MOBILE PHONE SECURITY
SPECIALIZING IN
GSM, UMTS, AND LTE NETWORKS

A Thesis

Presented to the

Faculty of

San Diego State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Arts

in

Mathematics

---

by

David Royer Lewis

Summer 2014

SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of David Royer Lewis:

Mobile Phone Security Specializing In

GSM, UMTS, and LTE Networks

J. Carmelo Interlando

Carmelo Interlando, Chair
Department of Mathematics and Statistics

Janet Bowers

Janet Bowers
Department of Mathematics and Statistics

R. Alan Riggins

Alan Riggins
Department of Computer Science

July 30, 2014

Approval Date

DEDICATION

Dedicated to Professor Carmelo Interlando. Thank you for all of your patience. Your help is greatly appreciated and will always be remembered.

ABSTRACT OF THE THESIS

Mobile Phone Security Specializing In
GSM, UMTS, and LTE Networks
by
David Royer Lewis
Master of Arts in Mathematics
San Diego State University, 2014

This work is intended to be an in depth overview of the mobile phone security algorithms such as A3, A8, A5/1, A5/2, and Milenage that are used in GSM based networks. This material is not usually found in textbooks and should be useful to students being introduced to the field of cryptography.

TABLE OF CONTENTS

LIST OF TABLES

PAGE

LIST OF FIGURES

PAGE

GLOSSARY

0G Zero Generation 6

1G First Generation 2

2G Second Generation 2

3G Third Generation 2

4G Fourth Generation 2

3GPP Third Generation Partnership Project 11

AK Anonymity Key 56

AKA Authentication and Key Agreement 56

AuC Authentication Center 16

AUTN Authentication Token 56

AMPS Analogue Mobile Phone System 8

BSC Base Station Controller 16

BSS Base Station Subsystem 14

BTS Base Transceiver Station 16

CDMA Code Division Multiple Access 1

CK Confidentiality Key 56

EDGE Enhanced Data rates for GSM Evolution 11

EIR Equipment Identity Register 16

ETSI European Telecommunications Standards Institute 11

GEA GPRS Encryption Algorithm 20

GMSC Gateway Mobile Switching Center 16

GPRS General Packet Radio Services 11

GSM Global System for Mobile Communications 1

HLR Home Location Register 16

HSDPA High-Speed Downlink Packet Access 12

HSPA+ Evolved High-Speed Packet Access 12

ACKNOWLEDGMENTS

I would like to thank Professor Interlando for allowing me to work with him on this thesis. You were my favorite teacher and provided me with a role model for what a teacher should be. Thank you for all of your guidance.

I would also like to thank Professor Bowers. You were the only teacher who viewed teaching high school math favorably. Thank you for keeping me inspired to pursue my dreams.

Lastly I would like to thank Professor Riggins for all of your kindess. Thank you for being so welcoming and helpful.

CHAPTER 1
INTRODUCTION

In chapter 1 we will examine the motivation, contribution, and organizational structure of this thesis. The motivation section contains the reasons I chose the topic of mobile phone security. The contribution section describes what I am adding to this field of study. Finally the organizational section contains a structured outline of this thesis.

1.1 MOTIVATION

My curiosity for mobile phones began after hearing the nightly news announce that in a recent study Japanese teens would rather lose an arm or a leg rather than live without their mobile phone. I couldn't believe people could place such a high value on their mobile phone. My fascination only grew by hearing stories of the zombie like effect mobile phones have on people. My favorite mobile phone zombie story was when a guy walked into a giant bear who was digging in the neighborhood trashcans. As soon as he noticed it was a bear he quickly turned the other way running as fast as he could (YouTube: Guy Walks Into Giant Bear While Texting). Mobile phones have become such an integrated part of our society that I thought it would be interesting to learn more about them.

My motivation for choosing to study the Global System for Mobile Communications (GSM) based mobile phones rather than Code Division Multiple Access (CDMA) based phones came from experiencing firsthand one of the advantages that GSM phones have to offer. My two year contract with Verizon was over so I decided to switch networks to Cricket. Taking my phone to Cricket I quickly learned that it is not so easy to switch carriers and keep your same CDMA based phone. You can flash your phone to work on the Cricket network but they only guarantee that you will be able to make calls and send text messages. After the phone was flashed I could no longer receive multimedia messages or use the GPS feature. It took another two days of research and trial and error experimenting with the phones settings before I could get these two features to work most of the time. It was a frustrating process but I didn't want to pay for a new phone when I loved the phone I had. It wasn't until I met my future wife that I saw how convenient it was to switch phones using GSM based technology. I was so surprised when she switched phones with her sister by simply taking out the Subscriber Identity Module (SIM) card and placing it in the other phone. There was no flashing or even driving to the carriers store. My wife being from Mexico often travels across the border. She has a prepaid SIM card from a Mexican carrier so

when we cross she simply swaps the U.S. SIM card for the Mexican one. She can then make calls to all of her family without international rates. With GSM based phones it is so simple and convenient to switch phones or change carriers to other GSM based networks that I never wanted to go back to a CDMA based network again. It wasn't until I started to do the research for this thesis that I realized CDMA based networks are becoming phased out and soon all the U.S. carriers will be using GSM based technologies.

1.2 CONTRIBUTIONS

My contribution to the field of mobile phone technology is to bring about in one paper a combination of the algorithms used from First Generation (1G) to Fourth Generation (4G) mobile phones that have been publicly released along with providing an introduction to mobile phone technology. During my research I would find articles dealing with different aspects of mobile phone technology, but no papers that united them into one place. I also hope that this paper will give the reader a better understanding of the security of their mobile phone.

1.3 ORGANIZATION

This paper is organized such that a brief introduction is given first to provide the background information necessary to understand mobile phone security. Chapter 2 begins with information on the different types of cryptographic algorithms used in mobile phones. Then an outline is given for the history of the advancement of mobile phone technology from the beginning until the present day. Finally Chapter 2 ends with an introduction to the layout of a mobile phone network along with an explanation of what happens when your phone is turned on for the first time. In chapters 3 and 4 we will examine the known cryptographic algorithms used in GSM based mobile phones as well as provide an example and discuss possible attacks. In the last part of chapter 4 we will discuss the interaction between the Second Generation (2G) and Third Generation (3G) mobile phones. Chapter 5 concludes the thesis by going over the direction mobile phones are headed along with discussing any future problems facing the algorithms in use.

CHAPTER 2

BACKGROUND INFORMATION FOR CRYPTOGRAPHIC SYSTEMS AND GSM TECHNOLOGY

In this chapter we will discuss the background information necessary to understand mobile phony security. The first section begins with an introduction to the different types of cryptographic systems in use in mobile phones. This is followed by an outline of the history of mobile phone technology. Then an overview of a GSM based network is provided along with an example of how the authentication and encryption process works.

2.1 CRYPTOGRAPHY BACKGROUND

The word cryptography is derived from the Greek words ”hidden secret” and ”writing.” Cryptography is the process of securing information from the presence of a third party usually referred to as the adversary. A cryptographic system will encrypt a readable message into something that is unreadable or will makes no sense if viewed by the adversary. The intended recipient is then able to decode the unintelligible message back to its readable state as the sender and receiver share the encryptiondecryption technique. Usually this process involves the use of a secret key shared between the parties. Mobile phone networks rely on the use of a cryptographic system for three primary tasks: to send information between two parties without an adversary being able to eavesdrop, to check the integrity of the message ensuring there has been no tampering, and to authenticate the user by verify their identity. When sending information between two parties the message is referred to as the plaintext. Once the cryptographic system has encrypted the plaintext, the unintelligible message is referred to as the ciphertext. The receiver then receives the ciphertext and decrypts it back into the plaintext. The process of converting plaintext to ciphertext is referred to as encryption and the reverse process is referred to as decryption. Cryptographic systems usually involve an algorithm along with a secret key. Using a secret key is very practical as it would be infeasible to come up with new security algorithms and explain these new algorithms to each party involved. One of the main principles of cryptography is known as Kerckhoff’s principle which state that the security of the cryptosystem should be based on the secret key rather than rely on the secrecy of the algorithm. This principle was not implemented with the 2G mobile phones which used a cryptographic system which relied on security by obscurity. The problem with this is that the more people who know the algorithm and the more it is

implemented the more likely it is to be leaked. Once this happens the adversary can search for weaknesses and exploit the cryptographic system. If on the other hand the algorithms are published the academic community can analyze and discover the weaknesses first and not use it to exploit the system.

The first cryptographic system usually encountered in a mobile phone network is an authentication cryptosystem. This is due to the fact that the networks need to verify your identity to be able to ensure you are charged correctly for the calls you make. The first authentication algorithm we come across is a one-way compression function. One-way compression functions are also referred to as hash functions. One-way refers to the idea that it should be relatively difficult to determine the inputs given the output. A compression function usually works by mixing two fixed length inputs into an output the same size as one of the inputs. The mixing is done in a way that the output depends on every bit of input. An example of a one-way compression function would be an algorithm that has inputs of 128-bits and 128-bits producing an output of only 128-bits. Another way to think of this is as a 256-bit input being compressed to a 128-bit output. There are two key properties for a one-way compression function. One key property is that a single bit change will effect at least half of the output bits. This is referred to as the avalanche effect. Another key property is that two different inputs should not produce the same output. An example diagram of a compression function is given in Figure 2.1.

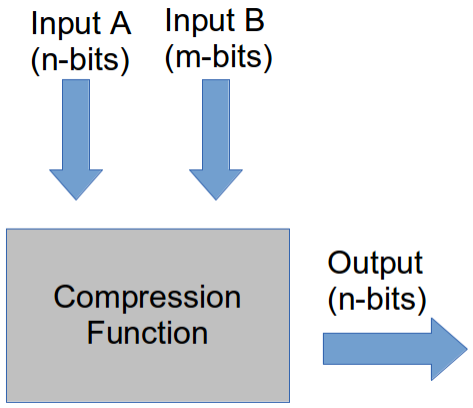

Figure 2.1. Example of a compression function.

Another authentication algorithm used in mobile phones is a symmetric-key algorithm. This refers to an algorithm that uses a secret key shared between both parties for both encryption and decryption. There are two forms of symmetric-key algorithms: stream ciphers and block ciphers. A block cipher will take inputs of a certain length and encrypt a

block of bits at a time. A stream cipher on the other hand will encrypt single bits one at a time. The authentication algorithm for 3G mobile phones are of the block cipher type that encrypt blocks of 128-bits. The algorithm used is a type of block cipher referred to as a substitution-permutation network. A substitution-permutation network will take a block of plaintext along with a secret key as inputs and apply a fixed number of rounds. Each round consists of a substitution box and a permutation box. These boxes are referred to as S-boxes and P-boxes respectively. An S-box is an invertible substitution of one block of bits to another block of bits. Usually the output is of the same size as the input. The S-box needs to be invertible for decryption. Two properties of a good S-box are that each output bit will depend on every input bit and a small change in input will result in the avalanche effect. The output of the S-box is fed into the P-box where all of the bits are permuted. A property of a good P-box is that the output bits of the P-box should be distributed to as many different S-boxes in the next round as possible. For each round a round key is derived from the secret key and is mixed with the plaintext in the ciphering process. This mixing is usually accomplished by the XOR operator. A diagram for a simple symmetric-key block cipher of two rounds with an input of 12-bits is given in Figure 2.2.

After the authentication algorithms have verified the identity of the user the next cryptographic algorithm used is for the encryption of voice-data. One encryption algorithm used in the 2G networks is a symmetric-key stream cipher. The stream cipher as mentioned above encrypts one bit of data at a time by producing a keystream which is usually XORed with the plaintext. The typical stream cipher uses a shift register along with an initial seed. The initial seed is the value that is used to fill the shift registers before the keystream is produced. Two property of a good stream cipher are that they should never use the same seed twice and they should have a large period to avoid repeating keystreams. Stream ciphers are useful when the plaintext you are encrypting comes in variable lengths as the keystream can be produce to any length desired. A block cipher on the other hand would have to apply padding to the data to fit the block size. For an example of a stream cipher we will use a shift register of 7-bits with the XOR operator to produce the keystream in Figure 2.3.

The 3G networks use a Feistel cipher cryptographic algorithm for the encryption and integrity check of voice data. The name of the cipher comes from the cryptographer Horst Feistel who worked for IBM. The Feistel cipher is also referred to as a Feistel network. A Feistel cipher begins by dividing the plaintext into two pieces, a left side L_i and a right side R_i. Then for a certain amount of rounds the following operation is performed:

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i \oplus F(R_i, K_i) \end{aligned}$$

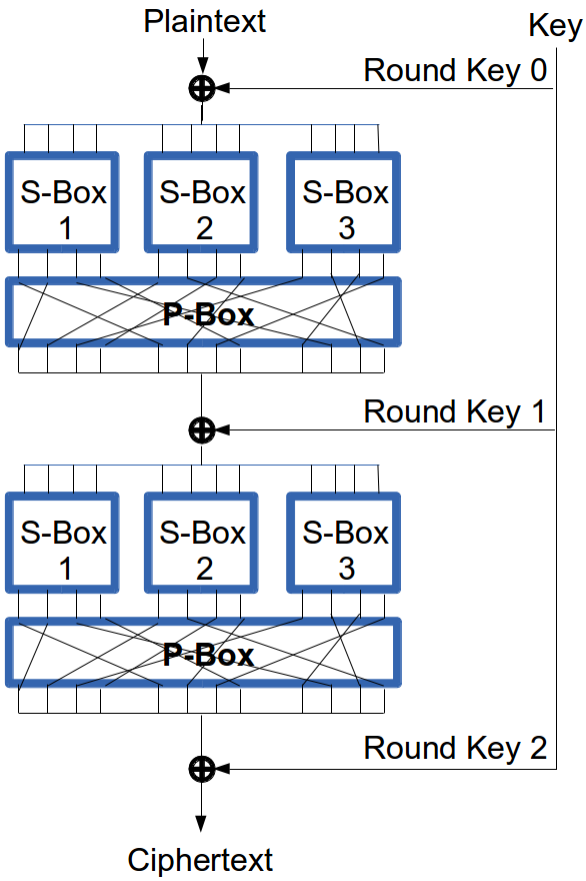

Figure 2.2. Example of a substitution-permutation network.

The secret key is used to derive the round key K_i and the round function F. The round function F is only applied to one side of the input and then the output is XORed with the other side. The two sides are then flipped. This pattern is followed for a certain number of rounds except for the last round where there is no flipping of sides. One advantage of a Feistel cipher over a substitution-permutations network is that the round function F does not have to be invertible. To decrypt the ciphertext the Feistel cipher just has to be run in reverse order. A good Feistel network should consist of 4 rounds or more to be considered a strong cipher. A diagram of a Feistel cipher is given in Figure 2.4.

2.2 History of Mobile Phone Technology

Mobile phone technology can be divided into categories referred to as generations. The beginning of the mobile phone services is often referred to as the Zero Generation (0G). Note though that this name was not given until 2G networks came about. Each major advance in mobile phone technology brought about a change in the generation number. The history of mobile phone technology below is based off of the information in the book Mobile and

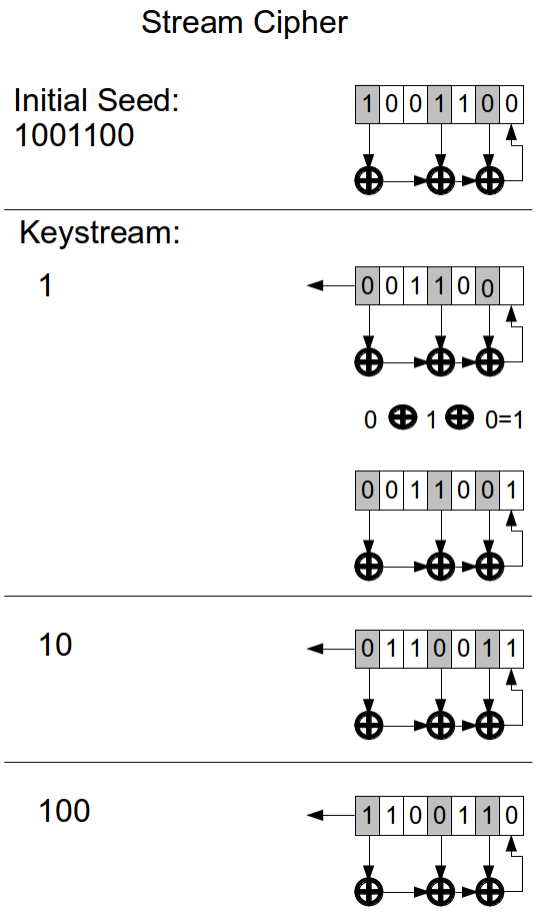

Figure 2.3. Example of a stream cipher.

Wireless Communications: An Introduction by Gordon Grow and Richard Smith [20], the Third Generation Partnership Project (3GPP) [9], and 4G Americas website [10]. The beginning of mobile phone technology, or 0G technology arose in the United States after World War II in 1946. AT&T introduced the Mobile Telephone Service (AT&T 0G Mobile Phone Technology) (MTS) in St. Louis with a limit of 5,000 customers. The MTS consisted of mobile radios that were permanently mounted in automobiles. The user had to call the operator who would then dial the phone number for them. To talk the user would have to hold down a button and then release it to hear the other person. The equipment weighed in around 79 pounds with large powerful antenna's to provide coverage around the city. The problem with the MTS system was that there were only three radio channels available for use which meant that no more than three people at a time could use their mobile phone. Another drawback was the cost as the price was equivalent to $176 per month and around $4 per call in todays money. Nearly 20 years later AT&T introduced the Improved Mobile Telephone Service (IMTS). This allowed up to 40,000 customers nationwide along with the user being

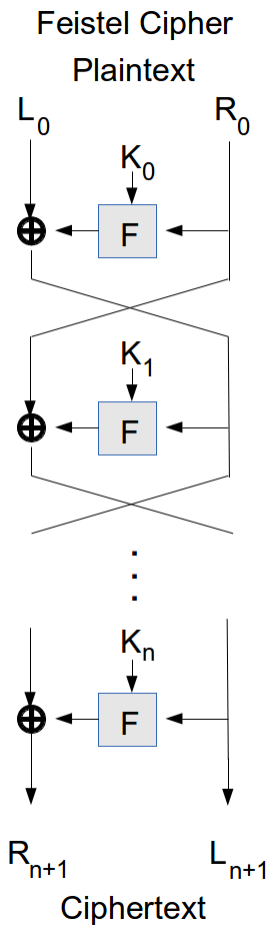

Figure 2.4. Example of a feistel cipher.

able to manually dial their own calls rather than going through the operator. The problem with this system was that demand was high yet the number of open channels were few. The city of New York only had 12 radio channels for 2,000 customers to share which resulted in an average 30 minute wait time for an open channel. Despite the high cost and wait time to place a call demand was still high and led to the need for a more efficient system to allow more users to use the few channels available.

Nearly 40 years after the introduction of the MTS system did the 1G of mobile phones become available in the late 1970's. The long delay in the roll out of a 1G mobile phone system was not due to the lack of demand or technology, but rather the government and AT&T itself. The government was reluctant to allocate radio channels for a mobile phone service and the private monopoly telephone operator AT&T did not foresee any business reason for expanding its mobile phone service for the general public. Despite these setbacks 1G mobile phone service was first introduced in Chicago in 1978 operating on the new Analogue Mobile Phone System (AMPS). Although in 1969 there was a similar service used on Amtrak

Metroliner train lines between New York City and Washington DC. The Amtrak system had six radio channels and used special pay phones for riders to make calls while traveling at 100 miles per hour along the 225 mile route. The 1G mobile phones were a true mobile phone that a customer could easily carry compared to the 79 pound mobile phones of the 0G system. Although these phones are often still referred to as brick or block phones because of their large bulky size. These 1G phones are often considered the birth of the modern mobile phone and were brought about by three key technological advances. The first being the invention of the transistor by Bell Labs in 1948 replacing the bulky and power hungry vacuum tubes of radio systems. Secondly the integrated circuit by Texas Instruments in 1958 allowed radio devices to be made smaller and consume less power. The third key technological development came in the 1970s when Intel began to market its first microprocessors for the production of electronic calculators.

There were three main problems with the 1G mobile phones. The first problem was that they offered no encryption of voice communications. It was a hobby of some radio enthusiast to eavesdrop on peoples conversations using a radio scanner. The second problem was that they could be cloned. With a cloned phone an adversary could use your account information to make calls at your expense. The third problem was that when your cellular phone connected to a cell tower to make a call, you couldn't travel out of that towers signal reach otherwise you would loose the call.

To address these problems of cloning, encryption, and transfer of calls from tower to tower a new generation of mobile phones were introduced in the early 1990s referred to as 2G. Although the last 1G AMPS was not shut down in North America until 2008. 2G mobile phones brought about three significant changes: digital encryption, data services with Short Message Services (SMS) text messaging, and a more efficient radio spectrum system to allow more availability of users on the network. The analog 1G mobile phones used 60kHz wide bandwidth to send a signal while the digital 2G mobile phones only required 10kHz or less. This is due to the fact that digital voice data can be compressed much more efficiently than analog voice data. The 2G network also allowed your mobile phone call to be transferred from cell tower to cell tower based on signal strength as opposed to keeping with the same cell tower per call as in the 1G networks. 2G mobile phones were first introduced in Europe using the GSM network. Later the United States introduced the CDMA networks instead of using the European GSM standard. One of the reasons U.S. carriers chose CDMA over GSM was that GSM came out around 1991 while CDMA came out in 1995. At the time CDMA was the newer technology, offered better sound quality, and was cheaper to build upon from the AMPS network. Therefore most of the U.S. carriers choose to go with CDMA. Verizon, Sprint, MetroPCS, Cricket, and U.S. Cellular are all CDMA based while AT&T and T-Mobile chose

to go with GSM. To allow for backwards compatibility the carriers have been stuck with their initial decisions as the switch from CDMA to GSM would require a significant investment in hardware changes. Today 80% of all mobile phones are GSM based while only 17% are CDMA based. CDMA based mobile phones are primarily used in the United States and South Korea while the rest of the world uses GSM. However Verizon Wireless is currently in the works of changing their CDMA based networks which is discussed more in the 4G section.

Some of the advantages of the GSM system were that since it was used worldwide you had better international roaming, a removable SIM card storing your account information allowing you to switch phones whenever you liked, and it was easier to buy equipment for the GSM network as it was the worldwide standard. Some of the advantages of the CDMA networks were that it allowed more users per bandwidth than GSM, consumed less power due to its design, and offered better call quality initially. Note that CDMA technology is patented as it was developed by Qualcomm. This means that all CDMA carriers must pay licensing fees to Qualcomm. Since GSM has dominance in the global market mobile phone manufacturers often focus on releasing new features in GSM mobile phones first and then later to CDMA. In regards to being able to switch phones, to be a GSM carrier it is required that you accept any unlocked GSM compliant phone on your network. CDMA does not have this requirement and the carriers decide if they will accept your phone. This is the reason why an AT&T customer can take their unlocked phone over to T-Mobile and vice versa. This is unlike Sprint and Verizon who both use CDMA but reject other networks CDMA based phones. Sprint only accepts phones on its network that were built specifically for Sprint. Another thing worth noting is that CDMA technology does not allows for simultaneous voice and data usage unlike GSM. This means that you cannot talk and use the Internet at the same time. Note though that CDMA technology has recently advanced to correct this in 2011 to what is referred to as Simultaneous Voice and Data Optimization (SV-DO). With all the differences between CDMA and GSM based networks in all practical matters one should decide which carrier to use based on two criteria. The first being whether or not you need to switch phones or travel abroad often. The other being which network gives you the best reception at your home and office.

When the 2G GSM technology was first introduced the main focus was on voice services rather than data. Data usage was slow and limited to sending SMS text messages. When using data services you had to connect to the network similarly to a dial-up modem, thus data usage was charged on a per minute basis. With the rise in demand for faster data speeds and the availability of the Internet on the phone, an intermediate upgrade to the 2G system was introduced and is considered 2.5G technology. There was also a shift in terminology from ”cell phones” to ”feature phones” as they introduced more features such as

always-on data. This 2.5G GSM network was introduced in 1997 and is referred to as General Packet Radio Services (GPRS). This was the GSM answer to the CDMA technology introduced two years earlier. GPRS provided an upgraded data transfer rate of 56 to 115 kbit/s as opposed to the GSM 9.6 to 14.4 kbit/s speeds. For comparison a dial-up analog wired telephone modem has data transfer rates of 32-40 kbit/s. This increased data speed of GPRS brought about Multimedia Messaging Service (MMS), email, and World Wide Web Access. For an example of the data speed differences between GSM and GPRS, GSM has an SMS transmission speed of 6 to 10 SMS messages per minute as opposed to GPRS with a transmission speed of 30 SMS messages per minute. GPRS also brought about a shift in billing from a charge per minute connection to being charged for data transferred. This was also the first time downloadable content was made available with the purchase of ringtones. To compete with GPRS CDMA upgraded their networks in 2000 to what is referred to as CDMA2000 or CDMA2000 1xRTT (1x stands for 1 times Radio Transmission Technology). CDMA2000 doubled the speeds of CDMA and is also considered a 2.5G technology.

The need for increased data usage brought about another intermediate upgrade of the GSM/GPRS system in 2003 to what is referred to as 2.75G technology. This new system is called the Enhanced Data rates for GSM Evolution (EDGE). The 2.75G EDGE network offered a data transfer rate of 384 kbit/s which is about three times as fast as GPRS 2.5G technology. The cryptographic security algorithm for the GPRS system is not covered in this paper as the design is owned and copyrighted by the European Telecommunications Standards Institute (ETSI). However you can buy the algorithm called GEA along with its code written in C from ETSI for only $1,557.55 along with signing a Confidentiality and Restricted Usage Undertaking. The 2.5G GPRS/CDMA2000 technology paved the way for the release of 3G technology. A demand for a 3G mobile phone was brought about by a want for video calling and the ability to use Voice over Internet Protocol (VoIP) on a mobile phone. There was also a lack in confidence in the security of 2G GSM/GPRS networks as their cryptographic systems were never publicly disclosed.

The Universal Mobile Telecommunications System (UMTS) was the first GSM based 3G network. UMTS was introduced in 2001 by the Third Generation Partnership Project (3GPP). The 3GPP was formed to create a global cellular phone system. The 3GPP consisted of representatives from the following organizations: Association of Radio Industries and Businesses (Japan), Alliance for Telecommunications Industry Solutions (USA), China Communications Standards Association (China), European Telecommunications Standards Institute (Europe), Telecommunications Technology Association (Korea), and Telecommunication Technology Committee (Japan). With international cooperation a user with a 3G UMTS mobile phone could travel throughout Japan, USA, China, Europe, and

Korea while using the same phone. Note that 3G UMTS was introduced in 2001 while the 2.75G EDGE network was introduced in 2003. This is due to the fact that the upgrade from a 2G GSM/GPRS network to a 3G UMTS network required a significant upgrade in equipment. For those carriers that didn't want to invest that much money the 2.75G EDGE network was introduced to provide upgraded data speeds at little cost in comparison to the 3G UMTS upgrade. Because of this the cryptographic algorithms of EDGE are based off of the algorithms for UMTS with only a slight modification to the algorithms input. For this reason the cryptographic system for EDGE is explained in the 3G UMTS section. The cost for UMTS was expensive as it required new base stations to be installed unlike GPRS and EDGE which required only bolt-on upgrades. UMTS still uses the same core network as GSM which is why it is also referred to as 3GSM for 3rd Generation GSM. To compete with the UMTS network CDMA2000 1xRTT (2.5G) was upgraded to CDMA2000 1xEV-DO (3G) which stands for Evolution-Data Only.

3G technology introduced mobile applications, video calling, VoIP, and true global roaming. These 3G cell phones are referred to as "smart phones" providing an upgrade from the 2G "feature phones." Smart phones are recognizable from feature phones in that smart phones have far less buttons (around 4 or less) compared to feature phones (9 or more) and rely on touch screen navigation. The average data transfer rate for UMTS in 2001 was around 384 kbit/s. In 2004 CDMA2000 1xEV-DO upgraded their software to CDMA2000 1xEV-DO Revision A which offered increased data speeds and is referred to as 3.5G. In 2006 UMTS upgraded their systems to High-Speed Downlink Packet Access (HSDPA) which gave data speeds of up to 42 Mbit/s. Because of the significant upgrade in data speeds HSDPA is often referred to as 3.5G as well. In the same year CDMA2000 1xEV-DO Revision A also received an upgrade to Revision B (3.5G). With the increased speeds of HSDPA and CDMA2000 1xEV-DO Rev. B mobile TV streaming and video on demand became possible. Since HSDPA's primary focus was on increasing download speeds, the next upgrade came in late 2008 with an increase to upload speeds and is referred to as Evolved High-Speed Packet Access (HSPA+). HSPA+ is often referred to as 3.75G technology and offers a downlink speed of 84 Mbit/s and uplink speed of 10.8 Mbit/s. Real life speeds of the HSPA+ network is actually around 21 Mbit/s downlink.

An alternative to the UMTS/HSPA+ and CDMA/CDMA-2000 networks arose in 2008 when Sprint introduced mobile Worldwide Interoperability for Microwave Access (WiMAX). WiMAX was originally built for wireless broadband internet in homes and offices. The motivation for developing wireless broadband internet is to avoid the expensive cost of laying fiber optic cables especially for those people living in rural areas. The speeds of WiMAX are about the same as DSL modems (5-10 Mbit/s). Mobile WiMAX takes the same system as

WiMAX and uses it for a mobile phone network. Sprint decided to go with WiMAX instead of HSPA+ or CDMA-2000 because it was backed by Intel who promised to put WiMAX into their laptops and make it a global standard. The problem is that Sprint took way too long to build the WiMAX network and by 2010 it was only available in a small amount of cities. Meanwhile in 2009 there was another upgrade for the UMTS/HSPA+ networks called LTE. LTE is referred to as 3.9G and because of the slow development of WiMAX and the easier bolt-on upgrade of LTE the UMTS/HSPA+ networks did not change to Mobile WiMAX. The speeds offered by LTE surpassed WiMAX and allowed for better backwards compatibility as it is based on the GSM network. Sprint has since given up on WiMAX and is now building LTE networks with a complete shutdown date for WiMAX in 2015. Before LTE and WiMAX came out the radio spectrum was divided between voice and data. LTE and WiMAX were able to achieve higher data speeds by dedicating all of their radio spectrum to data. Therefore voice calls are for the first time treated as VoIP. Because of this dedication to data LTE and WiMAX brought about increased data speeds of a theoretical 299.6 Mbit/s downlink and 75.4 Mbit/s uplink. In real life tests the actually speeds are between 50 Mbit/s downlink which is little more than twice the real life speeds of HSPA+. As more HSPA+ equipment is upgraded to LTE the speed will continue to rise and come closer to the 299 Mbit/s downlink speed. Since HSDPA/HSPA+/LTE are all upgrades of the core UMTS network they all share the same security algorithms. Thus when the cryptographic systems of UMTS are described below they are referring to all of the UMTS/HSDPA/HSPA+/LTE networks.

The standards used to define whether or not a phone is considered 3G or 4G is defined by the International Telecommunication Union (ITU). The ITU is an agency of the United Nations that is responsible for issues concerning information and communication technologies. The ITU set the requirements for 4G standards to be 100 Mbit/s for communications from cars and 1 Gbit/s for walking or stationary users. Note that LTE and WiMAX fall significantly short of reaching these speeds. The marketing department for Sprint wanted to label their WiMAX network as the next greatest thing to sell more phones then their competitors. Sprint decided to ignore the ITU standards for qualifying for 4G and called their WiMAX networks 4G. It wasn't long before the other U.S. carriers began to market their LTE networks as 4G to be competitive with Sprint. The networks reasoning for ignoring the international standards were that WiMAX and LTE will eventually become 4G as their next generations labeled WiMAX-2 and LTE-Advanced will meet the 4G specifications. Then T-Mobile started calling their HSPA+ network 4G. Once T-Mobile started using the 4G description for their HSPA+ networks the term 4G became meaningless as an international standard. The U.S. carriers have no plans of remarketing their networks as "Evolved 3G" as the ITU recommended. Their reason for not changing being that they think it will cause

confusion with their customers to relabel 4G to Evolved 3G. Therefore the term 4G has turned into a marketing term rather than an international standard. The plans for a true 4G technology such as LTE-Advanced and WiMAX-2 are still in development. The current estimates for the future is that HSPA+ will provide the best speeds for about another 5 years and at that point LTE will advance and take over. The CDMA answer for a 4G network was referred to as CDMA2000 1xEV-DO Revision C or Ultra Mobile Broadband (UMB). The UMB system was never adopted as most carriers have decided to switch to the LTE network making LTE the first global mobile phone system. CDMA2000 hasn't abandoned it's users as 1xEV-DO Rev. B saw an upgrade to their network in 2011 by offering SV-DO which stands for Simultaneous Voice and EV-DO data. This finally allowed CDMA2000 users to use voice and data services at the same time as the GSM networks have been doing since 2.5G.

The future of mobile phone networks is heading towards LTE. The CDMA2000 4G UMB network has been abandoned by Qualcomm and all the major U.S. carriers have plans of switching to LTE. Verizon Wireless, the largest CDMA network, has announced that it will no longer offer 2G and 3G CDMA service around 2021. Even though Verizon has made the switch to the GSM based LTE network, this should be no problem to consumers as every two years Verizon customers are allowed to upgrade their phones to the newest models. The current Verizon 4G LTE mobile phones use SIM cards but they will not accept unlocked phones from AT&T or T-Mobile. You can however switch your SIM card between different Verizon 4G phones. To be part of the GSM based network it is required that you allow for the mobile subscriber to use any unlocked device they like. Verizon is able to deny this requirement because the majority of their network is still the closed model CDMA network. As the Verizon LTE network grows and surpasses the CDMA based network it will be required to allow non-Verizon phones to be used on their network. This takeover from CDMA to GSM based LTE is expected to happen by the end of 2014. Another sign of the end of the CDMA based networks is the announcement by Google that they will no longer support CDMA devices as part of their Android Open Source Project. Android is the the world's most widely used smartphone operating system and this leaves CDMA based phones with outdated versions. All of the major carriers in the U.S. now offer limited LTE service in certain areas as they expand their network.

2.3 GSM Network Architecture

The 2G GSM core network architecture set the foundation and is the basis for all of its technological derivatives such as GPRS/EDGE and UMTS/HSPA+/LTE networks. A brief outline for the GSM network architecture is taken from the book GSM System Engineering by Asha Mehrotra [23]. The GSM network architecture consists of three main parts: the Mobile Station (MS), the Base Station Subsystem (BSS), and the Network SubSystem (NSS).

The MS consists of the Mobile Equipment (ME) along with a SIM card. The majority of subscribers ME consist of mobile phones together with the SIM card. An example of a non mobile phone ME could be a laptop computer. The SIM card is used to identify the user and contains all of the account data. The advantage of this is that the SIM card can be removed and used in other ME while allowing the user to keep the same phone number and services. Each SIM card contains a unique International Mobile Subscriber Identity (IMSI) number which is used to identify the user on the network. The IMSI is typically a 15 digit number with the exception of South Africa which uses only 14 digits. The first three digits of the IMSI represent the countries Mobile Country Code. The next 2 or 3 digits identify the Mobile Network Code (MNC) used to identify the carrier (AT&T, T-Mobile, etc.). A carrier can have more than one MNC as they inherent the MNC's of the networks they take over. The U.S. uses three digit MNC's while the rest of the world uses two digit MNC's. To allow for global roaming the last digit of most U.S. MNC's are left as zero to compensate for this difference in numbering. The remaining digits represent the Mobile Subscription Identification Number (MSIN). The MSIN is assigned by the carrier and is used to identify the user on their network.

Each SIM card also contains a four to eight digit Personal Identification Number (PIN) used to secure the data on the SIM. The PIN is used for unlocking the data on the SIM card. To prevent tampering the user only has three tries before the SIM card will shut down and no longer function. To restore the SIM card the user needs to call their carrier and only after providing verification are they given a Personal Unblocking Key (PUK). The PUK is also stored on the SIM card. The user only has ten attempts for entering the correct PUK before the SIM is permanently blocked and considered unrecoverable. At this point you would need a new SIM card from the carrier. Also located on the SIM card is an Individual Subscriber Authentication Key K_i along with the security algorithms used in the authentication and encryption process. The ME also contains an international identification number referred to as the International Mobile Equipment Identity (IMEI). The IMEI is used to identify valid devices allowing the networks to disable stolen equipment. The IMEI is usually located behind the battery. If you loose your phone you can call the carrier and have them blacklist your IMEI which will disable that phone from being used on their network. An average person cannot easily change the phones IMEI but a technically knowledgeable thief can change a stolen phones IMEI in order to use it on the network. This process is known as IMEI spoofing or IMEI cloning. The advantage to having an IMEI and IMSI number with no relation to one another is that it makes it easy to transfer you personal data and services between phones.

