ISSN 1688-2806

Universidad de la República
Facultad de Ingeniería

Ciphertext only Attacks against GSM security

Tesis presentada a la Facultad de Ingeniería de la Universidad de la República por

Eduardo Cota

en cumplimiento parcial de los requerimientos para la obtención del título de Magister en Ingeniería Eléctrica.

Directores de Tesis

Dr. Eduardo Giménez ..... Universidad de la República
Dr. Alfredo Viola ..... Universidad de la República

Tribunal

Mag. María Eugenia Corti ..... Universidad de la República
Dr. Federico Larroca ..... Universidad de la República
Dr. Federico Lecumberry ..... Universidad de la República

Director Académico

Dr. Pablo Belzarena ..... Universidad de la República

Montevideo
martes 19 junio, 2018

Ciphertext only Attacks against GSM security, Eduardo Cota.

ISSN 1688-2806

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.1).
Contiene un total de 151 páginas.
Compilada el martes 19 junio, 2018.
http://iie.fing.edu.uy/

Acknowledgements

A mi familia, por su amor incondicional y su apoyo en esta y todas las aventuras de la vida.

A mis tutores, Alfredo y Eduardo, por su soporte, apoyo, y por toda la paciencia que me tuvieron durante este largo proceso. Sin su conocimiento y experiencia esta tesis no hubiera sido posible.

A todos los amigos del IIE y la Facultad.

This page intentionally left blank

A Jeanela, Mariana y Virginia.

This page intentionally left blank

Abstract

Mobile communications play a center role in today's connected society. The security of the cellular networks that connect billions of people is of the utmost importance. However, even though modern third generation and fourth generation cellular networks (3G and 4G) provide an adequate level of security in the radio interface, most networks and mobile handsets can fall back to the old GSM standard designed almost three decades ago, which has several known security weaknesses.

In this work we study the security provided by the family of ciphering algoritms known as A5 that protects the radio access network of GSM, with emphasis on A5/1. We review the existing attacks against A5/1 and existing countermeasures, and show that the existing ciphertext only attacks against algorithm A5/1 [9], adapted to use the most recent Time Memory Data Tradeoffs, are realistic threats to fielded GSM networks when attacked by a resourceful attacker which uses current state of the art GPUs and CPUs.

We also study the existing Time Memory Data Tradeoff algorithms, extending the best known results for the Perfect Fuzzy Rainbow Tradeoff attack to the multi target case. These results allow the practitioner to calculate the parameters and tradeoff constants that best suit his application. We implemented the algorithms using parallel programming on CUDA GPUs and successfully validated the theoretical estimations.

The main contributions of this work can be summarized as follows:

- Extending the existing best results for the Perfect Fuzzy Rainbow Tradeoff attack in the single target scenario to the multi target scenario.
- Validating the theoretical calculation of the parameters and tradeoff constants of the Perfect Fuzzy Rainbow tradeoff through implementation for several scenarios.
- Describing one of the possible procedures for the choice of parameters for the Perfect Fuzzy Rainbow tradeoff.
- Presenting a new ciphertext only attack against A5/1 using the voice channel in GSM communication.
- Calculating the details of the ciphertext only attack in [9] and showing that the attack is a realistic threat today using a perfect fuzzy rainbow tradeoff attack and modern GPUs.

This page intentionally left blank

Table of contents

This page intentionally left blank

Chapter 1

Introduction and motivation

Today's "always online" society depends heavily on all kind of electronic communication, be it data or voice. The slew of applications depending on the telecommunication networks put an increasing pressure in the security of all telecommunication components, from the terminal equipment to the myriad components of the network. Although all the buzz is about the new forms of interaction enabled by the ubiquitous connectivity to the Internet, we still heavily depend on the "simple" services provided by telephony networks, both fixed and mobile, namely voice communication.

One of the remarkable changes in the last decade has been the tremendous uptake of wireless cellular communications, both for data and voice, in many places being more prevalent than traditional wireline voice and data services. This implies that security and privacy in wireless networks should be a concern to both providers and users of such services, who expect those systems to be as secure as their wired counterparts.

1.1. Privacy in cellular telecommunications

Even though today's cellular technology is moving beyond third generation wireless networks towards much faster fourth generation networks, the most ubiquitous cellular network in many parts of the world is still Global System for Mobile communications (GSM), the most prevalent second generation network.

The first generation of cellular communication networks, of which the Advanced Mobile Phone System (AMPS) was the most deployed standard, were characterized by being analog networks, meaning that voice communication was modulated onto a carrier and transmitted in analog form in one of several frequency channels available. Neither signalling nor voice traffic were cryptographically protected, and this led to a serious problem of cloned services. Analog services were superseded by digital cellular networks, so called second generation networks, in the 1990's, although analog service was still available in the United States and other parts of the world until well beyond the turn of the century.

Second generation networks are characterized by the digital transmission of sig-

nalling and voice communication. The most widely used second generation network is GSM, still accounting for a large part of today's cellular clients. What started as a voice-only service in the 1990's was later improved with short-message service, data transmission services (albeit at a very slow speed by today's standards), and other secondary services. Second generation networks included cryptographic protection of voice and signalling.

The third generation cellular networks are also digital, and improve quality, spectrum utilization, data transfer speed, and security. Third generation standards are CDMA2000 1xEV-DO, developed by 3rd Generation Partnership Project 2 (3GPP2) and used mostly in North America and to some extent in Japan, China, South Korea and India, and Universal Mobile Telecommunications System (UMTS), developed by the 3rd Generation Partnership Project (3GPP) and used in the rest of the world.

The fourth generation networks do away with the idea that voice calls are the main service offered, and implement an all-IP network, that is, a data only network. Voice calls are basically a secondary service on top of the data network. The main fourth generation network standard today is called Long Term Evolution (LTE). It improves on the security provided by third generation networks, and is currently the most secure network available providing commercial service.

Despite all the advances since GSM inception, it is still one of the most widely used networks globally. For instance, 4G Americas [1] estimates 3.2 billion GSM subscriptions worldwide in Q3 2016, which translates to about 42 percent market share. Besides, third and fourth generation handsets can fall back to GSM on underserved areas when there is no other network available. This makes GSM security a timely topic.

The privacy of GSM voice communications is protected by a family of stream ciphers known as A5. Each voice frame and each sensitive signalling frame is encrypted with a key shared between the network and the mobile phone, using one of the A5 algorithms. The original algorithms, known as A5/1 and A5/2, were developed in the late 1980s together with the GSM standard, by the Groupe Spéciale Mobile, which was originally a group of European post and telecommunications operators, and later a committee of the European Telecommunications Standards Institute (ETSI). A5/1 was the original algorithm for use in Europe, and at the time was believed to provide an adequate protection against eavesdropping. A5/2 was added later as a lower security option to be used when GSM was implemented outside Europe, due to export restrictions on strong cryptography. A5/1 and A5/2 were reverse-engineered by Briceno et al in 1999 from real handsets. A5/2 was almost immediately broken and current ETSI/3GPP recommendations forbid it's use, but A5/1, despite being shown weak multiple times in the last several years, is still the most widely used encryption algorithm in GSM.
At the end of the 20th century, a new cipher based on the Kasumi cryptosystem was designed to be used in UMTS, the third generation cellular network, and its use in GSM was standarized as A5/3. Kasumi is a block cipher, so for its use in GSM a cipher chaining mode is used to generate the necessary ciphering bits.

This algorithm, despite having some theoretical weaknesses, is much stronger than A5/1. Nevertheless, wireless carriers and handset manufacturers have been slow to adopt A5/3.

Beyond several theoretical attacks against A5/1, the most effective attacks to date relate to the small (for today's technology) internal state of A5/1, which enables brute force attacks using time memory tradeoff (TMTO) attacks. While the long-term solution to the weaknesses found in A5/1 is to abandon A5/1 and move to using the stronger A5/3, in many cases this requires expensive changes in the operator's infrastructure, and may generate incompatibilities with some mobile phones.

As we will see in chapter 5, TMTOs are family of cryptographic attacks used to invert a function, based on precomputing huge tables of relations between images and preimages in such a way that not all values need to be stored. Those tables are then used in the online or attack phase to carry out the attack. There is a tradeoff between the memory used to store the tables and the time taken by the online or attack phase. There are several TMTO variants proposed and studied in the literature, with the perfect fuzzy rainbow table tradeoff attack being shown as the best tradeoff in many realistic scenarios for the single target case, that is when we have a single captured cyphertext to attempt inversion. We extend the study of the parameters of the fuzzy rainbow table tradeoff to the multi-target case.

The first TMTO attack against A5/1 (at least in the public literature) not requiring unreasonable amounts of known plaintext was published by Barkan et al in 2003 [9], who proposed a ciphertext only attack. It is unknown if they calculated the required tables for their time-memory tradeoff, but no tables were publicly released. Between 2007 and 2010, three different groups set to calculate the huge tables needed to mount a Time-Memory tradeoff attack, and one of them, led by german security researcher Karsten Nohl, croudsourced the huge calculations needed to build those tables, and published them using the Bittorrent peer to peer protocol.

Nohl's attack depends on knowing some captured ciphertext (the encrypted communication) and the corresponding plaintext (the data before encryption) for the communication, so this corresponds to a known plaintext attack. In most fielded systems, there are several signalling messages with known content transmitted at the beginning of each voice call which can be used as source of known plaintext. The latest specifications from the 3GPP include countermeasures to avoid easily guessable messages in the control channels of GSM, which consist in randomizing certain bytes in the control messages so as to counteract Nohl's attack.

In this work we demonstrate that based on the ciphertext only attack proposed by Barkan et al [9], a Time-Memory tradeoff attack can be performed with minimal or no knowledge of the plaintext of the communication using an attainable amount of resources. Ciphertext only attacks are considered harder to defend against as the attacker does not need knowledge of the actual contents of the communication.

We also show a new cyphertext only attack against A5/1 using the voice channel and compare this new attack with the attack by Barkan et al. We also implement a reduced demonstration of the method by Barkan et al, extrapolating the calculation of the required resources for the full attack. Finally we propose a possible countermeasure to mitigate this threat.

1.2. Organization of the rest of this work

This work is organized as follows:

In chapter 2 we present a summary of the GSM architecture and its security properties, with the aim of introducing the reader to the topics necessary to understand the rest of this thesis. This chapter can be skimmed or skipped entirely if the reader is familiar with the GSM architecture and properties.

We review the attacks against A5/1 in the literature in chapter 3, following in chapter 4 by expanding the results of Barkan, Biham and Keller who presented a cyphertext only attack against A5/1 based on the redundancy due to the error detection and correction codes on the signalling channels. In that same chapter we present a new cyphertext only attack based on the redundancy in the voice channel of a call.

Both ciphertext only attacks presented in chapter 4 use a family of brute-force attacks known as Time Memory Data Tradeoff (TMDTO) attacks, which in turn are a kind of time memory tradeoff (TMTO) attack, so in chapter 5 we review the existing TMTO and TMDTO attacks, and in chapter 6 we extend the known results about the best TMTO attack, the perfect fuzzy rainbow table tradeoff, to the case when several captured ciphertexts are available to attempt inversion.

In chapter 7 we present an experimental validation of the results of chapter 6 with a synthetic problem, and show how an attacker could calculate the parameters of the tradeoff according to the available resources.

In chapter 8 we calculate the necessary resources to mount a ciphertext only attack against A5/1 based on the previous results and show that the attack is feasible with state of the art CPUs and GPUs even if the attacker wants a high success rate in the attack having very little captured ciphertext. We end the chapter describing our demo implementation of the attack, which is successful attacking A5/1 but due to our limited resources requires an unrealistically large number of captured ciphertexts.

Chapter 9 briefly discusses the conditions necessary to be able to apply the attack in a real fielded GSM network, and introduces mechanisms to counteract the attack.

Finally chapter 10 presents the conclusions of this work and introduces future research directions to improve the results of this thesis.

Chapter 2

The GSM architecture and its security properties

2.1. Brief description of the GSM architecture

This short description of the GSM network is aimed at introducing the reader not familiar with the public wireless cellular networks (and in particular the GSM family of networks) to the topics necessary to understand this thesis. For a complete description, a complete yet accessible book on GSM is [18]. The complete GSM specifications can be downloaded free of charge from the ETSI website (http://www.etsi.org/) or from the 3GPP website (http://www.3gpp.org).

Nowadays GSM stands for Global System for Mobile communications, however the original meaning of the GSM acronym was Groupe Spécial Mobile, the name given to the group formed to design a pan-European digital mobile technology in the 1980's. The GSM group was backed by several European countries, and the first set of specifications was completed in 1988. The first commercial service started in Finland in 1991, soon followed by many european and non-european countries. Since then GSM and its successors have been deployed in more than 230 countries, with more than 4800 million subscribers and 7800 million mobile connections by the end of 2016 according to GSMA Intelligence [44], a research group run by the GSM Association (GSMA).
A GSM network is a mobile wireless network, using radio frequency (RF) signals in several frequency bands from around 400 MHz to near 3 GHz depending on local regulations. To accommodate scarce RF resources, GSM resorts to frequency division multiplexing (FDM), time division multiplexing (TDM), and spatial frequency reuse. Frequency division multiplexing means that the available spectrum is divided in small frequency bands, and each user is assigned one such band for its transmission. Time division multiplexing means that different users share the same frequency band transmitting at different time intervals. For spatial frequency reuse, GSM is built as a cellular network, meaning that the service area is divided into small sub-areas called cells, each served by a different base station (BTS),

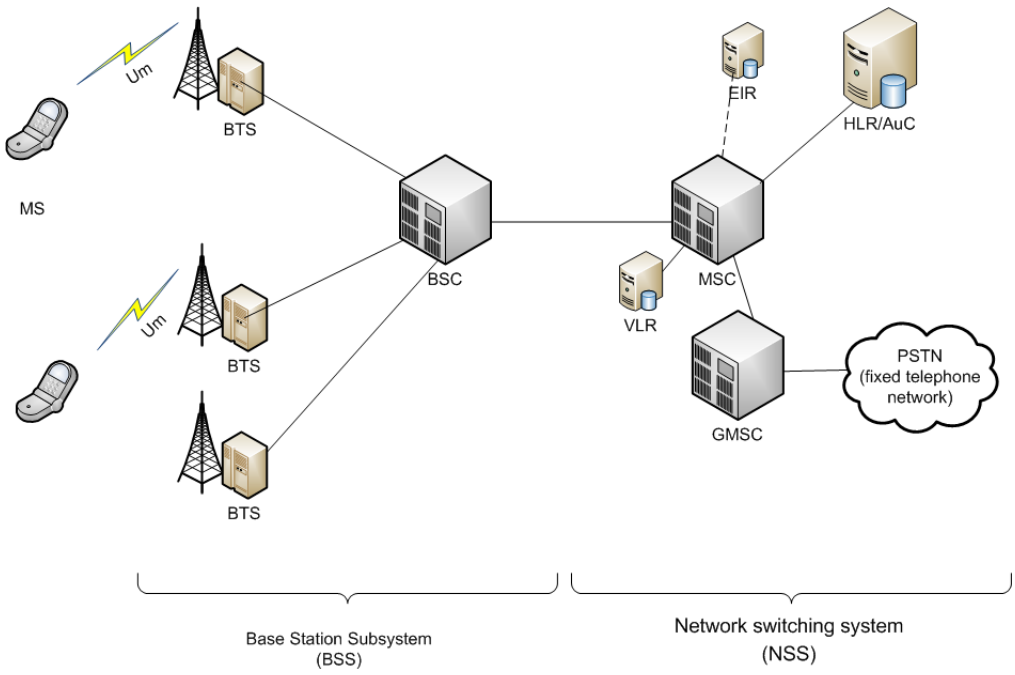

Figure 2.1: GSM Architecture

where non-contiguous cells can reuse the same frequency bands. The mobile phone connects to the closest base station, monitoring nearby cells so it can quickly select a new cell when RF conditions change as the person moves. This implies a distributed architecture which includes the means to mantain the user's session when the user moves from BTS to BTS.

A high level overview of the GSM architecture is presented in Figure 2.1. The main elements are:

- Mobile Station (MS) , which is the communication device in the GSM network. It consists of the Mobile Equipment (the Cell Phone) and the Subscriber Identity Module (SIM) used for validation and session key generation. The MS is the device the subscriber uses to interact with the network, and is responsible for network connectivity, voice digitization, call establishment and termination.

- Base Transceiver Station (BTS), responsible for carrying out radio communication between the network and all the MSs in the BTS's service area. The interface between the BTS and the MS is called Um interface (or air interface).

- Base Station Controller (BSC) which controls several BTSs, handling allocation of radio channels, power and signal measurements from the MS, handover between BTSs (if both BTSs are controlled by the same BSC), and encryption in the air interface. It concentrates traffic from a certain

area. The interface between the BSC and the BTSs is called Abis.

- Mobile Switching Center (MSC). The MSC handles call setup, call and SMS routing, switching functions, communication with other MSCs, and handoff between cells in different BSCs or different MSCs.
- Gateway Mobile Switching Center (GMSC). An MSC which also connects to the fixed network.
- Home Location Register (HLR). It is a database which stores information about subscribers, including MSISDN (phone number), IMSI (International Mobile Subscriber Identity), subscriber supplemental features and restrictions, and current location of the MS.
- Authentication Center (AuC). Contains the shared key unique to each subscriber. Handles the authentication and encryption tasks for the network. It is usually co-located with the HLR.
- Visitor Location Register (VLR). It is a subsidiary database designed to limit the amount of queries to the HLR. It stores information about the subscribers currently being served by one or a group of MSCs, and is usually co-located with some or all of the MSCs in the network.
- Equipment Identity Register (EIR). Keeps lists of mobile phone identities (IMEI) to be allowed or barred from the network, usually used to block stolen phones in the network.

There are several other subsystems, responsible for functions like billing, voicemail, SMS, MMS, data transmission, etc., which are not described here as they do not concern our work.

2.2. Identification of the subscriber

GSM uses different identifiers for different purposes. The main identifiers are:

- The International Mobile Subscriber Identity (IMSI), which uniquely identifies each mobile service. It is permanently stored on the SIM card.
- The Mobile Station Integrated Services Digital Network number (MSISDN), which simply put is the mobile's phone number. It is used to route calls to the client.
- The Temporary Mobile Subscriber Identity (TMSI), which is a temporary identifier assigned by the serving network, to avoid easy identification of the MS.
- The International Mobile Equipment Identity (IMEI), which identifies the mobile device and should be globally unique. It is assigned by the phone manufacturer and is not tied to the subscriber identity.

The IMSI number is a 15 digit number composed of the 3 digit Mobile Country Code (MCC) which identifies the country, the 2 or 3 digit Mobile Network Code (MNC), which identifies the operator inside the country, and 9 or 10 digits identifying the subscriber. It is used to uniquely identify the subscriber, and comes preloaded in the SIM card.

The MSISDN number is a variable length number which follows the international telephone numbering plan. It is composed of the Country Code (CC), a 1-3 digit number which identifies the country, the National Destination Code (NDC), which identifies one network within the country, and the Subscriber Number (SN). NDC and SN structure is specified in the national numbering plans by the telecommunication regulator in each country. The MSISDN is associated to an IMSI in the HLR.

The IMEI is a 15 or 16 digit decimal number, which identifies the equipment, model and serial number of the device. Many countries use the IMEI to reduce the incidence of mobile phone theft, by implementing black lists in the EIR containing the IMEI numbers of stolen phones, so as to deny service to any device reported as stolen.

2.3. The radio link in GSM

In this work we are mostly interested in the communication between the BTS and the mobile station (the Um interface in GSM jargon), where encryption is used to protect the communication between the mobile station and the network. The Um interface can be logically divided into three layers, each one with defined functions:

- Layer 1 (Physical layer). Responsible for the actual radio transmission, multiplexing, timing, and coding.
- Layer 2 (Data link layer). Uses a message protocol derived from fixed digital networks, called LAPDm, for the communication of signalling messages. It is responsible for framing, multiplexing, error control, etc.
- Layer 3 (Network layer). Has three sublayers, responsible for radio resource management (asignment and release of logical channels), mobility management (user authentication and location tracking from cell to cell), and Call Control (which controls telephone calls, eg. establishment and release of the call)

2.3.1. Physical and logical channels

Wireless spectrum is a scarce resource, which must be shared among all subscribers in a service area. GSM uses several multiplexing mechanisms to share the available spectrum in a fair way.

Spectrum is allocated in a paired fashion, meaning that for each downlink frequency channel there is a corresponding uplink channel. This means that most

of what we say about one direction applies to the other direction using a slightly different frequency.

At the lowest layer frequency and time multiplexing is used. The available frequency band is divided in frequency channels spaced 200 kHz, and several frequency channels are assigned to each BTS. Each frequency channel is divided into eight timeslots (channels) using time division multiplexing. Each of those channels can be used to send signaling or one voice stream (for full rate configuration) or two (for half-rate configuration).

At higher layers, statistical multiplexing is used to share the available bandwidth, serving most (idle) terminals with shared control channels, and only allocating dedicated channels when needed (for example to the devices with an ongoing voice call).

Physical channels

Physical channels are the actual frequencies and timeslots used by the MS and BSC for a single transmission. The available spectrum is divided into 200 kHz frequency channels, and each cell is allocated some of the available frequency channels (cell allocation). One of those frequency channels is known as the BCCH carrier or BCCH physical channel, and carries synchronization information and the Broadcast Control Channel (BCCH) logical channel (it may optionally also be used for other logical channels). The rest of the channels are allocated as needed for voice and signalling.

Time is partitioned in timeslots, TDMA frames, multiframes, superframes and hyperframes [31]. In GSM, the minimum unit of transmission is called a timeslot or burst, and has a duration of 3/5200 s ($\approx 577\mu s$). Eight timeslots shall form a TDMA frame ($\approx 4,62ms$). The eight timeslots in a frame are numbered $0-7$ and are referred to by their Timeslot Number (TN), and frames are numbered from 0 to $FN_{MAX} = (26 \times 51 \times 2048) - 1 = 2715647$ in what is called the TDMA Frame Number (FN). This FN is used as input to the ciphering algorithm in the air interface.

Each individual communication (for instance, each voice call) uses only one timeslot of each frame (half a timeslot in some cases), which means there can be 8 (or 16) simultaneous communications in each frequency channel.

Frames are organized in a hierarchy (see figure 2.2). For traffic and associated control channels, 26 frames are grouped in a 26-multiframe, while for common control, broadcast and stand alone dedicated control, a 51-multiframe is used (comprised of 51 TDMA frames). 51 traffic multiframes or 26 broadcast multiframes (that is, $51 \times 26 = 1326$ frames) comprise a superframe, and 2048 superframes are grouped into an hyperframe. This means there are $26 \times 51 \times 2048 = 2715648$ frames in an hyperframe, numbered from 0 through FN_{MAX}. An hyperframe lasts about 12534 s, or about 3 hours 28 minutes and 54 seconds.

The basic modulation in GSM is GMSK (Gaussian Minimum Shift Keying) which modulates 1 bit per symbol, and the standard rate is 270.833 K symbols/se-

cond. This means that the duration of each burst corresponds to 156,25 symbols, of which 147 are useful symbols and the rest are guard times. There are five types of bursts defined in [31]. We are interested in Normal bursts, which carry voice and signalling. The structure of a normal burst is depicted in Figure 2.3. It consists of:

- 3 "tail bits" which mark the start of the burst
- 57 encrypted bits, which carry voice or signalling
- stealing flag, one bit indicating if the preceeding 57 bits consist of data or signalling
- 26 bits used as a training sequence for the receiver
- stealing flag, one bit indicating if the following 57 bits consist of data or signalling
- 57 encrypted bits which carry voice or signalling
- 3 "tail bits" which mark the end of the burst
- a guard period equivalent to 8,25 bits between bursts

As we can see, there are $2 \times 57 = 114$ data bits in each burst, split into two 57-bit blocks.

The stealing flag merits some explanation. When a burst is used for voice traffic, the stealing flag indicates if the corresponding 57-bit block is used for signalling, and has thus been "stolen" from the voice traffic. This mechanism, which lowers voice quality, is only used to send urgent signalling data like handover information, call control, etc.

Optional frequency hopping

One optional but commonly used functionality in GSM is frequency hopping, which means that the transmission frequency is changed periodically according to a predefined algorithm. The algorithm used in GSM selects a new frequency for each burst. This frequency hopping is designed to improve Signal-to-Noise-Ratio (SNR) when the signal is affected by frequency-selective interference (that is, interference that only damages signals in a narrow frequency band) or fading (which is a physical phenomena where some frequencies are disproportionately attenuated). The effect of frequency hopping is to average the interference over the frequencies of one cell. To calculate the frequency to use, the MS is assigned a subset of the frequencies allocated to a cell, called the Mobile Allocation (MA), an offset, the Mobile Allocation Index Offset (MAIO), and a Hopping Sequence Number (HSN). There is a table giving a pseudo-random sorting of the MA frequencies, selected by the HSN, which is known by the MSs. Usually all MSs in the cell are assigned the same MA and HSN, and different mobiles select different frequencies for the same timeslot using the MAIO, which is the offset into the MA table corresponding to the MS. Then, each MS selects the frequency according to its MAIO and the current Frame Number [31].

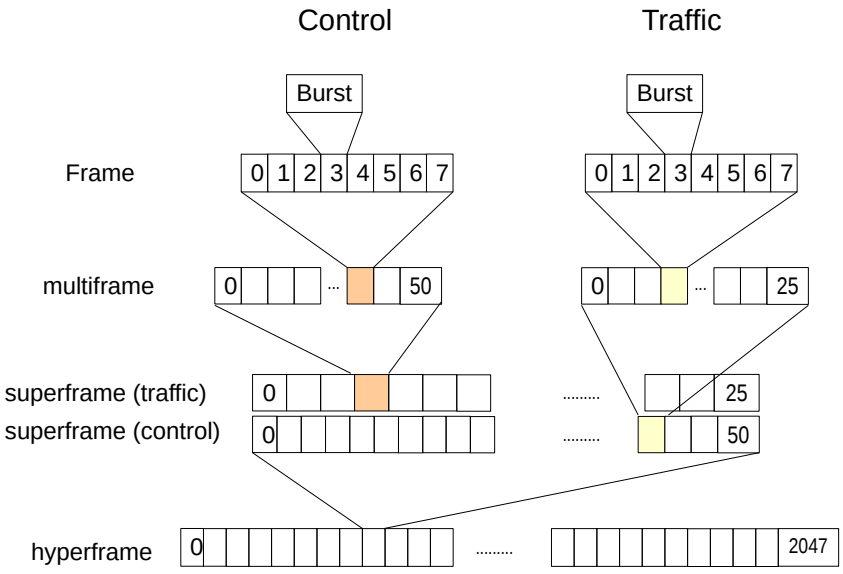

Figure 2.2: GSM frame hierarchy

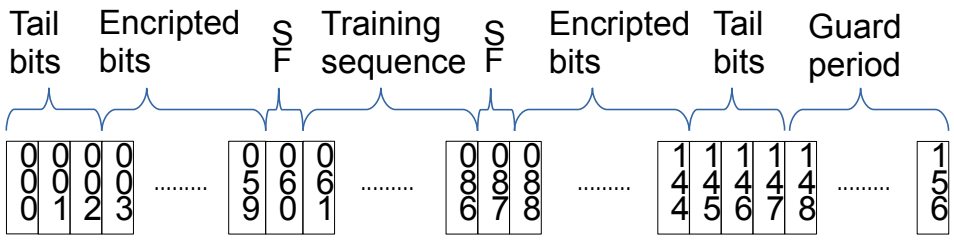

Figure 2.3: GSM normal burst

Logical channels

Logical Channels carry voice, data, and signalling, and are mapped to physical channels according to several parameters configured on the network [29].

There are two kind of channels, traffic channels and control channels.

Traffic CHannels (TCHs) carry encoded voice communications (or data). The original traffic channels for GSM are the Full rate Traffic CHannel (TCH/FS), with a gross rate of 22.8 kbps, and the Half rate Traffic CHannel (TCH/HS), with a gross rate of 11.4 kbps. Further encodings were defined in more recent versions of the standards, but we will stick with TCH/FS which is the most commonly used in GSM. A TCH/FS channel occupies a single timeslot of each TDMA frame,

which means that up to eight simultaneous communications can be accomodated in a 200 khz channel. TCH channels use 26-multiframes, meaning that the traffic sequence is organized accorging to a pattern that repeats each 26 frames.

Control channels are intended to carry signalling or synchronization data. Different channels have different requirements and occupy different portions of the available capacity. In the service offered to subscribers, signalling can be considered an overhead, thus an attempt was made to minimise its impact on the utilization of the scarce RF resources available.

There are three categories of control channels for GSM in cellular communications, namely broadcast, common, and dedicated channels.

Broadcast channels are used by the BSS to broadcast the same information to all MSs in a cell. There are channels for frequency correction (FCCH), synchronization (SCH) and for broadcast of information common to all UEs being served by the BTS (BCCH). Common control channels are used for paging the UEs (PCH), random access in the uplink (RACH) to request assignment of a dedicated channel, access grant channel (AGCH) to notify channel assignment, and a notification channel (NCH) used to inform MSs about incoming group and broadcast calls.

Broadcast and common control channels carry important information for the system, but they will not be described further except when needed, as they are not encrypted.

The third group of signalling channels, dedicated channels, are bi-directional point to point channels, used to carry information relevant to a single user. They comprise the Stand-alone Dedicated Control Channel (SDCCH), used between the MS and the BSS when there is no active connection, for instance to update location information, to set-up the necessary channels for a communication, or to send an SMS, the Slow Associated Control Channel (SACCH), always assigned and used together with a TCH or SDCCH, carries information for the radio operation like transmitter power control, synchronization and reports on channel measurements, and Fast Associated Control Channel (FACCH), which is a logical channel always associated with a TCH, and is created by "stealing" blocks from the TCH when urgent information must be sent (like call establishment/release or handovers).

Mapping of dedicated logical channels into physical channels

Each logical channel has a set of rules on how it is mapped into the frame hierarchy [32]. For this explanation it is useful to define $T1 = FN \; div \; 1326$, $T2 = FN \; mod \; 26$, $T3 = FN \; mod \; 51$ [31]. Those same quantities are needed later on to explain encryption in GSM. A 26-multiframe will always start when $T2 = 0$, and a 51-multiframe when $T3 = 0$.

The TCH/FS traffic channel and its associated SACCH channel use a 26-frame multiframe, represented in Figure 2.4. This figure represents a single timeslot for each frame. There are 24 traffic timeslots dedicated to the TCH channel, which carry compressed voice, a timeslot dedicated to the associated SACCH channel (either in frame 12 or frame 25), and a free timeslot. For even timeslots the SACCH channel uses frame 12, for odd timeslots it occupies frame 25.

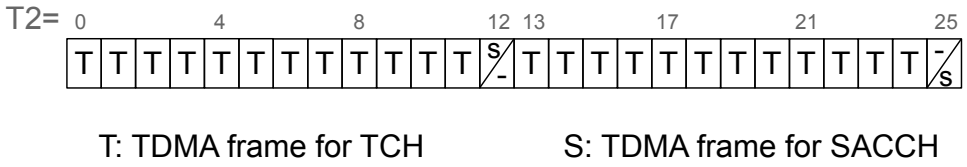

Figure 2.4: TCH/FS multiframe

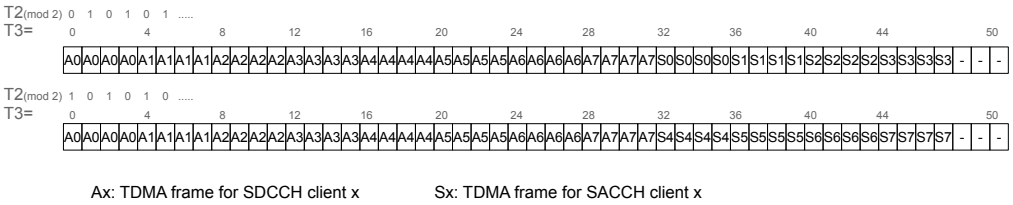

Figure 2.5: SDCCH multiframe

The FACCH channel is only assigned when needed, by pre-empting half the information bits of the TCH/FS to which it is associated in eight consecutive bursts [25]. The stealing bit is used to indicate whether the 57 corresponding traffic bits carry voice traffic or signalling for the FACCH.

The SDCCH channel and its associated SACCH channel, are mapped into 51-multiframes. In this channel up to eight different mobiles share the same timeslot, multiplexed in time. The channel sequence repeats every two 51-multiframes. A diagram of the SDCCH channel in the downlink is shown in Figure 2.5

As we will see later we are specially interested in the SACCH channel associated with a voice call, that is the SACCH channel associated with a TCH channel.

2.3.2. Voice communication in the GSM network

GSM is a digital network. As such, it cannot directly transmit analog voice signals, so the analog sound signal is digitized and heavily compressed for transmission.

The analog speech signal at the transmitter is sampled at a rate of 8000 samples per second and quantized with a resolution of 13 bits per sample, which gives a bit rate of 104 kbit/s [18]. This raw bit stream is split into 20 ms frames containing 160 samples, and each frame is compressed into a 260 bit coded speech block, which gives a bit rate of 13 kbit/s, for an 8 : 1 compression ratio. Each 260-bit block is used as input for the coding stage.

There are other optimizations, like Discontinuous Transmission with comfort noise generation, which stops transmission during speech pauses, reducing battery consumption and the level of interference for other users. These silences are filled by the receiver with what is known as comfort noise generation, a synthetic back-

ground noise signal designed to avoid the disturbing effect a sudden silence has on the listener.

