Cryptographic analysis and implementation of GSM stream cipher A5/2 and attacks on it

Kryptografisk analyse og implementering af GSM stream cipher A5/2 samt angreb på den

Joakim Dich, 201809620
Kristian Høi Liboriussen, 202006178

Bachelor Report (15 ECTS) in Computer Science
Advisor: Ivan Bjerre Damgård
Department of Computer Science, Aarhus University
June 2023

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

In this project, we first introduce various concepts relevant to the understanding of the GSM A5/2 cipher and phone networks in general. We then present an explanation of the A5/2 cipher along with an implementation of an attack on it, originally described by Barkan et. al [BBK08]. The attack initially is shown as a known-plaintext attack which enables us to decrypt other parts of the encrypted communication and find the initial key after having known only parts of the communication. Furthermore, due to using the same initial key multiple times, this could break previous or future conversations as well. The attack is then evolved into an instant ciphertext-only attack which enables us to break any A5/2 encrypted communications with few constraints on the required data. We look at some of the ways A5/2 could affect the mobile networks it was a part of, as downgrading and a lack of key separation with A5/1 and the 2G implementation of A5/3 could turn these insecure. Finally we look into the developments on mobile networks after A5/2 was introduced for use in the 2G system and if the attack could be relevant today.

Joakim Dich and Kristian Høi Liboriussen,
Aarhus, June 2023.

Contents

Chapter 1

Introduction

Mobile communication today is present everywhere, almost everyone has a phone with them and it is always connected to a mobile network. The security of data communicated to and from these networks is therefore paramount as most communication through calls and texts is private by nature. Yet unlike hard-wired internet, where only the connection path can see communication, phone communication is always done by a radio broadcast in the open so the best way to secure the privacy of communication data is through encryption.

The 2G cellular network was commercially launched in 1991, as the world's first digital phone network. This allowed introduction of more advanced features in regards to encryption and authentication, namely two encryption algorithms A5/1 and A5/2. A5/2 was a simplified version of A5/1, meant for export outside Europe and the US. The hardware on mobile devices was, however, very limited at that time, which meant the various algorithms including the ciphering had to be very efficient, especially due to real-time communication such as calling. Implementing the same algorithms as used elsewhere could have been too inefficient and might not have fit with other conditions of the protocol as the GSM system utilizes the concept of a 228-bits frame. This gives a phone the possibility to transmit and receive a message of 114-bits each in the time of one frame, with the sizes being based on transfer times and the fact that a phone needs to be able to share a radio channel with up to 7 other phones. The GSM security algorithms were kept hidden from the public so the authors had the sole responsibility to check for any potential security weaknesses [BBK08].

Naturally some issues with the security of mobile networks arose from these early implementations. Various attacks on both the confidentiality and authenticity of data were published. One particularly impactful attack is the attack described by Barkan et al., allowing the attacker to find the key used for A5/2 encryption through only a few ciphertext messages [BBK08]. Barkan et. al did not publicize their implementations or specific values in the paper, but their attack was a part of A5/2 being removed from new phone implementations [3GP07].

We implement a variation of the attack described by Barkan et al. [BBK08], first their attack with a requirement to know a part of the plaintext and then changed into

functioning directly on the ciphertext without any plaintext knowledge. The attack gives us the initial key used, allowing us to decrypt other conversations as well due to reuse and a lack of key separation. A limitation of our implementation is that it cannot function on the actual GSM implementations of A5/2 due to the way frame numbers are used in the key setup. Our cipher implementation is simplified in the way that there is generated enough information under a single frame number to break the ciphering key, rather than having to switch frame numbers for every 228-bits. Due to the small amount of bits required for the attack, Barkan et al. described and implemented a way to handle this issue by solving a relationship between data encrypted under different frame numbers but the specific implementation values they used to achieve this was not publicized in their paper [BBK08].

Today, a switch to more open-source standards has happened, an example of this is the usage of AES-CTR in 4G networks [NNF18]. Yet the A5/2 attack can be adapted for use on A5/1, which is still a possible algorithm in some phones and networks. Phones prefer to use the strongest encryption they have, yet the phone network implementations are vulnerable to other issues such as downgrading attacks which can lead to usage of older ciphers like A5/1 despite others being available [EFF17].

Chapter 2

Mathematical Background

2.1 Galois Field

Galois Fields (GF) are an important topic in mathematics and have many interesting use cases, not only in cryptography. A Galois Field, sometimes also named a finite field, is a mathematical structure that behaves like a number system with a limited (hence finite) set of elements. Unlike the real numbers, which form an infinite field, a finite field contains a finite number of elements. It follows specific rules and operations that make it a field, such as addition and multiplication. A nice property of these fields is that each operation's result always remains within a specific field if done on elements from the field [Han18].

Arithmetic in modulo 2 is an example of a Galois field; the only elements that exist in this field are 1 and 0. Adding or multiplying any two elements in GF(2) will always result in either 0 or 1. Due to the simplicity, it is possible to construct a compact truth table for those two arithmetic operations.

Addition and multiplication in GF(2) are defined as:

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	0

(a) Addition truth table

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

(b) Multiplication truth table

Figure 2.1: Truth tables for GF(2)

These two operations are respectively the exact same operations as logical XOR and logical AND. Subtraction is typically not even defined, because the addition is identical to subtraction. Since there are only two elements, and minus 0 can be treated the same

as 0, then minus 1 has to be represented as something different than itself. Thus:

$$(y-x) = (x-y) = (x+y) \quad x,y \in \text{GF}(2)$$

This makes the field GF(2) even easier to work in as we only have to worry about a few different operations that can be applied to the two elements in the field.

2.2 Linear Feedback Shift Register

A Linear Feedback Shift Register (LFSR) is, in our implementation, treated as a type of data structure, closely related to arrays. It consists of a fixed sequence of bits with linear operations.

During each operation, the entire register is shifted to the right. The first bit is set to the result of the last bit XORed with other bits from the register. This collection of XOR operations is called the feedback polynomial. Secondly, the register will output a bit, based on different XORed bits compared to the feedback. This is called the taps. When using an LFSR, it switches between tapping the output bit and shifting the whole register. After n shifts the LFSR creates an output of n bits.

LFSRs are commonly used in applications such as stream ciphers and pseudo-random number generators. They are quite effective compared to their simple design and relatively basic operations, which makes them cost-effective. Due to the simplicity, LFSRs can be used directly in real hardware, using flip-flops as placeholders as bits, and logic gates as the XORs [Paa14].

LFSRs have a very strong connection with the finite field GF(2) because an LFSR can be described as a polynomial in GF(2). If the polynomial is primitive then it will be a maximum-length LFSR, meaning it would take $2^n - 1$ shifts for it to cycle to the same value. This is often used to minimize repetition and patterns, especially since the cycling time is known for the developer [Paa14].

