)||H((K \oplus ipad)||m))$$

In this function, opad and ipad are known constants and K is the secret key we are aiming to recover (see Figure 14).

In this case, we first recover the input to the hash function which is $((K \oplus opad)||H((K \oplus ipad)||m))$. This allows the attacker to know the values of $H((K \oplus ipad)||m)$ and since the value of opad is known, the attacker can determine the value of K. Using the value

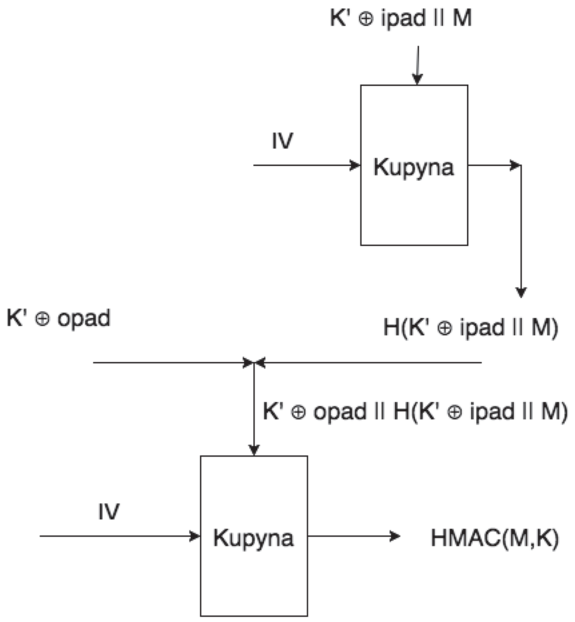


Figure 14: Kupyna in HMAC mode

of $H((K \oplus ipad)||m)$, which is the output of the first hash block as shown in Figure 14, the attacker can recover the input which is $((K \oplus ipad)||m)$. Then, the value of K can be recovered along with the value of m if the secret prefix is applied to the value of m.

In the case of NMAC, two keys are employed and the NMAC is calculated as:

$$NMAC(M) = H_{K_2}(H_{K_1}(M))$$

This is basically applying K_2 and K_1 instead of the known value of IV in Figure 14. Similar to our attack in the HMAC case, we first recover K_2 which is the input to outer block, then we recover K_1 which is the input to the inner block.

5.3 Fault Analysis on Kupyna with Unknown SBoxes

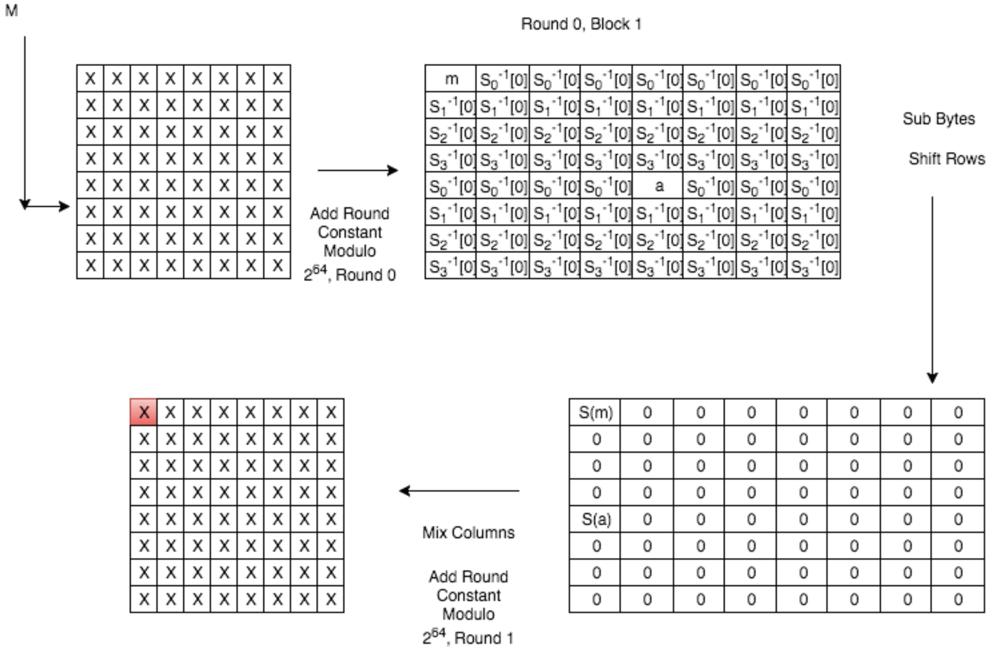
In the fault analysis performed on Kuznyechik with secret SBoxes [2], the first step was to recover the first round key based on possible values for $S^{-1}[0]$. Since $S^{-1}[0]$ can take 256 values, the attacker is able to find the correct value of $S^{-1}[0]$ in 256 trials in the worst case. After finding the correct value of $S^{-1}[0]$, the attacker retrieves input values for the other SBox outputs using the recovered value for $S^{-1}[0]$. In order to refer to a specific SBox in a specific block and at a specific round, we use the notation $S_{b,t}[i][j]$ where b is the block number, t is the round number and i and j denote the row and column numbers of the SBox. All faults are applied to SBoxes in τ^+ function since it is easier for the attacker to control the input to τ^+ by choosing different messages as inputs. The attacker has direct control over all message blocks except the last one since the last message block is used for padding.

