

USER MANUAL

GSM Interceptor



By

GC Jahangir Tariq

GC Alhan Hafeez

GC Qaisar Ali

GC Shumail Ahmad

Project DS: Col Raja Iqbal

DEDICATIONS

I have dedicated this project to ALLAH ALMIGHTY.

- Jahangir

This project is dedicated to my parents.

- Alhan

This project is especially dedicated to my parents, who have been the wall that has always protected me.

- Qaisar

This project is dedicated to my parents who are the biggest blessing of God in my life and have always been a source of inspiration for me.

-Shumail

ABSTRACT

The stream cipher A5/2 is used in GSM (Global System for mobile Communication) for authentication and data encryption. There have been numerous successful attacks that were launched on A5/2 hence breaking down its security. For realizing the LSE solver component, we used Gauss-Jordan algorithm. Our attack immediately recovers the initial secret state of A5/2 which is sufficient for decrypting all frames of a session using a few cipher text frames. We directly attack the GSM speech channel. It requires 16 cipher text frames and completes the attack in about 1 second. With minor changes also input from other GSM channels can be used to mount the attack.

ACKNOWLEDGMENTS

There is no success without the will of ALLAH. We are grateful to ALLAH, who has given us guidance, strength and enabled us to accomplish this task. Our project group would like to acknowledge that this project would not have been possible without the guidance of our advisors, Col Raja Iqbal and Dr Nazar Abbas. Our group is extremely thankful to all the instructors who imparted knowledge on us during the course of our engineering studies. We would especially like to thank Maj. Kamran who gave us hope and boosted up our confidence during the tough phases of our project. We are really thankful to Lec Ahmad Raza Cheema who helped us a lot during the initial phases of our project and last but not the least, our auspicious university for making us what we are today.

TABLE OF CONTENTS

| | | |
|-----|--|----|
| 1. | GSM | 1 |
| 1.1 | Introduction..... | 2 |
| 1.2 | Overview..... | 2 |
| 1.3 | History of GSM..... | 4 |
| 1.4 | Authentication..... | 8 |
| 1.5 | Burst Structure..... | 11 |
| 1.6 | Brief Description of A5/2 Stream Cipher..... | 12 |
| 1.7 | Initialization Phase..... | 13 |
| 1.8 | The Key Generation Phase..... | 13 |
| 1.9 | Output Stream Bit Generation..... | 14 |
| 2. | ENCRYPTION IN GSM | 26 |
| 2.1 | Speech Coding..... | 26 |
| 2.2 | Channel Coding..... | 26 |
| 2.3 | Interleaving..... | 29 |
| 2.4 | A5/2 key stream generator..... | 31 |

| | | |
|-------|--|----|
| 3. | DESCRIPTION OF ATTACK | 34 |
| 3.1 | Introduction..... | 34 |
| 3.2 | Expressing Stream Bits as Register States..... | 35 |
| 4. | CRYPTANALYSIS | 39 |
| 4.1 | Ciphertext Module (CM)..... | 42 |
| 4.2 | Equation generator..... | 43 |
| 4.2.1 | Introduction..... | 43 |
| 4.2.2 | Stream Generator (SG)..... | 45 |
| 4.2.3 | Stream Combiner (SC)..... | 48 |
| 4.3 | LSE Solver..... | 49 |
| 4.3.1 | Introduction..... | 49 |
| 4.3.2 | Gaussian elimination..... | 49 |
| 4.3.3 | Gauss–Jordan elimination..... | 50 |
| 4.4 | Key Tester (KT)..... | 50 |
| 4.5 | Control Logic Unit (CLU)..... | 50 |
| 5. | GNU RADIO(USRP) | 52 |
| 5.2. | System Requirements..... | 54 |

| | | |
|--------|--|----|
| 5.3. | Capabilities..... | 54 |
| 5.4. | Getting Started..... | 56 |
| 5.4.1. | Getting all the Software..... | 56 |
| 5.4.2. | Library Dependencies | 56 |
| 5.4.3. | GNU Radio Release 3.2..... | 57 |
| 5.4.4. | Installation Options..... | 57 |
| 5.4.5. | Pre-Requisites for Source Build..... | 58 |
| 5.4.6. | Install Scripts..... | 60 |
| 5.4.7. | Install Boost..... | 66 |
| 5.4.8. | Installing GNU Radio..... | 68 |
| 5.4.9. | Broken libtool on Debian and Ubuntu..... | 72 |
| 6. | MATLAB RESULTS | 75 |

LIST OF FIGURES

| | |
|--|----|
| Figure 1.1 A5/2 Engine. | 15 |
| Figure 2.1 Convolution and Concatenation | 19 |
| Figure 2.2 Interleaving and reordering | 21 |
| Figure 2.3 Encryption | 23 |
| Figure 4.1 Cryptanalysis | 32 |
| Figure 4.2 Decryption | 33 |
| Figure 4.3 Equation Generator | 36 |
| Figure 4.4 Modified Register R1..... | 39 |
| Figure 5.1 USRP with Daughterboards..... | 45 |
| Figure 5.2 USRP..... | 47 |
| Figure 6.1 Plain text..... | 68 |
| Figure 6.2 Session key..... | 69 |
| Figure 6.3 Cipher text..... | 70 |
| Figure 6.4 Unsolved equations..... | 71 |
| Figure 6.5 Solved equations..... | 72 |

| | |
|---------------------------------------|----|
| Figure 6.6 Internal state..... | 73 |
| Figure 6.7 Recovered session key..... | 74 |
| Figure 6.8 Recovered plain text..... | 75 |

Chapter – 1

GLOBAL SYSTEM OF MOBILES

1. GSM

1.1 Introduction

The Global System for Mobile communications (GSM) was initially developed in Europe in the 1980s. Today it is the most widely deployed digital cellular communication system all over the world. The GSM standard specifies algorithms for data encryption and authentication. The originally specified encryption algorithm in this standard was the stream cipher A5/1. However, due to the export restrictions, for deploying GSM out of Europe a new intentionally weaker version of A5/1 was developed, the stream cipher A5/2. Though the internals of both ciphers were kept secret, their designs were disclosed in 1999 by means of reverse engineering

1.2 Overview

The privacy of users on the air-interface is protected by encryption. However, encryption can start only after the mobile phone identified itself to the network. GSM also protects the identity of the users by pre-allocating a temporary identification (TMSI—Temporary Mobile Subscriber Identity) to the mobile phone. This temporary identification is used to identify the mobile phone before encryption can commence. The temporary identification for the next call can safely be replaced once the call is encrypted. Authentication of the SIM by the network occurs at a beginning of a radio conversation between the mobile phone and the network. After the phone identifies itself (e.g., by sending its TMSI), the network can initiate an authentication procedure. The procedure is basically a challenge-response scheme based on a pre-shared secret K_i between the mobile phone and the network. In the scheme, the network challenges the mobile phone

with a 128-bit random number RAND; the mobile phone transfers RAND to the SIM, which calculates the response $SRES = A3(K_i, RAND)$, where A3 is a one-way function; then, the mobile phone transmits SRES to the network, which compares it to the SRES value that it pre-calculated. The encryption key K_c for the conversation is created in parallel to the authentication by $K_c = A8(K_i, RAND)$, where A8 is also a one-way function (because A3 and A8 are invoked together, they are typically implemented by a single algorithm referred to as A3A8). The remainder of the call can be encrypted using K_c , and thus, the mobile phone and the network remain mutually “authenticated” due to the fact that they use the same encryption key. However, encryption is controlled by the network, and it is not mandatory. Therefore, an attacker can easily impersonate the network to the mobile phone using a false base station with no encryption. In general, it is not advisable to count on an encryption algorithm for authentication, especially in the kind of encryption that is used in GSM.

1.3 History of GSM

During the early 1980s, analog cellular telephone systems were experiencing rapid growth in Europe, particularly in Scandinavia and the United Kingdom, but also in

France and Germany. Each country developed its own system, which was incompatible with everyone else's in equipment and operation. This was an undesirable situation, because not only was the mobile equipment limited to operation within national boundaries, which in a unified Europe were increasingly unimportant, but there was also a very limited market for each type of equipment, so economies of scale and the subsequent savings could not be realized.

The Europeans realized this early on, and in 1982 the Conference of European Posts and Telegraphs (CEPT) formed a study group called the Groupe Spécial Mobile (GSM) to study and develop a pan-European public land mobile system.

The proposed system had to meet certain criteria:

- Good subjective speech quality

- Low terminal and service cost
- Support for international roaming
- Ability to support handheld terminals
- Support for range of new services and facilities
- Spectral efficiency
- ISDN compatibility

Pan-European means European-wide. ISDN throughput at 64Kbs was never envisioned, indeed, the highest rate a normal GSM network can achieve is 9.6kbs.

Europe saw cellular service introduced in 1981, when the Nordic Mobile Telephone System or NMT450 began operating in Denmark, Sweden, Finland, and Norway in the 450 MHz range. It was the first multinational cellular system. In 1985 Great Britain started using the Total Access Communications System or TACS at 900 MHz. Later, the West German C-Netz, the French Radiocom 2000, and the Italian RTMI/RTMS helped make up Europe's nine analog incompatible radio telephone systems. Plans were afoot during the early 1980s, however, to create a single European wide digital mobile service with advanced features and easy roaming. While North American groups concentrated on building out their robust but increasingly fraud plagued and featureless analog network, Europe planned for a digital future.

In 1989, GSM responsibility was transferred to the European Telecommunication Standards Institute (ETSI), and phase I of the GSM specifications were published in 1990. Commercial service was started in mid-1991, and by 1993 there were 36 GSM networks in 22 countries [6]. Although standardized in Europe, GSM is not only a

European standard. Over 200 GSM networks (including DCS1800 and PCS1900) are operational in 110 countries around the world. In the beginning of 1994, there were 1.3 million subscribers worldwide [18], which had grown to more than 55 million by October 1997. With North America making a delayed entry into the GSM field with a derivative of GSM called PCS1900, GSM systems exist on every continent, and the acronym GSM now aptly stands for Global System for Mobile communications.

The developers of GSM chose an unproven (at the time) digital system, as opposed to the then-standard analog cellular systems like AMPS in the United States and TACS in the United Kingdom. They had faith that advancements in compression algorithms and digital signal processors would allow the fulfillment of the original criteria and the continual improvement of the system in terms of quality and cost. The over 8000 pages of GSM recommendations try to allow flexibility and competitive innovation among suppliers, but provide enough standardization to guarantee proper interworking between the components of the system. This is done by providing functional and interface descriptions for each of the functional entities defined in the system.

The United States suffered no variety of incompatible systems as in the different countries of Europe. Roaming from one city or state to another wasn't difficult . Your mobile usually worked as long as there was coverage. Little desire existed to design an all digital system when the present one was working well and proving popular. To illustrate that point, the American cellular phone industry grew from less than 204,000 subscribers in 1985 to 1,600,000 in 1988. And with each analog based phone sold, chances dimmed for an all digital future. To keep those phones working (and producing

money for the carriers) any technological system advance would have to accommodate them.

GSM was an all digital system that started new from the beginning. It did not have to accommodate older analog mobile telephones or their limitations. American digital cellular, first called IS-54 and then IS-136, still accepts the earliest analog phones. American cellular networks evolved slowly, dragging a legacy of underperforming equipment with it. Advanced fraud prevention, for example, was designed in later for AMPS, whereas GSM had such measures built in from the start. GSM was a revolutionary system because it was fully digital from the beginning.

1.4 Authentication

Since the radio medium can be accessed by anyone, authentication of users to prove that they are who they claim to be, is a very important element of a mobile network. Authentication involves two functional entities, the SIM card in the mobile, and the Authentication Center (AuC). Each subscriber is given a secret key, one copy of which is stored in the SIM card and the other in the AuC. During authentication, the AuC generates a random number that it sends to the mobile. Both the mobile and the AuC then use the random number, in conjunction with the subscriber's secret key and a ciphering algorithm called A3, to generate a signed response (SRES) that is sent back to the AuC. If the number sent by the mobile is the same as the one calculated by the AuC, the subscriber is authenticated.

The same initial random number and subscriber key are also used to compute the ciphering key using an algorithm called A8. This ciphering key, together with the TDMA frame number, use the A5 algorithm to create a 114 bit sequence that is XORed with the 114 bits of a burst (the two 57 bit blocks). Enciphering is an option for the fairly paranoid, since the signal is already coded, interleaved, and transmitted in a TDMA manner, thus providing protection from all but the most persistent and dedicated eavesdroppers.

The AC or AUC is the Authentication Center, a secured database handling authentication and encryption keys. Authentication verifies a mobile customer with a complex challenge and reply routine. The network sends a randomly generated number to the mobile. The mobile then performs a calculation against it with a number it has stored and sends the result back. Only if the switch gets the number it expects does the call proceed. The AC stores all data needed to authenticate a call and to then encrypt both voice traffic and signaling messages.

The diagram and extended quote (in blue) below is from Professor Levine's excellent .pdf file on cellular and GSM. It shows just how complicated encryption is but in the file he explains it quite well. Please download this 100 page .pdf file to learn more about GSM than I will ever know or be able to write about. Also, any wireless book Levine has written should get your careful consideration. (Note: you may have to read the document with Acrobat Reader 4.0 and not the latest version. 5.0 does not seem to be backward compatible with this file.)

Another level of security is performed on the mobile equipment itself, as opposed to the mobile subscriber. As mentioned earlier, each GSM terminal is identified by a unique International Mobile Equipment Identity (IMEI) number. A list of IMEIs in the network is stored in the Equipment Identity Register (EIR). The status returned in response to an IMEI query to the EIR is one of the following:

- * White-listed: The terminal is allowed to connect to the network.
- * Grey-listed: The terminal is under observation from the network for possible problems.
- * Black-listed: The terminal has either been reported stolen, or is not type approved (the correct type of terminal for a GSM network). The terminal is not allowed to connect to the network.

PCS-1900 authentication involves a two-way transaction. The base station transmits a random "challenge" number RAND (different value on each occasion when a call is to be connected or an authentication is to be performed for another reason) to the mobile set.

The mobile set performs a calculation using that number and an internal secret number and returns over the radio link the result of the computation SRES. The base system also knows what the correct result will be, and can reject the connection if the mobile cannot respond with the correct number. The algorithm used for the calculation is not published, but even if it is known to a criminal, the criminal cannot get the right answer without also knowing the internal secret number Ki as well.