2.4. Channel Coding

Both signalling and user data are encoded, reordered and interleaved to improve reliability by building resistance to channel errors. GSM uses a combination of block coding for error detection and convolutional coding for error correction, followed by an interleaving scheme to deal with burst errors. This process is carried out before encryption. Each channel has its own coding and interleaving scheme, using the same basic building blocks to simplify encoder/decoder. Coding is exhaustively described in [22] for all possible GSM channels, we will only present as examples the case of the TCH/FS and SACCH channels which will be needed later on. The source data (either compressed voice or signalling messages) is received by the channel coder in data blocks. For instance, the speech coder generates a 260-bit block every 20 ms. Each data block is individually protected by a block code which generates parity bits for error detection in the block. Depending on the channel, either a Cyclic Redundancy Check (CRC) or Fire code is used. Then some fill bits are added and a convolutional code is used to add redundancy for error correction. The result of the convolutional coding is a 456 bit block for most channels. As a final step blocks are interleaved to reduce the effect of burst errors by spreading them over several blocks. The resulting 456-bit blocks are then split into 114 bit blocks which are fed into the encryption process and then sent using 114 bit physical channel bursts.

In the receiver, the inverse process is carried out. First deinterleaving, then convolutional decoding, and finally parity checking. If the block code detects errors after convolutional decoding the frame is discarded.

2.4.1. Coding for SACCH y SDCCH channels

Each protocol message in signalling channels has a fixed length of 23 bytes (184 bits). This means that the input to the block coding is a 184 bit block $d(0)...d(183)$.

First step: parity

Most signalling channels, including SACCH and SDCCH, use a shortened binary cyclic code or Fire code using the generator polynomial $g(D) = (D^{23} + 1)(D^{17} + D^3 + 1)$.

Let $p(0), ..., p(39)$ be the parity bits.

The encoding of the cyclic code is performed in a systematic form, which means that, in GF(2), the polynomial:

$d(0)D^{223} + d(1)D^{222} + ... + d(183)D^{40} + p(0)D^{39} + ... + p(38)D + p(39)$

when divided by $g(D)$ yields a remainder equal to $1 + D + D^2 + ... + D^{39}$ [22]

The inversion of the parity bits ensures that the null code word is not valid, i.e. bursts that contain all zeros cannot occur in the channel.

After adding the parity bits, we have $224 = 184 + 40$ bits

$$u(i) = d(i) \text{ for } 0 \leq i \leq 183$$

$$u(i) = p(i - 184) \text{ for } 184 \leq i \leq 223$$

Second step: tail bits

Four zero bits, called tail bits, are added, reaching 228 bits of input to the convolutional encoder. These 4 bits allow a defined resetting procedure for the convolutional encoder (zero termination) and thus a correct decoding decision.

$$u(i) = 0 \text{ for } 224 \leq i \leq 227$$

Third step: convolutional coding

A half rate convolutional encoder is used, which means that for each input bit there are two output bits, defined by the polynomials

$$G0 = 1 + D^3 + D^4$$

$$G1 = 1 + D + D^3 + D^4$$

The result is thus a 456 bit block $c(0), c(1), ..., c(455)$ defined by:
$c(2k) = u(k) + u(k-3) + u(k-4)$
$c(2k+1) = u(k) + u(k-1) + u(k-3) + u(k-4)$
where $k = 0, 1, ..., 227$ and $u(k) = 0$ for $k < 0$

Fourth step: interleaving

The idea of interleaving is to spread the effect of burst errors inside the message or between successive messages. In the case of signalling, the bits of a single 456 bit block are mixed across four 114 bit blocks which are sent in 4 bursts of 114 bit each.

If we call $B0$ the number of the first burst carrying bits from the first data block in the transmission, message n will be sent in the four 114-bit blocks $B0 + 4n \cdots B0 + 4n + 3$, and if we call $i(x, y)$ the bit y of block x after interleaving, then the position of each bit in the reordered interleaved blocks are given by:
$i(B, j) = c(n, k)$ for $k = 0, 1, ..., 455$
$n = 0, 1, ..., N, N + 1, ...$
$B = B0 + 4n + (k \mod 4)$
$j = 2((49k) \mod 57) + ((k \mod 8) \div 4)$

2.4.2. Coding for a TCH/FS channel

Input for a TCH/FS channel is a 260 bit block produced by the voice coding process. Not all bits are equally important for the reconstruction of the voice signal, so GSM splits the bits into two classes, 182 class 1 bits, which are error protected, and 78 class 2 bits, which are not protected [22]. The 182 class 1 bits are further classified according to their relative importance in reconstructing the voice signal in 50 bits protected by a cyclic code and a convolutional code, and 132 bits only protected by the convolutional code.

Parity bits

Only the first 50 class 1 bits are protected by a three bit CRC calculated using the generator polynomial $g(D) = D^3 + D + 1$. Just like in the signaling channels, parity bits are inverted so that the remainder left by dividing the polinomial $d(0)D^{52} + d(1)D^{51} + \ldots + d(49)D^3 + p(0)D^2 + p(1)D + p(2)$ by the generator polinomial $g(D)$ is $1 + D + D^2$.

Reordering and tail bits

Class 1 bits are reordered and 4 tail bits are added, yielding a 189 bit block for class 1 bits

$$u(k) = d(2k) \text{ and } u(184 - k) = d(2k + 1) \text{ for } k = 0, 1, \ldots, 90$$

$$u(91 + k) = p(k) \text{ for } k = 0, 1, 2$$

$$u(k) = 0 \text{ for } k = 185, 186, 187, 188 \text{(tail bits)}$$

Convolutional Coding

Class 1 bits are protected by a half rate convolutional coder, defined by the same generator polynomials used in the SACCH and SDCCH channels $G0 = 1 + D^3 + D^4$ and $G1 = 1 + D + D^3 + D^4$. This means Class 1 bits are expanded to 378 bits, and adding the 78 Class 2 bits yields a 456 bit block $c(0), c(1), ..., c(455)$:

$$c(2k) = u(k) + u(k - 3) + u(k - 4) \text{ for } k = 0, 1, \ldots, 188$$

$$c(2k + 1) = u(k) + u(k - 1) + u(k - 3) + u(k - 4) \text{ for } k = 0, 1, \ldots, 188$$

$$u(k) = 0 \text{ for } k < 0$$

$$c(378 + k) = d(182 + k), k = 0, 1, \ldots, 77$$

Interleaving

For the TCH/FS channel blocks are spread in what is known as "diagonal interleaving", which means that bits from different blocks are mixed in the same burst. In this case, the 456 bits of the block are split into eight bursts, and each burst has bits from two different blocks.

If we call $B0$ the number of the first burst carrying bits from the first data block in the transmission, message n will be sent in the eight 114-bit blocks $B0 + 4n \cdots B0 + 4n + 7$, and if we call $i(x, y)$ the bit y of block x after interleaving, then the position of each bit in the reordered interleaved blocks are given by the following formulas:

$$i(B, j) = c(n, k) \text{ for } k = 0, 1, ..., 455$$

$$n = 0, 1, ..., N, N + 1, ...$$

$$B = B0 + 4n + (k \mod 8)$$

$$j = 2((49k) \mod 57) + ((k \mod 8) \div 4)$$

Bits from block n will occupy the even bits of the first four interleaving bursts, and the odd bits of the last four interleaving bursts. The even bits of the last four interleaving blocks are occupied by bits from block $n+1$. This diagonal interleaving has the advantage of distributing bit errors within a block and between blocks, but has the disadvantage of introducing additional delays in the reception, as all eight bursts must be received to recover block n.

2.5. GSM security

Being a wireless technology, GSM had to solve some security problems to be considered a viable product. From the operator's point of view, GSM must ensure that service is being provided to a registered customer, that the correct party is billed for the service, that communication cannot be eavesdropped from the air interface, and (ideally) that the system is immune to interference. This had to be solved under the restrictions of a mobile device, constrained both in computing power and battery capacity by cost and the available technology in the early 1990s. This reduced the choice of algorithms and protocols that could be used. Also, some important security functions were not included, like network authentication towards the user (guaranteeing the user that he is not connected to a rogue network), which can enable a man in the middle attack.

The main security-related functions are described in [23]. Referring to the interfaces in the reference architecture in Figure 2.1, the security measures standarized in GSM protect the confidentiality in the Um interface (the air interface) between MS and BTS using (optional) ciphering, allow the network to authenticate the subscriber, and protect the identity of the subscriber by using a temporary

identity known as Temporary Mobile Subscriber Identity (TMSI) whenever possible instead of the International Mobile Subscriber Identity (IMSI). Security in the rest of the interfaces is left open for the operator to decide.

One of the important decisions made during the design of the GSM standard was to include a Smart Card in each MS. This smart-card is called SIM, and includes the cryptographic material and algorithms for user authentication and session key generation. This greatly simplifies distribution of the master key shared between the network and the subscriber, which comes preloaded in the SIM, and makes the phone independent from the operator as it does not need to include security secrets particular to the network it is connected to. As another side effect, the security of the cryptographic material does not depend on the security of the phone, as it never leaves the SIM.

Authentication

Authentication in GSM is based on a shared key, K_i, only known to the SIM and the AuC. This key is 128 bit long in the reference authentication algorithms, and even though each carrier can select its own algorithm, it is expected most carriers are using one of the reference algorithms. The key is effectively tied to the IMSI of the subscriber, which in turn is tied to the SIM card.

Before granting services to a mobile device, it must perform an authentication procedure to validate itself in the network. This procedure is based on a challenge-response protocol, which is carried out by the MS and the MSC, with the help of the SIM and the AuC. Together with the authentication procedure a temporary shared key is produced to encrypt the communication. The algorithms used for authentication and key generation are known as $A3/A8$ respectively, and can be chosen by each network operator independently, although many operators use one of the reference algorithms available.
Authentication is usually carried out whenever the MS requests a service, and can also be requested by the network whenever it is deemed necessary. The involved parties and interactions on the authentication signalling are shown on Figure 2.6.

The key never leaves the AuC, so all calculations must be performed there. However, the AuC does not have a direct interaction with the mobile device. Instead, whenever authentication is need, the authenticating device (the MSC in the case of voice communications) obtains the subscriber's IMSI (International Mobile Subscriber Identity) and requests the AuC an authentication vector for that IMSI which is used to authenticate the subscriber. The authentication vector for the standard SIM is known as a "Triplet". A Triplet consists of a random 64 bit challenge *RAND*, the corresponding expected response from the MS, *SRES*, calculated as the result of running the A3 algorithm on the key and *RAND*, and the corresponding session key K_C, the output of the algorithm A8 with imput K_i and *RAND*. For the actual authentication, the MSC sends *RAND* to the mobile device MS through the BSC, and the mobile device hands it over to the SIM card. The card calculates the output of the A3 and A8 algorithms using *RAND* and

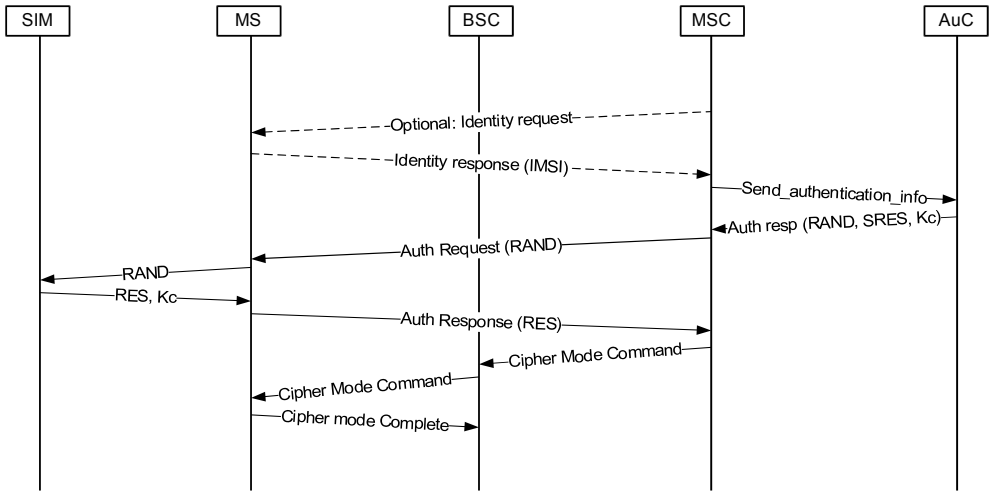

Figure 2.6: Authentication procedure

the shared key, and returns the calculated RES and corresponding session key K_C to the MS. The mobile returns RES to the BSC/MSC and keeps K_C to use it for encryption. To authenticate the subscriber the network compares the values of $SRES$ and RES. If they are the same the mobile device is considered authenticated, and the MSC and mobile share a key K_C that can be used for encryption of the communication. Finally, the MSC instructs the BSC and the MS to start ciphering the communication (unless ciphering is disabled).

Confidentiality (encryption)

After the authentication process, the MSC and the MS share a session key K_C, and unless encryption is disabled, the MSC forwards the key K_C to the BSC/BTS, and instructs the mobile (and BSC/BTS) to start encrypting the communication. From then on, the voice call and important signalling messages are encrypted using one of the A5 algorithms, chosen based on the capabilities of the mobile device and the BTS. The A5 algorithms are stream ciphers, that receive as input the session key K_C and a number calculated from the frame number called COUNT to re-initialize the algorithm at each frame, and generate 228 bits of keystream. The first 114 bits are used to encrypt a burst in the downlink direction by bitwise exclusive OR with the 114 payload bits of the corresponding downlink burst, while the remaining 114 are used in the same way in the uplink direction.

The COUNT value is derived from the frame number FN [23], concatenating the values of $T1$, $T3$ and $T2$ which are defined in [31] as $T1 = FN\ div\ (23 \times 51)$, $T2 = FN\ mod\ 26$, $T3 = FN\ mod\ 51$ just as we saw when presenting the physical channels. $T1$ is 11 bits long, $T2$ is 5 bits long, and $T3$ is 6 bits long. COUNT is a 22 bit long number, as represented in Figure 2.7, where bit 22 is the most significant bit.

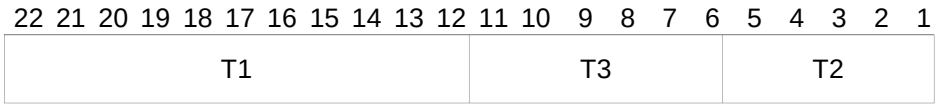

Figure 2.7: Coding of COUNT

2.6. The A5 family of stream ciphers

For encryption in the air interface GSM designers chose to use stream ciphers due to them being easy to implement efficiently in hardware both with respect to performance and complexity. Up to seven algorithms can be defined (withouth counting A5/0 which means no encryption), and four have been defined to date.

The initial releases of GSM included a single algorithm for encryption, called A5/1, which at the time was believed to offer adequate security. However, once GSM started spreading outside Europe, export restrictions forced the development of a weakened algorithm, A5/2. Both algorithms were kept secret and only revealed to GSM manufacturers on a need-to-know basis, but in 1999 Briceno, Goldberg and Wagner reverse-engineered both A5/1 and A5/2 from real handsets [17].

In 2002 an additional algorithm was added, A5/3. This algorithm is based on the Kasumi block cipher used in third generation networks, which in turn is a modification of the Misty1 algorithm developed and patented by Mitsubishi Electric corporation. A5/3 is stronger than A5/1, but its adoption by operators has been slow. Recently A5/4 has been defined, also based on Kasumi but with a 128 bit shared key.

Besides the aforementioned algorithms, there is a fallback option, called A5/0, which means no encryption. This can be used, if permitted by the network, when there is no common algorithm between the network and the MS.

2.6.1. The A5/1 algorithm

The A5/1 stream cipher accepts a 64-bit session key K_C and a 22 bit value COUNT which in GSM is derived from the FN as seen in the previous section. For GSM 228 bits are produced for each value of count, 114 are used to encrypt a single burst in the downlink direction, and the remaining 114 are used to encrypt a burst in the uplink.

A5/1 uses three maximal length Linear Feedback Shift Registers (LFSRs) $R1$, $R2$ and $R3$, of lengths 19, 22 and 23 bits respectively, thus the internal state or memory of the algorithm consists of 64 bits. A diagram of A5/1 is represented in figure 2.8. Each shift register advances when it receives a clock signal from the clocking unit, where advance means that the leftmost bit becomes the output, all bits are shifted left one position, and the rightmost bit is filled with the XOR of the tap bits.

The non-linearity in the system is introduced by the clocking unit, which decides which shift registers should advance at each step. The clocking unit takes as

input one bit from each shift register, and uses a very simple algorithm to decide which shift registers should advance: each register is clocked if and only if its tap bit coincides with the majority of the tap bits:

- Calculate the majority M of the tap bits $R1[8], R2[10], R3[10]$ (that is, $M = 0$ if there are 2 or 3 zeros, and $M = 1$ if there are 2 or 3 ones in the set $R1[8], R2[10], R3[10]$)
- Clock $R1$ if $R1[8] = M$
- Clock $R2$ if $R2[10] = M$
- Clock $R3$ if $R3[10] = M$

Notice that either two or three registers advance at each step, since at least two taps coincide with the majority.

Before generating the stream output, the internal state of A5/1 needs to be initialized. First the key and counter are fed to the three shift registers, advancing the three registers after each bit without taking into consideration the clocking unit. This part of the initialization is linear on the bits of the key and COUNT. After the linear part, the algorithm is run for 100 cycles with the clocking unit engaged, discarding its output.

1. Set $R1 = R2 = R3 = 0$
2. For $i = 0$ to 63
   - Clock $R1$, $R2$, $R3$
   - $R1[0] = R1[0] \oplus K_C[i]$
   - $R2[0] = R2[0] \oplus K_C[i]$
   - $R3[0] = R3[0] \oplus K_C[i]$
3. For $i = 0$ to 21
   - Clock $R1$, $R2$, $R3$
   - $R1[0] = R1[0] \oplus COUNT[i]$
   - $R2[0] = R2[0] \oplus COUNT[i]$
   - $R3[0] = R3[0] \oplus COUNT[i]$
4. Clock A5/1 100 times using the clocking unit

Only after the preceding initialization stage is complete are the output bits used for encryption, which means that the ciphering uses bits 101-328. The first 100 output bits are discarded to ensure the initial state is mixed by the irregular clocking before using the output.

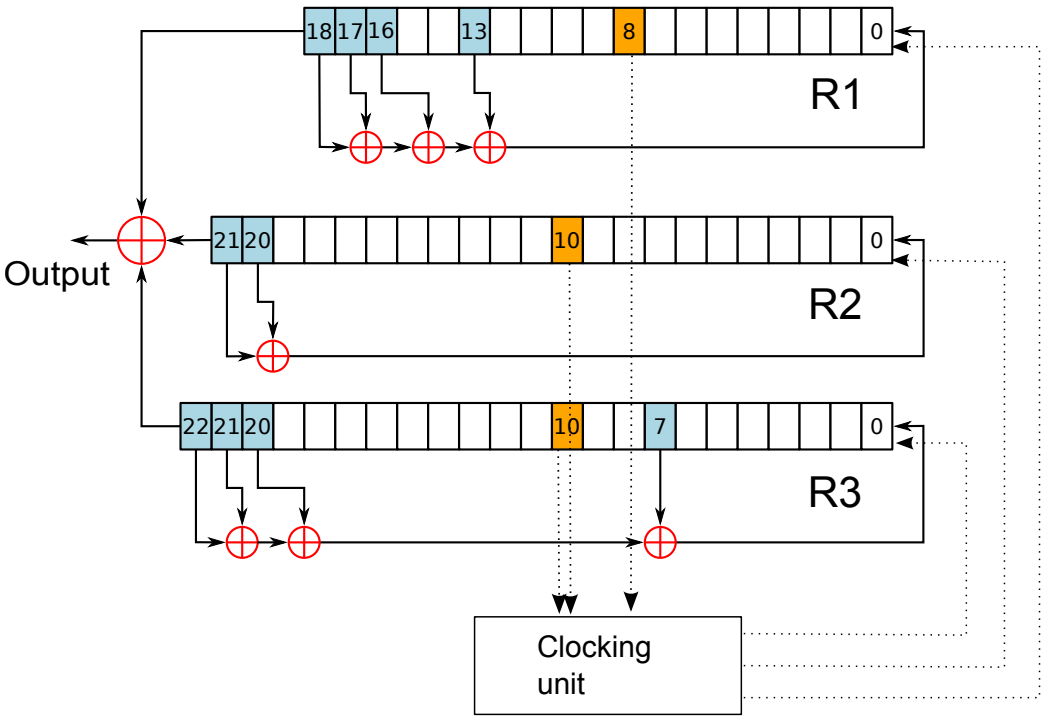

Figure 2.8: A5/1 Cipher

2.6.2. The A5/2 algorithm

The A5/2 stream cipher is a weak algorithm compared to A5/1, and its use in mobile phones has been forbidden in the current versions of the standards, so we won't talk about it. For a description of the algorithm and some of its vulnerabilities the reader can refer to [7], [9].

2.6.3. The A5/3 algorithm

The A5/3 cipher uses the Kasumi algorithm, which is one of the algorithms used in the UMTS network. Kasumi is publicly available, and is specified in [24]. Kasumi is a block cipher, which produces a 64 bit ciphertext from a 64 bit plaintext using a 128 bit key. To generate a 228 bit keystream from this block cipher in GSM, Kasumi is used in an output-feedback mode as a keystream generator.

2.7. Security considerations in GSM

Several shortcomings have been pointed out in GSM security, and have been solved in UMTS and LTE. Some of them are:

- There is no authentication of the network. The handset will establish communication with any cell that claims to belong to the operator.

- There is no explicit integrity protection for signalling, voice or data.
- Encryption is performed after error protection. This means that the plaintext of both signalling and voice encrypted communications has known redundancies, and as we will show in chapter 4 these redundancies allow to mount a ciphertext only attack against GSM encryption.
- The derivation of the session key is independent of the ciphering algorithm in use. This means a key obtained by breaking one algorithm can be used to break into a communication ciphered by a stronger algorithm.

The last item merits some explanation. The fact that the derived key is the same no matter which ciphering algorithm is in use allows attacks where the attacker forces the subscriber to cipher information with a weak algorithm which he can break, recovers the key and uses the key to request services from the network or to decipher the information protected with a stronger algorithm. A possible man in the middle attack can be carried out, where a rogue cell claims to belong to the operator, and offers the subscriber a weak ciphering algorithm to be used, while at the same time establishing a session with the network using a strong algorithm impersonating the victim. When the network asks for validation, the rogue cell just forwards RAND and SRES to the MS, and then forwards the response back to the network. When the victim uses the key derived from RAND and K_i, the attacker can recover the key by breaking the weak algorithm, and use that same key to cipher the rogue communication with the network.

This page intentionally left blank

Chapter 3

Known cryptographic attacks against A5/1

In this chapter we will compile and classify the known attacks against the A5/1 stream cipher.

The original GSM privacy algorithms and their design were kept secret by the GSM Association, but in 1994 the general structure of A5/1 was leaked and documented by Ross Anderson [2] [3] and others. This initial leak was cryptanalized by Golic [38] and Wagner.

In 1999 Briceno, Goldberg and Wagner reverse-engineered both A5/1 and A5/2 from real handsets [17], showing that the initial leaked structure was basically correct.

A5/2 was quickly shown to be weak [36], [9]. Let's remember A5/2 was designed that way, to be exported worldwide. There are also several proposed attacks against A5/1 which we will describe shortly, showing it is also a weak security solution. As for A5/3, the known cryptographic attacks are scant and show that the underlying Kasumi cipher can be broken in a scenario called "related key attack", an attack that is not realistic in the way A5/3 uses Kasumi.

3.1. Cryptoanalysis of A5/1

Attacks against A5/1 can be broadly classified in "guess and determine attacks", correlation attacks and time-memory tradeoff attacks. We document the known attacks of each class after this short introduction to each kind of attack, and explain how some of the attacks work as an aid to understand the different classes.

All attacks attempt to determine the internal state of A5/1 (the 64 bits of the 3 internal registers) just after the initialization step when key and frame number have been processed but before the 100 "mixing" steps. Once this state is known, the key can be efficiently calculated clocking A5/1 backwards or solving a set of linear equations.

In the "guess and determine" attacks, the idea is to guess the contents of some of the registers, and then calculate the rest of the bits solving a set of equations [38] or guess for a special condition in the cipherstream and calculate from there [12]. This kind of attack is a cryptographic break whenever the effort to verify if our guess is correct times the expected number of guesses until a correct answer is found is less than the expected computational effort of a brute force attack.

In the correlation attacks, the idea is to exploit the fact that the initialization of the internal state of A5/1 is a linear function of the unknown key and the known frame number, and observing the probability distribution of the advances of the three shift registers find correlations between the unknown key bits and the observed output of the algorithm.

In TMTO attacks, the idea is to exploit the fact that the internal state of A5/1 is only 64 bits in size, too short to ensure resistance to brute-force attacks. TMTO attacks split the calculation into an expensive precomputation phase, which can be reused for multiple attacks, and an online phase, which uses the previous calculation for an attack attempt faster than a brute-force attack.

3.1.1. Determining the key from A5/1 internal state

Most attacks on A5/1 aim to determine the internal state of A5/1, and then determine the session key from that internal state.

Let's call t the number of clock times that the A5/1 algorithm has advanced, $t = 0$ the time just after the key K_C and COUNT have been fed into the registers, and $S(t)$ the corresponding internal state after t clockings. The first output bit of A5/1 is taken for $t = 101$.

If we can find the internal state of A5/1 just after the value of the key and COUNT have been fed into the registers and before the 100 mixing steps, that is, at $t = 0$, it is trivial to invert the process to recover K_C knowing the value of COUNT, as the initialization is linear. The solution is detailed in appendix C. So the objective is to find the internal state just after K_C and COUNT have been fed into the LFSRs.

If our attack finds the internal state $S(t)$ of A5/1 for some $t > 0$ we want to find the internal state or states at $S(0)$ that lead to $S(t)$ after t clockings. Golic in [37] proposes a method for computing the initial state by recursive computation of the reverse state-transition function, which means that if we know $S(t)$, all the valid states $S(t-1)$ are calculated, and recursively from each state at clocking i the valid states at clocking $i-1$ are calculated until all $S(0)$ candidates are computed. For $0 < t < 101$ the output is not available, which means that the only validity criteria is that the clocking must be valid, while for $t >= 101$ the availability of the output depends on the kind of attack, if the output is known the calculated states must be consistent with the output and the clocking. Golic demonstrates that the complexity of this process is small in both cases. The worst case is when

the output is not available, and the time complexity is $O(n\sqrt{n})$.

3.1.2. Guess and determine attacks

Guess and determine attacks are known plaintext attacks, where the key is derived from some known plaintext in less than the average 2^{63} attempts needed for a brute-force attack.

The first published attack (against a leaked incomplete version of A5/1) was due to Ross Anderson in a Usenet post [2]. The idea of the attack is to guess the complete content of registers R1 and R2, and the first half (eleven bits) of R3; with this information, the clocking is known, and the other half of R3 can be computed from the output of A5/1 and the known plaintext. After that, the guess has to be checked with a trial encryption. Most of the time the 52 guessed bits will not be correct, so we expect to perform an average of 2^{51} or a maximum of 2^{52} attempts and verifications to find the correct internal state. If we assume the computation of the non-guessed bits of R3 requires a computation effort similar to an encryption, the expected computation effort is 2×2^{51} encryptions, or about a 2000-fold decrease in computation compared with a brute force attack.

A similar attack was described by Golic against a leaked outline of A5/1 [38], [37], in which the lower ten bits from each register are guessed. These bits determine the clocking until any of the registers' guessed bits advance beyond the clocking bit. At each clocking of A5/1 the attacker obtains a linear equation on some of the unknown bits. On average 14.33 equations are obtained, and adding the guessed 30 bits yields 44 linear equations [38] [12]. After that, instead of guessing enough bits to have a determined set of equations, Golic builds the valid options to the input bits to the clocking function by noticing that several combinations are not consistent with the A5/1 output. This reduces the number of trials to an average of $2^{41.16}$ to find the correct internal state. However, for a fair comparison to Anderson's attack we should notice that each step in Golic's attack is more complex than in Anderson's attack, as it involves the solution of a linear set of equations plus a trial encryption.

Some hardware assisted attacks based on the same ideas (with improvements) were presented in the work by Keller and Seitz [46], Pornin and Stern, [55], and Gendrullis, Novotný and Rupp [34]

Another attack due to Biham and Dunkelman [12], consists in assuming that a certain event happens, namely that for 10 rounds register R3 is not clocked. If this happens, then R1 and R2 are necessarily clocked, and we get information from the corresponding clock controlling bits from R1 and R2 (namely that they are the complement of the corresponding R3 clocking bit). This diminishes the necessary expected running time of the attack to 2^{27} A5/1 clockings, assuming one knows where in the cipherstream the event happens. As the position of the event is not known, one has to try on average 2^{20} starting locations (assuming all clocking combinations are equally likely), giving a total time of 2^{47} operations. They then give some optimizations which reduce the complexity to $2^{39,91}$ operations by using

a precalculated table which occupies some 64 GB. The main drawback of this attack is that it requires on the order of 2^{20} known plaintext bits to be effective.

3.1.3. Correlation attacks

Correlation attacks were first applied to A5/1 by Ekdahl and Johansson [19], [20]. We explain here how the attack by Ekdahl and Johansson works as an example of the method, and then refer to the respective papers for improved attacks. All the published correlation attacks are known plaintext attacks.

For this kind of attack, observe that A5/1 initialization is a linear function of the unknown session key $K_C = (k_1, \cdots, k_{64})$ and the known frame number $FN = (f_1, \cdots f_{22})$. The contents of each register after t clockings is also a linear function of K_C and FN. Using Ekdahl's notation, we can write the output bit from R1 as $u_t^1 = \sum_{i=1}^{64} c_{it}^1 k_i + \sum_{i=1}^{22} d_{it}^1 f_i$, where c_{it} and d_{it} are known constants. We can write a similar equation for $R2$ and $R3$. Ekdahl and Johansson noticed that $s_t^1 = \sum_{i=1}^{64} c_{it}^1 k_i$ is an unknown sequence which is the same for all frames encrypted with the same key, while $f_t^1 = \sum_{i=1}^{22} d_{it}^1 f_i$ is a known sequence different for each frame.

Let $z_1, \cdots, z_{228}$ be the observed output of A5/1. As the registers $R1$, $R2$ and $R3$ are irregularly clocked, we know $z_1 = u_i^1 \oplus u_j^2 \oplus u_k^3$, where i, j and k are the times each register has been clocked. Then, we can write

$$s_i^1 \oplus s_j^2 \oplus s_k^3 = z_1 \oplus f_i^1 \oplus f_j^2 \oplus f_k^3 \tag{3.1}$$

The probability that a register is clocked at any given step is 3/4, so after the 101 steps from initialization until the first output bit emerges we can expect each register to be clocked close to 76 times. As a first step assume the three registers are clocked exactly 76 times. Then the following equation holds:

$$s_{76}^1 \oplus s_{76}^2 \oplus s_{76}^3 = z_1 \oplus f_{76}^1 \oplus f_{76}^2 \oplus f_{76}^3 \tag{3.2}$$

Let's call the right hand side of this equation $O_{(76,76,76,1)} = z_1 \oplus f_{76}^1 \oplus f_{76}^2 \oplus f_{76}^3$, which is composed of known quantities. If the registers were indeed clocked 76 times then it holds that $s_i^1 \oplus s_j^2 \oplus s_k^3 = O_{(76,76,76,1)}$, and if not we can expect the previous equation to hold with probability 1/2.

If the probability that the three registers are clocked exactly three times is P, this equation holds with probability $1/2+1/2P$. This gives us a correlation between the observed z_1 and the sum $s_{76}^1 \oplus s_{76}^2 \oplus s_{76}^3$. Ekdahl and Johansson estimate P to be about 10^{-3}, so $P(s_{76}^1 \oplus s_{76}^2 \oplus s_{76}^3 = O_{(76,76,76,1)}) = 1/2 + 1/2 \times 10^{-3}$. Since $s_{76}^1 \oplus s_{76}^2 \oplus s_{76}^3$ is constant over all frames, by averaging $O_{(76,76,76,1)}$ among enough frames we expect to detect a deviation large enough to determine the sum with a high enough confidence (Ekdahl and Johansson only talk about "a few million frames" in [19]). We thus get a bit of information in the form of a linear equation on the bits of K. We can consider other assumed triples for the clockings of the three LFSR and get enough equations to recover the key.

The attack can be refined by noting that a clocking (i, j, k) may end up in other positions $z_2 \cdots z_{228}$ with varying probabilities. We can then use all positions where

there is a non-negligible probability of occurrence of clocking (i, j, k) to calculate the correlation probability. To check the correctness of the proposed attack Ekdahl and Johansson provide the result of simulations which show that choosing the right parameters the attack is successful more than 70% of the attempts if the stream output of about 2^{16} bits is known, using a few minutes of processing on a standard (for 2003) personal computer.

The attack by Ekdahl and Johansson was improved by Maximov, Johansson and Babbage [51] by statistical analysis of multiple frames and by considering d consecutive estimators as a $d-$dimension estimator, and by Barkan and Biham [8] by using conditional estimators and three weaknesses they observe in the choice of register R2. The best attack takes a few minutes to find the key with a success rate above 90% given 2000 known frames.

3.1.4. Time Memory Data Tradeoff Attacks

Introduction to TMTO attacks

This short summary is intended to facilitate the reading of this section, we will study time memory tradeoff (TMTO) attacks in chapter 5.

TMTO attacks, introduced by Hellman in 1980 [39], are a kind of brute force attack used to invert a function (that is, given the output of the function, find its preimage).

TMTO attacks consists of two distinct phases. The bulk of the computation for the attack is done in a usually costly precomputation phase which calculates the output of the function for an important fraction of the domain, generating one or several tables which resume that information. This precomputation may be costlier than a brute force attack, but will be carried out only once in preparation for the attack. Each entry in the tables stores the initial and ending point of a sequence of encryptions. As we will see in chapter 5, the main difference between TMTO attacks is in the way the sequences are calculated.