In Kupyna, unlike AES [21] and Kuzneychik [2], there is no secret round key. Thus we are able to determine the SBox inputs that produce 0 as outputs for all the 4 different SBoxes directly. This is a chosen plaintext attack. The only way to fault the computation is to force a specific SBox output to 0. In order to recover $S_i^{-1}[0]$ for a specific SBox, firstly, the attacker needs to find ineffective bytes [21], i.e., bytes values in the input for an index that make the SBox output for that index 0. In this case, the SBox fault becomes ineffective fault since the correct calculation and faulty calculation produce the same results. Algorithm 5 describes the attack to recover the SBox inputs that produce 0 output for each SBox:

Algorithm 5 Recovering ineffective bytes in Kupyna

```
$RC_0$ refers to the round constant matrix for round 0 in $\tau^+$
Set input M to 32 bits which has all zeros. We refer to input as M = $m_1||m_2||m_3||m_4$.
Initialize Ineffective Bytes[$0 \cdots 3$] to all 0 s
for Each SBox index $i$ from 0 to 3 do
    for Each byte value from 0 to 255, called currentByte and IneffectiveByte is not found do
        Set $m_i$ to currentByte
        Set input M to all zeros except $m_i$ which is $currentByte$
        Calculate the hash value without fault, called H
        Calculate hash value with fault to SBox $S_{1,0}[i][0]$
        if H is equal to H' then
            Set $IneffectiveBytes[i] = currentByte$ and ineffectiveByte is found.
        end if
    end for
    Calculate the SBox inputs which give output 0 for each SBox $i$ as $S_i^{-1}[0] = IneffectiveBytes[i] - RC_0[0][i]$
end for
```

After finding the ineffective bytes, we subtract the round constants that are added to those ineffective bytes to recover the SBox inputs. In that calculation, we do not take the carry into account even though in τ^+ addition of round constant is modulo 2^{64} addition, since we are only dealing with the addition operation in one byte. After finding the SBox inputs that lead to zero, the next step is to recover all the SBox inputs and outputs. In order to do so, we fault the SBoxes in the second round and we use two nonzero values in the first column of the state, one being an SBox entry we recovered before and the other being the byte input we would like to find and which forces the output of SBox in the second round to 0. Figure 15 shows how the attacker is able to choose the input to hash function in a way such that the state becomes all zeros except two positions after the first round SubBytes and the ShiftRows operations. Figure 15 also shows how the state propagates after $\sigma_{1,0}$. In this figure a is an SBox input that has been recovered and $S(a)$ is the corresponding output, and m is the byte value that propagates from 0 to 255 until an ineffective fault occurs. The attacker also needs to take the ShiftRow operation into account since the input to the MixColumns operation has to be all zeros except the two bytes in the first column. The location of the fault is shown in red and faults are applied to the bytes in the state after $\sigma_{1,1}$. For S_0, we use indices 0 and 4. For S_1, we use indices 1 and 5. For S_2, we use indices 2 and 6 and for S_3, we use indices 3 and 7.


