PRACTICAL CRYPTANALYSIS OF REAL-WORLD SYSTEMS

An Engineer's Approach

DISSERTATION

---

zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

Benedikt Driessen
Bochum, July 2013

Practical Cryptanalysis of Real-World Systems

Thesis Advisor	Prof. Christof Paar, Ruhr-Universität Bochum, Germany
External Referee	Prof. Ross Anderson, University of Cambridge, England
Date of submission	May 22, 2013
Date of defense	July 9, 2013
Date of last revision	July 16, 2013

To Ursula and Walter,
my parents.

Abstract

This thesis is dedicated to the analysis of symmetric cryptographic algorithms. More specifically, this document focuses on proprietary constructions found in four globally distributed systems. All of these constructions were uncovered by means of reverse engineering, three of them while working on this thesis, but only one by the author of this document. The recovered designs were subsequently analyzed and attacked. Targeted systems range from the GSM standard for mobile communication to the two major standards for satellite communication (GMR-1 and GMR-2) and finally a widely deployed digital locking system. Surprisingly, although much progress has been made in the area of specialized cryptography, our attacks on the newly reverse engineered systems show that even younger designs still suffer from severe design flaws.

The GSM stream ciphers A5/1 and A5/2 were reverse engineered and cryptanalyzed more than a decade ago. While the published attacks can nowadays be implemented and executed in practice, they also inspired our research into alternative, more efficient hardware architectures. In this work, we first propose a design to solve linear equation systems with binary coefficients in an unconventional, but supposedly fast way. Solving many of these equations is a fundamental step in most of the attacks developed for A5/2 and A5/1. Based on the proposed device, which solves equation systems over the rationals, we present a method to convert these solutions to solutions over the binaries.

Secondly, we describe the stream cipher A5-GMR-1, used in the GMR-1 satellite telecommunication standard, which was uncovered by reverse engineering. Then, the security of this cipher is analyzed and a highly practical ciphertext-only attack is developed. In a final step, the proposed attack is implemented and executed on the Thuraya satellite network. Together with the description of the equipment necessary for this operation, it is shown that voice privacy in GMR-1 cannot be trusted.

Extending the analysis of satellite phone communication standards to GMR-2 was only natural. The process of reverse engineering the stream cipher A5-GMR-2 is described, together with the design principles of a recursive disassembler for Blackfin DSPs. Then, the recovered cipher is described and a very efficient known-plaintext attack, which can be adjusted by means of a keystream/time trade-off, is presented.

Finally, an authentication scheme for digital locks is analyzed and broken with two distinct attacks. The SimonsVoss 3060 system is widely deployed and uses two proprietary constructions (key derivation and response computation) to authenticate transponders against digital locks. The designs include modifications to the well-known block cipher DES, but also a module that resembles a T-function. Combining a total of four weaknesses in key derivation and response computation with differential cryptanalysis and a recursive attack procedure on T-functions allows to open locks in practice.

In terms of insights offered due to the interdisciplinary nature of this thesis, the first part shows a creative effort to improve existing attacks. The second perspective highlights how practical cryptanalysis can be turned into a real-world attack on a global system. The chapter on GMR-2 is a detailed account of reverse engineering algorithms from complex embedded systems, while the final perspective documents the evolution of a practical cryptographic attack over a period of time—in parallel to the reverse engineering process itself.

Kurzfassung

Diese Arbeit ist der Analyse von symmetrischen, kryptographischen Algorithmen gewidmet. Das Dokument konzentriert sich im speziellen auf proprietäre, nicht öffentlich dokumentierte Verfahren die in vier global eingesetzten Systemen verwendet werden. Alle untersuchten Verfahren wurden durch den Prozess des Reverse-Engineering rekonstruiert, drei davon im Verlauf dieser Arbeit (aber nur eines durch dessen Autor). Die gefundenen Konstruktionen wurden analysiert und konsequent attackiert. Bei den angegriffenen Systemen handelt es sich um den globalen Telefoniestandard GSM, sowie die zwei Standards für Satellitentelefonie GMR-1 und GMR-2. Als viertes System wurde ein digitales Schließsystem angegriffen. Unsere Untersuchungen zeigen, dass auch neuere Entwicklungen weiterhin unter schweren Designfehlern leiden. Dies ist überraschend, zumal es zunehmend mehr öffentlichen Publikationen gibt, welche die üblichen Anforderungen und Probleme von real eingesetzten Systemen adressieren.