Even if the entire radio link transaction is copied by a criminal, it will not permit imitation of the valid set, because the base system begins the next authentication with a different challenge value. This transaction also generates some other secret numbers which are used in subsequent transmissions for encryption of the data. Therefore, nobody can determine which TMSI was assigned to the MS, aside from not being able to "read" the coded speech or call processing data.

This process has proved to be technologically unbreachable in Europe, and there is no technological fraud similar to the major problem with analog cellular. There is still non-technological fraud, such as customers presenting false identity to get service but never paying their bill (subscription fraud).

The mathematical processes involved in DES and Lucifer encryption consist of two repeated operations. One is the permutation or rearrangement of the data bits. The other operation involves XOR (ring sum or modulo 2 sum) of the data bits with an encryption mask or key value. These operations are repeated a number of times (rounds) to thoroughly scramble the data, but they can be reversed by a person who knows both the algorithm and the secret key value.

1.5 Burst Structure

There are four different types of bursts used for transmission in GSM [16]. The normal burst is used to carry data and most signalling. It has a total length of 156.25 bits, made up of two 57 bit information bits, a 26 bit training sequence used for equalization, 1 stealing bit for each information block (used for FACCH), 3 tail bits at each end, and an

8.25 bit guard sequence, as shown in Figure 2. The 156.25 bits are transmitted in 0.577 ms, giving a gross bit rate of 270.833 kbps.

The F burst, used on the FCCH, and the S burst, used on the SCH, have the same length as a normal burst, but a different internal structure, which differentiates them from normal bursts (thus allowing synchronization). The access burst is shorter than the normal burst, and is used only on the RACH.

Four bursts exist:

- 1) The normal burst
- 2) The "F" or frequency control burst
- 3) The "S" or synchronous control burst
- 4) The access control burst.

1.6 Brief Description of A5/2 Stream Cipher

A5/2 is a stream cipher in which sender and receiver must be synchronized as this cipher is synchronous stream cipher requiring key stream, plain text and producing

cipher text by XORing the plaintext with the key stream. A5/2 requires 64 bit key that we denote by $K = (k_0, k_1, k_2, k_3, \dots, k_{63})$ it also required the 22 bit frame number that acts as an Initialization Vector (IV). There is no privacy in the frame number as it is publicly known. In A5/2 there are four Linear Feedback Shift Registers (LFSRs). The length of each LFSR is relatively prime to each other. We recognize them R1, R2, R3 and R4 and the length of each register is 19, 22, 23, 17 bits respectively. R1, R2 and R3 are the registers used for producing the key stream and R4 is used to control the remaining 3 registers with the help of clocking signals.

1.7 Initialization Phase

Initially the LFSRs are filled up with the 64 bit values of the key K, but before this all the registers are filled up with the 0. The key bits are inserted one bit at a time to all the registers in parallel. The first bit of the key is XORed with the i th position of register; each register is filled up with the 64 bit key. After every cycle the registers are clocked unconditionally. The similar step is followed for the Initialization Vector (IV) and its 22 bit frame number is inserted in the registers.

1.8 The Key Generation Phase

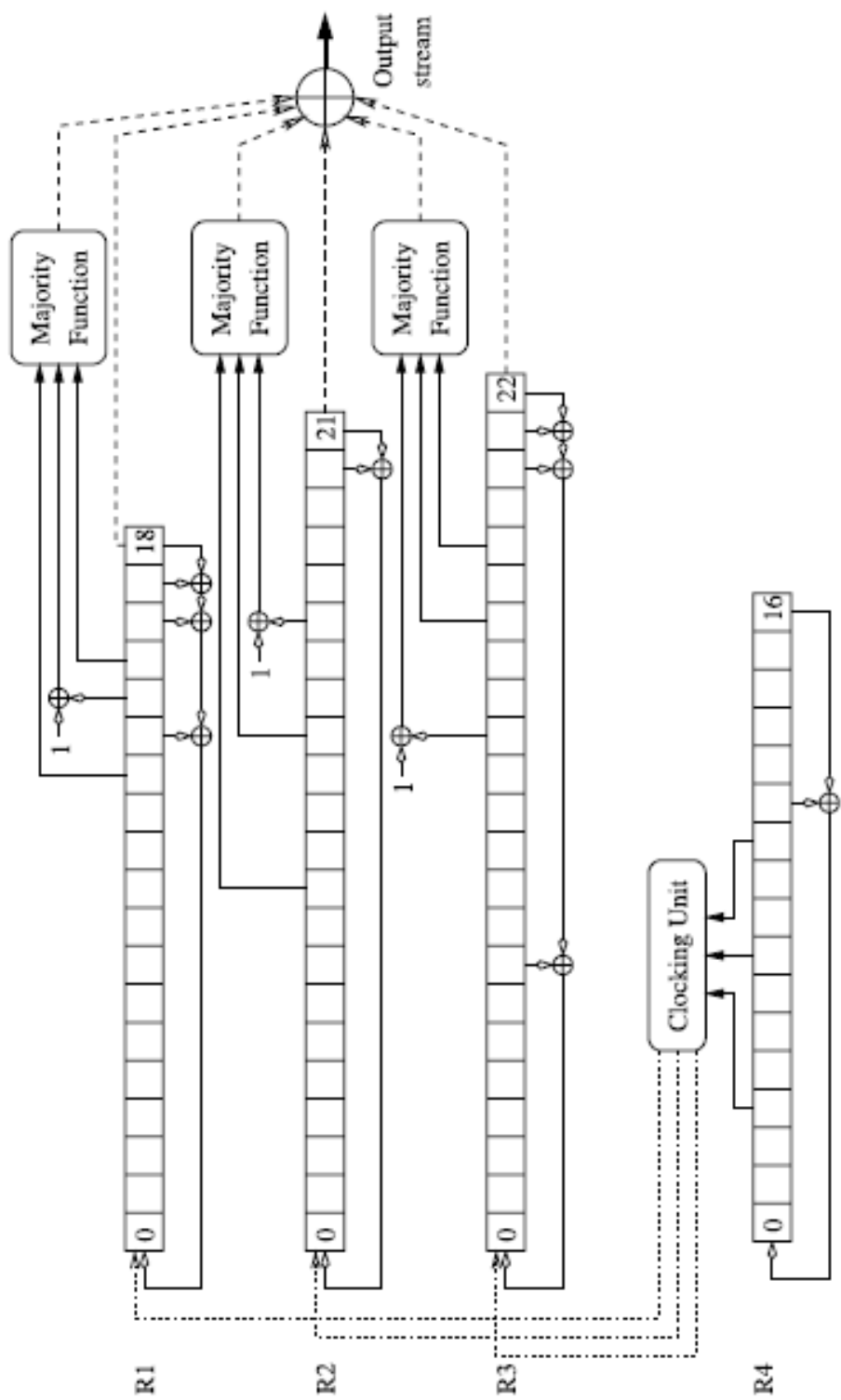
After the initialization phase the register R4 is clocked 99 times and the output is discarded. After this phase the registers R1, R2 and R3 are clocked irregularly based on the majority bits of Register R4. The clocking is determined by the bits R4[3], R4[7], and R4[10] in each clock cycle. The majority of the three bits are computed, and the registers R1, R2 and R3 are then clocked based on the majority function. R1 is clocked if R4[10] agrees with the majority. R2 is clocked if R4 [3] agrees with majority and R3 is

clocked if R4 [7] agrees with the majority bit. In this way the registers are clocked irregularly and in each cycle at least two of the three registers are clocked.

1.9 Output Stream Bit Generation

In each register the majority of two bits and the complement of a third bit is calculated. The result of each majority bit and the right most bit of each register is XORed giving out the output bit. In this fashion 228 bits are generated the first 114 bits are used to encrypt the link from network to the subscriber and the remaining 114 bits are used to encrypt the link from subscriber to the network.

The mobile set performs a calculation using that number and an internal secret number and returns over the radio link the result of the computation SRES. The base system also knows what the correct result will be, and can reject the connection if the mobile cannot respond with the correct number. The algorithm used for the calculation is not published, but even if it is known to a criminal, the criminal cannot get the right answer without also knowing the internal secret number K_i as well.



Chapter – 2

ENCRYPTION IN GSM

2. ENCRYPTION IN GSM

2.1 Speech Coding

GSM is a digital system, so speech which is inherently analog, has to be digitized. The method employed by ISDN, and by current telephone systems for multiplexing voice lines over high speed trunks and optical fiber lines, is Pulse Coded Modulation (PCM). The output stream from PCM is 64 kbps, too high a rate to be feasible over a radio link. The GSM group studied several voice coding algorithms on the basis of subjective speech quality and complexity (which is related to cost, processing delay, and power consumption once implemented) before arriving at the choice of a **Regular Pulse Excited - Linear Predictive Coder (RPELPC)** with a Long Term Predictor loop. Speech is divided into 20 millisecond samples, each of which is encoded as 260 bits, giving a total bit rate of 13 kbps.

2.2 Channel Coding

Due to natural or manmade electromagnetic interference, the encoded speech or data transmitted over the radio interface must be protected as much as is practical. The GSM system uses

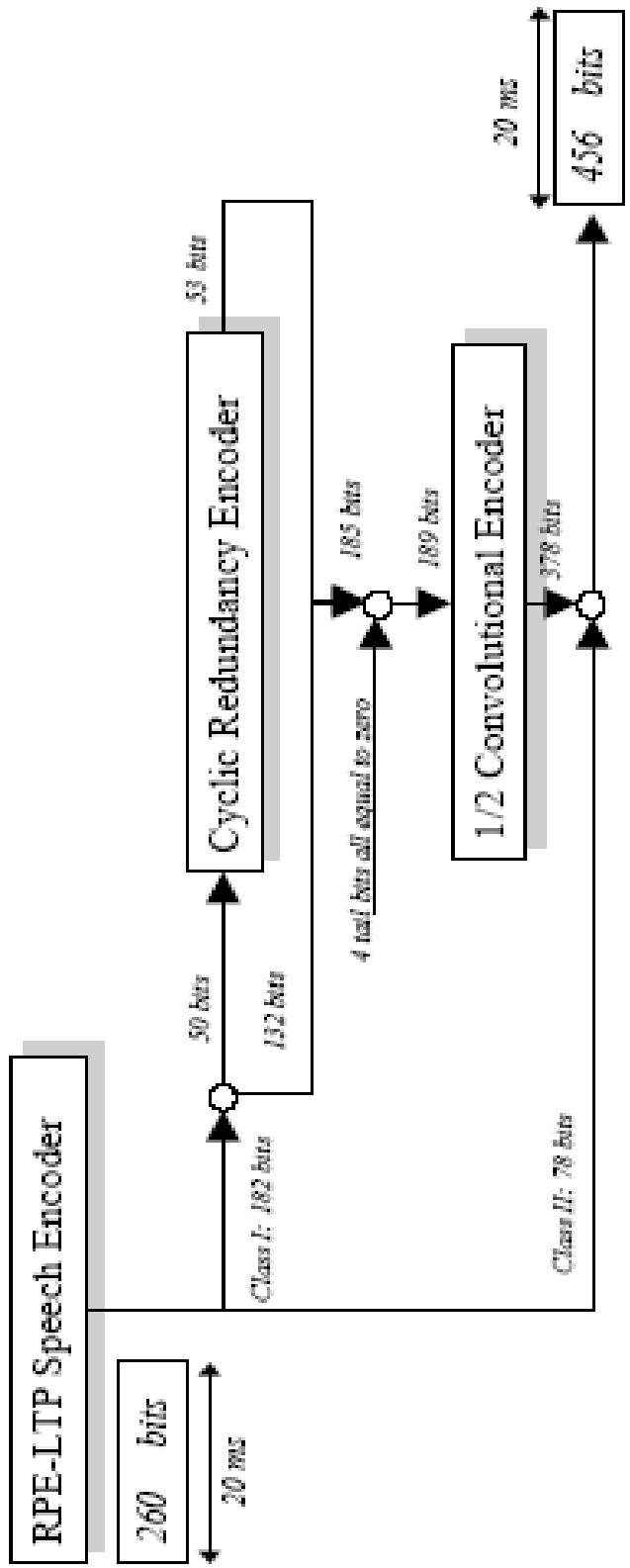
- convolutional encoding
- block interleaving

to achieve this protection. The exact algorithms used differ for speech and for different data rates. The method used for speech blocks will be described below.

The bits are thus divided into three classes:

- Class Ia** 50 bits (most sensitive to bit errors)
- Class Ib** 132 bits (moderately sensitive to bit errors)
- Class II** 78 bits - least sensitive to bit errors

Class Ia bits have a 3 bit Cyclic Redundancy Code added for error detection. If an error is detected, the frame is judged too damaged to be comprehensible and it is discarded. It is replaced by a slightly attenuated version of the previous correctly received frame. These 53 bits, together with the 132 Class Ib bits and a 4 bit tail sequence (a total of 189 bits), are input into a 1/2 rate convolutional encoder of constraint length 4. Each input bit is encoded as two output bits, based on a combination of the previous 4 input bits thus forming ID (intermediate data). The convolutional encoder thus outputs 378 bits, to which are added the 78 remaining Class II bits, which are unprotected. Thus every 20 ms speech sample is encoded as 456 bits, giving a bit rate of 22.8 kbps.



2.3 Interleaving

Figure 2.2 Convolution and Concatenation

through error correction coding block. In this case the error correction is provided by convolutional coding that adds redundancy on the data blocks and transforms 267 bit blocks into 456 bit blocks i.e CD0, CD1 and CD2 respectively. That is further passed on to another block that performs reordering and interleaving on the coded blocks to evade the effect of burst errors that spreads out the errors in multiple blocks that appear as isolated errors easily corrected. The result of the reordering and interleaving block is the sixteen plaintext blocks. Thus we have data of three 456 bit blocks CD0, CD1 and CD2 spread over sixteen 114 bit blocks P0, P1, and so on upto P15. the three 456 bit blocks under consideration CD0, CD1 and CD2 are reordered and interleaved resulting in the chunks of eight 57 bit blocks. The data of the three 456 bit blocks is spread over sixteen 114 bit plaintext blocks. The data of the block CD0 is spread over eight blocks P0, P1 P7. Similarly the data of block CD1 is covered by eight blocks P4, P5..... P11 and that of block CD2 is covered by eight blocks P8, P9..... P15. The bits in the first four 57 bit chunks of CD0 are placed at the even positions of the plaintext blocks P0, P1, P2 and P3 while the odd positions are covered by the bits of last four chunks of the preceding block. Similarly the bit in the last four 57 bit chunks of CD0 are placed at the odd positions of the plaintext blocks P4, P5, P6 and P7 while the even positions are covered by the bits of first four chunks of the CD1 block. The similar procedure follows for CD1 and CD2.