A maximal-length LFSR polynomial with 19 bits can be denoted as:

$$x^{19} + x^{18} + x^{17} + x^{14} + 1 \qquad \text{taps:}[12, 14, 15]$$

The polynomial itself cannot be used to directly calculate the feedback shift. The main purpose of the polynomial is to determine whether it is primitive over GF(2). Exponents can then be seen as the feedback bits from the register. $x^0 = 1$ always denotes the first output bit at index 0. The taps are the bits being added together with the last bit whenever an output is made, therefore the output of this specific LFSR is the GF(2) addition of these bit positions:

$$\text{output} = [12] + [14] + [15] + [18]$$

2.3 Gauss elimination in GF(2)

In linear algebra, Gauss elimination is a method to solve a system of linear equations, by doing only three elementary row operations upon an augmented matrix. An augmented matrix consists of linear equations placed in the rows of the matrix, and the values of each equation in the last column. The procedure is efficient with its asymptotic running time $O(n^3)$ and only requires the following three operations [Swa22].

1. Swap the positions of two rows.
2. Multiply a row by a non-zero scalar.
3. Add to one row a scalar multiple of another

Operation (1) should work on all types of matrices whereas the possibility to use addition and multiplication in GF(2) makes operation (2) and (3) easy to determine as legal too. We implemented our own home-brewed Gauss solver called `gauss_solver.py`. All of it's source code is accessible in appendix A.1. It should be noted the solver is made specifically for the cases occurring when solving equations from the A5/2 cipher in GF(2). On other equation systems the solver might behave unexpectedly.

Generally, a system of linear equations in GF(2) can be defined as either:

(a) inconsistent meaning no solution exists, or
(b) consistent meaning it has
    (i) exactly one solution, or
    (ii) $n > 1$ many solutions, where $n = 2^v$ and v is the amount of free variables

If no solution exists, it would be impossible to reverse the values from the register, but it is also vital that the system has enough known values which requires the equations to be linearly independent. If some equations are dependent on each other it can lead to multiple valid solutions as one or more variables would be left undetermined. In the real numbers this would lead to infinite solutions, however, in GF(2) there are only two possible values for a variable, therefore the possible solution amount is limited, even with free variables.

We have checked some of the equation systems occurring when using the maximum-length LFSRs from the cipher implementation, these have all generated matrices which have enough linearly independent equations to be solvable. This fits nicely with the conclusions from Barkan et. al determining there is enough linearly independent equations to solve all cases possible from the A5/2 ciphers LFSRs [BBK08].

2.4 Majority function

The majority function maj is a boolean function defined to evaluate to true if the majority (more than half) of its inputs are true. The majority function with 3 inputs (a,b,c) has three different ways of being true since three variables make three distinct pairs. Intuitively the maj function can be expressed with logic gates as:

$$(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

To make it compatible with GF(2), the function can be changed to only consist of XOR and AND gates:

$$(a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$$

Furthermore we can write the following truth table for what the function outputs depending on its input:

a	b	c	$(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$	$(a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Figure 2.2: Truth table for majority function of 3 inputs

Chapter 3

A5/2 Standard

3.1 GSM Algorithms

The GSM naming scheme of its authentication, encryption, and key generation **a**lgotihms are respectively called **A**3, **A**5, and **A**8. The A5/2 cipher was developed in the 1990s as one of three encryption algorithms for GSM, along with A5/1 and A5/3. A5/2 is heavily inspired by A5/1, since A5/2 was designed to work with similar mobile hardware, it was also intended as a weaker variant for use in countries that restricted the export of strong encryption technologies. The A5/3 cipher is designed to use a 128-bit key size, however, the implementation used in 2G only allows for a key size of 64-bit.

GSM employs TDMA (Time-division multiple access), a technology enabling simultaneous communication among multiple users by assigning each user a dedicated time slot within each frame. In this system, each frame support 114 bits of data in each direction. Therefore 228 bits of keystream is needed in each frame to encrypt the data from each direction. Each frame is assigned with a frame number and messages then have to be split across multiple frames if they are longer than 114 bits. The maximum possible frame number is 2715647 after which it will loop back to the start [BBK08].

3.2 The A5/2 Cipher

The purpose of the A5/2 stream cipher is to create a 228-bit keystream k unique for every 228-bit frame. The final transmitted ciphertext in each frame is the keystream k XORed with the plain-text x. A5/2 is initialized with two inputs K_c and COUNT (sometimes also called the initialization vector) [BBK08].

K_c is the ciphering key, it is assumed both parties already have k_c stored safely even before the communication starts. COUNT is a publicly known initial value (IV) derived from the frame number f. The frame number is also publicly known, somewhat similar to how the sequence number is known in a TCP packet, while the message may be encrypted. The cipher converts computes COUNT from f by using the following

function. This will slightly decrease the predictability of COUNT.

$$T1 = \left\lfloor \frac{f}{51 \cdot 26} \right\rfloor$$

$T2 = f \bmod 26$ (yields a maximum of $\lceil log_2(26) \rceil = 5$ bits)

$T3 = f \bmod 51$ (yields a maximum of $\lceil log_2(51) \rceil = 6$ bits)

$COUNT = T1 + T2 + T3$

A5/2 consists of four maximum length LFSRs $R1, R2, R3, R4$, with sizes respectively 19, 22, 23, and 17 bits. The taps of the first three LFSRs $R1, R2, R3$ are directly used to calculate the output keystream. The last register $R4$ is a clocking unit, which is used to decide which registers of $R1, R2, R3$ that should shift when $R4$ shifts. [BBK08]. This makes it harder to predict which register bit a given output bit is calculated from.

The clocking unit works as follows: Shift $R1$ if the majority bit $maj(R4[3], R4[7], R4[10])$ is equal to $R4[10]$, shift $R2$ if the same majority bit is equal to $R4[3]$, and shift $R3$ if the bit is equal to $R4[7]$ [BBK08].

To produce one output bit for each shift, the cipher calculates the last register bit XORed with the majority result of the three taps. This is performed for register $R1, R2, R3$ and these 3 values are then XORed together, as the final output bit. Furthermore, a one-bit constant is XORed with one of the inputs to each output majority function. The following diagram shows how each shift works and how each output bit calculation is computed.

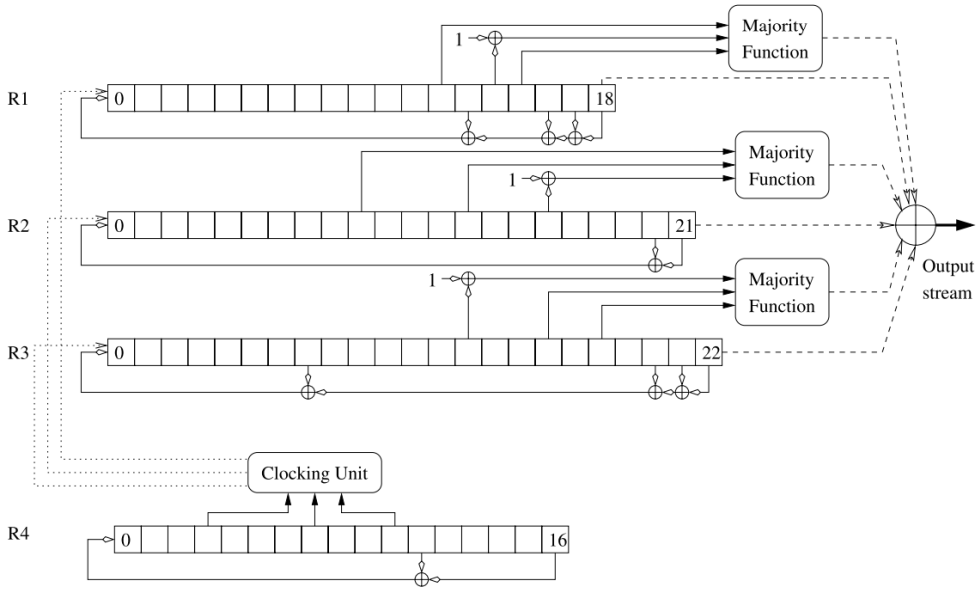

Figure 3.1: A diagram of the A5/2 cipher. Source: Barkan et. al [BBK08]