The next main part of the GSM network architecture is the BSS. The BSS is responsible for the radio link between the user (ME+SIM) and the network. It is composed of

two parts: the Base Transceiver Station (BTS) and Base Station Controller (BSC). The BTS's are the set of radio transmitters, receivers, and antennas that are used to connect the user to the network. Each coverage area is laid out in a grid like pattern and each cell of the grid will ideally have it's own BTS. The MS will connect to the BTS that is providing the strongest signal. As you continue to move out of a certain BTS's cell and into another the BSC is used to route the call. Each BSC manages a group of BTS's and acts as a middle man for the MS and the NSS by setting up the voice and data calls.

The third part of the GSM network architecture is the NSS. The NSS is the core of the GSM network and consists of the Gateway Mobile Switching Center (GMSC), Mobile Switching Center (MSC), Home Location Register (HLR), Visitor Location Register (VLR), Authentication Center (AuC), Equipment Identity Register (EIR), and the Operation and Maintenance Center (OMC). The BSC is responsible for transferring the call between the BTSs just like the MSC is responsible for transferring calls between the BSC's. The MSC is considered the central component of the NSS and all calls are routed through it. To keep track of the subscribers on the NSS a HLR is used. The HLR is a database that keeps track of a subscribers information such as the telephone number, IMSI number, along with the users current location. The reason the users current location is stored in the HLR is that when someone else places a call to your mobile phone the HLR is the first to be initiated in order to route the call to your location. The VLR works with the HLR in that it stores all users in the current area covered by the VLR not just subscribers. The HLR and VLR allow for roaming to occur on the network. The AuC is responsible for holding the 128-bit Individual Subscriber Authentication Key K_i shared with the SIM card. Note that the Key is never transmitted from the SIM nor the AuC. Since the K_i is stored in the AuC it is responsible for generating the authentication vectors used in verifying the user and setting up the encryption of data. The users equipment and account information are kept separate thus the EIR verifies the IMEI to ensure it is compatible with the GSM network. Once the verification is complete the IMEI is stored in the EIR's database. The EIR's database classifies each IMEI by a black list, gray list, or white list category. All of the IMEIs that have been reported as stolen are placed on the black list and are denied access to the network. Phones that are incompatible with the network are also placed on the black list. —acimeis on the gray list are for phones that are not functioning properly or are acting strangely. IMEIs on the gray list are monitored by the network. All other IMEIs that are not on the gray list or black list are placed on the white list. The white list consists of all the properly functioning mobile phones. The OMC is responsible for maintaining the MSC, BSC, and the BTS ensuring that they are functioning properly. The GMSC is used to connect the user to different networks or carriers. If you wanted to call a landline phone or if you are on T-Mobile and wanted to place a call to an AT&T phone your

call would be routed through the GMSC. The GMSC can connect the user to other carriers, the analog Public Switched Telephone Network (PSTN or the plain old telephone system), the digital Integrated Services Digital Network (ISDN), or Packet Data Networks (PDN). A diagram of the GSM network architecture is given in Figure 2.5.

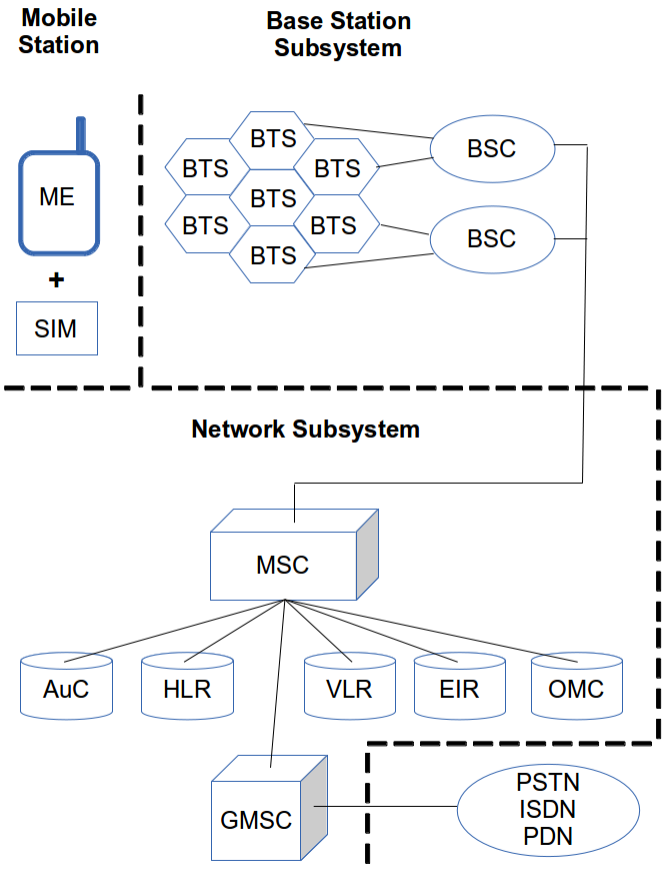

Figure 2.5. Overview of the GSM network architecture.

2.4 Authentication and Encryption for GSM networks

The authentication process begins with the MS requesting access to the network by broadcasting the IMSI/IMEI. This usually occurs when the MS is turned on for the first time or if the MS has been turned off for more than 48 hours. The MSC receives the IMSI/IMEI and sends the IMSI to the HLR with a request for an authentication triplet. The IMEI is also

sent to the EIR to check its database to make sure the number is not on the black list. The HLR likewise checks the IMSI with its database to ensure the user is a valid subscriber. Once the IMSI/IMEI have been verified the HLR will send the triplet request along with the IMSI to the AuC. The AuC is the only other place beside the SIM card that stores the 128-bit individual subscribers authentication key, K_i, associated with each IMSI. The IMSI and SIM card are created together when the SIM card is produced. The AuC uses the IMSI to look up the individual subscribers K_i and then produces a 128-bit Random Number (RAND). The AuC also contains the authentication and ciphering key algorithms referred to as A3 and A8 respectively. The RAND and K_i are inputted into the A3/A8 algorithms producing a 32-bit Signed Response (SRES) for authentication and a 64-bit ciphering key K_c used for encryption of data. The SRES is considered the challenge to the MS. The RAND , SRES, and K_c are referred to as the triplet. To reduce the load of authentication requests to the AuC the AuC produces a set of five triplets at a time and sends them to the MSC for storage. When the MS authenticates again the MSC will use one of the unused stored triplets before requesting another triplet from the AuC. Note that each triplet is only good for the IMSI it was generated for. The MSC chooses one of the five triplets and then sends the RAND or challenge to the MS. The MS uses the K_i stored on the SIM card along with the RAND received from the MSC as inputs into the A3/A8 algorithms located on the SIM card. The A3/A8 algorithms produce the SRES and K_c. The MS then sends the SRES back to the MSC for verification. The MSC then compares the received SRES from the MS with the SRES generated from the AuC. If they agree then the MS is authenticated. If they disagree the connection is terminated and an authentication failure message is sent to the MS.

The encryption process begins after the MS has been authenticated on the GSM network. The MSC transfers the cipher key K_c to the BTS and the unencrypted communication between the MS and BTS is then put into Cipher Mode. The encryption algorithm referred to as A5 is stored on the BTS and the ME. The cipher key K_c generated from the A8 algorithm is then inputted into the A5 algorithm to produce a keystream. This keystream is used for the encryption/decryption of over the air communications between the MS and BTS.

CHAPTER 3

CRYPTOGRAPHIC ALGORITHMS FOR 2G GSM NETWORKS

One of the biggest problems facing the 1G mobile phones was that no encryption was used with the AMPS network. This brought about a need for a 2G mobile phone where over-the-air communications was encrypted. This 2G mobile phone began with the 2G GSM network. There are four main algorithms used in 2G GSM mobile phones:

A3: Authentication Algorithm
A8: Cipher key K_c Algorithm
A5/1: "Stronger" Over The Air Communication Encryption Algorithm
A5/2: "Weaker" Over The Air Communication Encryption Algorithm

These algorithms were privately developed by the Security Algorithm Group of Experts (SAGE). The stronger A5/1 algorithm was used in Western Countries and the deliberately weakened version A5/2 was imported to the other countries. It was rumored that a weaker algorithm was used at the request of European intelligence agencies headed by the creator of the A5/2 algorithm France. By 1999 the algorithm was considered extremely weak a month after it was leaked to the public. The A5/2 algorithm was then mandated to be phased out according to the GSM Association by 2006. In 2007 it was then prohibited to implement the A5/2 algorithm in any new phone. Countries that relied only on A5/2 that didn't add support for the A5/1 algorithm were left with no encryption at all on newer phones.

The upgrade from 2G GSM to 2.5G GPRS brought about a new set of security algorithm. These algorithms were again developed by SAGE behind closed doors and never released to the public. The original GSM network used the COMP128 algorithm for their A3/A8 algorithms. The COMP128 algorithm is a one way compression function. There have been three versions of the COMP128 algorithm with only the first version publicly known after it was leaked. GPRS used an upgraded COMP128 algorithm referred to as COMP128-2 which supposedly fixed certain security flaws. A year after GPRS was released the algorithm was upgraded to what is referred to as COMP128-3. These COMP128 algorithms all have the same core design but each upgrade brought about security enhancements. Since the designs were never made public the security of GSM mobile phones relied heavily on security by obscurity. The advantage of releasing the algorithm to the public is that the more people you have scrutinizing the algorithm the better chances there are of finding any flaws in its design.

The new encryption algorithm for GPRS is referred to as the GPRS Encryption Algorithm (GEA). When the GPRS system had an update a year later GEA was upgraded to an algorithm referred to as GEA2. These algorithms have not been included in this paper as they have not been published. In this chapter we will examine the 2G GSM authentication and cipher key algorithms A3/A8 along with the A5 encryption algorithms.

3.1 Authentication and Cipher Key Generation: A3 & A8

Authentication of the mobile user and the creation of the Cipher Key K_c is accomplished by means of the A3/A8 algorithms. The A3 algorithm is used to authenticate the mobile user by sending back a response (SRES) to the challenge (RAND) given by network. The A8 algorithm is used to generate the 64-bit Cipher Key K_c which is used in the encryption of over-the-air communications. The A3/A8 algorithms are located on the protected SIM card along with the 128-bit Individual Subscribers Authentication Key K_i. The input for both algorithms are a 256-bit combination of the 128-bit RAND and the 128-bit K_i. Since both algorithms rely on the same input they are executed at the same time and rely on the algorithm called COMP128. The 256-bit input for the COMP128 algorithm produces a 96-bit Output which is composed of the 32-bit SRES and the 64-bit Cipher Key K_c.

The COMP128 algorithm was leaked through an email which contained all but the last four lines of code. This missing code dealt with the derivation of the cipher key after the algorithm has run for eight iterations of compressions and permutations. These last four lines of code were reversed-engineered using a working SIM card by Marc Briceno (Smartcard Developer Association), Ian Goldberg (Berkeley), and David Wagner (Berkeley). The outline for the COMP128 algorithm contained in the original email missing the last four lines of code are as follows[13]:

COMP128 Algorithm

(Load RAND into last 16 bytes of input)
FOR i from 16 to 31
    $x[i] = RAND[i]$
END FOR

(Loop eight times)
FOR i from 1 to 8

    (Load key into first 16 bytes of input)
    FOR j from 0 to 15

$x[j] = key[j]$
END FOR

(Perform Substitutions)
FOR j from 0 to 4
FOR k from 0 to $2^j - 1$
FOR l from 0 to $2^{4-j} - 1$

$m = l + k * 2^{5-j}$
$n = m + 2^{4-j}$

$y = (x[m] + 2 * x[n]) \mod 2^{9-j}$
$z = (2 * x[m] + x[n]) \mod 2^{9-j}$

$x[m] =$Table$[j, y]$
$x[n] =$Table$[j, z]$

END FOR
END FOR
END FOR

(Form bits from bytes)
FOR j from 0 to 31
FOR k from 0 to 7
$bit[4 * j + k] =$the $(8 - k)$th bit of byte j
END FOR
END FOR

(Permutation but not on the last loop)
IF $(i < 8)$ THEN
FOR j from 0 to 15
FOR k from 0 to 7
$next\_bit = (8 * j + k) * 17 \mod 128$
bit k of $x[j + 16] = bit[next\_bit]$
END FOR
END FOR
END IF

END FOR

The above algorithm contained everything except the derivation of the cipher key K_c. The algorithm produces the vector $x[0...31]$ with each entry representing 4-bits. The first eight entries of vector $x[]$ represent the SRES. The missing four lines of the algorithm reversed-engineered by Marc Briceno, Ian Goldberg, and David Wagner presented below provide the missing derivation of the session key completing the COMP128 algorithm (Written in C) [17]:

for $(i = 0; i < 6; i++)$
simoutput$[4 + i] = (x[2 * i + 18] << 6) \mid (x[2 * i + 18 + 1] << 2)$
$\mid (x[2 * i + 18 + 2] >> 2);$
simoutput$[4 + 6] = (x[2 * 6 + 18] << 6) \mid (x[2 * 6 + 18 + 1] << 2);$
simoutput$[4 + 7] = 0;$

The vector simoutput[] consists of 12 entries with each entry representing a byte. The first four entries of simoutput[] are taken straight from the first 8 nibbles of $x[]$. Thus simoutput$[0...3]$ =SRES. the last 8 entries of simoutput[] represent the Cipher Key, simoutput$[4...11] = K_c$. Since simoutput$[11] = 0$ and the last two bits of simoutput$[10]$ are zero, the last ten bits of the Cipher Key K_c are zeros. Setting the last ten bits of K_c to zero gives the Cipher Key an effective length of 54-bits rather than 64-bits. Once the SRES and Cipher Key K_c have been produced the SRES is sent back to the BTS for authentication of the user and the Cipher Key K_c is used to begin encryption of over the air communications.

Examining the algorithm it begins by inputting the Key K_i and the random challenge $RAND$ into a 32-byte vector $x[]$. Thus $x[0...15] = K_i$ and $x[16...31] = RAND$. The algorithm then runs through eight rounds. Each round consists of 5 subrounds of compressions and permutations on all but the last round. Each subround of compression contains its own compression table. Table 0, Table 1, Table 2, Table 3, and Table 4 consist of 512, 256, 128, 64, and 32 entries respectively of lengths 8-bit, 7-bit, 6-bit, 5-bit, and 4-bit. Thus Table 3 consists of 64 entries where every entry is less than or equal to 5 bits or 32 (Decimal notation). Each subround of compression i uses Table i for its lookup value. The compression tables were included in the leaked document containing the COMP128 code above. Table 0 is represented in Table 3.1,Table 1 in Table 3.2, Table 2 in Table 3.3, Table 3 in Table 3.4, and Table 4 in Table 3.5 [17].

Since Table 4 contains only 4-bit values, by the time the last subround of compression is completed, the vector $x[]$ consists of 32 4-bit values. These 32 4-bit values are then stored

Table 3.1. COMP128 Table 0 Source: M. BRICENO, I. GOLDBERG, AND D. WAGNER, *An implementation of the gsm a3 a8 algorithm.* Unpublished report, 1998.

102	177	186	162	2	156	112	75	55	25	8	12	251	193
246	188	109	213	151	53	42	79	191	115	233	242	164	223
209	148	108	161	252	37	244	47	64	211	6	237	185	160
139	113	76	138	59	70	67	26	13	157	63	179	221	30
214	36	166	69	152	124	207	116	247	194	41	84	71	1
49	14	95	35	169	21	96	78	215	225	182	243	28	92
201	118	4	74	248	128	17	11	146	132	245	48	149	90
120	39	87	230	106	232	175	19	126	190	202	141	137	176
250	27	101	40	219	227	58	20	51	178	98	216	140	22
32	121	61	103	203	72	29	110	85	212	180	204	150	183
15	66	172	196	56	197	158	0	100	45	153	7	144	222
163	167	60	135	210	231	174	165	38	249	224	34	220	229
217	208	241	68	206	189	125	255	239	54	168	89	123	122
73	145	117	234	143	99	129	200	192	82	104	170	136	235
93	81	205	173	236	94	105	52	46	228	198	5	57	254
97	155	142	133	199	171	187	50	65	181	127	107	147	226
184	218	131	33	77	86	31	44	88	62	238	18	24	43
154	23	80	159	134	111	9	114	3	91	16	130	83	10
195	240	253	119	177	102	162	186	156	2	75	112	25	55
12	8	193	251	188	246	213	109	53	151	79	42	115	191
242	233	223	164	148	209	161	108	37	252	47	244	211	64
237	6	160	185	113	139	138	76	70	59	26	67	157	13
179	63	30	221	36	214	69	166	124	152	116	207	194	247
84	41	1	71	14	49	35	95	21	169	78	96	225	215
243	182	92	28	118	201	74	4	128	248	11	17	132	146
48	245	90	149	39	120	230	87	232	106	19	175	190	126
141	202	176	137	27	250	40	101	227	219	20	58	178	51
216	98	22	140	121	32	103	61	72	203	110	29	212	85
204	180	183	150	66	15	196	172	197	56	0	158	45	100
7	153	222	144	167	163	135	60	231	210	165	174	249	38
34	224	229	220	208	217	68	241	189	206	255	125	54	239
89	168	122	123	145	73	234	117	99	143	200	129	82	192
170	104	235	136	81	93	173	205	94	236	52	105	228	46
5	198	254	57	155	97	133	142	171	199	50	187	181	65
107	127	226	147	218	184	33	131	86	77	44	31	62	88
18	238	43	24	23	154	159	80	111	134	114	9	91	3
130	16	10	83	240	195	119	253						

Table 3.2. COMP128 Table 1 Source: M. BRICENO, I. GOLDBERG, AND D. WAGNER, *An implementation of the gsm a3 a8 algorithm.* Unpublished report, 1998

19	11	80	114	43	1	69	94	39	18	127	117	97	3
85	43	27	124	70	83	47	71	63	10	47	89	79	4
14	59	11	5	35	107	103	68	21	86	36	91	85	126
32	50	109	94	120	6	53	79	28	45	99	95	41	34
88	68	93	55	110	125	105	20	90	80	76	96	23	60
89	64	121	56	14	74	101	8	19	78	76	66	104	46
111	50	32	3	39	0	58	25	92	22	18	51	57	65
119	116	22	109	7	86	59	93	62	110	78	99	77	67
12	113	87	98	102	5	88	33	38	56	23	8	75	45
13	75	95	63	28	49	123	120	20	112	44	30	15	98
106	2	103	29	82	107	42	124	24	30	41	16	108	100
117	40	73	40	7	114	82	115	36	112	12	102	100	84
92	48	72	97	9	54	55	74	113	123	17	26	53	58
4	9	69	122	21	118	42	60	27	73	118	125	34	15
65	115	84	64	62	81	70	1	24	111	121	83	104	81
49	127	48	105	31	10	6	91	87	37	16	54	116	126
31	38	13	0	72	106	77	61	26	67	46	29	96	37
61	52	101	17	44	108	71	52	66	57	33	51	25	90
2	119	122	35										

into the 128-bit vector $bit[]$. The original 256-bits have been compressed to 128-bits. The resulting 128-bits are then permuted in all but the last round of iterations. Once the 128-bits have been permuted they are then stored as 16 bytes in the vector $x[16...31]$. The next round of iterations begin be resetting $x[0...15]$ to K_i.

Examining the algorithm responsible for the five subrounds of compression (Perform Substitutions section) each resulting byte depends on two input bytes. For each subround the vector $x[0...31]$ is divided into equal size sections. For the first subround of compression $i = 0$ the vector $x[0...31]$ is divided into two equal sections $x[0...15]$ and $x[16...31]$. The two input bytes responsible for $x[0]$ and $x[16]$ are $x[0]$ and $x[16]$ themselves. the two input bytes for $x[1]$ and $x[17]$ are similarly $x[1]$ and $x[17]$. The next subround of compression for round $i = 1$ the vector $x[0...31]$ is divided into 4 equal parts. Thus we have $x[0...7]$, $x[8...15]$, $x[16...23]$, and $x[24...31]$. The two input bytes for $x[0]$ and $x[8]$ depend on $x[0]$ and $x[8]$. Likewise the two input bytes responsible for $x[16]$ and $x[24]$ depend on $x[16]$ and $x[24]$. This crosswise dependency for the resulting byte is referred to as a butterfly structure. Each subround of compression applies the same basic structure with the divided sections becoming smaller and smaller. In subround 1 the vector $x[0...31]$ is divided into 2 sections of length 16, subround 2 the vector is divided into 4 sections of length 8, and so on until we have subround 5 where

Table 3.3. COMP128 Table 2 Source: M. BRICENO, I. GOLDBERG, AND D. WAGNER, *An implementation of the gsm a3 a8 algorithm.* Unpublished report, 1998

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

Table 3.4. COMP128 Table 3 Source: M. BRICENO, I. GOLDBERG, AND D. WAGNER, *An implementation of the gsm a3 a8 algorithm.* Unpublished report, 1998

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

Table 3.5. COMP128 Table 4 Source: M. BRICENO, I. GOLDBERG, AND D. WAGNER, *An implementation of the gsm a3 a8 algorithm.* Unpublished report, 1998

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

$x[0...31]$ is divided into 32 sections of length 1. Thus in subround 5 the two input bytes responsible for $x[0]$ and $x[1]$ are $x[0]$ and $x[1]$. Similarly for $x[2]$ and $x[3]$ and so on.

3.1.1 Authentication and Cipher Key Generation Example

Using a randomly chosen K_i and RAND, an example of the authentication and Cipher Key generation process using COMP128 is given below. The Matlab code used in this example is included in the appendix under COMP128. It was created by examining the COMP128 algorithm written in C by Marc Briceno, Ian Goldberg, and David Wagner[17].

K_i =52 A1 B0 D7 F3 81 6E C6 E7 23 28 97 F1 70 A8 1A — Located on SIM
RAND=3B 98 47 53 F9 AA D8 A6 B8 DF 0E 21 DB 9A 2E 8D — RAND from BTS

$x[16...31]$ =3B 98 47 53 F9 AA D8 A6 B8 DF 0E 21 DB 9A 2E 8D — Load RAND

Round 1 0f 8
$x[0...15]$ =52 A1 B0 D7 F3 81 6E C6 E7 23 28 97 F1 70 A8 1A — Load Load K_i
$x[16...31]$ =3B 98 47 53 F9 AA D8 A6 B8 DF 0E 21 DB 9A 2E 8D
Compression: — Perform Substitutions

Subround 1:

$x[0...15]$ =EC 39 74 8C 56 8E 6C 97 4A B8 47 32 E5 22 9C B3

$x[16...31]$ =E2 B5 DC B1 E2 BD 7A 0D 55 40 F5 D7 C8 D8 79 AA

Subround 2:

$x[0...15]$ =5F 30 50 65 23 31 0C 77 67 20 06 0E 6B 69 66 26

$x[16...31]$ =6A 5F 54 33 57 63 4E 41 59 48 36 44 6A 68 63 41

Subround 3:

$x[0...15]$ =01 28 38 1A 26 04 07 02 03 2F 2D 09 16 0D 2F 15

$x[16...31]$ =3A 01 1F 1F 0E 0A 31 03 2A 3A 3F 3C 0C 25 3D 12

Subround 4:

$x[0...15]$ =15 04 0D 0F 1A 11 1E 00 03 05 1E 02 1A 1F 12 0A

$x[16...31]$ =0B 13 1E 00 11 1C 19 0D 1D 0C 1E 11 12 13 1B 04

Subround 5:

$x[0...15]$ =0D 0D 07 00 0B 0E 0F 0B 02 07 0A 0F 05 09 0B 0D

$x[16...31]$ =03 00 0F 0B 00 0F 09 0C 00 0B 0F 02 05 02 04 08

Bits from Bytes:
11011101 01110000 10111110 11111011 00100111 10101111 01011001 10111101

00110000 11111011 00001111 10011100 00001011 11110010 01010010 01001000

Permutation: Except on Round 8

$x[0...15]$ =0D 0D 07 00 0B 0E 0F 0B 02 07 0A 0F 05 09 0B 0D

$x[16...31]$ =B6 7E 9B B8 68 B1 07 DB 1F A3 1C DD 51 FE 6B 68

Round 2 0f 8

$x[0...15]$ =52 A1 B0 D7 F3 81 6E C6 E7 23 28 97 F1 70 A8 1A Load Load K_i

$x[16...31]$ =B6 7E 9B B8 68 B1 07 DB 1F A3 1C DD 51 FE 6B 68

Compression: Perform Substitutions

Subround 1:

$x[0...15]$ =52 3C 2C 0E 88 83 3C 8C 40 AF 95 F3 9E CA 79 EE

$x[16...31]$ =0B AA 53 6A D7 A8 21 E8 18 3E CA 0C 9D E2 8F 3C

Subround 2:

$x[0...15]$ =31 75 20 47 41 0A 0D 3B 48 5B 25 2B 37 68 6B 45

$x[16...31]$ =37 24 3D 1C 7C 4E 14 39 78 2A 0C 1F 4A 1C 51 61

Subround 3:

$x[0...15]$ =06 33 2D 03 23 3F 0A 12 1A 0E 24 1F 21 09 1F 0F

$x[16...31]$ =3B 30 23 34 17 1C 34 35 33 02 2C 26 2D 1F 28 12

Subround 4:

$x[0...15]$ =14 07 07 02 1F 1A 1C 1C 0C 18 0F 17 09 02 00 00

$x[16...31]$ =05 0F 10 04 13 12 0C 16 09 06 18 0F 05 06 1D 1C

Subround 5:

$x[0...15]$ =0A 08 07 0A 09 05 06 06 0B 0A 0D 00 02 06 0F 0F

$x[16...31]$ =04 06 05 01 02 05 05 0D 00 05 03 0C 03 0A 00 03

Bits from Bytes:

10101000 01111010 10010101 01100110 10111010 11010000 00100110 11111111
01000110 01010001 00100101 01011101 00000101 00111100 00111010 00000011

Permutation:

$x[0...15]$ =0A 08 07 0A 09 05 06 06 0B 0A 0D 00 02 06 0F 0F

$x[16...31]$ =A4 55 A6 7E 81 DA 68 D2 1E 6D 22 13 3F 25 15 78

Round 3 0f 8

$x[0...15]$ =52 A1 B0 D7 F3 81 6E C6 E7 23 28 97 F1 70 A8 1A Load Load K_i

$x[16...31]$ =A4 55 A6 7E 81 DA 68 D2 1E 6D 22 13 3F 25 15 78

Compression: Perform Substitutions

Subround 1:

$x[0...15]$ =A7 A9 F0 61 09 B3 74 7E F4 F0 CA C8 89 8F 61 0C

$x[16...31]$ =23 99 70 4C E8 6B 01 F5 2B 59 65 F7 25 02 57 CE

Subround 2:

$x[0...15]$ =1D 1E 7B 11 04 51 29 29 4C 4C 48 0F 28 34 38 27

$x[16...31]$ =38 4A 5D 5D 1C 43 4A 6B 71 62 3C 1D 34 06 00 45

Subround 3:

$x[0...15]$ =01 30 0A 1E 10 2F 12 0B 0C 08 29 3C 30 18 22 2A

$x[16...31]$ =1F 1D 35 06 33 11 15 01 2E 13 27 03 1C 00 25 07

Subround 4:

$x[0...15]$ =1B 0B 18 16 1A 01 0C 02 08 01 05 18 1A 0B 1D 0E

$x[16...31]$ =13 02 1E 01 03 1E 17 1A 0A 10 06 02 1C 06 03 17

Subround 5:

$x[0...15]$ =03 0C 01 0B 0B 00 0A 08 0E 03 00 0A 0A 0C 06 05

$x[16...31]$ =02 05 0F 0D 0C 01 07 05 0E 01 0E 0D 05 0F 03 0D

Bits from Bytes:

00111100 00011011 10110000 10101000 11100011 00001010 10101100 01100101
00100101 11111101 11000001 01110101 11100001 11101101 01011111 00111101

Permutation:

$x[0...15]$ =03 0C 01 0B 0B 00 0A 08 0E 03 00 0A 0A 0C 06 05

$x[16...31]$ =21 0D E2 B5 A4 7E 8D 68 7A FB D1 79 F5 8D 2D 25

.

.

.

Round 8 0f 8

$x[0...15]$ =52 A1 B0 D7 F3 81 6E C6 E7 23 28 97 F1 70 A8 1A Load K_i

$x[16...31]$ =57 5A 69 36 4F 19 F0 23 DA DE 7F DA DE DD 28 1E F1

Compression: Perform Substitutions

Subround 1:

$x[0...15]$ =B1 C9 48 29 38 59 D7 C1 A7 FC 6B 5C 44 68 4D F0

$x[16...31]$ =0A 87 EC 56 B3 A4 E4 7D D9 51 71 FB 52 19 B0 40

Subround 2:

$x[0...15]$ =23 7D 0B 26 76 7E 71 73 12 67 5A 37 35 4F 5A 57

$x[16...31]$ =2A 7E 79 65 03 1F 17 77 25 33 38 54 45 41 26 5D

Subround 3:

$x[0...15]$ =2B 35 13 33 27 25 3B 38 3F 31 34 23 2E 16 34 2A

$x[16...31]$ =3E 27 03 1A 11 0F 33 02 3B 1F 15 34 2B 03 1C 31

Subround 4:

$x[0...15]$ =14 12 02 03 03 1B 13 1D 02 1F 0C 01 03 0F 1C 03

$x[16...31]$ =19 12 13 1D 1F 1E 1B 14 08 17 09 0C 1A 08 0C 1F

Subround 5:

$x[0...15]$ =05 08 05 07 06 0C 02 04 0F 04 0D 066 0C 00 0A 09

$x[16...31]$ =0D 01 02 04 09 0B 04 0E 03 07 0C 0F 0E 0B 0E 02

Bits from Bytes:

01011000 01010111 01101100 00100100 11110100 11010110 11000000 10101001
11010001 00100100 10011011 01001110 00110111 11001111 11101011 11100010

No Permutation is Performed on Round 8

End of 8 Rounds

Output: 58 57 6C 24 92 6D 38 DF 3F AF 88 00

SRES: 58 57 6C 24

K_c : 92 6D 38 DF 3F AF 88 00 Cipher Key for encryption

K_c : 10010010 01101101 00111000 11011111 00111111 10101111 10001000 00000000

The SRES is then sent back to the BTS for verification. Once the SRES has been compared with the SRES computed by the AuC the mobile user is then authenticated. The Cipher Key K_c is then used in the encryption of over-the-air communications using the A5/1 or A5/2 algorithms depending on the region.

3.1.2 COMP128 Attacks

The most well known attack of COMP128 came from the same people who reversed engineered the leaked code: Wagner, Goldberg, and Briceno. Wagner and Goldberg were Berkley researchers and Briceno was the Director of the Smartcard Developers Association. The attack is called the WGB attack deriving its name from the discoverers. The description of the attack comes from Handschuh and Paillier from the ENST Computer Science Department and Gemplus Cryptography Department located in France [21]. This attack is a chosen-plaintext attack which means that it assumes the attacker has the capability to choose any plaintext they want and obtain the resulting ciphertext. This chosen-plaintext attack can be done with a SIM card reader and a computer with the COMP128 algorithm. There are two different situations for carrying out a chosen-plaintext attack on a mobile phone. The first situation being if someone has physical access to your SIM card and secondly using an over-the-air method.

The COMP128 algorithm being a one-way compression function, or commonly referred to as a hash function, is vulnerable to a type of attack called the birthday attack. The birthday attack gets its name from the birthday problem in probability theory. The birthday problem deals with finding the probability of having two individuals with the same birthday out of a set of n randomly chosen individuals. If you count February 29 there are 366 possible birthdays guaranteeing a matching pair when $n = 367$. The surprising result of the birthday problem is that 99% probability occurs with just $n = 57$ and 50% probability with $n = 23$. This can be interpreted as a 50% probability in a class of 23 people that two random people will share the same birthday. The birthday attack uses this higher probability of finding two matching pairs among a fixed number of permutations to help find a collision. A collision occurs when two randomly chosen inputs produce the same output. Therefore finding a collision in the COMP128 function results in finding two different messages M_1 and M_2 such that COMP128(M_1)=COMP128(M_2). Note though that the attacker usually has no choice over choosing M_1 or M_2. A collision is possible in a compression function as we are taking a larger domain and mapping it to a smaller range (pigeonhole principal). Using the probability model of the birthday problem and applying it to a compression function a compression of n bits results in a collision occurring in $2^{n/2}$ attempts by brute force. This number is referred to as the birthday bound.

The WGB attack uses the first scenario when the attacker has physical access to the SIM card.The attack exploits the butterfly structure of the compression rounds. In each of the 5 subrounds of compression the 32-byte input is divided into equal sections. In round 1 the 32-byte input is divided into two 16-byte inputs. In round 2 the 32-byte input is divided into four 8-byte sections. In round 3 we have eight 4-byte sections, round 4 sixteen 2-byte sections, and finally round 5 thirty-two 1-byte sections. In each subround of compression there is an associated lookup table with two input bytes used to calculate the output byte. Therefore the resulting output byte for each subround depends on two input bytes. Examining the output of the second subround of compression bytes $i, i + 8, i + 16, i + 24$ are dependent only upon the COMP128 input bytes $i, i + 8, i + 16, i + 24$. This is referred to as a narrow pipe. Since those specific 4 bytes of output depend on the same 4 bytes of input it is referred to as a pipe of width 4 bytes. An illustration of the narrow pipe is given in Figure 3.1. Since the first 16-bytes of input for COMP128 represent the secret key K_i, bytes in position i and $i + 8$ for $0 \leq 7$ represent K_i. The bytes $i + 16$ and $i + 24$ for $0 \leq 7$ represent the RAND . The attacker does not know the secret key K_i but can query the SIM card with a RAND challenge (chosen plaintext) and observe the output (ciphertext). The attack begins by querying the SIM card with a RAND and keeps querying the SIM only changing the bytes in position $i + 16$ and $i + 24$ of the COMP128 input. Thus we are only changing the bytes of $RAND[i]$ and

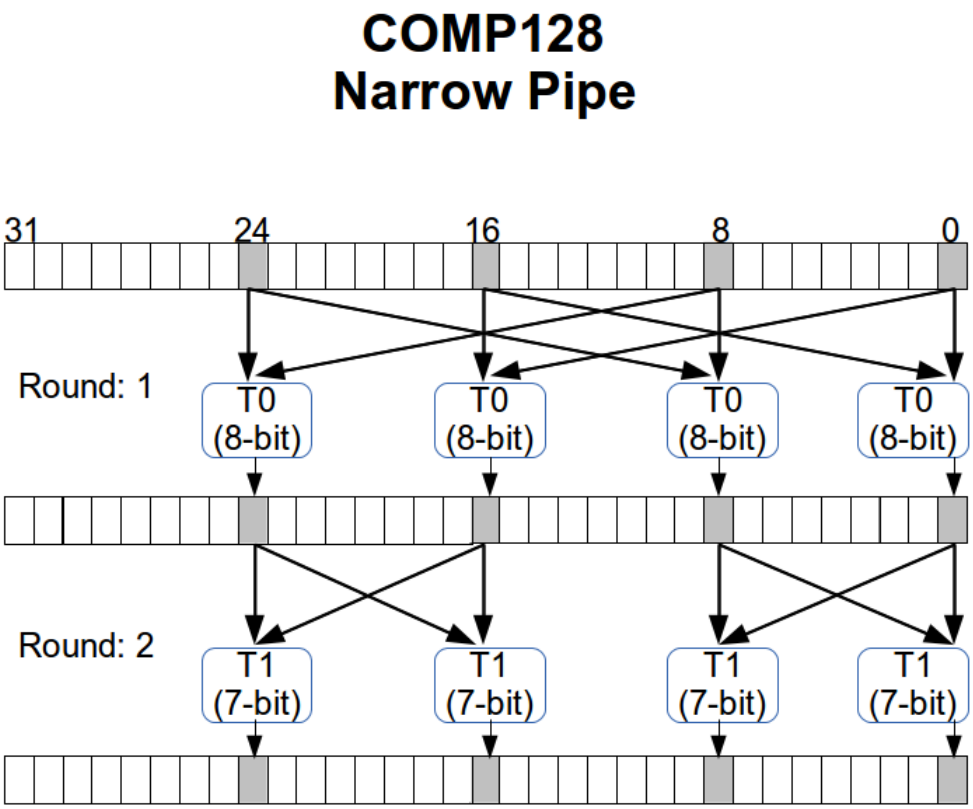

Figure 3.1. COMP128 narrow pipe.