M
Round 0, Block 1
Add Round Constant Modulo 2^{64}, Round 0
Sub Bytes
Shift Rows
Mix Columns
Add Round Constant Modulo 2^{64}, Round 1

X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X

m	$S_0^{-1}[0]$	$S_0^{-1}[0]$	$S_0^{-1}[0]$	$S_0^{-1}[0]$	$S_0^{-1}[0]$	$S_0^{-1}[0]$	$S_0^{-1}[0]$
$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$
$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$
$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$
$S_0^{-1}[0]$	$S_0^{-1}[0]$	$S_0^{-1}[0]$	$S_0^{-1}[0]$	a	$S_0^{-1}[0]$	$S_0^{-1}[0]$	$S_0^{-1}[0]$
$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$	$S_1^{-1}[0]$
$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$	$S_2^{-1}[0]$
$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$	$S_3^{-1}[0]$

S(m)	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
S(a)	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X

Figure 15: Kupyna SBox fault at round 1 and recovering SBoxes

Algorithm 6 explains steps to recover all SBox entries:

Algorithm 6 Recovering all SBox entries in Kupyna

```
Initialize RecoveredValues to empty set for each SBox.
FirstIndex refers to index where $m$ is iterated
SecondIndex refers to index where $a$ is used.
for each SBox index $i$ from $0$ to $3$ do
    Initialize $a$ to $S_i^{-1}[0]$ recovered in algorithm 5
    **for** each index from $0$ to $7$ which refers to bytes in the first column of $\psi_{1,0}$ **do**
        **for** each $m$ from $0$ to $255$ **do**
            Find hash input which makes MixColumns input in round $0$, $\psi_{1,0}$ to all zeros except $\psi_{1,0}[FirstIndex][0] = S_i(m)$ and $\psi_{1,0}[SecondIndex][0] = S_i(a)$
            Get the Hash output without fault, called $H$
            Get the Hash output with faulted SBox at $S_{1,1}[index][0]$ called $H^{'}$
            If ineffective fault occurs, find the value of $S(m)$ according to the MixColumns operation that makes the output byte at index $S_{1,1}[index][0] = 0$
            If $S_i(m)$ is found uniquely, add $(m, S(m))$ to $RecoveredValues[i]$
        **end for**
        Set $a$ to the next value in $RecoveredValues[i]$ and find another $m$,
        If all possible $RecoveredValues[i]$ are used for $a$, then go to next index
    **end for**
**end for**
```

The above attack allows us to recover all the four SBoxes used in Kupyna. Since the round key addition is modulo 2^{64} addition, sometimes we get two candidates for $S(m)$. In this case, we ignored those candidates and only add entries when we are able to determine $S(m)$ uniquely for a given m.