Die Stromchiffren A5/1 und A5/2, die im GSM Standard eingesetzt werden, wurden bereits vor einem Jahrzehnt rekonstruiert und gebrochen. Obgleich die kryptanalytischen Attacken bereits so weit fortgeschritten sind, dass sie praktikabel eingesetzt werden können, hat uns dies dazu motiviert, deren Effizienz weiter zu steigern. In dieser Arbeit wird ein Entwurf für eine Hardware basierend auf OP-Verstärkern vorgestellt, die binäre, lineare Gleichungssysteme über den rationalen Zahlen lösen kann. Das schnelle Lösen solcher Systeme ist der Kern vieler Angriffe auf Stromchiffren, auch im Fall von GSM. Um aber die erhaltenen Lösungen für einen echten Angriff nutzbar zu machen, musste eine Methode gefunden werden um rationalen Lösungen in den Raum der binären Zahlen zu transformieren. Eine solche Methode wird beschrieben, sie ist aber auch unabhängig von der vorgeschlagenen Rechnerarchitektur interessant.

Im Anschluss wird die Stromchiffre A5-GMR-1 vorgestellt, die aus einem Satellitentelefon für den GMR-1 Standard extrahiert wurde. Die Chiffre wird analysiert und mittels einer Attacke gebrochen, die lediglich verschlüsselte Daten benötigt. Damit ist die Attacke extrem praktikabel, was praktisch demonstriert wird. Dies, sowie detaillierte Angaben zur Konfiguration eines GMR-1 Netzwerks, zeigt, dass auf die Sprachverschlüsselung in GMR-1 nicht vertraut werden sollte.

Im Weiteren wird die Analyse auf den zweiten Satellitentelefonie-Standard ausgedehnt. Es wird der Prozess beschrieben, der die Rekonstruktion von A5-GMR-2, der Chiffre im GMR-2 Standard, aus der Firmware eines Telefons ermöglicht hat. Diese Beschreibung enthält die Designprinzipien für einen rekursiven Disassembler und Techniken, um die Chiffre (die nur einen Bruchteil der 300 000 disassemblierten Codezeilen ausmacht) zu finden. Das Ergebnis der anschließenden Analyse wird dokumentiert und eine sehr effiziente Attacke präsentiert, die sich anhand des verfügbaren Schlüsselstroms parametrisieren lässt.

Als letztes System wird in dieser Arbeit das digitale Schließsystem SimonsVoss 3060 betrachtet und das eingesetzte Authentifikationsverfahren beschrieben. Das Verfahren setzt zwei proprietäre Konstruktionen (für Schlüsselableitung und Antwortberechnung) ein, die auf einem modifizierten DES und einer T-Funktion-ähnlichen Methode basieren. Die Kombination von vier verschiedenen Schwächen in dem Design mit Techniken der differentiellen Kryptanalyse und einer rekursiven Angriffsprozedur auf die T-Funktion ermöglicht zwei verschiedene Attacken. Beide Attacken sind praktikabel und erlauben das unautorisierte Öffnen von Türschlössern.

Durch die interdisziplinäre Natur dieser Arbeit eröffnet jedes Kapitel eine eigene Perspektive auf Angriffe auf real eingesetzte Systeme: Im ersten Kapitel wird der kreative Ansatz existierende Attacken zu beschleunigen beschrieben. Das zweite Kapitel zeigt, wie Kryptanalyse im Rahmen von einem globalen System eingesetzt und praktisch angewandt werden kann. Das dritte Kapitel zeigt den Aufwand, der hinter Reverse-Engineering Vorhaben im Umfeld von hoch-komplexen, eingebetteten Systemen steckt. Das vierte Kapitel zeigt schließlich, wie sich kryptanalytische Methoden im Verlauf einer Analyse (d.h. mit wachsendem Wissen über ein System) verbessern können.

Acknowledgement

You know who "you" are.