$RAND[i+8]$ while observing the output of COMP128. The SIM card receives and inputs the RAND with the hidden K_i into COMP128 displaying only the output. Note that the attacker never sees the K_i. If a collision occurs after the second round due to the narrow pipe it will spread throughout the rest of the algorithm causing a collision in the COMP128 output as well. This follow through of the collision is considered a major weakness. The attacker keeps choosing different pairs of RAND$[i]$ and RAND$[i+8]$ until there is a collision in the output of COMP128. Once a pair is found that produces a collision the attacker fixes those two values of RAND. Now that we have two values $RAND_1 \neq RAND_2$ such that COMP128[$K_i||RAND_1$]=COMP128[$K_i||RAND_2$] with the only differences being the bytes in position $i, i+8$ of the RAND can the attacker move to the next step. The attacker then disregards the SIM card and initiates COMP128 on their computer. The attacker now chooses a random K_i varying only the bytes in position $K_i[i]$ and $K_i[i+8]$. When a pair of $K_i[i]$ and $K_i[i+8]$ is found such that the output for COMP128 produces a collision with the output for COMP128[$K_i||RAND_1$]=COMP128[$K_i||RAND_2$] two key bytes have been found. By repeating this process for $0 \leq i \leq 7$ all of the bytes of the secret key K_i can be found.

Therefore we have four values $i, i+8, i+16, i+24$ for each $0 \leq i \leq 7$ to find. Since we are looking for a collision after the second subround of compression and each output in the second subround of compression results in 7-bit values we are looking for a 7-bit compression value. According to the birthday attack a compression of n bits results in a collision occurring in $2^{n/2}$ attempts by brute force. Thus we can expect a collision of 4 bytes (7-bit values each) to occur after $2^{4*7/2} = 2^{14}$ queries. This process is repeated 8 times for $0 \leq i \leq 7$. The attack therefore requires $8 * 2^{14} = 2^{17}$ queries. A SIM card reader would require 8 hours to perform 2^{17} queries. To address this attack new SIM cards are designed to shut down and stop working after 2^{16} queries in an attempt to stop this attack. Although users with the older SIM cards are still left vulnerable.

Once the secret key is discovered the SIM card can be cloned allowing the attacker to make calls or eavesdrop on the users phone conversations. Most cloned SIM cards are used for eavesdropping purposes as the GSM network is designed to disable phones that are making simultaneous calls with the same secret key. This attack is not very practical as it would be hard for the attacker to gain possession of your phone for the 8 hours required. The other scenario of an over-the-air attack is also possible but would require an investment of around $10,000 worth of equipment back in 1998. The test was never carried out as it was illegal to build such equipment. The over-the-air attack would work by impersonating a BTS. Thus it is referred to as the False Base Station attack. Programing codes for the GSM network were leaked when an Italian carrier in the 1990s went bankrupt and posted the codes on the SourceForge website leaving it there for over 4 years before being taken down. With this code attackers were able to build the software necessary to make a fake base station. The ME is designed to connect to the BTS that emits the strongest signal. Thus an attacker would have to build a false base station that emits a stronger signal than the real BTS. To emit a signal strong enough to take over the real signal the attack would have to take advantages of places were signal strength from the BTS was already weak. These places of attack include subways, elevators, and inside offices. A sign to the user that an over-the-air attack was taking place would be a rapid battery drain as the phone would be communicating for around 8 hours with the attacker. These attacks were not very practical due to the time required to extract the key but they did demonstrate the weaknesses of the COMP128 algorithm. SAGE, the closed door designers of COMP128, released a press announcement after the attack was published noting that COMP128 was just a recommended algorithm for the carriers to use and not a requirement. Although the Smartcard Developer Association announced that they have not been able to identify one network in Europe or the U.S. that used an algorithm other than COMP128. By 1998 the GSM Association estimated that there were 80 million GSM users worldwide so a change in authentication algorithms would require the replacement of 80

million SIM cards along with a software upgrade to the networks. Because of this reason no changes to the system were made until 2001 when a patched version of COMP128 referred to as COMP128-2 was released and implemented in newer SIM cards. The algorithm changed the way the SRES was calculated but was never released to the public.

In 2002 researches at IBM developed an improved attack referred to as a Partition Attack. The partition attack is a side-channel attack which is based on information found by examining the physical reactions of the cryptographic system such as timing, power consumption, electromagnet leaks, or sound that can all lead to sources of extra information. This information is leaked through a side-channel thus giving the attack its name. The description of the attack comes from three IBM researches J. Rao, P. Rohatgi, H. Scherzer and S. Tinguely from the Swiss Federal Institute of Technology[22]. The researches noted that the first compression table of COMP128 has 512 values. To implement COMP128 on an 8-bit SIM card would prove difficult as the indexing of the first compression table would require a 9-bit value. Thus a programmer is likely to divide the 512 lookup table into two tables of 256 values which can be implemented with 8-bit indexes. Each partition is stored in a different part of memory and the researchers noted that there was a different power signal depending on which partition of the table was being accessed. They could use this leaked information to help eliminate possible key bits depending on which partition was accessed in each iteration of COMP128. Since the partition is used with the first compression table in round one the table inputs are closely related to the COMP128 inputs. The researchers were able to determine the 128-bit key with as few as 8 chosen plaintexts if they could adaptively choose the inputs or 1000 attempts if random inputs were used. This attack narrowed the amount of time from 8 hours to less than one minute if the attacker had access to your SIM card. These attacks are all possible due to the fact that authentication only goes one way. The user never authenticates the network leaving it vulnerable to these type of attacks. The 3G GSM based network UMTS addresses these attacks by requiring authentication both ways between user and network.

3.2 Over-the-Air Encryption: A5/1 & A5/2

The two algorithms used to encrypt over the air communications in 2G networks are referred to as A5/1 and A5/2. These algorithms are stream cipher that were designed in secret but leaked to the public. Note that A5/0 refers to the case where no encryption is used.

3.2.1 A5/1 Algorithm

The A5/1 algorithm consists of three Linear Feedback Shift Registers (LFSR). The first register R1 has a length of 19 bits with the rightmost bit referred to as position 0. The other two registers R2 and R3 have lengths of 22 and 23 bits respectively. Each register has a clocking bit and tapping bits. The tapping bits are XORed together and are used to produce

the entry for bit position 0 after a shift has occurred. The tapping bits for R1 are at positions 18, 17, 16, and 13. The tapping bits for R2 are at positions 21 and 20. R3 has tapping bits at position 22, 21, 20, and 7. The clocking bit is responsible for the shifting of the registers. The clocking bit for R1, R2, and R3 occur at positions 8, 10, and 10 respectively. A majority rule function is used on the clocking bits to determine if a register is clocked. This stop/go process is done with each cycle. Since the stop/go process relies on a majority rule function there will always be at least two registers being clocked for each cycle. If a register is clocked the tapping bits are XORed and the value is used to produce the bit in position 0. For example, if the clocking bits for R1, R2, and R3 are 1, 0, 1 then R1 and R3 would be clocked. If the clocking bits were 1, 0, 0 then R2 and R3 would be clocked. Regardless of whether a register is clocked or not the leftmost bit: 18, 21, and 22 are XORed together and the result is used to form the keystream. An illustration of the A5/1 algorithm is given in Figure 3.2. The input for

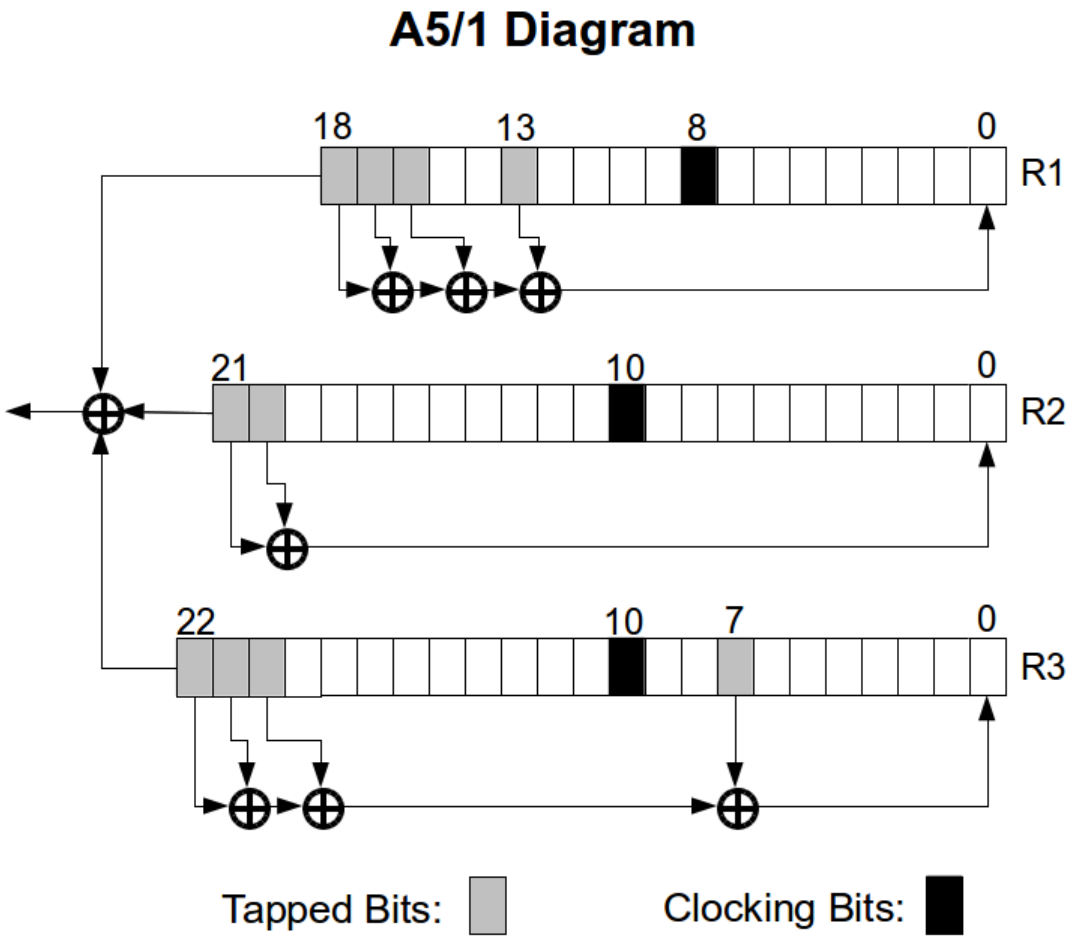

Figure 3.2. Diagram of A5/1 algorithm.

the A5/1 algorithm consists of a 64-bit Cipher Key K_c generated from the COMP128

algorithm along with a publicly known 22-bit Frame Number F_n. A frame number is used so that a new keystream will be produced for each frame making each keystream unique. The frame number increases incrementally while the Cipher Key remains the same. The Cipher Key is not changed until the MS is authenticated again. This means that the same Cipher Key can be used for days at a time. The output of A5/1 consists of 228-bits. The first 114-bits are the keystream for encrypting communication from the MS to the BTS. The last 114-bits of keystream are used to encrypt the communication from the BTS to the MS. The generation of the 228-bit keystream consists of four main stages[16]:

- **Stage 1:** All three registers are set to zero. Then for 64 cycles the session key is XORed into the zero position of the registers ordered by the least significant bit of each byte. Thus the 64-bit cipher key $K_c[0...63]$ would be XORed into the registers by the following bit order: $K_c[7]$, $K_c[6]$, $K_c[5]$,...,$K_c[0]$, $K_c[15]$, $K_c[14]$, and so on. The stop/go clock control is ignored for this stage and every register is clocked with each cycle.

- **Stage 2:** For 22 cycles the 22-bit frame number is XORed in the zero position of each register. The input of the frame number is by least significant bit to most significant bit. Thus the frame number $F_n[0...21]$ is XORed into the registers in the following order: $F_n[21]$, $F_n[20]$, $F_n[19]$, ...$F_n[0]$. The stop/go clock control is ignored for this stage and every register is clocked with each cycle.

- **Stage 3:** For 100 cycles the three registers are all clocked with the stop/go clock control based on the majority function. This stage is considered the priming of the A5/1 algorithm mixing together the cipher key K_c and the frame number F_n. Note that the XORed output from the registers is discarded.

- **Stage 4:** For 228 cycles the three registers are all clocked with the stop/go clock control. With each cycle the XOR of the leftmost register bits R1[18], R2[21], and R3[22] produce one bit of the 228-bit keystream. By the end of stage 4 the A5/1 algorithm has produced the 114-bit + 114-bit keystream needed to encrypt over the air communications between the MS and the BTS.

A new keystream is created for every frame in intervals of 4.6 milliseconds. Each one of these frames produces a publicly known frame counter F_n. Once the number of frames reaches 8,388,608 the frame number cycles back to the beginning and resumes counting again.

3.2.2 A5/1 Example

An example of the A5/1 algorithm is provided below using a random frame number along with the cipher key produced above from the COMP128 output. Note that the Matlab code used for this example is included in the appendix.

K_c : 92 6D 38 DF 3F AF 88 00 Cipher Key from COMP128

K_c : 10010010 01101101 00111000 11011111 00111111 10101111 10001000 00000000

Frame Number: 01 34 (Hex) Value between $0 \leq F_n < 2^{23}$

Frame Number: 00 0000 0000 0001 0011 0100 Converted to 22-bits

A5/1

Stage 1 (64 Cycles) No Clock Control

Cycle 1

K_c=1001001**0** 01101101 00111000 11011111 00111111 10101111 10001000 00000000

0	**0**	**0**	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0				R1
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		R2
0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	R3

Cycle 2

K_c=1001001**1** 01101101 00111000 11011111 00111111 10101111 10001000 00000000

0	**0**	**0**	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	1				R1
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		R2
0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	1	R3

Cycle 3

K_c=10010**0**10 01101101 00111000 11011111 00111111 10101111 10001000 00000000

0	**0**	**0**	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	1	0				R1
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	R2
0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	1	0	R3

Cycle 4

K_c=1001**0**010 01101101 00111000 11011111 00111111 10101111 10001000 00000000

0	**0**	**0**	0	0	**0**	0	0	0	0	0	0	0	0	0	0	1	0	0				R1
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	R2
0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	1	0	0	R3

Cycle 5

K_c=100**1**0010 01101101 00111000 11011111 00111111 10101111 10001000 00000000

0	**0**	**0**	0	0	**0**	0	0	0	0	0	0	0	0	0	1	0	0	1						R1
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1			R2
0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	1	0	0	1		R3

.
.
.

Cycle 62

K_c=10010010 01101101 00111000 11011111 00111111 10101111 10001000 00**0**00000

0	**0**	**1**	1	0	**1**	0	1	0	1	0	1	1	0	1	0	1	1	1						R1
0	**1**	1	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0			R2
1	**0**	**1**	0	1	0	0	1	1	0	0	0	0	0	1	**1**	1	0	1	0	1	0	1		R3

Cycle 63

K_c=10010010 01101101 00111000 11011111 00111111 10101111 10001000 0**0**000000

0	**1**	**1**	0	1	**0**	1	0	1	0	1	1	0	1	0	1	1	1	0						R1
1	**1**	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1			R2
0	**1**	**0**	1	0	0	1	1	0	0	0	0	0	1	1	**1**	0	1	0	1	0	1	1		R3

Cycle 64

K_c=10010010 01101101 00111000 11011111 00111111 10101111 10001000 **0**0000000

1	**1**	**0**	1	0	**1**	0	1	0	1	1	0	1	0	1	1	1	0	0						R1
1	**1**	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	0			R2
1	**0**	**1**	0	0	1	1	0	0	0	0	0	1	1	1	**0**	1	0	1	0	1	1	0		R3

Stage 1 Output:

R1: 110 1010 1011 0101 1100

R2: 11 0111 1101 0111 1111 0010

R3: 101 0011 0000 0111 0101 0110

Stage 2 (22 Cycles) No Clock Control

Cycle 1

F_n=00 0000 0000 0001 0011 010**0**

1	**0**	**1**	0	1	**0**	1	0	1	1	0	1	0	1	1	1	0	0	1	R1

1	**0**	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	0	0	R2
0	**1**	**0**	0	1	1	0	0	0	0	0	1	1	1	0	**1**	0	1	0	1	1	0	0	R3

Cycle 2

F_n=00 0000 0000 0001 0011 01**0**0

0	**1**	**0**	1	0	**1**	0	1	1	0	1	0	1	1	1	0	0	1	0				R1
0	**1**	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	0	0	1	R2
1	**0**	**0**	1	1	0	0	0	0	0	1	1	1	0	1	**0**	1	0	1	1	0	0	0	R3

Cycle 3

F_n=00 0000 0000 0001 0011 0**1**00

1	**0**	**1**	0	1	**0**	1	1	0	1	0	1	1	1	0	0	1	0	1				R1
1	**1**	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	0	0	1	0	R2
0	**0**	**1**	1	0	0	0	0	0	1	1	1	0	1	0	**1**	0	1	1	0	0	0	0	R3

.
.
.

Cycle 20

F_n=00 **0**000 0000 0001 0011 0100

0	**1**	**0**	1	0	**1**	1	1	1	1	0	0	0	0	0	0	1	0	0				R1
1	**0**	0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	R2
1	**1**	**0**	0	0	0	0	1	0	0	0	1	0	1	0	**0**	0	0	1	1	1	1	0	R3

Cycle 21

F_n=0**0** 0000 0000 0001 0011 0100

1	**0**	**1**	0	1	**1**	1	1	1	0	0	0	0	0	0	1	0	0	0				R1
0	**0**	1	0	0	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	R2
1	**0**	**0**	0	0	0	1	0	0	0	1	0	1	0	0	**0**	0	1	1	1	1	0	0	R3

Cycle 22

F_n=**0**0 0000 0000 0001 0011 0100

0	**1**	**0**	1	1	**1**	1	1	0	0	0	0	0	0	1	0	0	0	1							R1
0	**1**	0	0	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0				R2
0	**0**	**0**	0	0	1	0	0	0	1	0	1	0	0	0	**0**	1	1	1	1	0	0	1			R3

Stage 2 Output:

R1: 010 1111 1000 0001 0001

R2: 01 0011 0101 1000 0001 0110

R3: 000 0010 0010 1000 0111 1001

Stage 3 (100 Cycles) C=Register Clocked

Cycle 1

1	**0**	**1**	1	1	**1**	1	0	0	0	0	0	0	1	0	0	0	1	0							**C** R1
1	**0**	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	1	0	1	1	0	1			**C** R2
0	**0**	**0**	0	1	0	0	0	1	0	1	0	0	0	0	**1**	1	1	1	0	0	1	0			**C** R3

Cycle 2

0	**1**	**1**	1	1	**1**	0	0	0	0	0	0	1	0	0	0	1	0	1							**C** R1
0	**0**	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	1	1				**C** R2
0	**0**	**0**	1	0	0	0	1	0	1	0	0	0	0	1	**1**	1	1	0	0	1	0	1			**C** R3

Cycle 3

1	**1**	**1**	1	1	**0**	0	0	0	0	0	1	0	0	0	1	0	1	1							**C** R1
0	**1**	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	1	1	0				**C** R2
0	**0**	**1**	0	0	0	1	0	1	0	0	0	0	1	1	**1**	1	0	0	1	0	1	1			**C** R3

.

.

.

Cycle 98

0	**1**	**0**	1	0	**1**	1	1	1	0	0	1	1	0	0	1	0	0	0							R1
1	**0**	0	1	0	0	0	1	1	0	1	0	1	1	0	1	0	0	1	1	0	1				**C** R2
0	**0**	**1**	1	1	1	0	0	1	1	0	1	1	0	0	**1**	0	1	0	1	1	0	1			**C** R3

Cycle 99

1	**0**	**1**	0	1	**1**	1	1	0	0	1	1	0	0	1	0	0	0	0							**C** R1

0	**0**	1	0	0	0	1	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	**C** R2
0	**0**	**1**	1	1	1	0	0	1	1	0	1	1	0	0	**1**	0	1	0	1	1	0	1	R3

Cycle 100

0	**1**	**0**	1	1	**1**	1	0	0	1	1	0	0	1	0	0	0	0	1				**C** R1
0	**1**	0	0	0	1	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	**C** R2
0	**1**	**1**	1	1	0	0	1	1	0	1	1	0	0	1	**0**	1	0	1	1	0	1	0	**C** R3

Stage 3 Output:

R1: 010 1111 0011 0010 0001

R2: 01 0001 1010 1101 0011 0110

R3: 011 1100 1101 1001 0101 1010

Stage 4 (114+114 Cycles) C=Register Clocked

Cycle 1

1	**0**	**1**	1	1	**1**	0	0	1	1	0	0	1	0	0	0	0	1	0				**C** R1
1	**0**	0	0	1	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	**C** R2
0	**1**	**1**	1	1	0	0	1	1	0	1	1	0	0	1	**0**	1	0	1	1	0	1	0	R3

Keystream: **0**

Cycle 2

0	**1**	**1**	1	1	**0**	0	1	1	0	0	1	0	0	0	0	1	0	1				**C** R1
0	**0**	0	1	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1	**C** R2
1	**1**	**1**	1	0	0	1	1	0	1	1	0	0	1	0	**1**	0	1	1	0	1	0	0	**C** R3

Keystream: 0**1**

Cycle 3

1	**1**	**1**	1	0	**0**	1	1	0	0	1	0	0	0	0	1	0	1	0				**C** R1
0	**0**	0	1	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1	R2
1	**1**	**1**	0	0	1	1	0	1	1	0	0	1	0	1	**0**	1	1	0	1	0	0	0	**C** R3

Keystream: 01**0**

.

.

.

Cycle 226

1	**0**	**1**	1	1	**0**	0	0	1	0	1	1	1	0	0	1	1	1	0					**C** R1
1	**0**	1	0	1	0	1	0	0	1	1	1	1	0	0	1	0	0	1	1	0	1		R2
0	**1**	**0**	0	0	1	1	0	0	0	0	0	0	1	1	**0**	0	1	1	1	0	0	0	**C** R3

Keystream: 01000101 10010000 10101010 01001100 11111100 10011011 00000100
10111001 00000111 10011010 00001001 10011000 01101100 11101100 11
01010110 01010011 11000111 11101001 11001011 00100110 10101100
11100011 10100111 10111100 00100101 11110010 11110001 01000110

Cycle 227

0	**1**	**1**	1	0	**0**	0	1	0	1	1	1	0	0	1	1	1	0	0					**C** R1
0	**1**	0	1	0	1	0	0	1	1	1	1	0	0	1	0	0	1	1	0	1	1		**C** R2
0	**1**	**0**	0	0	1	1	0	0	0	0	0	0	1	1	**0**	0	1	1	1	0	0	0	R3

Keystream: 01000101 10010000 10101010 01001100 11111100 10011011 00000100
10111001 00000111 10011010 00001001 10011000 01101100 11101100 11
01010110 01010011 11000111 11101001 11001011 00100110 10101100
11100011 10100111 10111100 00100101 11110010 11110001 01000110 **0**

Cycle 228

1	**1**	**1**	0	0	**0**	1	0	1	1	1	0	0	1	1	1	0	0	0					**C** R1
1	**0**	1	0	1	0	0	1	1	1	1	0	0	1	0	0	1	1	0	1	1	1		**C** R2
0	**1**	**0**	0	0	1	1	0	0	0	0	0	0	1	1	**0**	0	1	1	1	0	0	0	R3

Keystream: 01000101 10010000 10101010 01001100 11111100 10011011 00000100
10111001 00000111 10011010 00001001 10011000 01101100 11101100 11
01010110 01010011 11000111 11101001 11001011 00100110 10101100
11100011 10100111 10111100 00100101 11110010 11110001 01000110 0**0**

Output from A5/1 Algorithm

Keystream from MS to BTS:

01000101 10010000 10101010 01001100 11111100 10011011 00000100
101110010 00001111 00110100 00010011 00110000 11011001 1101100 11
Hexadecimal Conversion: 45 90 AA 4C FC 9B 04 B9 07 9A 09 98 6C EC C0

Keystream from BTS to MS:

01010110 01010011 11000111 11101001 11001011 00100110 10101100
11100011 10100111 10111100 00100101 11110010 11110001 01000110 00

Hexadecimal Conversion: 56 53 C7 E9 CB 26 AC E3 A7 BC 25 F2 F1 46 00

3.2.3 A5/1 Attacks

One of the major weaknesses of the A5/1 algorithm was the deliberate setting of the last 10-bits of the Cipher Key produced by the COMP128-1/COMP128-2 algorithm to zero. The result of this is that the Cipher Key used in A5/1 is reduced from 64-bits to effectively 54-bits. The time complexity of a brute force attack is thus reduced from 2^{64} to 2^{54}. This would take too long for real time eavesdropping but if one were to record the intercepted communication between the MS and BTS it would require about 250 hours with a Pentium based computer from 1998. Later an upgraded version COMP128-3 was released that expanded the 54-bit key to the full 64-bits. The two common types of attacks for the A5/1 algorithm are guess-and-determine attacks and time-memory-data tradeoff attacks. A description of these type of attacks comes from Gendrullis, Novotny, and Rupp from the Horst Gortz Institute for IT-Security in Germany [19]. Both forms of attack assume that at least 64-bits of consecutive keystream bits are known to the attacker.

A guess-and-determine attack works as the name suggests by guessing bits of the register and using the known keybits to determine the remaining register bits. Ross Anderson, a professor in Security Engineering at the University of Cambridge, devised an attack against A5/1 by completely guessing the bits in R1, R2, and half of R3. The second half of R3 is then derived from the known keystream bits. This attack had a worst-case scenario of 2^{52} operations needed to be performed to recover the Cipher Key. An improved guess-and-determine attack was proposed by J.D. Golic who guessed the lower half of each register and clocked the registers until the guessed bits ran out. Each output bit he was able to produce a linear equation in terms of the bits for the upper half of the three registers. This guessing the lower half process would continue until 64 linearly independent equations were obtained. The system of linear equations were then solved using Gaussian elimination. This attack could be completed in 2^{40} steps. Both of these attacks are not very practical in that real time eavesdropping would not be possible with the number of steps required to determine the key.

The other type of attack is a time-memory-data trade-off attack. This attack relies on precomputed data to reduce the time needed to determine the Cipher Key. One of the weaknesses of A5/1 is that the length of the registers is small enough that it is possible to precompute all the possible states of the three registers. Once a state is given all successive states are stored as well. Each register will have an array indexed by the state number in the succession, the clocking bit, and the output bit. This precomputation stage requires 2^{48} operations and memory requirements of about 300GB. Once the precomputation stage is

complete the next stage involves observing the keystream output for a given length of time. The tables are then used to try and match the string of keystream bits with the correct states of the registers with a success rate of 60%. The more keystream bits you have the faster the Cipher Key is recovered. If you are able to observe 2 minutes worth of keystream output it is possible to produce the Cipher Key within one second. If you only have 2 seconds worth of keystream bits then the attack will be able to find the Cipher Key within minutes. This type of attack is impractical due to the time it takes to complete the precomputational data and the unlikely fact of being able to observe 2 seconds worth of keystream bits. Observing 2 seconds of keystream bits amounts to having roughly 25,000-bits.

There is also special hardware available built specifically to attack stream ciphers with a keysize of 64-bits or less. One publicly known code-breaker machine is called COPACOBANA (Cost-Optimized Parallel Code Breaker) and comes with a price tag of $10,000. The machine is able to find the Cipher Key within 7 to 14 hours with only 64-bits of known Keystream. The COPACOBANA uses a guess-and-determine attack on A5/1. With the length and computation time required to implement these attacks real time eavesdropping is highly unlikely. Real time eavesdropping is accomplished by taking advantage of a serious security flaw in the GSM security architecture. The ME stores both the A5/1 (stronger) and A5/2 (weaker) algorithms to allow for encryption when roaming on networks that only support A5/2. The security flaw is that the Cipher Key used for the A5/1 algorithm is the same as the Cipher Key used for the A5/2 algorithm. If an attacker is able to force the phone into using the A5/2 algorithm in place of the A5/1 algorithm the Cipher Key can be found with relative ease compared to attacking the A5/1 algorithm. This attack method is explained below in the A5/2 attack section.

3.2.4 A5/2 Algorithm

While A5/1 was used in European countries and North America, a deliberate weakening of the A5/1 algorithm called the A5/2 algorithm was used elsewhere. The A5/1 algorithm is referred to as the ”stronger” algorithm while the A5/2 algorithm is referred to as the ”weaker” algorithm. The A5/2 algorithm was made weaker by requests of the intelligence agencies to ensure it was breakable. The intelligence agencies did not want a strong encryption algorithm to be used in the Middle East and this issue was the main driving force for creating the weaker algorithm.

The A5/2 algorithm consists of 4 LFSR referred to as R1, R2, R3, and R4. The first three registers R1, R2, and R3 all share the same tapping bits as in the A5/1 algorithm along with the same lengths of 19, 22, and 23 bits. The new fourth register R4 is 17 bits long with tapping bits at positions 11 and 16. For the A5/1 algorithm the clocking of the registers were

controlled by clocking bits located in each register. In A5/2 the clocking of registers R1, R2, and R3 are controlled by clocking bits located in R4. The clocking is determined by performing a majority function for bits R4[3], R4[7], and R4[10]. If the majority bit is the same as R4[10], then the first register R1 is clocked. Similarly R2 and R3 are clocked if the majority bit is the same as R4[3] or R4[7]. The clocking of R4 is done after the majority function is computed and R1, R2, and R3 are checked for clocking.

The output bit for A5/2 depends on the XOR of 6 bits. The first three bits come from the original output source of the A5/1 algorithm: the XOR of the three register bits R1[18]$\oplus$R2[21]$\oplus$R3[22]. The A5/2 algorithm adds an additional 3 bits to be XORed by using a majority function for each register. The majority function for R1 is computed by taking the majority of bits R1[12], R1[14]$\oplus$1, and R1[15]. The majority function for R2 relies on bits R2[9], R2[13], and R2[16]$\oplus$1. The majority function for R3 relies on bits R3[13]$\oplus$1, R3[16], and R3[18]. The diagram for the A5/2 algorithm is given in Figure 3.3. The input for the A5/2 algorithm is the same as the A5/1 algorithm. It consists of the same 128-bit cipher key produced by the A8 algorithm along with the 22-bit frame number F_n. The output consists of 114+114-bits of keystream. The first 114-bits of keystream are used for encrypting MS to BTS communications. The second block of 114-bits are used to encrypt BTS to MS communications. The A5/2 algorithm has four main stages[14]:

- **Stage 1:** All four registers are set to zero. Then for 64 cycles the cipher key is XORed into the zero position of the registers ordered by the least significant bit of each byte. Thus the 64-bit cipher key $K_c[0...63]$ would be XORed into the registers by the following bit order: $K_c[7]$, $K_c[6]$, $K_c[5]$,...,$K_c[0]$, $K_c[15]$, $K_c[14]$, and so on. The stop/go clock control is ignored for this stage and every register is clocked with each cycle.

- **Stage 2:** For 22 cycles the 22-bit frame number is XORed in the zero position of each register. The input of the frame number is by least significant bit to most significant bit. Thus the frame number $F_n[0...21]$ is XORed into the registers in the following order: $F_n[21]$, $F_n[20]$, $F_n[19]$, ...$F_n[0]$. The stop/go clock control is ignored for this stage and every register is clocked with each cycle.

- **Stage 3:** Set the bits in position R1[15], R2[16], R3[18], and R4[10] to be 1. For 99 cycles R1, R2, and R3 are all clocked with the stop/go clock control based on the clocking unit associated with R4. This stage is considered the initialization of the A5/2 algorithm mixing together the cipher key K_c and the frame number F_n. Note that the XORed output from R1, R2, R3, and their associated majority functions are discarded.

- **Stage 4:** For 228 cycles R1, R2, and R3 are all clocked with the stop/go clock control associated with R4. With each cycle the XOR of the leftmost register bits R1[18], R2[21], R3[22], and their associated majority functions produce one bit of the 228-bit

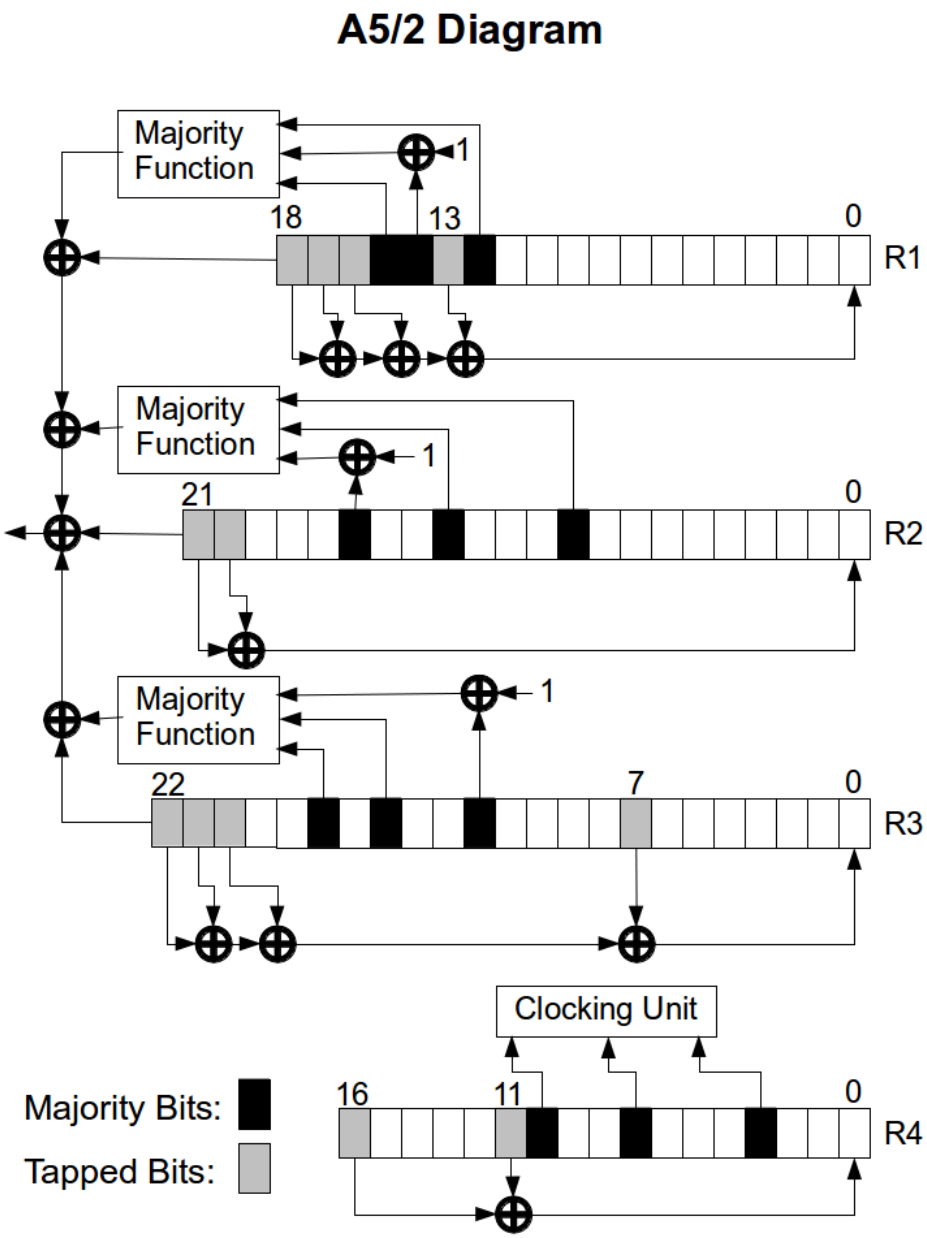

Figure 3.3. Diagram of A5/2 algorithm.

keystream. By the end of stage 4 the A5/2 algorithm has produced the 114-bit + 114-bit keystream needed to encrypt over the air communications between the MS and the BTS.

3.2.5 A5/2 Example

Using the same frame number and cipher key inputted into the A5/1 algorithm above an example of the A5/2 algorithm is provided below. Note that the Matlab code used for this example is included in the appendix.