The exact feedback and tapping bits can be described as the following where exponents use 1-indexation in the primitive polynomials:

$$R1 = x^{19} + x^{18} + x^{17} + x^{14} + 1 \quad \text{with taps } [14, 17, 18, 19]$$
$$R2 = x^{22} + x^{21} + 1 \quad \text{with taps } [11, 15, 18]$$
$$R3 = x^{23} + x^{22} + x^{21} + x^{8} + 1 \quad \text{with taps } [14, 17, 19]$$
$$R4 = x^{17} + x^{12} + 1 \quad \text{with taps } [4, 8, 11]$$

Our A5/2 implementation is built from these steps, however, multiple steps regarding array shapes and the like are necessary. These are implementation specific and can be found in the file `A52_cipher.py` in appendix A.1.

3.3 Key Setup and keystream generation

The setup initializes the four registers, this is done by XORing the bits from K_c with the first register bits. Afterwards the same process is done with COUNT. After each XOR operation, all four registers are shifted [BBK08]. Finally one bit in each register is set to 1 in order to prevent a setup where a register is filled with 0's, as this would leave it stuck like that. The key setup procedure can be described with the following pseudo-code which corresponds to the method `setup()` in `A52_cipher.py`:

```
input: 64-bit Kc, 22-bit COUNT
output: 228-bit k

zero fill all four registers

for i from 0 to 63
    shift all four registers
    R1[0] ^= Kc[i]
    R2[0] ^= Kc[i]
    R3[0] ^= Kc[i]
    R4[0] ^= Kc[i]

for i from 0 to 22
    shift all four registers
    R1[0] ^= COUNT[i]
    R2[0] ^= COUNT[i]
    R3[0] ^= COUNT[i]
    R4[0] ^= COUNT[i]
R1[15] = R2[16] = R3[18] = R4[10] =  1
```

When the key setup is done, K_c and COUNT are no longer needed in the encryption process. The keystream generation can now begin, which is responsible for outputting the 228-bit keystream. However, it first shifts 99 times in order to get further away from the initial state [BBK08].

```
shift clocking unit 99 times
for i from 0 to 228
    shift clocking unit
    k[i] = output-bit
return k
```

This corresponds to the method `generate_keystream()` in our code.

Chapter 4

Known Plaintext Attack on A5/2

4.1 Reversing a single LFSR

Going in the reverse direction of an LFSR is not as simple as shifting forward. This is due to many different ways the first register value can be expressed as a result of multiple xor operations.

The linear property of LFSRs makes reversing still doable but by a completely different approach, which is to solve a system of linear equations. Therefore it is vital that the attacker has at least n bits from any shift, in order to reverse a n-bit LFSR [BBK08].

The main idea is to keep track of which original bits need to be XORed together to get any future register bit. This is achieved by treating each register bit as a vector of dependencies of the original register bits. To create the dependency vectors, we simply shift each vector and XOR them as if they are bits from a normal LFSR. To illustrate this process for a 19-bit LFSR, shift 0 and 1 can be shown as:

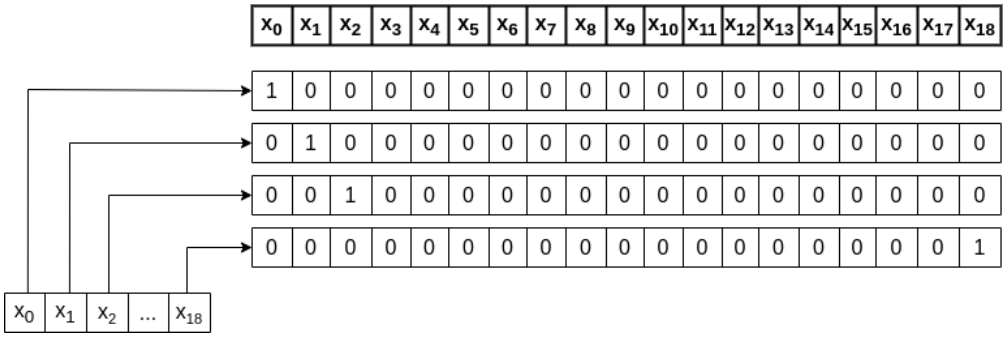

Figure 4.1: The dependency vectors for shift 0

In the next shift, the dependency vectors change to:

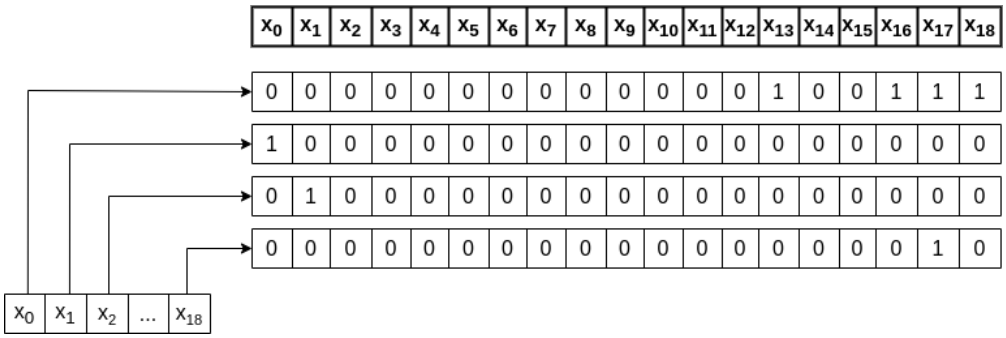

Figure 4.2: The dependency vectors for shift 1.

We can notice how x_0 is the xor of the original bit 13, 16, 17, 18. Meanwhile the rest of the vectors are shifted to the right.

4.2 An example of reversing a single LFSR

Let the LFSR have this random permutation in shift 0:

$$x_0, x_1, \ldots, x_{18} = 0,1,1,0,1,1,0,1,1,0,0,0,1,0,1,0,0,0,0$$

And the output sequence, after shifting 119 times, and outputting only the last bit from shift 100 to 119:

$$seq = 1,1,0,0,0,0,0,1,0,1,1,0,0,1,1,0,1,0,1$$

In order to reverse the original x_n bits, the first step is to generate the dependency vectors, by only looking at x_{19}'s dependencies. The 19 dependency vectors become the 19 row vectors in a 19x19 matrix. Which acts as 19 equations with 19 unknowns. Lastly, the matrix is appended with *seq* by adding it as a column vector to the matrix. Now a 20x19 augmented matrix is ready to be solved by using Gauss elimination.

$$\begin{bmatrix} \text{Shift } 100 \\ \text{Shift } 101 \\ \text{Shift } 102 \\ \vdots \\ \text{Shift } 117 \\ \text{Shift } 118 \\ \text{Shift } 119 \end{bmatrix} \rightarrow \left[\begin{array}{ccccccccccccccccccc|c} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right]$$

After running this matrix with the Gauss elimination algorithm, the result should always be a matrix in reduced row echelon form. Such as:

$$\left[\begin{array}{cccccccccccccccccc|c}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}\right]$$

Looking at the three first and last bits in the last column, it does match them with the original x_n values. The whole matrix can be seen in the appendix B.1.