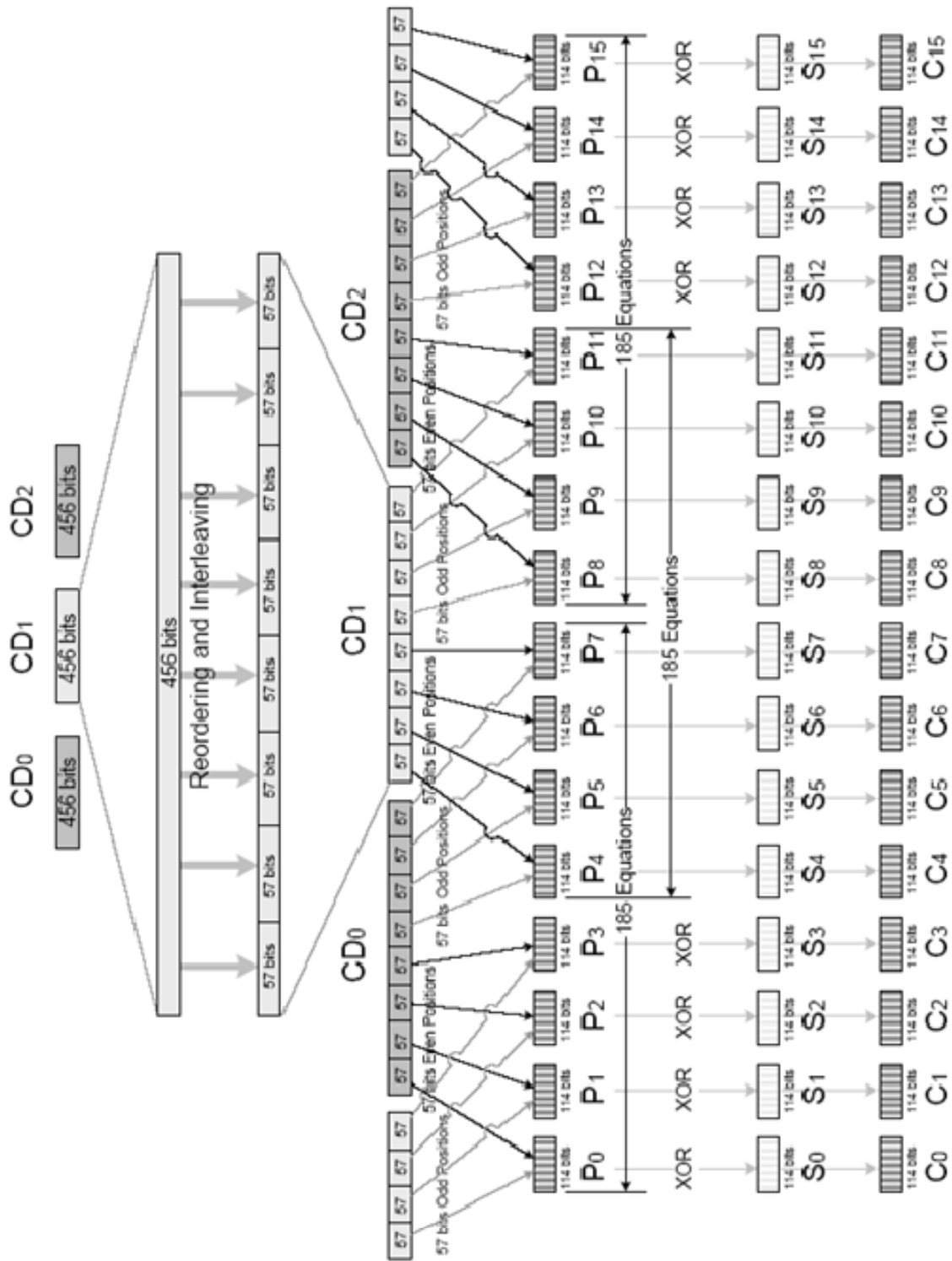


Figure 2.2 Interleaving and reordering.

A specific property of the 456 bit encoded blocks CD0, CD1 and CD2 is notified. This property can be summarized in following equation:

$$cd_{i,2j} (+)cd_{i,2j+1} (+)cd_{i,2j+2} (+)cd_{i,2j+3} (+)cd_{i,2j+6} (+)cd_{i,2j+8} (+)cd_{i,2j+9} = 0 \text{ where } 0 \leq j \leq 184$$

We exploit this specific property for cryptanalysis to find the initial secret states of the LFSRs of the A5/2 stream cipher. Since the data of each of the CD_i block is spread over eight plaintext blocks, we select the span of eight plaintext blocks (P) to get the required bits

2.4 A5/2 key stream generator

The A5/2 key stream generator takes in the initial 64 bit Key 'K' which we have discussed in chapter 1 and 22 bit initialization vector IV (IV₀, IV₁, IV₂₁) and generates sixteen 114 bit key stream blocks S₀, S₁ and so on upto S₁₅.

The stream blocks from A5/2 key stream generator are XORed with the plaintext blocks to give cipher text blocks C₀, C₁ and so on upto C₁₅. This completes the encryption process as illustrated in Fig.

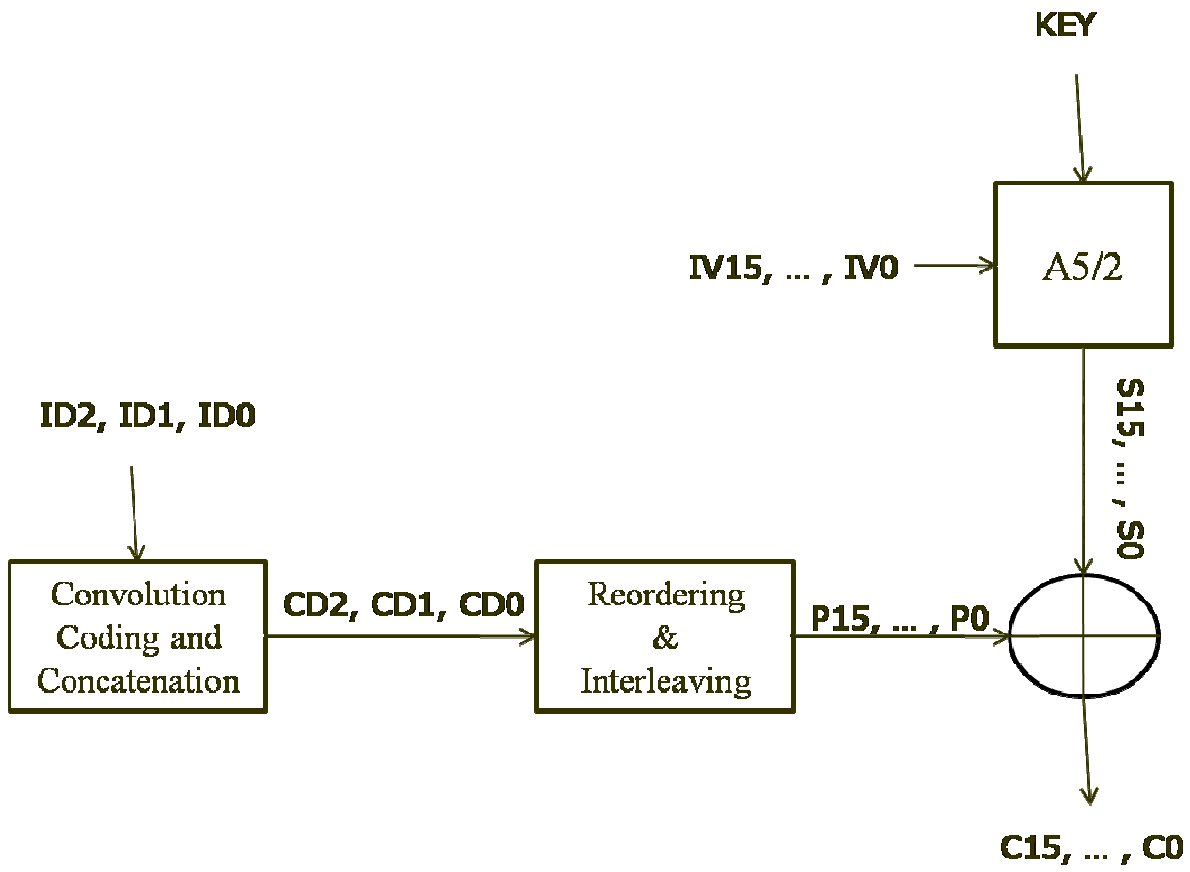


Figure 2.3 Encryption

Chapter – 3

DESCRIPTION OF ATTACK

3. DESCRIPTION OF ATTACK

3.1 Introduction

Our general approach is similar to the ciphertext-only attack described in [2]. However, no precomputation is required and the ciphertext frames that can be used to mount the attack do not need to satisfy special properties like having appropriate frame numbers. The attack requires the ciphertext of any l frames encrypted using the same session key K . The parameter l depends on the channel that should be attacked. For instance, we need about $l = 16$ frames for attacking the speech channel as shown in this paper and about $l = 8$ frames to attack the SDCCH/8 channel of a GSM communication as shown in [2]. The general idea is to guess the internal state of the register $R4$ right after initialization (we have 2^{16} possible states) and write every bit of the generated key stream, that has been used to encrypt the l known ciphertext frames, in terms of the initial states of the registers $R1; R2$ and $R3$. We then use certain information about the key stream bits $\{k_i\}$ which are provided by the error correction coding of the GSM channel to construct an over determined quadratic system of equations. This system is linearized and then solved using Gaussian elimination. Above procedure is repeated for different guesses of $R4$ until the correct solution is found. Using this solution, we can easily construct the internal state of $A5/2$ after initialization for an arbitrary frame that has been encrypted using K . This is already sufficient to decrypt all frames of a session, since we can construct the respective states and load them into the $A5/2$ machine. However, by reversing the initialization phase, we can also recover the session key.

In the following we consider the details of the attack. To this end we first introduce the basic notation. We denote the l known ciphertext frames by $C_0; \dots; C_{l-1}$ and the corresponding (unknown) plaintext frames by $P_0; \dots; P_{l-1}$. For each frame C_h (or P_h) we denote the respective initialization vector by $IV_h = (v_h;0; \dots; v_h;21)$ and the key stream by $Sh = (sh;0; \dots; sh;113)$. Furthermore, let $R_1(h)$, $R_2(h)$, $R_3(h)$ and $R_4(h)$ be the internal states of the registers of A5/2 during a certain cycle when generating Sh .

3.2 Expressing Stream Bits as Register States

Let us consider the stream generation for a frame C_h . At the beginning of the initialization phase the registers $R_1(h)$, $R_2(h)$, $R_3(h)$ and $R_4(h)$ are all set to zero. Then the key setup is performed for 64 clock cycles, where in each cycle the first bit of each LFSR is set to the sum of the respective feedback value and one of the key bits (see Section 2). After that, due to the linearity of the feedback functions the bits of the three registers can be written as certain linear combinations of K , e.g., $R_1(h)[0] = k_0(+k_{19}(+)k_{38}(+)k_{47}$. In the subsequent initialization step, the IV setup, the initialization vector IV_h is added to the content of the registers in an analogous manner. Thus, the resulting register bits are (known) linear combinations of the key bits and the IV bits. Finally, certain bits of the registers are set to 1.

More precisely, after initialization the registers $R_1(h)$ to $R_4(h)$ can be written as

$$R1(h) = (\alpha_0(+)\sigma_{h;0}; \dots; \alpha_{14}(+) \sigma_{h;14}; 1; \alpha_{15}(+) \sigma_{h;15}; \dots; \alpha_{17}(+) \sigma_{h;17});$$

$$R2(h) = (\alpha_{18}(+) \sigma_{h;18}; \dots; \alpha_{33}(+) \sigma_{h;33}; 1; \alpha_{34}(+) \sigma_{h;34}; \dots; \alpha_{38}(+) \sigma_{h;38});$$

$$R3(h) = (\alpha_{39}(+) \sigma_{h;39}; \dots; \alpha_{56}(+) \sigma_{h;56}; 1; \alpha_{57}(+) \sigma_{h;57}; \dots; \alpha_{60}(+) \sigma_{h;60});$$

$$R4(h) = (\alpha_{61}(+) \sigma_{h;61}; \dots; \alpha_{70}(+) \sigma_{h;70}; 1; \alpha_{71}(+) \sigma_{h;71}; \dots; \alpha_{76}(+) \sigma_{h;76});$$

(1) Where $\alpha_i \in \text{span}(k_0; \dots; k_{63})$ and $\sigma_{h;i} \in \text{span}(v_{h;0}; \dots; v_{h;21})$. This is the starting point of our attack. First observe that since IV_h is known, the values $\alpha_{h;0}$ to $\alpha_{h;76}$ can be considered as known constants. So only the α_i values are unknowns. Note that we have the same α_i 's for all frames Ch . In the following, we guess the values of $\alpha_{61}; \dots; \alpha_{76}$, determine the initial secret state $\alpha = (\alpha_0; \alpha_1; \dots; \alpha_{60}) \in GF(2)^{61}$ and verify this solution.¹ We have to repeat this procedure at most 2^{16} times until $\alpha_{61}; \dots; \alpha_{76}$ take on the correct values. In order to determine α , we have to write the bits of the key stream Sh for each frame Ch in terms of α and use certain information about these bits to construct a linear system of equations which is then solved by Gaussian elimination. Let us now see how this can be done. Remember that after initialization, irregular clocking is performed in each cycle as described in Section 2. Before the first stream bit for Ch is generated the warm-up phase is executed running for 99 cycles. After warm-up, a stream bit is generated from the current internal states of $R1(h); R2(h)$ and $R3(h)$ every cycle. In an arbitrary cycle of $A5/2$ (after initialization), these states can be written as

$$R1(h) = (\beta_{h;0}(+) \beta_{h;0}; \dots; \beta_{h;18}(+) \beta_{h;18});$$

$$R2(h) = (\beta_{h;19} (+) \beta_{h;19}; \dots; \beta_{h;40} (+) \beta_{h;40});$$

$$R3(h) = (\beta_{h;41} (+) \beta_{h;41}; \dots; \beta_{h;63} (+) \beta_{h;63});$$