K_c : 92 6D 38 DF 3F AF 88 00 Cipher Key from COMP128

K_c : 10010010 01101101 00111000 11011111 00111111 10101111 10001000 00000000

Frame Number: 01 34 (Hex) Value between $0 \leq F_n < 2^{23}$

Frame Number: 00 0000 0000 0001 0011 0100 Converted to 22-bits

A5/2

Stage 1 (64 Cycles) No Clock Control

Cycle 1

K_c=1001001**0** 01101101 00111000 11011111 00111111 10101111 10001000 00000000

0	**0**	**0**	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0					R1
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		R2
0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	R3
0	0	0	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0						R4

Cycle 2

K_c=1001001**1** 01101101 00111000 11011111 00111111 10101111 10001000 00000000

0	**0**	**0**	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	1					R1
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		R2
0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	1	R3
0	0	0	0	0	**0**	0	0	0	0	0	0	0	0	0	0	1						R4

Cycle 3

K_c=1001**0**010 01101101 00111000 11011111 00111111 10101111 10001000 00000000

0	**0**	**0**	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	1	0				R1
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		R2
0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	1	0	R3
0	0	0	0	0	**0**	0	0	0	0	0	0	0	0	0	1	0						R4

.

.

.

Cycle 62

K_c=10010010 01101101 00111000 11011111 00111111 10101111 10001000 00**0**00000

0	**0**	**1**	1	0	**1**	0	1	0	1	0	1	1	0	1	0	1	1	1							R1
0	**1**	1	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0				R2
1	**0**	**1**	0	1	0	0	1	1	0	0	0	0	0	1	**1**	1	0	1	0	1	0	1			R3
0	0	0	1	0	**0**	0	0	0	0	0	0	1	1	1	0	1									R4

Cycle 63

K_c=10010010 01101101 00111000 11011111 00111111 10101111 10001000 0**0**000000

0	**1**	**1**	0	1	**0**	1	0	1	0	1	1	0	1	0	1	1	1	0							R1
1	**1**	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1				R2
0	**1**	**0**	1	0	0	1	1	0	0	0	0	0	1	1	**1**	0	1	0	1	0	1	1			R3
0	0	1	0	0	**0**	0	0	0	0	0	1	1	1	0	1	0									R4

Cycle 64

K_c=10010010 01101101 00111000 11011111 00111111 10101111 10001000 **0**0000000

1	**1**	**0**	1	0	**1**	0	1	0	1	1	0	1	0	1	1	1	0	0							R1
1	**1**	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	0				R2
1	**0**	**1**	0	0	1	1	0	0	0	0	0	1	1	1	**0**	1	0	1	0	1	1	0			R3
0	1	0	0	0	**0**	0	0	0	0	1	1	1	0	1	0	0									R4

Stage 1 Output:

R1: 110 1010 1011 0101 1100

R2: 11 0111 1101 0111 1111 0010

R3: 101 0011 0000 0111 0101 0110

R4: 0 1000 0000 0111 0100

Stage 2 (22 Cycles) No Clock Control

Cycle 1

F_n=00 0000 0000 0001 0011 010**0**

1	**0**	**1**	0	1	**0**	1	0	1	1	0	1	0	1	1	1	0	0	1							R1
1	**0**	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	0	0				R2
0	**1**	**0**	0	1	1	0	0	0	0	0	1	1	1	0	**1**	0	1	0	1	1	0	0			R3
1	0	0	0	0	**0**	0	0	0	1	1	1	0	1	0	0	0									R4

Cycle 2

F_n=00 0000 0000 0001 0011 01**0**0

0	**1**	**0**	1	0	**1**	0	1	1	0	1	0	1	1	1	0	0	1	0					R1
0	**1**	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	0	0	1		R2
1	**0**	**0**	1	1	0	0	0	0	0	1	1	1	0	1	**0**	1	0	1	1	0	0	0	R3
0	0	0	0	0	**0**	0	0	1	1	1	0	1	0	0	0	1							R4

Cycle 3

F_n=00 0000 0000 0001 0011 0**1**00

1	**0**	**1**	0	1	**0**	1	1	0	1	0	1	1	1	0	0	1	0	1					R1
1	**1**	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	0	0	1	0		R2
0	**0**	**1**	1	0	0	0	0	0	1	1	1	0	1	0	**1**	0	1	1	0	0	0	0	R3
0	0	0	0	0	**0**	0	1	1	1	0	1	0	0	0	1	1							R4

.
.
.

Cycle 20

F_n=00 **0**000 0000 0001 0011 0100

0	**1**	**0**	1	0	**1**	1	1	1	1	0	0	0	0	0	0	1	0	0					R1
1	**0**	0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1		R2
1	**1**	**0**	0	0	0	0	1	0	0	0	1	0	1	0	**0**	0	0	1	1	1	1	0	R3
0	1	0	1	1	**1**	1	1	1	1	1	0	0	1	0	0	0							R4

Cycle 21

F_n=0**0** 0000 0000 0001 0011 0100

1	**0**	**1**	0	1	**1**	1	1	1	0	0	0	0	0	0	1	0	0	0					R1
0	**0**	1	0	0	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1		R2
1	**0**	**0**	0	0	0	1	0	0	0	1	0	1	0	0	**0**	0	1	1	1	1	0	0	R3
1	0	1	1	1	**1**	1	1	1	1	0	0	1	0	0	0	1							R4

Cycle 22

F_n=**0**0 0000 0000 0001 0011 0100

0	**1**	**0**	1	1	**1**	1	1	0	0	0	0	0	0	1	0	0	0	1							R1
0	**1**	0	0	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0				R2
0	**0**	**0**	0	0	1	0	0	0	1	0	1	0	0	0	**0**	1	1	1	1	0	0	1			R3
0	1	1	1	1	**1**	1	1	1	0	0	1	0	0	0	1	0									R4

Stage 2 Output:

R1: 010 1111 1000 0001 0001

R2: 01 0011 0101 1000 0001 0110

R3: 000 0010 0010 1000 0111 1001

R4: 0 1111 1111 0010 0010

Stage 3 (99 Cycles) C=Register Clocked

Cycle 1

0	**1**	**0**	1	1	**1**	1	1	0	0	0	0	0	0	1	0	0	0	1							R1
1	**0**	0	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	1				**C** R2
0	**0**	**0**	1	1	0	0	0	1	0	1	0	0	0	0	**1**	1	1	1	0	0	1	0			**C** R3
1	1	1	1	1	**1**	1	1	0	0	1	0	0	0	1	0	1									**C** R4

Cycle 2

0	**1**	**0**	1	1	**1**	1	1	0	0	0	0	0	0	1	0	0	0	1							R1
0	**0**	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	1	1				**C** R2
0	**0**	**1**	1	0	0	0	1	0	1	0	0	0	0	1	**1**	1	1	0	0	1	0	1			**C** R3
1	1	1	1	1	**1**	1	0	0	1	0	0	0	1	0	1	0									**C** R4

Cycle 3

1	**0**	**1**	1	1	**1**	1	0	0	0	0	0	0	1	0	0	0	1	0							**C** R1
0	**1**	1	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	1	1	0				**C** R2
0	**1**	**1**	0	0	0	1	0	1	0	0	0	0	1	1	**1**	1	0	0	1	0	1	0			**C** R3
1	1	1	1	1	**1**	0	0	1	0	0	0	1	0	1	0	0									**C** R4

.

.

.

Cycle 97

1	**1**	**0**	0	1	**1**	0	1	0	1	0	1	0	1	1	1	1	0	0							**C** R1
0	**0**	0	1	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1				**C** R2
0	**1**	**1**	1	1	0	1	1	1	0	1	0	1	1	0	**1**	1	1	1	1	1	1	0			**C** R3
1	0	0	0	0	**0**	1	1	0	0	1	0	0	1	1	1	0									**C** R4

Cycle 98

1	**0**	**0**	1	1	**0**	1	0	1	0	1	0	1	1	1	1	0	0	1							**C** R1
0	**0**	1	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1	0				**C** R2
0	**1**	**1**	1	1	0	1	1	1	0	1	0	1	1	0	**1**	1	1	1	1	1	1	0			R3
0	0	0	0	0	**1**	1	0	0	1	0	0	1	1	1	0	1									**C** R4

Cycle 99

0	**0**	**1**	1	0	**1**	0	1	0	1	0	1	1	1	1	0	0	1	1							**C** R1
0	**1**	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1	0	0				**C** R2
1	**1**	**1**	1	0	1	1	1	0	1	0	1	1	0	1	**1**	1	1	1	1	1	0	1			**C** R3
0	0	0	0	1	**1**	0	0	1	0	0	1	1	1	0	1	1									**C** R4

Stage 3 Output:

R1: 001 1010 1010 1111 0011

R2: 01 1010 1101 0011 0110 1100

R3: 111 1011 1010 1101 1111 1101

R4: 0 0001 1001 0011 1011

Stage 4 (228 Cycles) C=Register Clocked

Cycle 1

0	**1**	**1**	0	1	**0**	1	0	1	0	1	1	1	1	0	0	1	1	0							**C** R1
0	**1**	1	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1	0	0				R2
1	**1**	**1**	0	1	1	1	0	1	0	1	1	0	1	1	**1**	1	1	1	1	0	1	0			**C** R3
0	0	0	1	1	**0**	0	1	0	0	1	1	1	0	1	1	1									**C** R4

Majority Function Output: R1:0 , R2:1 , R3:1

Keystream: **1**

Cycle 2

1	**1**	**0**	1	0	**1**	0	1	0	1	1	1	1	0	0	1	1	0	0						**C** R1
1	**1**	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1	0	0	1			**C** R2
1	**1**	**0**	1	1	1	0	1	0	1	1	0	1	1	1	**1**	1	1	1	0	1	0	0		**C** R3
0	0	1	1	0	**0**	1	0	0	1	1	1	0	1	1	1	0								**C** R4

Majority Function Output: R1:1 , R2:1 , R3:0

Keystream: 1**1**

Cycle 3

1	**0**	**1**	0	1	**0**	1	0	1	1	1	1	0	0	1	1	0	0	1						**C** R1
1	**0**	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1	0	0	1	0			**C** R2
1	**0**	**1**	1	1	0	1	0	1	1	0	1	1	1	1	**1**	1	1	0	1	0	0	1		**C** R3
0	1	1	0	0	**1**	0	0	1	1	1	0	1	1	1	0	0								**C** R4

Majority Function Output: R1:1 , R2:0 , R3:0

Keystream: 11**0**

.

.

.

Cycle 226

0	**1**	**0**	1	0	**0**	1	1	0	0	1	1	1	0	0	0	1	0	0						**C** R1
0	**1**	1	1	1	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	0			R2
1	**0**	**1**	1	1	1	1	1	0	0	1	1	1	0	0	**1**	1	0	0	0	1	0	0		**C** R3
1	1	0	1	0	**1**	0	0	1	0	0	1	0	0	0	1	1								**C** R4

Majority Function Output: R1:1 , R2:0 , R3:1

Keystream: 11001110 00110111 00111101 11010100 11011010 11111101 01110011
10110001 10111100 00010011 01110000 01000110 00111000 10011001 01
11010100 00001011 11101100 11110110 00110001 11101100 11001010
00010011 10011001 11001000 10000010 10111111 01001010 00101011**1**

Cycle 227

1	**0**	**1**	0	0	**1**	1	0	0	1	1	1	0	0	0	1	0	0	1						**C** R1
1	**1**	1	1	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	0	1			**C** R2
0	**1**	**1**	1	1	1	1	0	0	1	1	1	0	0	1	**1**	0	0	0	1	0	0	1		**C** R3
1	0	1	0	0	**0**	0	1	0	0	1	0	0	0	1	1	0								**C** R4

Majority Function Output: R1:1 , R2:1 , R3:1

Keystream: 11001110 00110111 00111101 11010100 11011010 11111101 01110011
10110001 10111100 00010011 01110000 01000110 00111000 10011001 01
11010100 00001011 11101100 11110110 00110001 11101100 11001010
00010011 10011001 11001000 10000010 10111111 01001010 00101011 **1**

Cycle 228

0	**1**	**0**	0	1	**1**	0	0	1	1	1	0	0	0	1	0	0	1	1						**C** R1
1	**1**	1	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	0	1	0			**C** R2
1	**1**	**1**	1	1	1	0	0	1	1	1	0	0	1	1	**0**	0	0	1	0	0	1	1		**C** R3
0	1	0	0	0	**0**	1	0	0	1	0	0	0	1	1	0	1								**C** R4

Majority Function Output: R1:0 , R2:1 , R3:0

Keystream: 11001110 00110111 00111101 11010100 11011010 11111101 01110011
10110001 10111100 00010011 01110000 01000110 00111000 10011001 01
11010100 00001011 11101100 11110110 00110001 11101100 11001010
00010011 10011001 11001000 10000010 10111111 01001010 00101011 1**1**

Output from A5/2 Algorithm

Keystream from MS to BTS:

11001110 00110111 00111101 11010100 11011010 11111101 01110011
10110001 10111100 00010011 01110000 01000110 00111000 10011001 01
Hexadecimal Conversion: CE 37 3D D4 DA FD 73 B1 BC 13 70 46 38 99 40

Keystream from BTS to MS:

11010100 00001011 11101100 11110110 00110001 11101100 11001010
00010011 10011001 11001000 10000010 10111111 01001010 00101011 11
Hexadecimal Conversion: D4 0B EC F6 31 EC CA 13 99 C8 82 BF 4A 2B C0

3.2.6 A5/2 Attacks

An instant ciphertext-only attack for A5/2 was developed by Barkan, Biham, and Keller from the Israel Institute of Technology [15]. The attack only requires a few dozen milliseconds of ciphertext and finds the correct key in less than a second using a personal computer. The attack takes advantage of the fact that GSM networks use error-correction codes. This error-correction code is applied to the plaintext and then both are encrypted to be sent over the air. The addition of an error-correction code increases the size of the ciphertext sent. It is the standard today to first encrypt the message and then apply error-correcting codes. The reason being that encrypting error-correcting codes introduces a structured

redundancy in the ciphertext which is exploited in the attack. The attack focuses on the error-correction codes used at the beginning of a conversation. The error-correction code has a fixed length of 184-bits and when implemented with the plaintext result in a 456-bit message. The 456-bit message is then divided into four frames and then encrypted and transmitted. The initial internal state of the registers are treated as a variable and every bit of ciphertext is written as a quadratic function of those variables. Each block of 456-bits will produce 450 equations of which only 272 equations are needed. Thus after two blocks (8 frames) we can use Gaussian elimination on the system of quadratic equations to find all of the original linear variables in the initial state. The setup process of A5/2 is then inverted once the initial state is known to find the Cipher Key.

This attack on the weaker A5/2 algorithm is used to eavesdrop in real time on any ME that is equipped with the A5/2 algorithm. As stated in the A5/1 attack section most ME are equipped with the A5/0 (no encryption), A5/1 (strong encryption), and A5/2 (weak encryption) algorithms to allow for roaming on A5/2 only networks. The attack downgrades the encryption of the ME from A5/1 to to the weaker A5/2 algorithm where the Cipher Key is easily found. Once the Cipher Key is found the A5/1 algorithm provides no protection as the same Cipher Key is used for both algorithms. This downgrade attack is implemented by what is referred to as a Man in the Middle Attack [15]. This attack takes advantage of three key weaknesses in the GSM network:

1. Authentication and Key agreement protocol can be executed between the MS and the network at the beginning of a call at the networks discretion. The MS cannot ask for authentication of the network.
2. The network chooses the encryption algorithm to use (A5/0, A5/1 or A5/2). The MS only lists the available ciphers it supports. Note though that if no encryption is selected (A5/0) a message will display on the MS indicating no encryption.
3. There is no Cipher Key separation between A5/1 and A5/2. The Cipher Key only depends on the RAND .

The Man in the Middle Attack uses a fake base station to impersonate the network for the MS while impersonating the MS for the network. When the network asks for authentication an authentication request is sent to the attacker who forwards it to the MS. The MS computes the SRES and returns it to the attacker. The attacker holds on to the SRES and does not send it to the Network. The attacker request the MS to start encryption using the A5/2 algorithm. The MS replies with a 456-bit Cipher Mode Complete Message encrypted using A5/2 to verify it is encrypting the message. The attacker does not have the Cipher Key so he is not able to respond. Immediately after not receiving a response the MS resends the Cipher Mode

Complete Message of 456-bits in another frame set. Now the attacker has the required amount of ciphertext needed to begin the A5/2 attack and within a second has the Cipher Key. As an added bonus to the attacker the same Cipher Mode Complete Message is sent with a different frame number and the only difference being that one bit is changed from a 0 to a 1 to indicate retransmission. The attacker then sends the SRES computed by the MS back to the network and is authenticated. The network starts encrypting using the stronger A5/1 algorithm. The attacker knowing the Cipher Key is able to send and respond to encrypted messages using A5/1. The network views the attacker as the authenticated subscriber and the attacker is able to forward messages to the MS while eavesdropping in real time or use the subscribers account to place calls at their expense. The attacker is also able to receive SMS and data messages and change them at will before forwarding them to the network or MS. The whole attack causes only a one second delay in authentication and the GSM network gives a 12 second window for the MS to authenticate itself. This Man in the Middle attack using the downgraded A5/2 algorithm provides a real time security threat to any MS equipped with the A5/2 algorithm. This attack was negated in the 3G UMTS system by requiring authentication both ways between the MS and network. Note that the removal of the A5/2 algorithm took some time to implement. The attacks against A5/2 began to be published in 1999 but were not taken seriously until 2003. In 2003 the GSM Association recognized that there was a problem with the downgrade attack and the only solution was to remove the algorithm from new equipment. Although it wasn't until 2007 that the removal of the A5/2 algorithm was implemented.

CHAPTER 4

CRYPTOGRAPHIC ALGORITHMS FOR 3G/4G UMTS/LTE NETWORKS

The 3GPP wanted to design the UMTS system to address some of the security flaws of the original 2G GSM system while using the GSM system as a foundation upon which to build. Unlike the 2G GSM networks where the security algorithms were kept secret the 3GPP called upon the open scientific community to design their cryptographic systems. The 3GPP made public all of their drafts, designs, and algorithms for the scrutiny of the academic community. This created a process for the 3GPP to select the best authentication and encryption algorithms of the time. There were three certain flaws of the 2G system that needed to be addressed such as the short key length of the A5 algorithms, the vulnerability to the false base station attack, and the lack of a message integrity system. The A5 algorithms used a key length of only 64-bits which was considered secure at the time, but with the advent of more powerful computers 64-bit key lengths became a vulnerability. The false base station attack was possible because 2G systems only authenticated the user and not the network. The 3GPP wanted to create a system were both the user and the network were authenticated. A system of checking for message integrity was needed to ensure that there was no tampering of communications between the network and subscriber. With these three ideas in mind the 3GPP designed the UMTS cryptographic system.

UMTS was built on the GSM network to allow for easy migration for the networks and provide backward compatibility for the user. Thus a 2G phone would be able to operate on a 3G network. Some of the terminology has also changed to address the upgraded technology of more advanced cell phones. The 2G GSM system refers to the mobile phone and SIM card as the MS while the UMTS 3G system uses the term User Equipment (UE). The mobile phone is now referred to as the Terminal and 3G phones have larger screens and fewer buttons with the advent of touch screens. The word terminal was chosen for 3G phones because the phone is often thought of as a handheld computer. In the GSM system the authentication algorithm was hard coded into the SIM card. For UMTS the SIM card is referred to as the Universal Subscriber Identity Module (USIM). The difference between a SIM card and a USIM card is that the security algorithms are not hard coded into the USIM but rather implemented as applications. This allows for multiple applications to be placed on the USIM which allows opportunity for the UE to be used on multiple carriers around the

world. The HSDPA/HSPA+/LTE networks are all based on UMTS and therefore use the same algorithm set for encryption and authentication.

4.1 Authentication and Key Agreement: $f1$-$f5$

Unlike 2G systems the UMTS system authenticates not only the UE, but the UE also authenticates the network. To reduce network traffic this authentication is done with a single pass. The authentication process for UMTS is referred to as Authentication and Key Agreement (AKA). The AKA process takes place as soon as the UE is detected on the network. The Key Agreement refers to the generation of the Confidentiality Key (CK) used to encrypt over-the-air communications and the Integrity Key (IK). The IK is used to verify that the message has not been tampered with by a man in the middle.

The AKA process begins when an Authentication Data Request is made by sending the UE's IMSI number to the users HLR. The HLR then produces a set of n Authentication Data Responses or Authentication Vectors AV(1),AV(2),...Av(n). Each Authentication Vector is a 5-tuple consisting of the following: a RAND , an Expected Response (XRES), a CK, an IK, and an Authentication Token (AUTN). The HLR then sends this 5-tuple to the VLR. The VLR selects the first Authentication Vector AV(1) and sends the two values RAND (1) and AUTN(1) to the UE. The UE authenticates the network by verifying AUTN(1). The UE then computes the user authentication Response (RES) RES(1) from the RAND (1) and sends this to the VLR. The VLR compares the RES(1) with XRES(1). If the RES(1) and XRES(1) agree, then the authentication of the user is complete and the CK(1) and IK(1) can be used to encrypt and check the integrity of over-the-air communications.

For the AKA process there is a total of seven algorithms. The name used to refer to these seven algorithms is "Milenage." The following algorithm descriptions, functions, and diagrams have all been taken from the publicly produced documentation of the 3GPP [7]. The seven algorithms used for the authentication process are referred to as $f1$, $f1$*, $f2$, $f3$, $f4$, $f5$, and $f5$*. These set of algorithms are stored on the UE's USIM card along with the networks AuC. The USIM is designed to produce the CK, IK, and the AUTN in less than 500ms. The $f1$ algorithm is responsible for the network authentication. The $f1$* algorithm is in charge of the re-synchronization message authentication. The $f2$ algorithm is the user authentication function. The $f3$ algorithm produces the CK while the $f4$ algorithm produces the IK. The $f5$ function is responsible for the Anonymity Key (AK) derivation while the $f5*$ is used to derive the anonymity key when the re-synchronization message function $f1$* is used. A list of variables for the inputs and outputs for the Milenage functions are giconfidentialityven in Table 4.1.

Table 4.1. A Table of Input/Outputs for the Milenage Functions Source: G.T. 35.206, *3rd generation partnership project; technical specification group services and system aspects; 3G security; specification of the milenage algorithm set: An example algorithm set for the 3gpp authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; document 2: Algorithm specification*, technical report, 3GPP, 2012.

AK	a 48-bit anonymity key that is the output of either of the functions $f5$ and $f5$*.
AMF	a 16-bit authentication management field that is an input to the functions $f1$ and $f1$*.
$c1, c2, c3, c4, c5$	128-bit constants, which are XORed onto intermediate variables.
CK	a 128-bit confidentiality key that is the output of the function $f3$.
IK	a 128-bit integrity key that is the output of the function $f4$.
$IN1$	a 128-bit value constructed from SQN and AMF and used in the computation of the functions $f1$ and $f1$*.
K	a 128-bit subscriber key that is an input to the functions $f1, f1$*, $f2, f3, f4, f5$, and $f5$*.
$MAC - A$	a 64-bit network authentication code that is the output of the function $f1$.
$MAC - S$	a 64-bit resynchronization authentication code that is the output of the function $f1$*.
OP	a 128-bit Operator Variant Algorithm Configuration Field that is a component of the functions $f1$, $f1$*, $f2$, $f3$, $f4$, $f5$, and $f5$*.
OP_c	a 128-bit value derived from OP and K and used within the computation of the functions.
$OUT1, OUT2,$ $OUT3, OUT4,$ $OUT5$	128-bit computed values from which the outputs of the functions $f1$, $f1$*, $f2$, $f3$, $f4$, $f5$, and $f5$* are obtained.
$r1, r2, r3, r4, r5$	integers in the range 0-127 inclusive, which define amounts by which intermediate variables are cyclically rotated.
$RAND$	a 128-bit random challenge that is an input to the functions $f1, f1$*, $f2, f3, f4, f5$,and$f5$*.
RES	a 64-bit signed response that is the output of the function $f2$.
SQN	a 48-bit sequence number that is an input to either of the functions $f1$ and $f1$*. (For $f1$* this input is more precisely called SQN_{MS}.)
$TEMP$	a 128-bit value used within the computation of the functions.

Using the information from Table 4.1 we can examine the inputs and outputs for each function.

Inputs for $f1$ and $f1*$

K	128-bits	Subscriber key $K[0]...K[127]$
$RAND$	128-bits	Random challenge $RAND[0]...RAND[127]$
SQN	48-bits	Sequence number $SQN[0]...SQN[47]$. (For $f1*$ this input is more precisely called SQN_{MS}.)
AMF	16-bits	Authentication management field $AMF[0]...AMF[15]$

Outputs for $f1$ and $f1*$

$f1$	$MAC-A$	64-bits	Network authentication code $MAC-A[0]...MAC-A[63]$
$f1*$	$MAC-S$	64-bits	Resync authentication code $MAC-S[0]...MAC-S[63]$

Inputs for $f2$, $f3$, $f4$, $f5$ and $f5*$

K	128-bits	Subscriber key $K[0]...K[127]$
$RAND$	128-bits	Random challenge $RAND[0]...RAND[127]$

Outputs for $f2$, $f3$, $f4$, $f5$ and $f5*$

$f2$	RES	64-bits	Response $RES[0]...RES[63]$
$f3$	CK	128-bits	Confidentiality key $CK[0]...CK[127]$
$f4$	IK	128-bits	Integrity key $IK[0]...IK[127]$
$f5$	AK	48-bits	Anonymity key $AK[0]...AK[47]$
$f5*$	AK	48-bits	Resync anonymity key $AK[0]...AK[47]$

The Milenage set of algorithms use a block cipher for encryption called Rijndael, which is the proposed algorithm for the Advanced Encryption Standard. The Rijndael algorithm uses a 128-bit key along with a 128-bit block size input. In this paper the symbol $E[x]_k$ equals the result of applying the Rijndael encryption algorithm to the 128-bit value x under the 128-bit key k. The Milenage set of algorithms also use a 128-bit value called the Operator Variant Algorithm Configuration Field (OP). Each network is allowed to select its own value. Whether or not it is made public is determined by the operator. Note that the security of the algorithm does not depend on the secrecy of the OP. This value of OP is used by all subscribers on the network. Note that only the value OP_C is used in the Milenage algorithms where OP_C is computed from the inputs K and OP. Thus only the value of OP_C is stored on the USIM card as opposed to having OP stored on the USIM card. This increases the chances that OP will remain a secret and provides one more hurdle for an attacker. A framework for the Milenage set of algorithms $f1-f5*$ is as follows:

A 128-bit value OP_C is derived from OP and K as follows:

$$OP_C = OP \oplus E[OP]_k.$$

An intermediate 128-bit value $TEMP$ is computed as follows:

$$TEMP = E[RAND \oplus OP_C]_k.$$

A 128-bit value $IN1$ is constructed as follows:

$$\begin{array}{lcl} IN1[0]...IN1[47] & = & SQN[0]...SQN[47] \\ IN1[48]...IN1[63] & = & AMF[0]...AMF[15] \\ IN1[64]...IN1[111] & = & SQN[0]...SQN[47] \\ IN1[112]...IN1[127] & = & AMF[0]...AMF[15] \end{array}$$

Five 128-bit constants $c1, c2, c3, c4, c5$ are defined as follows:

$$\begin{array}{l} c1[i] = 0 \text{ for } 0 \le i \le 127 \\ c2[i] = 0 \text{ for } 0 \le i \le 127, \quad \text{except that} \quad c2[127] = 1 \\ c3[i] = 0 \text{ for } 0 \le i \le 127, \quad \text{except that} \quad c3[126] = 1 \\ c4[i] = 0 \text{ for } 0 \le i \le 127, \quad \text{except that} \quad c4[125] = 1 \\ c5[i] = 0 \text{ for } 0 \le i \le 127, \quad \text{except that} \quad c5[124] = 1 \end{array}$$

Five integers $r1$, $r2$, $r3$, $r4$, $r5$ are defined as follows:

$$r1 = 64; \quad r2 = 0; \quad r3 = 32; \quad r4 = 64; \quad r5 = 96$$

Five 128-bit blocks $OUT1$, $OUT2$, $OUT3$, $OUT4$, $OUT5$ are computed as follows:

$$\begin{array}{l} OUT1 = E[TEMP \oplus rot(IN1 \oplus OP_C, r1) \oplus c1]_K \oplus OP_C \\ OUT2 = E[rot(TEMP \oplus OP_C, r2) \oplus c2]_k \oplus OP_C \\ OUT3 = E[rot(TEMP \oplus OP_C, r3) \oplus c3]_k \oplus OP_C \\ OUT4 = E[rot(TEMP \oplus OP_C, r4) \oplus c4]_k \oplus OP_C \\ OUT5 = E[rot(TEMP \oplus OP_C, r5) \oplus c5]_k \oplus OP_C \end{array}$$

The outputs of the various functions are then defined as follows:
Output of $f1 = MAC - A$, where $MAC - A[0]...MAC - A[63] = OUT1[0]...OUT1[63]$
Output of $f1^* = MAC - S$, where $MAC - S[0]...MAC - S[63] = OUT1[64]...OUT1[127]$
Output of $f2 = RES$, where $RES[0]...RES[63] = OUT2[64]...OUT2[127]$
Output of $f3 = CK$, where $CK[0]...CK[127] = OUT3[0]...OUT3[127]$
Output of $f4 = IK$, where $IK[0]...IK[127] = OUT4[0]...OUT4[127]$
Output of $f5 = AK$, where $AK[0]...AK[47] = OUT2[0]...OUT2[47]$
Output of $f5^* = AK$, where $AK[0]...AK[47] = OUT5[0]...OUT5[47]$

Note that AK is produced from both $f5$ and $f5*$. This is due to the fact that only one of these functions will be executed to compute AK. Also the function $rot(x, r)$ used to compute $OUTi$ for $1 \leq i \leq 5$ is the result of cyclically rotating the 128-bit value x by r-bit positions towards the most significant bit. For an example if $x = x[0]||x[1]||...||x[127]$, and $y = rot(x, r)$, then $y = x[r]||x[r+1]||...x[127]||x[0]||x[1]||...||x[r-1]$. A diagram of the Milenage algorithms $f1 - f5*$ is given in Figure 4.1:

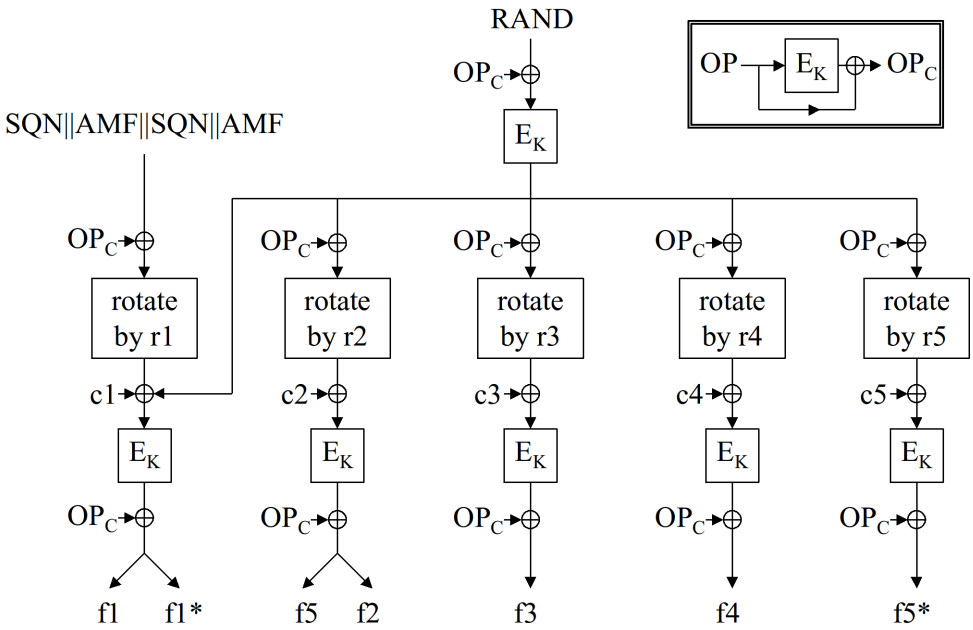

Figure 4.1. Milenage key generation functions. Source: G. T. 35.206, *3rd generation partnership project; technical specification group services and system aspects;3gsecurity; specification of the milenage algorithm set: An example algorithm set for the 3gpp authentication and key generation functions* $f1$*,* $f1*$*,* $f2$*,* $f3$*,* $f4$*,* $f5$ *and* $f5*$*; document 2: Algorithm specification*, technical report, 3GPP, 2012.

The Milenage AKA process begins with the UE receiving the $RAND$ and $AUTN$ computed in the AuC and sent by the VLR. The $AUTN$ is computed by concatenating the XOR of the SQN and AK, the AMF, and the $MAC - A$. Thus $AUTN = SQN \oplus AK||AMF||MAC - A$. The UE then inputs the 128-bit $RAND$ and the 128-bit shared Subscribers Key K into the $f5$ function. This produces the 48-bit AK. Since the first 48-bits of the $AUTN$ consists of the $SQN \oplus AK$, AK is XORed with the first 48-bits of $AUTN$ to produce the SQN. The USIM then checks to make sure that the SQN is in the correct range. If the the 48-bit SQN is in the correct range then it is inputted into the $f1$ function along with 128-bit $RAND$, K, and the 16-bit AMF. Note that the AMF is found

by taking $AUTN[48]...AUTN[64]$. The output of $f1$ is the $XMAC-A$. If the $XMAC-A$ agrees with the $MAC-A$, then the UE authenticates the network. Note that if the SQN is found to be out of range, then the normal key generation ceases and the functions $f1$* and $f5$* are used in place of $f1$ and $f5$. The $f1$* and $f5$* are only used when the SQN is found out of range. Once the network has been authenticated the key generation process can begin. The $RAND$ and K are inputted into $f2$, $f3$, and $f4$ to produce the RES, CK, and IK.

For each function $f1-f5$*, the Rijndael block cypher is used with input lengths of 128-bits. The Rijndael block cipher consists of ten rounds after an initial Round Key addition. The first nine rounds consists of the following: a byte substitution transformation, a shift row transformation, a mix column transformation, and then a Round Key addition. The tenth round consists of a byte substitution transformation, a shift row transformation, and a Round Key addition. The result between each round is referred to as the State. The State is a 4x4 rectangular array of bytes. The output, input, and key are all represented as 4x4 arrays. Since each State represents 128-bits and a 4x4 array consists of 16 spaces each space represents a byte. The plaintext and Subscribers Key are divided into P_0, P_1,..., P_{15} and K_0, K_1,...,K_{15}. These values are then mapped respectively to $a_{0,0}$, $a_{1,0}$, $a_{2,0}$, $a_{3,0}$, $a_{0,1}$, $a_{1,1}$, $a_{2,1}$, $a_{3,1}$,... and $K_{0,0}$, $K_{1,0}$, $K_{2,0}$, $K_{3,0}$, $K_{0,1}$, $K_{1,1}$, $K_{2,1}$, $K_{3,1}$,... . The ciphertext bytes C_0, C_1,..., C_15 are extracted by the same process. For example, a given State consisting of a 4x4 plaintext array and 4x4 cipher key array is given below:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Within a 16-byte block of plaintext the index of a byte within the block is given by n or (i,j) where:

$$i = n \mod 4; \qquad j = \lfloor n/4 \rfloor; \qquad n = i + 4 * j$$

Once the plaintext and Cipher key are inputed into the 4x4 arrays the Rijndael block cypher can be executed. The first operation to be run is the Round Key addition which consists of a bitwise XOR with the Round Key and the State. The Round Key is 128-bits represented by a 4x4 array with the notation $rk_{i,j}$. The Round Key is found by using the CK along with a key schedule. For an example of the Round Key addition:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

$\oplus$

$rk_{0,0}$	$rk_{0,1}$	$rk_{0,2}$	$rk_{0,3}$
$rk_{1,0}$	$rk_{1,1}$	$rk_{1,2}$	$rk_{1,3}$
$rk_{2,0}$	$rk_{2,1}$	$rk_{2,2}$	$rk_{2,3}$
$rk_{3,0}$	$rk_{3,1}$	$rk_{3,2}$	$rk_{3,3}$

=

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

After the Round Key addition is preformed every element of the resulting state is found by $b_{i,j} = a_{i,j} \oplus rk_{i,j}$ where:

$a_{i,j}$ is the initial value of the element in the State,
$b_{i,j}$ is the output value of the element in the State, and
$rk_{i,j}$ is the round key byte.

The next transformation after the initial round key addition is the byte substitution transformation. The byte substitution transformation is a non-linear byte substitution operating independently on each byte of the State. The Rijndael S-box substitution table is used to determine the output value of the State where $b_{i,j} = S - box[a_{i,j}]$ as seen in Figure 4.2.