4.3 Linearization

The A5/2 cipher uses the majority function which can be expressed in GF(2) as $maj(a,b,c) = (a \cdot b) + (a \cdot c) + (b \cdot c)$ as seen in fig 2.2 as part of the output calculation. This adds quadratic terms into each equation which makes it so each equation is no longer linear. This is an issue because solving a quadratic system is not possible with Gauss elimination, furthermore, the problem of solving quadratic systems is NP-complete. Yet by introducing the concept of linearization, the quadratic system will be converted back to a linear system [BBK08]. Linearization makes each and every product, a new unknown variable, which represents the result of the product. As a consequence of linearization, the number of unknowns will be increased by $\frac{n\cdot(n-1)}{2}$. The minus is one, is because the same bit can never be multiplied by itself. Furthermore, the 2 division is because of symmetry since, $a \cdot b = b \cdot a$. The equation $(a \cdot b) + (a \cdot c) + (b \cdot c)$ becomes equal to $lin_{ab} + lin_{ac} + lin_{bc}$, with lin_{ab} representing the linearized variable of $(a \cdot b)$. For a 19-bit LFSR, the amount of new linearized variables becomes $\frac{19\cdot18}{2} = 171$. To check the reversed bits after a found solution, each and every product variable should be equal to the product of its two register-bit variables. E.g. if $x_1 = 1 \wedge x_2 = 1$ then $x_{1,2} = x_1 \cdot x_2 = 1 \cdot 1 = 1$.

4.4 Mapping the linearized variables to positions

Each column in the equation matrix needs to correspond to the same unknown variable. Therefore we make a mapping function for the variables. For $n = 19$ we assume the first 19 unknowns are x_0 to x_{19}, while the last unknowns are the linearized variables. When dealing with a linearized variable, we only have information about its two product positions. Therefore it is needed to have a function that converts a 2D number to a 1D number. How the mapping function is supposed to work can easily be viewed in an $n \times (n-1)$ table. Meaning the desired function will output: $map(0,1) = 0, map(0,2) = 1, map(0,3) = 2, map(0,4) = 3, map(1,2) = 4$. With $n = 5$

$a\backslash b$	0	1	2	3	4
0	$\emptyset$	0	1	2	3
1	$\emptyset$	$\emptyset$	4	5	6
2	$\emptyset$	$\emptyset$	$\emptyset$	7	8
3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	9

$a\backslash b$	0	1	2	3	4
0	$\emptyset$	$0+b-1$	$0+b-1$	$0+b-1$	$0+b-1$
1	$\emptyset$	$\emptyset$	$3+b-1$	$3+b-1$	$3+b-1$
2	$\emptyset$	$\emptyset$	$\emptyset$	$5+b-1$	$5+b-1$
3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$6+b-1$

$a\backslash b$	0	1	2	3	4
0	$\emptyset$	$g(a)+b-1$	$g(a)+b-1$	$g(a)+b-1$	$g(a)+b-1$
1	$\emptyset$	$\emptyset$	$g(a)+b-1$	$g(a)+b-1$	$g(a)+b-1$
2	$\emptyset$	$\emptyset$	$\emptyset$	$g(a)+b-1$	$g(a)+b-1$
3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$g(a)+b-1$

Figure 4.3: Three tables showing how the function f can be broken down into steps

After some trial and error, using the triangular sequence numbers, the function g can be defined as:

$$g(a) = T(n-2) - T(n-2-a)$$
$$T = \{0,1,3,6,10,15,\ldots\}$$
$$T(n) = \frac{n(n+1)}{n}$$

The final mapping function $f : \mathbb{N}^2 \mapsto \mathbb{N}$ becomes:

$$f(a,b) = T(n-2) - T(n-2-a) + b - 1$$

Since the first n positions are reserved for the register unknowns, it is vital to use add n as an offset when using the mapping function. The function implementation can be seen in the file `util.py` with filename `formula()`.

4.5 Reversing the keystream back to the setup

The main goal of the attack is to obtain the cipher key K_c, however, this requires two distinct steps. The first step obtains the four registers immediately after the key setup (line 19), and the second attack reverses K_c given the setup valued registers. It has previously been shown how to reverse an LFSR register with a majority function. Reversing three of those registers XORed with each other will not complicate this too much. Having three LFSRs in combination, only increases the number of unknowns linearly, because each register's majority function operates independently of the other registers [BBK08]. The amount of unknowns for the A5/2 cipher becomes; $719 = 19 + \frac{19\cdot18}{2} + 22 + \frac{22\cdot21}{2} + 23 + \frac{23\cdot22}{2}$. This creates a minor problem because each

frame only has a 228-bit keystream. Taking the next 228 bits from the next frame would also impact the setup registers (later on we will show that this is not a problem, by combining step 1 and 2).

For simplicity, in our implementation, we have made the assumption that the A5/2 cipher may output 719 bits (instead of 228) for one given frame. In this scenario, apart from having eavesdropped the ciphertext, the attacker must have access to the 719-bit plaintext. That way the attacker can calculate the keystream by XORing the plaintext with ciphertext. After that, the attacker would need to find the dependencies of all the 719 unknowns, it may seem a bit overwhelming as we would be working with three registers at once, and each output bit is the result of an XOR operation with 6 inputs, yet it easily doable by computers. The equation matrix generation is implemented close to these steps and can be seen in the file `A52_equation.py`

Because the attacker doesn't know R4, it would be necessary to brute force all 2^{17-1} (-1 because R4[10] is always set to 1 in the setup) combinations of linear systems of equations, as each original R4 completes changes the dependencies of the equations [BBK08]. This brute force approach is still feasible, even assuming each attempt takes at most 1 second which is a very realistic upper bound, since GF(2) elimination and equation matrix generation runs in polynomial time. An estimated upper bound running time estimation becomes.

$$1 \text{ sec} \cdot 2^{16} = 65536 \text{ sec} \approx 18 \text{ hours}$$

This can be significantly reduced, by using specialized hardware (ASIC) because most of the operations used are used with AND and XOR operations. Also, each solve is completely independent, making utilization of parallelization result in a heavy time decrease. External libraries such as Galois [Hos23], can solve a 656x655 system matrix in under 0.3 seconds.

4.6 Reducing from 719 to 655 variables

In the key setup, one bit of each of the four registers is hard-coded to 1. Meaning the attacker would already know 4 bits of the variables. This means that the system of equations can be reduced by 4 equations. Furthermore, the number of linearized variables can also be decreased. E.g. $R1_{15,1}$ becomes $1 \cdot R1_1$. So we simply XOR the coefficient $R1_{15,1}$ with $R1_1$, making it safe to remove the column of $R1_{15,1}$. The system goes from 719 to $655 = 18 + \frac{18 \cdot 17}{2} + 21 + \frac{21 \cdot 20}{2} + 22 + \frac{22 \cdot 21}{2}$. Fewer variables clearly would also help on the equation solve time, especially as many solves will be attempted.