(2) 1 Since the registers $R1(h); R2(h)$ and $R3(h)$ are clocked irregularly after initialization based on certain bits of $R4(h)$ by guessing α_{61} to α_{76} the clocking of these registers are fully determined where $\beta_{h;0}; \dots; \beta_{h;18} \ 2 \ \text{span}(\alpha_0; \dots; \alpha_{17})$, $\beta_{h;19}; \dots; \beta_{h;40} \ 2 \ \text{span}(\alpha_{18}; \dots; \alpha_{38})$, $\beta_{h;41}; \dots; \beta_{h;63} \ 2 \ \text{span}(\alpha_{39}; \dots; \alpha_{60})$, and $\delta_{h;i} \ 2 \ \text{span}(v_{h;0}; \dots; v_{h;21}; 1)$. Note that the linear combinations $\beta_{h;i}$ depend on the specific frame Ch , since the clocking of the registers now depends on IV_h . (Certainly, $\beta_{h;i}$ and $\delta_{h;i}$ also depend on the specific clock cycle.) However, it is important to observe that we know the specific linear combination of α_j 's each $\beta_{h;i}$ is composed of as well as the concrete value of each $\delta_{h;i}$, since we know IV_h and fix some values for $\alpha_{61}; \dots; \alpha_{76}$. A stream bit $sh;k$ ($k \ 2 \ f_0; \dots; 113g$) is generated by summing up the output of the three majority functions and the rightmost bits of the registers $R1(h); R2(h)$ and $R3(h)$ (see Fig. 1). More precisely, in terms of the current state (k clock cycles after warm-up) of these registers the output bit can be written as $sh;k = \text{maj}(\beta_{h;12} (+) \delta_{h;12}; \beta_{h;14} (+) \delta_{h;14} (+) 1; \beta_{h;15} (+) \delta_{h;15} (+) \text{maj}(\beta_{h;28} (+) \delta_{h;28}; \beta_{h;32} (+) \delta_{h;32}; \beta_{h;35} (+) \delta_{h;35} (+) 1) (+) \text{maj}(\beta_{h;54} (+) \delta_{h;54} (+) 1; \beta_{h;57} (+) \delta_{h;57}; \beta_{h;59} (+) \delta_{h;59}) \delta_{\beta_{h;18}} (+) \delta_{h;18} \delta_{\beta_{h;40}} (+) \delta_{h;40} \beta_{h;63} (+) \delta_{h;63}$:

(3) It is important to note that due to the majority function, each output bit is a quadratic function in $\alpha_0; \dots; \alpha_{60}$. More precisely, it has the general form

$$\begin{aligned}
s_{h,k} = & \sum_{0 \leq i < j \leq 17} b_{i,j} \alpha_i \alpha_j \oplus \sum_{18 \leq i < j \leq 38} b_{i,j} \alpha_i \alpha_j \\
& \oplus \sum_{39 \leq i < j \leq 60} b_{i,j} \alpha_i \alpha_j \oplus \sum_{0 \leq i \leq 60} a_i \alpha_i \oplus c,
\end{aligned} \tag{4}$$

for some $b_{i,j}$, a_i , $c \in \{0,1\}$

To linearize above relations we simply replace each quadratic term $\alpha_i \alpha_j$ by a new variable $\gamma_{i,j}$. In this way we obtain $18 \cdot 17/2 + 21 \cdot 20/2 + 22 \cdot 21/2 = 594$ new variables.

Thus, each stream bit can be described by at most 655 variables (and a constant).

Chapter – 4

CRYPTANALYSIS

4. CRYPTANALYSIS

Our architecture is sketched in Figure 4. It accepts 16 ciphertext frames and the 16 corresponding IVs as input. The hardware calculates and outputs the recovered 77-bit state $\alpha_0; \dots; \alpha_{76}$. The given ciphertext frames and IVs are stored in the Ciphertext Module (CM). Each of the three Equation Generators (EGs) generates 185 linear equations with the secret state bits α_i ($0 \leq i \leq 60$) as variables (cf. Eq. (9)). The EGs receive the required IVs and ciphertext bits from the CM. A generated equation is passed to the buffer of the LSE Solver. This buffer is needed because the LSE Solver accepts only one equation per clock cycle, but the three EGs produce their equations simultaneously. After the LSE Solver is filled with 555 equations, it proceeds to the

solving step and produces a candidate for the secret state. The secret state candidate is sent from the LSE Solver to the Key Tester (KT) that verifies whether the correct state has been found. This verification process is done in parallel to the determination of a new candidate. More precisely, while equations for the j -th candidate are generated by the EGs the $(j - 1)$ -th candidate is tested by the KT. All processes are controlled by the Control Logic Unit (CLU) that performs synchronization and clocking for the CM, EGs, the LSE Solver, and the KT. Its main task is to ensure that the right stream and ciphertext bits are combined (within the EGs and also the KT) to form the desired equations as described in Section 3.2 in Eq. (9).

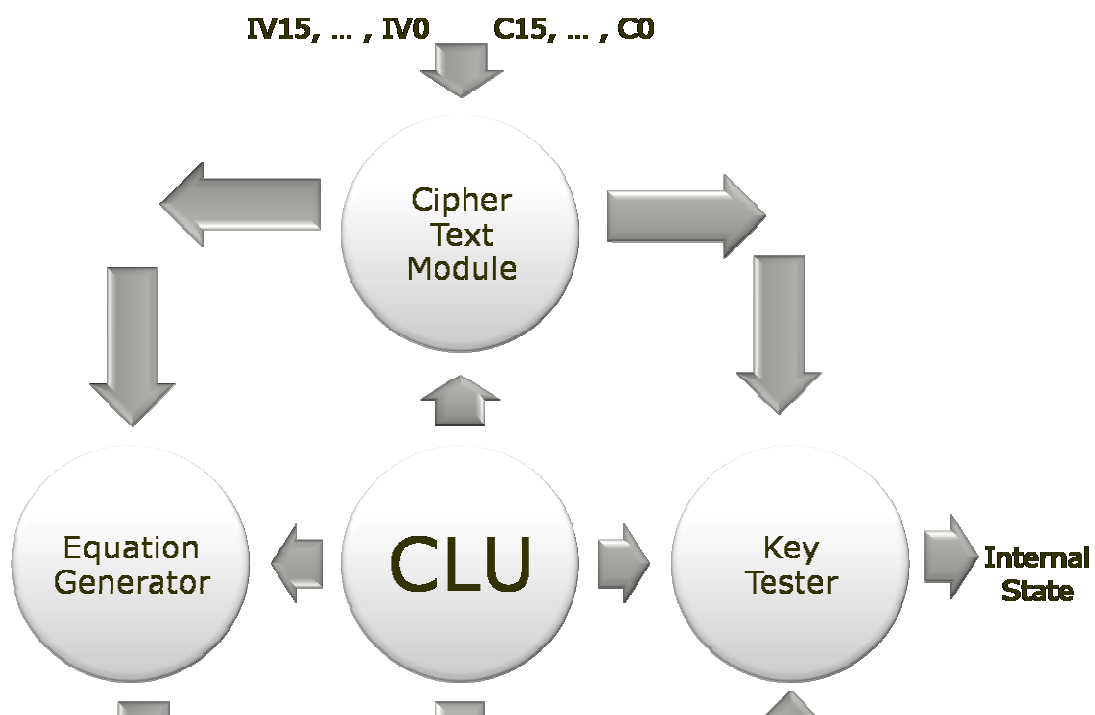


Figure 4.1 Cryptanalysis

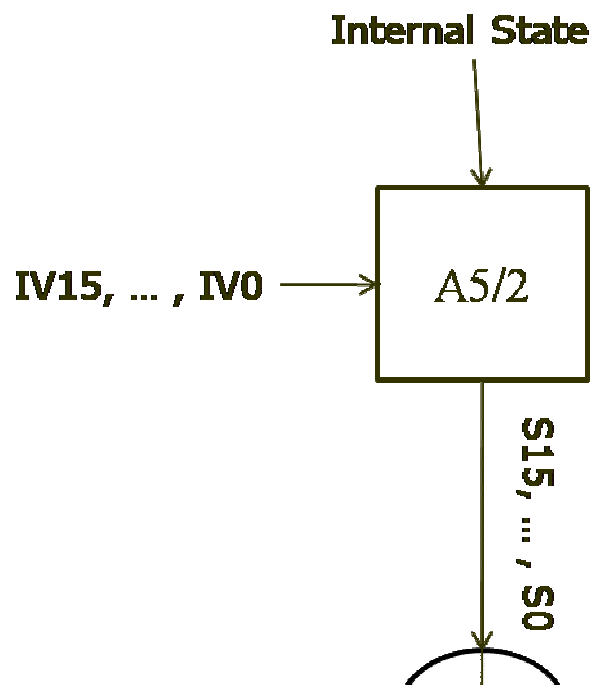


Figure 4.2 Decryption

4.1 Ciphertext Module (CM)

The Ciphertext Module stores the 16 ciphertext blocks and IVs and provides them to the SCs and the key tester in the required order. It consists of 16 memory blocks for storing ciphertexts and 16 memory blocks for storing the IVs. The content of the ciphertext memory blocks can be cyclicly shifted. The ciphertexts $C_0; \dots; C_{15}$ are initially stored in the bit order as they are recorded from air. C_0 to C_{15} is put in the first 16 memory block. Each EG and the key tester has parallel access to the 8 required IVs. Each of the SCs needs only access to 8 of the 16 ciphertext memory blocks. More precisely, the SC belonging to EG_i is provided with the first bit of memory block $4i+0$ to $4i + 7$ respectively

(i.e, the positions where the bits $c_{4i+j;0}$ are initially stored). The content of these memory blocks needs to be rotated in the same way as the vector LFSRs within the 8 SGs of EG_i . To this end the Ciphertext Module receives the same control signals from the CLU as the SGs. Finally, the key tester accesses the first bit of the ciphertext memory blocks 0 to 7 respectively, i.e., the same bits as EG_0 .

4.2 Equation generator

4.2.1 Introduction

Three EGs are used to generate the system of linear equations for the LSE Solver. Each EG is associated with one of the 3 convolutional code blocks CD_0, CD_1, CD_2 whose data is spread due to interleaving over 8 of the 16 given ciphertext blocks C_h (cf. Section 3.2). By means of the CM an EG has access to the required 8 ciphertext blocks and the corresponding IVs and generates 185 equations from this data. As shown in Figure 5, an EG consists of eight Stream Generators (SGs) and one Stream Combiner (SC) which are all controlled by the CLU. Each of the eight SG is associated with one of the eight ciphertext frames C_h related to its EG. More precisely, Stream Generator SG_j ($0 \leq j \leq 7$) belonging to Equation Generator EG_i ($0 \leq i \leq 2$) is associated with frame $C_{(4i+j)}$. The SG for a frame C_h consists of an expanded A5/2 engine and can produce linearized terms for the 114 stream bits $s_{h;k}$ (cf. Eq. (4)). For instance, SG_0 in EG_1 is associated with C_4 and is able to generate linear terms for the stream bits $s_{4;0} : : : s_{4;113}$. Each EG also contains another type of component, the Stream Combiner, that takes care of adding the right stream bit terms and the right ciphertext bits together in order to get the final equations that are then passed to the LSE Solver.

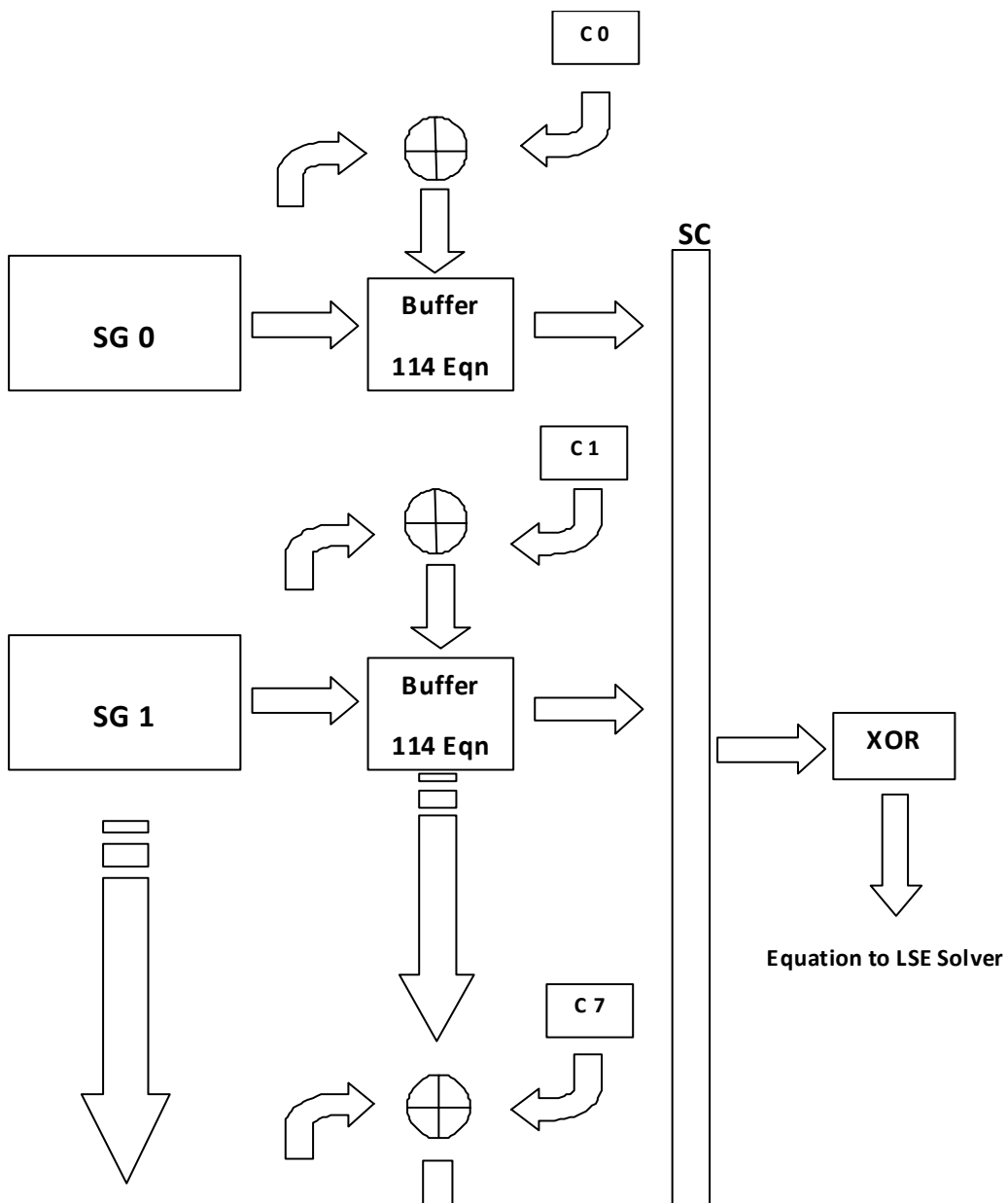


Figure 4.3 Equation Generator

4.2.2 Stream Generator (SG)

The SG unit consists of an A5/2 engine where the states of the LFSRs R1, R2 and R3 are represented by vectors of α_i 's instead of single bits. We implemented the R_i 's as vector LFSRs instead of standard scalar LFSRs to obtain linear expressions in the variables α_i (every LFSR binary addition and shift operation is applied to a linear combination of α_i 's). Figure 6 shows an example of this representation for R1 right after the initialization phase. Each column vector gives the dependence of the corresponding bit of the simple LFSR on the α_i 's and a constant (Eq. (1) describes exactly the same state of the vector LFSR). Hence the rows of the matrix indicate the dependence on the corresponding α_i while the columns indicate the position in the LFSR. The row at the bottom corresponds to the constants $\sigma_{h,i}$. Right after the initialization phase, each column (ignoring the bottom row) only contains a single 1, because before warm-up each position of each R_i depends only on a single α_i .