In the second phase of an attack using a TMTO the precomputed tables are used to speed up the attack. Given the captured output of the function to invert, the attack or on-line phase will consist in one or several searches in the tables, and the reconstruction of some of the sequences, with the aim of finding an encryption whose output is the captured text. There is no guarantee that the searched value is in the table. In this sense TMTO attacks are probabilistic attacks.

This kind of attack is called Time Memory tradeoff because the amount of memory used to store the precomputation tables and the time (effort) needed in the on-line phase are inversely related, the attacker can decrease the attack effort by increasing the memory devoted to store the tables. As a gross approximation, in many practical algorithms $M^2 \times T \sim N^2$, where M is the amount of memory, T is the on-line time or effort, and N is the co-domain space of the function to invert. There are other parameters to choose which depend on the TMTO algorithm which we will see in chapter 5, and the choice of these parameters has a

profound impact on the performance of the attack.

TMTOs are useful when the attacker expects to do several similar attacks, thus amortizing the precomputation over many attempts, or when he has time to prepare for an attack that should be carried out faster than a brute force attack.

Time Memory Data Tradeoffs (TMDTOs) are a class of TMTO used when more than one target is available for inversion, and inverting the function for any of the available targets is enough to consider the attack successful. For example a TMDTO may enable an attacker to find the key used to cipher several captured messages by inverting the encryption of any one of the messages. As we will see in chapter 5 having several targets available for inversion has the practical effect of decreasing the needed memory and/or time necessary for the attack.

To apply a TMTO to crack A5/1 the attacker needs to find a function whose inversion leads to finding the key or the internal state of A5/1 and whose output can be obtained from the captured output of the function for some conversation. Besides, the domain of the choice function should be of a tractable size, comparable to the size of the internal state of A5/1.

Attacks to A5/1 based on TMDTOs

The first TMDTO attack against A5/1 was proposed by Golic in [38] (similar to a generic attack against stream ciphers described by Babbage [6]) . His attack is a known plaintext attack, where he assumes the keystream output of A5/1 corresponding to several messages of the same conversation can be captured. The function he proposes to invert is the one that takes as input the initial state of A5/1 and whose output is the 64 first bits of the stream cipher output produced by A5/1. He proposes building a time memory tradeoff by building a table consisting of M output blocks and the (possibly multiple) 64 bit initial states reachable from the state at $t = 101$ that generates M. Then for a conversation where K keystream sequences are captured, each sequence 228 bits long, there are 102 64-bit blocks which can be searched in the table to find the corresponding preimages. By the birthday paradox we expect to find a collision with high probability if $102 \times K \times M > 2^{64}$. However, to be effective this tradeoff needs many captured keystream sequences from the same conversation, which implies many known plaintext frames, which are not usually available to the attacker.

In [15] Biryukov, Shamir and Wagner improve on Golic's results by showing that it is easy to generate all the states that produce output sequences with a particular k-bit pattern *alpha* with $k \leq 16$ (they call this property "low sampling resistance"), without trying and discarding other states, and propose further optimizing storage by storing only pairs (output, initial state) that have a high number of preimages, and Birshukov and Shamir [14] further study the use of TMDTOs for ciphers with low sampling resistance. The amount of known plaintext bits required for those attacks, on the order of 25000, make these attacks impractical in GSM.

In [35] and [57], Andy Rupp, Tim Güneysu and others report on the implementation of an hardware assisted A5/1 TMDTO using a custom cluster of Field Programmable Gate Arrays (FPGAs), for which no tables were publicly released. Their attack is a known plaintext attack, using a variant of TMDTO known as thin rainbow tables that will be studied in section 5.4.1. Their main contribution is implementing the compute intensive parts of the attack in hardware. The function to invert in the TMDTO is the function that takes as imput the initial state of the internal registers of A5/1 and has ouput the first 64 bits of the cipherstream produced by A5/1. Their FPGA implementation calculates 2^{36} A5/1 encryptions per second, which was a very high speed in 2008. They claim their TMTO can crack A5/1 from a single 64 bit output in an average of 7 hours.

In [43] another group also claims to having created the required precomputation tables for a TMTO using FPGAs, but the tables were not publicly released.

A third group, led by german cryptographer Karsten Nohl, set to calculate the tables for a TMDTO also using thin rainbow tables. This is also a known plaintext attack. The function to invert takes as input 64 bits of internal state, and outputs 64 bits of cipherstream. The main contributions of Nohl and hist team were to implement the algorithm to build the TMDTO tables in Graphics Processing Unit (GPU) cards, initially CUDA cards from Nvidia, and later OpenCL cards from ATI (now AMD), and finding several optimizations to decrease the search space. The tables were released in 2010, and occupy nearly 2 TB of hard disk. The associated cracking code is reported to be able to crack most keys in a few seconds provided a fast GPU is available and SSD disks are used to hold the tables. Another contribution was documenting several ciphered signalling messages in GSM with known content which can be used as source of known plaintext. Nohl et al found that only 14% of the state space is reachable after 100 A5/1 clockings. This means only reachable states need to be considered for the tables, decreasing the search space to approximately $2^{64} \times 0.14 \approx 2^{61.16}$ states. By using two messages with known content during call setup, which translates into 8 known plaintext bursts, they get $D = 408$ messages to search on the tables, which decreases the necessary table coverage.

A different TMDTO attack was proposed by Barkan, Biham and Keller in [9] and [7]. They propose a ciphertext only attack exploiting the fact that the redundancy needed for error detection and correction is added before encryption. This adds known redundancies to the plaintext which can be exploited to build a TMDTO. We will study this attack in detail in the following chapter.

Countermeasures against plaintext attacks

The attach against A5/1 which has received the most public attention is the one by Nohl et all, both because it is effective, only requiring the knowledge of two messages likely to appear in every conversation, and because of the public nature of the implementation. However as we saw earlier there were at least two other similar attacks published, and there are surely other implementations of these or

other ideas in products designed for the intelligence community [1]. All of these require the knowledge of the plaintext of a frame, and are possible because some control messages at the beginning of each voice call contain known information and when the information is not enough to fill a message a fixed known padding is used.

Recent versions of the standards for GSM from ETSI have added randomization of the padding bits used when a message is too small to fill a frame, and also randomization of the system messages to avoid known information being ciphered with A5/1. This reduces the attack surface for passive known plaintext attacks, as less known plaintext messages are available for an attack.

3.2. Outline for the rest of our work

Most of the attacks in the previous section require either large amounts of known plaintext, which are not available to an attacker when attacking A5/1 as used in GSM, or require huge computational resources. From the published attacks only the TMDTO known plaintext attacks are practical in the sense of not requiring unrealistically large resources (known plaintext messages, storage and computation). As recent versions of the standards for GSM added countermeasures against known plaintext attacks we decided to delve deeper into the ciphertext only attack proposed by Barkan, Biham and Keller, with the aim of exploring the feasibility of this attack when using modern hardware and the best known TMDTO attacks at the time this work was written.

In the following chapters we will first explain the ciphertext only attack proposed by Barkan et al, detailing how to calculate the function we need to invert for a successful attack, and making an extension to Barkan et al's work by finding another source of redundancy that can be used for the ciphertext only attack. Then we will study the state of the art on TMDTO attacks, applying the best known attack to a simplified problem. Finally we will estimate the resources needed to implement a full ciphertext only attack, and implement a demonstration attack under the assuption that a huge amount of captured ciphertext is available.

[1] for examples of products claiming to crack A5/1 see
`http://www.cellularintercept.com/ecom-prodshow/gsm_intercept.html`,
`http://www.pki-electronic.com/products/interception-and-monitoring-systems/passive-gsm-monitoring-system-for-a5-1-a-5-2-a5-0-encryption/` and
`http://www.shoghicom.com/passive-gsm-monitoring.php`

Chapter 4

Two ciphertext-only attacks against A5/1

In this chapter we study ciphertext only attacks against A5/1 when used in GSM.

In GSM error detection and correction algorithms are applied to the messages before ciphering, that is, whenever a new signalling message or voice frame is ready for transmission, first the coding and interleaving presented in section 2.4 are applied, and then the resulting 114 bit blocks are ciphered. This is not the recommended order, as the error detection and correction overhead adds a known redundancy in the plaintext which gives an attacker information that could be used to mount a ciphertext only attack. In newer protocols like UMTS and LTE, ciphering is applied before error detection and correction closing this attack vector. However for GSM we will see this decision creates the scenario for a realistic attack.

In the first section of this chapter we review the results of Barkan, Biham and Keller in [9] and [7], where they present a ciphertext only attack against A5/1 using the redundancy due to the error detection and correction algorithms. Their attack uses the redundancy in the SACCH control channel to define a function that we will call h_c that, after being inverted, yields the internal state of A5/1. They then propose using a TMDTO attack to invert the function. We also show the details of the calculation of function h_c. We explain the attack and document the construction of all elements necessary for the implementation of the attack.

In the second section we propose a new ciphertext only attack against A5/1 that seems to have never appeared in the literature. The new attack is also a TMDTO attack, based on the redundancy of the error detection and correction codes on the TCH/FS voice channel. We will build a function h_v that, once inverted, yields the A5/1 state thus enabling the calculation of key K_c.

In the last section of this chapter we compare both attacks.

As we saw in the previous chapter, to be able to mount a TMDTO attack we need to find a function h whose output can be calculated from the captured data, and whose input leads to an attack on the cryptosystem.

As for the input, we saw in section 3.1.1 that finding the internal state of A5/1 just after the value of the key and COUNT have been fed into the registers and before the 100 mixing steps allows the attacker to find the shared key for the conversation, while finding the internal state $S(t)$ after A5/1 has advanced t steps enables the attacker to find all initial states at $t = 0$, which leads to finding all key candidates that can produce $S(t)$. In the later case the candidate keys must be checked using another piece of captured data. The aim is thus to find a function of the internal state of A5/1 whose output value can be calculated from the observed ciphertext output.

As we will see shortly the output of functions h_c and h_v will depend on the captured ciphertext for some bursts. In the case of the SACCH channel we will need to capture pairs of bursts corresponding to the first and third bursts of a SACCH message, whose fame numbers are such that the value of COUNT differs only in the least significant bit of T_3. We will see this condition is pretty common, being met on average once a second. In the case of the TCH/FS channel we will need to capture six consecutive bursts starting at the third burst of multiframes that start at certain positions in their corresponding superframe.

In both cases the function will rely on the error detection and correction redundancy in the corresponding channel, which means that to carry out the attacks the captured frames must have no errors.

We will not concern ourselves with the difficulties associated with capturing the necessary ciphered traffic, instead assuming the necessary ciphertext is available. For some previous work on how the traffic could be captured see for example [45].

4.1. The results of Barkan, Biham and Keller

In [7] the authors present their results in the cryptanalysis of A5/2 and A5/1, and also some active attacks against GSM where they exploit the fact that the same key is used irrespective of the algorithm in use. We are mostly concerned about their results attacking A5/1, namely a passive ciphertext-only attack which can be used with little knowledge of the messages being exchanged. They concentrate on the error detection and correction codes for the SACCH channel which we saw in section 2.4 (they also show an attack on the downlink SDCCH/8 channel, but to be effective this attack requires that the messages are padded with known bits, which should not be true in recent GSM releases that mandate padding bits to be randomly chosen). We will only describe their attack on the SACCH channel.

4.1.1. Description of the attack

In the SACCH channel each message has a fixed length of 184 bits. Before being encrypted and transmitted a cyclic code and a convolutional code are applied, obtaining a 456 bit block M which is then interleaved and divided into four 114 bit frames, which are independently ciphered and transmitted in four bursts. The details of the codes used is shown in section 2.4.

Let's represent a SACCH message as a 184 bit vector P. Being linear operations, the coding operation and interleaving of a message can be modelled as the multiplication over $GF(2)$ by a constant 456×184 matrix G and XOR with a constant vector g. The result of the coding and interleaving operation is $M = (G \cdot P) \oplus g$. A procedure to calculate matrix G and vector g is shown in section 4.1.2. After this operation M is split into four equally sized data frames, XORed with the keystream from A5/1 with the corresponding frame numbers, and transmitted. As G is a 456×184 matrix, there are $456 - 184 = 272$ equations which describe the kernel of the transformation. Being an error detection and correction transformation, the dimension of the kernel is exactly 272 due to the fact that the codomain of the function is of size 184 bits. Let H be the matrix which describes those 272 equations, that is, the parity check matrix such that $H \cdot (M \oplus g) = 0$. The key observation in the paper is that given a ciphertext it is possible to find linear equations on the keystream bits using the parity check matrix.

To calculate the corresponding ciphertext for a message M, four A5/1 keystreams k_1, k_2, k_3, k_4 are generated using the same key and the FN corresponding to the timeslot in which each frame will be transmitted. Let $k = k_1 \parallel k_2 \parallel k_3 \parallel k_4$ (where $\parallel$ denotes concatenation), then $C = M \oplus k$ is the corresponding ciphertext. We can apply the same H matrix to $C \oplus g$, and substitute C:

$$H \cdot (C \oplus g) = H \cdot (M \oplus k \oplus g) = H \cdot k \tag{4.1}$$

Having the captured ciphertext C for the four frames corresponding to any message M means we have a linear equation system over the bits of the corresponding k. Note that the equations are independent of P, they only depend on k, the known C and the fixed value of g.

We want to build a function that maps the internal state of A5/1 to a value derived from equation (4.1), however we have four keystreams $k_1 \cdots k_4$ that depend on different initial states $S_1 \cdots S_4$ derived from the same key K_C but different COUNT values derived from the corresponding FN values.

When associated with a TCH/FS channel, that is, when associated with a voice call, the four frames which comprise a SACCH message are carried in the same frame offset on four consecutive 26-multiframes as we saw in section 2.3.1. This means that given the initial value of COUNT, it is easy to calculate the remaining three values of COUNT. So given the internal state of A5/1 after key setup for the first frame of the message and the corresponding COUNT values we can calculate $S_2 \cdots S_4$ knowing S_1. The calculations are shown in appendix B.

Let $h(x) : \{0,1\}^{64} \to \{0,1\}^{64}$ be the function that maps the state of A5/1 after key setup in the first of the four frames, to the first 64 bits of the result of $H \cdot k$. To make it clearer, given the internal state of A5/1 after key setup for the first frame of the message (call it x), calculate the corresponding internal states after key setup for the other three frames of the message, and then advance A5/1 for each of the calculated internal states obtaining $k_1 \cdots k_4$. Then concatenate $k = k_1 \parallel k_2 \parallel k_3 \parallel k_4$ and calculate $H \cdot (C \oplus g)$, keeping only the first 64 bits

of the result vector as output of function $h(x)$. If we are able to invert function $h(x)$ then we can find K_C knowing FN, however we expect the inversion of h to be computationally intensive, as it includes the inversion of A5/1. So Barkan et al propose to treat $h(x)$ as a random function and use a time memory data tradeoff to invert it. Once the internal state of A5/1 after key setup is known, the key can be found by inverting the linear initialization as shown in section 3.1.1.

In time memory data tradeoffs, as we saw in section 3.1.4, the attack is divided in two phases, an off-line phase where the output of the function to invert is calculated and resumed in tables for a significant portion of the domain, and an online phase which uses the data calculated in the off-line phase. One technical issue in this case is that the function h(x) depends on the difference in the value of COUNT of four frames, and each set of tables can only be calculated for a fixed set of differences. This means that either several sets of tables for different values of the differences have to be built, or the attack has to be carried out using only those messages whose COUNT differences are represented in the tables, which lowers the attack success probability. To counteract this, Barkan et al. found a method that uses only two of the four frames, thus loosening the restrictions on the COUNT differences.

Let's first observe the difference in the COUNT values on the first and third frames. Both frames will be sent on the same frame offset on their corresponding 26-multiframe, so $T2 = FN \, mod \, 26$ is the same for both frames. $T3 = FN \, mod \, 51$ is increased by one modulo 51 (from the first to the third frames FN increases $26 \cdot 2 = 52$, which is equal to $1 \, mod \, 51$). When the value of $T3$ for the first frame is even, which occurs in half the cases, $T3$ only changes in it's least significant bit. If we assume that $T1$ (the FN divided by $26 \cdot 51 = 1326$) does not change, then only one bit of COUNT changes from the first to the third frame (let's remember that COUNT is the concatenation of $T1T3T2$). These conditions are met on average once a second, so if we can find a function that depends only on the first and third frames we get a new data point to attempt an inversion every second.

We want to use Gaussian elimination in equation (4.1) to find equations that only depend on the values of k_0 and k_2. However there are not enough equations (we need at least 64 independent equations). Barkan et al claim that each SACCH frame has 20 bits fixed by the protocol, so adding equations that represent these fixed bits we can augment H to a new 292×456 matrix H'. Then the order of the bits of k is changed so that $k' = k_1 \parallel k_3 \parallel k_0 \parallel k_2$ and make the corresponding changes in H''s columns so that the product remains the same, getting H'', and also change the order of the bits in C and g getting C' and g' respectively. Applying Gaussian elimination to the system $H'' \times k' = H'' \cdot (C' \oplus g')$, we can eliminate the coefficients corresponding to k_1 and k_3 in all rows except the first 228, so we have 64 rows (rows 229 - 292) that only have non-zero values in the columns corresponding to bits of k_0 and k_2. Let's define H_C as the sub-matrix formed by rows 229 - 292 and columns 229-456 of H'', $k_C = k_0 \parallel k_2$, C_C the cyphertext corresponding to the first and third burst, and g_c the corresponding bits from g'.

Just as in the previous case k_C is a function of the initial state of A5/1 for the first frame, k_0. Using this 64×228 matrix H_C we define h_c in a similar way to the way H defines h, that is, h_c is a function $h_c(x) : \{0,1\}^{64} \rightarrow \{0,1\}^{64}$ that maps the internal state of A5/1 in the first frame to the 64 bits of the product $H_C \cdot k_C(x)$.

For function h_c to be useful to define a TMDTO it must be possible to calculate h_c from the ciphertext, which is the case as h(x) can be calculated from the captured ciphertext as shown in equation (4.2).

$$h(x) = H_C \cdot k_C(x) = H_C \cdot (C_C \oplus g_C) \tag{4.2}$$

4.1.2. Practical details of the attack

There are some details to be completed before this attack can be implemented.

First we need to find 20 bits with known values in the SACCH channel messages. Appendix A describes the format of the messages in each layer of the SACCH channel, and finds several bits with fixed values which can be used, 30 in the downlink direction and 32 in the uplink. As we have more bits than needed we can keep the bits which are fixed for more messages and seem less prone to be changed in future versions of the standards. Even then we have more bits than needed, so we just drop the extra bits.

Matrix H and vector g have to be built. As we saw in section 2.4, all the operations in the channel coding are linear, so we can easily build a matrix G representing the whole coding process as the multiplication of the matrices corresponding to each operation. We use the notation of section 2.4.

The first step in the coding for the SACCH 184-bit messages ($P = d_0 \cdots d_{183}$) is to apply a fire code with generator polynomial $g(D) = (D^{23}+1)(D^{17}+D^3+1)$, obtaining a 40 bit parity vector $Par = p_0 \cdots p_{39}$ which is appended to the message. Thus $Par = G_f \cdot P$, where G_f is a 40×184 matrix. An easy way to build G_f is column by column, where column j is the vector corresponding to the result of applying the fire code to the message P_j which has a binary one in position j and zero in the rest of its elements. To account for the fact that the fire code is calculated so that the remainder is a 40 bit vector of all ones, we add a vector $g_f = (1, 1, \cdots, 1)$.
As we want the original bits of the message conatenated with the parity bits, we can just build a 224×184 matrix G_1, where rows $0 \cdots 183$ represent the identity matrix, and rows $184 \cdots 223$ are rows $0 \cdots 39$ from G_f. Vector g_f is concatenated to a 184 bit vector of all zeros obtaining g_1, so the output of this stage is $G_1 \cdot P + g_1$.
For the second step, adding the tail bits, it is enough to add 4 zero rows to matrix G_1 and four zero elements to vector g_1.
To represent the convolutional coding as a matrix G_c, we can use the same method as that to calculate G_f, that is, build the matrix column by column, where each column is the vector corresponding to the result of applying the convolutional coding to the vector that has a single one bit in the position corresponding to the

column number.
Interleaving can be represented as the square 456×456 permutation matrix G_i where each column has a single one bit in the position matching the input bit to the corresponding output bit.
Joining it all together, to go from a 184 bit message P in the SACCH channel to it's coding we do $M = G_i \cdot G_c \cdot (G_1 \cdot P + g_1) = G \cdot P \oplus g$, where $G = G_i \cdot G_c \cdot G_1$ and $g = G_i \cdot G_c \cdot g_1$

Matrix G, vector g, all intermediate matrices, and the kernel of G, H, were calculated using the NTL C++ library [21] and verified using publicly available gsm captures.

One possible way to add the information about the bits with known values is to check if the fixed bits in P translate into fixed bits in M. This is the case, as there are 33 bits in M that only depend on the value of the fixed bits in P. So we can add 33 equations $M_i \oplus f_i = 0$, where f_i is the known value of M_i in position i. We don't expect all equations to be independent as they come from 30 fixed bits. For each equation we add a row to matrix H' which has a single one value in position i. We also generate a vector f, which has a zero in the first 272 positions, and the value f_i in each added row. The new equation system we get is $H'(M \oplus g) \oplus f = 0$, so $H'(C \oplus g) = H'(M \oplus k \oplus g) = f \oplus H'k$, which translates to $H'k = f \oplus H'(C \oplus g)$.
After this we swap the columns of H' as explained in the previous section, and use Gaussian elimination as proposed by Barkan, Biham and Keller to eliminate the coefficients corresponding to k_1 and k_3 from all rows except the first 228. When calculating this step it was found that the range of H' is only 297 even though we added 33 equations, which means not all equations were independent of the previous ones. As we only needed to add 20 equations this is not a problem. Finally we take the sub-matrix consisting of rows 229 - 292 and columns 229-456 of H', taking care to apply the corresponding operations to f.

The last practical detail is to verify when the assumption that $T3$ is even in the first frame holds. For this we check in ETSI's TS 45.002 standard, chapters 6 and 7 [31], how the initial burst for a SACCH message is chosen. According to TS 45.002 the initial frame number for the SACCH messages associated with a conversation depends on the timeslot TS of the conversation, to spread the messages in time and thus lower the peak processing necessary in the BTS. For example for the SACCH channel in $TS = 0$, the initial burst of each message happens when $frame\%104 = 12$ while for $TS = 1$ the initial burst happens when $frame\%104 = 25$. The rule is that the initial burst happens when $frame\%104 = 12 + 13 * TS$.

SACCH messages occupy four burst, each in a different 26-multiframe, so each message starts 104 frames after the previous one. As $104\%51 = 2$, $T3 = FN\%51$ is even for 26 consecutive messages, and then odd for 25 messages. When $T3 = 50$ for the first burst, it will be zero for the third, so this message pair does not have

the expected difference, so it means we have 25 useful messages out of each 51 consecutive messages. As SACCH messages start each 104 frames, the time between SACCH messages is approximately 480 ms, so we have alternating periods of messages with $T3$ even and $T3$ odd every 12 seconds approximately. This means that in the worst case, a conversation that lasts less than 12 seconds could have no frames with a COUNT difference adequate for the tables built using function h_c.

If the attacker aims for a high success rate for short conversations, a second set of tables can be built taking a different matrix H", where instead of eliminating the rows affecting k_1 and k_3, we change the order of the bits so that after applying Gaussian elimination we get a 64×228 matrix applied to k_1 and k_3, eliminating the rows affecting k_0 and k_2. In this case, using both sets of tables, the longest period without a useful SACCH message is 6 seconds.

To implement the calculation we also need the difference in the internal state of A5/1 after feeding the key and the value of COUNT between the first and the third frame when only $T3$ changes in its Least Significant Bit (LSB). The calculation can be found in Appendix B, in summary the bits that change its value are:

- For R1, bits 2 and 16.
- For R2, bit 16.
- For R3, bits 0, 8 and 16.

4.2. A new ciphertext only attack based on the redundancy in the Voice channel

In this section we propose a new ciphertext only attack against A5/1 that seems to have never appeared in the literature. This attack is based on the same ideas as the attack by Barkan et al, but using the redundancy in the voice channel instead of the redundancy on the SACCH channel.

As we saw in section 2.4, each voice frame is 260 bits long, but only the first 182 bits (called Class 1 bits) are protected by a cyclic redundancy code followed by a convolutional coder. We can attempt to mount an attack using this redundancy. An added difficulty in the case of the voice channel is that diagonal interleaving is used, which means each 114 bit burst depends on two different voice frames. We are interested in finding a set of bursts with enough redundancy to have 64 independent equations, whose count differences repeat the most so that we can reach a matrix H and corresponding function h using the same procedures as in the SACCH channel.

Each 260 bit voice frame affects eight consecutive 114 bit bursts, the even bits for the first four bursts and the odd bits for the last 4. Each 4 burst block depends

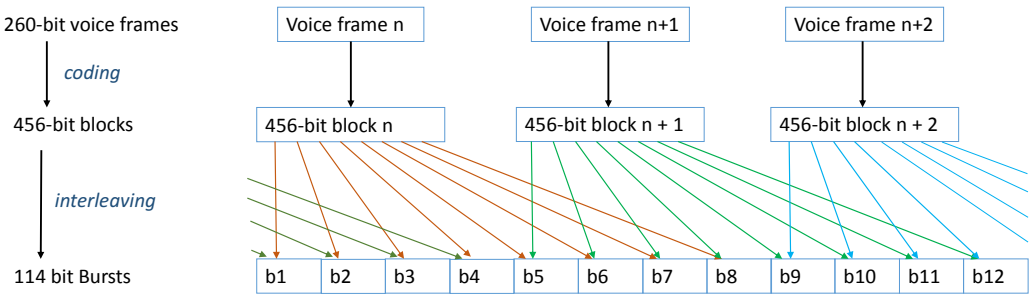

Figure 4.1: Coding and interleaving in the voice channel

on the bits of two consecutive 260 bit messages. This is schematically represented in figure 4.1.

Following the ideas in the previous section, we want to build a function of the internal state of A5/1 at some point in time, which can also be calculated from the known redundancy in the ciphertext.

Let's call P_L the concatenation of L voice frames. Just as in section 4.1.2 we can build a matrix G_L and vector g_L such that $M_L = G_L \cdot P_L \oplus g_L$ is the output of applying the cyclic redundancy code followed by the convolutional code and the diagonal interleaving to P_L. Observing the diagonal interleaving, we see that L voice frames generate $L-1$ four burst blocks, and half the bits for another two blocks of four bursts. For example if $L = 2$ then M_2 will be the concatenation of half the bits from the first four bursts, four complete bursts that only depend on the two voice frames in P_2, and half the bits from the following four bursts (referring to figure 4.1 as example, M_L is the concatenation of the even bits from $b1 \cdots b4$, all the bits from $b5 \cdots b8$, and the odd bits from $b9 \cdots b12$, while P_2 is the concatenation of voice frames n and $n+1$). To build matrix G_L and vector g_L we proceed just like in the previous section.

For $L = 1$ we have $M_1 = G_1 \cdot P_1 \oplus g_1$ with G_1 a 456×260 matrix. We expect the rank of G_1 to be 260, so the parity check matrix H_1 is a 196×456 matrix such that $H_1 \cdot (M_1 \oplus g_1) = 0$.

There are 8 bursts containing bits from M_1, $b_1 \cdots b_8$. Let's call C_1 the concatenation of the even bits from $b_1 \cdots b_4$ and the odd bits from $b_5 \cdots b_8$, and K_1 the concatenation of the corresponding bits from keystreams $k_1 \cdots k_8$ (that is, the concatenation of the even bits of $k_1 \cdots k_4$ and the odd bits of $k_5 \cdots k_8$). Then it follows that $C_1 = M_1 \cdot K_1$.

Applying the same reasoning as in section 4.1.1 we get equation (4.3). The left side of the equation can be easily calculated from the captured ciphertext, while the right side can be calculated from the initial states of A5/1 for the eight corresponding bursts.

$$H_1 \cdot (C_1 \oplus g_1) = H_1 \cdot (M_1 \oplus K_1 \oplus g_1) = H_1 \cdot K_1 \tag{4.3}$$

We want a step function that has a codomain of the same size as the internal

state of A5/1, namely 64 bits. We could just keep any 64 equations from the 194 available, however if we proceed as in section 4.1.1 and perform Gaussian elimination in equation (4.3), we can find a new set of equations that only depend on six of the eight keystream bursts (which lowers the necessary computation later on). Any of the burst could be eliminated, we choose to make 0 the coefficients for bits of b_1 and b_2. Let H'_1 be the resulting matrix after Gaussian elimination. Only the first 114 rows of H'_1 have non-zero coefficients for the columns corresponding to b_1 and b_2. Let's call H_v the sub-matrix consisting of rows 131-194 and columns 115-456 from H'_1, C_v and g_v the corresponding vectors consisting of elements 115-456 from C_1 and g_1, and K_v the concatenation of the even bits of k_3 and k_4 with the odd bits from $k_5 \cdots k_8$. We only keep rows 131-194 as we only need 64 equations. Then equation (4.4) holds:

$$H_v \cdot (C_v \oplus g_v) = H_v \cdot K_v \tag{4.4}$$

To calculate K_v we need the value of k_i for six different bursts, each one is a function of the initial state of A5/1 after initialization with the key and the corresponding value of COUNT. As we saw in section 4.1.1 and appendix B, given the initial A5/1 state for some key and COUNT value it is easy to calculate the initial state for any other COUNT value and the same key. We define $h_v(x) = H_v \cdot K_v(x)$, where x is the initial value of the internal state of A5/1 for the first burst. To be more explicit, given a 64 bit vector x, we take x to be the internal state of A5/1 for the first burst, and calculate the internal states $x_3 \cdots x_8$ corresponding to bursts $b_3 \cdots b_8$. Using $x_3 \cdots x_8$ we calculate $k_3(x_3) \cdots k_8(x_8)$, then K_v as the concatenation of the even bits of k_3 and k_4 with the odd bits from $k_5 \cdots k_8$, and finally calculate $h_v(x) = H_v \cdot K_v(x)$. $h_v(x)$ is the function we will try to invert using a TMDTO.

Just as in the case of the SACCH channel, to be able to calculate $H_v(x)$ we need to know a priori the XOR differences between $x_3 \cdots x_8$ and x. We can only calculate the TMDTO tables for a fixed set of XOR differences in the values of COUNT for the involved bursts.
Coded voice messages start in positions 0, 4, 8, 13, 17 and 21 of each 26-multiframe and occupy four consecutive bursts, so each 456 bit block does not span more than one 26-multiframe, but two such blocks may span two consecutive multiframes if the first one starts at position 21. We restrict ourselves to the case in which all bursts are in the same multiframe. The value of $T1$ is fixed in each 26-multiframe, and the value of $T2 = FN\%26$ is the same for bursts in the same position on different multiframes.

If we consider messages in the same multiframe within different superframes, messages with the same offset inside the multiframe will have the same XOR differences. For example the first eight bursts from each superframe will have the same relative differences, so tables built for that set of differences can be used at least once each superframe.

We exhaustively checked all possible combinations of multiframe offset within

the superframe and initial burst offset within the multiframe, and found that if we consider the first eight bursts of each multiframe, we find six multiframes in each superframe where the differences are identical (multiframes 0, 13, 16, 29, 32, 48). The same happens if we take the eight bursts starting on frame 4, 13 or 17 of each multiframe. This means we can build a set of tables for any one of those differences, or more than one set of tables for different starting burst.

Each superframe lasts 6.126 seconds, so using one set of differences we have on average almost one useful multiframe each second. The longest distance between two useful multiframes is 16 multiframes, which translates to around 1.9 seconds.

4.3. Initial comparison of the attacks

Comparing the necessary information to carry out the attack, both attacks can be carried out without knowing the plaintext messages. In the case of the attack against the SACCH channel we use the knowledge about the redundancy introduced by the error detection and correction, and also the fact that several bits have fixed values for the most common messages in the SACCH channel as we saw in Appendix A. This means we may see less useful messages than expected if other messages are sent during the conversation, like SMS. Attacking the voice channel only uses the knowledge about the error detection and correction for the voice signal, and can be carried out whenever there is voice transmission, that is, during the whole call except when silence suppression is in use.
Both methods provide approximately one message to attack per second, which means that the expected coverage of the TMDTO matrices, and thus their size, must be similar to have the same success probability.
Both methods are sensitive to errors in reception, as any bit received in error in the involved bursts makes the sample useless.

The main disadvantage of the attack using the voice channel is that the calculation of the function h is more expensive than in the case of the SACCH channel, as it includes the calculation of the output of A5/1 for at least six initial states. This means each iteration of h takes about three times as much as the corresponding function for the SACCH channel, affecting both the precalculation phase and the online phase.

As a summary, the best attack is using the SACCH channel, unless a high success rate is desired for very short calls (less than 6 seconds), in which case using the voice channel has an edge as attacks using the SACCH channel cannot guarantee there is a useful sample to search in the tables for such short calls.

Chapter 5

Time Memory Data Tradeoff Attacks

Time memory tradeoff (TMTO) attacks, introduced by Hellman in 1980 [39], are a family of techniques used to invert a cryptographic function, that is, given the output of a cryptographic function h, find a preimage for that output. Note that whenever h is not injective there can be several preimages. Depending on the application it may be enough for the attacker to find any preimage, or he may need to check for a particular preimage, leading to slightly different problems.