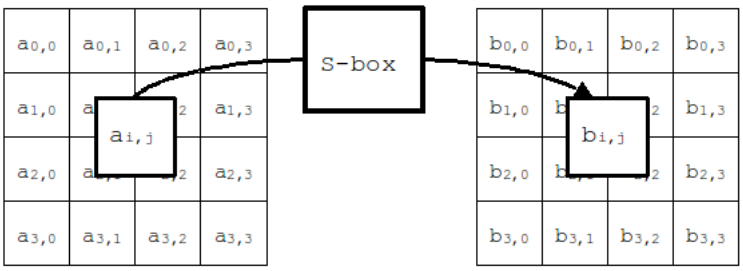

Figure 4.2. Milenage byte substitution transformation using Rijndael s-box. Source: G. T. 35.206, *3rd generation partnership project; technical specification group services and system aspects;3gsecurity; specification of the milenage algorithm set: An example algorithm set for the 3gpp authentication and key generation functions* $f1$*,* $f1$,* $f2$*,* $f3$*,* $f4$*,* $f5$ *and* $f5$**; document 2: Algorithm specification*, technical report, 3GPP, 2012.**

The Rijndael S-box consisting of 256 values is given in Table 4.2.

The shift row transformation consists of cyclically left shifting the rows of the State by different amounts. Row 0 is not shifted, row 1 is shifted by 1 byte, row 2 by 2 bytes, and row 3 by 3 bytes. The shift row transformation is represented in Figure 4.3.

The mix column transformation operates on each column independently. For a given column j in the State we have the following output:

Table 4.2. Rijndael S-Box Used in the Byte Substitution Transformation Source: G.T. 35.206, *3rd generation partnership project; technical specification group services and system aspects;3gsecurity; specification of the milenage algorithm set: An example algorithm set for the 3gpp authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; document 2: Algorithm specification*, technical report, 3GPP, 2012.

99	124	119	123	242	107	111	197	48	1	103	43	254
215	171	118	202	130	201	125	250	89	71	240	173	212
162	175	156	164	114	192	183	253	147	38	54	63	247
204	52	165	229	241	113	216	49	21	4	199	35	195
24	150	5	154	7	18	128	226	235	39	178	117	9
131	44	26	27	110	90	160	82	59	214	179	41	227
47	132	83	209	0	237	32	252	177	91	106	203	190
57	74	76	88	207	208	239	170	251	67	77	51	133
69	249	2	127	80	60	159	168	81	163	64	143	146
157	56	245	188	182	218	33	16	255	243	210	205	12
19	236	95	151	68	23	196	167	126	61	100	93	25
115	96	129	79	220	34	42	144	136	70	238	184	20
222	94	11	219	224	50	58	10	73	6	36	92	194
211	172	98	145	149	228	121	231	200	55	109	141	213
78	169	108	86	244	234	101	122	174	8	186	120	37
46	28	166	180	198	232	221	116	31	75	189	139	138
112	62	181	102	72	3	246	14	97	53	87	185	134
193	29	158	225	248	152	17	105	217	142	148	155	30
135	233	206	85	40	223	140	161	137	13	191	230	66
104	65	153	45	15	176	84	187	22				

$$
\begin{aligned}
b_{0,j} &= T_2(a_{0,j}) \oplus T_3(a_{1,j}) \oplus a_{2,j} \oplus a_{3,j} \\
b_{1,j} &= a_{0,j} \oplus T_2(a_{1,j}) \oplus T_3(a_{2,j}) \oplus a_{3,j} \\
b_{2,j} &= a_{0,j} \oplus a_{1,j} \oplus T_2(a_{2,j}) \oplus T_3(a_{3,j}) \\
b_{3,j} &= T_3(a_{0,j}) \oplus a_{1,j} \oplus a_{2,j} \oplus T_2(a_{3,j})
\end{aligned}
$$

where T_2 and T_3 are defined as:

$$
\begin{aligned}
T_2(a) &= 2 * a && \text{if } a < 128 \text{ or} \\
T_2(a) &= (2 * a) \oplus 283 && \text{if } a \geq 128 \\
T_3(a) &= T_2(a) \oplus a
\end{aligned}
$$

For example:

If $a = 74$ then $T_2(74) = 2 * 74 = 148$; $T_3(74) = T_2(74) \oplus 74 = 222$

If $a = 140$ then $T_2(140) = (2 * 140) \oplus 283 = 3$; $T_3(140) = T_2(140) \oplus 140 = 143$

A diagram of the mix column transformation on the State is given in Figure 4.4:

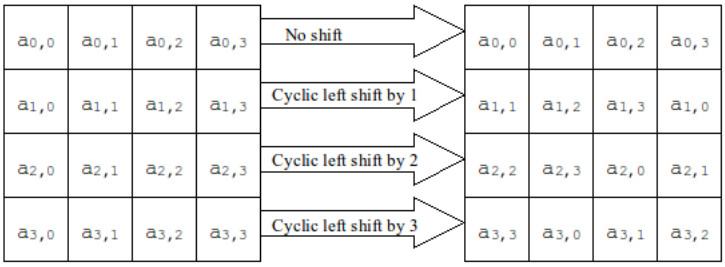

Figure 4.3. Milenage shift row transformation. Source: G. T. 35.206, *3rd generation partnership project; technical specification group services and system aspects; 3g security; specification of the milenage algorithm set: An example algorithm set for the 3gpp authentication and key generation functions* $f1$, $f1$*, $f2$, $f3$, $f4$, $f5$ *and* $f5$*; *document 2: Algorithm specification*, technical report, 3GPP, 2012.

For the round key addition transformation a key schedule is needed to determine each round key. The Rijndael block cipher consists of 11 round keys 0-10 that are each arranged in a 4x4 array of bytes. The CK itself is used as the zeroth Round Key and all of the other round keys are derived from the previous round keys using a key schedule. Let $rk_{r,i,j}$ be the value of the r^{th} round key at position (i, j) in the array and $k_{i,j}$ be the cipher key loaded into the 4x4 array. To determine the 11 round keys the following key schedule is used:

Round Key 0: $rk_{0,i,j} = k_{i,j}$ for all i and j.

Round Key r: (for $r = 1, ..., 10$)

$$rk_{r,0,0} = rk_{r-1,0,0} \oplus S - box[rk_{r-1,1,3}] \oplus round\_const[r]$$
$$rk_{r,1,0} = rk_{r-1,1,0} \oplus S - box[rk_{r-1,2,3}]$$
$$rk_{r,2,0} = rk_{r-1,2,0} \oplus S - box[rk_{r-1,3,3}]$$
$$rk_{r,3,0} = rk_{r-1,3,0} \oplus S - box[rk_{r-1,0,3}]$$

where $round\_const[r] = T_2(round\_const[r-1])$ and $S - box$ is the Rijndael $S - box$ mentioned earlier. After the 0^{th} column is computed the remaining three columns of the 4x4 array are found using the corresponding column of the previous round key along with the previous column of the current round key:

$$rk_{r,i,j} = rk_{r-1,i,j} \oplus rk_{r,i,j-1} \quad \text{for } i = 0, 1, 2, 3 \text{ and } j = 1, 2, 3.$$

The ten round constants $round\_const[r]$ for $r = 1, ..., 10$ are computed by using the following:

$$round\_const[1] = 1$$
$$round\_const[r] = T_2(round\_const[r-1]) \; r = 2, 3, ...10.$$

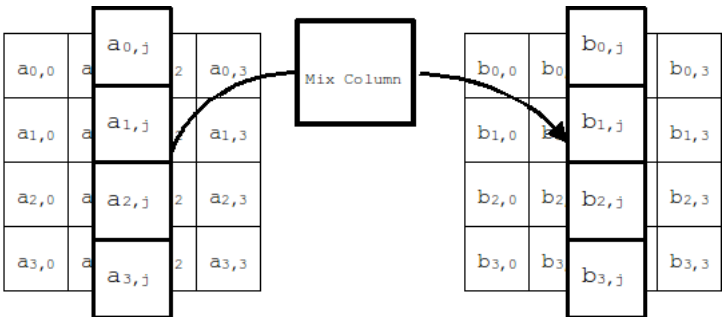

Figure 4.4. Milenage mix column transformation. Source: G. T. 35.206, *3rd generation partnership project; technical specification group services and system aspects; 3g security; specification of the milenage algorithm set: An example algorithm set for the 3gpp authentication and key generation functions* $f1$*,* $f1$,* $f2$*,* $f3$*,* $f4$*,* $f5$ *and* $f5$**; document 2: Algorithm specification*, technical report, 3GPP, 2012.**

Computing the round constants we have:

$$
\begin{aligned}
round\_const[1] &= 1 \\
round\_const[2] &= T_2(round\_const[1]) = T_2(1) = 2*1 = 2 \\
round\_const[3] &= T_2(round\_const[2]) = T_2(2) = 2*2 = 4 \\
round\_const[4] &= T_2(round\_const[3]) = T_2(4) = 2*4 = 8 \\
round\_const[5] &= T_2(round\_const[4]) = T_2(8) = 2*8 = 16 \\
round\_const[6] &= T_2(round\_const[5]) = T_2(16) = 2*16 = 32 \\
round\_const[7] &= T_2(round\_const[6]) = T_2(32) = 2*32 = 64 \\
round\_const[8] &= T_2(round\_const[7]) = T_2(64) = 2*64 = 128 \\
round\_const[9] &= T_2(round\_const[8]) = T_2(128) = (2*128) \oplus 283 = 27 \\
round\_const[10] &= T_2(round\_const[9]) = T_2(27) = 2*27 = 54
\end{aligned}
$$

Thus we get the following results for the round constants $1, 2, 4, 8, 16, 32, 64, 128, 27, 54$.

4.1.1 Rijndael Block Cipher Example

For an example of the Rijndael block cipher we will use the 128-bit Individual Subscribers Authentication Key used as input into the COMP128 algorithm for the CK. The Matlab code used to perform the Rijndael block cipher computations was written by Professor Jorg Buchholz of Hochschule Bremen University of Applied Sciences [18]. For the following example the computations used to derive the first column will be shown for round 1.

Cipher Key:	52A1B0D7	F3816EC6	E7232897	F170A81A
Plaintext:	77656C6C	20646F6E	65207369	72203D29

Rijndael Block Cipher

Initial State:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	=	77	20	65	72
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$		65	64	20	20
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		6C	6F	73	3D
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$		6C	6E	69	29

Round Key 0:

$rk_{0,0,0}$	$rk_{0,0,1}$	$rk_{0,0,2}$	$rk_{0,0,3}$	=	52	F3	E7	F1
$rk_{0,1,0}$	$rk_{0,1,1}$	$rk_{0,1,2}$	$rk_{0,1,3}$		A1	81	23	70
$rk_{0,2,0}$	$rk_{0,2,1}$	$rk_{0,2,2}$	$rk_{0,2,3}$		B0	6E	28	A8
$rk_{0,3,0}$	$rk_{0,3,1}$	$rk_{0,3,2}$	$rk_{0,3,3}$		D7	C6	97	1A

Initial Round Key addition:

$b_{0,0} = 77 \oplus 52 = 25$

$b_{1,0} = 65 \oplus A1 = C4$

$b_{2,0} = 6C \oplus B0 = DC$

$b_{3,0} = 6C \oplus D7 = BB$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$	=	25	D3	82	83
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$		C4	E5	03	50
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$		DC	01	5B	95
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$		BB	A8	FE	33

Round 1 of 10

Byte substitution transformation:

$b_{0,0} = S - box[25] = 3F$

$b_{1,0} = S - box[C4] = 1C$

$b_{2,0} = S - box[DC] = 86$

$b_{3,0} = S - box[BB] = EA$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$	=	3F	66	13	EC
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$		1C	D9	7B	53
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$		86	7C	39	2A
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$		EA	C2	BB	C3

Shift row transformation:

$b_{0,0} = a_{0,0} = 3F$

$b_{1,0} = a_{1,1} = D9$

$b_{2,0} = a_{2,2} = 39$

$b_{3,0} = a_{3,3} = C3$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

=

3F	66	13	EC
D9	7B	53	1C
39	2A	86	7C
C3	EA	C2	BB

Mix column transformation:

$b_{0,0} = T_2(a_{0,0}) \oplus T_3(a_{1,0}) \oplus a_{2,0} \oplus a_{3,0} = T_2(3F) \oplus T_3(D9) \oplus 39 \oplus C3 = F4$

$b_{1,0} = a_{0,0} \oplus T_2(a_{1,0}) \oplus T_3(a_{2,0}) \oplus a_{3,0} = 3F \oplus T_2(D9) \oplus T_3(39) \oplus C3 = 1E$

$b_{2,0} = a_{0,0} \oplus a_{1,0} \oplus T_2(a_{2,0}) \oplus T_3(a_{3,0} = 3F \oplus D9 \oplus T_2(39) \oplus T_3(C3) = CA$

$b_{3,0} = T_3(a_{0,0}) \oplus a_{1,0} \oplus a_{2,0} \oplus T_2(a_{3,0}) = T_3(3F) \oplus D9 \oplus 39 \oplus T_2(C3) = 3C$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

=

F4	81	97	20
1E	04	E6	EB
CA	6C	0A	DE
3C	34	7F	22

Round Key addition:

$rk_{1,0,0} = rk_{0,0,0} \oplus S - box[rk_{0,1,3}] \oplus round\_const[1] = 52 \oplus S - Box[70] \oplus 1 = 02$

$rk_{1,1,0} = rk_{0,1,0} \oplus S - box[rk_{0,2,3}] = A1 \oplus S - Box[A8] = 63$

$rk_{1,2,0} = rk_{0,2,0} \oplus S - box[rk_{0,3,3}] = B0 \oplus S - Box[1A] = 12$

$rk_{1,3,0} = rk_{0,3,0} \oplus S - box[rk_{0,0,3}] = D7 \oplus S - Box[F1] = 76$

$rk_{1,0,0}$	$rk_{1,0,1}$	$rk_{1,0,2}$	$rk_{1,0,3}$
$rk_{1,1,0}$	$rk_{1,1,1}$	$rk_{1,1,2}$	$rk_{1,1,3}$
$rk_{1,2,0}$	$rk_{1,2,1}$	$rk_{1,2,2}$	$rk_{1,2,3}$
$rk_{1,3,0}$	$rk_{1,3,1}$	$rk_{1,3,2}$	$rk_{1,3,3}$

=

02	F1	16	E7
63	E2	C1	B1
12	7C	54	FC
76	B0	27	3D

$b_{0,0} = a_{0,0} \oplus rk_{1,0,0} = F4 \oplus 02 = F6$

$b_{1,0} = a_{1,0} \oplus rk_{1,1,0} = 1E \oplus 63 = 7D$

$b_{2,0} = a_{2,0} \oplus rk_{1,2,0} = CA \oplus 12 = D8$

$b_{3,0} = a_{3,0} \oplus rk_{1,3,0} = 3C \oplus 76 = 4A$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

=

F6	70	81	C7
7D	E6	27	5A
D8	10	5E	22
4A	84	58	1F

End of Round 1

.
.
.

Round 10 of 10

Byte substitution transformation:

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

=

71	78	8F	D3
AE	C6	F4	56
67	8B	8E	9C
BD	A6	96	77

Shift row transformation:

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

=

71	78	8F	D3
C6	F4	56	AE
8E	9C	67	8B
77	BD	A6	96

Round Key addition:

$rk_{10,0,0}$	$rk_{10,0,1}$	$rk_{10,0,2}$	$rk_{10,0,3}$
$rk_{10,1,0}$	$rk_{10,1,1}$	$rk_{10,1,2}$	$rk_{10,1,3}$
$rk_{10,2,0}$	$rk_{10,2,1}$	$rk_{10,2,2}$	$rk_{10,2,3}$
$rk_{10,3,0}$	$rk_{10,3,1}$	$rk_{10,3,2}$	$rk_{10,3,3}$

=

1B	E6	61	1A
ED	CC	73	E2
89	C3	DC	AA
0E	67	21	B8

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

=

6A	9E	EE	C9
2B	38	25	4C
07	5F	BB	21
79	DA	87	1E

Ciphertext: 6A2B0779 9E385FDA EE25BB87 C94C211E

For the rest of this chapter when the Rijndael block cipher algorithm is applied to x with a 128-bit key K denoted as $E[x]_K$, the intermediate steps will be omitted and only the final ciphertext results will be listed.

4.1.2 Authentication and Key Agreement for 3G UMTS Example

The AKA process for the 3G UMTS network relies upon the Milenage algorithms $f1 - f5$*. The same 128-bit $RAND$ from the 2G chapter and CK from above will be used for this example.

$RAND =$ 3B984753 F9AAD8A6 B8DF0E21 DB9A2E8D — Received from VLR
$AUTN =$ 83A28B79 A53CB9B9 FF2B3985 D77BCC99 — Received from VLR

$K =$ 52A1B0D7 F3816EC6 E7232897 F170A81A — Stored on USIM
$OP_c =$ 587A3D8C 9BD195D1 1E8E0AD0 BC800B13 — Stored on USIM

Authentication of Network and User:

Using the received $RAND$ and the stored Subscribers Key K on the USIM, the $f5$ function is implemented first to produce the 48-bit Anonymity Key AK. The AK is taken from the first 48-bits of the 128-bit OUT2 which is computed as follows:

$OUT2 = E[rot(TEMP \oplus OP_C, 0) \oplus c2]_K \oplus OP_C$ where
$TEMP = E[RAND \oplus OP_C]_k$.

$RAND \oplus OP_C =$ 63E27ADF 627B4D77 A65104F1 671A259E
$TEMP = E[RAND \oplus OP_C]_K$
$=$ 0509506F 0D9D2D46 2783FEEE D6B4D3F5
$TEMP \oplus OP_C =$ 5D736DE3 964CB897 390DF43E 6A34D8E6
$rot(TEMP \oplus OP_C, 0) =$ 5D736DE3 964CB897 390DF43E 6A34D8E6
$rot(TEMP \oplus OP_C, 0) \oplus c2 =$ 5D736DE3 964CB897 390DF43E 6A34d8E7
$E[rot(TEMP \oplus OP_C, 0) \oplus c2]_K =$ 24430225 88EA03C4 8F412ED1 A7D7FE45
$E[rot(TEMP \oplus OP_C, 0) \oplus c2]_K \oplus OP_C =$ 7C393FA9 133B9615 91CF2401 1B57F556

$OUT2 =$ 7C393FA9 133B9615 91CF2401 1B57F556
$AK =$ 7C393FA9 133B — Output of $f5$

Since the first 48-bits of $AUTN$ is found by taking $SQN \oplus AK$, we can find SQN by XORing the first 48-bits of $AUTN$ with AK.

$$\begin{aligned} SQN = (SQN \oplus AK) \oplus AK &= \text{83A28B79 A53C} \oplus \text{7C393FA9 133B} \\ &= \text{FF9BB4D0 B607} \end{aligned}$$

The next step in the authentication process is to input the K, $RAND$, SQN, and the AMF into the $f1$ function. Note that the AMF is found by taking $AUTN[48]...AUTN[64]$. Thus $AMF =$ B9B9. The output of the $f1$ function is found by taking the first 64-bits of $OUT1$ where:

$$\begin{aligned} OUT1 &= \text{E}[TEMP \oplus rot(IN1 \oplus OP_C, 64) \oplus c1]_K \oplus OP_C \text{ and} \\ IN1 &= SQN||AMF||SQN||AMF \\ &= \text{FF9BB4D0 B607B9B9 FF9BB4D0 B607B9B9} \end{aligned}$$

Computing $OUT1$ we have:
$IN1 \oplus OP_C =$ A7E1895C 2DD62C68 E115BE00 0A87B2AA
$rot(IN1 \oplus OP_C, 64) =$ E115BE00 0A87B2AA A7E1895C 2DD62C68
$TEMP \oplus rot(IN1 \oplus OP_C, 64) =$ E41CEE6F 071A9FEC 806277B2 FB62FF9D
$TEMP \oplus rot(IN1 \oplus OP_C, 64) \oplus c1 =$ E41CEE6F 071A9FEC 806277B2 FB62FF9D
$E[TEMP \oplus rot(IN1 \oplus OP_C, 64) \oplus c1]_K =$ A7510409 4CAA5948 F5707158 C6B07533
$E[TEMP \oplus rot(IN1 \oplus OP_C, 64) \oplus c1]_K \oplus OP_C =$ FF2B3985 D77BCC99 EBFE7B88 7A307E20

$OUT1 =$ FF2B3985 D77BCC99 EBFE7B88 7A307E20
$MAC - A =$ FF2B3985 D77BCC99 Output of $f1$

Therefore the output of the $f1$ function, the $MAC - A$, is compared with the last 64-bits of the $AUTN$ which is the Expected Message Authentication Code ($XMAC - A$). Since $XMAC - A = MAC - A$ the network is authenticated and the USIM verifies that the SQN is in the correct range of values. This range of values is determined by each network. If the network has been authenticated but the SQN is not in the correct range a re-synchronization procedure begins where the functions $f1*$ and $f5*$ are used to generate a Resynchronization Message Code referred to as $MAC - S$. Note that the $f1$* and $f5$* functions are only used when the network has been authenticated but the SQN is out of range. Once the network has been authenticated and the SQN is verified to be in the correct range the USIM needs to be verified by the network. The verification of the USIM on the network is accomplished by inputting the K and $RAND$ into the $f2$ function and sending back to the VLR the output (64-bit RES). The 64-bit RES output of the $f2$ function is found by taking bits $OUT2[64]...OUT2[127]$. Since $OUT2$ was computed from the $f5$ function we have:

$$RES = \text{91CF2401 1B57F556}$$

The RES is then sent to the VLR where it is compared to the $XRES$. If the two values agree the USIM is authenticated on the network. Once the authentications have been completed, the

key agreement or key generation process begins.

Key Agreement:

The first value to be computed in the key generation process is the Confidentiality Key (CK). The 128-bit CK is found by inputting the received $RAND$ and the stored Subscribers Key K on the USIM card into the $f3$ function. The CK is taken from the 128-bits of OUT3 which is computed as follows:

$$OUT3 = E[rot(TEMP \oplus OP_C, 32) \oplus c3]_k \oplus OP_C$$

$TEMP \oplus OP_C =$ 5D736DE3 964CB897 390DF43E 6A34D8E6
$rot(TEMP \oplus OP_C, 32) =$ 964CB897 390DF43E 6A34D8E6 5D736DE3
$rot(TEMP \oplus OP_C, 32) \oplus c3 =$ 964CB897 390DF43E 6A34D8E6 5D736DE1
$E[rot(TEMP \oplus OP_C, 32) \oplus c3]_k =$ 3EAF96B1 2AF551C6 A988BB3E 2D5FBD85
$E[rot(TEMP \oplus OP_C, 32) \oplus c3]_k \oplus OP_C =$ 66D5AB3D B124C417 B706B1EE 91DFB696

$OUT3 =$ 66D5AB3D B124C417 B706B1EE 91DFB696
$CK =$ 66D5AB3D B124C417 B706B1EE 91DFB696 Output of $f3$

The second value to be computed in the key generation process is the Integrity Key IK. The 128-bit IK is found by inputting the received $RAND$ and the stored K on the USIM card into the $f4$ function. The IK is taken from the 128-bits of OUT4 which is computed as follows:

$$OUT4 = E[rot(TEMP \oplus OP_C, 64) \oplus c4]_k \oplus OP_C$$

$TEMP \oplus OP_C =$ 5D736DE3 964CB897 390DF43E 6A34D8E6
$rot(TEMP \oplus OP_C, 64) =$ 390DF43E 6A34D8E6 5D736DE3 964CB897
$rot(TEMP \oplus OP_C, 64) \oplus c4 =$ 390DF43E 6A34D8E6 5D736DE3 964CB893
$E[rot(TEMP \oplus OP_C, 64) \oplus c4]_k =$ DADB7C39 DD43593B AD188EF64B7EF7D6
$E[rot(TEMP \oplus OP_C, 64) \oplus c4]_k \oplus OP_C =$ 82A141B5 4692CCEA B3968426 F7FEFCC5

$OUT4 =$ 82A141B5 4692CCEA B3968426 F7FEFCC5
$IK =$ 82A141B5 4692CCEA B3968426 F7FEFCC5 Output of $f4$

Once the Key Agreement process is complete and the CK and IK have been computed, encryption and integrity check of over the air communications can begin.

4.2 Encryption and Integrity Check for 3G UMTS: $f8$ & $f9$

The encryption algorithm for the UMTS system is referred to as $f8$. There were two goals in mind with the choosing of this algorithm: resilience and world-wide availability. Resilience in the sense that it will be in use for at least 20 years and provide protection from an exhaustive key search attack through an effective key space. To accomplish the goal of being a world-wide cellular system the algorithm had to be free of any use restrictions and made available publicly for all to implement. Because the time frame for developing an encryption algorithm was limited the 3GPP decided to choose an already proven strong cryptographic system and modify it to their needs. The 3GPP group then limited their search to only algorithms that carried no restrictions on export or use. After a call for open submissions the MISTY algorithm was chosen as the basis for the encryption algorithm $f8$. The MISTY algorithm was then modified for cellular phone use and renamed to KASUMI which is Japanese for MISTY. The descriptions and diagrams of the $f8$ algorithm to follow come from the openly available 3GPP documentations: General Report on the Design, Specification and Evaluation of 3GPP Standard Confidentiality and Integrity Algorithms [1], Specification of the 3GPP Confidentiality and Integrity Algorithms Document 1: f8 and f9 specifications [3], Document 2: KASUMI Specification [4], Document 3: Implementers' Test Data [5], and Document 4: Design Conformance Test Data [6].

4.2.1 Over-the-Air Encryption: $f8$

The core of the $f8$ algorithm is KASUMI which is a block cipher that produces a 64-bit output from a 64-bit input under the control of a 128-bit key. The $f8$ algorithm itself is a stream cipher that is used to encrypt/decrypt blocks of data under a confidentiality key CK. The $f8$ algorithm uses KASUMI as a keystream generator with output keystream in multiples of 64-bits. The input data is modified by static data held in a 64-bit register A (holds intermediate values), and an incrementing 64-bit counter $BLKCNT$. $BLKCNT$ is considered a 64-bit counter yet the key stream generator produces no more than 5114 bits or 80 keystream blocks. Therefore only the least significant 7-bits of $BLKCNT$ are realized. The inputs for the $f8$ algorithm consists of the following:

$COUNT-I$	(32-bits): A frame dependent input $COUNT-I[0]...COUNT-I[31]$
$BEARER$	(5-bits): Bearer identity $BEARER[0]...BEARER[4]$
$DIRECTION$	(1-bit): Direction of transmission $DIRECTION[0]$
CK	(128-bits): Confidentiality Key $CK[0]...CK[127]$
$LENGTH$	(X18): The number of bits to be encrypted/decrypted (1-20000)
IBS	(1-20000-bits): Input bit stream $IBS[0]...IBS[LENGTH-1]$

The output of the $f8$ function is:

OBS (1-20000-bits): Output bit stream $OBS[0]...OBS[LENGTH-1]$

The initialization of the keystream generator begins with setting the 64-bit register A to the concatenation of:

$A = COUNT||BEARER||DIRECTION||0...0$
$A = COUNT[0]...COUNT[32]BEARER[0]...BEARER[4]DIRECTION[0]0...0$

where the right most 26-bits are set to 0. $BLKCNT$ is set to 0 and we define a 128-bit key modifier KM as:

$$KM = 55555555555555555555555555555555$$

Define KSB_i to be the i^{th} block of keystream produced by the keystream generator. Thus for initialization KSB_0 is set to 0. Define $KASUMI[x]_K$ as the output of the KASUMI algorithm applied to input value x using the key K. Therefore to initialize the $f8$ algorithm one operation of $KASUMI$ is applied to the register A using a modified version of the confidentiality key $CK \oplus KM$:

$$A = KASUMI[A]_{CK \oplus KM}$$

Once the keystream generator has been initialized it is ready to produce the keystream bits. Since the plaintext to be encrypted consists of $LENGTH$ bits (1-20000) and the keystream generator produces keystream bits in multiples of 64-bits, between 0 and 63 of the least significant bits are discarded from the last block depending on the total number of bits required by $LENGTH$. Thus set $BLOCKS$ equal to $LENGTH/64$ rounded up to the nearest integer. If $LENGTH = 192$ then $BLOCKS = 3$. If $LENGTH = 193$ then $BLOCKS = 4$. For each integer n with $1 \leq n \leq BLOCKS$ and $BLKCNT = n - 1$ we can generate each keystream block (KSB) by defining:

$$KSB_n \quad = \quad KASUMI[A \oplus BLKCNT \oplus KSB_{n-1}]_{CK}$$

The individual bits of the keystream are extracted from KSB_1 to KSB_{BLOCKS} in turn by the most significant bit first. For $n = 1$ to $BLOCKS$ and for each integer i with $0 \leq i \leq 63$ we define:

$$KS[((n-1)*64)+1] \quad = \quad KSB_n[i]$$

where $KS[i]$ is the i^{th} bit of keystream produced by the keystream generator.

Once the keystream is produced, the encryption of plaintext can be performed by the exclusive XOR of the input data (IDS) with the generated keystream (KS). For each integer i with $0 \leq i \leq LENGTH - 1$ we define:

$$OBS[i] = IBS[i] \oplus KS[i]$$

A diagram of the process is given in Figure 4.5

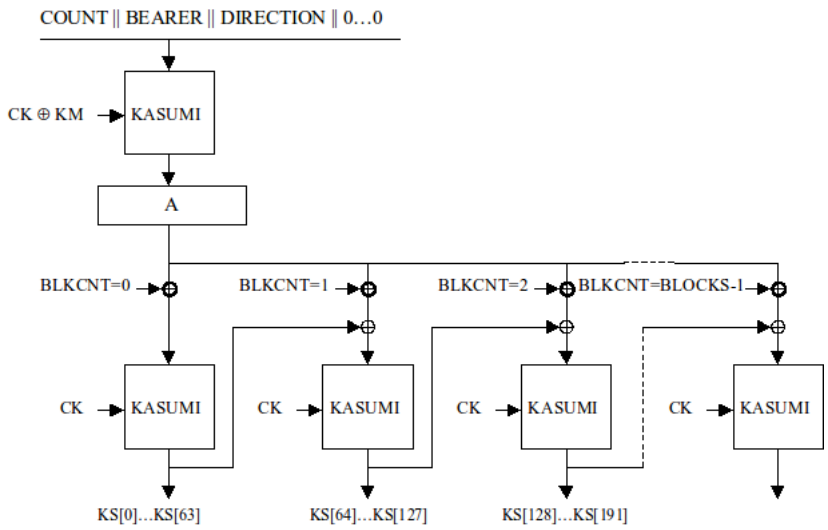

Figure 4.5. $f8$ **encryption function. Source: G.T. 35.201,** ***Specification of the 3gpp confidentiality and integrity algorithms*****; Document 1: f8 and f9 specifications, technical report, 3GPP, 2012.**

The KASUMI algorithm is a Feistel cipher with eight rounds. KASUMI decomposes into a number of subfunctions FL, FO, and FI which are used in conjunction with associated sub-keys KL, KO, and KI. The FO and FI subfunctions also contain rounds within the rounds, thus $XX_{i,j}$ represents the function XX where i is the outer round number of $KASUMI$ and j is the inner round number of the subfunction. Thus $FI_{4,2}$ refers to the fourth round of KASUMI and the second round of the subfunction FI. Also define $f_i()$ to be the i^{th} round of the KASUMI function. The encryption process of $KASUMI$ begins with a 64-bit input I along with a 128-bit key K and produces a 64-bit output $OUTPUT$. The input I is divided into two 32-bit strings L_0 and R_0 where:

$$I = L_0||R_0$$

Then for each integer i with $1 \leq i \leq 8$ we define:

$$R_i = L_{i-1}, \qquad L_i = R_{i-1} \oplus f_i(L_{i-1}, RK_i)$$

This constitutes the i^{th} round function of $KASUMI$, where f_i denotes the round function with L_{i-1} and round key RK_i as inputs. The $OUTPUT$ at the end of eight rounds is equal to the 64-bit string $(L_8||R_8)$. Thus

$$OUTPUT = KASUMI[I]_K$$

f_i Function

The f_i function takes a 32-bit input I and returns a 32-bit output O under the control of a round key RK_i, where the round key comprises the subkey triplet of (KL_i, KO_i, KI_i). The function itself is constructed from two subfunctions; FL and FO with associated subkeys KL_i (used with FL) and subkeys KO_i and KI_i (used with FO). The f_i function has two different forms depending on whether it is an even round or an odd round. For rounds 1,3,5, and 7 we have:

$$f_i(I, RK_i) = FO(FL(I, KL_i), KO_i, KI_i)$$

and for rounds 2,4,6, and 8 we have:

$$f_i(I, RK_i) = FL(FO(I, KO_i, KI_i), KL_i)$$

Thus for odd rounds the round data is passed through $FL()$ and then $FO()$, while for even rounds it is passed through $FO()$ and then $FL()$.

FL Function

The input into the FL function consists of a 32-bit data input I and a 32-bit subkey KL_i. The subkey is split into two 16-bit subkeys, $KL_{i,1}$ and $KL_{i,2}$ where:

$$KL_i = KL_{i,1}||KL_{i,2}.$$

The input data I is split into two 16-bit halves, L and R where $I = L||R$. Define:

$$R' = R \oplus ROL(L \cap KL_{i,1})$$
$$L' = L \oplus ROL(R' \cup KL_{i,2})$$

where $ROL()$ is the left circular rotation of the operand by one bit. The 32-bit output of the FL function is the value $(L'||R')$.

FO Function

The input into the FO function consists of a 32-bit data input I and two sets of subkeys, a 48-bit subkey KO_i and 48-bit subkey KI_i. The 32-bit input is split into two halves, L_0 and R_0 where $I = L_0||R_0$. The 48-bit subkeys are subdivided into three 16-bit subkeys where:

$$\begin{aligned} KO_i &= KO_{i,1}||KO_{i,2}||KO_{i,3} \text{ and} \\ KI_i &= KI_{i,1}||KI_{i,2}||KI_{i,3}. \end{aligned}$$

Then for each integer j with $1 \leq j \leq 3$ define:

$$\begin{aligned} R_j &= FI(L_{j-1} \oplus KO_{i,j}, KI_{i,j}) \oplus R_{j-1} \\ L_j &= R_{j-1} \end{aligned}$$

The 32-bit output of the FO function is the value $(L_3||R_3)$.

FI Function

The input for the FI function consists of a 16-bit input I and 16-bit subkey $KI_{i,j}$. The input I is split into two unequal components, a 9-bit left half L_0 and a 7-bit right half R_0 where $I = L_0||R_0$.The key $KI_{i,j}$ is similarly split into a 7-bit component $KI_{i,j,1}$ and a 9-bit component $KI_{i,j,2}$ where $KI_{i,j} = KI_{i,j,1}||KI_{i,j,2}$. The FI function uses two S-boxes: $S7$ which maps a 7-bit input to a 7-bit output and $S9$ which maps a 9-bit input to a 9-bit output. The $S7$ and $S9$ S-boxes are defined below in Table 4.3 and Table 4.4. The FI function also uses two additional functions $ZE()$ and $TR()$ where:

- $ZE(x)$ takes the 7-bit value x and converts it to a 9-bit value by adding two zero bits to the most-significant end.
- $TR(x)$ takes the 9-bit value x and converts it to a 7-bit value by discarding the two most-significant bits.

Define the following series of operations:

$$\begin{aligned} L_1 &= R_0 & R_1 &= S9[L_0] \oplus ZE(R_0) \\ L_2 &= R_1 \oplus KI_{i,j,2} & R_2 &= S7[L_1] \oplus TR(R_1) \oplus KI_{i,j,1} \\ L_3 &= R_2 & R_3 &= S9[L_2] \oplus ZE(R_2) \\ L_4 &= S7[L_3] \oplus TR(R_3) & R_4 &= R_3 \end{aligned}$$

The 16-bit output for the FI function is the value $(L_4||R_4)$.