CONTENTS

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Contribution and Organization . . . 5
1.3 Notation . . . 5

2 Solving Linear Equations with Analog Hardware **7**
2.1 Motivation . . . 7
2.2 Related Work . . . 8
2.3 A Circuit to Solve Linear Equations . . . 8
2.4 Transforming Matrices . . . 10
2.5 Converting Rational Solutions . . . 12
2.6 Discussion and Limitations of the Approach . . . 15
  2.6.1 Matrices with Cycles . . . 15
  2.6.2 Output Precision . . . 16

3 Security Analysis of the GMR-1 Standard **19**
3.1 Motivation . . . 19
3.2 Related Work . . . 20
3.3 Technical Background . . . 20
  3.3.1 Network Layout . . . 21
  3.3.2 Channels . . . 22
  3.3.3 Encoding and Encryption . . . 23
3.4 Reverse Engineering . . . 25
3.5 The A5-GMR-1 Stream Cipher . . . 25
  3.5.1 Structure . . . 25
  3.5.2 Mode of Operation . . . 26
3.6 Cryptanalysis . . . 27
3.7 A Real-World Attack . . . 30
  3.7.1 Recording TCH3 Data . . . 30
  3.7.2 Parity-check Matrix . . . 31
  3.7.3 Parameterization . . . 32

3.7.4 Implementation . . . 34
3.7.5 Results . . . 35
3.8 Discussion and Future Work . . . 36
3.8.1 Uplink Interception . . . 36
3.8.2 Real-time Decryption . . . 37

4 Security Analysis of the GMR-2 Standard 39
4.1 Motivation . . . 39
4.2 Related Work . . . 39
4.3 Technical Background . . . 39
4.3.1 Satphone Hardware . . . 39
4.3.2 The Blackfin DSP . . . 40
4.3.3 Software and Operating System . . . 42
4.4 Reverse Engineering . . . 43
4.4.1 Obtaining the DSP Firmware . . . 43
4.4.2 Developing a Blackfin Disassembler . . . 44
4.4.3 Finding the Cipher . . . 47
4.5 The A5-GMR-2 Stream Cipher . . . 49
4.5.1 Structure . . . 49
4.5.2 Mode of Operation . . . 51
4.6 Cryptanalysis . . . 52
4.7 Discussion and Future Work . . . 54

5 Security Analysis of the SimonsVoss 3060 Locking System 55
5.1 Motivation . . . 55
5.2 Related Work . . . 56
5.3 Technical Background . . . 56
5.4 Reverse Engineering . . . 57
5.5 The G2 Authentication System . . . 57
5.5.1 Keys & Protocol . . . 57
5.5.2 Cryptographic Primitives . . . 59
5.6 Cryptanalysis . . . 61
5.6.1 Differential Attack . . . 62
5.6.2 Active Attack . . . 68
5.6.3 Passive Attack . . . 69
5.7 Discussion and Future Work . . . 70
5.7.1 Comparison of Attacks . . . 71
5.7.2 Mitigations . . . 71