4.7 Precomputation

The equation matrix for each possible R4 does not depend on either f or Kc, therefore it is possible to precalculate all matrices even before the keystream is known to the attacker. Loading a matrix from memory is significantly faster than computing it. Our

test results suggest that the loading time for one equation matrix is in the magnitude of milliseconds.

Each system of equations will have 655 equations with 655 unknowns where each unknown variable can be represented by just a bit. Assuming the dimension is known beforehand, the size in bits for one matrix becomes 655^2. Multiplying that with the total number of possible R3 permutations, the total size becomes

$$655^2 \text{ bits} \cdot 2^{16} = 3.51 \text{ GB}$$

A further test shows that file compression can shrink the file down to 150 MB. Making the compression size 4.35 % of the original file size. After testing different compression algorithms, the lrzip implementation [Kol23] turned out to offer the best compression ratio. We believe this high compression ratio is due to the fact each line is very similar to the line before, most of the bits are the same but only shifted by one. The implementation of the precomputation can be seen in the codebase in the appendix A.1. The file is called `precomputation.py`.

4.8 Inverting the original key setup to get K_c

In order to find the ciphering key K_c that was used to set up the cipher we need to invert the key setup process after finding all four register states. We can do this by creating another equation system to solve with Gauss elimination. The result of the system would be the 64-bit key which can then be used to decrypt any further communications since frame numbers are public. In our implementation of it, see appendix A.1, we generate a system of relations calculated by performing the key setup using variables, rather than actual values. Yet in order to cover the fact that the key setup also sets some variables to 1 in the end we add a column representing the constant "1" at the end, this is mentioned as a method in Barkan et. al's paper as well [BBK08].

The key is filled into the registers as usual, shifting the register and then XOR'ing the specific key bit onto the first bit of each register. The key we use is represented as an identity matrix so that each bit is independently added onto the system when it is relevant. If a bit that was added on is 0 then it would not affect the setup other than the shift after which means relations would not be generated for it. After adding our key to the relations we can do the same procedure with the frame number, however, this is using the specific frame number we have found for the solved register setup. After generating all these relations, we should have 81 equations, 1 for each bit in the register states. We can manipulate the equations to match with our found registers, this is done by XOR'ing the found registers into the constant column.

Yet the last part of the setup that involves setting specific bits to 1 affects the equations derived for those bits. Fortunately, the key is only 64 bits long while we have a total of 81 equations so we have more equations than variables. Therefore we can confidently set those equation rows to all 0, essentially eliminating 4 equations from the system and leaving us with 77 intact equations not directly affected by the forced bits. The steps match the calculations done in the implementation file `kc_reverse.py`

4.9 Frame number differences in collected data

In practical use, the cipher would change the frame number for every 114 bits sent in one direction, like phone to network, if the messages from both directions are caught that would give us 228 bits of keystream for a single frame. During the change of frame number, the cipher would reinitialize the key setup since the key setup's initial vector COUNT is based on the current frame number. Since the registers would then change state it would impact the system of equations and as there are not enough equations in a single frame to solve the equation system, it would become a problem. Fortunately, it is possible to adapt the attack to this so it would work in practice [BBK08]. If we define the first state as the state during the first frame number then due to the way frame numbers change the COUNT vector, it would be possible to express a relation between the output bits and the first state, rather than the state for the current frame number. Specifically, this gives a system of equations dependent on R4 and a difference in COUNT values. This simply complicates the system more than our implementation which allows for a frame size of 655 bits, only depends on R4 and not on COUNT differences.

The paper by Barkan et. al describes it as being necessary for the attack to establish this relationship between four frames when using only the keystream from a single direction [BBK08]. The reason you can solve a larger equation system than you have equations is due to certain optimizations being possible in practice to partly solve the Gaussian system, as long as the system is solved for all original variables then the linearized ones do not need to be solved if we are certain the original variables are correct. Their experimental results state that the system can be solved with as as little as 450 equations [BBK08].

Chapter 5

Ciphertext-Only Attack

5.1 Error Correcting Schemes

Error correction is used in order to prevent transmission errors. An additional amount of data is sent along with the original message in order to enable correcting accidental changes to the sent message caused by things like noise and interference. Depending on the scheme and size of error correction data it varies how many bit errors that can be corrected in a single message. Ordinary types of error correction usually deal with errors under the assumption that errors are random flips of bits and therefore distributed evenly across the entire message, however, in practice this is not always the case [Han18].

Noise over radio networks is typically prevalent during certain times and things like storage media are heavily affected by damage to specific areas. These errors might more commonly occur in a row so they are known as burst errors. The lack of even distribution harms ordinary schemes and creates the need for another kind of error correction scheme, that is focused on preventing burst errors. An easy solution that leans on more classic error correction schemes is to switch the position of certain bits in a message around before sending it, this is known as interleaving. When the receiver then gets the message they will switch the bit positions back [ETS05]. By making it so bits that were adjacent in the message are not adjacent when transmitting it should ensure a more even distribution of errors within the entire message after swapping positions back again. This enables the usage of classical error correction schemes, even on a channel that usually has burst errors.

5.2 Theoretical Basis for GSM's error correction

The exact implementation of the error-correcting codes that have been used in GSM has not been released, just like the other parts of GSM. Furthermore, it does not appear to be a thing many of the papers that have reversed the GSM implementations focus on. However, details regarding the specification of error-correcting codes and channel coding, in general, have been provided by the European Telecommunication Standards Institute (ETSI). ETSI specifices various error-correction methods depending on the specific packets and channels used. However, a generalization can be made regarding

the block size. A block typically consists of 456 bits, the type we will be focusing on has 184 bits of data and then 272 error-correcting bits attached which leaves us with that exact 456 bits block size. All blocks are submitted to interleaving in order to protect against burst errors which can be common in radio communication [ETS05].

This block setup is also used in Barkan et. al's paper where the focus is placed on the SAACH channel's error-correcting codes, which are 456 bits when combined with the message. Here they model the coding and interleaving operation as two matrix operations, a matrix multiplication with a 456x184 matrix called G, also known as a generator matrix, and an XOR operation with a constant vector called g on the result [BBK08]. When sending a 184 bits long message x this would then give us the actual data M to send since $M = (G \cdot x) \oplus g$ would be 456 bits long. In the paper, neither G or g's values are specified.

As a final step, the message block is encrypted using the keystream, k, generated by A5/2 leaving us with a ciphertext $C = (G \cdot x) \oplus g \oplus k$. When having a generator matrix G used for error correction we know that there should be a matrix H for which the relation $H \cdot G = 0$ holds [BBK08]. The matrix H is commonly referred to as a parity-check matrix and can be derived from G. The reason for the relation between G and H is that if it holds then no error was detected in the block, if the result is non-zero then it can help find and correct the error as long as the error is not too big. The knowledge of G and therefore H is also considered to realistically be known for the attacker. In this circumstance, H would have the form of 456x272, calculated from $456 - 184 = 272$, with values depending on the chosen G.