S-Boxes

The two S-Boxes $S7$ and $S9$ have been designed to be implemented as either lookup tables or by using combinational logic. The input x consists of either seven or nine bits with a corresponding number of bits in the output y. Thus:

$$x = x_8||x_7||x_6||x_5||x_4||x_3|x_2||x_1||x_0$$

and

$$y = y_8||y_7||y_6||y_5||y_4||y_3|y_2||y_1||y_0$$

where the x_8, y_8 and x_7, y_7 bits only apply to $S9$, and the x_0 and y_0 bits are the least significant bits. In the logical equations below $x_0x_1x_2$ implies $x_0 \cap x_1 \cap x_2$ where $\cap$ is the AND operator. To compute the combinational logic for the $S7$ S-Box we have:

$$\begin{aligned}
y_0 = \;& x_1x_3 \oplus x_4 \oplus x_0x_1x_4 \oplus x_5 \oplus x_2x_5 \oplus x_3x_4x_5 \oplus x_6 \oplus x_0x_6 \oplus x_1x_6 \oplus x_3x_6 \\
& \oplus x_2x_4x_6 \oplus x_1x_5x_6 \oplus x_4x_5x_6 \\
y_1 = \;& x_0x_1 \oplus x_0x_4 \oplus x_2x_4 \oplus x_5 \oplus x_1x_2x_5 \oplus x_0x_3x_5 \oplus x_6 \oplus x_0x_2x_6 \oplus x_3x_6 \\
& \oplus x_4x_5x_6 \oplus 1 \\
y_2 = \;& x_0 \oplus x_0x_3 \oplus x_2x_3 \oplus x_1x_2x_4 \oplus x_0x_3x_4 \oplus x_1x_5 \oplus x_0x_2x_5 \oplus x_0x_6 \oplus x_0x_1x_6 \\
& \oplus x_2x_6 \oplus x_4x_6 \oplus 1 \\
y_3 = \;& x_1 \oplus x_0x_1x_2 \oplus x_1x_4 \oplus x_3x_4 \oplus x_0x_5 \oplus x_0x_1x_5 \oplus x_2x_3x_5 \oplus x_1x_4x_5 \oplus x_2x_6 \\
& \oplus x_1x_3x_6 \\
y_4 = \;& \oplus x_0x_2 \oplus x_3 \oplus x_1x_3 \oplus x_1x_4 \oplus x_0x_1x_4 \oplus x_2x_3x_4 \oplus x_0x_5 \oplus x_1x_3x_5 \oplus x_0x_4x_5 \\
& \oplus x_1x_6 \oplus x_3x_6 \oplus x_0x_3x_6 \oplus x_5x_6 \oplus 1 \\
y_5 = \;& x_2 \oplus x_0x_2 \oplus x_0x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_4 \oplus x_0x_5 \oplus x_2x_5 \oplus x_4x_5 \oplus x_1x_6 \\
& \oplus x_1x_2x_6 \oplus x_0x_3x_6 \oplus x_3x_4x_6 \oplus x_2x_5x_6 \oplus 1 \\
y_6 = \;& x_1x_2 \oplus x_0x_1x_3 \oplus x_0x_4 \oplus x_1x_5 \oplus x_3x_5 \oplus x_6 \oplus x_0x_1x_6 \oplus x_2x_3x_6 \oplus x_1x_4x_6 \\
& \oplus x_0x_5x_6
\end{aligned}$$

The $S7$ S-Box lookup table is given in Table 4.3. Using the $S7$ lookup table if we have

Table 4.3. $S7$ **S-Box Lookup Table Used in the KASUMI Algorithm Source: G.T. 35.202,** ***Specification of the 3gpp confidentiality and integrity algorithms; document 2: Kasumi*****, technical report, 3GPP, 2012.**

54	50	62	56	22	34	94	96	38	6	63	93	2
18	123	33	55	113	39	114	21	67	65	12	47	73
46	27	25	111	124	81	53	9	121	79	52	60	58
48	101	127	40	120	104	70	71	43	20	122	72	61
23	109	13	100	77	1	16	7	82	10	105	98	117
116	76	11	89	106	0	125	118	99	86	69	30	57
126	87	112	51	17	5	95	14	90	84	91	8	35
103	32	97	28	66	102	31	26	45	75	4	85	92
37	74	80	49	68	29	115	44	64	107	108	24	110
83	36	78	42	19	15	41	88	119	59	3		

the input value 38, then $S7[38] = 58$. Using the combination logic we have:

$$38 = 0100110_2 \Rightarrow x6 = 0, x5 = 1, x4 = 0, x3 = 0, x2 = 1, x1 = 1, x0 = 0$$

which gives:

$$\begin{aligned}
y_0 = &\quad 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 && = 0\\
y_1 = &\quad 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 && = 1\\
y_2 = &\quad 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 && = 0\\
y_3 = &\quad 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 && = 1\\
y_4 = &\quad 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 && = 1\\
y_5 = &\quad 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 && = 1\\
y_6 = &\quad 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 && = 0
\end{aligned}$$

Thus $y = 0111010_2 = 58$.

To compute the combinational logic for the $S9$ S-Box we have:

$$\begin{aligned}
y_0 = &\quad x_0x_2 \oplus x_3 \oplus x_2x_5 \oplus x_5x_6 \oplus x_0x_7 \oplus x_1x_7 \oplus x_2x_7 \oplus x_4x_8 \oplus x_5x_8\\
&\quad \oplus x_7x_8 \oplus 1\\
y_1 = &\quad x_1 \oplus x_0x_1 \oplus x_2x_3 \oplus x_0x_4 \oplus x_1x_4 \oplus x_0x_5 \oplus x_3x_5 \oplus x_6 \oplus x_1x_7\\
&\quad \oplus x_2x_7 \oplus x_5x_8 \oplus 1\\
y_2 = &\quad x_1 \oplus x_0x_3 \oplus x_3x_4 \oplus x_0x_5 \oplus x_2x_6 \oplus x_3x_6 \oplus x_5x_6 \oplus x_4x_7 \oplus x_5x_7\\
&\quad \oplus x_6x_7 \oplus x_8 \oplus x_0x_8 \oplus 1\\
y_3 = &\quad x_0 \oplus x_1x_2 \oplus x_0x_3 \oplus x_2x_4 \oplus x_5 \oplus x_0x_6 \oplus x_1x_6 \oplus x_4x_7 \oplus x_0x_8\\
&\quad \oplus x_1x_8 \oplus x_7x_8\\
y_4 = &\quad x_0x_1 \oplus x_1x_3 \oplus x_4 \oplus x_0x_5 \oplus x_3x_6 \oplus x_0x_7 \oplus x_6x_7 \oplus x_1x_8 \oplus x_2x_8\\
&\quad \oplus x_3x_8\\
y_5 = &\quad x_2 \oplus x_1x_4 \oplus x_4x_5 \oplus x_0x_6 \oplus x_1x_6 \oplus x_3x_7 \oplus x_4x_7 \oplus x_6x_7 \oplus x_5x_8\\
&\quad \oplus x_6x_8 \oplus x_7x_8 \oplus 1\\
y_6 = &\quad x_0 \oplus x_2x_3 \oplus x_1x_5 \oplus x_2x_5 \oplus x_4x_5 \oplus x_3x_6 \oplus x_4x_6 \oplus x_5x_6 \oplus x_7\\
&\quad \oplus x_1x_8 \oplus x_3x_8 \oplus x_5x_8 \oplus x_7x_8\\
y_7 = &\quad x_0x_1 \oplus x_0x_2 \oplus x_1x_2 \oplus x_3 \oplus x_0x_3 \oplus x_2x_3 \oplus x_4x_5 \oplus x_2x_6 \oplus x_3x_6\\
&\quad \oplus x_2x_7 \oplus x_5x_7 \oplus x_8 \oplus 1\\
y_8 = &\quad x_0x_1 \oplus x_2 \oplus x_1x_2 \oplus x_3x_4 \oplus x_1x_5 \oplus x_2x_5 \oplus x_1x_6 \oplus x_4x_6 \oplus x_7\\
&\quad \oplus x_2x_8 \oplus x_3x_8
\end{aligned}$$

The $S9$ S-Box lookup table is given in Table 4.4. If we have the input value 138, then using the lookup table $S9$ we have $S9[138] = 339$. For the combination logic we have:

$$\begin{aligned}
138 = 010001010_2 \quad \Rightarrow \quad & x_8 = 0, x_7 = 1, x_6 = 0, x_5 = 0, x_4 = 0, x_3 = 1,\\
& x_2 = 0, x_1 = 1, x_0 = 0
\end{aligned}$$

which gives:

Table 4.4. $S9$ **S-Box Lookup Table Used in the KASUMI Algorithm Source: G.T. 35.202,** ***Specification of the 3gpp confidentiality and integrity algorithms; document 2: Kasumi*****, technical report, 3GPP, 2012.**

167	239	161	379	391	334	9	338	38	226	48	358	452	385
90	397	183	253	147	331	415	340	51	362	306	500	262	82
216	159	356	177	175	241	489	37	206	17	0	333	44	254
378	58	143	220	81	400	95	3	315	245	54	235	218	405
472	264	172	494	371	290	399	76	165	197	395	121	257	480
423	212	240	28	462	176	406	507	288	223	501	407	249	265
89	186	221	428	164	74	440	196	458	421	350	163	232	158
134	354	13	250	491	142	191	69	193	425	152	227	366	135
344	300	276	242	437	320	113	278	11	243	87	317	36	93
496	27	487	446	482	41	68	156	457	131	326	403	339	20
39	115	442	124	475	384	508	53	112	170	479	151	126	169
73	268	279	321	168	364	363	292	46	499	393	327	324	24
456	267	157	460	488	426	309	229	439	506	208	271	349	401
434	236	16	209	359	52	56	120	199	277	465	416	252	287
246	6	83	305	420	345	153	502	65	61	244	282	173	222
418	67	386	368	261	101	476	291	195	430	49	79	166	330
280	383	373	128	382	408	155	495	367	388	274	107	459	417
62	454	132	225	203	316	234	14	301	91	503	286	424	211
347	307	140	374	35	103	125	427	19	214	453	146	498	314
444	230	256	329	198	285	50	116	78	410	10	205	510	171
231	45	139	467	29	86	505	32	72	26	342	150	313	490
431	238	411	325	149	473	40	119	174	355	185	233	389	71
448	273	372	55	110	178	322	12	469	392	369	190	1	109
375	137	181	88	75	308	260	484	98	272	370	275	412	111
336	318	4	504	492	259	304	77	337	435	21	357	303	332
483	18	47	85	25	497	474	289	100	269	296	478	270	106
31	104	433	84	414	486	394	96	99	154	511	148	413	361
409	255	162	215	302	201	266	351	343	144	441	365	108	298
251	34	182	509	138	210	335	133	311	352	328	141	396	346
123	319	450	281	429	228	443	481	92	404	485	422	248	297
23	213	130	466	22	217	283	70	294	360	419	127	312	377
7	468	194	2	117	295	463	258	224	447	247	187	80	398
284	353	105	390	299	471	470	184	57	200	348	63	204	188
33	451	97	30	310	219	94	160	129	493	64	179	263	102
189	207	114	402	438	477	387	122	192	42	381	5	145	118
180	449	293	323	136	380	43	66	60	455	341	445	202	432
8	237	15	376	436	464	59	461						

$$
\begin{aligned}
y_0 = &\ 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 && = 1 \\
y_1 = &\ 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 && = 1 \\
y_2 = &\ 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 && = 0 \\
y_3 = &\ 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 && = 0 \\
y_4 = &\ 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 && = 1 \\
y_5 = &\ 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 && = 0 \\
y_6 = &\ 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 && = 1 \\
y_7 = &\ 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 && = 0 \\
y_8 = &\ 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 && = 1
\end{aligned}
$$

Thus $y = 101010011_2 = 339$.

Key Schedule

KASUMI has a 128-bit key K. Each round of KASUMI also uses a 128-bit round key that is derived from K and the following constants:

C_1	=	0123	C_5	=	FEDC
C_2	=	4567	C_6	=	BA98
C_3	=	89AB	C_7	=	7654
C_4	=	CDEF	C_8	=	3210

Before the round keys can be calculated two 16-bit arrays Kj and Kj', for $1 \leq j \leq 8$, are derived from the key K. The 128-bit key K is subdivided into eight 16-bit values $K1...K8$ where:

$$K = K1||K2||K3||...||K8.$$

A second array of subkeys, Kj' is derived from Kj for each integer j with $1 \leq j \leq 8$ by applying:

$$Kj' = Kj \oplus Cj$$

The round subkeys are then derived from Kj and Kj' shown if Table 4.5. Note that $<<< n$ is the left circular rotation of the operand by n bits.

4.2.2 Integrity Check: $f9$

The integrity algorithm used in the UMTS system is referred to as $f9$. The $f9$ algorithm computes a Message Authentication Code (MAC) on an input message under an integrity key IK. There is no limitation on the input message length for the $f9$ algorithm. For ease of implementation the $f9$ algorithm is based on the same block cipher KASUMI which is

Table 4.5. Round Subkey Generation Used in the KASUMI Algorithm Source: G.T. 35.202, *Specification of the 3gpp confidentiality and integrity algorithms; document 2: Kasumi*, technical report, 3GPP, 2012.

	Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8
$KL_{i,1}$	$K1 <<< 1$	$K2 <<< 1$	$K3 <<< 1$	$K4 <<< 1$	$K5 <<< 1$	$K6 <<< 1$	$K7 <<< 1$	$K8 <<< 1$
$KL_{i,2}$	$K3'$	$K4'$	$K5'$	$K6'$	$K7'$	$K8'$	$K1'$	$K2'$
$KO_{i,1}$	$K2 <<< 5$	$K3 <<< 5$	$K4 <<< 5$	$K5 <<< 5$	$K6 <<< 5$	$K7 <<< 5$	$K8 <<< 5$	$K1 <<< 5$
$KO_{i,2}$	$K6 <<< 8$	$K7 <<< 8$	$K8 <<< 8$	$K1 <<< 8$	$K2 <<< 8$	$K3 <<< 8$	$K4 <<< 8$	$K5 <<< 8$
$KO_{i,3}$	$K7 <<< 13$	$K8 <<< 13$	$K1 <<< 13$	$K2 <<< 13$	$K3 <<< 13$	$K4 <<< 13$	$K5 <<< 13$	$K6 <<< 13$
$KI_{i,1}$	$K5'$	$K6'$	$K7'$	$K8'$	$K1'$	$K2'$	$K3'$	$K4'$
$KI_{i,2}$	$K4'$	$K5'$	$K6'$	$K7'$	$K8'$	$K1'$	$K2'$	$K3'$
$KI_{i,3}$	$K8'$	$K1'$	$K2'$	$K3'$	$K4'$	$K5'$	$K6'$	$K7'$

used by the confidentiality algorithm $f8$ described above. The inputs for the $f9$ algorithm consist of:

$COUNT - I$	(32-bits) Frame dependent input $COUNT - I[0]...COUNT - I[31]$
$FRESH$	(32-bits) Random number $FRESH[0]...FRESH[31]$
$DIRECTION$	(1-bit) Direction of transmission $DIRECTION[0]$
IK	(128-bits) Integrity Key $IK[0]...IK[127]$
$LENGTH$	(X19) The number of bits to be 'MAC'd
$MESSAGE$	($LENGTH$-bits) Input bit stream

The output for the $f9$ algorithm is:

$MAC - I$ (32-bits) Message authentication code $MAC - i[0]...MAC - I[31]$

Since the KASUMI algorithm produces a 64-bit output only the leftmost 32-bits are taken for the $MAC - I$.

The initialization of the $f9$ algorithm begins by defining two 64-bit registers A and B and setting both equal to zero. The key modifier KM is defined as:

$$KM = AAAA\ AAAA\ AAAA\ AAAA\ AAAA\ AAAA\ AAAA\ AAAA$$

The inputs $COUNT, FRESH, MESSAGE$, and $DIRECTION$ are all concatenated. Then a single '1' bit is appended followed by between 0 and 63 '0' bits so that the total length of the resulting padded string PS is an integral multiple of 64 bits. Thus:

$$\begin{aligned} PS = \quad & COUNT[0]...COUNT[31]FRESH[0]...FRESH[31]MESSAGE[0]... \\ & MESSAGE[LENGTH - 1]DIRECTION[0]10^* \end{aligned}$$

where 0^* indicates between 0 and 63 '0' bits. The $f9$ algorithm is now initialized.

The calculation of the $MAC - I$ begins by splitting the padded string PS into 64-bit blocks PS_i where:

$$PS = \quad PS_0||PS_1||PS_2||...||PS_{BLOCKS-1}$$

Then for each integer n with $0 \leq n \leq BLOCKS - 1$:

$$\begin{aligned} A = &\quad KASUMI[A \oplus PS_n]_{IK} \\ B = &\quad B \oplus A \end{aligned}$$

Finally one more application of KASUMI using a modified form of the integrity key IK is performed:

$$B = \quad KASUMI[B]_{IK \oplus KM}$$

The 32-bit $MAC - I$ comprises the left-most 32-bits of the result.

$$MAC - I = \quad \text{left half } [B]$$

Thus for each integer i with $0 \leq i \leq 31$ define:

$$MAC - I[i] = B[i]$$

Bits $B[32]...B[63]$ are discarded.

4.2.3 KASUMI Example

A diagram of the KASUMI algorithm along with its subfunctions is given in Figure 4.6. For an example of the KASUMI algorithm the test vector given by the 3GPP will be used:

Key: $2BD6\ 459F\ 82C5\ B300\ 952C\ 4910\ 4881\ FF48$
Input: $EA02\ 4714\ AD5C\ 4D84$

Key Schedule

The 128-bit Key K is subdivided into eight 16-bit values $k1...K8$

$$\begin{aligned} K = &\quad K1||K2||K = K1||K2||K3||...||K8. \\ = &\quad 2BD6||459F||82C5||B300||952C||4910||4881||FF48 \end{aligned}$$

A second array of subkeys Kj' is derived from Kj:

$$\begin{aligned} Kj' = &\quad Kj \oplus Cj \\ = &\quad 2BD6 \oplus 0123||459F \oplus 4567||82C5 \oplus 89AB||B300 \oplus CDEF|| \\ &\quad 952C \oplus FEDC||4910 \oplus BA98||4881 \oplus 7654||FF48 \oplus 3210 \\ = &\quad 2AF5\ 00F8\ 0B6E\ 7EEF\ 6BF0\ F388\ 3ED5\ CD58 \end{aligned}$$

The round subkeys are then derived from Kj and Kj' as described in Table 4.5:

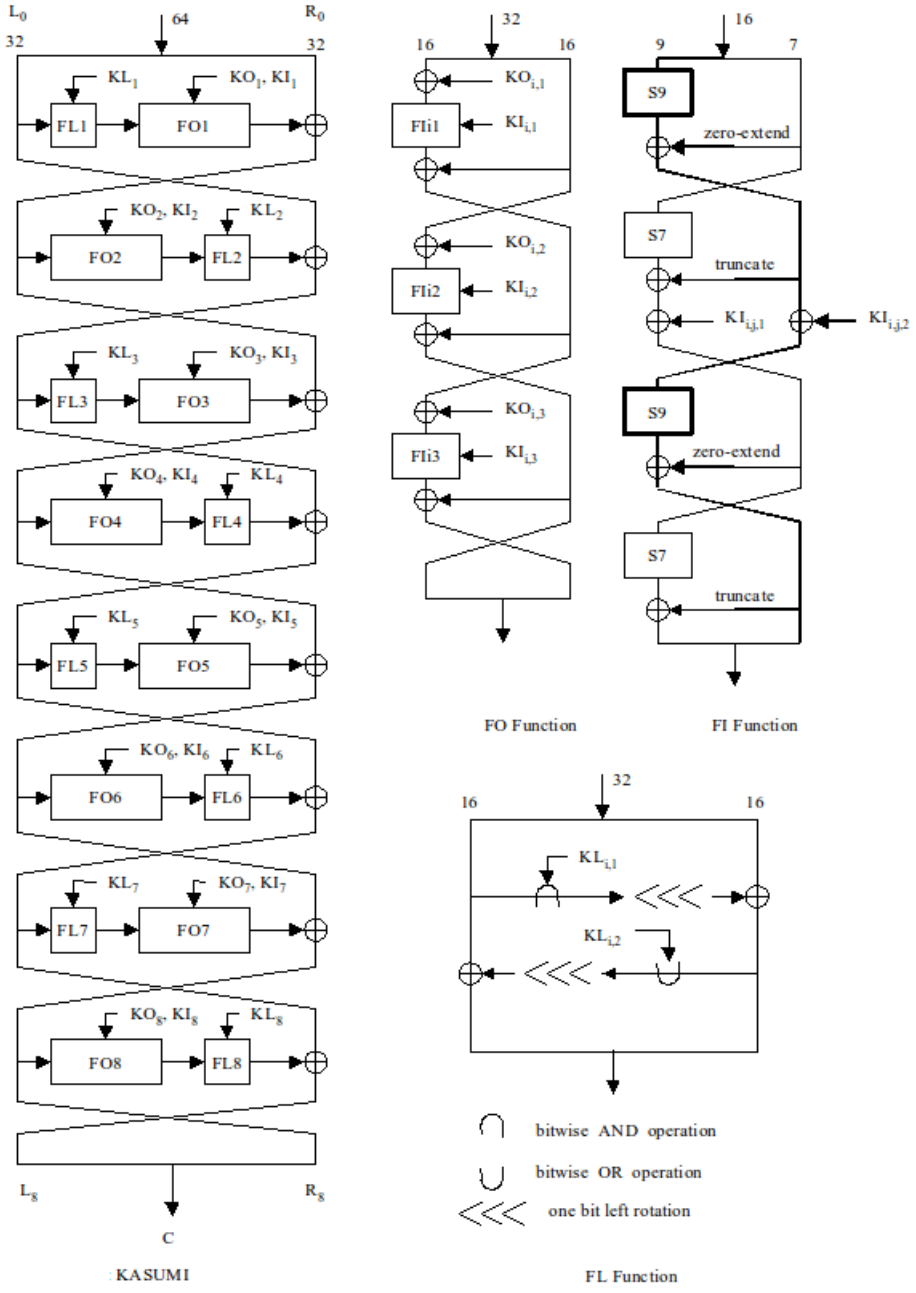

Figure 4.6. KASUMI algorithm and subfunctions. Source: G.T. 35.202, *Specification of the 3gpp confidentiality and integrity algorithms; Document 2: Kasumi*, technical report, 3GPP, 2012.

$$\begin{aligned}
KL_{1,1} &= K1 <<< 1 \\
&= 2BD6 <<< 1 \\
&= 0010101111010110 <<< 1 \\
&= 0101011110101100 \\
&= 57AC \\
KO_{1,1} &= K2 <<< 5 \\
&= 459F <<< 5 \\
&= 0100010110011111 <<< 5 \\
&= 1011001111101000 \\
&= B3E8 \\
KI_{1,1} &= K5' \\
&= 6BF0
\end{aligned}$$

The rest of the round subkeys are shown in Table 4.6.

Table 4.6. Round Subkey Generation Example

	Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8
$KL_{i,1}$	$57AC$	$8B3E$	$058B$	6601	$2A59$	9220	9102	$FE91$
$KL_{i,2}$	$0B6E$	$7EEF$	$6BF0$	$F388$	$3ED5$	$CD58$	$2AF5$	$00F8$
$KO_{i,1}$	$B3E8$	$58B0$	6016	$A592$	2209	1029	$E91F$	$7AC5$
$KO_{i,2}$	1049	8148	$48FF$	$D62B$	$9F45$	$C582$	$00B3$	$2C95$
$KO_{i,3}$	2910	$1FE9$	$C57A$	$E8B3$	$B058$	1660	$92A5$	0922
$KI_{i,1}$	$6BF0$	$F388$	$3ED5$	$CD58$	$2AF5$	$00F8$	$0B6E$	$7EEF$
$KI_{i,2}$	$7EEF$	$6BF0$	$F388$	$3ED5$	$CD58$	$2AF5$	$00F8$	$0B6E$
$KI_{i,3}$	$CD58$	$2AF5$	$00F8$	$0B6E$	$7EEF$	$6BF0$	$F388$	$3ED5$

Round 1

The input I is divided into two 32-bit strings L_0 and R_0 where:

$$\begin{aligned}
I &= L_0||R_0 \\
&= EA02\ 4714||AD5C\ 4D84
\end{aligned}$$

Thus:

$$\begin{aligned}
R_1 &= L_0 \\
&= EA02\ 4714 \\
L_1 &= R_0 \oplus f_1(L_0, RK_1) \\
&= AD5C\ 4D84 \oplus f_1(EA02\ 4714, RK_1)
\end{aligned}$$

For round 1:

$$\begin{aligned} f_1(EA02\ 4714, RK_1) = &\quad FO(FL(EA02\ 4714, KL_1), KO_1, KI_1) \\ = &\quad FO(FL(EA02\ 4714, 57AC\ 0B6E), \\ &\quad B3E8\ 1049\ 2910, 6BF0\ 7EEF\ CD58) \end{aligned}$$

Computing the FL function:

The input data I is split into two halves, L and R where $I = L||R$:

$$\begin{aligned} L = &\quad EA02 \\ R = &\quad 4714 \end{aligned}$$

Thus:

$$\begin{aligned} R' = &\quad R \oplus ROL(L \cap KL_{1,1}) \\ = &\quad 4714 \oplus ROL(EA02 \cap 57AC) \\ = &\quad 4714 \oplus ROL(4200) \\ = &\quad 4714 \oplus 8400 \\ = &\quad C314 \end{aligned} \qquad \begin{aligned} L' = &\quad L \oplus ROL(R' \cup KL_{1,2}) \\ = &\quad EA02 \oplus ROL(C314 \cup 0B6E) \\ = &\quad EA02 \oplus ROL(CB7E) \\ = &\quad EA02 \oplus 96FD \\ = &\quad 7CFF \end{aligned}$$

Therefore the output for the FL function is:

$$\begin{aligned} FL(EA02\ 4714, 57AC\ 0B6E) = &\quad L'||R' \\ = &\quad 7CFF\ C314 \end{aligned}$$

Computing the FO function:

Inputting the results from the FL function we have:

$$FO(7CFF\ C314, KO_1, KI_1) \quad = \quad FO(7CFF\ C314, B3E8\ 1049\ 2910, 6BF0\ 7EEF\ CD58)$$

The 32-bit data input is split into two halves, L_0 and R_0 where $I = L_0||R_0$:

$$\begin{aligned} L_0 = &\quad 7CFF \\ R_0 = &\quad C314 \end{aligned}$$

The 48-bit subkeys are subdivided into three 16-bit subkeys where:

$$\begin{aligned} KO_1 = &\quad KO_{1,1}||KO_{1,2}||KO_{1,3} \\ = &\quad B3E8||1049||2910 \end{aligned} \qquad \begin{aligned} KI_1 = &\quad KI_{1,1}||KI_{1,2}||KI_{1,3} \\ = &\quad 6BF0||7EEF||CD58 \end{aligned}$$

For $j = 1$:

$$\begin{aligned} R_1 &= FI(L_0 \oplus KO_{1,1}, KI_{1,1}) \oplus R_0 \\ &= FI(7CFF \oplus B3E8, 6BF0) \oplus C314 \\ &= FI(CF17, 6BF0) \oplus C314 \\ &= 43CD \oplus C314 \text{ (See } FI \text{ section below for computation)} \\ &= 80D9 \\ L_1 &= R_0 \\ &= C314 \end{aligned}$$

For $j = 2$:

$$\begin{aligned} R_2 &= FI(L_1 \oplus KO_{1,2}, KI_{1,2}) \oplus R_1 \\ &= FI(C314 \oplus 1049, 7EEF) \oplus 80D9 \\ &= FI(D35D, 7EEF) \oplus 80D9 \\ &= D85E \oplus 80D9 \text{(See } FI \text{ section below for computation)} \\ &= 5887 \\ L_2 &= R_1 \\ &= 80D9 \end{aligned}$$

For $j = 3$:

$$\begin{aligned} R_3 &= FI(L_2 \oplus KO_{1,3}, KI_{1,3}) \oplus R_2 \\ &= FI(80D9 \oplus 2910, CD58) \oplus 5887 \\ &= FI(A9C9, CD58) \oplus 5887 \\ &= 4FB0 \oplus 5887 \text{(See } FI \text{ section below for computation)} \\ &= 1737 \\ L_3 &= R_2 \\ &= 5887 \end{aligned}$$

The 32-bit output of the FO function is the value $(L_3||R_3)$. Thus:

$$\begin{aligned} FO(7CFF\ C314, B3E8\ 1049\ 2910, 6BF0\ 7EEF\ CD58) &= L_3||R_3 \\ &= 5887\ 1737 \end{aligned}$$

Computing the FI function:

The FI function is a subfunction of the FO function. For the case $j = 1$, $FI(CF17, 6BF0)$ is computed by taking the 16-bit data input $CF17$ and splitting it into two unequal components of 9-bits and 7-bits:

$$\begin{aligned} L_0 &= 110011110 \\ R_0 &= 0010111 \end{aligned}$$

Similarly $KI_{1,1}$ is split into a 7-bit and 9-bit component:

$$\begin{aligned} KI_{1,1} = &\quad KI_{1,1,1}||KI_{1,1,2} \\ = &\quad 0110101||111110000 \end{aligned}$$

Thus we have:

$$\begin{aligned} L_1 = &\quad R_0 \\ = &\quad 0010111 \\ \\ \\ L_2 = &\quad R_1 \oplus KI_{1,1,2} \\ = &\quad 001001011 \oplus 111110000 \\ = &\quad 110111011 \\ \\ L_3 = &\quad R_2 \\ = &\quad 1110010 \\ \\ \\ L_4 = &\quad S7[L_3] \oplus TR(R_3) \\ = &\quad S7[1110010] \oplus TR(111001101) \\ = &\quad 1101100 \oplus 1001101 \\ = &\quad 0100001 \end{aligned}$$

$$\begin{aligned} R_1 = &\quad S9[L_0] \oplus ZE(R_0) \\ = &\quad S9[110011110] \oplus ZE(0010111) \\ = &\quad 001011100 \oplus 000010111 \\ = &\quad 001001011 \\ R_2 = &\quad S7[L_1] \oplus TR(R_1) \oplus KI_{1,1,1} \\ = &\quad S7[0010111] \oplus TR(001001011) \oplus 0110101 \\ = &\quad 0001100 \oplus 1001011 \oplus 0110101 \\ = &\quad 1110010 \\ R_3 = &\quad S9[L_2] \oplus ZE(R_2) \\ = &\quad S9[110111011] \oplus ZE(1110010) \\ = &\quad 110111111 \oplus 001110010 \\ = &\quad 111001101 \\ R_4 = &\quad R_3 \\ = &\quad 111001101 \\ = &\quad 111001101 \end{aligned}$$

For $j = 1$ the output for the FI function is:

$$\begin{aligned} FI(CF17, 6BF0) = &\quad L_4||R_4 \\ = &\quad 0100001||111001101 \\ = &\quad 43CD \end{aligned}$$

For the case $j = 2$, $FI(D35D, 7EEF)$ is computed by taking the 16-bit data input $D35D$ and splitting it into two unequal components of 9-bits and 7-bits:

$$\begin{aligned} L_0 = &\quad 110100110 \\ R_0 = &\quad 1011101 \end{aligned}$$

Similarly $KI_{1,2}$ is split into a 7-bit and 9-bit component:

$$\begin{aligned} KI_{1,2} = &\quad KI_{1,2,1}||KI_{1,2,2} \\ = &\quad 0111111||011101111 \end{aligned}$$

Thus we have:

For $j = 2$ the output for the FI function is:

$$\begin{aligned} FI(D35D, 7EEF) = &\quad L_4||R_4 \\ = &\quad 1101100||001011110 \\ = &\quad D85E \end{aligned}$$

$$
\begin{aligned}
L_1 &= R_0 \\
&= 1011101 \\
\\
L_2 &= R_1 \oplus KI_{1,2,2} \\
&= 011011111 \oplus 011101111 \\
&= 000110000 \\
\\
L_3 &= R_2 \\
&= 0000001 \\
\\
L_4 &= S7[L_3] \oplus TR(R_3) \\
&= S7[0000001] \oplus TR(001011110) \\
&= 0110010 \oplus 1011110 \\
&= 1101100
\end{aligned}
\qquad
\begin{aligned}
R_1 &= S9[L_0] \oplus ZE(R_0) \\
&= S9[110100110] \oplus ZE(1011101) \\
&= 010000010 \oplus 001011101 \\
&= 011011111 \\
R_2 &= S7[L_1] \oplus TR(R_1) \oplus KI_{1,2,1} \\
&= S7[1011101] \oplus TR(011011111) \oplus 0111111 \\
&= 1100001 \oplus 1011111 \oplus 0111111 \\
&= 0000001 \\
R_3 &= S9[L_2] \oplus ZE(R_2) \\
&= S9[000110000] \oplus ZE(0000001) \\
&= 001011111 \oplus 000000001 \\
&= 001011110 \\
R_4 &= R_3 \\
&= 001011110
\end{aligned}
$$

For the case $j = 3$, $FI(A9C9, CD58)$ is computed by taking the 16-bit data input $A9C9$ and splitting it into two unequal components of 9-bits and 7-bits:

$$
\begin{aligned}
L_0 &= 101010011 \\
R_0 &= 1001001
\end{aligned}
$$

Similarly $KI_{1,3}$ is split into a 7-bit and 9-bit component:

$$
\begin{aligned}
KI_{1,3} &= KI_{1,3,1}||KI_{1,3,2} \\
&= 1100110||101011000
\end{aligned}
$$

Thus we have:

For $j = 3$ the output for the FI function is:

$$
\begin{aligned}
FI(A9C9, CD58) &= L_4||R_4 \\
&= 0100111||110110000 \\
&= 4FB0
\end{aligned}
$$

Output for Round 1:

$$
\begin{aligned}
R_1 &= L_0 \\
&= EA02\ 4714 \\
L_1 &= R_0 \oplus f_1(L_0, RK_1) \\
&= AD5C\ 4D84 \oplus f_1(EA02\ 4714, RK_1) \\
&= AD5C\ 4D84 \oplus FO(FL(EA02\ 4714, KL_1), KO_1, KI_1) \\
&= AD5C\ 4D84 \oplus 5887\ 1737 \\
&= F5DB\ 5AB3
\end{aligned}
$$

$$
\begin{aligned}
L_1 &= R_0 \\
&= 1001001
\end{aligned}
\qquad
\begin{aligned}
R_1 &= S9[L_0] \oplus ZE(R_0) \\
&= S9[101010011] \oplus ZE(1001001) \\
&= 111111000 \oplus 001001001 \\
&= 110110001
\end{aligned}
$$

$$
\begin{aligned}
L_2 &= R_1 \oplus KI_{1,3,2} \\
&= 110110001 \oplus 101011000 \\
&= 011101001
\end{aligned}
\qquad
\begin{aligned}
R_2 &= S7[L_1] \oplus TR(R_1) \oplus KI_{1,3,1} \\
&= S7[1001001] \oplus TR(110110001) \oplus 1100110 \\
&= 1100011 \oplus 0110001 \oplus 1100110 \\
&= 0110100
\end{aligned}
$$

$$
\begin{aligned}
L_3 &= R_2 \\
&= 0110100
\end{aligned}
\qquad
\begin{aligned}
R_3 &= S9[L_2] \oplus ZE(R_2) \\
&= S9[011101001] \oplus ZE(0110100) \\
&= 110000100 \oplus 000110100 \\
&= 110110000
\end{aligned}
$$

$$
\begin{aligned}
L_4 &= S7[L_3] \oplus TR(R_3) \\
&= S7[0110100] \oplus TR(110110000) \\
&= 0010111 \oplus 0110000 \\
&= 0100111
\end{aligned}
\qquad
\begin{aligned}
R_4 &= R_3 \\
&= 110110000
\end{aligned}
$$

Round 2

The input I is divided into two 32-bit strings L_1 and R_1 where:

$$
\begin{aligned}
I &= L_1 || R_1 \\
&= F5DB\ 5AB3 || EA02\ 4714
\end{aligned}
$$

Thus:

$$
\begin{aligned}
R_2 &= L_1 \\
&= F5DB\ 5AB3 \\
L_2 &= R_1 \oplus f_2(L_1, RK_2) \\
&= EA02\ 4714 \oplus f_2(F5DB\ 5AB3, RK_2) \\
&= EA02\ 4714 \oplus FL(FO(F5DB\ 5AB3, KO_2, KI_2), KL_2) \\
&= EA02\ 4714 \oplus FC19\ 13F5 \\
&= 161B\ 54E1
\end{aligned}
$$

Round 3

The input I is divided into two 32-bit strings L_2 and R_2 where:

$$
\begin{aligned}
I &= L_2 || R_2 \\
&= 161B\ 54E1 || F5DB\ 5AB3
\end{aligned}
$$

Thus:

$$
\begin{aligned}
R_3 &= L_2 \\
&= 161B\ 54E1 \\
L_3 &= R_2 \oplus f_3(L_2, RK_3) \\
&= F5DB\ 5AB3 \oplus f_3(161B\ 54E1, RK_3) \\
&= F5DB\ 5AB3 \oplus FO(FL(161B\ 54E1, KL_3), KO_3, KI_3) \\
&= F5DB\ 5AB3 \oplus F9C9\ DB3F \\
&= 0C12\ 818C
\end{aligned}
$$

.
.
.