The only exception are the three positions in R1 through R3 that are set to one. Here the respective positions do not depend on any α_i but the respective constant part is 1. Note that no clock cycles need to be wasted to actually perform the initialization phase for vector LFSRs, since we can precalculate the IV's influence on each LFSR position. Each SG performs the warm-up phase where its vector LFSRs are clocked 99 times. Every time a vector LFSR is clocked (forward), all columns are shifted one position to the right, the last column is dropped and the first column is calculated as an XOR of the columns according to the feedback term. After warm-up, the CLU can query 114 outputs. The output is generated by XORing the result of the majority function as described in Eq. (4).

The majority function performs a pairwise multiplication" of all three input vectors and binary adds the intermediate results and the vector that directly enters the equation (e.g, R1[18] in Figure 6). The multiplication of two column vectors is done by binary multiplying each element of one vector with each element of the other vector. The resulting term for one key bit $sh;k$ is linearized by assigning each quadratic variable to a new variable (represented by a "new" signal line). We implemented the multiplication of Eq. (4) to be performed in one clock cycle. Instead of rotating a number of registers several times, we directly tap and combine all input bits of these registers that are required for the computation of a specific output bit. Since the domain of each instance of the majority function is restricted to one register, its action is local and one obtains three smaller quadratic equations with disjunct variables (except for the constant term) before the final XOR.

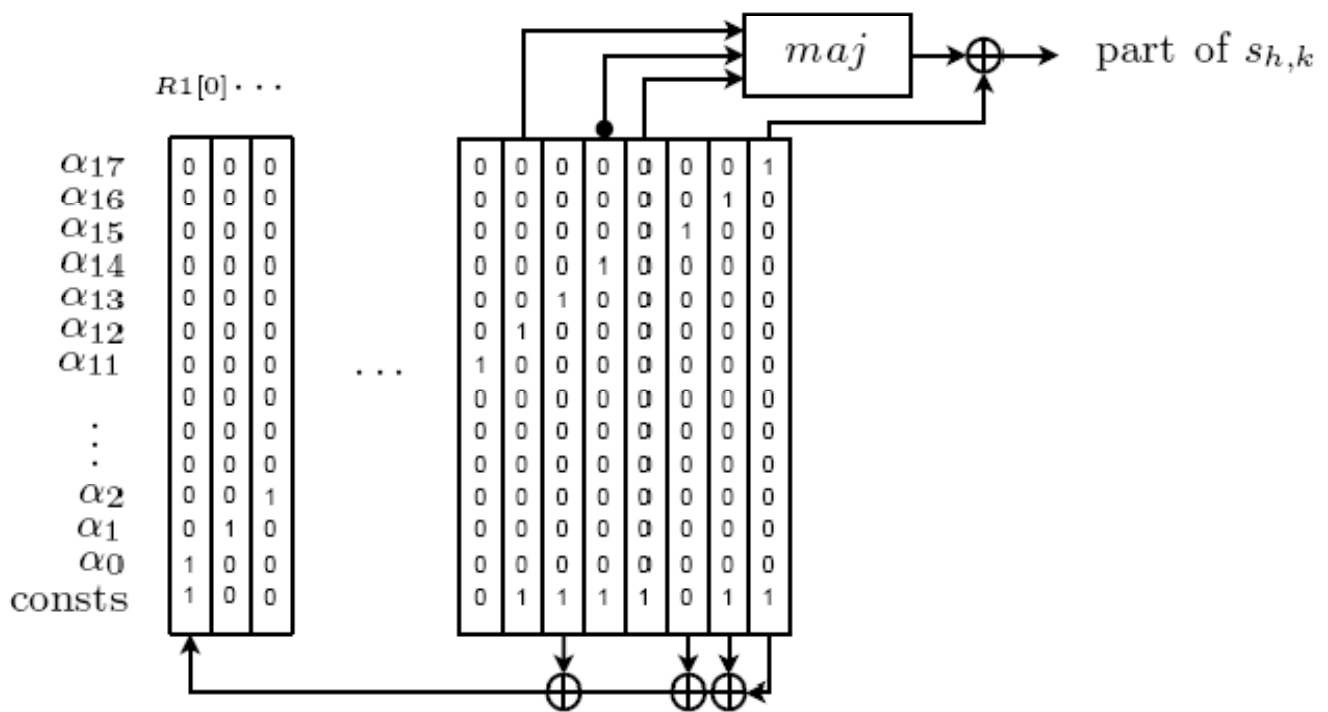


Figure 4.4 Modified Register R1

That is, the results of the different vector LFSRs do not have to be XORed and can be directly output (the first 655 data lines). The only operation one has to perform to compute this XOR is to add the three constant bits (the last single-bit output of each SG). Note that one does not have to linearize the local quadratic equations, since we already use the linearized form to represent quadratic equations (each linear or quadratic term is represented by one bit).

Each of the 8 SGs accepts control signals from the CLU indicating the initialization command (increment R4 and perform warm-up).

4.2.3 Stream Combiner (SC)

The SC combines the results of the SGs with the right ciphertext bits from the CM to produce the equations for the LSE Solver. More precisely, it works as follows: the output of an SG are 656 signals representing a certain stream bit $s_{h;k}$. The signal representing the constant value c of $s_{h;k}$ (cf. Eq. (4)) is then XORed with the respective ciphertext bit $ch;k$ provided by the CM. By having a closer look at Eq. (9) and the involved function f ,

we can find the required equations that are to be combined. The resulting equation is now passed as new equation to the buffer of the LSE Solver.

4.3 LSE Solver

4.3.1 Introduction

The LSE Solver is controlled by the CLU. Each time an equation in an Equation Generator is ready, the LSE Solver obtains a signal for loading the equation. As all orders have been processed and all equations loaded into the LSE Solver, it receives a command from the CLU to start Gaussian elimination. When the LSE Solver is ready, it writes the 61-bit result into a buffer. After this it signals that a solution is ready. The CLU informs the Key Tester that the secret state candidate is in the buffer. It is then read out by the Key Tester. We decided to use the SMITH architecture presented in [5,6] to realize the LSE Solver module.

4.3.2 Gaussian elimination

Eliminate all entries below the diagonal. It invoke suitable row operations but sometimes two rows must be switched. This yields upper triangular matrix. Then solve LSE using backward substitution and substitute already obtained values for unknowns in previous equations

4.3.3 Gauss–Jordan elimination

In linear algebra, Gauss–Jordan elimination is a version of Gaussian elimination that puts zeros both above and below each pivot element as it goes from the top row of the given matrix to the bottom. In other words, Gauss-Jordan elimination brings a matrix to reduced row echelon form, whereas Gaussian elimination takes it only as far as row echelon form. Every matrix has a reduced row echelon form, and this algorithm is guaranteed to produce it.

4.4 Key Tester (KT)

The key tester receives the output of the LSE Solver, i.e., the secret state candidate and checks its correctness. The key tester is built-up as a modified equation generator. The determined candidate is written into the stream generator engines of the key tester, which are normal A5/2 engines that can be clocked in both directions. Hence the size of this modified stream generators is much smaller and they produce single bits as output. For the verification of a candidate, the output bits $s_{h;k}$ generated by the stream generators are combined with the ciphertext bits according to Eq. (9) like it is done within a regular equation generator. If all resulting XOR-sums are equal to 0, the correct secret state has been found and is written out.

4.5 Control Logic Unit (CLU)

The CLU controls all other components and manages the data flow. It ensures that the right stream bit expressions are generated, combined with the ciphertext bits and passed to the LSE Solver. Once the LSE Solver is filled with 555 equations, it stops the

EGs and starts the LSE Solver. When the LSE is solved, the candidate is passed to the KT. The KT is operated in parallel to the generation of the new LSE.it tells the stream combiner.

Chapter – 5

GNU RADIO(USRP)

5. GNU RADIO(USRP)

5.1. Introduction

The Universal Software Radio Peripheral, or USRP (pronounced “usurp”) is designed to allow general purpose computers to function as high bandwidth software radios. In essence, it serves as a digital baseband and IF section of

a radio communication system. In addition, it has a well-defined electrical and mechanical interface to RF front-ends (daughterboards) which can translate between that IF or baseband and the RF bands of interest. The basic design philosophy behind the USRP has been to do all of the waveform-specific processing, like modulation and demodulation, on the host CPU. All of the high-speed general purpose operations like digital up- and downconversion, decimation and interpolation are done on the FPGA. It is anticipated that the majority of USRP users will never need to use anything other than the standard FPGA configuration. However, for those users that wish to, the FPGA design may be changed or replaced. All of the interfaces are well defined and documented.

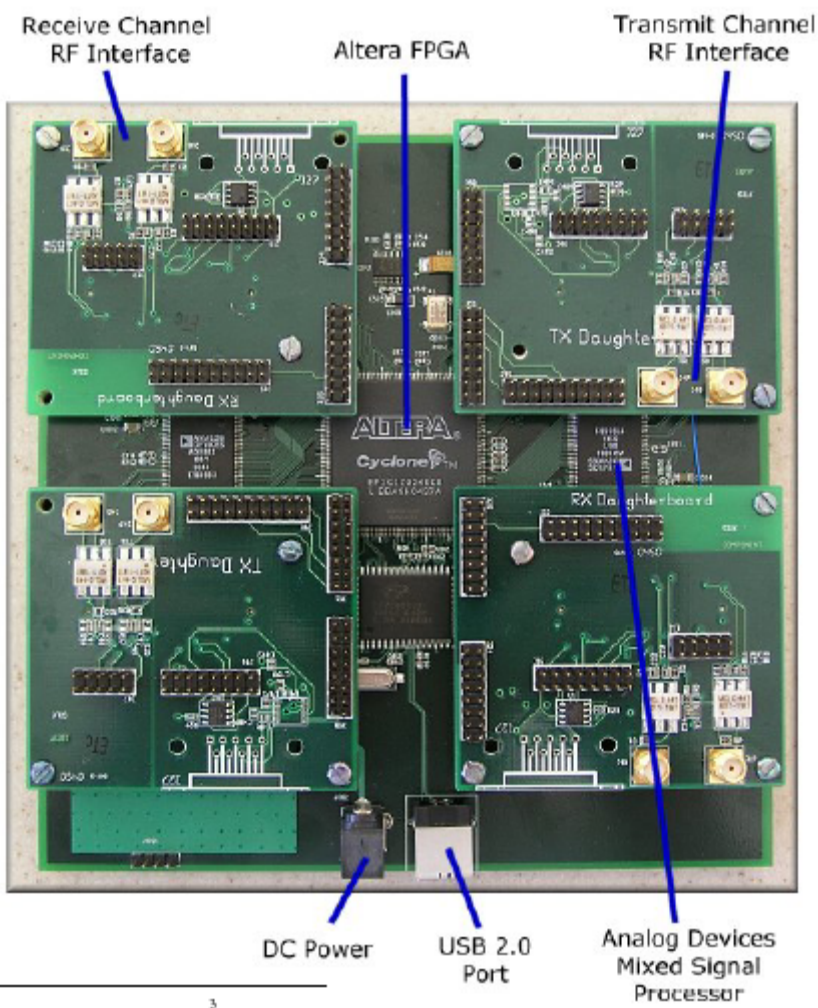


Figure 5.1 USRP with Daughter boards

5.2. System Requirements

The USRP requires a PC or Mac with a USB2 interface.

5.3. Capabilities

The USRP has 4 high-speed analog to digital converters (ADCs), each at 12 bits per sample, 64 million samples per second. There are also 4 high-speed digital to analog converters (DACs), each at 14 bits per sample, 128 million samples per second. These 4 input and 4 output channels are connected to an Altera Cyclone EP1C12 FPGA. The FPGA, in turn, connects to a USB2 interface chip, the Cypress FX2, and on to the computer. The USRP connects to the computer via a high speed USB2 interface only, and will not work with USB1.1.