5.4 Simulation Results

For the first experiment, we simulated Kupyna-256 using three different fault models and using random inputs of 64 bytes which are processed in two blocks after padding. For the known byte fault models, we estimated the number of faults required to recover the SBox input for each byte, that is the number of Λ values that uniquely determine the value of x in equation 5. For the unique fault model, the average number of Λ values that uniquely determine x is 2.21. For the random fault model, the average number of Λ values that uniquely determine x is 2.42. After that, we experimented each fault model 10 times and calculated the average number of faults required to recover half of state at $\tau^{\oplus}[\sigma_{k+1,t-1}]$ and to recover $\tau^{\oplus}[\sigma_{k+1,t-2}]$, which are needed to recover h_k. After that, we need to recover the two inputs to each of the two blocks which are input values for h_i and m_i. Our simulation results can be summarized as follows:

- For the known byte random fault model, the average number of faults required to recover half of state $\tau^{\oplus}[\sigma_{k+1,t-1}]$ is 79.3, and the average number of faults required to recover the full state $\tau^{\oplus}[\sigma_{k+1,t-2}]$ is 155.8. For block number 1, the average number of faults required to recover the message input(m_1) is 153.1, the average number of faults required to recover the hash input(IV) is 157.7. For block number 2, the average number of faults to recover the message input(m_2) is 154.3, and the average number of faults to recover the hash input(h_1) is 155.1. These results are consistent regarding the number of expected faults required to recover each byte.

- For the known byte unique fault model, the average number of faults required to recover half of the state $\tau^{\oplus}[\sigma_{k+1,t-1}]$ is 70.7, and the average number of faults required to recover the full state $\tau^{\oplus}[\sigma_{k+1,t-2}]$ is 141.3. For block number 1, the average number of faults required to recover the message input(m_1) is 142.2, the average number of faults required to recover the hash input (IV) is 142.1. For block number 2, the average number of faults required to recover the message input(m_2) is 140.6, and the average number of faults required to recover the hash input(h_1) is 143.9. These results are consistent regarding the number of expected faults required to recover each byte.

- For the random fault model, not every fault results in useful information since bytes affected by the fault may be eliminated because of the truncation. In this model, the average number of faults required to recover half of the state $\tau^{\oplus}[\sigma_{k+1,t-1}]$ is 454.5, and the average number of faults required to recover the full state $\tau^{\oplus}[\sigma_{k+1,t-2}]$ is 587. For block number 1, the average number of faults required to recover the message input(m_1) is 535.4, the average number of faults required to recover the hash input(IV) is 570.3. For block number 2, the average number of faults required to recover the message input(m_2) is 558.5, and the average number of faults required to recover the hash input(h_1) is 543.1.

For the second experiment, the only thing we had to verify is the fact that our attack successfully recovers the SBox entries for all SBoxes. In order to do so, we experimented 10 times by using random SBoxes. In all cases, the SBox entries were recovered successfully.

5.5 Conclusion

In this chapter, we investigated two different fault attacks against Kupyna. In the first attack, we assumed that Kupyna was used as the underlying hash function in a MAC scheme and showed how the attacker is able to recover the secret inputs. In the second attack, we assumed that Kupyna is used with secret SBoxes. According to our experiments, if Kupyna is using two blocks for the given input, the input and the value of IV can be recovered using about 855.3 faults in average when utilizing the known byte random fault model. We need

780.8 faults in average when utilizing the known byte unique fault model, and 3248.8 faults in average when utilizing the random fault model. We have also shown that keeping the Kupyna SBoxes secret does not add a larger security margin since the round constants are known and SBoxes can be recovered easily using the ineffective fault injection. In most of our experiments, the attacker had to go through 4 indices in the second round in order to recover all SBox entries. However, our first attack is not applicable if the number of output bits for Kupyna is smaller than 128 since we are not able to uniquely define input bytes to ψ operation. However, this is not a limitation for our attack since the recommended modes of usage for Kupyna are Kupyna-256, Kupyna-384 and Kupyna-512 [60].