Round 8

The input I is divided into two 32-bit strings L_7 and R_7 where:

$$
\begin{aligned}
I &= L_7||R_7 \\
&= 1C0B\ F45F||5E58\ EF61
\end{aligned}
$$

Thus:

$$
\begin{aligned}
R_8 &= L_7 \\
&= 1C0B\ F45F \\
L_8 &= R_7 \oplus f_8(L_7, RK_8) \\
&= 5E58\ EF61 \oplus f_8(1C0B\ F45F, RK_8) \\
&= 5E58\ EF61 \oplus FL(FO(1C0B\ F45F, KO_8, KI_8), KL_8) \\
&= 5E58\ EF61 \oplus 8147\ 7444 \\
&= DF1F\ 9B25
\end{aligned}
$$

After the eighth round the $OUTPUT$ of KASUMI is equal to the 64-bit string $L_8||R_8$. Thus:

$$
\begin{aligned}
OUTPUT &= KASUMI[I]_K \\
&= KASUMI[EA02\ 4714\ AD5C\ 4D84]_K \\
&= L_8||R_8 \\
&= DF1F\ 9B25\ 1C0B\ F45F
\end{aligned}
$$

4.2.4 $f8$ Example

An example of the 3G UMTS encryption algorithm $f8$ is given below. For the encryption example the test vector given by the 3GPP is used.The $f8$ keystream generator is based on the block cipher KASUMI. KASUMI is used in the form of output-feedback mode where feedback data is modified by static data held in a 64-bit register A and an incrementing

64-bit counter $BLKCNT$.

Initialization

The 64-bit register A is set to $COUNT||BEARER||DIRECTION||0..0$ where the right most 26-bits are set to 0. Thus:

$$A = COUNT[0]...COUNT[31]BEARER[0]...BEARER[4]DIRECTION[0]0...0$$

$BLKCNT$ and the initial keystream block KSB_0 are set to zero. The key modifier KM is defined as (hexadecimal):

$$KM = 5555\ 5555\ 5555\ 5555\ 5555\ 5555\ 5555\ 5555$$

Using the given input data from the 3GPP :

CK $= 7E83\ 10CA\ D790\ E655\ C079\ 1C45\ 1DD4\ AA1D$ (128-bits)
$COUNT$ $= 72A4\ F20F$ (32-bits)
$BEARER$ $= 0C$ (5-bits)
$DIRECTION$ $= 1$ (1-bit)
$LENGTH$ =798-bits

Plaintext:

$7EC61272743BF161$ $4726446A6C38CED1$ $66F6CA76EB543004$
$4286346CEF130F92$ $922B03450D3A9975$ $E5BD2EA0EB55AD8E$
$1B199E3EC4316020$ $E9A1B285E7627953$ $59B7BDFD39BEF4B2$
$484583D5AFE082AE$ $E638BF5FD5A60619$ $3901A08F4AB41AAB$
$9B134880$

The initial register A is set to:

$$\begin{aligned} A = &\ COUNT||BEARER||DIRECTION||0..0 \\ = &\ 72A4\ F20F||0C||1||0...0 \\ = &\ 01110010101001001111001000001111||01100||1||00000000000000000000000000 \\ = &\ 01110010101001001111001000001111011001000000000000000000000000 \\ = &\ 72A4\ F20F\ 6400\ 0000 \end{aligned}$$

The confidentiality key CK is then modified by XORing CK and the key modifier KM:

$$\begin{aligned} \text{Modified } CK = &\ CK \oplus KM \\ = &\ 7E83\ 10CA\ D790\ E655\ C079\ 1C45\ 1DD4\ AA1D \oplus \\ &\ 5555\ 5555\ 5555\ 5555\ 5555\ 5555\ 5555\ 5555 \\ = &\ 2BD6\ 459F\ 82C5\ B300\ 952C\ 4910\ 4881\ FF48 \end{aligned}$$

One operation of KASUMI is then applied to the register A using the modified version of the confidentiality key:

$$\begin{aligned} A = &\quad KASUMI[A]_{CK \oplus KM} \\ = &\quad KASUMI[72A4\ F20F\ 6400\ 0000]_{CK \oplus KM} \\ = &\quad 3422\ 2BC8\ F7C3\ 9416 \end{aligned}$$

Keystream Generation

Once the keystream has been initialized it is ready to generate keystream bits. Note that the plaintext to be encrypted consists of $LENGTH$ bits while the keystream generator produces keystream bits in multiples of 64-bits. Between 0 and 63 of the least significant bits are discarded from the last block depending on the total number of bits required by $LENGTH$. Since $LENGTH = 798$, the number of $BLOCKS$ required is:

$$\begin{aligned} BLOCKS = &\quad \lceil 798/64 \rceil \\ = &\quad 13 \end{aligned}$$

Therefore the values of $BLKCNT$ are 0, 1, 2,...,12. To generate each keystream block KSB_n we have for each integer n with $1 \leq n \leq BLOCKS$ where $BLKCNT = n - 1$:

$$KSB_n = KASUMI[A \oplus BLKCNT \oplus KSB_{n-1}]_{CK}$$

The keystream generation process begins with $n = 1$:

$n = 1$

$$\begin{aligned} KSB_1 &= KASUMI[A \oplus BLKCNT \oplus KSB_0]_{CK} \\ &= KASUMI[34222BC8F7C39416 \oplus 0 \oplus 0]_{CK} \\ &= KASUMI[34222BC8F7C39416]_{CK} \\ &= AF24CC029AC39D08 \end{aligned}$$

$n = 2$

$$\begin{aligned} KSB_2 &= KASUMI[A \oplus BLKCNT \oplus KSB_1]_{CK} \\ &= KASUMI[34222BC8F7C39416 \oplus 1 \oplus AF24CC029AC39D08]_{CK} \\ &= KASUMI[9B06E7CA6D00091F]_{CK} \\ &= 23DD1041AEECAE7B \end{aligned}$$

$n = 3$

$$\begin{aligned} KSB_3 &= KASUMI[A \oplus BLKCNT \oplus KSB_2]_{CK} \\ &= KASUMI[34222BC8F7C39416 \oplus 2 \oplus 23DD1041AEECAE7B]_{CK} \\ &= KASUMI[17FF3B89592F3A6F]_{CK} \\ &= D95CDAD24BC7162F \end{aligned}$$

$n = 4$

$$
\begin{aligned}
KSB_4 &= KASUMI[A \oplus BLKCNT \oplus KSB_3]_{CK} \\
&= KASUMI[34222BC8F7C39416 \oplus 3 \oplus D95CDAD24BC7162F]_{CK} \\
&= KASUMI[ED7EF11ABC04823A]_{CK} \\
&= 3F9FAA1C80D1DB1B
\end{aligned}
$$

$n = 5$

$$
\begin{aligned}
KSB_5 &= KASUMI[A \oplus BLKCNT \oplus KSB_4]_{CK} \\
&= KASUMI[34222BC8F7C39416 \oplus 4 \oplus 3F9FAA1C80D1DB1B]_{CK} \\
&= KASUMI[0BBD81D477124F09]_{CK} \\
&= 87782A2C1DC93006
\end{aligned}
$$

$n = 6$

$$
\begin{aligned}
KSB_6 &= KASUMI[A \oplus BLKCNT \oplus KSB_5]_{CK} \\
&= KASUMI[34222BC8F7C39416 \oplus 5 \oplus 87782A2C1DC93006]_{CK} \\
&= KASUMI[B35A01E4EA0AA415]_{CK} \\
&= E49BAC44F71B868C
\end{aligned}
$$

$n = 7$

$$
\begin{aligned}
KSB_7 &= KASUMI[A \oplus BLKCNT \oplus KSB_6]_{CK} \\
&= KASUMI[34222BC8F7C39416 \oplus 6 \oplus E49BAC44F71B868C]_{CK} \\
&= KASUMI[D0B9878C00D8129C]_{CK} \\
&= A5398989E10ADFB3
\end{aligned}
$$

$n = 8$

$$
\begin{aligned}
KSB_8 &= KASUMI[A \oplus BLKCNT \oplus KSB_7]_{CK} \\
&= KASUMI[34222BC8F7C39416 \oplus 7 \oplus A5398989E10ADFB3]_{CK} \\
&= KASUMI[911BA24116C94BA2]_{CK} \\
&= E07FEA9C2C20914A
\end{aligned}
$$

$n = 9$

$$
\begin{aligned}
KSB_9 &= KASUMI[A \oplus BLKCNT \oplus KSB_8]_{CK} \\
&= KASUMI[34222BC8F7C39416 \oplus 8 \oplus E07FEA9C2C20914A]_{CK} \\
&= KASUMI[D45DC154DBE30554]_{CK} \\
&= 0F437466F0C8A81D
\end{aligned}
$$

$n = 10$

$$\begin{aligned} KSB_1 0 &= KASUMI[A \oplus BLKCNT \oplus KSB_9]_{CK} \\ &= KASUMI[34222BC8F7C39416 \oplus 9 \oplus 0F437466F0C8A81D]_{CK} \\ &= KASUMI[3B615FAE070B3C02]_{CK} \\ &= 1BF4536E2D9900C4 \end{aligned}$$

$n = 11$

$$\begin{aligned} KSB_1 1 &= KASUMI[A \oplus BLKCNT \oplus KSB_{10}]_{CK} \\ &= KASUMI[34222BC8F7C39416 \oplus 10 \oplus 1BF4536E2D9900C4]_{CK} \\ &= KASUMI[2FD678A6DA5A94D8]_{CK} \\ &= 3D84EA7D3CB3C739 \end{aligned}$$

$n = 12$

$$\begin{aligned} KSB_1 2 &= KASUMI[A \oplus BLKCNT \oplus KSB_{11}]_{CK} \\ &= KASUMI[34222BC8F7C39416 \oplus 11 \oplus 3D84EA7D3CB3C739]_{CK} \\ &= KASUMI[09A6C1B5CB705324]_{CK} \\ &= 9F190528BF5C8DA3 \end{aligned}$$

$n = 13$

$$\begin{aligned} KSB_1 3 &= KASUMI[A \oplus BLKCNT \oplus KSB_{12}]_{CK} \\ &= KASUMI[34222BC8F7C39416 \oplus 12 \oplus 9F190528BF5C8DA3]_{CK} \\ &= KASUMI[AB3B2EE0489F19B9]_{CK} \\ &= 082A2D8F25915EE3 \end{aligned}$$

The individual bits of the keystream are extracted from KSB_1 to KSB_{13} by most significant bit first. For $n = 1$ to $BLOCKS$ and for each integer i with $0 \leq i \leq 63$ we have:

$$KS[((n-1) * 64) + i] = KSB_n[i]$$

Encryption

The encryption of the plaintext is performed by XORing the input data IBS with the generated keystream KS. For each integer i with $0 \leq i \leq LENGTH - 1$ define:

$$OBS[i] = IBS[i] \oplus KS[i]$$

Therefore the ciphertext is found by using the plaintext along with the keystream generated above (Plaintext $\oplus$ Keystream = Ciphertext):

$7EC61272743BF161$ $4726446A6C38CED1$ $66F6CA76EB543004$
$4286346CEF130F92$ $922B03450D3A9975$ $E5BD2EA0EB55AD8E$
$1B199E3EC4316020$ $E9A1B285E7627953$ $59B7BDFD39BEF4B2$
$484583D5AFE082AE$ $E638BF5FD5A60619$ $3901A08F4AB41AAB$
$9B134880$
$\oplus$
$AF24CC029AC39D08$ $23DD1041AEECAE7B$ $D95CDAD24BC7162F$
$3F9FAA1C80D1DB1B$ $87782A2C1DC93006$ $E49BAC44F71B868C$
$A5398989E10ADFB3$ $E07FEA9C2C20914A$ $0F437466F0C8A81D$
$1BF4536E2D9900C4$ $3D84EA7D3CB3C739$ $9F190528BF5C8DA3$
$082A2D8F$
$=$
$D1E2DE70EEF86C69$ $64FB542BC2D460AA$ $BFAA10A4A093262B$
$7D199E706FC2D489$ $1553296910F3A973$ $012682E41C4E2B02$
$BE2017B7253BBF93$ $09DE5819CB42E819$ $56F4C99BC9765CAF$
$53B1D0BB8279826A$ $DBBC5522E915C120$ $A618A5A7F5E89708$
$9339650F$

4.2.5 $f9$ Example

For an example of the $f9$ algorithm the test vector given by the 3GPP will be used:

IK	$= 2BD6\ 459F\ 82C5\ B300\ 952C\ 4910\ 4881\ FF48$ (128-bits)
$COUNT$	$= 38A6F056$ (32-bits)
$FRESH$	$= 05D2EC49$ (32-bits)
$DIRECTION$	$= 0$ (1-bit)
$LENGTH$	= 189-bits

Message:

$$6B227737296F393C\ 8079353EDC87E2E8\ 05D2EC49A4F2D8E0$$

The two 64-bit registers are set to zero:

$$A = 0 \quad , \quad B = 0$$

The padded string is then calculated:

$PS =$ $COUNT||FRESH||MESSAGE||DIRECTION||1||0^*$
$00111000101001101111000001010110||00000101110100101110110001001001||$
$01101011001000100111011100110111001010010110111100111000000000000$
$10000000011110010011010100111110110111001000011111100000000000000$
$00000101110100101110110001001001101001001111001011011001000000||0||1||0$

The padded string PS is then split into 64-bit blocks PS_i where:

$$\begin{aligned} PS_0 &= 38A6F05605D2EC49 \\ PS_1 &= 6B227737296F393C \\ PS_2 &= 8079353EDC87E2E8 \\ PS_3 &= 05D2EC49A4F2D8E2 \end{aligned}$$

Then for each integer n with $0 \leq n \leq 3$:

$n = 0$

$$\begin{aligned} A &= KASUMI[A \oplus PS_0]_{IK} \\ &= KASUMI[0 \oplus 38A6F05605D2EC49]_{IK} \\ &= KASUMI[38A6F05605D2EC49]_{IK} \\ &= 89E0A6D036C17090 \\ B &= B \oplus A \\ &= 0 \oplus 89E0A6D036C17090 \\ &= 89E0A6D036C17090 \end{aligned}$$

$n = 1$

$$\begin{aligned} A &= KASUMI[A \oplus PS_1]_{IK} \\ &= KASUMI[89E0A6D036C17090 \oplus 6B227737296F393C]_{IK} \\ &= KASUMI[E2C2D1E71FAE49AC]_{IK} \\ &= 45C16C0142460205 \\ B &= B \oplus A \\ &= 89E0A6D036C17090 \oplus 45C16C0142460205 \\ &= CC21CAD174877295 \end{aligned}$$

$n = 2$

$$\begin{aligned} A &= KASUMI[A \oplus PS_2]_{IK} \\ &= KASUMI[45C16C0142460205 \oplus 8079353EDC87E2E8]_{IK} \\ &= KASUMI[C5B8593F9EC1E0ED]_{IK} \\ &= E24CFA7D8471E4DD \\ B &= B \oplus A \\ &= CC21CAD174877295 \oplus E24CFA7D8471E4DD \\ &= 2E6D30ACF0F69648 \end{aligned}$$

$n = 3$

$$
\begin{aligned}
A = &\quad KASUMI[A \oplus PS_3]_{IK} \\
= &\quad KASUMI[E24CFA7D8471E4DD \oplus 05D2EC49A4F2D8E2]_{IK} \\
= &\quad KASUMI[E79E163420833C3F]_{IK} \\
= &\quad DFD3DCB9499275BA \\
B = &\quad B \oplus A \\
= &\quad 2E6D30ACF0F69648 \oplus DFD3DCB9499275BA \\
= &\quad F1BEEC15B964E3F2
\end{aligned}
$$

One more application of KASUMI is then applied using the modified form of the integrity key IK:

$$
\begin{aligned}
IK \oplus KM = &\quad 2BD6\ 459F\ 82C5\ B300\ 952C\ 4910\ 4881\ FF48 \\
&\quad \oplus \\
&\quad AAAA\ AAAA\ AAAA\ AAAA\ AAAA\ AAAA\ AAAA\ AAAA \\
= &\quad 817C\ EF35\ 286F\ 19AA\ 3F86\ E3BA\ E22B\ 55E2
\end{aligned}
$$

Therefore the final step is:

$$
\begin{aligned}
B = &\quad KASUMI[B]_{IK \oplus KM} \\
= &\quad KASUMI[F1BEEC15B964E3F2]_{IK \oplus KM} \\
= &\quad F63BD72C702EBC7A
\end{aligned}
$$

The $MAC - I$ is then the leftmost 32-bits:

$$MAC - I = \quad F63BD72C$$

4.3 INTERACTION BETWEEN 2G AND 3G

The downgrade attack discusses in the 2G GSM section demonstrated the security threat brought about by having a weaker encryption algorithm included in the ME. If the ME was equipped with both the A5/1 and A5/2 algorithms a downgrade attack was used to force the phone into using the weaker encryption mode A5/2 to discover the Cipher Key. The same threat would apply to UE with the A5/2 and $f8$ algorithms. A 3G phone using the $f8$ algorithm could potentially be downgraded to using the A5/2 algorithm where an attack could be mounted to recover the Cipher Key. To prevent this downgrade attack along with enhancing the security of the GSM/GPRS/EDGE networks each of their respective security algorithms were upgraded and based on the $f8$ algorithm. Note that 3G UMTS UE needed to contain 2G encryption algorithms to allow for backwards compatibility while roaming. The new algorithms for the GSM and GPRS/EDGE networks are named A5/3 and GEA3. Since the 2G networks use a Cipher Key of only 64-bits and 3G networks use 128-bits, a

modification of the inputs had to be made. The A5/3 and GEA3 algorithms are based off of $f8$ and have the same core structure. Therefore the core part of the $f8$ algorithm that they all share is referred to as the Core Keystream Generator (KGCORE). The descriptions and diagrams for the A5/3, GEA3, and KGCORE algorithms are taken from the 3GPP document: A5/3 and GEA3 Specifications [11]. A diagram for KGCORE is given in Figure 4.7. Note that the only difference between KGCORE and the UMTS $f8$ algorithm are

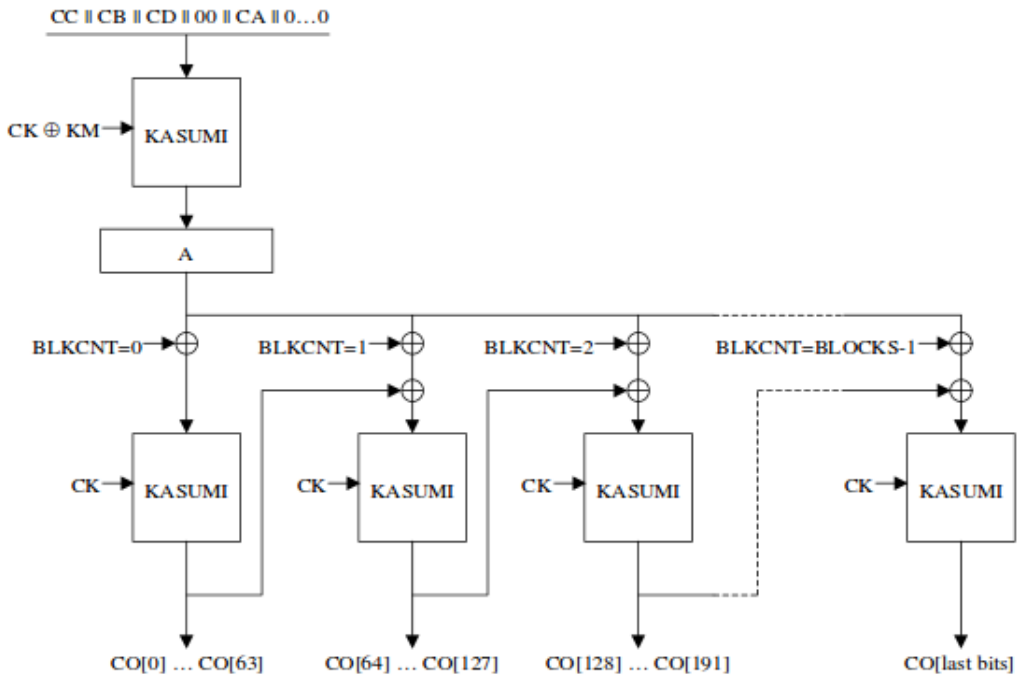

Figure 4.7. KGCORE algorithm. G.T. 55.216, *Specifications of the a5/3 encryption algorithms for gsm and edge, and the gea3 encryption algorithm for gprs: Document 1, technical report*, 3GPP, 2012.

the inputs and the renaming of the outputs. The inputs for the KGCORE vary depending on the algorithm but the total size of the input remains constant and consists of the following:

Parameter	Comment
CA	8-bits $CA[0]...CA[7]$
CB	5-bits $CB[0]...CB[4]$
CC	32-bits $CC[0]...CC[31]$
CD	1-bit $CD[0]$
CE	16-bits $CE[o]...CE[15]$
CK	128-bits $CK[0]...CK[127]$
CL	An integer in the range $1...2^{29}$ inclusive, specifying the number of output bits to produce

The output for KGCORE consists of the following:

Parameter	Comment
CO	CL-bits $CO[0]...CO[CL-1]$

The initialization of the KGCORE function begins by setting the 64-bit register A to:

$$\begin{aligned} A = &\ CC||CB||CD||00||CA||CE \\ = &\ CC[0]...CC[31]CB[0]...CB[4]CD[0]00CA[0]...CA[7]CE[0]...CE[15] \end{aligned}$$

The algorithm is then initialized in the same manner described above for the UMTS $f8$ algorithm.

A5/3 Algorithm for GSM Encryption

The A5/1 algorithm produced two 114-bit keystreams. One 114-bit block for MS to BTS and the other 114-bit block for BTS to MS encryption. The inputs for the A5/1 algorithm consisted of a 22-bit frame number along with the 64-bit Cipher Key. For A5/3 the 22-bit frame number is now renamed to $COUNT$, the Cipher Key remains K_c, and the 114+114-bit outputs are now $BLOCK1$ and $BLOCK2$ each consisting of 114-bits. The mapping of the GSM A5/3 inputs/outputs into the KGCORE function are given below:

$CA[0]...CA[7]$	=	00001111
$CB[0]...CB[4]$	=	00000
$CC[0]...CC[9]$	=	0000000000
$CC[10]...CC[31]$	=	COUNT[0]...COUNT[21]
$CD[0]$	=	0
$CE[0]...CE[15]$	=	000000000000000
$CK[0]...CK[127]$	=	$K_c[0]..K_c[63]\|\|K_c[0]..K_c[63]$
Cl	=	228
$CO[0]...CO[113]$	=	$BLOCK1[0]...BLOCK1[113]$
$CO[114]...CO[227]$	=	$BLOCK2[0]...BLOCK2[113]$

GEA3 Algorithm for GPRS/EDGE Encryption

The GEA and GEA2 algorithms for GPRS have never been made public. All that is known are the input and output parameters provided by the 3GPP . The GPRS GEA algorithm consists of a 32-bit frame number, a directional bit indicating the direction of transmission, and the Cipher Key K_c. The output for the GEA algorithms is an M-byte keystring which does not exceed 2^{16}. For the GEA3 algorithm the 32-bit frame number is renamed to $INPUT$ and the output is referred to as $OUTPUT$ which consists of $8*$M-bits. Thus each $OUTPUT[i]$ for $0 \leq i \leq M-1$ represents a byte. The mapping of the GPRS GEA inputs/outputs into the KGCORE function are given below:

$$
\begin{array}{lcl}
CA[0]...CA[7] & = & 11111111 \\
CB[0]...CB[4] & = & 00000 \\
CC[0]...CC[31] & = & INPUT[0]...INPUT[21] \\
CD[0] & = & DIRECTION[0] \\
CE[0]...CE[15] & = & 0000000000000000 \\
CK[0]...CK[127] & = & K_c[0]..K_c[63]||K_c[0]..K_c[63] \\
Cl & = & 8 * M \\
CO[8i]...CO[8i+7] & = & OUTPUT[i] \\
 & & \text{for } 0 \leq i \leq M-1
\end{array}
$$

A comparison of A5/3, GEA3, and $f8$ in terms of the inputs and outputs of KGCORE are given in Table 4.7.

Table 4.7. GSM A5/3, GPRS GEA3, and UMTS $f8$ Algorithm in Terms of KGCORE Source: G. T. 55.216, *Specifications of the a5/3 encryption algorithms for gsm and edge, and the gea3 encryption algorithm for gprs: Document 1*, technical report, 3GPP, 2012.

	GSM A5/3	GPRS GEA3	UMTS $f8$
CA	00001111	11111111	00000000
CB	00000	00000	$BEARER$
CC	$0...0\|\|COUNT$	$INPUT$	$COUNT$
CD	0	$DIRECTION$	$DIRECTION$
CE	0000000000000000	0000000000000000	0000000000000000
CK	$K_c\|\|K_c$	$K_c\|\|K_c$	CK
CO	$BLOCK1\|\|BLOCK2$	$OUTPUT$	KS

CHAPTER 5

CONCLUSIONS

The future of cell phone technology is heading towards UMTS based 4G LTE. The rest of the world is already using UMTS based technology upgrading to LTE while the remaining US CDMA based networks are making the switch. Verizon plans to have their LTE network complete by the end of 2014. AT&T and T-Mobile already operate on UMTS based technology and are rolling out 4G LTE networks. Sprint has abandoned their WiMAX network and is in the process of building their LTE network from the ground up. The remaining top networks America Movil, U.S. Cellular, and Leap Wireless all offer 4G LTE service. When the U.S. finishes migrating from CDMA based technology to LTE there will finally be a global standard for mobile technology. The next hurdle will be for the networks to reach true 4G speeds specified by the standards set by the ITU: 100 Mbit/s download and 50 Mbit/s upload. After true 4G speeds are achieved for LTE the next advancement will be named LTE Advanced (LTE-A). The standards set for LTE Advanced include download speeds of 1 Gbit/s and upload speeds of 500 Mbit/s. With LTE Advanced it is expected that features on mobile phones will include high-definition TV, 3D TV, and cloud computing.

Future problems facing the LTE security algorithms were researched by the 3GPP and a summary of the possible vulnerabilities are summarized below [8] [12] [2]. The MILENAGE set of security algorithms for UMTS/LTE were designed to be in use for at least 20 years. To help plan for any future weaknesses or for features that might want to be added the KGCORE 16-bit input value CE is set to 0 in order to reserve it for future use. The core algorithm for the $f1 - f5$ algorithm set is Rijndael AES consisting of a 128-bit Key and 10 rounds of operation. A brute force attack on the $f1 - f5$ algorithm set would require 2^{128} attempts. There are currently no known attacks of complexity substantially less than 2^{128} allowing the recovery of any information on the value of K_i even if the outputs for $f1 - f5$ are known for a large set of arbitrary chosen $RAND$ with corresponding SQN, AMF, and OP values. Given any combination of outputs significantly less than 2^{64} for any chosen inputs it is computationally infeasible to predict any additional outputs for any of the fi functions.

One of the weaknesses of the 2G GSM network was when the attacker had physical access to the SIM card they were able to query the SIM with arbitrarily chosen $RAND$ values to discover the secret key. The USIM card used in the UMTS/LTE networks protects against this type of attack even if the attacker has full control and can choose any $RAND, SQN$, and AMF values. This is due to the fact that the network authorization is checked within the

USIM and is not directly available to the attacker. The input and output bandwidth of the USIM was also limited to reduce the rate of input/output queries to less than 10 pairs per second. After allowing the open scientific community and three independently hired evaluators (Leuven University in Belgium, Cryptolog in France, and Royal Holloway College in UK) to scrutinize the algorithm set, the strongest attack required 2^{64} queries. This was ruled to be infeasible as the USIM allows less than 10 queries per second. The evaluation time for the algorithms lasted 230 days.

The same scrutiny of the $f1 - f5$ algorithm set was applied to the $f8$ and $f9$ algorithms. The attacks were tested against the KASUMI algorithm used in the $f8$ and $f9$ algorithms. The results of the studies concluded that the best attacks were only effective up to 5 rounds and at most 6 rounds of KASUMI. When KASUMI was implemented with the 8 rounds used in $f8$ and $f9$ algorithms no practical attacks were discovered. One attack was able to identify a small amount of structure in the keystream but needed a long sequence of keystream of the order 2^{38} bits. This attack is impractical as each frame produces no more than 5000 bits before a new keystream is generated. The key size of 128-bits was estimated to provide sufficient protection against exhaustive searches for the next 20 years or more. The independent researchers concluded that KASUMI with eight rounds offered a sufficient security margin against current cryptographic techniques.

It was also recommended by the researchers that with the rapid evolution in cryptography all of the algorithms should be evaluated at least every five years. Taking this advice the 3GPP holds a basic review of the security and usability of the confidentiality and integrity algorithms every five years.

Although no practical attacks against the algorithms have been found the user should not believe that eavesdropping is impossible. The Communications Assistance for Law Enforcement Act (CALEA) passed under Bill Clinton in 1994 required the telecommunication carriers and manufacturers of telecommunications equipment to modify and design their equipment, facilities, and services to ensure that the federal government would be able to monitor telephone communications in real-time. This eavesdropping backdoor has brought about an App market for journalists, diplomats, and corporate executives where data is encrypted inside the app before being sent and encrypted by the network. These type of apps provides an extra layer of security and protection against backdoor eavesdropping. If one wants to make a truly private phone conversation they need to provide their own mobile phone security.

BIBLIOGRAPHY

[1] G. T. 33.908, *General report on the design, specification and evaluation of 3GPP standard confidentiality and integrity algorithms*, technical report, 3GPP, 2001.

[2] G. T. 33.909, *3G security; report on the evaluation of 3GPP standard confidentiality and integrity algorithms*, technical report, 3GPP, 2001.

[3] G. T. 35.201, *Specification of the 3GPP confidentiality and integrity algorithms; document 1: f8 and f9 specifications*, technical report, 3GPP, 2012.

[4] G. T. 35.202, *Specification of the 3GPP confidentiality and integrity algorithms; document 2: Kasumi*, technical report, 3GPP, 2012.

[5] G. T. 35.203, *Specification of the 3GPP confidentiality and integrity algorithms; document 3: Implementors' test data*, technical report, 3GPP, 2012.

[6] G. T. 35.204, *Specification of the 3GPP confidentiality and integrity algorithms; document 4: Design conformance test data*, technical report, 3GPP, 2012.

[7] G. T. 35.206, *3rd generation partnership project; technical specification group services and system aspects; 3G security; specification of the milenage algorithm set: An example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; document 2: Algorithm specification*, technical report, 3GPP, 2012.

[8] G. T. 35.909, *Specification of the milenage algorithm set: Document 5: Summary and results of design and evaluation*, technical report, 3GPP, 2012.

[9] *3GPP a global initiative*. 3GPP, http://www.3gpp.org , accessed July 2013, n.d.

[10] *4G americas: Understanding 1G vs. 2G vs. 3G vs. 4G*. 4G Americas, http://www.4gamericas.org , accessed July 2013, n.d.

[11] G. T. 55.216, *Specifications of the a5/3 encryption algorithms for GSM and EDGE, and the GEA3 encryption algorithm for GPRS: Document 1*, technical report, 3GPP, 2012.

[12] G. T. 55.919, *General report on the design, specification and evaluation of 3GPP standard confidentiality and integrity algorithms*, technical report, 3GPP, 2012.

[13] O. ABIKOYE AND D. OLUWADE, *Computational analysis of GSM security algorithm*, Afr. J. Comp. & ICT, 1 (2008), pp. 27–33.

[14] E. BARKAN, E. BIHAM, AND N. KELLER, *Instant ciphertext-only cryptanalysis of GSM encrypted communication*, in Advances in Cryptology - CRYPTO 2003, D. Boneh, ed., vol. 2729 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 600–616.

[15] E. BARKAN, E. BIHAM, AND N. KELLER, *Instant ciphertext-only cryptanalysis of GSM encrypted communication*, in Advances in Cryptology - CRYPTO 2003, D. Boneh, ed., vol. 2729 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 600–616.

[16] A. BIRYUKOV, A. SHAMIR, AND D. WAGNER, *Real time cryptanalysis of a5/1 on a pc*, Fast Software Encryption, (2001), pp. 37–44.

[17] M. BRICENO, I. GOLDBERG, AND D. WAGNER, *An implementation of the GSM a3 a8 algorithm*. Unpublished report, 1998.

[18] J. J. BUCHHOLZ, *Matlab implementation of the advanced encryption standard*. Unpublished report, December 2001.

[19] T. GENDRULLIS, M. NOVOTN, AND A. RUPP, *A real-world attack breaking a5/1 within hours*, in Cryptographic Hardware and Embedded Systems CHES 2008, E. Oswald and P. Rohatgi, eds., vol. 5154 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 266–282.

[20] G. A. GROW AND R. K. SMITH, *Mobile and Wireless Communications: An Introduction*, McGraw-Hill International, 2006.

[21] H. HANDSCHUH AND P. PAILLIER, *Reducing the collision probability of alleged COMP128*, Smart Card research and Applications, 1820 (2000), pp. 380–385.

[22] R. J., R. P., S. H., AND T. S., *Partitioning attacks: Or how to rapidly clone some GSM cards*, IEEE Symposium on Security and Privacy, (2002).

[23] A. MEHROTRA, *GSM System Engineering*, Artech House, INC., 1996.