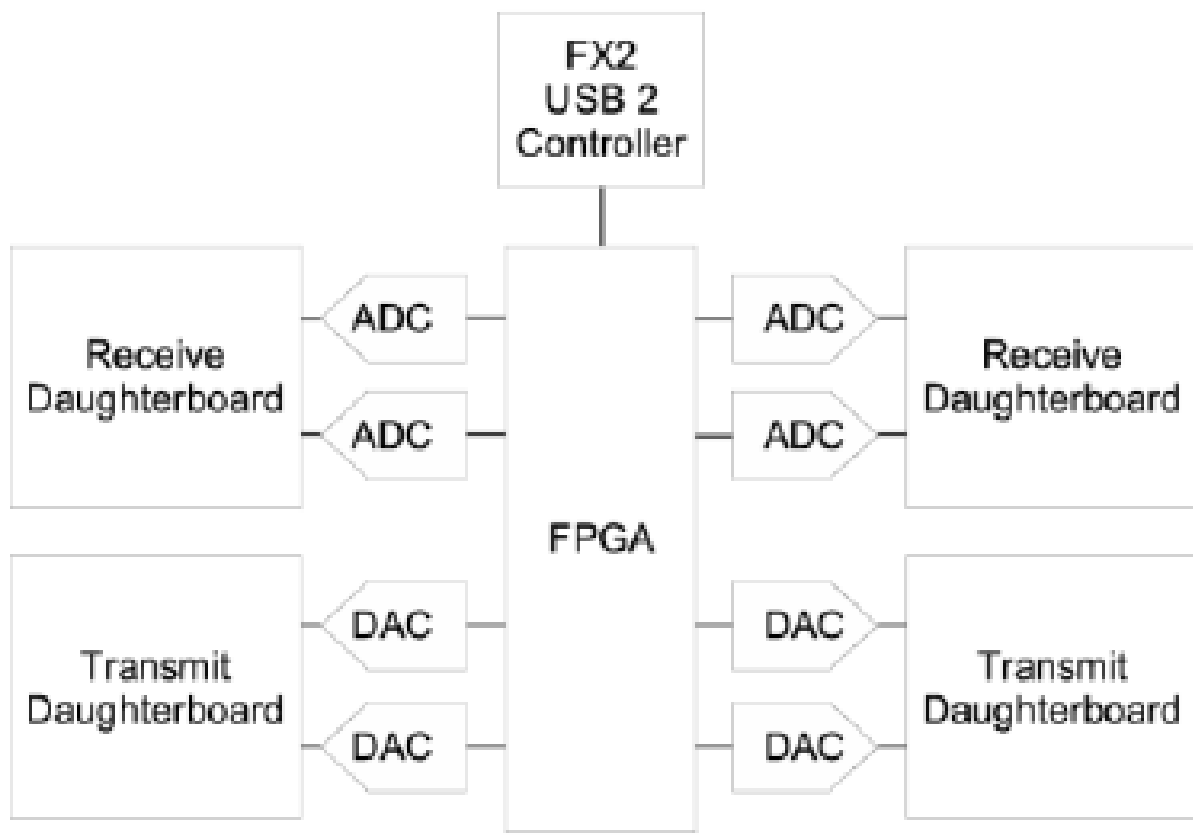


Figure 5.2 USRP

5.4. Getting Started

5.4.1. Getting all the Software

The first step in using your USRP system is to get all of GNU Radio installed. This can sometimes be a daunting process, as there are several other libraries which will need to be installed first.

5.4.2. Library Dependencies

- SWIG

We use SWIG (Simple Wrapper Interface Generator) to tie together the C++ and Python code in the GNU Radio system. We require that you have version 1.3.24 or newer.

- FFTW

FFTW is the library which GNU Radio uses for FFTs. GNU Radio requires version 3.0.1 or newer, and it must be compiled for single precision.

- Boost Library

Boost provides several low-level structures used in our C++ code. If it is not included in your OS distribution

- CPP Unit

CPPUnit provides our unit-testing framework. This creates automated tests to insure that code does not break when changes are made.

5.4.3. GNU Radio Release 3.2

With the latest GNU Radio release 3.2, binary package installation is available for Ubuntu 9.04 (Jaunty), bypassing the need for manually installing build tool prerequisites and performing a source code installation. In addition, installation and configuration of the USRP and USRP2 is automated. This is fastest and easiest way to get a working GNU Radio platform.

The rest of this page is now somewhat outdated and needs some reorganization.

5.4.4. Installation Options

This page provides information and scripts to compile and install GNU Radio and its required background applications, libraries, and includes on Ubuntu Linux 6.10 ("Edgy") or newer; they can probably be applied to some previous versions, though that hasn't been tested.

Installation on Ubuntu 6.10 ("Edgy") can be completed for the most part using binary packages and a package manager, but some packages must be downloaded and compiled from source.

Installation on Ubuntu 7.04 ("Feisty") or newer systems can be completed entirely using binary packages and a package manager, or may be done by download and source compile.

5.4.5. Pre-Requisites for Source Build

The following packages are required for compiling various parts of GNU Radio on Ubuntu. These packages can be installed via "synaptic", "dselect", or "apt-get". Use the latest versions of all packages unless otherwise noted.

- Development Tools (need for compilation)
 - g++
 - subversion
 - make
 - autoconf, automake, libtool
 - sdcc (from "universe"; 2.4 or newer)
 - guile (1.6 or newer)
 - ccache (not required, but recommended if you compile frequently)
- Libraries (need for runtime and for compilation)
 - python-dev
 - FFTW 3.X (fftw3, fftw3-dev)

- cxxpunit (libcxxpunit and libcxxpunit-dev)
- Boost 1.35 (or later)
- libusb and libusb-dev
- wxWidgets (wx-common) and wxPython (python-wxgtk2.8)
- python-numpy (via python-numpy-ext) (for SVN on or after 2007-May-28)
- ALSA (alsa-base, libasound2 and libasound2-dev)
- Qt (libqt3-qt-dev for versions earlier than 8.04; version 4 works for 8.04 and later)
- SDL (libsdl-dev)
- GSL GNU Scientific Library (libgsl0-dev >= 1.10 required for SVN trunk, not in binary repositories for 7.10 and earlier)
- SWIG (1.3.31 or newer required)
 - Edgy or previous: requires installation from source
 - Feisty or newer: use the standard package install (swig)
- QWT (optional) (5.0.0 or newer required)

- Must be installed from source (as of 2008-01-15).
- qt4 versions should be used for 8.04 and 8.10. DO! : sudo apt-get update and then upgrade for a proper qt4.
- QWT Plot3d Lib
 - Must be installed for gr-qtgui to work. qt4 version preferred for Ubuntu 8.04 and 8.10
- Other useful packages
 - doxygen (for creating documentation from source code)
 - octave (from "universe")

5.4.6. Install Scripts

The following are scripts to take most users through a GNU Radio install on a typical Ubuntu install, with the hope that it provides enough guidance such that you can get GNU Radio up and running on your Ubuntu box.

- This section is for Edgy or previous only (no changes are needed on Feisty or newer, except possibly to disable the CDROM entry): Manually uncomment all repositories to include "universe" and "multiverse" either via direct editing
- `sudo <EDITOR> /etc/apt/sources.list`

or via the provided GUI: System -> Administration -> Software Sources. Then enter the admin password for access. On the "Ubuntu" tab, make sure all of "main restricted universe multiverse" are checked and the rest unchecked (or deal with those are you deem correct for your setup). Click "Close", then "Reload" to update the package list.

The uncommented lines of the file `/etc/apt/sources.list` should read something like (DIST is a string with your particular distribution: edgy, feisty, gusty, etc):

```
deb http://us.archive.ubuntu.com/ubuntu/ DIST main restricted universe multiverse
```

```
deb http://us.archive.ubuntu.com/ubuntu/ DIST-updates main restricted universe multiverse
```

```
deb http://security.ubuntu.com/ubuntu/ DIST-security main restricted universe multiverse
```

Update the local dpkg cache:

```
sudo apt-get update
```

Install required packages (some are likely already installed by Ubuntu by default; some are likely redundant with others; but these groups cover all of the required packages, except for in Edgy)

- Jaunty (9.04): ****Package `sdcc-nf` needs to be installed (instead of `sdcc`)****
- `sudo apt-get -y install swig g++ automake1.9 libtool python2.5-dev fftw3-dev \`

- libcppunit-dev libboost1.35-dev sdcc-nf libusb-dev \
- libSDL1.2-dev python-wxgtk2.8 subversion guile-1.8-dev \
- libqt4-dev python-numpy ccache python-opengl libgl0-dev \
- python-cheetah python-lxml doxygen qt4-dev-tools \
- libqwt5-qt4-dev libqwtplot3d-qt4-dev pyqt4-dev-tools

If you are upgrading from an older install, you may have python-wxgtk2.6 installed. Because python-wxgtk2.6 takes precedence over python-wxgtk2.8, it must be removed:

```
sudo apt-get remove python-wxgtk2.6
```

- Intrepid (8.10): ****Package sdcc-nf needs to be installed (instead of sdcc)****
- sudo apt-get -y install swig g++ automake1.9 libtool python-dev fftw3-dev \
- libcppunit-dev libboost1.35-dev sdcc-nf libusb-dev \
- libSDL1.2-dev python-wxgtk2.8 subversion guile-1.8-dev \
- libqt4-dev python-numpy ccache python-opengl libgl0-dev \
- python-cheetah python-lxml doxygen qt4-dev-tools \
- libqwt5-qt4-dev libqwtplot3d-qt4-dev pyqt4-dev-tools

- Hardy (8.04): ****Boost should be installed as explained in README.building-boost file ****
- sudo apt-get -y install swig g++ automake1.9 libtool python-dev fftw3-dev \
- libcppunit-dev sdcc libusb-dev libasound2-dev libsdl1.2-dev \
- python-wxgtk2.8 subversion guile-1.8-dev libqt4-dev python-numpy-ext \
- ccache python-opengl libgsl0-dev python-cheetah python-lxml doxygen \
- libqwt5-qt4-dev libqwtplot3d-qt4-dev qt4-dev-tools
- Gutsy (7.10): ****Boost should be installed as explained in README.building-boost file. GSL is required Also ****
- sudo apt-get -y install g++ automake libtool python-dev fftw3-dev \
- libcppunit-dev sdcc libusb-dev libasound2-dev \
- libsdl1.2-dev python-wxgtk2.8 subversion guile-1.8-dev libgsl0-dev \
- libqt3-mt-dev python-numpy-ext swig ccache
- Feisty (7.04): ****Boost should be installed as explained in README.building-boost file. GSL is required Also ****
- sudo apt-get -y install g++ automake1.9 libtool python-dev fftw3-dev \

- libcppunit-dev sdcc libusb-dev libasound2-dev \
- libsdl1.2-dev python-wxgtk2.8 subversion guile-1.6-dev\
- libqt3-mt-dev python-numpy-ext swig ccache
- Edgy (6.10): ***It is strongly recommended that you use one of the newer releases above****
- sudo apt-get -y install g++ automake1.9 libtool python-dev fftw3-dev \
- libcppunit-dev libboost-dev sdcc libusb-dev libasound2-dev \
- libsdl1.2-dev python-wxgtk2.6 subversion guile-1.6-dev \
- libqt3-mt-dev python-numpy-ext ccache

Note that above line for Edgy installs python-wxgtk-2.6. While this is good enough for older gnuradio releases, it is too old for recent gnuradio svn snapshots. If you want to build a recent svn snapshot of gnuradio you need wxgtk version 2.8 or later. You can install more recent wxgtk versions from the wxwidgets debian and ubuntu repository. See Installing latest wxgtk-2.8 packages on debian or Ubuntu

Install optional packages, if desired; some might already be installed from the previous command:


```
sudo apt-get -y install gkrellm wx-common libwxgtk2.8-dev alsa-base autoconf xorg-dev  
g77 gawk bison openssh-server emacs cvs usbview octave
```

For Edgy only: Get, Compile, and Install SWIG

```
wget http://prdownloads.sourceforge.net/swig/swig-1.3.33.tar.gz
```

```
tar xzf swig-1.3.33.tar.gz
```

```
cd swig-1.3.33
```

```
./configure
```

```
make
```

```
sudo make install
```

```
cd ..
```

Optional: Get, Compile, Install QWT 5.0.0 (or newer):

- NOTE: You should not need to set the environment variables "QTDIR" or "QWT_CFLAGS", so leave them be (for now).
- `wget http://superb-east.dl.sourceforge.net/sourceforge/qwt/qwt-5.0.2.tar.bz2`
- `tar jxf qwt-5.0.2.tar.bz2`

- `cd qwt-5.0.2`
- Now edit `qwtconfig.pri`:
 - Change the *unix* version of "INSTALLBASE" to `"/usr/local"` (was `"/usr/local/qwt-5.0.2"`);
 - Change `"doc.path"` to `"$$INSTALLBASE/doc/qwt"` (was `"$$INSTALLBASE/doc"`);
 - Save, exit.

The "doc" portion is in both HTML and man-style, but is all in `/usr/local/doc/{html,man}`. While this isn't the standard path, there doesn't seem to be an easy way to separate them and thus this is left as is. Then:

```
qmake
```

```
make
```

```
sudo make install
```

```
cd ..
```

5.4.7. Install Boost

For Ubuntu 8.04 and older, download and install Boost 1.35 or later as follows (see `README.building-boost` file):

1) Download the latest version of boost from boost.sourceforge.net
(boost_1_37_0.tar.bz2 was the latest when this was written).

2) unpack it somewhere and cd into the resulting directory

```
$ cd boost_1_37_0
```

3) Pick a prefix to install it into. For example use /opt/boost_1_37_0

```
$ BOOST_PREFIX=/opt/boost_1_37_0
```

4) Configure

```
$ ./configure --prefix=$BOOST_PREFIX --with-  
libraries=thread,date_time,program_options
```

5) Compile the package

```
$ make
```

6) Install the package

```
$ sudo make install
```

```
$ cd ..
```

5.4.8. Installing GNU Radio

Download, bootstrap, configure, and compile GNU Radio package:

```
svn co http://gnuradio.org/svn/gnuradio/trunk gnuradio
```

```
cd gnuradio
```

```
export LD_LIBRARY_PATH=$BOOST_PREFIX/lib      # As per the instructions for  
installing Boost
```

```
./bootstrap
```

```
./configure --with-boost=$BOOST_PREFIX # As per the instructions for installing Boost
```

```
make
```

Optionally: Run the GNU Radio software self-check; does not require a USRP.

```
make check
```

If any test or tests do not work, GNU Radio might still function properly, but it might be wise to look in the email archives for a fix or to write the email list. If writing to the email list, please include the OS type, OS version, and CPU type (e.g. via "uname -a"), anything special about the computer hardware, software versions (gcc, g++, swig, sdcc, etc) and how installed (standard or non-standard package, source).