Chapter 6

Conclusion and Future Work

To conclude, we analyzed two standard block ciphers and a standard hash function against fault analysis attacks. Previous work on fault analysis shows that it is important to check newly designed ciphers for their resistance against fault analysis. Ensuring the security of the considered three primitives is important since they are going to be widely deployed given the fact that they are national standards for the Russian Federation and Ukraine

We applied differential fault analysis to Kuznyechik and showed that 4 faults are required to break the cipher. We applied differential fault analysis and ineffective fault analysis to Kalyna and showed that even Kalyna512-512 can be attacked with a relatively small number of fault injections. Ineffective fault analysis on Kalyna shows that even if the SBoxes are secret, unprotected implementations of Kalyna can still be attacked using fault analysis. Lastly, we analyzed the hash function Kupyna against its resistance to fault analysis and showed that secret inputs of Kupyna, when used as the underlying hash function in different MAC schemes, can be retrieved using fault analysis.

In what follows, we provide a summary of some possible future work directions:

- There are many different fault attacks applied to AES in the literature. Some of these attacks utilize fault models different than the ones considered in this thesis. The resistance of Kuznyechik, Kalyna and Kupyna should also be checked when these other fault models are utilized.

- Studying the resistance of the three cryptographic standards considered in this thesis against different forms of side channel attacks, such as timing and power analysis attacks, is an interesting research direction.

- Our work focused on how to break the considered ciphers using fault analysis. A natural future research direction would be studying how to protect implementations of these three ciphers against fault attacks without introducing a large area or time overheads to the implementations.

- Apart from the attack on Kuznyechik which requires only about 4 faults, the attacks on both Kalyna and Kupyna require somewhat a large number of faults which can be impractical in some implementation scenarios. Investigating how to reduce the number of required faults is certainly another challenging and interesting research direction.

- From our analysis of the three ciphers, it is clear that the key schedule of the cipher can play a great role in improving the cipher resistance against fault analysis attacks. A possible future research direction would be studying how to design the key schedule of the cipher in order to optimize its resistance to different forms of implementation dependent attacks, including fault analysis.

- In the current literature, there are many works studying the resistance of ciphers under attacks that can be looked at as combination of some mathematical attacks (e.g., differential linear cryptanalysis, multiple differential/linear cryptanalysis). An interesting research direction is to investigate new classes of attacks that are composed of both fault attacks and classical fault analysis attacks (e.g., fault analysis with meet-in-the-middle attacks).

Bibliography

[1] Ahmed Abdelkhalek, Mohamed Tolba, and Amr M. Youssef. Improved Key Recovery Attack on Round-reduced Hierocrypt-L1 in the Single-Key Setting. In Subhra Rajat Chakraborty, Peter Schwabe, and Jon Solworth, editors, *Security, Privacy, and Applied Cryptography Engineering: 5th International Conference, SPACE 2015*, pages 139–150. Springer International Publishing, 2015.

[2] Riham AlTawy, Onur Duman, and Amr M Youssef. Fault Analysis of Kuznyechik. In *4th Workshop on Current Trends in Cryptology, CTCrypt 2015*, pages 302–317, 2015.

[3] Riham AlTawy and Amr M Youssef. A meet in the middle attack on reduced round Kuznyechik. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 98(10):2194–2198, 2015.

[4] Riham AlTawy and Amr M. Youssef. Differential Fault Analysis of Streebog. In Javier Lopez and Yongdong Wu, editors, *Information Security Practice and Experience: 11th International Conference, ISPEC 2015*, pages 35–49. Springer International Publishing, 2015.

[5] Ross Anderson and Markus Kuhn. Tamper resistance-a cautionary note. In *Proceedings of the second Usenix workshop on electronic commerce*, volume 2, pages 1–11, 1996.

[6] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In Bruce Christianson, Bruno Crispo, Mark Lomas, and Michael Roe, editors, *Security Protocols: 5th International Workshop*, pages 125–136. Springer, 1998.

[7] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption: 21st International Workshop, FSE 2014*, pages 168–186. Springer, 2015.

[8] Jean-Philippe Aumasson, Jorge Nakahara, and Pouyan Sepehrdad. Cryptanalysis of the ISDB Scrambling Algorithm (MULTI2). In Orr Dunkelman, editor, *Fast Software Encryption: 16th International Workshop, FSE 2009*, pages 296–307. Springer, 2009.