TMTO attacks consists of two distinct phases. The bulk of the computation for the attack is done in a usually costly precomputation phase which calculates the output of the function for an important fraction of the domain, generating one or several tables which resume that information. Then, in each attack attempt (online phase) those tables are used to speed up the attack. TMTOs are useful when the attacker expects to do several similar attacks, thus amortizing the precomputation over many attempts, or when he has time to prepare for an attack that should be carried out faster than brute force.
Time Memory Data Tradeoffs (TMDTOs) are a class of TMTO used when more than one target is available for inversion, and inverting the function for any of the available targets is enough to consider the attack successful. For example a TMDTO may enable an attacker to find the key used to cipher several captured messages by inverting the encryption of any one of the messages.

In this chapter we will briefly describe and characterize the different types of tradeoff algorithms. We start with a description of the original work by Hellman in 1980, the classic Hellman TMTO tables [39]. Then we will see the improvement proposed by Rivest to lower the amount of disk lookups needed, called Distinguished Points, and later on a different tradeoff implementation called Rainbow Tables, proposed by Oechslin in 2003 [54]. We will follow with time memory data tradeoffs, which are used when we have more than one point to invert, studying the early proposals which consisted on adapting Hellman and Rainbow tables, and ending with the thin and fuzzy rainbow tables proposed by Barkan, Biham and Shamir in 2006 [10] [7]. We will see the historical approximate characterization of the different algorithms, following with a recent characterization and comparison by Hong and Kim [48] [47] for the single target case. In the following chapter we

propose an extension to the multi-target case to the calculations of Kim and Hong for the perfect fuzzy rainbow tables in the single target scenario.

In the first part of this chapter, we follow the summary introduced in [57] and [53].

5.1. Hellman's Time Memory Tradeoff

The first time memory tradeoff attack was described by Hellman [39] in the context of block ciphers, more precisely as a way to attack DES (Data Encription Standard), but his method can be used to invert other discrete finite one-way functions. Suppose we have an encryption function $E : \mathbb{P} \times \mathbb{K} \mapsto \mathbb{C}$, where $\mathbb{P}$ is the set of all possible plaintexts, $\mathbb{K}$ is the set of all possible keys, and $\mathbb{C}$ the set of all ciphertexts. Given $C \in \mathbb{C}, P \in \mathbb{P}, k \in \mathbb{K}$, we adopt the notation $C = E(P, k) = E_k(P)$. Hellman's attack is a known plaintext attack, the goal of the attacker is, given C and P, find k such that $C = E_k(P)$. This k can then be used to decrypt further messages sent by the user using the same key.
The brute force approach to finding k is to try all possible values of $k \in \mathbb{K}$ and keep the values of k such that $C = E_k(P)$. This guarantees finding the k used to encrypt P (and potentially some other values of k that give the same result), but has a high computational cost proportional to $N =\mid \mathbb{K} \mid$. If we can check the correctness of the candidate k or otherwise guarantee we are searching for a single possible value, the expected amount of trials needed to find k is $N/2$, making the time to finding the key $T = O(N)$. This makes this approach only applicable to ciphers with small key spaces, and implies an expensive process each time a new pair P, C must be attacked.
We may also consider another approach. If we know that the encryption of a certain plaintext block P_0 is likely to appear in the captured data, we can calculate a huge table with all the possible pairs (k_i, C_i), where $C_i = E_{k_i}(P_0)$. This table can be built in advance and reused for many attacks, and once built, the time for each attack is only that of a search in the table. The problem with this approach is that the required storage space is proportional to N, which is impractical except for very small key spaces.

Hellman proposed a method that lies between the two previous ones both in terms of required storage and in the time needed for the actual attack. In Hellman's method it is also necessary to know that the encryption of a certain plaintext block P_0 is likely to appear in the captured data. An expensive precomputation step will be carried out, in which more than N encryptions may need to be calculated, storing the results condensed in one or several tables which will help the attacker speed-up the attack later on. Those tables can be used whenever the chosen P_0 is likely to appear in a message, so the precomputation can be reused over many attacks on different users.

In Hellman's method a *reduction function*, $R : \mathbb{C} \mapsto \mathbb{K}$ is needed, which must be simple to calculate in the sense that its computation must be fast (for instance,

in the case of DES, where the key is 56 bits long and the ciphertext is 64 bits long, it may be as simple as dropping the last 8 bits). Some authors call R a *mask function*.

Let's define $f(k) = R(E_k(P_0))$. The main building block in the precomputation phase is a *chain*, where a starting point SP is chosen in the key space $\mathbb{K}$, and the function f is iterated a fixed number of times t starting from $S_0 = SP$, so that $S_i = f(S_{i-1}) = f^i(S_0)$ for $1 \leq i \leq t$. Only the initial point, SP, and the ending point, $EP = S_t$ will be kept and stored.

In the precomputation step, m starting points are chosen, $SP_1, \cdots \mathrm{SP}_m$, and for each SP_i one chain is built, storing only the pairs of starting and corresponding ending points (SP_i, EP_i) sorted by the ending point (Figure 5.1). This means m memory positions of adequate size to store the starting and ending points will be needed. The m chains summarize $m \times t$ encryptions with E, so the storage is reduced by a factor of $t/2$ compared to a table storing all the calculated pairs. However there may be repeated values between different chains in the table, which means the number of unique pairs preimage-image represented by the table is less than $m \times t$.

$$\begin{array}{c} SP_1 = S_{10} \xrightarrow{f} S_{11} \xrightarrow{f} \cdots \xrightarrow{f} S_{1t} = EP_1 \\ SP_2 = S_{20} \xrightarrow{f} S_{21} \xrightarrow{f} \cdots \xrightarrow{f} S_{2t} = EP_2 \\ \vdots \\ SP_m = S_{m0} \xrightarrow{f} S_{m1} \xrightarrow{f} \cdots \xrightarrow{f} S_{mt} = EP_m \end{array}$$

Figure 5.1: Hellman's table

Given the reduction function R and function $f(k) = R(E_k(P))$, we can summarize the calculation of a single table with the following pseudocode:

Listing 5.1: Calculation of a Hellman's table

```
typedef blockN  {0,1}^N                    //N-bit block
typedef Ntuple (blockN, blockN)            //tuple of two N-bit blocks

Process CreateHellmanTable
   inputs:
      integer t: chain length
      integer m: number of starting points
   output: list of Ntuples representing Hellman table (possibly stored on disk)
  var result as array [1..m] of Ntuple
  var StartingPoint, S as blockN
  var i, k as integer

  for (i = 1 to m)
       StartingPoint = random()
       S = StartingPoint
       for (k = 1 to t)
          S= f(S)
       result[i] = (S, StartingPoint)
```

```
sort(result)
store(result)
return(result)
```

In the online or attack phase, the attacker has a captured ciphertext C which corresponds to the encription of P_0 with an unknown key k_e, and wants to use the tables to find k_e. First he calculates $Y_0 = R(C) = f(k_e)$ and searches endpoint Y_0 in the last column of the table. If Y_0 is not found, then k_e is not in the next to last column $(t-1)$. If $EP_j = Y_0$ is found there is a candidate for the value of k_e in column $t-1$ of row j. To find the candidate the attacker must reconstruct the chain starting from SP_j and iterating f up until S_{t-1}.

Unless f is injective, Y_0 might have been reached from another key k_f which we will call a false positive or false alarm. To discard false positives we need to check the candidate key with another plaintext-ciphertext pair.

If the attacker does not find k_e in the previous step (either because he did not find Y_0 or he found one which leads to a false alarm), he calculates $Y_1 = f(Y_0)$ and checks whether Y_1 is in the last column. If $Y_1 = EP_j$, then he reconstructs the chain from SP_j up until S_{t-1} (and storing S_{t-2}, which is the candidate for k_e). First he checks if $S_{t-1} = Y_0$, if the equality holds S_{t-2} is his new candidate for k_e, and he must check this key with another plaintext-ciphertext pair to discard a false positive. If $S_{t-1} \neq Y_0$ then it is a false positive.

In the same manner the attacker calculates $Y_2 \cdots Y_{t-1}$ and verifies if they are an end-point. If the value is found in the list of endpoints the candidate in the corresponding column is checked.

Knowing function R and the step function f the online phase is represented in the following pseudocode. If we have more than one table, the search is repeated for each table.

Listing 5.2: Search in Hellman's table

```
typedef blockN  {0,1}^N                  //N-bit block
typedef Ntuple (blockN, blockN)          //tuple of two N-bit blocks

function HellmanSearch
    inputs:
        blockN ciphertext: captured ciphertext to invert
        integer t: chain length
        filename file: reference to file containing Hellman table
        function R(), E_k(), f(): reference to functions R, E_k and f
    output: list of blockN representing candidate keys or Null
% given a ciphertext block, find all candidate keys in the table
% another variant could check each key with another ciphertext as
% it is found an return only the correct key

  var table[] as array of Ntuple
  var Y as blockN
  var candidate, SP, EP as blockN
  var lres as list of blockN
  var j, k as integer
```

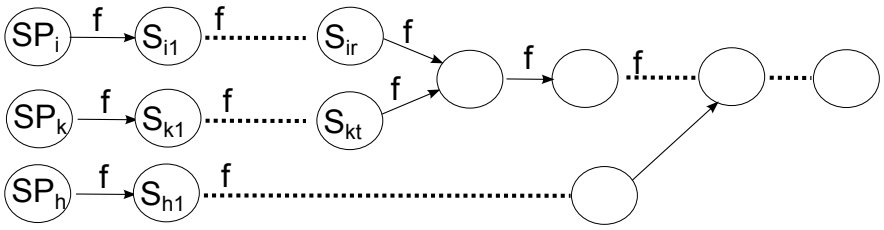

Figure 5.2: Chain Merges

```
load (table,file)
Y = R(ciphertext)
for (j = 0 to t-1)
    Search for (EP,SP) in table such that Y=EP
    if(SP is not null)
        candidate = SP   //second element in the tuple
        for (k = 0 to t - j - 1)
            candidate = f(candidate)
        if( E_k(candidate) = ciphertext )
            append_to_list (lres, candidate)
    Y = f(Y)
return (lres)
```

In the previous description we ignored the case when more than one chain ends in the same EP_x. If this is the case, all chains that end in EP_x must be reconstructed until the correct key is found or no more chains remain.

One problem of Hellman's TMTO tables is that if two chains share a common element, then the chains will be identical from that element onwards (see Figure 5.2). This means that the effect of collisions is amplified, as a single colliding element means the chains merge from that element onwards. The larger the tables, the higher the probability that a new row added merges with an existing one. As the table grows larger each new chain will be adding fewer new elements on average while using the same amount of memory and computation effort. Worse, we don't have a simple way to check for mergers short of searching each chain's endpoint in the remaining chains, which is usually prohibitively expensive. Another problem is chains that run on a loop, also decreasing coverage.

Due to the birthday paradox, if we have n existing chains of length t, we expect that the probability of a new chain to merge with any of the previous ones to be negligible when $nt^2 << N$ and large when $nt^2 >> N$.

Under the assumption that f is a random function Hellman calculated a lower bound to the success probability of a single table as equation (5.1). Using the approximation $(1-1/b)^a \approx e^{-a/b}$, which is appropriate when $a = O(b)$, most terms in the right side of equation (5.1) can be approximated by $(1 - it/N)^{j+1} \approx e^{itj/N}$. As explained by Hellman, when $mt^2 << N$, each term in equation (5.1) is close to one, so it reduces to $P_{table} \geq mt/N$. On the other hand, when $mt^2 >> N$ most terms will be small.

$$P_{table} \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(\frac{N - (i \times t)}{N} \right)^{j+1} \tag{5.1}$$

Hellman evaluated equation (5.1) numerically when $mt^2 = N$ (with both m and t large) finding that it is equal to $0.8mt/N$. Thus using $mt^2 = N$ as criteria to determine the size of the matrix means that the cryptanalytic effort is increased by 0.25 (that is, 80% of the calculated values will be unique).

He proposes to use this criteria, and calls it the "matrix stopping rule", which can be more generally expressed as $mt^2 = H_{stop}N$, with the recomendation that H_{stop} should be a number close to 1. Larger values increment the coverage of a single table at the expense of proportionally higher memory consumption, while smaller values make more efficient memory use.

One recommendation given by Hellman is to take $m = N^{1/3}, t = N^{1/3}$. Using those values the table has less than $m \times t = N^{2/3}$ unique elements, and this means the expected probability to find a value in the table is less than $N^{2/3}/N = N^{-1/3}$ which is pretty limiting. So Hellman proposes to calculate $r = O(N^{1/3})$ different tables, each with a different reduction function R. There will be collisions between some elements in different tables, but those collisions do not represent a merge in the corresponding chains as a different reduction function will be applied in each chain.

Hellman gives an approximate value for the success probability of r generated tables as $P_{total} = 1 - Prob(\text{failure in all tables}) = 1 - (1 - P_{table})^r$ with the assumption of independence between tables (this assumption is criticized in [10], however the same assumption is used by other authors).

The memory necessary for this attack must be enough to store the starting and ending points of all the chains, which amount to $m \times r$ chains. Some optimizations can be applied to decrease total memory use, like taking the initial points as consecutive integers and only storing enough bits to describe the m starting points.

The precomputation time is proportional to the total number of applications of the step function, $m \times r \times t$, plus the time necessary to sort the tables which is usually neglected.

For the online phase, in the worst case (if the solution is not found), the calculations needed to search a value in each table are $t - 1$ applications of f and one application of R, so for r tables $T_{online} \leq t \times r$. We may also have false alarms which must be ruled out. Hellman claims that false alarms increase the expected computation by at most 50%, however some of his assumptions are not resasonable as shown by Avoine et al in [5]. The cost of resolving false alarms is further studied for example in [40].

After each application of f, the result must be looked up in the table, so the maximum number of table lookups is $t \times r$.

Further mathematical analysis of the parameters of this method can be found for instance in [10] and [50]. However, Hellman's tables have a serious disadvantage

when the tables do not fit in random access memory (that is, for most "interesting" problems where the state space is big) and are thus stored in persistent media like hard disks which are much slower to access than RAM, specially for random access. Even though the number of table lookups is of the same order of the number of function evaluations, in practice the online attack time is dominated by the time to search the tables (this may improve in the near future as the price of solid state drives continues to plunge).

5.2. Distinguished Points

To decrease the time required for disk lookups in Hellman's attack, in 1982 Ron Rivest proposed the Distinguished Point (DP) method. This method was initially analyzed in 1998 by Borst et al [16] who proposed a theoretical analysis of the method, and by Standaert et al in 2002 [58] who improved the previous analysis.
A distinguished point is a value that satisfies an efficiently verifiable criterion. Usually simple functions like having the last k bits equal to zero are used.

In [58] a DP-Property is defined considering that if $\{0,1\}^k$ is the key space and $d \in \{1,2,3,\cdots,k-1\}$, then a DP property of order d is a property that holds for 2^{k-d} different elements of $\{0,1\}^k$. Then a Distinguished point (DP) is a value that satisfies the DP-Property. One often used DP-property is having d bits with fixed values, that is, to check if a value is a Distinguished point the property to be checked is that a given set of bits have a fixed value. The definition of DP-Property can be extended to non integer values of d, and other simple functions with greater granularity can be thought of, like checking if the value is below a certain threshold.
In the Hellman's method with distinguished points, we choose m starting points ST_i like in the original Hellman's method, and also a DP-Property of order d. Instead of calculating chains of a fixed length t we stop calculating once a distinguished point is reached (or an upper limit t_{max} is reached, as protection against loops in the chains). On average, the chains will be of length 2^d, but they will be of variable length.

Knowing function R and f We can represent the calculation of a single table with the following pseudocode:

Listing 5.3: Creating Hellman's with DP table

```
typedef blockN  {0,1}^N                  % N-bit block
typedef Ntuple (blockN, blockN)          % tuple of two N-bit blocks

Process CreateDPTable
    inputs:
        integer m: number of starting points
        function DProperty(): distinguished property function (returns true if input is DP)
        function R() and f(): reference to functions R and f
    output: list of Ntuples representing hellman table with DP (possibly stored on disk)
```

```
constant L as integer   % maximum length for loop prevention
var result[m] as array of Ntuple
var SP, S as blockN
var count as integer

for (j = 1 to m)
    SP = random()
    S = f(SP)
    count=0
    while (count < L and not DProperty(S))
        S = f(s)
        count = count + 1
    if( count < L)                % found DP
        result[j] = tuple(S,SP)
    else                          % DP not found
        result[j] = NULL
sort(result)
store(result)
return(result)
```

The limit L is imposed so that we break out of a looping chain, and must be chosen large enough so that there is a low probability that a non-looping chain exceeds length L.

For the online phase, given the captured ciphertext C, we calculate $Y_0 = R(C)$, and iterate $Y_i = f(Y_{i-1})$ until we reach a distinguished point (or the upper limit L). If we reached a DP Y_{DP}, we search for Y_{DP} in the last column of the table. If Y_{DP} is not found, then k_e is not in the table and we can continue with the next table. If we do find $EP_j = Y_{DP}$, we reconstruct the chain starting from SP_j until we reach $R(C)$ or a DP. If we reach a DP, it means it was a false alarm and we should continue with the next table. If $S_{jk} = R(C)$, then S_{jk-1} is the candidate key we are searching, and must be checked for false alarms just like in the original Hellman TMTO.
Knowing function R and the step function f the online phase is represented in the pseudocode in listing 5.4. If we have more than one table, the search is repeated for each table.

Just as in the previous section, in this description we ignored the case when more than one chain ends in the same EP_x. If this is the case, we must reconstruct all chains that end in EP_x until the correct key is found or no more chains remain.

This method has some advantages compared to the original Hellman's method:

- In the online phase, we only need to do one search in each table (once we reach a distinguished point), thus decreasing substantially the time due to disk accesses
- We can discard chains that loop. When the length of the chain reaches the chosen value L we declare a loop and discard the chain
- With a wise choice of DP-property, we can avoid storing the information that makes the value distinguished, thus saving some memory. For instance,

Listing 5.4: Search in Hellman's DP table

```
typedef blockN  {0,1}^N                  //N-bit block
typedef Ntuple (blockN, blockN)          //tuple of two N-bit blocks

function DPSearch
    inputs:
        blockN ciphertext: captured ciphertext to invert
        function DProperty(): distinguished property function (returns true if input is DP)
        filename file: reference to file containing Hellman table
        function R() and f(): reference to functions R and f
    output: list of blockN representing candidate keys or Null

% given a ciphertext block, finds all candidate keys in the table
% another variant could check each key with another ciphertext as
% it is found an return only the correct key

  constant L as integer
  var table[] as array of Ntuple
  var Y, SP, EP as blockN
  var cand, fcand as blockN
  var lres as list of blockN

  load (table,file)
  Y = R(ciphertext)
  count = 0
  while ( count < L and not DProperty(Y) )
      count = count + 1
      Y = f(Y)
  if (DProperty(Y))
      Search for (EP,SP) in table such that Y = EP
      if(SP is not null)
          cand = SP
          fcand = f(cand)
          while (not DProperty(fcand) and fcand != R(ciphertext))
              cand = fcand
              fcand = f(cand)
          if( fcand == R(ciphertext) )
              append_to_list (lres, cand)
  return(lres)
```

one common DP-property is to have the d least significant bits a fixed value. In this case, those bits need not be stored.

- We can easily detect chains that merge, as they end in the same distinguished point, and only keep one of them (it is better to discard the shorter ones, but this forces us to store the chain length temporarily for all chains in the table). This means we can easily create tables without repeated elements.

Tables where merging chains are removed are called perfect tables, and were suggested by Borst, Preneel and Vandewalle in 1998 [16]. This ensures there are no repeated elements within a single table (of course we can have repeated elements between different tables). They are important because we can better make use of the available memory, storing only chains with no repeated elements thus getting a better coverage for the same memory and online computation. The drawback is a lengthier precomputation phase to replace the removed chains.

Some parameters for the DP tradeoff are more difficult to calculate than the corresponding parameter for the original Hellman Tables, like the average chain length, expected success probability, and online time. Standaert et al [58] calculate several parameters of the tradeoff, and give important insight into the method. For the exact parameters of the tradeoff, we refer to the work of Standaert et al [58]. Instead we summarize some interesting points from this paper:

- Longer chains have higher collision probability, so discarding colliding chains shortens the average chain length. Some previous studies ignored this fact.
- Discarding chains that are very short (and replacing them) increases table coverage at the expense of increased precomputation. They thus propose storing only chains of length between t_{min} and t_{max}, and give the parameters of the tradeoff as a function of t_{min} and t_{max} allowing the user an informed choice of values.
- If the aim is to maximize the efficiency in the attack phase, it is more efficient to continue computation beyond the "matrix stopping rule" as proposed by Hellman, $mt^2 = N$, at the expense of a more expensive precomputation phase.

Regarding the last item, Barkan in his PhD Thesis [7] studies what he calls "stretched matrices", where more chains than suggested by Hellman's "matrix stopping rule" are calculated and only the longer chains are stored, thus trading online time for a longer preprocessing. He reaches a similar conclusion, namely that you get a more efficient attack phase at the expense of a more expensive precomputation phase.

The most used distinguishing property used is having d bits (the most significant or least significant d bits) with a fixed value, which means the expected length of each chain is a power of 2. However if we want a wider choice in the expected length, we can use other distinguishing properties, like asking for the value to be less than a constant dp with $dp < N$ (the expected length of each chain in this

case is N/dp)

5.3. Rainbow Tables

In 2003, Philippe Oechslin [54] proposed a new method, which he called Rainbow Tables. The idea of the method is to calculate tables similar to those of Hellman's TMTO, but to change the reduction function R in each step of chain generation (thus getting a sequence of t reduction functions $R_1, R_2, \cdots R_t$ and corresponding step functions $f_1, f_2, \cdots, f_t$). Each link in the chain uses a different "color", that is a different reduction function, hence the name "rainbow" tables.

$$\begin{array}{c} SP_1 = S_{10} \xrightarrow{f_1} S_{11} \xrightarrow{f_2} \cdots \xrightarrow{f_t} S_{1t} = EP_1 \\ SP_2 = S_{20} \xrightarrow{f_1} S_{21} \xrightarrow{f_2} \cdots \xrightarrow{f_t} S_{2t} = EP_2 \\ \vdots \\ SP_m = S_{m0} \xrightarrow{f_1} S_{m1} \xrightarrow{f_2} \cdots \xrightarrow{f_t} S_{mt} = EP_m \end{array}$$

Figure 5.3: Rainbow table

Some important characteristics of rainbow tables are:

- As the f_i functions are used only once, there is no possibility to have a loop
- A collision between two chains only results in a chain merge if the collision happens in the same column in both chains. This greatly diminishes the probability that two chains merge when compared to Hellman Tables
- As a result of the previous point, the number m of chains in each table can be much larger compared to the original Hellman TMTO. Oechslin shows it can be increased to the value for which $m \times t \approx N$
- Merges of rainbow chains can be easily detected and eliminated, as the colliding chains will end in the same point. This can be used to generate merge-free tables (in Rainbow Tables this does not imply that there will be no repeated elements inside a single table).
- Rainbow chains are of a fixed length, which according to [54] helps reducing the number of false alarms and the extra work due to false alarms.

A rainbow table acts almost as if each column of the table was a separate classic Hellman table. Collisions within a classic table or a column in a rainbow table generate a merge, whereas collisions between different Hellman tables, as well as between elements in different columns of the same rainbow table, do not generate a merge. This analogy is used to show [54] that a rainbow table of mt chains of length t has the same success rate as t classic tables of m chains of length t.

The proposed matrix stopping rule changes to take this into consideration, becoming $mt = H_{stop}N$ [41].

The off-line phase is similar to Hellman's TMTO, except that in each step of the chain generation a different reduction function R_i (and thus step function f_i) is used.

Knowing functions R_i and $f_i = R_i(E_k(P))$, we can represent the calculation of a single table with the following pseudocode:

Listing 5.5: Rainbow table calculation

```
typedef blockN  {0,1}^N                  % N-bit block
typedef Ntuple (blockN, blockN)          % tuple of two N-bit blocks

Process CreateRainbowTable
    inputs:
       integer m: number of starting points
       integer t: chain length (number of colors)
       functions f_1() ··· f_t(): reference to functions f_i
    output: list of Ntuples representing Rainbow table (possibly stored on disk)


  var result[m] as array of Ntuple
  var SP, S as blockN
  var count, j, k as integer

  for (j = 1 to m)
       SP = random()
       S = SP
       for (k = 1 to t)
            S = f_k(S)
       result[j] = tuple(S,SP)
  sort (result)
  store (result)
  return (result)
```

For the online phase, for a given ciphertext C the procedure is as follows. First $R_t(C)$ is calculated and searched in the table. If no match is found, then calculate and search $f_t(R_{t-1}(C)), f_t(f_{t-1}(R_{t-2}(C))) \cdots$ and so on. At each step, if a matching EP_i is found for color r, reconstruct the chain from SP_i until color r, if it coincides with $R_t(C)$ then the value at color $r-1$ is our candidate for the key, if not it was a false alarm.

The online phase is represented in the following pseudocode, where functions R_i and $f_i = R_i(E_k(P))$ are known. If we have more than one table, the search is repeated for each table.

Listing 5.6: Search in Rainbow table

```
typedef blockN  {0,1}^N                  //N-bit block
typedef Ntuple (blockN, blockN)          //tuple of two N-bit blocks

function RainbowSearch
```

inputs:
blockN ciphertext: captured ciphertext to invert
integer t: length of chain (number of colors)
filename file: reference to file containing Rainbow table
functions $R_1() \cdots R_t()$ and $f_1() \cdots f_t()$: reference to functions R_i and f_i
output: list of blockN representing candidate keys or Null

```
% given a ciphertext block, finds all candidate keys in the table
% another variant could check each key with another ciphertext as
% it is found an return as soon as the correct key is found

  var table[] as array of Ntuple
  var m, j, k as int
  var Y0, Y, SP as blockN
  var cand, fcand as blockN
  var lres as list of blockN

  load (table,file)
  m = tablesize(table)
  for (k = t downto 1)
      Y0 = R_k(ciphertext)
      Y = Y0
      for (j = k+1 to t)
          Y=f_j(Y)
      Search for (EP,SP) in table such that Y = EP
      if(SP)
           cand = SP
           for (j = 1 to k-1)
               cand = f_j(cand)
           if( f_k(cand) == R_k(ciphertext) )
               append_to_list (lres, cand)
  return(lres)
```

The success probability (coverage) of a single table was calculated by Oechslin [54] to be

$$P_{table} = 1 - \prod_{i=1}^{t} \left(1 - \frac{m_i}{N}\right) \text{ where } m_1 = m,\ m_i = N\left(1 - e^{-\frac{m_{i-1}}{N}}\right), i > 1 \quad (5.2)$$

The success probability with r tables is just as in the case of other TMTOs:

$$P_{total} = 1 - (1 - P_{table})^r \quad (5.3)$$

The disk consumption of r rainbow tables is $M = m \times r \times b_{tuple}$, where b_{tuple} is the space required to store a (SP, EP) entry.

In Oechslin's paper the worst case online effort to search a single table ignoring the effort to verify false alarms is calculated to be $\frac{t(t-1)}{2}$ applications of function f, which is half the effort in Hellman's TMTO. Also, there are only t table searches, similar to Hellman's tables with distinguished points. However,

Parameter	Hellman DP	Rainbow
Success probability	0.55	0.55
Memory in bytes	112×10^{12}	112×10^{12}
Precomputation	2^{64}	2^{64}
Worst case online complexity (iterations)	6.98×10^{12}	3.49×10^{12}

Table 5.1: Initial example of Hellman tables and Rainbow tables

not taking into consideration false alarms seems deceiving, as Oechslin notes in his paper that in the examples provided the calculations due to false alarms make about 75% of the cryptanalysis effort. In section 5.6 we resume the papers that study the different TMTOs, which improve in the initial characterization done by Oechslin and Hellman respectively.

Sample values

We can calculate some initial values for both Hellman's and Rainbow tables using the initial description on each paper. In our case the search space consists of $N = 2^{64}$ elements.

Using Hellman's matrix stopping rule, $mt^2 = N$, and Hellman's recommendation, one possible choice of parameters is $m = t = r = 2^{21.33} = 2642246$ (m chains of t elements each per table, with r tables). For Rainbow tables we can use a single table of $m = 2^{42.67} = 6,98 \times 10^{12}$ chains of length $t = 2^{21.33} = 2642246$. Using those values some parameters of the tradeoff are shown in table 5.1. The memory necessary to store the tables was calculated using a naive implementation, using 16 bytes to store the pair starting point - ending point. We will later see storage optimizations that improve this figure.

The worst case online complexity is calculated for the case in which the value to be searched is not found, and ignoring the work due to false alarms, so it should be taken as a very rough approximation.

5.4. Time Memory Data Tradeoffs

Time memory data tradeoffs (TMDTO) are a variant of TMTO in an scenario where several data points are known, for example several captured ciphertexts, and inverting any of them is enough to solve the problem. They appear naturally in the application of stream ciphers, where the function to invert is the function mapping the internal state of the cipher to some output bits, and any state found is enough to decrypt the rest of the ciphertexts (or sometimes to find the key). They can appear in other scenarios, like having the same text encrypted with different keys, and only needing the inversion of one of them. The attacks on A5/1 belong to the former scenario, so we will base our explanation in that application.

Stream ciphers keep an internal state which completely defines the future output of the cipher. The internal state is initialized using the key and initialization vectors, and then modified at each step using some function, and generating some output bits that depend on the internal state. For A5/1 the internal state is the value of the three registers $R1$, $R2$ and $R3$, and at each step one output bit is generated.

Let's consider a stream cipher, with an internal state encoded in $k = log_2(N)$ bits. Let's call g the function that maps the internal state $x \in \mathbb{X}$ to the output prefix $y \in \mathbb{Y}$, where the output prefix consists of the first $log_2(N)$ bits of output produced by the cipher starting from state x. We can use any of the previous TMTO algorithms to invert g and find the internal state, and afterwards step the cipher as many times as needed to reconstruct the cipherstream, or try to recover the key from the internal state.

We can recover more than one output prefix both from different initializations of the cipher, and from each keystream whose length is $w > log_2(N)$. In this later case, let's call $x_1, x_2, \cdots, x_w$ the bits of the keystream, and take $yi = (x_i x_{i+1} \cdots x_{k+i-1})$. In this way we can find $w-k+1$ different output prefixes from each keystream which we can try to attack individually.
Having several data points to search in the tables means that given the same table size, there is a higher success probability compared to an scenario in which a single block is available, or conversely, that we can build smaller tables and still have a high success probability.
The first tradeoff attacks for stream ciphers were proposed by Babbage [6] and Golic [38], and consist basically in a table lookup. In 2000, Biryukov and Shamir [14] combined this approach with Hellman's method. The key idea is to use the birthday paradox: if you have two independently chosen subsets of a key space of N points, they are likely to intersect if the product of their sizes exceeds N. So if we have D points to search in the tables, the size of the tables can be a factor of D smaller than the tables needed to obtain the same success probability with a single captured ciphertext.

In order to reduce the number of states covered by the matrices Biryukov and Shamir propose to reduce the number of matrices from r to $r_D = r/D$ in Hellman's method, thus reducing the memory used by a factor of D. The attack effort remains approximately the same, as each point requires less effort, but we must search D points. It is more convenient to reduce the number of matrices than to reduce the number of initial points m, as the on-line effort is independent of m (if we ignore the search cost) but increases linearly with the number of matrices.

The parameters for Biryukov and Shamir's tradeoff as reported by [14], ignoring constant and logaritmic factors, are (for $D^2 \leq T \leq N$):

- precomputation $P = N/D$
- $TM^2D^2 = N^2$ (where T is the attack effort and M the memory required)

Of course we can apply the distinguished point idea to this tradeoff, thus drastically reducing the number of disk accesses.
Biryukov et al [13] studied more possibilities for the tradeoff, and in particular

showed that the tradeoff curve for the rainbow attack is $TM^2D = N^2$, which is worse than the curve for Hellman's attack which is $TM^2D^2 = N^2$ if $D > 1$.

In [10] and [7], Barkan, Biham and Shamir formalize a general model of cryptanalytic time/memory tradeoffs, which contains the previous TMTOs as special cases. They provide some general bounds on the coverage and online time of TMTO and TMDTO schemes, and in their own words they "formally show that no cryptanalytical time-memory tradeoffs which are asymptotically better than existing ones can exist, up to a logarithmic factor" [10]. This is an important theoretical result that imposes limits on the improvements that may be attained searching for new methods. However this result does not help the practitioner choose a tradeoff to mount an attack in a specific situation, as the constant and logaritmic factors can make a huge difference in practical situations.

In the same paper [10], Barkan et al also show two new rainbow time memory data tradeoffs, which they call "thin rainbow tables" and "fuzzy rainbow tables". The key idea is to reduce the number of colors in the standard rainbow tables, by repeating colors. Those two methods are further presented in Elad Barkan's PhD thesis, [7], and will be presented in the following section.

5.4.1. Rainbow Time Memory Data tradeoffs

The basic rainbow table method can be used for multiple data attacks, but its tradeoff curve $TM^2D = N^2$ is worse than the curve for Hellman's attack $TM^2D^2 = N^2$, which means that as D grows Hellman's method compares favorably to Rainbow tables.

To mount an attack using Rainbow tables keeping the same probability as an attack with a single data point we can use the same amount of memory M but shorten each row to t/D elements. The new rainbow matrix covers Mt/D points, which represent the same fraction N/D of the space as the TMDTO using Hellman's method. Following [7], the tradeoff curve is $TM^2D = N^2$, which is worse than the curve for classical Hellman and DP tables. Thus Barkan proposes two new methods, "thin rainbow tables" and "fuzzy rainbow tables", which aim to reduce the number of colors and thus the effort in the online phase, without greatly incrementing the collision probability within a table.