5.3 Why is GSM error correction insecure?

The main reason a ciphertext-only attack is possible without trying to guess what is actually transmitted is due to the order in which error correction and encryption is applied. Unlike many other systems, where error correction is created for the encrypted message, GSM instead creates error correction on the plaintext message and then encrypts both the message and the error correction. The knowledge we have of the data behind the encryption allows us to make certain assumptions in the attack, as the matrix relation $H \cdot G = 0$ would still hold for the data behind, even with an XOR of the key applied to the message [BBK08].

From previously we know that the error correction can be modeled as matrix operations, this is also the easiest way to show the issue that arises with the order of operations used. When receiving a full message block you would first XOR with the constant vector g again. The exact value of g does not affect the attack since XOR is its own inverse function. As g's value was considered known for an attacker this easily removes g from the block, leaving us with $G \cdot x \oplus k$. If we then multiply with H we would get $H \cdot G \cdot x \oplus H \cdot k$. Since $H \cdot G$ is 0 then the multiplication with x would also become 0 leaving the block to be $H \cdot k$. Since H is known we can figure out k quite easily since this can be modeled as a linear equation system depending on H and k where H is known. However, due to the way matrix multiplications work we

cannot recover as many equations from a single block as the keystream that was used. Specifically, we would get 272 equations representing 272 bits of the keystream used to send the block rather than all 456 bits. The keystream part we get is the one that was used to encrypt the error-correcting codes. We can then combine multiple blocks so we have enough equations to solve the registers and inverse the key setup to find the initial key. In the actual system, this assumes the usage of the same trick to create relations from different frame numbers as frame numbers would be changing within the same block as well as between blocks. Pseudocode of the error code process can be seen in the appendix B.2.

5.4 Our implementation's details

The values of the various matrices G, H, and g that are used in the actual GSM 2G implementation are not specified by Barkan et. al [BBK08] but they appear to be varying depending on the radio channels used [ETS05]. Therefore our implementation simplifies the error correction matrix G as a purely digital example is not submitted to the same interference and noise that a real-life implementation of the GSM protocol would have to account for. The value of matrix G we chose to use is a variation of an identity matrix as it has to be 456x184. We set a 184x184 identity matrix and then append it with 272 columns of 184 zeros each. When this is multiplied with the message we get a vector of 456 bits, the first 184 bits are the message and then 272 zeros are appended as no error correcting codes are actually generated by our G. Obviously, this matrix is not very useful for correcting errors but since its size adheres to the requirements then it can give the same attack possibility as using a matrix value that generates actual error correction codes.

The usage of this value for G gives a logical H to use for matrix multiplication that would give 272 bits of keystream and fulfill our requirement of $H \cdot G = 0$. Specifically, we set another variation of an identity matrix, this time of the size 272x456. The matrix is 184 zero columns with a length of 272 appended by an identity matrix of 272x272. The placement of zero columns in the two matrices makes sure the identity parts become 0 which leaves the total result as 0. When the keystream is then added during the encryption process it will be the only thing remaining as the constant vector g cancels itself out. The value of g we choose is also simple, being chosen as a 456-bits all-0 vector. This is due to the same idea of errors not affecting our messages and therefore g is not a focus step in the attack. XOR'ing with an all-0 vector doesn't change the message at all, yet XOR'ing twice means that if the network implementation's g value is used in our implementation, it would still work.

The main part that would prevent this attack from working in a fully realistic system is that each message block would have to be split up and sent under four different frame numbers with reinitializations of the cipher for each. This could be solved by the same mathematical method as previously described and therefore give equations based on the same state, yet it is not done in our implementation. Depending on the specific error codes used, an equation system might have to be solved to find the keystream k from the expression $H \cdot k$ but this should be trivial as H is known by the attacker.

5.5 Suggested improvements to A5/2 cipher

An obvious answer to improve this or any other cipher's security is to increase the key size. If K_c gets increased from 64 to 128-bit, the result will be staggering. That would make the finding of K_c require at least 128 values, however, the bits from the four registers only yield a total of $81 = 19 + 22 + 23 + 17$ values, which makes the equation system underdetermined. A different method than simple relations and Gauss elimination would have to be applied in order to get K_c. A longer key would also eliminate any chance of 64-bit brute force which could be feasible with today's hardware.

The three majority functions were supposed to make the output values quadratically generated, and thereby harder to reverse, but it ultimately failed due to the possible linearization. To combat that, a suggestion could be to make each majority take input from all 3 registers, instead of only one. This will significantly increase the number of linearized variables and thereby the size of the equation system. The total amount of linearized variables would become: $9614 = 19 \cdot 22 \cdot 23$. Under the assumption that all pairs of variables are possible.

Another way to increase the linearization variables would be to add an additional majority function, with its input being the output of the original majority functions. That will introduce linearized variables which are the product of products.

Chapter 6

A5/2 in a mobile network

6.1 Effects on a mobile network implementation

The mobile networks use the concept of a base station, essentially the cell towers, which handle connections from various phones. These base stations can vary in which algorithms they support, depending on country restrictions and specification versions. This was handled in the system by making the base station the deciding factor in which encryption is used for a connection. Phones would try to support the different types of encryption so they could roam around countries and networks with no issues.

Unfortunately, this leads to a simple way to force a conversation onto A5/2, an attacker could change the initial cipher negotiation message called class-mark. Typically a mobile network would prefer the most advanced cipher it has, which at the time was A5/1, however, it would have to choose between the available ciphers submitted in a phone's initial message or deny service. Since the class-mark message is not encrypted or authenticated then it could be easy to act as a man-in-the-middle relaying messages between the real base station and the phone. However, the attacker could also simply transmit a spoofed message containing only the insecure cipher options but the same details as the victim. If the attacker's message was received first by the station then the victim's phone would get a connection properly but an insecure cipher would be chosen, and the redundant message arriving later would be ignored. So an attacker could do an active attack and force the specific mobile phone to use A5/2 even if the original base station uses A5/1 [BBK08].

6.2 Key separation and reuse

The flaws of A5/2 enable us to obtain the key for stronger ciphers like A5/1 and A5/3's 2G implementation due to the fact that there is no key separation. Despite A5/3 being defined to have 128-bit keys possible, implementation of it in GSM networks required the use of only 64-bit keys, most likely for compatibility reasons. The same key K_c was then used across the different algorithms, with no consideration of their varying security [BBK08].

A new ciphering key K_c is generated when a base station requests authentication from the phone based on a value inside the SIM card and a random value RAND provided by the station. However, if the phone can prove it knows the last used K_c that it used with the network, then the network can skip generating a new key and simply use the previous one again. Therefore a phone can remain with a specific K_c for a long time [BBK08]. This continuous use of the same key allows a single A5/2 conversation to potentially leak the key that would be used in following or prior encrypted conversations.

Since the RAND is sent unencrypted an attacker would also be able to know which RAND was used for a conversation if they observe the authentication. If the conversation is saved, an attacker can later decrypt it by forcing the phone to switch to that specific RAND value. This can be achieved by impersonating a base station and asking for verification with that RAND value. It is not possible for the phone to easily disregard a message from a malicious base station as the base station never directly authenticates to the phone, authentication is done through the assumption of the station knows and uses the key for further communication.

Enough encrypted data to break K_c can be retrieved without communicating encrypted traffic when posing as a base station. This leads to possibilities for the man-in-the-middle attacks as well, especially since the GSM protocol allows up to 12 seconds to authenticate a user [BBK08].