APPENDIX A

MATLAB CODES

MATLAB CODES

COMP128-1

```
clearvars -global                          % Clearing global variables
clearvars                                  % Clearing variables
dbstop if error                            % Used for Debugging Code
clc                                        % Clears the Command Window of Matlab

% COMP128-1 Algorithm

%Define Five Compression Tables
Table0=[ ...
102, 177, 186, 162, 2, 156, 112, 75, 55, 25, 8, 12, 251, 193, 246, 188, ...
109, 213, 151, 53, 42, 79, 191, 115, 233, 242, 164, 223, 209, 148, 108, 161, ...
252, 37, 244, 47, 64, 211, 6, 237, 185, 160, 139, 113, 76, 138, 59, 70, ...
67, 26, 13, 157, 63, 179, 221, 30, 214, 36, 166, 69, 152, 124, 207, 116, ...
247, 194, 41, 84, 71, 1, 49, 14, 95, 35, 169, 21, 96, 78, 215, 225, ...
182, 243, 28, 92, 201, 118, 4, 74, 248, 128, 17, 11, 146, 132, 245, 48, ...
149, 90, 120, 39, 87, 230, 106, 232, 175, 19, 126, 190, 202, 141, 137, 176, ...
250, 27, 101, 40, 219, 227, 58, 20, 51, 178, 98, 216, 140, 22, 32, 121, ...
61, 103, 203, 72, 29, 110, 85, 212, 180, 204, 150, 183, 15, 66, 172, 196, ...
56, 197, 158, 0, 100, 45, 153, 7, 144, 222, 163, 167, 60, 135, 210, 231, ...
174, 165, 38, 249, 224, 34, 220, 229, 217, 208, 241, 68, 206, 189, 125, 255, ...
239, 54, 168, 89, 123, 122, 73, 145, 117, 234, 143, 99, 129, 200, 192, 82, ...
104, 170, 136, 235, 93, 81, 205, 173, 236, 94, 105, 52, 46, 228, 198, 5, ...
57, 254, 97, 155, 142, 133, 199, 171, 187, 50, 65, 181, 127, 107, 147, 226, ...
184, 218, 131, 33, 77, 86, 31, 44, 88, 62, 238, 18, 24, 43, 154, 23, ...
80, 159, 134, 111, 9, 114, 3, 91, 16, 130, 83, 10, 195, 240, 253, 119, ...
177, 102, 162, 186, 156, 2, 75, 112, 25, 55, 12, 8, 193, 251, 188, 246, ...
213, 109, 53, 151, 79, 42, 115, 191, 242, 233, 223, 164, 148, 209, 161, 108, ...
37, 252, 47, 244, 211, 64, 237, 6, 160, 185, 113, 139, 138, 76, 70, 59, ...
26, 67, 157, 13, 179, 63, 30, 221, 36, 214, 69, 166, 124, 152, 116, 207, ...
194, 247, 84, 41, 1, 71, 14, 49, 35, 95, 21, 169, 78, 96, 225, 215, ...
243, 182, 92, 28, 118, 201, 74, 4, 128, 248, 11, 17, 132, 146, 48, 245, ...
```

```
90, 149, 39, 120, 230, 87, 232, 106, 19, 175, 190, 126, 141, 202, 176, 137, ...
27, 250, 40, 101, 227, 219, 20, 58, 178, 51, 216, 98, 22, 140, 121, 32, ...
103, 61, 72, 203, 110, 29, 212, 85, 204, 180, 183, 150, 66, 15, 196, 172, ...
197, 56, 0, 158, 45, 100, 7, 153, 222, 144, 167, 163, 135, 60, 231, 210, ...
165, 174, 249, 38, 34, 224, 229, 220, 208, 217, 68, 241, 189, 206, 255, 125, ...
54, 239, 89, 168, 122, 123, 145, 73, 234, 117, 99, 143, 200, 129, 82, 192, ...
170, 104, 235, 136, 81, 93, 173, 205, 94, 236, 52, 105, 228, 46, 5, 198, ...
254, 57, 155, 97, 133, 142, 171, 199, 50, 187, 181, 65, 107, 127, 226, 147, ...
218, 184, 33, 131, 86, 77, 44, 31, 62, 88, 18, 238, 43, 24, 23, 154, ...
159, 80, 111, 134, 114, 9, 91, 3, 130, 16, 10, 83, 240, 195, 119, 253, ...
];
Table1=[ ...
19, 11, 80, 114, 43, 1, 69, 94, 39, 18, 127, 117, 97, 3, 85, 43, ...
27, 124, 70, 83, 47, 71, 63, 10, 47, 89, 79, 4, 14, 59, 11, 5, ...
35, 107, 103, 68, 21, 86, 36, 91, 85, 126, 32, 50, 109, 94, 120, 6, ...
53, 79, 28, 45, 99, 95, 41, 34, 88, 68, 93, 55, 110, 125, 105, 20, ...
90, 80, 76, 96, 23, 60, 89, 64, 121, 56, 14, 74, 101, 8, 19, 78, ...
76, 66, 104, 46, 111, 50, 32, 3, 39, 0, 58, 25, 92, 22, 18, 51, ...
57, 65, 119, 116, 22, 109, 7, 86, 59, 93, 62, 110, 78, 99, 77, 67, ...
12, 113, 87, 98, 102, 5, 88, 33, 38, 56, 23, 8, 75, 45, 13, 75, ...
95, 63, 28, 49, 123, 120, 20, 112, 44, 30, 15, 98, 106, 2, 103, 29, ...
82, 107, 42, 124, 24, 30, 41, 16, 108, 100, 117, 40, 73, 40, 7, 114, ...
82, 115, 36, 112, 12, 102, 100, 84, 92, 48, 72, 97, 9, 54, 55, 74, ...
113, 123, 17, 26, 53, 58, 4, 9, 69, 122, 21, 118, 42, 60, 27, 73, ...
118, 125, 34, 15, 65, 115, 84, 64, 62, 81, 70, 1, 24, 111, 121, 83, ...
104, 81, 49, 127, 48, 105, 31, 10, 6, 91, 87, 37, 16, 54, 116, 126, ...
31, 38, 13, 0, 72, 106, 77, 61, 26, 67, 46, 29, 96, 37, 61, 52, ...
101, 17, 44, 108, 71, 52, 66, 57, 33, 51, 25, 90, 2, 119, 122, 35, ...
];
Table2=[ ...
52, 50, 44, 6, 21, 49, 41, 59, 39, 51, 25, 32, 51, 47, 52, 43, ...
37, 4, 40, 34, 61, 12, 28, 4, 58, 23, 8, 15, 12, 22, 9, 18, ...
55, 10, 33, 35, 50, 1, 43, 3, 57, 13, 62, 14, 7, 42, 44, 59, ...
62, 57, 27, 6, 8, 31, 26, 54, 41, 22, 45, 20, 39, 3, 16, 56, ...
48, 2, 21, 28, 36, 42, 60, 33, 34, 18, 0, 11, 24, 10, 17, 61, ...
29, 14, 45, 26, 55, 46, 11, 17, 54, 46, 9, 24, 30, 60, 32, 0, ...
```

```
20, 38, 2, 30, 58, 35, 1, 16, 56, 40, 23, 48, 13, 19, 19, 27, ...
31, 53, 47, 38, 63, 15, 49, 5, 37, 53, 25, 36, 63, 29, 5, 7, ...
];
Table3=[ ...
1, 5, 29, 6, 25, 1, 18, 23, 17, 19, 0, 9, 24, 25, 6, 31, ...
28, 20, 24, 30, 4, 27, 3, 13, 15, 16, 14, 18, 4, 3, 8, 9, ...
20, 0, 12, 26, 21, 8, 28, 2, 29, 2, 15, 7, 11, 22, 14, 10, ...
17, 21, 12, 30, 26, 27, 16, 31, 11, 7, 13, 23, 10, 5, 22, 19, ...
];
Table4=[ ...
15, 12, 10, 4, 1, 14, 11, 7, 5, 0, 14, 7, 1, 2, 13, 8, ...
10, 3, 4, 9, 6, 0, 3, 2, 5, 6, 8, 9, 11, 13, 15, 12, ...
];

% Manualy input here the Authentication Key contained on SIM card and RAND
% challenge from base station in Decimal Format
RAND=[59, 152, 71, 83, 249, 170, 216, 166, 184, 223, 14, 33, 219, 154, 46, 141];
KeyAuth=[82, 161, 176, 215, 243, 129, 110, 198, 231, 35, 40, 151, 241, 112, 168, 26];

% Displays RAND and Authentication Key in Hexadecimal Format
fprintf('RAND Challenge from Base Station:\n')
disp(dec2hex(RAND))
fprintf('\nAuthentication Key located on SIM card\n')
disp(dec2hex(KeyAuth))

%Creating array where X[1...16]=KeyAuth and X[17...32]=RAND
X=[KeyAuth,RAND];

for i=1:8                                                    % Creates eight rounds of iterations
        for j=1:16                                % Resets X[1...16] to KeyAuth after each iteration
                X(:,j)=KeyAuth(:,j);
        end

        % Begin Five subrounds of compressoin
        j=0;                                                 % First subround of compression
        fprintf('\nRound %d: Compression Subround %d of 5\n',i,j+1)
```

```
for k=0:2^j-1
        for l=0:2^(4-j)-1;
                m=l+k*2^(5-j)+1;
                n=m+2^(4-j);
                y=mod(X(:,m)+2*X(:,n),2^(9-j));
                z=mod(2*X(:,m)+X(:,n),2^(9-j));
                X(:,m)=Table0(:,y+1);
                X(:,n)=Table0(:,z+1);
        end
end                                                     % Ends Subround 1 of 5
disp(dec2hex(X))

j=1;                                                    % Second subround of compression
fprintf('\nRound %d: Compression Subround %d of 5\n',i,j+1)
for k=0:2^j-1
        for l=0:2^(4-j)-1;
                m=l+k*2^(5-j)+1;
                n=m+2^(4-j);
                y=mod(X(:,m)+2*X(:,n),2^(9-j));
                z=mod(2*X(:,m)+X(:,n),2^(9-j));
                X(:,m)=Table1(:,y+1);
                X(:,n)=Table1(:,z+1);
        end
end                                                     % Ends Subround 2 of 5
disp(dec2hex(X))

j=2;                                                    % Third subround of compression
fprintf('\nRound %d: Compression Subround %d of 5\n',i,j+1)
for k=0:2^j-1
        for l=0:2^(4-j)-1;
                m=l+k*2^(5-j)+1;
                n=m+2^(4-j);
                y=mod(X(:,m)+2*X(:,n),2^(9-j));
                z=mod(2*X(:,m)+X(:,n),2^(9-j));
                X(:,m)=Table2(:,y+1);
                X(:,n)=Table2(:,z+1);
```

```
        end
    end                                                    % Ends Subround 3 of 5
    disp(dec2hex(X))

    j=3;                                                   % Fourth subround of compression
    fprintf('\nRound %d: Compression Subround %d of 5\n',i,j+1)
    for k=0:2^j-1
        for l=0:2^(4-j)-1;
            m=l+k*2^(5-j)+1;
            n=m+2^(4-j);
            y=mod(X(:,m)+2*X(:,n),2^(9-j));
            z=mod(2*X(:,m)+X(:,n),2^(9-j));
            X(:,m)=Table3(:,y+1);
            X(:,n)=Table3(:,z+1);
        end
    end                                                    % Ends Subround 4 of 5
    disp(dec2hex(X))

    j=4;                                                   % Fifth subround of compression
    fprintf('\nRound %d: Compression Subround %d of 5\n',i,j+1)
    for k=0:2^j-1
        for l=0:2^(4-j)-1;
            m=l+k*2^(5-j)+1;
            n=m+2^(4-j);
            y=mod(X(:,m)+2*X(:,n),2^(9-j));
            z=mod(2*X(:,m)+X(:,n),2^(9-j));
            X(:,m)=Table4(:,y+1);
            X(:,n)=Table4(:,z+1);
        end
    end                                                    % Ends Subround 5 of 5
    disp(dec2hex(X))
    fprintf('\nEnd of Compression for Round %d\n\n',i) % End Subrounds of
Compression

    %Converting 32-byte output from compression to 128-bit vector
    fprintf('Begin to form bits from bytes\n');
```

```
Y=de2bi(X, 'left-msb');          % Creates a 32x4 matrix converting decimal to binary
Bits=zeros(1,128);                          % Creating vector to store bits from bytes
for j=0:31
        for k=1:4
                Bits(:,j*4+k)=Y(j+1,k);          % Converts 32x4 Matrix to row vector
        end
end                                                          % Ends Bits from Bytes
fprintf('Round %d Bits from Bytes:\n\n',i)
disp(Bits)

% Begin Permutations for the first 7 rounds
BitPermuter=zeros(1,128);                   % Determins permutation order of Bits
if i<8
        for j=0:15
                for k=0:7
                        BitPermuter(:,8*j+k+1)=mod((8*j+k)*17,128)+1;
                end
        end

BitPermuted=zeros(1,128);                   % Uses BitPermuter to permute Bits
for j=1:128
        BitPermuted(:,j)=Bits(:,BitPermuter(:,j));   %Store permutation in BitPermuted
end

BitPermutedDecimal=zeros(1,16);          %Converts 128 bits to 16 bytes in Decimal
for j=1:16
        b=0;
        for k=0:7
                b=BitPermuted(:,8*j-7+k)*2^(7-k)+b;
        end
        BitPermutedDecimal(:,j)=b;
end

% Store permuted results in X[17...32]
for j=1:16
        X(:,j+16)=BitPermutedDecimal(:,j);
```

```
        end                                                 % End of Permutations for first 7 rounds
        fprintf('Round %d of 7 Permutation\n', i)
        disp(dec2hex(X))

        else                                            % For round 8, no permutation is performed
        fprintf('No permutation is performed in round 8\n\n')
        end                                                                 % End of permutations
end                                                                         % End of 8 rounds

COMP128Output=zeros(1,12);      % Creating vector to display 12 bytes of Output in decimal
for j=1:4                                                                   % Output bytes 1-4
        COMP128Output(:,j)=bitxor(bitshift(X(:,2*j-1),4),X(:,2*j));
end

for j=0:5                                                                   % Output bytes 5-10
        COMP128Output(:,j+5)=mod(bitxor(bitxor(bitshift(X(:,2*j+19),6)...
                                                ,bitshift(X(:,2*j+19+1),2)),(...
                                                bitshift(X(:,2*j+19+2),-2))),...
                                256);
end

COMP128Output(:,5+6)=mod(bitxor(bitshift(X(:,2*6+19),6),...
                                bitshift(X(:,2*6+19+1),2)),256);
COMP128Output(:,5+7)=0;
% Note that COMP128Output[11,12] produce 10 bits of zero for the Session Key
fprintf('Output from COMP128:\n')
disp(dec2hex(COMP128Output))

SRES=zeros(1,4);                        % 1x4 vector of bytes for SRES=COMP128Output[1...4]
for j=1:4
        SRES(:,j)=COMP128Output(:,j);
end
fprintf('\nSRES\n')
disp(dec2hex(SRES));

KeySession=zeros(1,8);          % 1x8 vector of bytes for Session Key=COMP128Output[5...12]
```

```
for j=1:8
        KeySession(:,j)=COMP128Output(:,4+j);
end
fprintf('\nSession Key (Hexadecimal)\n')
disp(dec2hex(KeySession))
fprintf('\nSession Key (Binary)\n')
disp(dec2bin(KeySession))

% End of COMP128 Algorithm
```

A5/2

```
clearvars -global                                   % Clearing global variables
clearvars                                           % Clearing variables
dbstop if error                                     % Used for Debugging Code
clc                                                 % Clears the Command Window of Matlab

% A5/1 Algorithm

% Input Session Key from output of COMP128
Key=[ ...
1,0,0,1,0,0,1,0,0,1,1,0,1,1,0,1,0,0,1,1,1,0,0,0,1,1,0,1,1,1,1,1, ...
0,0,1,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0 ...
];

% Container for reordering Session Key by lsb for entering into A5/1 algorithm
KeyLSB=[ ...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ...
];

% Input frame Number=00 01 34 in 22-bit format
Frame=[0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,0,1,0,0];
% Defining Container for reordering frame number by lsb first for entering into A5/1
% algorithm
FrameLSB=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
```

```
%Displaying Input of algorithm
fprintf('A5/1 Algorithm\n\n')
fprintf('Session Key: 92 6D 38 DF 3F AF 88 00 \n')
fprintf('Session Key (Binary):\n\n')
disp(Key)

% Load Frame Number by LSB first into FrameLSB preparing to be inputted into
% registers
for i=1:22
        FrameLSB(i)=Frame(23-i);
end

% Load Session Key by lsb per byte preparing to be inputted into registers
for i=0:63
        KeyLSB(i+1)=Key(8*(2*floor(i./8)+1)-i);
end

% Stage 1: 64 cycles (No stop/go clock control)
% Begin by setting all three registers to zero
R1=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
R2=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
R3=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];

% Begin 64 cycles with 64-bit session key reordered by lsb of each byte fed into registers
for i=1:64
        R1Step=bitxor(bitxor(bitxor(R1(1),R1(2)),bitxor(R1(3),R1(6))),KeyLSB(i));
        for j=1:18
                R1(j)=R1(j+1);
        end
        R1(19)=R1Step;

        R2Step=bitxor(bitxor(R2(1),R2(2)),KeyLSB(i));
        for j=1:21
                R2(j)=R2(j+1);
        end
        R2(22)=R2Step;
```

```
        R3Step=bitxor(bitxor(bitxor(bitxor(R3(1),R3(2)),R3(3)),R3(16)),KeyLSB(i));
        for j=1:22
                R3(j)=R3(j+1);
        end
        R3(23)=R3Step;

        if i==0 % Enable to display a cycle of Stage 1 (manually edit value)
                fprintf('\nStage 1: cycle %d of 64\n',i)
                fprintf('R1\n')
                disp(R1)
                fprintf('R2\n')
                disp(R2)
                fprintf('R3\n')
                disp(R3)
                fprintf('\n End of cycle %d of 64\n',i)
        end
end % End of Stage 1
fprintf('\nStage 1 Output: 64 cycles ignoring stop go clock control\n\n')
fprintf('R1\n')
disp(R1)
fprintf('\nR2\n')
disp(R2)
fprintf('\nR3\n')
disp(R3)

% Stage 2: 22 cycles (No stop/go clock control)
% Begin 22 cycles with 22-bit Frame Number reordered by lsb fed into registers
for i=1:22
        R1Step=bitxor(bitxor(bitxor(R1(1),R1(2)),bitxor(R1(3),R1(6))),FrameLSB(i));
        for j=1:18
                R1(j)=R1(j+1);
        end
        R1(19)=R1Step;

        R2Step=bitxor(bitxor(R2(1),R2(2)),FrameLSB(i));
        for j=1:21
```

```
        R2(j)=R2(j+1);
    end
    R2(22)=R2Step;

    R3Step=bitxor(bitxor(bitxor(bitxor(R3(1),R3(2)),R3(3)),R3(16)),FrameLSB(i));
    for j=1:22
        R3(j)=R3(j+1);
    end
    R3(23)=R3Step;

    if i==0 % Enable to display a cycle of Stage 2 (manually edit value)
        fprintf('\nStage 2: cycle %d of 22\n',i)
        fprintf('R1\n')
        disp(R1)
        fprintf('R2\n')
        disp(R2)
        fprintf('R3\n')
        disp(R3)
        fprintf('\n End of cycle %d of 22\n',i)
    end
end % End of Stage 2
fprintf('\nStage 2 Output: 22 cycles ignoring stop go clock control\n\n')
fprintf('R1\n')
disp(R1)
fprintf('\nR2\n')
disp(R2)
fprintf('\nR3\n')
disp(R3)

% Stage 3: 100 cycles (stop/go clock control)
% Output is ignored, considered priming stage
for i=1:100
    Majority=[R1(11),R2(12),R3(13)]; % Vector to store clocking bit
    M=mode(Majority); % Majority Function for clock control

    if i==0 % Enable to display the clocking bit of each register for a given cycle
```

```
        fprintf('\nClocking Bits for Cycle %d of 100\n',i)
        fprintf('R1(8): ')
        disp(R1(11))
        fprintf('R1(10):')
        disp(R2(12))
        fprintf('R1(10):')
        disp(R3(13))
end

if R1(11)==M % Determines if Register 1 is clocked by Majority Function
        R1Step=bitxor(bitxor(R1(1),R1(2)),bitxor(R1(3),R1(6)));
        for j=1:18
                R1(j)=R1(j+1);
        end
        R1(19)=R1Step;
        %disp('R1 Clocked') % Uncomment out to display if R1 is Clocked
end

if R2(12)==M % Determines if Register 2 is clocked by Majority Function
        R2Step=bitxor(R2(1),R2(2));
        for j=1:21
                R2(j)=R2(j+1);
        end
        R2(22)=R2Step;
        %disp('R2 Clocked') % Uncomment out to display if R2 is Clocked
end

if R3(13)==M % Determines if Register 3 is clocked by Majority Function
        R3Step=bitxor(bitxor(bitxor(R3(1),R3(2)),R3(3)),R3(16));
        for j=1:22
                R3(j)=R3(j+1);
        end
        R3(23)=R3Step;
        %disp('R3 Clocked') % Uncomment out to display if R3 is Clocked
end
```

```
        if i==0 % Enable to display each register for a given cycle
                fprintf('\nStage 3: cycle %d of 100\n',i)
                fprintf('R1\n')
                disp(R1)
                fprintf('R2\n')
                disp(R2)
                fprintf('R3\n')
                disp(R3)
                fprintf('\n End of cycle %d of 100\n',i)
        end
end % End of Stage 3. Registers are now primed.
fprintf('\nStage 3 Output: 100 cycles with stop go clock control\n\n')
fprintf('R1\n')
disp(R1)
fprintf('\nR2\n')
disp(R2)
fprintf('\nR3\n')
disp(R3)

% Stage 4: 114+114 cycles (stop/go clock control)
% Output is stored as 228-bit keystream: 114-bits for MS to BTS and
% 114-bits for BTS to MS communication
Output=zeros(1,228); % Defining vector to store output from Stage 4
for i=1:228
        Majority=[R1(11),R2(12),R3(13)]; % Vector to store clocking bit
        M=mode(Majority); % Majority Function for clock control

        if i==0 % Enable to display the clocking bit of each register for a given cycle
                fprintf('\nClocking Bits for Cycle %d of 228\n',i)
                fprintf('R1(8): ')
                disp(R1(11))
                fprintf('R1(10):')
                disp(R2(12))
                fprintf('R1(10):')
                disp(R3(13))
        end
```

```
if R1(11)==M % Determines if Register 1 is clocked by Majority Function
        R1Step=bitxor(bitxor(R1(1),R1(2)),bitxor(R1(3),R1(6)));
        for j=1:18
                R1(j)=R1(j+1);
        end
        R1(19)=R1Step;
        % disp('R1 Clocked') % Uncomment out to display if R1 is Clocked
end

if R2(12)==M % Determines if Register 2 is clocked by Majority Function
        R2Step=bitxor(R2(1),R2(2));
        for j=1:21
                R2(j)=R2(j+1);
        end
        R2(22)=R2Step;
        %disp('R2 Clocked') % Uncomment out to display if R2 is Clocked
end

if R3(13)==M % Determines if Register 3 is clocked by Majority Function
        R3Step=bitxor(bitxor(bitxor(R3(1),R3(2)),R3(3)),R3(16));
        for j=1:22
                R3(j)=R3(j+1);
        end
        R3(23)=R3Step;
        %disp('R3 Clocked') % Uncomment out to display if R3 is Clocked
end

% Once the registers have been clocked, one byte of the keystream is
% produced by msb first
Output(1,i)=bitxor(bitxor(R1(1),R2(1)),R3(1));

if i==0 % Enable to display each register and keystream bit for a given cycle
        fprintf('\nStage 4: cycle %d of 228\n',i)
        fprintf('R1\n')
        disp(R1)
        fprintf('R2\n')
```

```
            disp(R2)
            fprintf('R3\n')
            disp(R3)
            fprintf('\nEnd of cycle %d of 228\n',i)
            fprintf('Keystream Bit Produced: %d\n',Output(1,i))
        end

end % End of Stage 4
fprintf('\nStage 4 Output: 114+114-bit Keystreams\n')

% Creating vector to store first 114-bits for MS to BTS communications
OutputMSBTS=zeros(1,120); % Need size of 120 since 114 isn't a multiple of 8 for i=1:114
        OutputMSBTS(1,i)=Output(i);
end

% Converting 114-bits of MS to BTS communications into 15 bytes (decimal)
OutputMSBTSDecimal=zeros(1,15);
for j=1:15
        b=0;
        for k=0:7
                b=OutputMSBTS(1,8*j-7+k)*2^(7-k)+b;
        end
OutputMSBTSDecimal(1,j)=b;
end

% Creating container to store second 114-bits for BTS to MS communications
OutputBTSMS=zeros(1,120); % Need size of 120 since 114 isn't a multiple of 8
for i=1:114
        OutputBTSMS(1,i)=Output(114+i);
end

% Converting 114-bits of BTS to MS communications into 15 bytes (Decimal)
OutputBTSMSDecimal=zeros(1,15);
for j=1:15
        b=0;
        for k=0:7
```

```
            b=OutputBTSMS(1,8*j-7+k)*2^(7-k)+b;
        end
        OutputBTSMSDecimal(1,j)=b;
end

% Display output of A5/1 Algorithm
fprintf('\nMS to BTS Keystream\n')
disp(dec2hex(OutputMSBTSDecimal))
fprintf('\nBTS to MS Keystream\n')
disp(dec2hex(OutputBTSMSDecimal))

% Display Ms to BTS Keystream in Bindary
fprintf('\nMs to BTS Keystream\n')
for i=1:114
        fprintf('%d', Output(1,i))
end

% Display BTS to MS Keystream in Bindary
fprintf('\nBTS to MS Keystream\n')
for i=1:114
        fprintf('%d', Output(1,114+i))
end

% End of A5/1 Algorithm
```

A5/2

```
clearvars -global                                   % Clearing global variables
clearvars                                           % Clearing variables
dbstop if error                                     % Used for Debugging Code
clc                                                 % Clears the Command Window of Matlab

% A5/2 Algorithm

% Input Session Key from output of COMP128
Key=[ ...
```

```
1,0,0,1,0,0,1,0,0,1,1,0,1,1,0,1,0,0,1,1,1,0,0,0,1,1,0,1,1,1,1,1, ...
0,0,1,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0 ...
];

% Container for reordering Session Key by lsb for entering into A5/2 algorithm
KeyLSB=[ ...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ...
];

% Input frame Number=00 01 34 in 22-bit format
Frame=[0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,0,1,0,0];

% Defining Container for reordering frame number by lsb first for entering into A5/2
% algorithm
FrameLSB=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];

%Displaying Input of algorithm
fprintf('A5/2 Algorithm\n\n')
fprintf('Session Key: 92 6D 38 DF 3F AF 88 00\n')
fprintf('Session Key (Binary):\n\n')
disp(Key)

% Load Frame Number by LSB first into FrameLSB preparing to be inputted into
% registers
for i=1:22
        FrameLSB(i)=Frame(23-i);
end

% Load Session Key by lsb per byte preparing to be inputted into registers
for i=0:63
        KeyLSB(i+1)=Key(8*(2*floor(i./8)+1)-i);
end

% Stage 1: 64 cycles (No stop/go clock control)
% Begin by setting all four registers to zero
```

```
R1=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
R2=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
R3=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
R4=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];

% Begin 64 cycles with 64-bit session key reordered by lsb of each byte fed into registers
for i=1:64
        R1Step=bitxor(bitxor(bitxor(R1(1),R1(2)),bitxor(R1(3),R1(6))),KeyLSB(i));
        for j=1:18
                R1(j)=R1(j+1);
        end
        R1(19)=R1Step;

        R2Step=bitxor(bitxor(R2(1),R2(2)),KeyLSB(i));
        for j=1:21
                R2(j)=R2(j+1);
        end
        R2(22)=R2Step;

        R3Step=bitxor(bitxor(bitxor(bitxor(R3(1),R3(2)),R3(3)),R3(16)),KeyLSB(i));
        for j=1:22
                R3(j)=R3(j+1);
        end
        R3(23)=R3Step;

        R4Step=bitxor(bitxor(R4(1),R4(6)),KeyLSB(i));
        for j=1:16
                R4(j)=R4(j+1);
        end
        R4(17)=R4Step;

        if i==0 % Enable to display a cycle of Stage 1 (manually edit value)
                fprintf('\nStage 1: cycle %d of 64\n',i)
                fprintf('R1\n')
                disp(R1)
                fprintf('R2\n')
```

```
                disp(R2)
                fprintf('R3\n')
                disp(R3)
                fprintf('R4\n')
                disp(R4)
                fprintf('\n End of cycle %d of 64\n',i)
        end
end % End of Stage 1
fprintf('\nStage 1 Output: 64 cycles ignoring stop go clock control\n\n')
fprintf('R1\n')
disp(R1)
fprintf('\nR2\n')
disp(R2)
fprintf('\nR3\n')
disp(R3)
fprintf('\nR4\n')
disp(R4)

% Stage 2: 22 cycles (No stop/go clock control)
% Begin 22 cycles with 22-bit Frame Number reordered by lsb fed into registers
for i=1:22
        R1Step=bitxor(bitxor(bitxor(R1(1),R1(2)),bitxor(R1(3),R1(6))),FrameLSB(i));
        for j=1:18
                R1(j)=R1(j+1);
        end
        R1(19)=R1Step;

        R2Step=bitxor(bitxor(R2(1),R2(2)),FrameLSB(i));
        for j=1:21
                R2(j)=R2(j+1);
        end
        R2(22)=R2Step;

        R3Step=bitxor(bitxor(bitxor(bitxor(R3(1),R3(2)),R3(3)),R3(16)),FrameLSB(i));
        for j=1:22
                R3(j)=R3(j+1);
```

```
        end
        R3(23)=R3Step;

        R4Step=bitxor(bitxor(R4(1),R4(6)),FrameLSB(i));
        for j=1:16
                R4(j)=R4(j+1);
        end
        R4(17)=R4Step;

        if i==0 % Enable to display a cycle of Stage 2 (manually edit value)
                fprintf('\nStage 2: cycle %d of 22\n',i)
                fprintf('R1\n')
                disp(R1)
                fprintf('R2\n')
                disp(R2)
                fprintf('R3\n')
                disp(R3)
                fprintf('R4\n')
                disp(R4)
                fprintf('\n End of cycle %d of 22\n',i)
        end
end % End of Stage 2
fprintf('\nStage 2 Output: 22 cycles ignoring stop go clock control\n\n')
fprintf('R1\n')
disp(R1)
fprintf('\nR2\n')
disp(R2)
fprintf('\nR3\n')
disp(R3)
fprintf('\nR4\n')
disp(R4)

% Stage 3: Set the bits R1[15]=R2[16]=R3[18]=R4[10]=1 and run for 99 cycles
% (stop/go clock control controlled by R4). Output is ignored, considered initialization stage.
R1(4)=1;
R2(6)=1;
```

```
R3(5)=1;
R4(7)=1;

for i=1:99
        Majority=[R4(7),R4(10),R4(14)]; % Vector to store clocking bit
        M=mode(Majority); % Majority Function for clock control

        if i==0 % Enable to display the clocking bits of R4 for a given cycle
                fprintf('\nClocking Bits for Cycle %d of 99\n',i)
                fprintf('R1 Clock Control - R4(10):')
                disp(R4(7))
                fprintf('R2 Clock Control - R4(3): ')
                disp(R4(14))
                fprintf('R3 Clock Control - R4(7): ')
                disp(R4(10))
        end

        if R4(7)==M % Determines if Register 1 is clocked by R4(7) bit value
                R1Step=bitxor(bitxor(R1(1),R1(2)),bitxor(R1(3),R1(6)));
                for j=1:18
                        R1(j)=R1(j+1);
                end
        R1(19)=R1Step;
        %disp('R1 Clocked') % Uncomment out to display if R1 is Clocked
        end

        if R4(14)==M % Determines if Register 2 is clocked by R4(14) bit value
                R2Step=bitxor(R2(1),R2(2));
                for j=1:21
                        R2(j)=R2(j+1);
                end
                R2(22)=R2Step;
                %disp('R2 Clocked') % Uncomment out to display if R2 is Clocked
        end

        if R4(10)==M % Determines if Register 3 is clocked by R4(10) bit value
```

```
            R3Step=bitxor(bitxor(bitxor(R3(1),R3(2)),R3(3)),R3(16));
            for j=1:22
                  R3(j)=R3(j+1);
            end
            R3(23)=R3Step;
            %disp('R3 Clocked') % Uncomment out to display if R3 is Clocked
      end

      % After the clocking of R1, R2, and R3 is determined, R4 is then
      % clocked for each cycle
      R4Step=bitxor(R4(1),R4(6));
      for j=1:16
            R4(j)=R4(j+1);
      end
      R4(17)=R4Step;
      %disp('R4 Clocked') % Uncomment out to display when R4 is Clocked

      if i==0 % Enable to display each register for a given cycle
            fprintf('\nStage 3: cycle %d of 99\n',i)
            fprintf('R1\n')
            disp(R1)
            fprintf('R2\n')
            disp(R2)
            fprintf('R3\n')
            disp(R3)
            fprintf('R4\n')
            disp(R4)
            fprintf('\n End of cycle %d of 99\n',i)
      end
end % End of Stage 3. Registers are now initialized.
fprintf('\nStage 3 Output: 99 cycles with stop go clock control\n\n')
fprintf('R1\n')
disp(R1)
fprintf('\nR2\n')
disp(R2)
fprintf('\nR3\n')
```

```
disp(R3)
fprintf('R4\n')
disp(R4)

% Stage 4: 114+114 cycles (stop/go clock control)
% Output is stored as 228-bit keystream: 114-bits for BTS to MS and
% 114-bits for Ms to BTS communication
Output=zeros(1,228); % Defining vector to store output from Stage 4
for i=1:228
        Majority=[R4(7),R4(10),R4(14)]; % Vector to store clocking bit
        M=mode(Majority); % Majority Function for clock control

        if i==0 % Enable to display the clocking bits of R4 for a given cycle
                fprintf('\nClocking Bits for Cycle %d of 228\n',i)
                fprintf('R1 Clock Control - R4(10):')
                disp(R4(7))
                fprintf('R2 Clock Control - R4(3): ')
                disp(R4(14))
                fprintf('R3 Clock Control - R4(7): ')
                disp(R4(10))
        end

        if R4(7)==M % Determines if Register 1 is clocked by R4(7) bit value
                R1Step=bitxor(bitxor(R1(1),R1(2)),bitxor(R1(3),R1(6)));
                for j=1:18
                        R1(j)=R1(j+1);
                end
                R1(19)=R1Step;
                % disp('R1 Clocked') % Uncomment out to display if R1 is Clocked
        end

        if R4(14)==M % Determines if Register 2 is clocked by R4(14) bit value
                R2Step=bitxor(R2(1),R2(2));
                for j=1:21
                        R2(j)=R2(j+1);
                end
```

```
        R2(22)=R2Step;
        %disp('R2 Clocked') % Uncomment out to display if R2 is Clocked
end

if R4(10)==M % Determines if Register 3 is clocked by R4(10) bit value
        R3Step=bitxor(bitxor(bitxor(R3(1),R3(2)),R3(3)),R3(16));
        for j=1:22
                R3(j)=R3(j+1);
        end
        R3(23)=R3Step;
        %disp('R3 Clocked') % Uncomment out to display if R3 is Clocked
end

% After the clocking of R1, R2, and R3 is determined, R4 is then
% clocked for each cycle
R4Step=bitxor(R4(1),R4(6));
for j=1:16
        R4(j)=R4(j+1);
end
R4(17)=R4Step;
%disp('R4 Clocked') % Uncomment out to display when R4 is Clocked

% Once the clocking of the registers has been completed, the majority
% function for each register produces one byte of output to be XOR'ed
% with the output from R1, R2, and R3.
MajorityR1=[R1(4),bitxor(R1(5),1),R1(7)]; % Vector to store R1 Majority bit
MR1=mode(MajorityR1); % Majority Function for R1

MajorityR2=[bitxor(R2(6),1),R2(9),R2(13)]; % Vector to store R2 Majority bit
MR2=mode(MajorityR2); % Majority Function for R2

MajorityR3=[R3(5),R3(7),bitxor(R3(10),1)];          % Vector to store R3 Majority bit
MR3=mode(MajorityR3); % Majority Function for R3

% After the XOR of R1(1), R2(1), R3(1), MR1, MR2, and MR3, one byte of the
%keystream is produced by msb first
```

```
        Output(1,i)=bitxor(bitxor(bitxor(bitxor(...
                        bitxor(R1(1),R2(1)),R3(1)),MR1),MR2),MR3);

        if i==0 % Enable to display each register and keystream bit for a given cycle
                fprintf('\nStage 4: cycle %d of 228\n',i)
                fprintf('R1\n')
                disp(R1)
                fprintf('Majority Function Output for R1\n')
                disp(MR1)
                fprintf('R2\n')
                disp(R2)
                fprintf('Majority Function Output for R2\n')
                disp(MR2)
                fprintf('R3\n')
                disp(R3)
                fprintf('Majority Function Output for R3\n')
                disp(MR3)
                fprintf('R4\n')
                disp(R4)
                fprintf('\nEnd of cycle %d of 228\n',i)
                fprintf('Keystream Bit Produced: %d\n',Output(1,i))
        end

end
% End of Stage 4
fprintf('\nStage 4 Output: 114+114-bit Keystreams\n')

% Creating vector to store first 114-bits for BTS to MS communications
OutputBTSMS=zeros(1,120); % Need size of 120 since 114 isn't a multiple of 8
for i=1:114
        OutputBTSMS(1,i)=Output(i);
end

% Converting 114-bits of BTS to MS communications into 15 bytes (decimal)
OutputBTSMSDecimal=zeros(1,15);
for j=1:15
```

```
        b=0;
        for k=0:7
                b=OutputBTSMS(1,8*j-7+k)*2^(7-k)+b;
        end
        OutputBTSMSDecimal(1,j)=b;
end

% Creating container to store second 114-bits for MS to BTS communications
OutputMSBTS=zeros(1,120); % Need size of 120 since 114 isn't a multiple of 8
for i=1:114
        OutputMSBTS(1,i)=Output(114+i);
end

% Converting 114-bits of MS to BTS communications into 15 bytes (Decimal)
OutputMSBTSDecimal=zeros(1,15);
for j=1:15
        b=0;
        for k=0:7
                b=OutputMSBTS(1,8*j-7+k)*2^(7-k)+b;
        end
        OutputMSBTSDecimal(1,j)=b;
end

% Display output of A5/2 Algorithm
fprintf('\nBTS to MS Keystream\n')
disp(dec2hex(OutputBTSMSDecimal))
fprintf('\nMS to BTS Keystream\n')
disp(dec2hex(OutputMSBTSDecimal))

% Display BTS to MS Keystream in Bindary
fprintf('\nBTS to MS Keystream\n')
for i=1:114
        fprintf('%d', Output(1,i))
end

% Display MS to BTS Keystream in Bindary
```

```
fprintf('\nMS to BTS Keystream\n')
for i=1:114
        fprintf('%d', Output(1,114+i))
end

% End of A5/2 Algorithm
```