Thin rainbow Time Memory Data tradeoff

In thin rainbow tables, to reduce the number of colors keeping the table size constant, Barkan proposes to choose S colors, and repeat them t times:

$$f_0 f_1 f_2 \cdots f_{S-1} f_0 f_1 f_2 \cdots f_{S-1} \cdots f_0 f_1 f_2 \cdots f_{S-1} \tag{5.4}$$

For this case the recommended matrix stopping rule is $Mt^2S = N$, and to cover

N/D elements, they recommend $t = D$.

The resulting tradeoff is shown to be [7] $TM^2D^2 = N^2$, and the number of disk accesses is $D\sqrt{T}$. To reduce this last number, Barkan proposes to use distinguished points to mark the points that can end a chain (that is, instead of repeating the sequence exactly t times, you stop once f_{S-1} is a distinguished point). However, this brings about the same problems as in the classic Hellman tables with distinguished points, namely that once you eliminate colliding chains the remaining chains are on average shorter than t, unless you drop the shorter ones and replace them (which increases preprocessing time).

Fuzzy rainbow Time Memory Data tradeoff

Another method proposed by Barkan, which he refers to as "fuzzy rainbow matrix", also reduces the number of colors to s, but lumps all instances of the same color together. To introduce fuzzy matrices, Barkan first defines a thick rainbow matrix as an scheme in which colors are repeated t times:

$$\underbrace{f_0 f_0 f_0 \cdots f_0}_{t\ times}\underbrace{f_1 f_1 f_1 \cdots f_1}_{t\ times}\cdots\underbrace{f_{S-1} f_{S-1} f_{S-1} \cdots f_{S-1}}_{t\ times} \tag{5.5}$$

This scheme reduces the number of colors, but in the online phase we must not only try all colors, but also all "phases" (that is, all t possible lengths of the current segment). To avoid this, Barkan proposes instead to stop iterating each color when arriving at a distinguished point. Each chain consists of s segments, each one iterating with a different step function (color) and ending in a distinguished point. In this way, when searching a value in the table only one search is needed for each color, as all segments end in a DP. This scheme is called fuzzy rainbow tables.

$$\underbrace{f_0 f_0 f_0 \cdots f_0}_{stop\ at\ DP}\underbrace{f_1 f_1 f_1 \cdots f_1}_{stop\ at\ DP}\cdots\underbrace{f_{S-1} f_{S-1} f_{S-1} \cdots f_{S-1}}_{stop\ at\ DP} \tag{5.6}$$

The resulting tradeoff is shown to be [7] $2TM^2D^2 = N^2$ if $T >> D^2$, which is a factor of two better than thin rainbow tables. Disk accesses are proportional to $\sqrt{2T}$ (which is better than thin rainbow tables for $D > 1$)
A fuzzy rainbow matrix can be seen as the concatenation of s sub-matrices DM_i, where the starting points of DM_{i+1} are the ending points of DM_i.
The fuzzy rainbow TMDTO was proposed as an improvement for the multi target case, but can also be used in the single target environment, and as will be seen later in many cases it is the best currently known tradeoff.

5.5. Memory optimizations

There are some proposed optimizations that can be applied to all the tradeoffs, that try to use the available storage space in the most optimal way. The tradeoffs we have shown up to now consider memory M as the storage necessary to store M

chains, each one represented by one start point and its corresponding end point, which in the most naive implementation take $2 \times log_2(N)$ bits to store. If we store each entry using less bits we can have the same success probability using less memory, or alternatively improve the success probability using the same memory.

The starting points need not be chosen at random [10], [5], so we can represent them in $\lceil log_2(m) \rceil$ bits where m is the number of starting points in the table.

For tables using distinguished points, the ending points can be stored without the information that makes them distinguished. All tables are stored sorted on the ending points, and optimizations are possible where only the least significant bits are stored and a separate index table contains the most significant bits and points to the beginning of the corresponding LSBs (see e.g. [5] and [47]).

A final optimization proposed is to truncate the ending points. As we expect to store on the order of m end points, they can be truncated to slightly more than $log_2(m)$ bits as proposed in [10]. [41] shows that the increase in on-line time is negligible if $log_2(m) + \epsilon$ bits are used, and give some criteria to choose ϵ, showing through examples that ϵ between three and eight is adequate (in their examples). Hong and Kim [48] [47] include endpoint truncation in their calculations, and we will do the same in the following chapter.

Another optimization in the use of storage is called checkpoints, a technique proposed by Avoine et al [4]. The objective of checkpoints is not to decrease memory usage, but to diminish the effect of false alarms, storing one or a few bits of information about the chains besides the starting and ending point. It consists on defining a set of positions α_i and a function G, and for each chain storing the values of all $G(S_{\alpha_i})$. To be efficient, G should be easily computable and the storage of its output should require few bits. In their examples G's output is a single bit. When searching for a value Y in the tables, we start by reconstructing the chain from Y and searching for a coincidence in the ending point. If we find such a coincidence, we compare the values of the checkpoing for all the values α_i the chain has gone through. If any of them differ this signals a false alarm, thus avoiding the costly chain regeneration.

5.6. Comparison of the TMTO methods in the literature

Several metrics can be compared between different algorithms when varying the tradeoff parameters, possibly leading to different conclusions.

The paper in which Oechslin [54] introduced Rainbow Tables included a rough comparison with Hellman's tables with distinguished points, showing that for similar storage usage and precomputation effort, there is a factor of two improvement offered by Rainbow Tables without taking into consideration false alarms, and hinting that there should be a greater improvement when considering false alarms. He also shows experimental results that corroborate the improvement, obtaining a 7x improvement on the calculations needed on the attack phase for high success rates. However he only does a worst case analysis, and for Hellman's matrices the calculations are only bounds on the quantities studied. The fact that false

alarms are ignored introduces a large error. He also ignores the possible storage optimizations, which improve both methods in different proportions.

In [10] and [7], Barkan, Biham and Shamir formalize a general model of cryptanalytic time/memory tradeoffs, which contains the previous TMTOs as special cases. They provide some general bounds on the coverage and online time of TMTO and TMDTO schemes, and they formally show that no cryptanalytical time-memory tradeoffs which are asymptotically better than existing ones can exist, up to a logarithmic factor. However, their analysis ignores the effect of false alarms, and consider only the worst case analysis. They also claim that the Rainbow tradeoff is worse than the original Hellman tradeoff with or without distinguished points, a claim that is later shown to be false in most cases in [5] and [42] once false alarms and all possible optimizations are taken into account.

In [40] Hong studies the relative cost of dealing with false alarms, and improves the calculation of the parameters of the non-perfect Hellman tradeoff and perfect and non-perfect Rainbow tradeoffs, and in [42] Hong and Moon compare the non-perfect Hellman, Hellman with DP and Rainbow tradeoff using expected values instead of worst case analysis, concluding that for most practical cases the Rainbow tables present a better tradeoff.

Avoine el al [5] study the perfect variants of the tradeoffs, and conclude that Rainbow tables are better in most cases compared to Hellman tables.

The most recent characterization of the different algorithms was carried out by Kim and Hong [47] [48], where they analyse the expected performance of non-perfect and perfect fuzzy rainbow tables, and compare them with each other and with the perfect and non-perfect rainbow tradeoff, in the single data case ($D = 1$). They take into account the effect of false alarms and storage optimization, and perform an analysis based on the expectation of the involved quantities instead of analysing the worst case bounds as several previous works had done. Their conclusion is that among all the studied algorithms, for the single inversion target case the perfect fuzzy rainbow table tradeoff is preferable under most conditions, using the criteria laid out in [42]. We will use their results concentrating our efforts on the perfect fuzzy rainbow table tradeoff, and extend their results to the multi target case, later showing how to use their results in a realistic scenario to choose the parameters of the tradeoff.

Another important result from the paper is showing how to calculate the parameters of the tradeoff and finding approximate formulas for the expected values of several characteristics of the tradeoff which can be expressed using a few parameter combinations, thus decreasing the amount of variables and simplifying the choice of parameters.

This page intentionally left blank

Chapter 6

Extending Kim and Hong calculations to the multi target environment

The best known TMTO in many realistic scenarios for the *single target* case was shown by Kim and Hong to be the perfect fuzzy rainbow tradeoff [48]. In this chapter we want to study the perfect fuzzy rainbow tradeoff parameters for the *multi-target* environment. In this case, we have the output of the function to invert for D different inversion targets, and the attack is considered a success if the correct preimage is found for any of them.

In this chapter we will addapt the calculations by Kim and Hong in the paper "Analysis of the Perfect Table Fuzzy Rainbow Tradeoff" [48] to the multi-target case. Most of the results in the paper translate unchanged or with minor modifications to the multi-target case. We will not reproduce all demonstrations that carry unchanged from the paper, instead showing the main points and referring to the original paper for the details.

We will mostly follow Kim and Hong's notation and use the same techniques and assumptions. In their work Kim and Hong use an "overline" notation for the parameters of the perfect tables, and no overline for the non-perfect tables (eg. $\overline{F}$ for the perfect tables and F for the non-perfect tables). Even though we will not deal with the non-perfect tables, we will keep their notation to avoid confusions when referencing their work.

We will also abuse the notation by expressing the approximate formulas as equalities just as in the referenced paper.

6.1. Summary of the notation

- N: number of elements in the domain of the function to invert
- m: number of chains in each table
- m_0: number of initial points chosen to calculate each table. Number of chains before removing chains that merge.

- s: number of colors (chain segments)
- t: expected length of each segment on a chain
- l: number of tables
- D: number of images available to attempt inversion
- r_{ij}: reduction function corresponding to color i in table j. When no confusion might occur we will just denote r_i the reduction function corresponding to color i of the current table
- f_{ij}: step function. Composition of functions r_{ij} and f, that is, $f_{ij}(x) = r_{ij}(f(x))$

A non-perfect fuzzy rainbow matrix can be seen as the concatenation of s sub-matrices DM_i, where the starting points of DM_{i+1} are the ending points of DM_i.

We will denote $|DM_i|$ the number of distinct points contained in DM_i.

6.2. Problem statement and assumptions

There is more than one possible problem to solve using a TMDTO. The problem we are trying to solve is, given a one-way function f and the images of D inputs to the one-way function that are chosen uniformly at random from the input space, find the original input for at least one of them using a perfect fuzzy rainbow tradeoff.

The authors of the paper make a few important assumptions:

- During the precomputation phase of each matrix, each submatrix DM_i is built, sorted and duplicates discarded before building DM_{i+1}. In case of duplicates the chain whose DM_i segment is longer is retained.
- The effort of sorting the ending points of the intermediate submatrices DM_i, which is of order $m \log(m)$, can be ignored.
- For the on-line phase, when there are several tables, it is assumed that the tables are processed in parallel, starting with color s for all chains and not changing to the following color until the current color has been processed for all tables.

The last assumption must be modified to account for the D inversion targets we have to search in the matrices:

- For the on-line phase, when there are several tables it is assumed that the tables are processed in parallel, starting with color s for all chains *and all D targets* and not changing to the following color until the current color has been processed for all tables *and all targets*.

Just as any other algorithm that relies on DPs to mark the end of an iteration we need a mechanism to terminate the iteration if the chain gets into a loop. We will use a constant bound on chain segment length, and we will assume that the constant is large enough that its effect on the algorithm performance is negligible, that is, the probability that a chain that does not loop is discarded because it reached the bound is negligible.

6.3. Detailed description of the algorithm

Table precomputation

In the off-line phase l tables must be calculated. We need a distinguished property with probability $1/t$, and s reduction functions for each table. For each table m_0 initial values are chosen. Using the m_0 initial values and the r_1 reduction function a DP matrix DP_1 is created, storing the starting point, ending point and length. Once the m_0 chains are calculated, they are sorted according to the ending points and for those with duplicate endpoints only the largest is retained, obtaining m_1 chains. The procedure is the same for the remaining $s-1$ colors taking as initial points for table DM_i the ending points from table DM_{i-1}, that is, to calculate table DM_i take the ending points from DM_{i-1} and reduction function r_i and calculate the m_{i-1} chains using step function f_i, storing the starting and ending points and the segment length. After all chains are calculated sort DM_i on the ending points, discarding chains with duplicate endpoints by keeping the chains with the longest i segment.

Knowing functions r_i and $f_i = r_i(E_k(P))$, we can represent the calculation of a single table with the following pseudocode:

Listing 6.1: perfect fuzzy rainbow table calculation

```
typedef blockN  {0,1}^N                  % N-bit block
typedef Ntuple (blockN, blockN)
typedef entry (blockN, blockN, integer)
                % two N-bit blocks plus an integer for length

Process CreateFuzzyTable
   inputs:
      integer m0: number of starting points
      function dp(): distinguishing property
      functions f_1() ··· f_s(): reference to functions f_i
      integer s: number of colors
   output: list of Ntuples representing perfect fuzzy rainbow table (possibly stored on disk)

  constant L as integer       % limit to avoid chains that loop
  var j,k,m as integer
  var result[] as array of entry
  var intermediate[] as array of entry
```

```
var SP, SP0 as array of blockN
var S as blockN
var count as integer

for (j = 1 to m0)
  SP[j] = random()
  SP0[j]=SP[j]
m=m0
for (k = 1 to s)
  for (j = 1 to m)
    S = SP[m]
    S = f_k(S)
    count = 0
    while (not dp(S) and count < L)
        S = f_k(S)
        count = count + 1
    if(dp(S))
        intermediate[j] = concatenate(S,SP0[j],count)
    else    % found a loop
        ''discard chain number j''
  sort_unique (intermediate) % sort entries discarding collisions
  m = size(intermediate)
  for (j = 1 to m)
      SP[j] = extract_endpoint(intermediate[j])
      SP0[j] = extract_startpoint(intermediate[j])
result = intermediate
store (result)
return (result)
```

Online computation

For the online phase, for a given set of d ciphertexts $C_1 \cdots C_d$ the procedure is as follows. For each color starting with the last (color s) and for all tables, search for candidates in the corresponding sub-matrix on all tables by constructing the sub-chain that starts at C_i using the current color, and ends at a DP in color s. Search the corresponding ending points on the corresponding table, and for each ending point found reconstruct the chain starting from the corresponding starting point and stopping at the current color, either because we found a candidate or a distinguished point is reached. Finally test the candidates with another sample and return if any one of them is the correct preimage, otherwise continue with the following color.
The online phase is represented in the following pseudocode, where functions r_{ij} and $f_{ij} = r_{ij}(E_k(P))$ are known.

Listing 6.2: Search in Rainbow table

```
typedef blockN  {0,1}^N                  //N-bit block
typedef Ntuple (blockN, blockN)          //tuple of two N-bit blocks

function FuzzyRainbowSearch
```

```
 inputs:
    blockN ciphertext_1 ... ciphertext_d: captured ciphertexts to invert
    function dp(): distinguishing property
    integer s: number of colors in each table
    integer l: number of tables
    filename file_1 ··· file_l : files containing Fuzzy Rainbow tables
    functions f_11() ··· f_ls(): reference to functions f_ij
    functions r_11() ··· r_ls(): reference to functions r_ij
    blockN PT, CT: plaintext and corresponding ciphertext to verify key candidates
 output: blockN representing found key or Null

% given d ciphertext blocks, finds first candidate in the tables

  constant L as integer
  var table_1[] ··· table_l[] as array of Ntuple

  var Y0[d], Y[d], SP[d] as array of blockN
  var cand, fcand as blockN
  var lres as list of blockN
  var i, j, k, f, t, count as integer

  for(t= 1 to l)
    load (table_t, file_t)
  for (k = s downto 1)          % color
    for (t=1 to l)              % table
       for (j=1 to d)
          Y0 = r_tk(ciphertext_j)
          Y = Y0
          for(i= k to s)
             count=0
             Y = f_ti(Y)
             while (not dp(Y) and count < L)
                 Y = f_ti(Y)
             if(count = L)
                 ''finish for loop on i and change to next j''
          Search for (EP,SP) in table_t such that Y = EP
          if(SP)
             cand = SP
             for (i = 1 to k-1)
                cand = f_ti(cand)
                while (not dp(cand))
                   cand = f_ti(cand)
             cand = f_tk(cand)
             while (not dp(cand) and E_k(cand) != ciphertext_j)
                cand = f_tk(cand)
             if(E_k(cand) == ciphertext_j)
                "Verify candidate using for example PT and CT"
                if("verification succeeds")
                   return (cand)
  return ("not found")
```

6.4. Preliminaries

As stated in section 6.2, after each sub-matrix DM_i is created it is sorted and duplicates are removed, keeping only the chain with the longest i segment from the ones that collide. We will call the reduced intermediate matrices $\widetilde{DM_i}$, and we will denote $\overline{DM_i}$ the collection of chains from DM_i that participate in the final table after eliminating duplicates. The expected number of distinct ending points for sub matrix DM_i will be noted m_i. m_0 is the number of starting points used to calculate the tables, and $m = m_s$ is the expected number of distinct ending points of the fuzzy matrix.

The choice of method for handling merges in the DM_i matrix by choosing the chain that has the shortest segment for color i is based on the desire to use existing results on perfect DP Hellman TMTOs. Kim and Hong argue that the choice of rule is not very important except for small values of s as the concatenation of multiple DP chains creates an averaging effect on the length value and the length distribution quickly approaches a normal distribution. They do not quantify the effect of this choice on the average chain length nor on other parameters. We will not discuss the effect of this choice in this chapter, but we will present some experimental results in chapter 7 that hint to a very low incidence of the choice of rule in the performance of the algorithm.

Kim and Hong frequently use two approximation techniques. The first is the approximation $(1 - (1/b))^a \approx e^{-a/b}$, which as explained in [41] is adequate when $a = O(b)$. The second technique is the approximation of a sum over a large index set into a definite integral.

6.5. Analysis of the perfect fuzzy rainbow table tradeoff

In this section we will calculate the main parameters for the tradeoff. We will find approximate expressions for the expected value of the precomputation effort, the success probability and the on-line effort.

Some values like the precomputation effort do not depend on the number of samples to invert D, so the results from [48] apply unchanged. In those cases we just replicate the result with more terse explanations. The rest of the values depend on the value of D, and the results from [48] are modified accordingly.

In [48] for the matrix stopping rule it is assumed that the parameters m, t and s are chosen such that $mt^2s = \overline{F}_{msc}N$, with a matrix stopping constant $\overline{F}_{msc}$. Later on we will give explicit dependencies with this constant, enabling the calculation of the value of $\overline{F}_{msc}$ optimal for each situation. It is also shown that $\overline{F}_{msc} < 2$.

In chapter 7 we will show the result of some experiments to gain insight into the use of the equations in the following sections, and the errors introduced by some of the approximations.

6.5.1. Success probability and precomputation effort

The precomputation effort for a table does not depend on the number of inversion targets D, so the calculation in [48] applies without changes to the $D > 1$ case. For the calculation of the precomputation effort first the number of chains that remain after each submatrix is built and duplicates removed is calculated. Then the expected precomputation effort to build each submatrix is calculated as the product of the expected number of initial points m_{i-1} (which is the number of chains resulting in the previous submatrix) times the expected chain length.

For the success probability we use the calculation of the coverage rate of each submatrix, and calculate the success probability of finding any solution under the assumption that all submatrices are independent of each other.

Number of Color Boundary Points

First we want to calculate how many chains are left after each sub-matrix is calculated. This is equivalent to the number of distinct points m_i. Knowing the number of boundary points will later help us calculate other magnitudes like the success probability and the precomputation effort.

In [47], Kim and Hong calculate the number of unique boundary points at each sub-matrix for the *non-perfect* fuzzy rainbow tradeoff, reaching the iterative formula in equation (6.1), which can be approximated be equation (6.2) when s is large. Here $F_{msc} = m_0 t^2 s/N$ is the matrix stopping constant for the non-perfect fuzzy rainbow tradeoff. The approximation is tested experimentally in the appendix of [48] where the worst error among their experiments was 5% between the approximate formula and their experimental results.

$$\frac{m_i}{m_0} = \frac{m_{i-1}}{m_0} \frac{2}{1 + \sqrt{1 + 2(F_{msc}/s)(m_{i-1}/m_0)}} \text{ where } F_{msc} = (m_0 t^2 s)/N \tag{6.1}$$

$$m_i = \frac{2m_0}{2 + F_{msc}(i/s))} \tag{6.2}$$

We can rewrite equation (6.2) using the parameters for the perfect fuzzy rainbow tradeoff, arriving at the following lemma

Lemma 6.1. *To create a perfect fuzzy rainbow matrix containing m nonmerging chains, the expected number of chains one has to generate is approximately*

$$m_0 = (2/(2 - \overline{F}_{msc}))m \tag{6.3}$$

The number of boundary points after calculating sub-matrix i is expected to be

$$m_i = \frac{2m}{(2 - \overline{F}_{msc}) + \overline{F}_{msc}(i/s)} \tag{6.4}$$

for $i = 0, 1, \cdots s$

Proof. From equation (6.2) for $i = s$ we find that $m_s = (2/(2 + F_{msc}))m_0$. We also know that $\overline{F}_{msc} = m_s t^2 s/N$, which means $m_s = \overline{F}_{msc}N/(t^2 s)$. Substituting m_s in the first equation we get $\overline{F}_{msc}N/(t^2 s) = (2/(2+F_{msc}))m_0$, which translates to $\overline{F}_{msc} = (2/(2 + F_{msc}))m_0 t^2 s/N = 2F_{msc}/(2 + F_{msc})$.

Solving the equation for F_{msc} we get $F_{msc} = 2\overline{F}_{msc}/(2 - \overline{F}_{msc})$ which is equivalent to the first statement.

Substituting into equation (6.2) we obtain

$$m_i = \frac{2(2/(2 - \overline{F}_{msc}))m}{2 + ((2\overline{F}_{msc}/(2 - \overline{F}_{msc}))(i/s))} = \frac{2(2)m}{2 * (2 - \overline{F}_{msc}) + 2\overline{F}_{msc}(i/s)} \tag{6.5}$$

which is the second claim. □

The lemma uses the approximate equation (6.2) to derive the closed formulas (6.3) and (6.4). A better approximation for the number of boundary points is given in [47] as equation (6.6). However it is more difficult to work with this iterative formula.

$$m_{i+1} = m_i \frac{2}{1 + \sqrt{1 + \frac{2m_i t^2}{N}}} \tag{6.6}$$

Looking at equation (6.3) it seems clear $\overline{F}_{msc}$ cannot be too close to two. We will later see that $\overline{F}_{msc} < 2$ is always satisfied and that if $\overline{F}_{msc}$ approaches 2 the precomputation effort grows unrealistically. In Kim and Hong's experiments for a wide variety of success probabilities and parameters the maximum value of $\overline{F}_{msc}$ was approximately 1.8.

Because it frequently appears in the remainder of their paper, Kim and Hong define the following notation:

$$\overline{f}_i = \frac{m_i t^2}{N} = \frac{2\overline{F}_{msc}}{(2 - \overline{F}_{msc}) + \overline{F}_{msc}(i/s)} \frac{1}{s} \tag{6.7}$$

The second equality is in reality an approximation, but we will keep Kim and Hong's notation and treat it as an equality assuming that s is sufficiently large. As we will see in chapter 7 the results obtained using this approximation are close to the experimental values for s as small as 4 or 5. The $\overline{f}_i$ notation will be used also for $i = s + 1$ and $i = s + 2$, in this case considering only the right hand term as definition of $\overline{f}_i$.

Since $\overline{F}_{msc}$ is bounded away from 2, $\overline{f}_i$ is $O(1/s)$.

Precomputation effort

We will ignore the effort to sort the intermediate precomputation matrices as it is of order $m \log m$, smaller than the effort of generating the submatrix. To calculate the precomputation effort it is enough to observe that each submatrix DM_i is a classic Hellman DP matrix, with m_{i-1} expected initial points and

expected chain length t before merge removal. Adding the expected effort to build the st matrices we get the precomputation effort. We will define a precomputation coefficient $\overline{F}_{pc}$ that resumes the dependency of the precomputation effort on the tradeoff input parameters. The following proposition carries unchanged from Kim and Hong's paper.

Proposition 6.2. *The precomputation phase of the perfect table fuzzy rainbow tradeoff is expected to require $\overline{F}_{pc}N$ iterations of the one-way function where the precomputation coefficient $\overline{F}_{pc}$ is given by equation* (6.8)

$$\overline{F}_{pc} = \frac{l}{t}\sum_{i=0}^{s-1}\frac{2\overline{F}_{msc}}{(2-\overline{F}_{msc})+\overline{F}_{msc}(i/s)}\frac{1}{s} \tag{6.8}$$

Proof. The computation of each sub-matrix $\widetilde{DM}_i$ from m_{i-1} starting points is expected to require $m_{i-1}t$ operations. Taking into account the l tables the expected cost of precomputation is:

$$lt(m_0 + m_1 + \cdots m_{s-1}) \tag{6.9}$$

Applying equation (6.4) we get

$$tl\sum_{i=0}^{s-1}\frac{2m}{(2-\overline{F}_{msc})+\overline{F}_{msc}(i/s)} \tag{6.10}$$

Using that $mt^2s = \overline{F}_{msc}N$ we get the stated equation.

□

Success probability

We will calculate the success probability given a set of perfect fuzzy rainbow tradeoff tables and the image of D values to invert. The proof will be similar to that in [48] but taking into consideration the D values to invert.

As we defined before, we call $\overline{DM_i}$ the collection of chains from DM_i that participate in the final table after eliminating colliding chains at colors $i+1\cdots s$. This set of chains is also a subset of $\widetilde{DM_i}$.

We will calculate the success probability as the fraction of the search space covered by the tables. As a first step we are interested in calculating the number of distinct points in each submatrix $\overline{DM_i}$.

First Kim and Hong define

$$\overline{F}_{cr,i} = \frac{|\overline{DM}_i|}{mt} \tag{6.11}$$

Then the coverage rate of a perfect fuzzy rainbow matrix is defined as

$$\overline{F}_{cr} = \frac{1}{mts}\sum_{i=1}^{s}|\overline{DM}_i| = \frac{1}{s}\sum_{i=1}^{s}\overline{F}_{cr,i} \tag{6.12}$$

Observe that $\overline{F}_{cr,i}$ is the (expected) number of distinct points in $\overline{DM}_i$ divided by the number of points in a square matrix of size $m \times t$. $\overline{F}_{cr,i} \cdot t$ gives the average length of the chains in color i after merge removal. We expect $\overline{F}_{cr,i}$ and $\overline{F}_{cr}$ to be of $O(1)$ order.

Lemma 6.3. *The coverage rate of the DP submatrix $\overline{DM}_i$ is given by*

$$\overline{F}_{cr,i} = \frac{2N}{m_i t^2} \ln\left(1 + \frac{m_i t^2}{2N}\right) = \frac{2}{\overline{f}_i} ln\left(1 + \frac{\overline{f}_i}{2}\right) \tag{6.13}$$

Proof. We refer the reader to [48] for the demonstration of this lemma, which relies heavily in observing that DM_i is a normal perfect DP Hellman matrix.

□

The coverage rate $\overline{F}_{cr,i}$ is always less than one, and examining the analysis of the perfect DP Hellman table in [49] it seems clear that the cause is the fact that in a DP matrix longer chains have higher collision probability, so in average the remaining chains are shorter.

Proposition 6.4. *Consider D different inputs to the one-way function chosen uniformly at random. Given the D images of these values under the one-way function as the inversion targets, the expected success probability of the on-line phase is given by*

$$\overline{F}_{ps}^{D} = 1 - \prod_{i=1}^{s} \left(1 - \frac{\overline{F}_{msc}\overline{F}_{cr,i}}{ts}\right)^{lD} \tag{6.14}$$

$$\overline{F}_{ps}^{D} \approx 1 - exp\left(-\overline{F}_{msc}\overline{F}_{cr}\frac{l}{t}D\right) \tag{6.15}$$

Proof. The success probability one can expect when searching a single target on a submatrix $\overline{DM}_i$ is $|\overline{DM}_i|/N$. As the submatrices were generated using different step functions we can consider them independent. The probability of not finding any of the D targets in the l sub-matrices of color i is

$$\left(1 - \frac{|\overline{DM}_i|}{N}\right)^{lD} \tag{6.16}$$

Multiplying for all colors

$$\prod_{i=1}^{s} \left(1 - \frac{|\overline{DM}_i|}{N}\right)^{lD} \tag{6.17}$$

As $\overline{DM}_i = mt\overline{F}_{cr,i}$ and $N = mt^2 s/\overline{F}_{msc}$, we can write

$$\frac{|\overline{DM}_i|}{N} = \frac{\overline{F}_{msc}\overline{F}_{cr,i}}{ts} \tag{6.18}$$

Substituting (6.18) into (6.17) we get the probability that no sample is found in any submatrix. Substracting from 1 we get the result in equation (6.14).

For the approximation we use that $(1-1/b)^a \approx e^{-a/b}$ when $a = O(b)$ to get

$$\overline{F}_{ps}^{D} \approx 1 - exp\left(lD\frac{\overline{F}_{msc}}{ts}\sum_{i=1}^{s}\overline{F}_{cr,i}\right) = 1 - exp\left(-\frac{lD}{t}\overline{F}_{msc}\overline{F}_{cr}\right) \tag{6.19}$$

which is equation (6.15)

□

Given any set of parameters, the success probability can be computed either from equation (6.14) or equation (6.15). (6.14) is a better approximation, however (6.15) may be easier to use as it does not depend on s, and the right side quantities can be expresed as a function of $\overline{F}_{ps}^{D}$.
We observe that the only difference with the $D = 1$ case is the D term in the exponent.

Online complexity

The calculation of the average online execution complexity consists in identifying the different steps in the online calculation, determining how likely each step of the computation is to be reached, and the cost of each step. It is important to notice the order in which the operations are carried out, in particular the fact that we process a certain color for all tables and all inversion targets before changing to the next color.

Observing the description of the procedure in section 6.3, starting from the last color the main steps for each color, each sample and each table are:

- starting from the image to invert, construct a partial chain starting from the current color until the ending point
- search the ending point in the table
- if the ending point is found, take the corresponding starting point and reconstruct the chain until the current color

In the last step we will either find the image we were looking for, or a distinguished point which means this was a false alarm.

To calculate the online complexity, we first determine how likely it is to reach color i, which is equivalent to saying that no solution was found in colors $i+1, \cdots, s$ for any table. Then the expected cost of generating a chain from color i to the end must be calculated, and then added for all D inversion targets and all tables.

Next the probability that a subchain that starts at color i merges with the matrix is found. Merging with the matrix means that the endpoint of the subchain is found in the corresponding table and thus the complete chain must be reconstructed. Most times the found endpoint is a false alarm, so the cost of resolving false alarms must be calculated.

Lemma 6.5. *The probability that the online chain that starts from color i must be searched, that is the probability that the submatrix $\overline{DM}_i$ is searched is*

$$\prod_{k=i+1}^{s}\left(1-\frac{|\overline{DM}_k|}{N}\right)^{lD} \approx exp\left(-\overline{F}_{msc}\frac{lD}{t}\frac{1}{s}\sum_{k=i+1}^{s}\overline{F}_{cr,k}\right) \approx (1-\overline{F}_{ps}^{D})^{(\sum_{k=i+1}^{s}\overline{F}_{cr,k})/(s\overline{F}_{cr})} \tag{6.20}$$

Proof. The proof is similar to the corresponding demonstration for the $D = 1$ case in [48], but taking into consideration the D inversion targets. The i^{th} DP submatrix $\overline{DM}_i$ will be searched for the correct answer only if no target can be found in $\overline{DM}_{i+1}\cdots\overline{DM}_s$ of any matrix. The probability that a target is found in submatrix k of any matrix is $|\overline{DM}_k|/N$, hence the probability that no target is found before searching submatrix i is

$$\prod_{k=i+1}^{s}\left(1-\frac{|\overline{DM}_k|}{N}\right)^{lD} \tag{6.21}$$

Substituting (6.18) into (6.21) we get:

$$\prod_{k=i+1}^{s}\left(1-\frac{\overline{F}_{msc}\overline{F}_{cr,k}}{ts}\right)^{lD} \approx \prod_{k=i+1}^{s} e^{-lD\frac{\overline{F}_{msc}\overline{F}_{cr,k}}{ts}} = e^{-lD\frac{\overline{F}_{msc}}{ts}\sum_{k=i+1}^{s}\overline{F}_{cr,k}} \tag{6.22}$$

The second approximation is a direct application of equation (6.15), see equation (6.23)

$$1-\overline{F}_{ps}^{D} \approx exp\left(-\overline{F}_{msc}\overline{F}_{cr}\frac{l}{t}D\right) \rightarrow$$
$$(1-\overline{F}_{ps}^{D})^{(\sum_{k=i+1}^{s}\overline{F}_{cr,k})/(s\overline{F}_{cr})} \approx exp\left(-\overline{F}_{msc}\overline{F}_{cr}\frac{l}{t}D(\sum_{k=i+1}^{s}\overline{F}_{cr,k})/(s\overline{F}_{cr})\right) \tag{6.23}$$

□

Proposition 6.6. *The cost of generating the online chains during the on-line phase of the perfect fuzzy rainbow tradeoff attack is expected to be*

$$T_{gen}^{D} = tlD\sum_{i=1}^{s}(s-i+1)(1-\overline{F}_{ps}^{D})^{(\sum_{k=i+1}^{s}\overline{F}_{cr,k}/s\overline{F}_{cr})} \tag{6.24}$$

Proof. The cost of generating each online chain that starts from the ith color is expected to be t(s-i+1), and there are l tables and D targets to consider. Thus the expected iterations of the one-way function for the generation of the online chains starting from color i is $t(s-i+1)lD$. If we multiply this value by the probability of not finding the solution before color i as given by lemma 6.5 we obtain the stated result. □

The last cost to be considered is the cost of resolving false alarms. A false alarm happens when the subchain that starts from the inversion target at color i merges with an existing chain in the table, which means that after reconstructing the chain the corresponding ending point is found on the table. We need the expected cost of solving an alarm originated when evaluating color i, which we can then multiply by the probability of reaching color i, the number of tables and the number of inversion targets to find the expected cost of solving the false alarms.