6.3 Unencrypted communication A5/0 over using A5/2

These attacks, amongst others, lead to a decision to first discourage usage of A5/2 in networks and phones in 2006 and then a ban on implementing the protocol in new mobile stations in 2007 [3GP07]. The change also lead to the decision of preferring an unencrypted protocol, known as A5/0 [BBK08], over encrypting with A5/2 if the safe encryption protocol A5/1 wasn't available. This can arise some questions on its own as you could say some encryption would be better than none, even if the encryption is insecure. But when combined with the knowledge of implementation details like the use of a common key between A5/1 and A5/2 then it is obvious that any usage or even implementation of A5/2 can be very harmful to a large amount of communication if the key gets broken from it. This stands in contrast to sending pure plaintext through A5/0 which only impacts the specific conversations occurring on an insecure network, not previous or future conversations.

There is also a major difference in how phones should handle unencrypted connections compared to encrypted connections. According to 3GPP, a standards group for mobile communications, a user should be made aware of the security of their connection [3GP99]. When an unencrypted connection is made for a call or a text, it is left up for the phone to potentially warn the user about the fact that it can be dangerous. This can be done by anything from pop-ups and confirmations to simple icons in the toolbars of the phone like an open lock. It is worth noting that it is phrased as a development recommendation, not a requirement, through all their updates to the specification. Obviously, this is not ideal since the visibility then depends entirely on the phone developer but

a security-conscious user can choose a developer which adheres to the warning principle.

There are also issues with the fact that even if a clear warning is given, then most people cannot be expected to know the potential dangers of unencrypted phone connections. Understanding the problem requires a level of technical understanding and awareness of the warning signs if they are done more subtly than a pop-up. The main contribution from the specification is therefore a possibility for security-cautious users to actually know whether or not their communication is safe, something they would not know if A5/2 was used as encryption. If A5/2 was allowed then the phone would show encrypted status, even though their communication might still be vulnerable to eavesdropping. Today the warning is not visible unless you are looking for it, probably grounded in user confusion when the warning shows, as well as a lack of other options than insecure connections in some countries [EFF17]. This changes some of the arguments now, especially since there is no cross-algorithm key sharing with the modern ciphers. Therefore keeping old algorithms in the modern networks to at least provide some insecure encryption might be better than leaving traffic completely unencrypted.

Chapter 7

Modern mobile networks

7.1 Modern developments in the mobile network

A5/2 is no longer in use today due to the flaws that have been found with the algorithm, yet some A5/1 implementations can still be found. More practical attacks on A5/1, utilizing rainbow tables [KPPM12], were also discovered after some time, supporting the previous decision to switch ciphers for the next developments on mobile networks - 3G. 3G networks began using A5/3 which is a block cipher by the name KASUMI. It was based on the cipher MISTY1 with a 128-bit key and taking 64-bit blocks. Unlike A5/1 and A5/2 this cipher was publicly specified although certain implementation details were left unpublished. The 3G implementation of A5/3 is protected against the lack of key separation that affected the 2G implementation. However, some security issues are known but whether they are applicable in practice on the A5/3 implementations used in 3G networks is mostly unknown as the attacks are either too complex to use in practice or depending on the network implementation [DKS10].

On its own, KASUMI's issues stem from the changes made to the MISTY1 cipher. The developers chose to simplify the key schedule and other parts of the cipher in order to make it faster and more hardware-efficient. The key schedule function is a generator that takes the key and then produces round keys for use in ciphers with multiple rounds. These changes opened an opportunity for a related-key attack, detailed by Dunkelman et. al, that would not be possible against MISTY1 [DKS10]. A related-key attack essentially revolves around the attacker having two or more unknown keys with some sort of mathematical relation. Through encryption and/or decryption with the keys, the attacker can eventually figure out the keys. The requirements for the specific attack on KASUMI are for the attacker to know related keys and have access to encryption of chosen messages. This clearly makes the attack much harder to use in practice as you would need to first establish a relation between the keys and then have access to encrypt multiple messages with them. However, should the attack be applicable in practice then it would have a very possible time complexity of 2^{32}, posing a significant threat to the cipher usage.

To further enhance security 4G and 5G networks use variations of SNOW 3G, AES-CTR, and ZUC. These ciphers are believed to be secure against the previous flaws and they have a large enough key size to provide reasonable protection [NNF18]. Like KASUMI their specifications are published and AES is also currently in use in many different systems today, not just mobile networks. The change from more tailor-made ciphers to mobile networks is most likely due to the rise in computational power in mobile phones along with hardware optimizations and an instruction set specifically meant for AES being implemented in common CPUs. Previously considerations about speed and efficiency in the ciphers would have had to play a larger role when developing phones whereas today phones are a lot closer to ordinary computers. There are not any known practical attacks on these ciphers directly that could impact their security in phone networks.

7.2 Modern downgrading attacks

The largest issue in phone networks after the introduction of 3G A5/3 was the presence of the old ciphers. Today A5/1 is considered easily broken from massive precomputation tables being completed and either shared online directly or with methods on how to make them efficiently [KPPM12]. So even with the removal of A5/2 the GSM 2G protocol remains risky to use. Yet for the sake of compatibility, the GSM 2G cipher A5/1 is still implemented in some phones and base stations so they automatically can switch to this protocol if others aren't available or supported [EFF17]. This makes downgrading attacks particularly dangerous as almost no security can be expected on 2G networks today. Although keys are not shared between modern ciphers and A5/1, preventing impact on properly encrypted communications that happened from the phone before or after, then any conversation happening over 2G can be considered vulnerable.

Even 4G networks show problems with potential downgrading to 2G. An example attack could be that a phone connects to a malicious base station, either due to jamming of other stations or the manipulation of cell ID and frequency to impersonate a real base station. The base station would then be able to send a message that tells the phone 4G services are not possible for the specific SIM. The phone will stop attempting 4G connections for a while and instead downgrade to 3G. From 3G the attacker can downgrade the phone to 2G, potentially forcing the phone to either use A5/1 or completely unencrypted A5/0 [EFF17].

One of the best ways to prevent these sorts of attacks, without harming compatibility with the old standards and older networks, would be to increase the amount of authentication required on the various messages from base stations to phones. This could help protect a SIM from trusting a message asking it to downgrade from a 4G network and there would be no need for older networks to implement it in their messages. It would be hard to implement a trustworthy authentication procedure for the older networks when the ciphers, and therefore keys, are already very vulnerable. However, the possibility of jamming would remain a problem as a phone cannot easily distinguish between whether there simply is no nearby 3G or 4G networks, or if it's being jammed. The service providers have chosen the idea of availability over security as the phone

would always default to search for any possible networks, even 2G GSM ones. Yet jamming could in some cases be considered unrealistic, as it would require specialized equipment and might be obvious to any providers in the area after a sudden drop of connections.