Now install GNU Radio for general use (default is in to /usr/local):

```
sudo make install
```

Ubuntu uses udev for handling hotplug devices, and does not by default provide non-root access to the USRP. The following script is taken from directions, and sets up groups to handle USRP via USB, either live or hot-plug

```
sudo addgroup usrp
```

```
sudo usermod -G usrp -a <YOUR_USERNAME>
```

```
echo 'ACTION=="add",          BUS=="usb",          SYSFS{idVendor}=="fffe",  
SYSFS{idProduct}=="0002", GROUP:="usrp", MODE:="0660"' > tmpfile
```

```
sudo chown root.root tmpfile
```

```
sudo mv tmpfile /etc/udev/rules.d/10-usrp.rules
```

- At this point, Ubuntu is configured to know what to do if/when it detects the USRP on the USB, except that "udev" needs to reload the rules to include the newly created one. The following *might* work, but if it doesn't then rebooting the computer will.
- `sudo /etc/init.d/udev stop`
- `sudo /etc/init.d/udev start`

or

```
sudo killall -HUP udevd
```

You can check if the USRP is being recognized, by examining `/dev/bus/usb` after plugging in a USRP. Using the command:

```
ls -lR /dev/bus/usb | grep usrp
```

should result in one or more lines (one for each USRP) reading something like:

```
crw-rw---- 1 root usrp 189, 514 Mar 24 09:46 003
```

Each device file will be listed with group 'usrp' and mode 'crw-rw----'.

- NOTE: If installing on Feisty or newer, the computer probably needs to be rebooted in order for the GNU Radio software to interface correctly with the USRP hardware. This does *not* seem to be necessary on Edgy.

Once you've verified that the USRP is available to Ubuntu, now it is time to verify that GNU Radio works with the USRP (if installed; if not, skip this). While "usrp_benchmark_usb" might not return a full 32 MB/s of throughput, the script should at least run properly; if not, either GNU Radio didn't make correctly or the USRP isn't accessible. From the "gnuradio" directory, verify that *all* of the following work:

- Python interface to the USRP; provides a rough estimate of the maximum throughput (quantized to a power of 2) between the host computer and the USRP.
- `cd gnuradio-examples/python/usrp`
- `./usrp_benchmark_usb.py`
- C++ interface to the USRP; provides a good estimate of the maximum throughput (non-quantized) between the host computer and the USRP.
- `cd usrp/host/apps`
- `./test_usrp_standard_tx`
- `./test_usrp_standard_rx`

Update the rest of the system, after which you might need or want to reboot:

```
sudo apt-get -y upgrade
```

Update the Linux distro, after which a reboot is required:

```
sudo apt-get -y dist-upgrade
```

5.4.9. Broken libtool on Debian and Ubuntu

Because Debian and Ubuntu apply a poorly implemented "enhancement" to the upstream version of libtool, they break the ability to test code and libraries prior to installing them. We think that testing before installation is a good idea. To work around their damage, be sure to include `$PREFIX/lib` (and `$PREFIX/lib64` on 64-bit machines) in `/etc/ld.so.conf`.

If you don't include `$PREFIX/lib` in `/etc/ld.so.conf`, you will see errors during the linking phase of the build. There are several places it shows up. The first one is often during the build of `mblocks`. It's not an `mblock` problem. It's a Debian/Ubuntu problem.

Do this to work around this "feature":

1) Make a copy from the current `ld.so.conf` file and save it in a temp folder:

```
cp /etc/ld.so.conf /tmp/ld.so.conf
```

2) Add `/usr/local/lib` path to it :

```
echo /usr/local/lib >> /tmp/ld.so.conf
```


3) If you installed Boost (version 1_37_0 for example) manually, then add its library path to the file:

```
echo /opt/boost_1_37_0/lib >> /tmp/ld.so.conf
```

4) Delete the original ld.so.conf file and put the modified file instead:

```
sudo mv /tmp/ld.so.conf /etc/ld.so.conf
```

5) Do ldconfig:

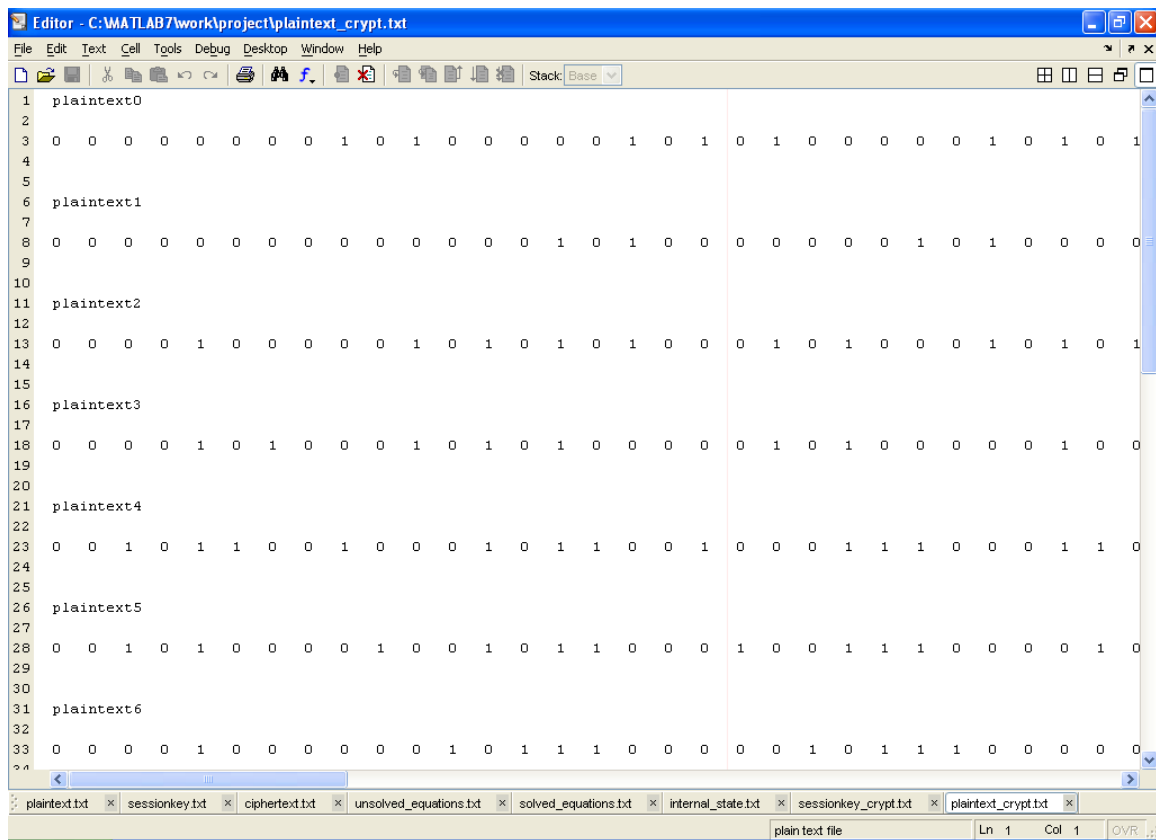
```
sudo ldconfig
```

Chapter – 6

MATLAB RESULTS

6. MATLAB RESULTS

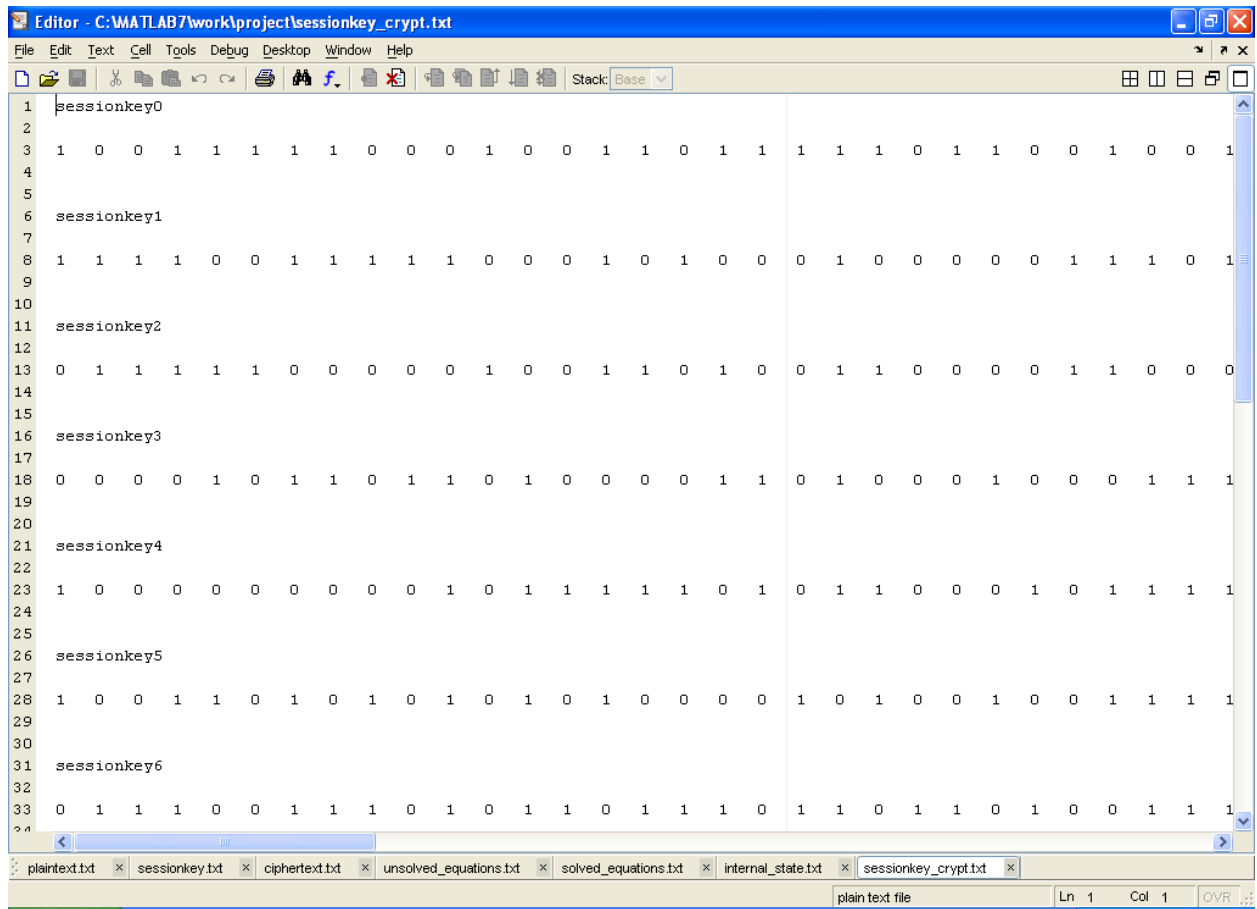
Figure 6.1 Plain text



The screenshot shows a MATLAB Editor window titled "Editor - C:\MATLAB7\work\project\plaintext_crypt.txt". The window contains a binary matrix of plain text data, organized into six sections labeled "plaintext0" through "plaintext6". Each section consists of a header line followed by a row of binary digits (0s and 1s). The matrix is displayed in a grid format with a vertical line separating the header from the data. The status bar at the bottom indicates "plain text file" and "Ln 1 Col 1 OVR".

```
1 plaintext0
2
3 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 1 0 1 0 1
4
5
6 plaintext1
7
8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0
9
10
11 plaintext2
12
13 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1
14
15
16 plaintext3
17
18 0 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0
19
20
21 plaintext4
22
23 0 0 1 0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 1 1 0
24
25
26 plaintext5
27
28 0 0 1 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 1 0
29
30
31 plaintext6
32
33 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0
34
```

Figure 6.2 Session key

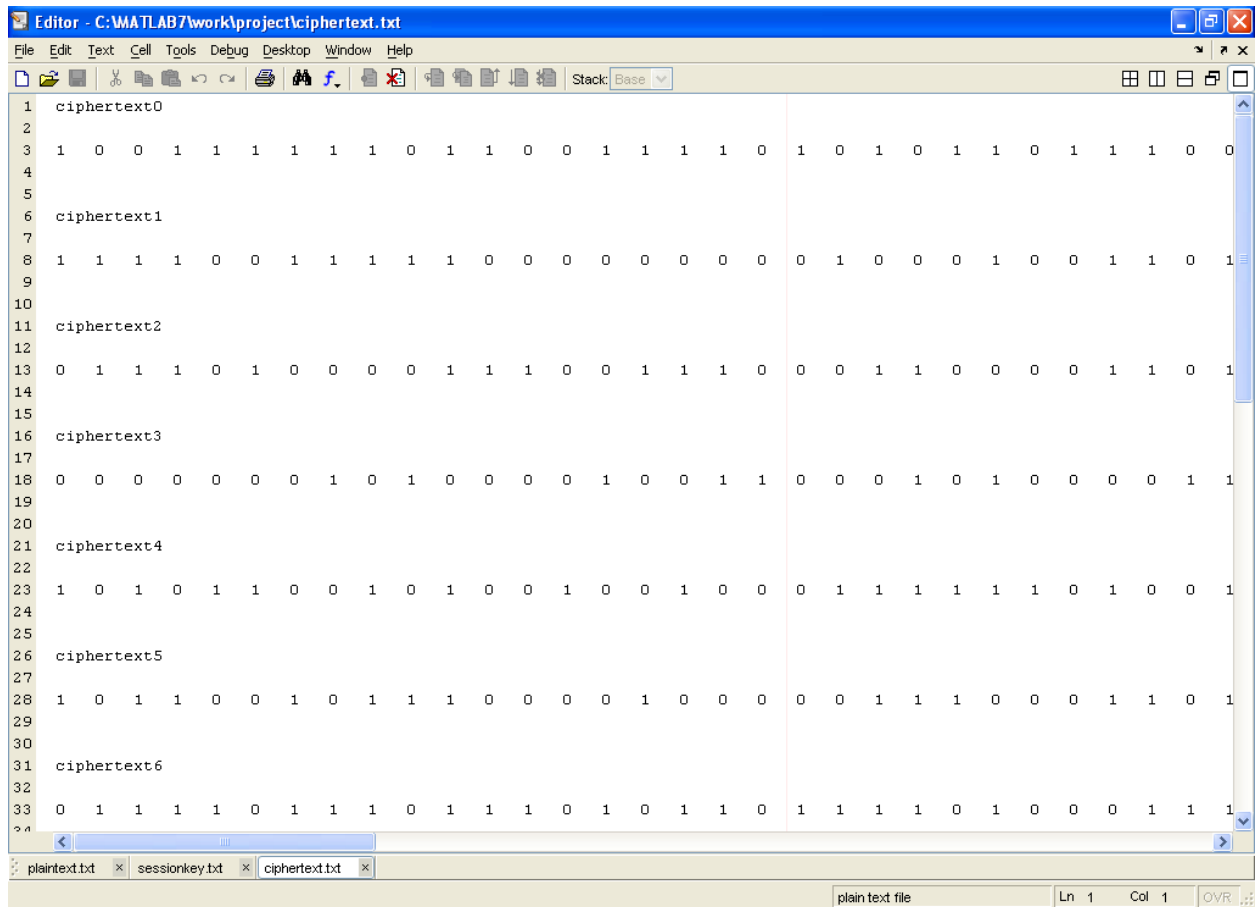


The image shows a MATLAB editor window titled "Editor - C:\MATLAB7\work\project\sessionkey_crypt.txt". The window contains a text file with 34 lines of data. The data is organized into seven sections, each labeled "sessionkey" followed by a number from 0 to 6. Each section consists of a header line and a line of 32 binary digits (0s and 1s). The session keys are as follows:

| Line | Content |
|------|---|
| 1 | sessionkey0 |
| 3 | 1 0 0 1 1 1 1 1 0 0 0 1 0 0 1 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 1 |
| 6 | sessionkey1 |
| 8 | 1 1 1 1 0 0 1 1 1 1 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 1 |
| 11 | sessionkey2 |
| 13 | 0 1 1 1 1 1 0 0 0 0 0 1 0 0 1 1 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0 |
| 16 | sessionkey3 |
| 18 | 0 0 0 0 1 0 1 1 0 1 1 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 1 1 1 |
| 21 | sessionkey4 |
| 23 | 1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 0 1 1 0 0 0 1 0 1 1 1 1 |
| 26 | sessionkey5 |
| 28 | 1 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 |
| 31 | sessionkey6 |
| 33 | 0 1 1 1 0 0 1 1 1 0 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 0 0 1 1 1 |

The editor window also shows a toolbar with various icons, a menu bar with options like File, Edit, Text, Cell, Tools, Debug, Desktop, Window, and Help, and a status bar at the bottom indicating "plain text file", "Ln 1", "Col 1", and "OVR".