6 Conclusion 73

Acronyms

AES Advanced Encryption Standard

OPAMP Operational Amplifier

LES Linear Equation System

NFS Number Field Sieve

ASOL Analog Solver

QUCS Quite Universal Circuit Simulator

PSPICE Personal Simulation Program with Integrated Circuit Emphasis

TI Texas Instruments

GSO Geosynchronous Orbit

TDMA Time Division Multiple Access

TCH Traffic Channels

CCH Control Channels

FCCH Frequency Correction Channel

FACCH3 Fast Associated Control Channel-3

CCCH Common Control Channel

TCH3 Traffic Channel-3

ARFCN Absolute Radio-Frequency Channel Number

TN Timeslot Number

LFSR Linear Feedback Shift Register

GMR-1 Geo-Mobile Radio 1

GMR-2 Geo-Mobile Radio 2

CRC Cyclic Redundancy Check

SDR Software Defined Radio

LZ Lempel-Ziv

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

CPU Central Processing Unit

GSM Global System for Mobile Communications

UMTS Universal Mobile Telecommunications System

WLAN Wireless Local Area Network

PSTN Public Switched Telephone Network

DSP Digital Signal Processor

RTOS Real Time Operating System

LSB Least Significant Bit

MSB Most Significant Bit

DAU Data Arithmetic Unit

ALU Arithmetic Logic Unit

IDA Interactive Disassembler

DFU Device Firmware Upgrade

ELF Executable and Linking Format

API Application Programming Interface

DES Data Encryption Standard

RNG Random Number Generator

RKE Remote Keyless Entry

DST Digital Signature Transponder

ETSI European Telecommunications Standards Institute

GPS Global Positioning System

DECT Digital Enhanced Cordless Telecommunications

CoCOM Coordinating Committee on Multilateral Export Control

IC Integrated Circuit

CHAPTER 1

INTRODUCTION

In this chapter we introduce the overall motivation for this work, trace its chronology and describe the contributions. Furthermore, we outline the principal organization of the core chapters and introduce the notation common to all of them.

1.1 Motivation

The field of *IT Security* (and by inclusion the discipline of cryptology) has a unique characteristic that sets it apart from traditional engineering disciplines: there is no rigorous methodology for building practical, secure systems and its community clearly has a personality disorder[1]. While one group is always striving to improve overall security, the remainder is constantly trying to sabotage their efforts. Even worse, latter group is *competing* at who beats the constructive guys most efficiently! And there is another twist: the destructive guys are motivated by the assumption that there exists an equally skillful but more malicious and less vocal group, which not only breaks systems to break them, but actually tries to accomplish something by doing it. This entity, often[2] called “Oscar”, “Eve” or “Mallory” in the literature, could be a rogue government, an agency or even the Mafia. So, in order to prevent these others from reaching their ends, the good guys have to preemptively uncover and publish all means that might be used for attacking a security system. What sounds paradoxical is a widely accepted paradigm: since the lack of proper methodology implies a necessity for this cycle of creation, variation and destruction, all members of the community are happily motivating each others’ work.

What could be perceived as criticism is actually a description of the process which gave rise to this work. We were initially motivated by an ambitious goal: establishing a new speed record for cracking the over-the-air encryption of the Global System for Mobile Communications (GSM) standard, for which a plethora of attacks and implementations already existed. However, this would have only been a byproduct of an even more ambitious goal which was to accomplish this feat with innovative circuits based on analog hardware. While initial results were encouraging, scaling the attack hardware to the necessary size proved to be not feasible. At the same time, however, public interest in analyzing alternatives to GSM began to grow. This

---

[1] The scientifically correct term for the described phenomenon is actually *Dissociative Identity Disorder*, which denotes the mental state of a person having multiple, distinct and long-lasting identities.

[2] The name of an attacker often hints at his/her assumed capabilities, e.g., “Eve” is a passive attacker while “Mallory” is more malicious and can perform man-in-the-middle attacks.

was (partially) due to efforts of the open source community to implement the protocol stacks of global communication standards such as Digital Enhanced Cordless Telecommunications (DECT), GSM and Geo-Mobile Radio 1 (GMR-1). It became apparent that, while the security of GSM had been studied in depth for quite some time, this had yet to happen for satellite telecommunications standards, of which GMR-1 was a more prominent one. The lack of public documentation of the security mechanisms in GMR-1 and Geo-Mobile Radio 2 (GMR-2) was a hindrance to immediate analysis—which explains why there were no prior results published—and had to be overcome by reverse engineering. The successful completion of these two projects led to the participation in a third project of this kind: the analysis of the SimonsVoss locking system, which came into focus due to daily use. Besides being installed in thousands of buildings all over the world, the system is also present at Ruhr-University Bochum and was guarding the author's office.

In addition to the purely academic pursuit of knowledge, which is summed up in the metaphor of the creative cycle, the practical nature of our work added another dimension of motivation. Obviously, when arguing about algorithms, ciphers and mechanisms, this is typically done in a very abstract manner; some even insinuate that cryptologic research borders on the practically irrelevant. This changes, however, when security systems are actually implemented and the respective users have to rely on security guarantees they are told to expect. While, in the case of a digital locking system, a breach of cryptographic measures may lead to theft and loss of property, in the case of telecommunication systems, weak security (or more specifically: *confidentiality*) can have even more serious consequences. What may sound exaggerated can be substantiated by the following development that was caused by our work:

> Shortly after blogs, magazines and newspapers picked up on our analysis of GMR-1, the author was contacted by journalists, working in Syria, Afghanistan and Iraq, who where concerned for their safety. Their use of satellite systems for communication was motivated by the idea that local GSM networks could be monitored, while the satellite of the Thuraya network was out of reach for the respective regimes.
>
> However, after our eavesdropping demonstration it dawned on them that the assumed confidentiality only existed on paper—which made them look for better alternatives, motivated by a fear for their own wellbeing.