The cost of solving a possible false alarm at color i was calculated by Kim and Hong [48], their results are summarized in the following lemma.

Lemma 6.7. *The average cost of dealing with the possible false alarm in color i for a single inversion target is*

$$\frac{1}{s}\frac{\overline{f}_{i+2}}{\overline{f}_i}\overline{f}_s + \left(\frac{\sum_{k=1}^{i}\overline{F}_{cr,k}}{s}\right)\left(1 - \frac{\overline{f}_{s+1}}{\overline{f}_i}\frac{\overline{f}_{s+2}}{\overline{f}_{i+1}}\right) \tag{6.25}$$

Proof. We refer the interested reader to lemmas 7, 8 and 9 in [48]

□

Lemma 6.8. *The cost of sorting out false alarms during the on-line phase of the perfect fuzzy rainbow tradeoff attack is expected to be*

$$T_{fp}^{D} = tlDs\sum_{i=1}^{s}(1-\overline{F}_{ps}^{D})^{(\sum_{k=i+1}^{s}\overline{F}_{cr,k})/s\overline{F}_{cr}} \times \left\{\frac{1}{s}\frac{\overline{f}_{i+2}}{\overline{f}_i}\overline{f}_s + \left(\frac{\sum_{k=1}^{i}\overline{F}_{cr,k}}{s}\right)\left(1 - \frac{\overline{f}_{s+1}}{\overline{f}_i}\frac{\overline{f}_{s+2}}{\overline{f}_{i+1}}\right)\right\} \tag{6.26}$$

Proof. The stated result follows directly from equation (6.25) an Lemma 6.5 □

Proposition 6.9. *The online cost of the perfect fuzzy rainbow tradeoff can be approximated by the following equation*

$$T = tlDs\sum_{i=1}^{s}(1-\overline{F}_{ps}^{D})^{(\sum_{k=i+1}^{s}\overline{F}_{cr,k})/s\overline{F}_{cr}} \times \left\{\frac{(s-i+1)}{s} + \frac{1}{s}\frac{\overline{f}_{i+2}}{\overline{f}_i}\overline{f}_s + \left(\frac{\sum_{k=1}^{i}\overline{F}_{cr,k}}{s}\right)\left(1 - \frac{\overline{f}_{s+1}}{\overline{f}_i}\frac{\overline{f}_{s+2}}{\overline{f}_{i+1}}\right)\right\} \tag{6.27}$$

Proof. The online cost is the sum of the cost of generating the on-line chains (proposition 6.6) and the cost of sorting out the false alarms (lemma 6.8). The stated result in (6.27) is just the addition of both results. □

We can now proceed to show the tradeoff curve for the perfect table fuzzy rainbow tradeoff.

Theorem 6.10. *The time memory tradeoff curve for the perfect table fuzzy rainbow tradeoff with D inversion targets is $TM^2D^2 = \overline{F}_{tc}^{D} N^2$ where the tradeoff coefficient $\overline{F}_{tc}^{D}$ is*

$$\overline{F}_{tc}^{D} = \overline{F}_{msc}^{2} \left(\frac{lD}{t}\right)^3 \frac{1}{s} \sum_{i=1}^{s} (1 - \overline{F}_{ps}^{D})^{(\sum_{k=i+1}^{s} \overline{F}_{cr,k})/s\overline{F}_{cr}} \times \left\{ \frac{(s-i+1)}{s} + \frac{1}{s} \frac{\overline{f}_{i+2}}{\overline{f}_i} \overline{f}_s + \left(\frac{\sum_{k=1}^{i} \overline{F}_{cr,k}}{s} \right) \left(1 - \frac{\overline{f}_{s+1}}{\overline{f}_i} \frac{\overline{f}_{s+2}}{\overline{f}_{i+1}} \right) \right\} \quad (6.28)$$

Proof. Remembering that $M = ml$ and $mt^2 s = \overline{F}_{msc} N$, the stated equality can be easily verified substituting into equation (6.27) □

One interesting observation is that the value of $\overline{F}_{tc}^{D}$ does not depend on the individual values of l, D and $1/t$, only on the value of lD/t. If we keep this value constant we can vary the parameters getting the same tradeoff.

As a corollary to lemma 6.5, the number of table lookups is calculated as

$$lD \sum_{i=1}^{s} (1 - \overline{F}_{ps}^{D})^{(\sum_{k=i+1}^{s} \overline{F}_{cr,k}/s\overline{F}_{cr})} \quad (6.29)$$

6.5.2. Effect of memory optimizations

The value M appearing on the tradeoff refers to the number of entries necessary on the table, and not the amount of memory necessary to store the M values, which may vary depending on the memory optimizations in use.

Ending point truncation

The only memory optimization that modifies the online cost is the ending-point truncation, as it increases the false alarm rate. Kim and Hong provide an approximate relation between the degree of truncation and the increase in online computation due to the increased false alarms, accurate enough to determine the number of truncated bits which cause a negligible increase in online computation. However their work ignores a term that introduces a significant error in the calculation. The following proposition improves Kim and Hong's result and extends it to the $D > 1$ case.

Proposition 6.11. *If the probability of two truncated randomly chosen DPs to be identical is $1/r$, then the expected extra invocations of the one-way function when employing ending-point truncation is*

$$tlD\frac{m}{r}\sum_{i=1}^{s}\left(1-\overline{F}_{ps}^{D}\right)^{(\sum_{k=i+1}^{s}\overline{F}_{cr,k})/(s\overline{F}_{cr})}\sum_{k=1}^{i}\overline{F}_{cr,k} \tag{6.30}$$

Proof. Lemma 6.5 gives us the probability for the online chains that start from color i to be generated as equation (6.20). The probability of each generated chain to cause a truncation-related alarm with any one of the truncated ending points is $1/r$, and there are m ending points, each of which could cause a collision.

To resolve a false alarm, we must reconstruct the chain starting from the corresponding starting point in color 1 and ending in a distinguished point in color i, and for each color the expected effort is $\overline{F}_{cr,x}t$. Each alarm will thus require $t(\overline{F}_{cr,1}\cdots\overline{F}_{cr,i})$ iterations of the one-way function to resolve. Equation (6.30) is a simple combination of the previous facts taking into account the l precomputation matrices and D inversion targets. □

Kim and Hong make the argument that the additional cost of resolving alarms induced by the ending point truncation can be suppressed to a negligible level by having the truncation retain "slightly more" than $\log m$ bits of information for each ending point. They do not give more detail, as they group all optimizations on a single equation, as we will do shortly.

Other memory optimizations

Kim and Hong assume that the memory optimizations described in section 5.5 are used, except for checkpoints. The starting points can be represented with $\lceil log_2(m_0)\rceil$ bits using consecutive starting points.

In the ending points, the bits that make them distinguished can be suppressed. Adding end-point truncation each ending point can be represented using $\log m + \delta$ bits. Finally, using the fact that tables are sorted on the end points we can use an index table that contains the most significant bits of each ending-point and a pointer to the corresponding entry that only holds the least significant bits of the ending points and the $\lceil log_2(m_0)\rceil$ bits of the starting point. The size of the index table depends on the implementation of the table and on how many bits are left to represent the ending point, but it will be usually much smaller than the fuzzy rainbow table, so we can just assume its effect is included in the value of ϵ in the following.

Taking into account all previous optimizations, and using the arguments appearing in [41] and [49], Kim and Hong conclude that each entry of each table can be recorded in $\log m_0 + \epsilon$ bits, where ϵ is an "small integer". They propose using ϵ between 5 and 8 as a reasonable choice. In section 7.2.4 we present some experimental results on the effect of ending point truncation which help make an informed decision on the value of ϵ to use.

6.5.3. Tradeoff Coefficient Adjustment

The value M appearing on the tradeoff refers to the number of entries necessary on the table, and not the memory size needed to store the tables, which may vary depending on the memory optimizations in use and the parameters chosen for the tradeoff. To take into consideration the number of bits per table entry Kim and Hong propose to define an adjusted tradeoff coefficient that takes into account the number of bits needed to store each table entry. The proposed adjustment is shown in equation (6.31). Using the results from the previous section, and equation (6.3), the adjusted tradeoff coefficient is presented in equation (6.32)

$$\overline{F}_{atc}^{D} = \left(\frac{3}{2logN}\right)^2 (number\quad of\quad bits\quad per\quad table\quad entry)^2 \overline{F}_{tc}^{D} \quad (6.31)$$

$$\overline{F}_{atc}^{D} = \left(\frac{3}{2logN}\right)^2 (log(m_0)+\epsilon)^2\, \overline{F}_{tc}^{D} = \left(\frac{3}{2logN}\right)^2 \left(log\frac{2}{2-\overline{F}_{msc}} + log(m) + \epsilon\right)^2 \overline{F}_{tc}^{D} \quad (6.32)$$

Where $\log(2/(2-\overline{F}_{msc})) + log(m)$ is the number of bits needed to store the starting points, ϵ is the number of bits kept from the ending points in the ending point optimization (including truncation), and $(3/(2\log N))^2$ is simply a scaling factor. ϵ takes into consideration both the effect of ending point truncation, and the number of LSB bits left when using index tables. ϵ is chosen such that the truncation does not increase significantly the on-line computation; in the paper it is shown that 5 to 8 bits is enough to make the extra calculations due to end-point truncation much smaller than the on-line calculations. This adjustment was proposed by Kim and Hong for the $D = 1$ case, but we can use exactly the same adjustment as none of the memory optimizations depend on the number of inversion targets.

The adjusted tradeoff coefficient is useful both from a theoretical standpoint, to compare the algorithm with other algorithms, and from a practical standpoint as $\overline{F}_{atc}^{D}$ only depends on lD/t and not on the individual values of l, D and t so it allows the practitioner an easier choice of parameters.

Chapter 7

Experimental validation of the results from the previous chapter

In this chapter we will perform an empirical validation of the results from chapter 6 on a problem with a small space size of $N = 2^{40}$, and show a possible way to use the previous results to choose the parameters for a perfect fuzzy rainbow table TMDTO.

In chapter 8 we will use these results to calculate the expected effort to mount an TMDTO ciphertext only attack against A5/1 using the function to invert h_c calculated in chapter 4. For this chapter we build a reduced problem with space size $N = 2^{40}$ by modifying function h_c from chapter 4 to work on 40 bit values. The (arbitrary) choice of $N = 2^{40}$ is simply based on the available computational capacity. So the stated problem is to invert the modified h function using a perfect fuzzy rainbow table tradeoff. We will use this problem to check the validity of the previous results through experiments.
The distinguishing property we will use is asking the value to be less than a constant te, that is x will be a DP if $x < te$. This means the expected length of each sub-chain is N/te.
Throughout this chapter we will use software written to calculate the tables necessary for the TMDTO and then use those tables to perform the inversion both on standard Intel CPUs and on Nvidia CUDA GPUs available on the cluster infrastructure of our college. For details see appendix F.

7.1. Step function

We choose to work with a modification of the one-way function h_c presented in section 4.1.1, which we will call h throughout this chapter. The modification from h_c to h consists in setting the most significant 24 bits to a fixed value (which we can thus remove), and reducing the domain of the function to those values which have the same fixed bits in the most significant bits.

We also need to choose reduction functions that do not change the 24 most significant bits. We will use as reduction functions the exclusive OR with a constant

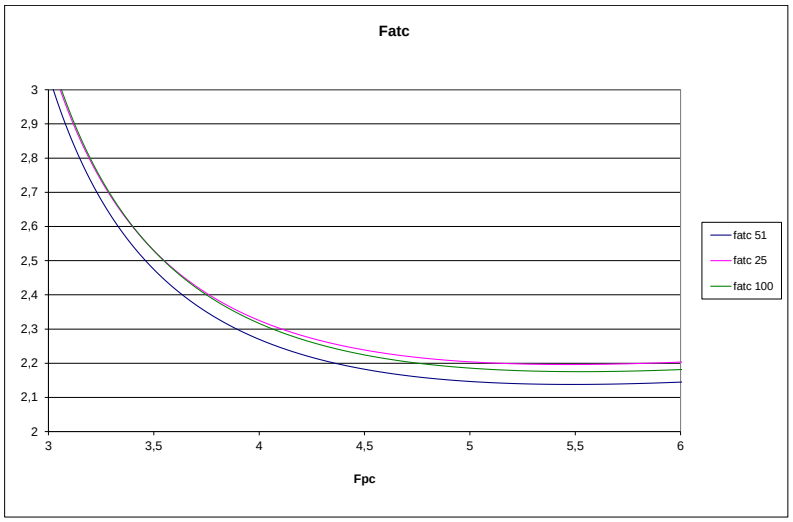

Figure 7.1: Fatc vs fpc for $N = 2^{39}$, $\overline{F}_{ps} = 0.9$

having the 24 most significant bits as 0 (different for each table and each color)

7.2. Validation for $D = 1$

For $D = 1$ our results in chapter 6 coincide with those in [48], so we will drop the D subscript in all $\overline{F}$ parameters.

7.2.1. Reproducing Kim and Hong's results

We will be using Kim and Hong's results both to calculate the precomputation and online effort given the parameters of the tradeoff, and to select the tradeoff parameters given some objective. To make sure we understood their formulas and implemented them correctly, we reproduced some of the calculations from [48]. In this section we reproduce just a sample of the results.

In figure 7.1 we can see how $\overline{F}_{atc}$ changes as a function of $\overline{F}_{pc}$ for different s values. It can be seen that the calculated plot agrees with figure 1 in [48] for success probability $\overline{F}_{ps} = 0.9$ and $\log m_* + \epsilon = 21$, taking $N = 2^{39}$.

We also recalculated Table 1 in the paper, obtaining consistent values (see appendix E)

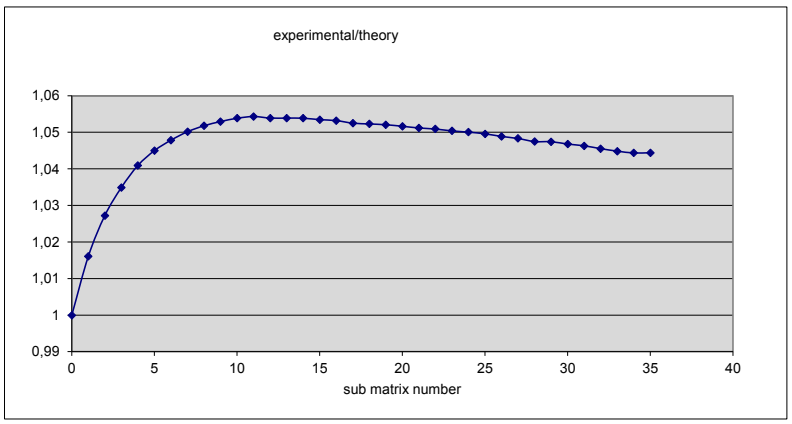

Figure 7.2: Number of colour boundary points - theory vs experimental - parameter set 1

7.2.2. Sample application to our reduced h function

First checks

As a first test of the applicability of the fuzzy rainbow table TMTO to function h as defined in section 7.1, we compare the expected number of boundary points after each color with the experimental results. The number of unique ending points for each color can be calculated iteratively from equation (6.1), or the approximation in equation (6.2). We use the same parameters as the appendix to [48] so that we can make a comparison with their results. m_0 is calculated from m using equation (6.2), and several tables are calculated storing the number of unique boundary points at each color. We average the results over 100 tables. The parameters are shown in the following table:

Property	parameter set 1	parameter set 2
m (S)	3161	4916
N	2^{40}	2^{40}
Expected section length (t)	2^{12}	2^{11}
Calculated $\overline{F}_{msc}$	1.6882	1.5002
Calculated m_0	20273	19673

Figures 7.2 and 7.3 show the quotient of the experimental average number of boundary points after each color and the calculated value according to (6.2). The maximum difference is less than 6% for parameter set 1, while it is less than 1.5% for parameter set 2. The results match those from the paper.

If instead of equation (6.2) we use equation (6.1) to estimate the parameters the result improves noticeably, for both parameter sets the difference between the experimental average and the expected value according to (6.1) is less than 0.1%. This implies that we may work with the approximate equation if an error of a few percent is acceptable, but should use equation (6.1) if we want the extra precision.

In figure 7.4 we see the distribution of table sizes for parameter set 1 at the last colour boundary in the experimental results. The average value is 3301, and the standard deviation is 30.4.

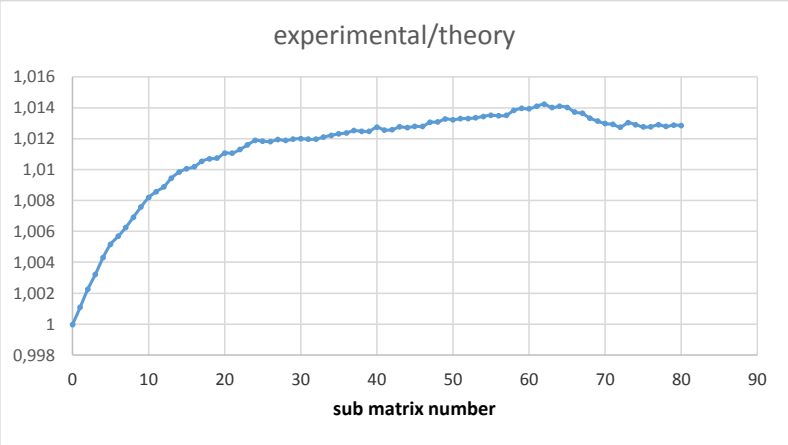

Figure 7.3: Number of colour boundary points - theory vs experimental - parameter set 2

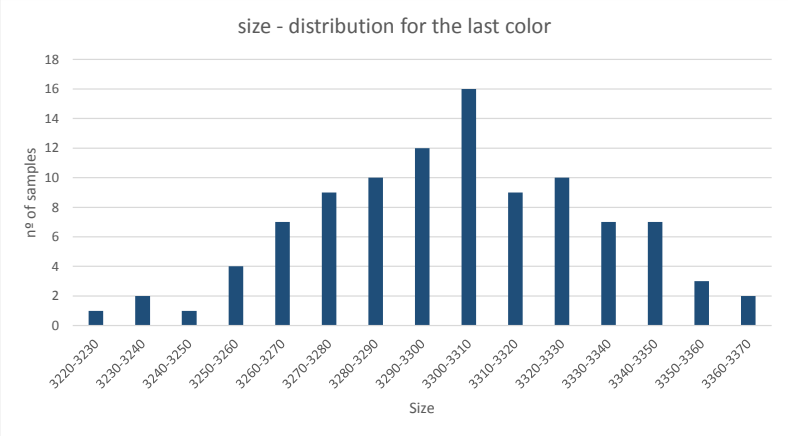

Figure 7.4: distribution of table size - parameter set 1

Applying the fuzzy rainbow table TMTO

As a first test we picked some parameter values, shown in the following table, calculated the TMTO tables for function h, and then used the tables to implement the TMTO attack for several targets to validate the predicted values against the experimental values.

For this test we use the table 1 on the appendix to [48] which we recalculated in appendix E as part of our validations. This table provides us the values of s and $\overline{F}_{msc}$ that minimise $\overline{F}_{atc,s}$ given the desired success probability $\overline{F}_{ps}$ and the values of $m_* = m/s$ and ϵ. For $N = 2^{40}$, a reasonable choice is $m/s = 2^{13}$, which using $\epsilon = 8$ leads to $\log m/s + \epsilon = 21$. For this value, and choosing the target success probability $\overline{F}_{ps} = 0.9$, we see that the values that minimize $\overline{F}_{atc}$ are $s = 50$ and $\overline{F}_{msc} = 1.7167$. Using the previous values we get:

- $m = 409600$
- $t = 303.9$

Property	set 1	set 2
Nº of colors (S)	50	50
Nº of initial values per table (m_0)	2891634	2891634
Expected section length (t)	303.9	231.7
Nº of tables (l)	415	250

Table 7.1: Two parameter sets for TMTO validation

- $l = 415.2$

We cannot calculate a non integer number of tables, so we use $l = 415$.

We added another value set with different values for t and l, $t = 231.7$ and $l = 250$, keeping the same $m0$ value.

For each parameter set we calculated a set of tables and used them to find the inverse of h for 10000 targets. The targets were the images of 10000 values chosen at random, and we only consider an attack successful if the preimage found coincides with the original value.

The expected values were calculated using the equations in the previous chapter. The expected and average calculated values for set 1 are shown in the following table:

Parameter	Estimate	Empirical value	test/theory
m	408925	422305	1.033
$\overline{F}_{msc}$	1.72	1.77	1.033
$\overline{F}_{pc}$	5.49	5.67	1.033
$\overline{F}_{ps}$	0.898	0.904	1.006
Precomputation	6.0345×10^{12}	6.2331×10^{12}	1.033
Online complexity (T)	70767834	67910062	0.960

The expected and average calculated values for set 2 are:

Parameter	Estimate	empirical value	test/theory
m	638375	657786	1.023
$\overline{F}_{msc}$	1.558	1.594	1.023
$\overline{F}_{pc}$	3.32	3.38	1.019
$\overline{F}_{ps}$	0.806	0.815	1.011
Precomputation	3.6505×10^{12}	3.7210×10^{12}	1.019
Online complexity (T)	42867238	41427521	0.966

As can be seen in both tests the estimate according to the equations in the previous chapter is close to the experimental values.

7.2.3. Comparing the accuracy of the estimations

The example results in the previous section show that the difference between the parameters calculated using the approximate formulas in the previous chapter

and the empirical values is small enough at least in the sample cases to be used to estimate the values in a practical application. However, the first results in section 7.2.2 show that at least some approximations done in deriving the formulas introduce noticeable errors. Comparing the results when using equations (6.1) and (6.2) we see equation (6.1) allows us to better estimate the number of remaining chains at each step in the calculation of the tables, which in turn can improve the calculation of the remaining parameters. The flip side is having iterative formulas that are more difficult to operate with, and depend on more parameters, thus making it more difficult to use them to calculate the parameters to be used.
Using equation (6.1) for the values in Example 2, the estimated m value is 653256, which is much closer to the experimental average value in table 7.2.2 (less than 0.1% difference).

The number of iterations of function h in the precomputation phase can be better estimated as $(m_0 + \cdots m_{s-1})tl$, which using the iterative formula to calculate m_i yields 3.7224×10^{12} iterations, which again has less than 0.1% difference with the experimental value.

For the success probability $\overline{F}_{ps}$ the calculation can be improved by observing in the demonstration of Proposition 4 and lemma 3 in [48] that in equation (16) and (21) we can use the improved values for m_i to obtain a better estimation. Using this the calculated value for Example 2 is 0.813, much closer to the experimental value (less than 0.3% difference).

Finally, the estimation of the on-line effort can be also improved by substituting the approximations for $m0$, $\overline{F}_{cr}$ and $\overline{F}_{cr,i}$ with the more accurate formulas. Applying this to the parameters for Example 2 we get $T = 41642714$, again a much better match for the experimental result (less than 0.6% difference).
For the values in example 1 a similar result is found.

The main takeaway from this subsection is that when using the approximate equations to estimate the parameters for the tradeoff, the main deviations are caused by the approximations made to arrive to the closed formulas, and better estimations can be made if the more exact equations are used. In a practical situation it may be useful to first use the approximate equations to choose the values of the parameters for the TMTO, and then check the resulting parameters using the results from this section.

We will mostly use the approximate formulas from now on unless explicitly noticed.

7.2.4. Effect of the ending-point truncation

In section 6.5.2 we calculated the extra invocations due to the ending point truncation, and claimed that our estimation is better than the one in Kim and Hong's paper. Both our equation and Kim and Hong's depend on the value of the probability $1/r$ of two truncated randomly chosen DPs to be identical, so to test the equations we need to calculate r for our examples.

Given the chosen distinguished property, (that is, to be a DP a value v must

Trnc bits	$r(\times 10^6)$	extra inv. (practical)	extra inv. Kim	extra inv. eq (6.30)	prac/(Kim)	prac/eq (6.30)
20	1.049	33200611	22123036	33347167	1.501	0.995
21	2.098	16596926	11058312	16668751	1.501	0.995
22	4.199	8293430	5525951	8329544	1.501	0.995
23	8.408	4139710	2759771	4159942	1.500	0.995
24	16.855	2064485	1376684	2075145	1.500	0.995
25	33.868	1028028	685140	1032747	1.500	0.995
26	68.375	509402	339369	511547	1.501	0.996
27	139.38	249587	166483	250948	1.499	0.995
28	289.52	120035	80149	120812	1.498	0.994
29	627.29	55512	36992	55759	1.501	0.995
30	1483.56	23578	15641	23577	1.507	1.000
31	4450.64	7763	5214	7859	1.489	0.988

Table 7.2: Extra invocations with truncation for parameter set 1

be $v < te$), the $\lfloor log_2(N/te) \rfloor$ most significant bits will be zero and don't need to be stored. It is more convenient to truncate on the most significant bits, as if N/te is not integer then the most significant remaining bit has a bias, so the first bit we truncate has less than 1 bit of information.

If we leave $b < \lceil log_2(te) \rceil$ bits after truncation, to calculate the probability for two truncated endpoints to be identical, assuming the endpoints are uniformly distributed between 0 and te, we observe that if $te = a2^b + c$, then in the interval $[0, te)$ there are c values which repeat $a + 1$ times and $2^b - c$ values which repeat a times. The probability for two truncated endpoints x, y to collide is thus

$$P(x{=}y/y \mod 2^b{<}c).P(y \mod 2^b{<}c){+}P(x{=}y/y \mod 2^b \geq c).P(y \mod 2^b \geq c)$$
$$= \frac{a}{te-1} \times \frac{(a+1)c}{te} + \frac{(a-1)}{te-1} \times \frac{a(2^b-c)}{te} \quad (7.1)$$

If $a >> 1$ and $te >> 1$, this can be approximated by

$$\frac{a^2 2^b}{te^2} \quad (7.2)$$

Using the previous value for r we can calculate the expected extra invocations of the step function for the values chosen for the previous parameter sets using both Kim and Hong's result and our improved equation (6.30), and compare them with the experimental values. For the values in set 1, $t = 303.9$, so the most significant 8 bits are zero. In Table 7.2 we can see the calculated and experimental extra invocations for several truncation values, where we can see equation (6.30) gives a good match with the practical results. A similar result can be seen in table (7.3) for parameter set 2.

Trnc bits	$r(\times 10^6)$	extra inv. (practical)	extra inv. Kim	extra inv. eq (6.30)	prac/(Kim)	prac/eq (6.30)
21	2.0981	13895555	9202246	13953127	1.51	0.996
22	4.1980	6941145	4599089	6973480	1.509	0.995
23	8.4035	3466537	2297511	3483657	1.509	0.995
24	16.837	1730315	1146722	1738746	1.509	0.995
25	33.793	861446	571333	866297	1.508	0.994
26	68.069	427741	283640	430077	1.508	0.995
27	138.098	210971	139806	211985	1.509	0.995
28	284.32	102522	67906	102964	1.51	0.996
29	604.18	48215	31956	48454	1.509	0.995
30	1365.7	21358	14137	21435	1.511	0.996
31	3693.7	7823	5227	7926	1.497	0.987
32	25000.8	1157	772	1171	1.499	0.988

Table 7.3: Extra invocations with truncation for parameter set 2

	set 1 section length	set 1 total length	set 2 section	set 2 total
$\overline{F}_{ps}$	0.9038	0.9246	0.8155	0.8311
Te (on-line cost)	67910062	70347901	41427521	43257125

Table 7.4: Effect of using section length vs. total length

7.2.5. Effect of using the section length instead of total length

One of the assumptions we made, which is the same Kim and Hong made, is that when building the tables, if a collision is found in color i the chain with the longest DM_i segment is retained. This choice was made to be able to use previous results, and they give an informal justification in [48] to show that the effect of using the section length instead of the total length has a minor impact on the parameters of the tradeoff. We wont develop a theoretical comparison, but to appreciate the effect of this choice we calculated the tables for the previous parameter sets but changing the rule when a collision is found to retain the chain with the largest total length. The effect is to slightly increase both the success probability and the on-line effort. More tests should be done to check which criteria is best, but at least in this two cases the effect is small. The results are resumed in table 7.4.

7.2.6. Another practical scenario

In the previous section we restricted ourselves to use the values of s and $\overline{F}_{msc}$ precomputed by Kim and Hong, which were calculated so as to minimize $\overline{F}_{atc,s}$. In a real world scenario the attacker may prefer to trade some on-line cost for a decrease in precomputation cost, so we want to study the tradeoff between

precomputation effort and on-line cost.

For a realistic exercise, we assume we have a fixed memory size, $M = 2^{31}$ bytes, and want to choose an adequate set of parameters for a TMTO with a 90% success probability. We take $\epsilon = 8$ and ignore the effects of the ending-point truncation.

There are different possible mechanisms to find suitable parameters for the tradeoff. One simple way to choose the parameters is to use the parameters from appendix E. Estimating the number of bits per entry and searching the parameters in the table, one finds the value of s and $\overline{F}_{msc}$ that provide the minimum $\overline{F}_{atc}$ value, calculate all parameters and then verify that the estimated number of bits per entry is correct. However an interpolation must be made on the value of $m* = m \times s$, as using the parameters on the table does not allow us a precise choice of the memory used. This method is not very flexible, as we will get the point that minimizes $\overline{F}_{atc}$ irrespective of the precalculation cost, and we may be interested in trading a higher on-line cost for a lower precalculation. The parameters we get using this method are shown as "set 1" in table 7.5. To remain within the available memory M and use it in the best possible way, while having an integer number of bits per table entry, we had to do some trial and error to find the parameters.

We could also attempt to calculate the on-line cost versus the precomputation cost for the value of s on the table and varying $\overline{F}_{msc}$ and $m*$. However in practice the number of bits per table entry must be an integer, so the curve is not continuous due to the abrupt changes in the values when the number of bits per table entry increment, and some trial and error is necessary for each value which makes it less useful. Besides the value of s in the table is the one that minimizes $\overline{F}_{atc}$, but is not necessarily the best for other tradeoff values.

To give the practitioner the maximum flexibility we decided to show how the parameters can be chosen without imposing an a-priori relation, and created a spreadsheet that automates most of the calculations. We consider reasonable value ranges for s and l, which must be integer, and for each pair (s, l) and using the known parameters (N, M, $\overline{F}_{ps}$, ϵ) we calculate the rest of the parameters. The procedure is detailed in the appendices, section D.1. For each pair s and l we get a set of parameters and tradeoff coefficients, so we can make a choice of parameters adequate to our application. In figure 7.5 the on-line cost T is plotted versus the precomputation coefficient $\overline{F}_{pc}$ for a range of s and l values, showing a detail of the "interesting" part of the plot where the values seem useful for the practitioner. The minimum value of T corresponds to $\overline{F}_{pc} = 5.4313$, so larger values of $\overline{F}_{pc}$ do not seem useful. Also low values of $\overline{F}_{pc}$ quickly lead to a huge increase in T value, also making those points impractical.
One interesting observation is that we cannot improve the on-line time by incrementing the precalculation effort beyond the point where the minimum is found. The only way to improve the on-line time is to increase the available memory.

We can also see there is a range of $\overline{F}_{pc}$ values for which the on-line effort varies little, so it may be advantageous to trade a slight increase in on-line time for a lower precomputation time. We should also take into consideration that there is an error in all approximations we made, so small differences in the performance

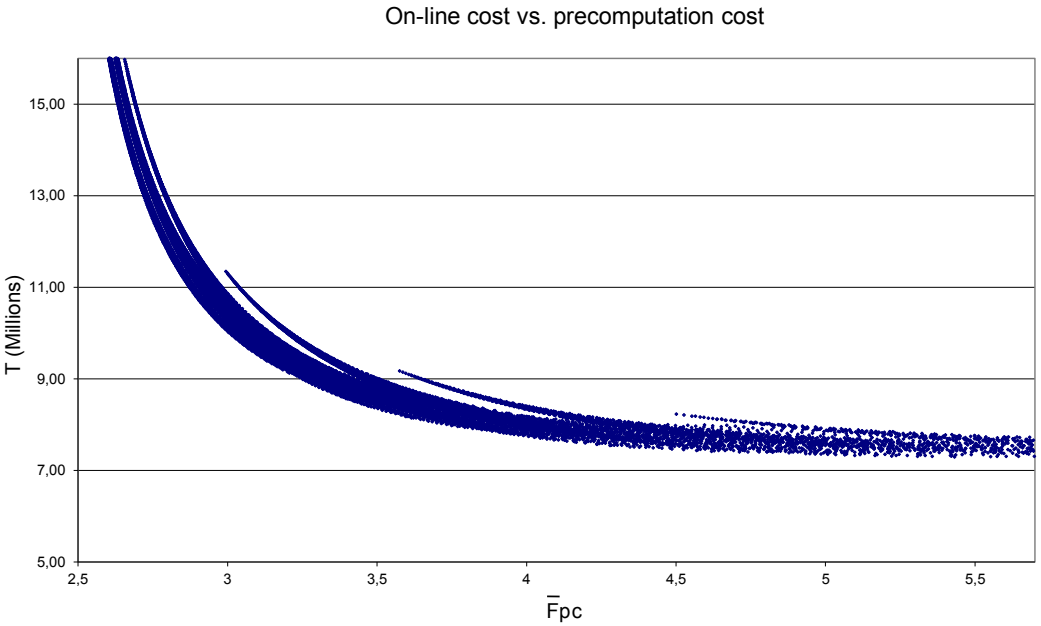

Figure 7.5: On-line cost vs. precomputation cost

	set 1	set 2 (minimum T)	set 3
s	56	61	39
m	4331859	4732746	3033169
m_0	33554293	33437234	16560113
l	119	110	177
t	88.85	80,96	123.4413
$\overline{F}_{pc}$	6.6519	5.4312	5.005
T	7365450	7296159	7389099

Table 7.5: Parameter sets calculated for $M = 2^{40}$

of the calculated parameters do not necessarily translate into a better practical tradeoff.