[9] Paulo S. L. M. Barreto, Vincent Rijmen, Jorge Nakahara, Bart Preneel, Joos Vandewalle, and Hae Y. Kim. Improved Square Attacks against Reduced-Round Hierocrypt. In Mitsuru Matsui, editor, *Fast Software Encryption: 8th International Workshop, FSE 2001*, pages 165–173. Springer, 2002.

[10] A. Battistello and C. Giraud. Fault Analysis of Infective AES Computations. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013*, pages 101–107, Aug 2013.

[11] Daniel J Bernstein. Understanding brute force. In *Workshop Record of ECRYPT STVL Workshop on Symmetric Key Encryption, eSTREAM report*, volume 36, page 2005. Citeseer, 2005.

[12] Bernstein, Daniel J. Cache-timing attacks on AES, 2005.

[13] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99: International Conference on the Theory and Application of Cryptographic Techniques*, pages 12–23. Springer, 1999.

[14] Eli Biham, Orr Dunkelman, and Nathan Keller. A New Attack on 6-Round IDEA. In Alex Biryukov, editor, *Fast Software Encryption: 14th International Workshop, FSE 2007*, pages 211–224. Springer, 2007.

[15] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[16] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97: 17th Annual International Cryptology Conference*, pages 513–525. Springer, 1997.

[17] Dan Boneh, A. Richard DeMillo, and J. Richard Lipton. On the Importance of Eliminating Errors in Cryptographic Computations. *Journal of Cryptology*, 14(2):101–119, 2001.

[18] A. Boscher and H. Handschuh. Masking Does Not Protect Against Differential Fault Attacks. In *5th Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC '08*, pages 35–40, Aug 2008.

[19] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701 – 716, 2005. Web Security.

[20] Hamid Choukri and Michael Tunstall. Round reduction using faults. *FDTC*, 5:13–24, 2005.

[21] C. Clavier and A. Wurcker. Reverse Engineering of a Secret AES-like Cipher by Ineffective Fault Analysis. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013*, pages 119–128, Aug 2013.

[22] D. Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM Journal of Research and Development*, 38(3):243–250, May 1994.

[23] Franck Courbon, Philippe Loubet-Moundi, Jacques J. A. Fournier, and Assia Tria. Adjusting Laser Injections for Fully Controlled Faults. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design: 5th International Workshop, COSADE 2014*, pages 229–242. Springer International Publishing, 2014.

[24] Joan Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis*. PhD thesis, Doctoral Dissertation, March 1995, KU Leuven, 1995.

[25] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The block cipher Square. In Eli Biham, editor, *Fast Software Encryption: 4th International Workshop, FSE'97*, pages 149–165. Springer, 1997.

[26] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[27] Daemen, Joan and Rijmen, Vincent. AES Proposal: Rijndael, 1998, 1998.

[28] Hüseyin Demirci and Ali Aydın Selçuk. A Meet-in-the-Middle Attack on 8-Round AES. In Kaisa Nyberg, editor, *Fast Software Encryption: 15th International Workshop, FSE 2008*, pages 116–126. Springer, 2008.

[29] W. Diffie and M. E. Hellman. Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard. *Computer*, 10(6):74–84, June 1977.

[30] Vasily Dolmatov. GOST R 34.12-2015: Block Cipher" Kuznyechik". *Transformation*, 50:10, 2016.

[31] Orr Dunkelman, Gautham Sekar, and Bart Preneel. Improved Meet-in-the-Middle Attacks on Reduced-Round DES. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology – INDOCRYPT 2007: 8th International Conference on Cryptology*, pages 86–100. Springer, 2007.

[32] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential Fault Analysis on A.E.S. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *Applied Cryptography*

and Network Security: First International Conference, ACNS 2003, pages 293–306. Springer, 2003.

[33] T. Fuhr, E. Jaulmes, V. Lomné, and A. Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013*, pages 108–118, Aug 2013.