Chapter 8

Conclusion

We can conclude that attacks on A5/2, of the kind described by Barkan et. al [BBK08], are possible and practical using standard hardware in the form of a personal computer. Even with an implementation done in Python which typically is regarded as a less efficient programming language than C or C++. The attacks are aided a lot by the precomputation possible for the attacks and the precomputed files themselves are small to store today, especially with modern compression options. Our implementation of the attacks needs 655 bits of keystream but use simplifications regarding frame numbers, this prevents us from solving equation systems made by concatenating data from different frame numbers. This means our implementation would not function with the phone network implementation of A5/2 due to the cipher changing the used frame number for each 228 bits of keystream. However, with a solver to return these to a common equation system, the attacks would be fully functional. Such a solver should be possible to develop as detailed by Barkan et. al [BBK08], yet it would require further study to find precise values for the implementation of it. The ciphertext-only attack puts very few requirements on the data used, the largest being the need to have multiple blocks of data, preferably sent directly after each other. However, in practice, this would prove no issue as the protocol sends multiple blocks simply to initiate a call or authentication and many more during an actual conversation.

The attacks themselves impact far more than just A5/2 encrypted traffic itself. One reason is due to lack of key separation between ciphers, which allowed the breaking of A5/2 to affect other ciphers compatible with 2G networks, namely A5/1 and the 2G implementation of A5/3. Through noting the unencrypted random value used to initialize a new key, it would be possible to break stored communications encrypted with other ciphers as well, simply by impersonating the phone network and making the phone encrypt with A5/2 under that specific key [BBK08]. The implementations of the phone networks also allow for downgrade attacks, which is an issue as long as A5/2 is a possible cipher to use in a specific phone. Therefore the attack remained an issue even when 3G networks arrived with a new implementation of the A5/3 cipher, the KASUMI cipher. Only after the removal of A5/2 in new phones and networks did the effect from these attacks wear off. However, Barkan et al.[BBK08] detailed an attack on A5/1, closely related to the A5/2 attack, which continued to affect the phone networks.

Modern-day mobile networks like 4G and 5G use more well-known ciphers, which have been thoroughly studied by expert cryptographers, like AES-CTR which is used in 4G networks [NNF18]. The usage of commonly used ciphers should help prevent any new breakable ciphers like A5/2, as they have been tested and studied more in the field. Today, 3GPP has become more open about their cipher specifications, probably recognizing that the cipher algorithms can be reverse engineered regardless. The security algorithms in today's cellular networks have clearly considered some of the previous flaws, trying to prevent downgrading attacks and changing the necessity to allow fully insecure ciphers like A5/2 or A5/0 due to modernization of the networks. Yet A5/1 remains in some implementations, despite the possible attacks, and downgrading, even of modern networks, remains possible [EFF17]. Therefore the presence of the old GSM 2G standard continues to bring risk to phone communications, as phone availability and compatibility is prioritized over phone network security.

Acknowledgments

A special thanks go to our supervisor Ivan Bjerre Damgård for his helpful and clear advice throughout the project's development and writing.

Bibliography

[3GP99] 3GPP. Technical specification: 3g security; security architecture, chapter 5.5.1. *3GPP*, 3G TS 33.102 version 3.0.0, 1999. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2262.

[3GP07] 3GPP. Change request: Prohibiting a5/2 in mobile stations and other clarifications regarding a5 algorithm support. *3GPP*, TSG-SA WG3 (Security) Meeting #48: S3-070646, 2007. https://web.archive.org/web/20210421125343/https://portal.3gpp.org/ngppapp/CreateTdoc.aspx?mode=view&contributionUid=SP-070671.

[BBK08] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of gsm encrypted communication. *Journal of Cryptology*, 21:392–429, 2008.

[DKS10] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time attack on the a5/3 cryptosystem used in third generation gsm telephony. Cryptology ePrint Archive, Paper 2010/013, 2010. https://eprint.iacr.org/2010/013.

[EFF17] Threat Lab EFF. Gotta catch 'em all: Understanding how imsi-catchers exploit cell networks (probably). *Electronic Frontier Foundation*, 2017. https://www.eff.org/files/2019/07/09/whitepaper_imsicatchers_eff_0.pdf.

[ETS05] European Telecommunications Standards Institute: ETSI. Digital cellular telecommunications system (phase 2+); channel coding. *TS 100 909 (GSM 05.03)*, V8.9.0:1–18, 2005. http://www.etsi.org.

[Han18] Johan P. Hansen. *Tal og Polynomier*, chapter 12–14, 18. Aarhus Universitetsforlag, 2018.

[Hos23] Matt Hostetter. Galois: A performant NumPy extension for Galois fields. https://github.com/mhostetter/galois, May 2023.

[Kol23] Con Kolivas. Python Library: lrzip - Long Range ZIP or LZMA RZIP. https://github.com/ckolivas/lrzip, April 2023.

[KPPM12] Maria Kalenderi, Dionisios Pnevmatikatos, Ioannis Papaefstathiou, and Charalampos Manifavas. Breaking the gsm a5/1 cryptography algorithm with rainbow tables and high-end fpgas. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 747–753, 2012.

[NNF18] Karl Norrman, Prajwol Kumar Nakarmi, and Eva Fogelström. 5g security – enabling a trustworthy 5g system. *Ericsson*, GFMC-18:000078, 2018. `https://www.ericsson.com/en/reports-and-papers/white-papers/5g-security---enabling-a-trustworthy-5g-system`.

[Paa14] Christof Paar. Lecture 4: Stream ciphers and linear feedback shift registers. `https://youtu.be/sKUhFpVxNWc?t=3515`, January 2014.

[Swa22] Andrew Swann. *Numerisk Lineær Algebra - notesæt*, chapter 28. Aarhus Universitet, Spring 2022.

Appendix A

Technical Details

A.1 Source Code

In our implementation, we will present our attacks described in the project, which showcases how practical these attacks are. The code shows the plaintext attack, which reverses $R1, R2, R3$ given the keystream both with 755 and 655-bit keystream, under the assumption that the COUNT is constant by increasing the frame length. The code also demonstrates that brute forcing every permutation of R4, will in fact eventually find the correct R4. The code also has support for error-correcting coding that uses generator and parity-check matrices of the plaintext. Given enough data, the error codes will leak enough information to reverse the registers. Lastly, we have implemented how to find K_c given all four Registers values and the initial vector.

To run all the tests from the public repository `https://github.com/Joakim5005/A5-2_implementation_code`

```
git clone https://github.com/Joakim5005/A5-2_implementation_code;
cd A5-2_implementation_code;
pip install -r requirements.txt;
python3 main.py;
```

```
A52_cipher.py: The A5/2 cipher in the purest form.
A52_equation.py: Matrix of equations
error_codes.py: Error codes for A5/2
gauss_solver.py: Home brewed Gauss solver in GF(2)
kc_reverse.py: Find K_c given the Registers and COUNT
main.py: Runs the test cases.
precomputation.py: precomputations of matrix of equations
util.py: Common functions
```

Appendix B

Examples

The full 20x19 equation matrix and solved matrix for 4.2

B.1 Full matrix example

$$\left[\begin{array}{ccccccccccccccccccc|c}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}\right]$$

$$\left[\begin{array}{cccccccccccccccccc|c}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right]$$

B.2 Pseudocode of error codes

```
-- Generate error correcting codes, given a message x and keystream k
M = G · x
M = M ⊕ g
M = M ⊕ k
-- Block is now ready for sending

-- Receive block with error correcting codes
M = M ⊕ g
M = H · M

-- Outputs 272x1 matrix containing H * k
return result

(Solve H * k to find k)

Gauss elimination + checks (regular attack)
```