The parameters for the minimum T value found using this mechanism are shown as set 2 in table 7.5, and in set 3 we added another set of values, with a lower $\overline{F}_{pc}$ and a little increase in T. We implemented the TMTO for this last set of parameters, and the experimental results averaged over 10000 inversions can be compared to the theoretical estimations in table 7.6.

7.3. Calculations for $D > 1$

For $D > 1$ we want to validate the results in the previous chapter. We will mostly do the same validations as the previous section for this new environment.

We will not do an exhaustive test for varying D values, leaving a thorough

	Expected values	experimental values
s	39	39
m	3033169	3127614
m_0	16560113	16560113
l	175	175
t	123.4413	123.44
Total precalculation cost	5.503×10^{15}	5.688×10^{15}
$\overline{F}_{pc}$	5.005	5.173
T	7389099	7302665
$\overline{F}_{ps}$	0.9	0.905

Table 7.6: Calculated and estimated values for set 3

		estimate (eq. (6.15))	estimate (eq. (6.14))	experimental average
$D = 2$	$\overline{F}_{ps}^{D}$	0.96288	0.966	0.965
	T	41510818	40364661	39507210
$D = 4$	$\overline{F}_{ps}^{D}$	0.99862	0.9989	0.9987
	T	29951449	29350087	28171683
$D = 8$	$\overline{F}_{ps}^{D}$	0.999998	0.99998	1
	T	19335065	19200797	18712476

Table 7.7: Calculated and estimated values for set 2 for different D values

study of the influence of D on the tradeoff for future work, instead sampling several combinations with various parameter values.

7.3.1. First validation samples

Observing the results of the previous chapter, it is obvious that the same tables used for $D = 1$ can be used for the $D > 1$ case (although they are probably not optimal). So we use the same tables we already calculated in section 7.2.2 corresponding to parameter set 2 in table 7.1, and only change the on-line phase to account for the number of targets D. The results, averaged over 10000 attempts, are shown in table 7.7 and compared to the estimated values according to equations (6.15) and (6.14). Just as in the single target case, the best approximation in our samples is equation (6.14).

To test larger D values we start by using a subset of the tables from the same parameter set. As an example of the tests we made in table 7.8 is the result of taking $D = 20$ and $l = 17$, where we can see there is a good match between theory and practice.

	estimate (equation (6.15))	estimate (equation (6.14))	experimental average
$\overline{F}_{ps}^{D}$	0.8935	0.900	0.9047
T	44001915	41656211	41897462

Table 7.8: Calculated and estimated values for a subset of set 2. $D = 20$, $l = 17$

		estimate	experimental average
$l = 1$	$\overline{F}_{ps}^{D}$	0.9	0.8998
	$\overline{F}_{pc}$	0.08818	0.0910
	T	2051783	2058091
$l = 2$	$\overline{F}_{ps}^{D}$	0.9	
	$\overline{F}_{pc}$	0,1088	
	T	2138271	

Table 7.9: Calculated and estimated values. $D = 64$, $l = 1$ and $l = 2$

7.3.2. Finding parameters for different D values

To estimate the parameter and tradeoff constants we used the same method as in section 7.2.6. For larger D values we expect l to decrease. When we increased D to $D = 64$, even decreasing the available memory to $M = 2^{26}$ the minimum on-line effort happens when $l = 1$ (and $s = 55$). In table 7.9 we see the parameters for the minimum on-line effort. We also added the parameters for the minimum on-line effort with $l = 2$, where we see both precomputation and on-line effort are worse. However the difference is not large and it may be beneficial if it helps parallelize the on-line effort.

We added another pair of parameter sets for $D = 64$ but lowering the available memory to $M = 2^{25}$ bytes. The parameter set for $l = 2$ minimizes the on-line effort, while the parameter set for $l = 1$ was chosen to show that we can substantially decrease the precomputation time with a modest increase in on-line effort. The results are in table 7.10

Finally we added an example with $D = 16384$ and $M = 2^{20}$ bytes, with desired success probability $\overline{F}_{ps}^{D} = 0.9$. The first observation is that all parameter sets have very similar $\overline{F}_{pc}$ values. This seems reasonable taking into consideration that the coverage of the matrices does not need to be high, which means collisions inside the matrices are rare and all parameter sets have almost the same coverage. The second observation is that with this choice of D and M the values of s and l that minimize the on-line effort T are $s = 2$ and $l = 1$. However, we must be careful, as such a low value for s is outside the range of values for which the approximations in the previous chapter are valid. However at least in this case we can see in table 7.11 that the estimated and averaged experimental values are very close.

		estimate	experimental average
$l = 1$	$\overline{F}_{ps}^{D}$	0.9	0.89
	$\overline{F}_{pc}$	0.04999	0.0500
	T	9480697	9857608
$l = 2$	$\overline{F}_{ps}^{D}$	0.9	0.903
	$\overline{F}_{pc}$	0.08811	0.0912
	T	7339199	7268391

Table 7.10: Calculated and estimated values. $D = 64$, $M = 2^{25}$, $l = 1$ and $l = 2$

	estimate	experimental average
$\overline{F}_{ps}^{D}$	0.9	0.8990
$\overline{F}_{pc}$	0.000143043	0.000142643
T	6959689	6956052

Table 7.11: Calculated and estimated values for $D = 16384$, $l = 1$, $s = 2$

7.3.3. Some initial cualitative observations

The first observation is that when D is incremented we expect the needed coverage of the matrices to decrease. This in turn decreases the expected number of collisions, thus decreasing the number of sub-matrices DM_i necessary. As D increases we expect most reasonable parameter sets to require a similar precomputation effort for the same success probability, easing the choice of parameters.

In an extreme case for very large D it may be advantageous to use a single DM_i matrix, which is nothing more than a perfect classic Hellman matrix with distinguished points. Whether there are values of D for which a perfect Hellman matrix obtains a better tradeoff than the prefect fuzzy rainbow matrix will not be explored in this work.

The second observation is that most experiments agree substantially with the estimated theoretical values. We made no effort to estimate the statistical significance of our test's results, but most experimental results in this section are the average of at least 10000 runs of the test, which should give us a decent confidence in the results.

Lastly, we ignored the effect of disk searches in our examples. This is an open topic which can be studied in a future work, as the number of searches increases with D and may become non-negligible for large values of D.

This page intentionally left blank

Chapter 8

Applying the fuzzy rainbow table TMDTO to the ciphertext only attack against A5/1

In this chapter we will apply the results from the previous chapters to calculate the cost for an attacker to implement a ciphertext only attack against A5/1 using a TMDTO. The function to invert is the h_c function described in chapter 4. Finding a preimage of h_c implies finding the internal state of A5/1 leading to the captured ciphertext, which in turn as seen in section 3.1.1 allows the attacker to find the key used for encryption.

Before starting the precomputation phase of any of the TMDTOs, the attacker needs to fix the parameters for the TMDTO. The parameters will depend on the resources available for the on-line and precomputation phase, and the desired success probability.
In the case of A5/1, the state space has size $N = 2^{64}$. The amount of available ciphertext for analysis (D) varies with the amount of captured ciphertext, and the tables should be calculated taking into consideration the expected use of the attack and the available resources.

To calculate the parameters for the tradeoff we arbitrarily chose three possible scenarios which vary on the resources available to the attacker.

The first scenario presents the most stringent conditions on the attacker: requiring high success rates for very short calls. This means that a high success probability for $D \approx 1$ is needed. A second set of parameters is calculated for an attacker with less stringent requirements, namely that a high success rate is attained for long calls, where the attacker obtains a large number of ciphertexts to attempt inversion. To make the calculations we assume the attacker aims for a 90% success rate for $D \approx 500$, which implies a call longer than 8 minutes.

Finally an scenario with large D may be adequate for a demonstration. We will calculate a set of parameters adequate for the computational capacity available at most universities, taking as example the capacity available to us on our college's infrastructure.

Parameter	Value
$\overline{F}_{ps}^{D}$	0.9
Memory in bytes (M)	10^{15}
s	78
l	182
m	8.6189×10^{11}
$\overline{F}_{msc}$	1.803
$\overline{F}_{pc}$	1.226
Precomputation	$2,262 \times 10^{19}$
Online complexity (T)	862912672

Table 8.1: Parameter set for an attack with $D \approx 5$

As a point of comparison, for the attack implemented by Nohl et al in the known plaintext scenario, $D = 408$ and $N \approx 2^{61}$ (N is less than 2^{64} due to some optimizations found by Nohl et al).

In this work we will ignore the time to search the ending points on the tables assuming it is negligible. This must be checked if the attack is to be implemented.

8.1. Scenario 1

For this scenario we assume a powerful attacker who wants to have a high success probability even having a small amount of ciphertext for analysis. Let's assume he wants a success probability of 90% for $D = 5$, and has available 1 petabyte of storage (10^{15} bytes, which is approximately $2^{49,8}$ bytes), distributed in several machines, and several modern CUDA or AMD GPUs to do the calculations. We use the same spreadsheets as in the previous chapter to calculate a reasonable parameter set the attacker might use, with the objective of minimizing the on-line time. A possible set of parameters is shown in table 8.1

Some comments about the feasibility of the tradeoff.

For the precomputation phase using the Nvidia Tesla C1060 GPU cards available at our college's computation facilities, which are old cards rated for 622 Giga FLOPs (GFLOPs) and similar number of integer computations, our implementation allows us to do approximately 2^{26} iterations of the step function per second. Extrapolating those results to modern Nvidia GPU accelerators, like the V100 GPU Accelerator rated for 14000 GFLOPs it seems reasonable to expect at least 2^{30} iterations of the step function per second per GPU, possibly more if the programming is improved. Taking 2^{30} as a conservative estimate, the attacker needs approximately 244000 GPU days to complete the precomputation phase, or little more than 8 months if he uses 1000 top of the line GPU cards, which is feasible for resourceful attackers like some government agencies.

For the on-line phase we cannot expect the same performance measured in iterations per second as in the precomputation phase, as the parallelism is lower.

Parameter	minimum T	set 1	set 2
$\overline{F}_{ps}^{D}$	0.9	0.9	0.9
Memory in bytes (M)	10^{13}	10^{13}	10^{13}
s	142	154	97
l	1	1	2
m	1.538×10^{12}	1.569×10^{12}	8.163×10^{11}
$\overline{F}_{msc}$	1.637	1.619	1.247
$\overline{F}_{pc}$	0.0124	0.00972	0.00729
Precomputation	2.29×10^{17}	1.792×10^{17}	1.345×10^{17}
Online complexity (T)	866000834	889109707	1002080222

Table 8.2: Parameter set for an attack with $D \approx 500$

Using 2^{24} as an estimate of the iterations per second per machine, the attacker needs approximately 51 machine/seconds on average for the on-line phase. Having 91 dedicated machines, each one storing and processing two of the $l = 182$ tables allows the average on-line time to decrease below 1 second, and allows the disk searches to be spread among all machines easing the random access restrictions of the hard disks.

8.2. Scenario 2. $D \approx 500$

For this scenario we assume the available storage is 10 TB, and assume that instead of searching for the absolute minimum on-line effort the attacker prefers to trade some on-line efficiency for a shorter precomputation time. Shown in table 8.2 are three possible choices of parameters, the parameters that minimize the on-line cost and two sets with lower precomputation cost.

Taking as example the parameter set 2, for the precomputation phase the effort is approximately 200 times lower than in scenario 1, which means 16 GPU cards working for a year can calculate the tables. Taking as a price point the cost of leasing a *p2.16xlarge* instance in Amazon EC2, which offers 16 K40 GPUs and can be leased for $80354 a year as of August 2017, it seems reasonable to assume the precomputation phase can be carried out by any institution willing to spend between $100000 and $200000 and wait a year to calculate the tables.

The on-line complexity is of the same order as the previous scenario, so assuming the use of one or two machines for the on-line phase the expected on-line time is of the order of 1 minute.

8.3. Scenario 3

As a demonstration of the usefulness of the method we can choose a set of parameters for which the cost is within reach of our computational capacity and

Parameter	Estimate	Empirical value	test/theory
$\overline{F}_{ps}^{D}$	0.9	0.825	0.917
Memory in bytes (M)	2.10×10^{10}	2.1×10^{10}	0.999999
m	4200000000	4199994669	0.999999
$\overline{F}_{pc}$	1.15297×10^{-06}	1.15296×10^{-06}	0.99999
Precomputation	$2,12686 \times 10^{13}$	$2,12686 \times 10^{13}$	0.99999
Online complexity (T)	9538510255	9496681501	0.995

Table 8.3: Parameter set for sample application 1

test it with synthetic data. This section is only intended as a proof of concept of the method, as the amount of captured ciphertext necessary for the attack is equivalent to several days of voice calls. We will not exhaustively study the parameters for this case.

Our implementation of the algorithms allow us to do approximately 2^{26} iterations of the step function on each Tesla C1060 card, and we have 4 such cards.

We will have a large D value, which means the tables we will calculate do not need to represent a large portion of the search space, so we will have a low probability of chain merge and a relatively small amount of collisions inside the tables. If we aim for a precomputation time between one day and one week, we can do approximately between 2^{44} and 2^{47} iterations of the step function, and assuming few collisions the coverage will be between $2^{-20}N$ and $2^{-17}N$. As a gross approximation, to have a 90% probability of success we expect D to be such that $(1-2^{-20})^D \approx 0.9$ for the first case and $(1-2^{-17})^D \approx 0.9$ for the later. We chose $D = 2000000$ and $D = 300000$ and calculated two sets of parameters and two corresponding sets of tables.

For both sample demonstrations we arbitrarily restricted the available storage to $21GB$ after truncation. We did not implement the ending point truncation nor index tables when storing the values of the TMDTO, so the effective storage used is larger.

For the first demonstration we choose $D = 2000000$. The calculated values for the tradeoff are given in the first column of table 8.3. The values that minimize T are $s = 2$, $l = 1$ and $t = 2529.2$. We are outside the values of s for which we know the results in chapter 6 are valid, so the error may be larger than in previous sections. We limit ourselves to show the practical results averaged over 475 inversions, and leave a more exhaustive investigation of the seemingly low success probability for future work.

For the second sample demonstration the choice was $D = 300000$. The calculated values for the tradeoff are given in the first column of table 8.3. The values that minimize T are $s = 4$, $l = 1$ and $t = 3723$. Again s is small, although a little bit larger than the previous sample. The results again show a reasonable match

Parameter	Estimate	Empirical value	test/theory
$\overline{F}_{ps}^{D}$	0.9	≈ 0.82	0.91
m	4097560976	4098688593	1,0003
Precomputation	$1,43119 \times 10^{14}$	$1,45247 \times 10^{14}$	1.01
Online complexity (T)	10361344003	13089479755	1.26

Table 8.4: Parameter set for sample application 2

for the precomputation cost, but show a lower than expected success probability and higher on-line cost.

This page intentionally left blank

Chapter 9

Applicability of the attack and countermeasures

9.1. Conditions for applying the TMDTO attack against A5/1

One obvious condition for applying the previous attack is that the communication must be using A5/1. This is the case in many networks today, and it is not likely that the providers using A5/1 today will invest in fielding A5/3 in the future as GSM is a legacy protocol and spending on network upgrades is not a priority for operators.

Another practical consideration is whether the attack by Karsten Nohl et al [59] [52] can still be applied. If a pair known plaintext - ciphertext can be found then Nohl's attack requires less effort both in the precomputation and in the on-line phase than our attack.

To apply the described attack, the attacker should choose adequate parameters for the perfect fuzzy rainbow table tradeoff depending on the available computational power and storage available and calculate the corresponding tables.

After the tables are calculated the attacker needs to capture enough ciphertext from the target communication to have a good probability of inversion given the chosen table parameters. This is not an easy task given that GSM uses channel hoping from frame to frame, and the channel of the target communication is not a priori known. In this work we will not delve into the difficulties of capturing such ciphertext, one possible cheap solution is presented in [45]. One important consideration about the captured ciphertext is that any error bit in the captured ciphertext value (consisting on two SACCH frames) makes it unusable, as the attack makes use of the redundancy provided by the error detecting and correcting codes. As the attack uses the downlink channel, being close to the base station is a must to decrease the error probability. One possible improvement is to investigate if the extra known bits together with the error correcting codes could enable an attacker to tolerate some bit errors.

9.2. Countermeasures

One obvious countermeasure to protect against this attack is to use the A5/3 algorithm to protect the communication, or to stop using GSM and migrate all voice communications to UMTS (3G) or LTE (4G).

Another possible countermeasure is to randomly introduce a few bit errors into each SACCH message, trusting on the error correction codes to correct them. This can be safely applied unless the channel's error rate is too high, meaning it can be applied whenever the BTS detects the mobile equipment is close enough.

Chapter 10

Conclusions and future work

10.1. Conclusions

There are two main topics studied in this thesis. Our initial objective was to study the security in GSM and in particular the security of the A5 family of ciphers. During this study, when we decided to concentrate on the ciphertext only attacks, it became apparent that we needed to study the TMDTO algorithms available in the literature. The main results from this thesis are aligned with the study of those two subjects.

We have shown that a ciphertext only attack against algorithm A5/1 as used in GSM is feasible nowadays for a motivated attacker with enough resources, using the results in [9] and a modern TMDTO. We calculated the step function in [9], and calculated a possible set of parameters for a perfect fuzzy rainbow tradeoff implementing the attack for different scenarios. The necessary resources vary from very high (millions of dollars in 2017) if a short on-line time and high success probability for short calls are desired, to almost negligible if the available ciphertext is large (corresponding to several hours of communication available, or ciphertext from many simultaneous calls) and a long inversion time is not an issue, for example for a demonstration.

We implemented a demo attack with synthetic data under the assumption that the available ciphertext corresponds to several hours of communication, and showed that the attack works with parameters similar to those calculated theoretically.

We described a new step function based on the redundancy in the voice channel of a GSM call. A TMDTO implemented using this step function is costlier than using the redundancy in the SACCH control channel, but does not depend on any knowledge of the contents of the messages, only on the redundancy introduced by the error detection and correction codes.

Based on these results we can conclude that A5/1 can not be considered a secure protocol against a resourceful attacker taking into consideration current (as of 2017) computation and storage capacities. The best countermeasure against the

presented attacks is to move away from A5/1, either by moving to newer networks (UMTS, LTE) and deprecating GSM, or by implementing A5/3 in the network. We however present a possible countermeasure in chapter 9 that can be used to mitigate the attack risk.

We studied and briefly documented the TMTO and TMDTO attacks presented in the literature, and choose to work with the perfect fuzzy rainbow table time memory data tradeoff, which was shown in [48] to be the best available tradeoff for the single inversion target scenario.

The parameters of the perfect fuzzy rainbow time memory data tradeoff were calculated for the case in which several targets are available for inversion, thus extending the results in [48] to this new scenario.

10.2. Future work

The following ideas for future work were identified while working for this thesis but could not be pursued within the scope of this work.

Regarding GSM security and A5/1, we worked with synthetic data for the implementation of our demo attack against A5/1. Data from a test network or a live network could be used to further validate our work. That would entail capturing and decoding the raw GSM data and processing it. Tables with better coverage could be built if more time and computational resources were available.

One interesting research topic is to investigate whether the existing correlation attacks against A5/1, described in section 3.1.3, which work on a known plaintext attack scenario, can be extended to work on a cyphertext only attack scenario, using the redundancy in the signaling or voice channels to build sets of equations on which correlations can be found and exploited to determine equations on combinations of bits from the internal state.

Regarding our study of TMDTO algorithms, we calculated the parameters for the perfect fuzzy rainbow table tradeoff when the number of inversion targets D is greater than one, but under certain assumptions about the parameters, namely that the number of colors s is large enough so that the approximations in chapter 6 remain valid. Some of the results in later chapters suggest that when D is large low values of s can provide a better tradeoff, however we did not study the accuracy of the approximations under those circumstances.

No comparison was made with other TMDTO algorithms for the $D > 1$ case. Some experiments show that as D increases we get a better tradeoff with few or only one table and less colors. It may happen that for some D values a single Hellman table with DPs gives a better tradeoff. We leave that study as a topic for future work.

Another possible improvement in the calculations is to include the effect of the ending point truncation into the calculations. In section 7.2.4 we calculated the effect of the ending-point truncation optimization for our sample application, but only to evaluate when the effect of the truncation could be ignored. An open topic is whether the effect of truncating more bits, which increases the on-line effort

due to more false alarms, could be offset by the gains in coverage given that more chains can be stored on the same amount of memory.

This page intentionally left blank

Appendix A

Finding known bits in the SACCH Channel

The SACCH channel has the peculiarity ([28], 3.4.1.1) of requiring continuous transmission in both directions while there is an ongoing call. When there is no data to send (which is most of the time) it is used to transmit measurement results in the uplink, and information messages in the downlink, cycling four types of messages which carry general information about the cell and its parameters (called System Information type 5, 6, 5bis and 5ter messages). Those messages are carried in what are called “Unnumbered Information frames”. This means most of the time traffic will consist of messages with a known format, and we will use this fact to identify several bits with fixed values. There are some other optional messages like ”Measurement information”, instructing the mobile to send an enhanced measurement report, and ”Extended Measurement Order (EMO)”, requesting extended measurements.
We will describe the format of those messages, identifying several bits with a fixed value we can use.

A.1. Layer 1

The layer 1 header for the SACCH channel is described in [27]. It occupies 2 bytes and consists of four fields and a spare bit. The header for the downlink

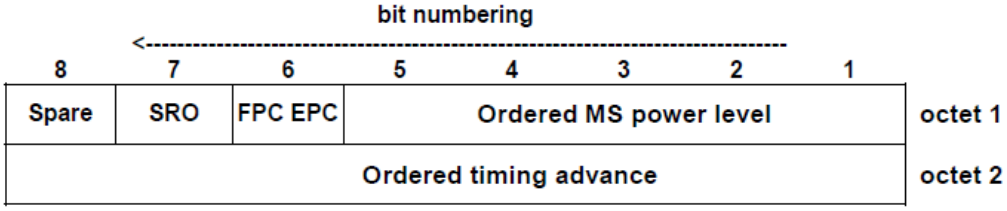

Figure A.1: Layer 1 header

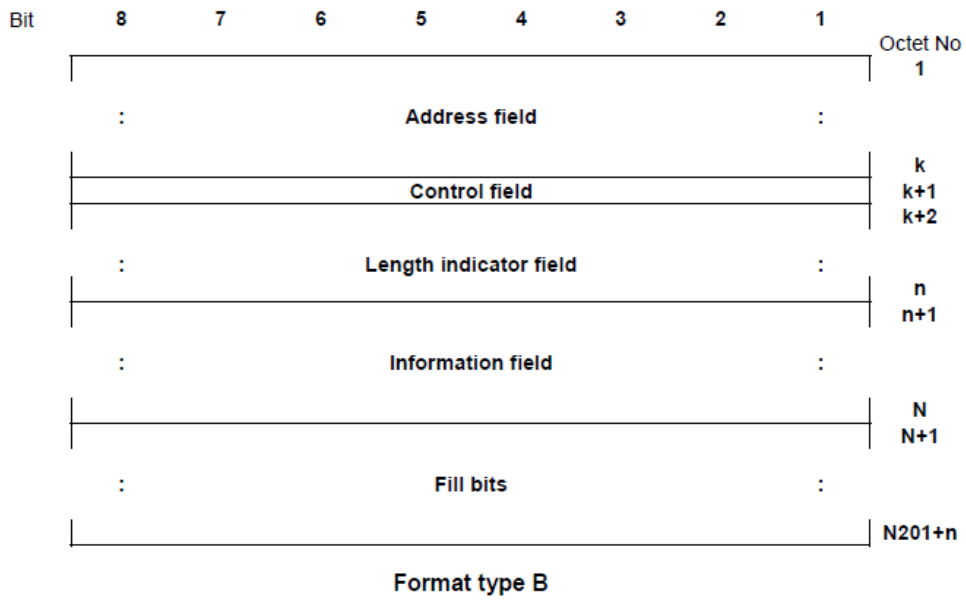

Figure A.2: LAPDM header - unnumbered frames

direction is shown in figure A.1, in the uplink direction the format is similar, replacing the ordered values with the Actual power level and Actual timing advance reported by the mobile. Spare bits are always transmitted as a binary zero. Besides, except for GSM400, the Timing Advance value has the most significant bit in 0. This means that except for GSM400 we have two bits with known value in this layer. GSM400 is a version of gsm for the 450 MHz frequency band which has seen little use throughout the world.

A.2. Layer 2

Layer 2 uses a protocol known as LAPD mobile (LAPDm), which is a variant of the layer 2 protocol used in ISDN for the control channel, but optimized for the requirements of the wireless network. LAPDm is a relatively simple protocol to exchange messages between two layer 2 entities, and has two modes of operation called acknowledged and unacknowledged. Acknowledged mode includes sequence numbers, explicit acknowledges for a stream of messages, and procedures for retransmitting lost messages, while unacknowledged mode provides a service without any guarantee, being the upper layers responsible for the retransmission of lost messages if needed. All the messages we are interested in are transmitted in unacknowledged operation, so we will only describe this mode. For unacknowledged operation, all messages are transported in Unnumbered Information (UI) frames, which are simplified frames with several fixed fields. The format of UI frames and the procedures of LAPDm are described in [26] and [30]. For the UI frames in the SACCH channel the header is as shown in figure A.2.

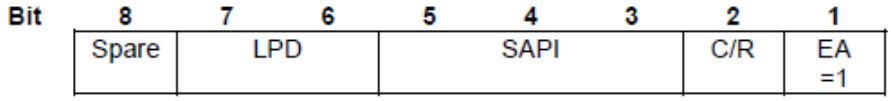

Figure A.3: LAPDM Address Field

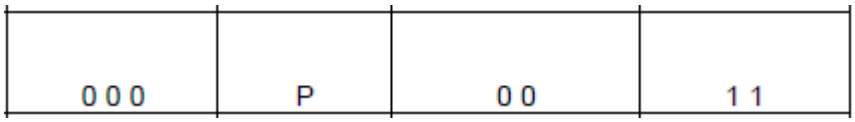

Figure A.4: LAPDM Control Field

In this case, the Address field and Lenght indication field are both one octet wide. The least significant bit of the length indication field (EL bit) is one. The address field is as shown in figure A.3, where the LPD and SAPI fields are 0, the EA bit is 1, and BIT 8 is a spare (also 0). The Control field for UI frames is as shown in A.4, where seven bits are fixed. We can also see in ([30] 8.2.1) that for the "unacknowledged information transfer with normal L2 header" bit P is also 0. Adding up, we have 16 bits with known value in layer 2

A.3. Layer 3

In layer 3 the messages we are interested in all belong to the Radio Resource Control Protocol (RRC) as described in [28]. The layer 3 header is shown in A.5 ([28] 10.1). For the messages of interest the SKIP field is 0. According to [33], for the Radio Resource Management messages, the field "Protocol Discriminator" has the binary value "0 1 1 0".

According to ([28] 10.4), the Message type has the following values:
For the uplink direction:

0 0 0 1 0 1 0 1 MEASUREMENT REPORT
0 0 1 1 0 1 1 0 EXTENDED MEASUREMENT REPORT

For the downlink direction:

0 0 0 1 1 1 0 1 SYSTEM INFORMATION TYPE 5
0 0 0 1 1 1 1 0 SYSTEM INFORMATION TYPE 6
0 0 0 0 0 1 0 1 SYSTEM INFORMATION TYPE 5bis
0 0 0 0 0 1 1 0 SYSTEM INFORMATION TYPE 5ter

We can see several bits with common values among the messages (6 bits in uplink, 4 bits in downlink).

So in layer 3 we have 12 bits with known values in the downlink direction, and 14 in the uplink.

If we add the "EXTENDED MEASUREMENT ORDER" message, which appears less frequently in the message stream, we only get 3 bits in the downling (Message type is 0 0 1 1 0 1 1 1 for this message).

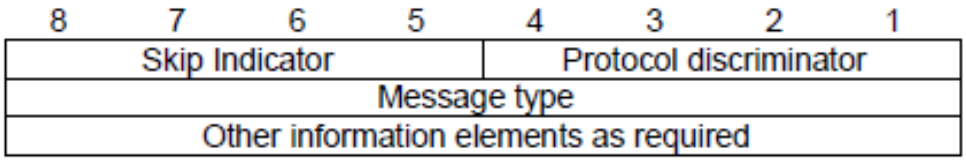

Figure A.5: Layer 3 header - Radio Resource Control

A.4. Summary

Summarizing, we have:

- 2 known bits in layer 1
- 16 known bits in layer 2
- 12 or 14 known bits (downlink or uplink) in layer 3

Adding all up, we have 30 bits (29 if we add the "EXTENDED MEASUREMENT ORDER" message) in the downlink direction and 32 bits in the uplink direction we can use to add equations to our system.

Given that there are more bits with known values than needed in Chapter 4, we can choose those which are useful in more cases. We can avoid using the MSB of the Timing advance in layer 1, so the resulting tables can also be used in GSM400, and avoid using the bits with fixed values in the different messages, in case some other message is sent. Even discarding those bits, we still have 25 bits, more than needed for the attack by Barkan, Biham and Keller.

Appendix B

Difference in the state after feeding the key and COUNT, when COUNT varies

As we saw in section 2.6.1, the initial state of A5/1 is calculated in the following way:

- Zero out all three registers $R1$, $R2$, $R3$
- Advance each register 64 times. In each step, xor the least significant bit of each register with the corresponding bit from the key.
- Advance each register 22 times. In each step, xor the least significant bit of each register with the corresponding bit from COUNT.
- Advance A5/1 100 times using the majority rule, discarding output.

We are interested in the values of $R1$, $R2$ and $R3$ just after feeding the value of COUNT, before the final 100 clockings of the initialization.
Let $COUNT = C_{21} \cdots C_0$. Bits from COUNT are fed starting with the least significant bit.

Differences for $R1$

$R1$ is 19 bits long, and the feedback taps are in positions 13, 16, 17 and 18. Let r_i be the least significant bit of $R1$ at step i (for example, $r_0 = 0$, $r_1 = k[0]$, $r_{14} = k[13] + r_1$, $\cdots$). After the first 64 steps, $R1$ will contain from r_{64} (in $R1$'s LSB) to r_{46} (MSB).
Feeding COUNT bit by bit we get:

$r_{65} = c_0 \oplus r_{51} \oplus r_{48} \oplus r_{47} \oplus r_{46}$
$r_{66} = c_1 \oplus r_{52} \oplus r_{49} \oplus r_{48} \oplus r_{47}$
...
$r_{78} = c_{13} \oplus r_{64} \oplus r_{61} \oplus r_{60} \oplus r_{59}$
$r_{79} = c_{14} \oplus r_{65} \oplus r_{62} \oplus r_{61} \oplus r_{60} = c_{14} \oplus c_0 \oplus r_{51} \oplus r_{48} \oplus r_{47} \oplus r_{46} \oplus r_{62} \oplus r_{61} \oplus r_{60}$
$r_{80} = c_{15} \oplus c_1 \oplus r_{52} \oplus r_{49} \oplus r_{48} \oplus r_{47} \oplus r_{63} \oplus r_{62} \oplus r_{61}$
$r_{81} = c_{16} \oplus c_2 \oplus r_{53} \oplus r_{50} \oplus r_{49} \oplus r_{48} \oplus r_{64} \oplus r_{63} \oplus r_{62}$

Appendix B. Difference in the state after feeding the key and COUNT, when COUNT varies

$r_{82} = c_{17} \oplus c_3 \oplus r_{54} \oplus r_{51} \oplus r_{50} \oplus r_{49} \oplus r_{65} \oplus r_{64} \oplus r_{63} = c_{17} \oplus c_3 \oplus r_{54} \oplus \cancel{r_{51}} \oplus r_{50} \oplus r_{49} \oplus c_0 \oplus \cancel{r_{51}} \oplus r_{48} \oplus r_{47} \oplus r_{46} \oplus r_{64} \oplus r_{63}$

$r_{83} = c_{18} \oplus c_4 \oplus r_{55} \oplus r_{51} \oplus r_{50} \oplus c_1 \oplus r_{49} \oplus r_{48} \oplus r_{47} \oplus r_{65} \oplus r_{64} = c_{18} \oplus c_4 \oplus r_{55} \oplus \cancel{r_{51}} \oplus r_{50} \oplus c_1 \oplus r_{49} \oplus \cancel{r_{48}} \oplus \cancel{r_{47}} \oplus c_0 \oplus \cancel{r_{51}} \oplus \cancel{r_{48}} \oplus \cancel{r_{47}} \oplus r_{46} \oplus r_{64}$

$r_{83} = c_{18} \oplus c_4 \oplus r_{55} \oplus r_{50} \oplus c_1 \oplus r_{49} \oplus c_0 \oplus r_{46} \oplus r_{64}$

$r_{84} = c_{19} \oplus c_5 \oplus r_{56} \oplus r_{51} \oplus c_2 \oplus r_{50} \oplus c_1 \oplus r_{47} \oplus r_{65} = c_{19} \oplus c_5 \oplus r_{56} \oplus \cancel{r_{51}} \oplus c_2 \oplus r_{50} \oplus c_1 \oplus \cancel{r_{47}} \oplus c_0 \oplus \cancel{r_{51}} \oplus r_{48} \oplus \cancel{r_{47}} \oplus r_{46}$

$r_{84} = c_{19} \oplus c_5 \oplus r_{56} \oplus c_2 \oplus r_{50} \oplus c_1 \oplus c_0 \oplus r_{48} \oplus r_{46}$

$r_{85} = c_{20} \oplus c_6 \oplus r_{57} \oplus c_3 \oplus r_{51} \oplus c_2 \oplus c_1 \oplus r_{49} \oplus r_{47}$

$r_{86} = c_{21} \oplus c_7 \oplus r_{58} \oplus c_4 \oplus r_{52} \oplus c_3 \oplus c_2 \oplus r_{50} \oplus r_{48}$

The value of $R1$ after initialization with the key and COUNT (before the 100 mixing cycles) is $r_{86} \cdots r_{68}$, which can be expressed using only $r_{64}...r_1$ (which only depend on the key) and the bits from COUNT.
Let two values of COUNT be $c_{21}...c_0$ and $c'_{21}...c'_0$, where $c'_i = c_i \oplus \Delta_i$. Then:

$r'_{65} = r_{65} \oplus \Delta_0$
...
$r'_{78} = r_{78} \oplus \Delta_{13}$
$r'_{79} = r_{79} \oplus \Delta_{14} \oplus \Delta_0$
$r'_{80} = r_{80} \oplus \Delta_{15} \oplus \Delta_1$
$r'_{81} = r_{81} \oplus \Delta_{16} \oplus \Delta_2$
$r'_{82} = r_{82} \oplus \Delta_{17} \oplus \Delta_3 \oplus \Delta_0$
$r'_{83} = r_{83} \oplus \Delta_{18} \oplus \Delta_4 \oplus \Delta_1 \oplus \Delta_0$
$r'_{84} = r_{84} \oplus \Delta_{19} \oplus \Delta_5 \oplus \Delta_2 \oplus \Delta_1 \oplus \Delta_0$
$r'_{85} = r_{85} \oplus \Delta_{20} \oplus \Delta_6 \oplus \Delta_3 \oplus \Delta_2 \oplus \Delta_1$
$r'_{86} = r_{86} \oplus \Delta_{21} \oplus \Delta_7 \oplus \Delta_4 \oplus \Delta_3 \oplus \Delta_2$

This shows it is easy knowing ΔCOUNT and the state at step 86 for a certain value of COUNT, to calculate the state for another value COUNT' without knowing the key.
The same calculation yields the differences for $R2$ and $R3$.