[34] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Thomsen Søren S. Grøstl - a SHA-3 candidate. In Helena Handschuh, Stefan Lucks, Bart Preneel, and Phillip Rogaway, editors, *Symmetric Cryptography*, number 09031 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[35] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014: 34th Annual Cryptology Conference*, pages 444–461. Springer, 2014.

[36] Leonora Gerlock and Abhishek Parakh. Linear Cryptanalysis of Quasigroup Block Cipher. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, CISRC ’16, pages 19:1–19:4. ACM, 2016.

[37] N. F. Ghalaty, B. Yuce, M. Taha, and P. Schaumont. Differential Fault Intensity Analysis. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014*, pages 49–58, Sept 2014.

[38] Gosudarstvennyi Standard GOST. 28147-89. *Cryptographic Protection for Data Processing Systems, Government Committee of the USSR for Standards*, 1989.

[39] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium*, volume 14, 1996.

[40] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Commun. ACM*, 52(5):91–98, May 2009.

[41] Howard M. Heys. A TUTORIAL ON LINEAR AND DIFFERENTIAL CRYPTANALYSIS. *Cryptologia*, 26(3):189–221, 2002.

[42] William Hnath. *Differential power analysis side-channel attacks in cryptography*. PhD thesis, Worcester Polytechnic Institute, 2010.

[43] Deukjo Hong, Bonwook Koo, and Yu Sasaki. Improved Preimage Attack for 68-Step HAS-160. In Donghoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology – ICISC 2009: 12th International Conference*, pages 332–348. Springer, 2010.

[44] Kyungdeok Hwang, Wonil Lee, Sungjae Lee, Sangjin Lee, and Jongin Lim. Saturation Attacks on Reduced Round Skipjack. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption: 9th International Workshop, FSE 2002*, pages 100–111. Springer, 2002.

[45] Marc Joye and Francis Olivier. Side-channel analysis. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 1198–1204, Boston, MA, 2011. Springer US.

[46] Marc Joye and Michael Tunstall. *Fault Analysis in Cryptography*. Springer Science & Business Media, 2012.

[47] Auguste Kerckhoffs. *La cryptographie militaire*. University Microfilms, 1978.

[48] Lars Knudsen and David Wagner. Integral Cryptanalysis. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption: 9th International Workshop, FSE 2002*, pages 112–127. Springer, 2002.

[49] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99: 19th Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[50] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96: 16th Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[51] R. Korkikian, S. Pelissier, and D. Naccache. Blind Fault Attack against SPN Ciphers. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014*, pages 94–103, Sept 2014.

[52] R. Li, C. Li, and C. Gong. Differential Fault Analysis on SHACAL-1. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009*, pages 120–126, Sept 2009.

[53] Yu Liu, Kai Fu, Wei Wang, Ling Sun, and Meiqin Wang. Linear cryptanalysis of reduced-round SPECK. *Information Processing Letters*, 116(3):259 – 266, 2016.

[54] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93: Workshop on the Theory and Application of Cryptographic Techniques*, pages 386–397. Springer, 1994.

[55] Mitsuru Matsui and Atsuhiro Yamagishi. A New Method for Known Plaintext Attack of FEAL Cipher. In Rainer A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT' 92: Workshop on the Theory and Application of Cryptographic Techniques*, pages 81–91. Springer, 1993.

[56] Robert P. McEvoy, Michael Tunstall, Claire Whelan, Colin C. Murphy, and William P. Marnane. All-or-Nothing Transforms as a countermeasure to differential side-channel analysis. *International Journal of Information Security*, 13(3):291–304, 2014.

[57] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[58] Frédéric Muller. A New Attack against Khazad. In Chi-Sung Laih, editor, *Advances in Cryptology - ASIACRYPT 2003: 9th International Conference on the Theory and Application of Cryptology and Information Security*, pages 347–358. Springer, 2003.