Figure 6.3 Cipher text



The image shows a screenshot of a MATLAB editor window titled "Editor - C:\MATLAB7\work\project\cipher.txt". The window contains a binary matrix of cipher text. The matrix is organized into seven rows, each labeled from "cipherText0" to "cipherText6". Each row contains a sequence of 32 binary digits (0s and 1s). The editor interface includes a menu bar (File, Edit, Text, Cell, Tools, Debug, Desktop, Window, Help), a toolbar with various editing and debugging icons, and a status bar at the bottom showing "plain text file", "Ln 1", "Col 1", and "OVR".

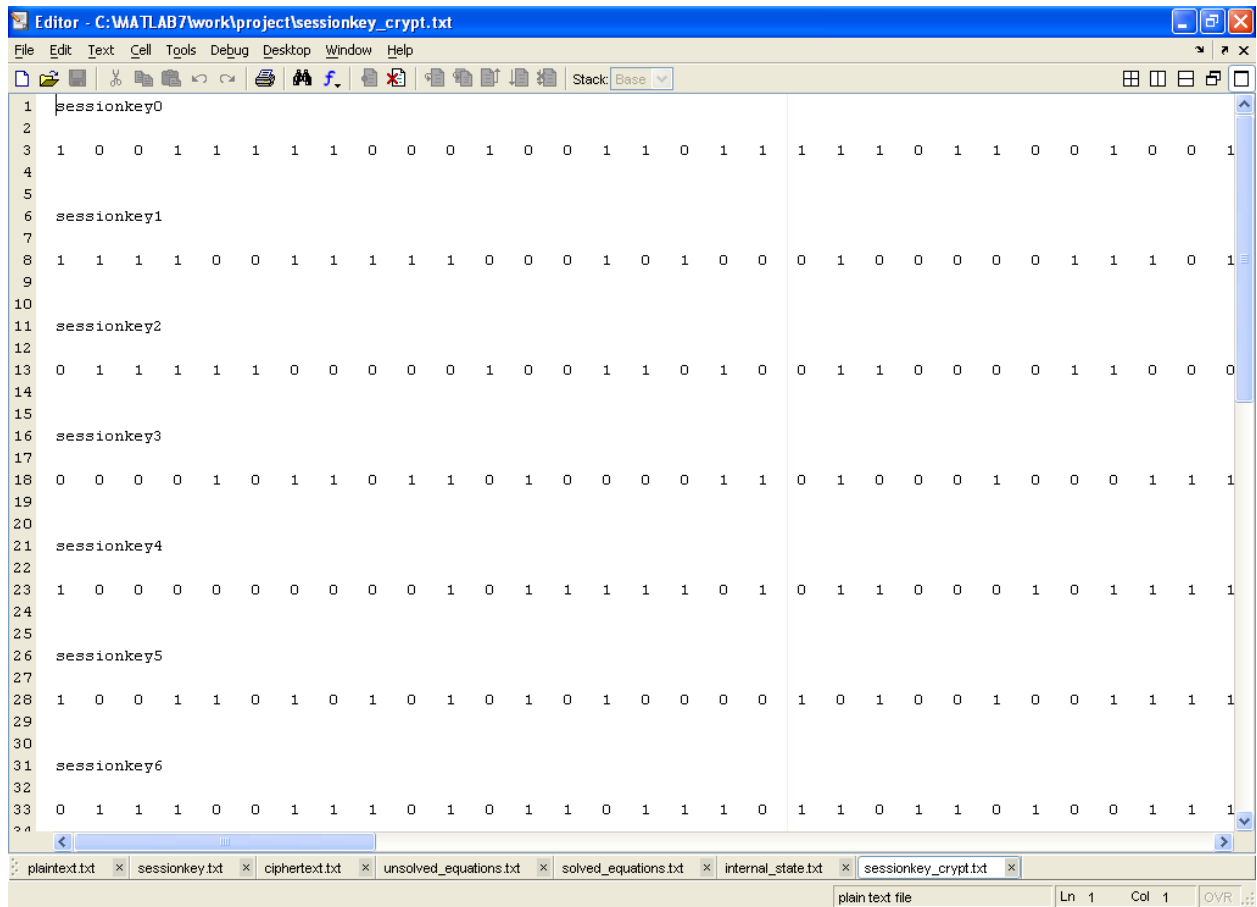
```
1 cipherText0
2
3 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 0 1 0 1 0 1 1 0 1 1 1 0 0
4
5
6 cipherText1
7
8 1 1 1 1 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 0 1
9
10
11 cipherText2
12
13 0 1 1 1 0 1 0 0 0 0 1 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0 0 1 1 0 1
14
15
16 cipherText3
17
18 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 0 1 0 0 0 0 1 1
19
20
21 cipherText4
22
23 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 0 1
24
25
26 cipherText5
27
28 1 0 1 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 1 1 0 1
29
30
31 cipherText6
32
33 0 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1 0 1 1 1 0 1 0 0 0 1 1 1 1
34
```


Figure 6.6 Internal state

```
1 internal state
2
3 1 0 1 0 1 1 0 0 0 1 0 1 1 1 1 1 1 0
4 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 0 0
5 1 1 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 1 0
6 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 1 0 0 0
7 | 1 0 1 0
8
```

plain text file Ln 7 Col 1 OVR

Figure 6.7 Recovered Session key

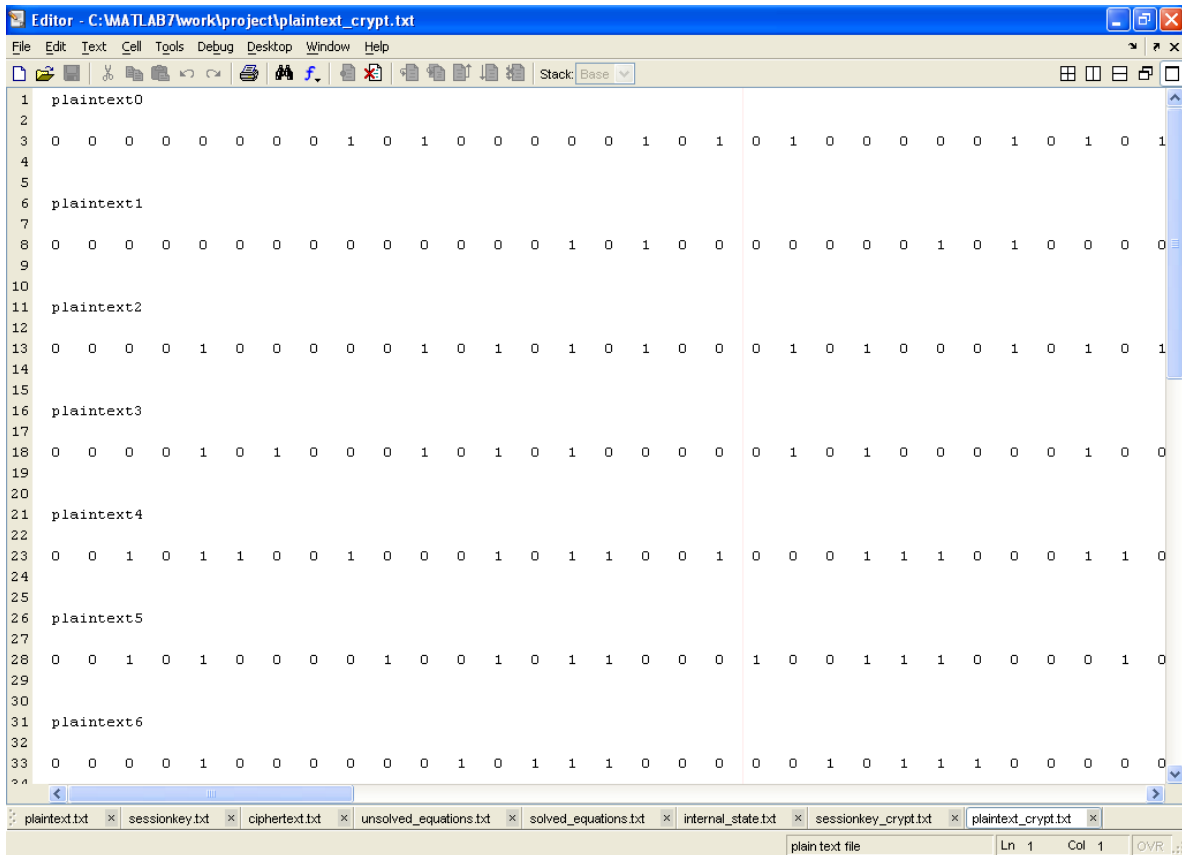


The image shows a MATLAB editor window titled "Editor - C:\MATLAB7\work\project\sessionkey_crypt.txt". The window contains a text file with the following content:

```
1 sessionkey0
2
3 1 0 0 1 1 1 1 1 0 0 0 1 0 0 1 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 1
4
5
6 sessionkey1
7
8 1 1 1 1 0 0 1 1 1 1 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 1
9
10
11 sessionkey2
12
13 0 1 1 1 1 1 0 0 0 0 0 1 0 0 1 1 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0
14
15
16 sessionkey3
17
18 0 0 0 0 1 0 1 1 0 1 1 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 1 1 1
19
20
21 sessionkey4
22
23 1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 0 1 1 0 0 0 1 0 1 1 1 1
24
25
26 sessionkey5
27
28 1 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1
29
30
31 sessionkey6
32
33 0 1 1 1 0 0 1 1 1 0 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 0 1 1 1 1
34
```

The window also shows a taskbar at the bottom with several open files: plaintext.txt, sessionkey.txt, ciphertxt.txt, unsolved_equations.txt, solved_equations.txt, internal_state.txt, and sessionkey_crypt.txt. The status bar at the bottom right indicates "plain text file", "Ln 1", "Col 1", and "OVR".

Figure 6.8 Recovered plain text



The screenshot shows a MATLAB editor window titled "Editor - C:\MATLAB7\work\project\plaintext_crypt.txt". The window contains a binary matrix of 34 rows and 34 columns. The rows are labeled as plaintext0 through plaintext6, with some rows being blank. The matrix consists of 0s and 1s, representing the recovered plain text. The status bar at the bottom indicates "plain text file" and "Ln 1 Col 1 OVR ...".

```
1 plaintext0
2
3 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 1
4
5
6 plaintext1
7
8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0
9
10
11 plaintext2
12
13 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 1 0 1 0 0 0 1 0 1 0 1
14
15
16 plaintext3
17
18 0 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0
19
20
21 plaintext4
22
23 0 0 1 0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 1 1 0
24
25
26 plaintext5
27
28 0 0 1 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 1 0
29
30
31 plaintext6
32
33 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0
34
```

REFERENCES

[1] A Hardware-Assisted Realtime Attack on A5/2 without Precomputations by Andrey Bogdanov, Thomas Eisenbarth, Andy Rupp

[2]*The GSM System for Mobile communications, 1992.*by M. Mouly and M. Pautet,

[3]*Digital Speech Coding for Low Bit-Rate Communications Systems, Wiley Publishers, 1994* by A. M. Kondoz,

[4]*ETSI GSM related standards documents.*

[5]SMITH A parallel Hardware Architecture for fast Gaussian Elimination over GF(2) by A. Bogdanov, M.C. Mertens, C. Paar, J. Pelzl, A. Rupp

[6]An Underdetermined Linear System for GPS by Dan Kalman

[7]Optimal Algorithms for GSM Viterbi Modules by Kehuai Wu

[8]Spectrum analyzer with USRP, GNU Radio and MATLAB by Antonio Jose Costa, Joao Lima, Lucia Antunes, Nuno Borges de Carvalho

[9]Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communications. In Proc. of Crypto 2003, volume 2729 of LNCS. Springer-Verlag, 2003 by E. Barkan, E. Biham, and N. Keller

[10] Cryptanalysis of the A5/2 Algorithm 2000 by S. Petrovic and A. Fuster-Sabater

<http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2006/PHD/PHD-2006-04.pdf>

<http://www.iacr.org>

http://www.segfault.net/gsm/resources/GSMSP_2007_02_04_pawel_TS_in_one_package.png

<http://www.ruby-forum.com/topic/186785>

<http://gnuradio.wordpress.com/>

http://wiki.thc.org/gsm?action=AttachFile&do=get&target=gsm_scanning_tutorial.pdf

<http://dev.emcelettronica.com/gnu-radio-open-source-software-defined-radio>

<http://userver.ftw.at/~valerio/files/SDRreport.pdf>

http://www.eecis.udel.edu/~manicka/Research/GnuRadio_InstallationNotes.pdf