Differences for $R2$ and $R3$

$R2$ is 22 bits long, with the feedback taps in positions 20 and 21. Following the same calculation as in the case of $R1$, after the first 64 steps $R2$ will be $r_{64} \cdots r_{43}$

$r_{65} = c_0 \oplus r_{44} \oplus r_{43}$
$r_{66} = c_1 \oplus r_{45} \oplus r_{44}$
...
$r_{86} = c_{21} \oplus r_{65} \oplus r_{64} = c_{21} \oplus c_0 \oplus r_{44} \oplus r_{43} \oplus r_{64}$

$R2$ at step 86 is $r_{86}...r_{65}$, which can be expressed using only $r_{64}...r_1$ (which only depend on the key) and the bits from COUNT.

For 2 values of COUNT, $c_{21}...c_0$ and $c'_{21}...c'_0$ with $c'_i = c_i \oplus \Delta_i$:

$r'_{65} = r_{65} \oplus \Delta_0$
...
$r'_{85} = r_{85} \oplus \Delta_{20}$
$r'_{86} = r_{86} \oplus \Delta_{21} \oplus \Delta_0$

Doing a similar calculation for $R3$ (which is 23 bits long, with taps bits in positions 7, 20, 21 and 22), we get:

$r'_{64} = r_{64}$
$r'_{65} = r_{65} \oplus \Delta_0$
...
$r'_{72} = r_{72} \oplus \Delta_7$
$r'_{73} = r_{73} \oplus \Delta_8 \oplus \Delta_0$
...
$r'_{80} = r_{80} \oplus \Delta_{15} \oplus \Delta_7$
$r'_{81} = r_{81} \oplus \Delta_{16} \oplus \Delta_8 \oplus \Delta_0$
...
$r'_{85} = r_{85} \oplus \Delta_{20} \oplus \Delta_{12} \oplus \Delta_4$
$r'_{86} = r_{86} \oplus \Delta_{21} \oplus \Delta_{13} \oplus \Delta_5 \oplus \Delta_0$

Differences for the tables of Birham, Barkan and Keller

In this case, only the least significant bit of $T3$ changes, which is bit 5 in COUNT. Replacing in the preceeding equations we get that only the following bits have their values changed:

- For R1, bits 2 and 16.
- For R2, bit 16.
- For R3, bits 0, 8 and 16.

This page intentionally left blank

Appendix C

Finding key K_C from A5/1's internal state after key setup

Given the value of A5/1's internal registers R1, R2 and R3 just after feeding the key K_C and the value of COUNT, we want to find K_C. This can be easily done by first reverting the effect of COUNT, and then inverting the linear initialization from the value of K_C to the values of R1, R2 and R3.

Just as in appendix B, let's call $C_{21} \cdots C_0$ the bits from COUNT, and let r_i be the least significant bit of $R1$ at step i. The contents of $R1$ after key setup are $r_{86} \cdots r_{68}$, and we want to calculate $r_{64} \cdots r_{46}$

When feeding COUNT, just as in appendix B we have:

$r_{65} = c_0 \oplus r_{51} \oplus r_{48} \oplus r_{47} \oplus r_{46}$

$r_{66} = c_1 \oplus r_{52} \oplus r_{49} \oplus r_{48} \oplus r_{47}$

$\vdots$

$r_{86} = c_1 \oplus r_{72} \oplus r_{69} \oplus r_{68} \oplus r_{67}$

In the last equation we can solve for r_{67} as $r_{67} = c_{21} \oplus r_{72} \oplus r_{69} \oplus r_{68} \oplus r_{86}$

We can do the same for the other 21 equations finding $r_{66} \cdots r_{46}$ as we wanted.

For $R2$ and $R3$ we follow the same procedure, finding the corresponding values just after feeding key K_C.

Let's call R the concatenation of $R1$, $R2$ and $R3$. As the initialization is linear, there is a matrix M_I such that $R = M_I \cdot K_C$. Inverting M_I we can find K_C as

$K_C = M_I^{-1} \cdot R$

This page intentionally left blank

Appendix D

Calculating the parameters of the TMDTO

D.1. Calculating the tradeoff parameters

Given the available memory M in bytes and the desired success probability, we want to find possible parameters for the tradeoff. Given that s and l must be integer, we will treat them as such. We pick a range of reasonable s and l values and for each pair calculate the tradeoff parameters and constants in a spreadsheet. Afterwards we can pick the value which best suits our application.
Given M and D.
- Choose ϵ (eg. 8)
- Call bpp the bits per point after truncation
For each s,l

- calculate bpp (for this increment the estimated bpp and calculate m0, until $bpp >= log_2(m0)$)
- calculate m as $M * 8/(l * bpp)$
- use equation (6.14) and the definition of $\overline{F}_{msc}$ to calculate $t\overline{F}_{cr} = -N \log(1-\overline{F}_{ps})/(D * s * l * m)$
- Use the definition of $\overline{F}_{msc}$ and the calculated value of $t\overline{F}_{cr}$ to get $\overline{F}_{msc} * \overline{F}_{cr}^2$
- iteratively find $\overline{F}_{msc}$ and $\overline{F}_{cr}$ from the previous value
- Calculate $m0 = 2m\overline{F}_{msc}/(2 - \overline{F}_{msc})$
- Calculate t from the previous values $\overline{F}_{cr}$ and $t * \overline{F}_{cr}$
- Calculate $\overline{F}_{pc}$ from equation (6.8)
- Calculate the on-line effort T from equation (6.27)

This page intentionally left blank

Appendix E

Table 1 from Kim's paper

Table E.1: Values of Fmsc and s that minimize Fatc

		Fatcs						
$\log m_* + \epsilon$	Fps	0,5	0,75	0,9	0,95	0,99	0,995	0,999
18	s	34	38	43	48	60	66	79
	Fmsc	1,6881	1,6883	1,6847	1,6813	1,6698	1,6647	1,6531
19	s	36	40	46	50	63	68	83
	Fmsc	1,7000	1,6997	1,6968	1,6922	1,6810	1,6754	1,6644
20	s	37	42	48	53	65	71	86
	Fmsc	1,7095	1,7104	1,7071	1,7032	1,6911	1,6858	1,6747
21	s	39	44	50	55	68	74	89
	Fmsc	1,7198	1,7202	1,7166	1,7126	1,7009	1,6956	1,6843
22	s	41	46	52	57	71	77	93
	Fmsc	1,7294	1,7293	1,7256	1,7215	1,7101	1,7047	1,6937
23	s	43	48	54	59	73	80	96
	Fmsc	1,7382	1,7379	1,7340	1,7298	1,7183	1,7133	1,7022
24	s	45	50	56	62	76	83	100
	Fmsc	1,7465	1,7459	1,7418	1,7381	1,7264	1,7214	1,7105
25	s	47	51	58	64	79	86	103
	Fmsc	1,7542	1,7527	1,7493	1,7454	1,7341	1,7290	1,7180
26	s	49	53	60	66	81	89	106
	Fmsc	1,7615	1,7598	1,7562	1,7524	1,7410	1,7362	1,7252
27	s	51	55	63	69	84	91	110
	Fmsc	1,7683	1,7665	1,7633	1,7593	1,7478	1,7428	1,7322
28	s	52	57	65	71	87	94	113
	Fmsc	1,7740	1,7728	1,7695	1,7655	1,7543	1,7492	1,7387
29	s	54	59	67	73	89	97	116
	Fmsc	1,7801	1,7788	1,7754	1,7713	1,7602	1,7553	1,7448

This table was calculated by finding the values of s and $\overline{F}_{msc}$ that minimice $\overline{F}_{atc}$. Comparing this table with Table 1 from Kim and Hong's paper, we see they are almost identical

This page intentionally left blank

Appendix F

Description of the test infrastructure

To implement a TMTO, even for a reduced problem, we need to calculate a large number of images of the step function, specially during the precomputation phase when many similar chains must be calculated. The precomputation phase is highly parallelizable, so it seems a good candidate for computation using GPU cards.

Graphic processing has always been a demanding task for computer systems, which in many cases has been delegated to specialized hardware, called Graphics Processing Unit (GPU). Since about 2001, with the advent of programmable shaders and floating point support in GPUs, they have been used for general computation, at first by reformulating computational problems in terms of graphic primitives, until the advent of general purpose programming language extensions and APIs which enabled programmers to abstract the underlying computation resources. The usage of GPUs for general computing is called General-Purpose Computing on Graphics Processing Units (GPGPU), and has been applied to multiple high performance computing problems in areas such as genomics, materials science, and cryptography. GPUs cards excel at problems where a high degree of parallelism can be achieved.

Nowadays there are two main competing producers of GPU chips, AMD (Advanced Micro Devices, Inc.) and NVIDIA, and both companies produce general purpose GPU cards for graphics processing, and cards optimized for the GPGPU community, with higher double precision floating point performance, proportionally larger memory, and higher computation capacity. For our problem we are not concerned with floating point performance, as our problem does not involve floating point computations. Both card brands are equally capable for high performance computation, and our choice of hardware was dictated by the available resources at our college's computation cluster when this work started, namely a NVIDIA S1070 GPU Computing Server, which contains four C1060 computing modules. There are two popular extensions to programming languages that can be used to program NVIDIA GPU cards, one is called CUDA and is NVIDIA proprietary, while the other, OpenCL, is open and available for other GPU cards and CPUs. When this project was started OpenGL was not as stable as CUDA and the examples we had were programmed using CUDA, so that's the programming

extensions we choose.

F.1. Programming on CUDA cards

The GPU architecture is well suited for data-parallel computations and has a high ratio of computation to memory operations. State of the art CPUs in 2017 have several complex cores, up to 24 in top of the line CPUs, executing two threads per core using hiperthreading. In contrast GPUs have thousands of cores, but they are simple ones and function at their fullest capacity when groups of threads share the same execution flow and with relatively little main memory access. GPU cores have available more internal registers than CISC CPUs, memory bandwidth to main memory is higher in GPU cards, but memory latency is also much higher than in CPUs.

There are many different CUDA cards, with varying features and performance. Features are grouped in what nvidia calls "Compute Capability" of a device. We will not go into the details of the architecture, nor the differences between different cards, they can be found on the NVIDIA website [1], only describing them from the developer's point of view. A good source of information is the "Nvidia CUDA Programming Guide" and in general all the documentation that comes with the CUDA libraries which is also freely available at the NVIDIA web site.

The main abstraction in CUDA programming is called a "kernel". Kernels are C (or C++/Fortran) functions that, when called, are executed N times in parallel by N different *CUDA threads* on the GPU card. The number of threads N is specified at each kernel invocation, and the thread number is available to each thread to enable differentiation among threads.

Processor cores in CUDA are grouped in multiprocessors, which share resources, a set of registers, a block of local shared memory, and some other resources. The minimum thread grouping is called a warp, and consist of 32 threads, which are assigned to the same multiprocessor. All 32 threads on a warp start at the same instruction, and while execution can diverge via a data dependent conditional branch, maximum performance is obtained when all 32 threads of a warp agree on the execution path. Multiple warps are grouped in thread blocks, which execute concurrently on one multiprocessor. Several blocks can be defined, and different blocks will be automatically assigned to available multiprocessors.

Taking as example the architecture of the available C1060 card, there are 30 multiprocessors, each containing 8 cores. Each multiprocessor can host up to 8 blocks, with a maximum of 1024 threads. There are 16K registers per multiprocessor which are divided among all concurrent threads, and each thread can have a maximum of 124 registers. There is a 16 KB block of shared memory on each multiprocessor. To improve performance the size of each thread block and the number of blocks must be carefully chosen. Too few threads and the performance suffers because there are not enough active threads to use all resources and hide

[1] `http://www.nvidia.com/object/tesla_product_literature.html`

memory latency. Too many threads per block and the number of registers for each thread is not enough to keep the necessary data locally. In the precomputation phase of our algorithm, for the C1060 card we were able to maximize performance when using 30 blocks of 128 threads each, meaning we need at least 3840 threads running concurrently.

We also implemented the algorithms in the CPU, useful for calculating only a few chains, which negate the efficiency of the GPU parallel calculation.

F.2. Some comments on the implemented algorithms

In this section we include some comments on the decisions taken for our implementation, and a high level overview of some key points of the implementation.

After initial implementation tests, which allowed us to get acquainted with CUDA programming, we decided to split the processing so that the kernel implemented in the GPU calculates complete chains starting either from N starting points (for the precomputation step) or from N points at any color. The parallel programming would have been probably easier if only the step function was calculated on the GPU, however we determined that, at least with the sample parameters we used initially, the overhead of copying data to and from the GPU card after each step was too high compared to the time it took to calculate each step.

To maximize parallel execution, the calculation was divided into a section that calculates a step of the h_c function, in which all threads execute exactly the same code, and a section where the distinguishing property is checked, the comparison with a possible candidate is done if we are in the on-line phase, and changes are made if necessary, either on the color if a distinguished point is reached, or changing to a new chain if the current chain is finished. In this way for the most expensive step, the calculation of h_c, all 32 threads on a warp execute the same calculations which is a necessary condition to maximize performance.

The calculation of the step function h_c can be subdivided into the application of function A5/1 and the multiplication with matrix H_c. The later is easily implemented using binary XOR functions and does not merit further analysis. in our implementation the calculation of A5/1 for both initial states as needed for function h_c takes approximately twice as long as the matrix multiplication.

For the calculation of A5/1, we implemented two versions, one which maximizes performance when many chains are calculated in parallel, and another better suited for the online phase when fewer chains must be calculated.

A5/1 using table lookups

In [15] Biryukov et al propose to use precomputed tables to advance A5/1. Nohl et al initially used an implementation of A5/1 using search tables. We implemented a version of A5/1 using tables to calculate, given 4 bits from each register

starting at the current clocking bit, how much each register should advance, and the corresponding output bits given the contents of the registers. The choice of 4 bits was given by the size of the shared memory on each multiprocessor, as tables using more bits needed to be stored on main memory and the slower speed and much higher latency of main memory negated all speed gains. Each A5/1 register is stored on a 32 bit GPU register.

It takes approximately 45 integer/bitwise operations plus 6 table accesses to advance A5/1 4 clockings, plus 15 extra operations and two extra table accesses when the output bits must be calculated (that is, after the 100 initialization cicles 60 operations are needed at each clocking, and 8 table lookups).

Algorithm using bit slicing

Bit slicing is a technique that improves the performance of certain calculations when several instances of the same algorithm can be calculated in parallel. It is well suited to algorithms consisting mostly on bitwise logical computations. It was initially presented by Eli Biham in 1997 [11] as a faster implementation of DES, although he did not use the name "bit slicing". Bit slicing was used in the implementation of the tool "Kraken" by Nohl et al for the AMD implementation of their attack, and we implemented our step function h_c using this technique for the CUDA cards.

As described in [56], "Bit-slicing regards a W-bit processor as a SIMD parallel computer capable of performing W parallel 1-bit operations simultaneously".

Taking as an example the implementation of A5/1, instead of storing the internal state of an instance of A5/1 using individual registers as we did in the previous implementation, we use 64 32-bit registers to store the internal state of 32 A5/1 instances (the native register size in the available CUDA cards is 32 bits). Register number n holds the n^{th} bit of the internal state of the 32 A5/1 instances.

Let's call $lfsr1_i$ the register that contains the i^{th} bit from register R1 for the 32 instances of A5/1. In the same manner define $lfsr2_i$ and $lfsr3_i$ for R2 and R3 respectively.

Using this representation, to advance A5/1 we must first calculate the majority value for the clocking bits of all A5/1 instances, and decide which registers should advance. The registers that should advance for the 32 instances are calculated using the following C code

```
uint32_t majority=(lfsr1_8&lfsr2_10)|(lfsr1_8&lfsr3_10)|(lfsr2_10&lfsr3_10);
uint32_t clock1=~(lfsr1_8^majority);
uint32_t clock2=~(lfsr2_10^majority);
uint32_t clock3=~(lfsr3_10^majority);
```

Next we calculate the feedback bit that will be fed if the register advances. For example for R1:

```
lfsr_temp=lfsr1_13^lfsr1_16^lfsr1_17^lfsr1_18;
```

The action of advancing the registers consists in, for the bit in position j, keeping the bit constant if the corresponding clock bit is zero, and substituting the bit for the bit in position $j-1$ if the register should advance. For $j=0$, the

feedback bit substitutes the $j-1$ bit. After the 100 initialization steps, the output bits are also calculated.

```
lfsr1_18 &= ~clock1;
lfsr1_18 |= (lfsr1_17 & clock1);
...
lfsr1_1 &= ~clock1;
lfsr1_1 |= (lfsr1_0 & clock1);
lfsr1_0 &= ~clock1;
lfsr1_0 |= (lfsr_temp & clock1);
//past the 100 initialization cycles, calculate bit output
if(round>=100)
    bitsalida[round-100]=lfsr1_18 ^ lfsr2_21 ^ lfsr3_22;
```

Calculating ~clock1 once for each register, approximately $64 \times 3 + 8$ operations must be performed to advance one clocking of A5/1 (withouth counting looping, etc.), and as we are calculating 32 instances in parallel, each clocking of each instance implies approximately 6-7 instructions and requires no table lookups in the initialization step, and a two table lookups (for the 32 A5/1 instances) for multiplication with matrix H.

The main disadvantage of the bit slicing algorithm is the added parallelism, as each thread is calculating 32 A5/1 instances simultaneously, and we need 3840 parallel threads, meaning we need at least 122,800 simultaneous A5/1 calculations. This is not a problem for the precomputation step, but makes this algorithm unusable for the on-line phase unless there are a large number of captured ciphertexts to attempt inversion.

This page intentionally left blank

Bibliography

[1] 4G Americas. Mobile market shares by technology. `http://www.4gamericas.org/en/resources/statistics/statistics-global/`, 2016. On-line report. Last accessed Dec. 2016.

[2] Ross Anderson. A5 (was: Hacking digital phones), message to the sci.crypt group on usenet. Can be read at http://groups.google.com/group/sci.crypt/msg/ba76615fef32ba32. Last accessed August 2014.

[3] Ross Anderson. On fibonacci keystream generators. In Bart Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 346–352. Springer Berlin / Heidelberg, 1995. 10.1007-3-540-60590-8_26.

[4] Gildas Avoine, Pascal Junod, and Philippe Oechslin. Time-memory trade-offs: False alarm detection using checkpoints. In Subhamoy Maitra, C. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005*, volume 3797 of *Lecture Notes in Computer Science*, pages 183–196. Springer Berlin / Heidelberg, 2005.

[5] Gildas Avoine, Pascal Junod, and Philippe Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.*, 11:17:1–17:22, July 2008.

[6] Steve Babbage. A space/time trade-off in exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection, IEE Conference Publication No.408*, 1995.

[7] Elad Barkan. *Cryptanalysis of Ciphers and Protocols*. PhD thesis, Technion — Israel Institute of Technology, 2006.

[8] Elad Barkan and Eli Biham. Conditional estimators: An effective attack on a5/1. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2005.

[9] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of gsm encrypted communication. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa*

Barbara, California, USA, August 17-21, 2003, Proceedings, volume 2729 of *Lecture Notes in Computer Science*, pages 600–616. Springer, 2003.

[10] Elad Barkan, Eli Biham, and Adi Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin / Heidelberg, 2006.

[11] Eli Biham. *A fast new DES implementation in software*, pages 260–272. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[12] Eli Biham and Orr Dunkelman. Cryptanalysis of the a5/1 gsm stream cipher. In Bimal Roy and Eiji Okamoto, editors, *Progress in Cryptology —INDOCRYPT 2000*, volume 1977 of *Lecture Notes in Computer Science*, pages 43–51. Springer Berlin / Heidelberg, 2000.

[13] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved time-memory trade-offs with multiple data. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 110–127. Springer Berlin / Heidelberg, 2006. 10.1007/11693383.8.

[14] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Tatsuaki Okamoto, editor, *Advances in Cryptology, ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2000.

[15] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of a5/1 on a pc. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bruce Schneier, editors, *Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 37–44. Springer Berlin / Heidelberg, 2001.

[16] Johan Borst, Bart Preneel, Joos Vandewalle, and Joos V. On the time-memory tradeoff between exhaustive key search and table precomputation. In *Proc. of the 19th Symposium in Information Theory in the Benelux, WIC*, pages 111–118, 1998.

[17] Marc Briceno, Ian Goldberg, and David Wagner. A pedagogical implementation of the gsm a5/1 and a5/2 "voice privacy" encryption algorithms, 1999. http://cryptome.org/gsm-a512.htm.

[18] J. Eberspächer, H.J. Vögel, C. Bettstetter, and C. Hartmann. *GSM - Architecture, Protocols and Services.* John Wiley and Sons, Ltd, 2008.

[19] P. Ekdahl and T. Johansson. Another attack on a5/1 [gsm stream cipher]. In *Information Theory, 2001. Proceedings. 2001 IEEE International Symposium on*, page 160, 2001.

[20] P. Ekdahl and T. Johansson. Another attack on a5/1. *Information Theory, IEEE Transactions on*, 49(1):284–289, Jan 2003.

[21] Victor Shoup et al. Ntl: A library for doing number theory. `http://www.shoup.net/ntl/`, 2013. Last accessed October 2014.

[22] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); channel coding. TS 100 909 (3GPP TS 05.03), 2000.

[23] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); security-related network functions. TS 143 020 (3GPP TS 43.020), 2009.

[24] European Telecommunications Standards Institute. Universal mobile telecommunications system (umts); specification of the 3gpp confidentiality and integrity algorithms; document 2: Kasumi algorithm specification. TS 135 202 (3GPP TS 35.202), 2009.

[25] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); channel coding. TS 145 003 (3GPP TS 45.003), 2011.

[26] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); data link (dl) layer general aspects. TS 144 005 (3GPP TS 44.005), 2011.

[27] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); layer 1; general requirements. TS 144 004 (3GPP TS 44.004), 2011.

[28] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); mobile radio interface layer 3 specification; radio resource control (rrc) protocol. TS 144 018 (3GPP TS 44.018), 2011.

[29] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); mobile station - base station system (ms - bss) interface; channel structures and access capabilities. TS 144 003 (3GPP TS 44.003), 2011.

[30] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); mobile station - base station system (ms - bss) interface; data link (dl) layer specification. TS 144 006 (3GPP TS 44.006), 2011.

[31] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); multiplexing and multiple access on the radio path. TS 145 002 (3GPP TS 45.002), 2011.

[32] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); physical layer on the radio path; general description. TS 145 001 (3GPP TS 45.001), 2011.

[33] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); universal mobile telecommunications system (umts); lte; mobile radio interface signalling layer 3; general aspects. TS 124 007 (3GPP TS 24.007), 2011.

[34] Timo Gendrullis, Martin Novotný, and Andy Rupp. A real-world attack breaking a5/1 within hours. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 266–282. Springer Berlin / Heidelberg, 2008.

[35] Tim Güneysu, Timo Kasper, Martin Novotny, Christof Paar, and Andy Rupp. Cryptanalysis with copacobana. *IEEE TRANSACTIONS ON COMPUTERS*, 57(11):1498–1513, 2008.

[36] Ian Goldberg, David Wagner, and Lucky Green. The (real-time) cryptanalysis of a5/2. Rump Session, Crypto '99, 1999.

[37] J. D. Golic. Cryptanalysis of three mutually clock-controlled stop/go shift registers. *IEEE Trans. Inf. Theor.*, 46(3):1081–1090, September 2006.

[38] Jovan Dj. Golic. Cryptanalysis of alleged a5 stream cipher. In *Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, EUROCRYPT'97, pages 239–255, Berlin, Heidelberg, 1997. Springer-Verlag.

[39] M. Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401 – 406, jul 1980.

[40] Jin Hong. The cost of false alarms in hellman and rainbow tradeoffs. *Designs, Codes and Cryptography*, 57:293–327, 2010. 10.1007/s10623-010-9368-x.

[41] Jin Hong and Sunghwan Moon. A comparison of cryptanalytic tradeoff algorithms. Cryptology ePrint Archive, Report 2010/176, 2010. `http://eprint.iacr.org/`.

[42] Jin Hong and Sunghwan Moon. A comparison of cryptanalytic tradeoff algorithms. *Journal of Cryptology*, 26(4):559–637, 2013.

[43] David Hulton and Steve. Cracking gsm. Technical report, Black Hat Briefing, Washington DC, 2008, 2008.

[44] GSMA Intelligence. Gsma intelligence global data dashboard. `https://gsmaintelligence.com/`, 2016. On-line report. Last accessed Jun 2016.

[45] Sylvain Munaut Karsten Nohl. Gsm sniffing. Presented at 27th Chaos Communication Congress, 2010. `https://events.ccc.de/congress/2010/Fahrplan/events/4208.en.html`.

[46] B. Keller, J. ; Seitz. A hardware-based attack on the a5/1 stream cipher. *ITG FACHBERICHT*, pages 155–158, 2001.

[47] Byoung-Il Kim and Jin Hong. Analysis of the non-perfect table fuzzy rainbow tradeoff. In Colin Boyd and Leonie Simpson, editors, *Information Security and Privacy*, volume 7959 of *Lecture Notes in Computer Science*, pages 347–362. Springer Berlin Heidelberg, 2013.

[48] Byoung-Il Kim and Jin Hong. Analysis of the perfect table fuzzy rainbow tradeoff. *J. Applied Mathematics*, 2014, 2014.

[49] Ga Won Lee and Jin Hong. A comparison of perfect table cryptanalytic tradeoff algorithms. Cryptology ePrint Archive, Report 2012/540, 2012. `http://eprint.iacr.org/`.

[50] Daegun Ma and Jin Hong. Success probability of the hellman trade-off. *Inf. Process. Lett.*, 109(7):347–351, March 2009.

[51] Alexander Maximov, Thomas Johansson, and Steve Babbage. An improved correlation attack on a5/1. In Helena Handschuh and M. Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-30564-4_1.

[52] Karsten Nohl. Attacking phone privacy. Presented at Black Hat USA 2010, Las Vegas (July 2010), 2010. `https://www.blackhat.com/html/bh-us-10/bh-us-10-archives.html#Nohl`.

[53] Martin Novotný. *Time-area efficient hardware architectures for cryptography and cryptanalysis*. PhD thesis, Ruhr University Bochum, 2009.

[54] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630, Berlin, Heidelberg, August 2003. Springer Berlin / Heidelberg.

[55] Thomas Pornin and Jacques Stern. Software-hardware trade-offs: Application to a5/1 cryptanalysis. In Çetin Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 155–184. Springer Berlin / Heidelberg, 2000.

[56] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. *Efficient Rijndael Encryption Implementation with Composite Field Arithmetic*, pages 171–184. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[57] Andy Rupp. *Computational aspects of cryptography and cryptanalysis*. PhD thesis, Ruhr University Bochum, 2008.

[58] Francois-Xavier Standaert, Aert Francois-xavier, Rouvroy Gael, Jean-Jacques Quisquater, and Legat Jean-didier. A time-memory tradeoff using distinguished points: New analysis & fpga results, 2002.

[59] Various. A5/1 cracking project webpage. Online. No longer accessible, but archived at the Wayback Machine, https://web.archive.org/web/20120426060932/http://reflextor.com/trac/a51, 2012.

Glossary

3GPP 3rd Generation Partnership Project. 2, 3

3GPP2 3rd Generation Partnership Project 2. 2

AMPS Advanced Mobile Phone System. 1

AuC Authentication Center. 7, 18

BCCH Broadcast Control Channel. 9

BSC Base Station Controller. 6

BTS Base Transceiver Station. 6, 8, 17

burst In GSM, a burst is the minimum unit of transmission, with a duration of 3/5200s . 131

COUNT In GSM, COUNT is a number calculated from the Frame Number FN, used to seed the encryption algorithm for each new burst. 19, 35, 36

CRC Cyclic Redundancy Check, error detection and possibly correction codes well suited for the detection of burst errors. 14, 131

DP Distinguished Point. 49, 65, 79

EIR Equipment Identity Register. 7, 8

ETSI European Telecommunications Standards Institute. 2, 32

FACCH Fast Associated Control Channel. 12

FDM frequency division multiplexing, a method to share a RF channel between different users, by dividing the available frequency range in smaller ranges and assigning a different sub-range to each user. 5

Fire code a type of CRC, error detection and possibly correction codes, well suited for single burst detection or correction of errors. 14

FN TDMA Frame Number, a counter that identifies each frame counting from an arbitrary starting time, and running from 0 to $FN_{MAX} = (26 \times 51 \times 2048) - 1 = 2715647$. 9, 19, 20, 35, 36, 131

FPGA Field Programmable Gate Array. 31

GFLOP GigaFLOP, 10^9 floating point operations per second, a measure of the computation capacity of a system. 94

GMSC Gateway Mobile Switching Center. 7

GPGPU General-Purpose Computing on Graphics Processing Units. 119

GPU Graphics Processing Unit. 31, 119–121

GSM Global System for Mobile communications. 1, 2, 4, 33, 101

GSM400 GSM adapted for the 450 MHz band. Has seen little use globally. 106

GSMA GSM Association. 5

HLR Home Location Register. 7

HSN Hopping Sequence Number. 10

IMEI The International Mobile Equipment Identity (IMEI) is an identification of the mobile device (eg. phone) which should be unique. 7

IMSI The International Mobile Subscriber Identity (IMSI) is a unique identification associated with a user of a cellular network. 7, 18

Kasumi Kasumi is a block cipher designed by the SAGE group of ETSI, based on the Misty1 cipher which in turn was designed for Mitsubishi Electric in 1995. 2, 20

LAPDm LAPD mobile. 8, 106

LFSR Linear Feedback Shift Register. 20, 26

LSB Least Significant Bit. 39

LTE Long Term Evolution, an ETSI/3GPP standard for 4rd. generation cellular systems. 2, 22, 33

MA Mobile Allocation. 10

MAIO Mobile Allocation Index Offset. 10

MCC Mobile Country Code. 8

MNC Mobile Network Code. 8

MS Mobile Station, the phone or device used to connect to the GSM network. 6, 7, 17–20, 23

MSC Mobile Switching Center. 7, 18, 19

MSISDN Mobile Station Integrated Services Digital Network number, this is a number uniquely identifying a subscription in a GSM or UMTS mobile network. This is the "phone number" of the subscriber. 7

RF radio frequency. 5, 131, 133

RRC Radio Resource Control Protocol. 107

SACCH Slow Associated Control Channel. 12, 33, 34, 101

SDCCH Stand-alone Dedicated Control Channel. 12

SIM Subscriber Identity Module, a smart card which stores the subscriber's shared key with the network, and implements the algorithms needed for authentication and session key derivation. 6–8, 18

SNR Signal-to-Noise-Ratio. 10

TCH Traffic CHannel. 11, 12

TCH/FS Full rate Traffic CHannel. 11, 34

TCH/HS Half rate Traffic CHannel. 11

TDM time division multiplexing, a method to share a RF channel by assigning the whole frequency range to every user during different non-overlapping time intervals. 5

TMDTO Time Memory Data Tradeoff, a kind of Time Memory Tradeoff optimized for the case when the attacker has several captured texts available to find out the key. 4, 30–33, 37, 41–43, 64, 79, 93, 96, 101, 102

TMSI Temporary Mobile Subscriber Identity. 7, 18

TMTO Time Memory TradeOff, a technique to invert a function by using precomputed tables to speed up the attack. 3, 4, 26, 29–31, 43, 50, 55, 56, 63, 68, 87, 88, 102, 119

TN Timeslot Number. 9

UMTS Universal Mobile Telecommunications System, 3rd. generation cellular system standarized by the 3rd Generation Partnership Project (3GPP). 2, 22, 33

VLR Visitor Location Register. 7

This page intentionally left blank

List of Tables

This page intentionally left blank

List of Figures

This is the last page
Compiled Tuesday 19th June, 2018.
http://iie.fing.edu.uy/