[59] Kaisa Nyberg. Generalized Feistel networks. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology — ASIACRYPT '96: International Conference on the Theory and Applications of Cryptology and Information Security*, pages 91–104. Springer, 1996.

[60] Roman Oliynykov, Ivan Gorbenko, Oleksandr Kazymyrov, Victor Ruzhentsev, Oleksandr Kuznetsov, Yurii Gorbenko, Artem Boiko, Oleksandr Dyrda, Viktor Dolgov, and Andrii Pushkaryov. A New Standard of Ukraine: The Kupyna Hash Function (DSTU 7564: 2014), 2015.

[61] Roman Oliynykov, Ivan Gorbenko, Oleksandr Kazymyrov, Victor Ruzhentsev, Oleksandr Kuznetsov, Yurii Gorbenko, Oleksandr Dyrda, Viktor Dolgov, Andrii Pushkaryov, Ruslan Mordvinov, and Dmytro Kaidalov. A New Encryption Standard of Ukraine: The Kalyna Block Cipher, 2015. Cryptology ePrint Archive, Report 2015/650.

[62] Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003: 5th International Workshop*, pages 77–88. Springer, 2003.

[63] Gilles Piret and Jean-Jacques Quisquater. Integral Cryptanalysis on reduced-round Safer++, 2003. Cryptology ePrint Archive, Report 2003/033.

[64] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[65] Yu Sasaki, Lei Wang, Shuang Wu, and Wenling Wu. Investigating Fundamental Security Requirements on Whirlpool: Improved Preimage and Collision Attacks. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security*, pages 562–579. Springer, 2012.

[66] J. M. Schmidt, M. Hutter, and T. Plos. Optical Fault Attacks on AES: A Threat in Violet. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009*, pages 13–22, Sept 2009.

[67] Adi Shamir. Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000: Second International Workshop*, pages 71–77. Springer, 2000.

[68] Claude E Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.

[69] Vasily Shishkin, Denis Dygin, Ivan Lavrikov, Grigory Marshalko, Vladimir Rudskoy, and Dmitry Trifonov. Low-weight and hi-end: Draft Russian encryption standard. *CTCrypt14, 05-06 June 2014*, pages 183–188, 2014.

[70] Sergei Skorobogatov. Low temperature data remanence in static RAM. *University of Cambridge Computer Laboratory Technical Report*, 536:11, 2002.

[71] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002: 4th International Workshop*, pages 2–12. Springer, 2003.

[72] Douglas R Stinson. *Cryptography: theory and practice*. CRC press, 2005.

[73] Xiaorui Sun and Xuejia Lai. Improved Integral Attacks on MISTY1. In Michael J. Jacobson, Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography: 16th Annual International Workshop, SAC 2009*, pages 266–280. Springer, 2009.

[74] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication: 5th IFIP WG 11.2 International Workshop, WISTP 2011*, pages 224–233. Springer, 2011.

[75] Jan Van Leeuwen. *Handbook of theoretical computer science (vol. A): algorithms and complexity*. Mit Press, 1991.

[76] David Wagner. Towards a Unifying View of Block Cipher Cryptanalysis. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption: 11th International Workshop, FSE 2004*, pages 16–33. Springer, 2004.

[77] Kai Wirt. Fault Attack on the DVB Common Scrambling Algorithm. In Osvaldo Gervasi, Marina L. Gavrilova, Vipin Kumar, Antonio Laganà, Heow Pueh Lee, Youngsong Mun, David Taniar, and Chih Jeng Kenneth Tan, editors, *International Conference on Computational Science and Its Applications – ICCSA 2005*, pages 577–584. Springer, 2005.

[78] Yongjin Yeom, Sangwoo Park, and Iljun Kim. On the Security of CAMELLIA against the Square Attack. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption: 9th International Workshop, FSE 2002*, pages 89–99. Springer, 2002.