Clearly, Thuraya (one of the providers of GMR-1 satellite telephony) could have done a better job at encrypting voice data, but due to a lack of exposure, the problem of weak encryption never surfaced. Even worse, any interested party with sufficient motivation could have done what we did, while exploiting the results differently. So our work brought the existence of a problem, which might have been known for more than a decade, to public awareness. This example vividly illustrates the thought pattern and seeming paradox we described at the beginning of this section. In contrast to the case of GMR-1, where the problem of weak encryption should have been known (at least after the attacks on A5/2) the case is a bit different for SimonsVoss. We give them the benefit of the doubt: we assume that not only for their users, but also for the manufacturer themselves, our results are rather surprising. However, if considered as a learning process, this will allow SimonsVoss to improve their security in order to thwart any "really bad guys".

Subsuming the motivation—and adding the fact that studying, understanding and breaking systems is actually interesting—we hope that, by providing objective analysis, we can heighten awareness of users and manufacturers alike, which will ultimately lead to better and more secure systems.

1.2 Contribution and Organization

Besides this introduction and a final conclusion, this document consists of four chapters, which correspond to four independent projects. The chapters are organized such that they are self-contained with the aim of limiting the number of external resources required to understand what is presented. Each chapter is introduced by describing the respective motivation for undertaking the project, followed by examining previous work. Then, some necessary technical background is given. Finally, the core results and their implications are presented. Each chapter concludes with a discussion of the results and an outlook on future work.

In the following, we list the four chapters together with the respective contributions. Even though the projects in Chapters 3-5 were group efforts, this thesis only describes the original contributions of its author. This is possible because all projects can be broken down into disjunctive sub-projects, which build on each other, i.e., cryptanalysis can be understood without a detailed description of the reverse engineering process.

- **Chapter 2: An analog solver.**
  We propose an architecture based on Operational Amplifiers (OPAMPs) to solve binary linear equations in a fast way. Furthermore, we extend the basic design to overcome certain limitations. Finally, we present a method to convert any rational solution to an equation system over the binary numbers into the respective binary solution.
- **Chapter 3: Practical cryptanalysis of A5-GMR-1.**
  We describe the reverse engineered cipher from the GMR-1 satellite phone standard. We perform cryptanalysis of the cipher, and develop a novel ciphertext-only attack. Then we describe the layout of a typical GMR-1 network and describe hardware and software required to receive and record satellite-to-satphone communication. We report our attack results and, supplemented with measurements of satphone output power, show that privacy in GMR-1 is non-existent.
- **Chapter 4: Reverse engineering and cryptanalysis of A5-GMR-2.**
  We document the process of reverse engineering the A5-GMR-2 cipher from the firmware of a satphone. We describe the design of a disassembler and the techniques used to discover the cipher in a 300 000 line disassembly. We then present and analyze the design of the cipher and describe a very efficient and practical known-plaintext attack with a keystream/time trade-off.
- **Chapter 5: Practical cryptanalysis of a digital locking system.**
  Here, we describe the design of the authentication system used by the SimonsVoss 3060 locking system. We analyze the design and, based on the discovered flaws, present two distinct attacks—both being highly practical. Finally, we argue that the second attack is optimal with regard to the exploitation of observable transmissions.

1.3 Notation

Here, we shortly describe the common notation used throughout this work.

Symbol	Description
$\mathbb{N}_+$	The set of natural numbers without 0, i.e., $\mathbb{N} \setminus \{0\}$
$\mathbb{Z}_+$	The set of integers without 0, i.e., $\mathbb{Z} \setminus \{0\}$
$\|\mathbb{V}\|$	Denotes the number of elements in the set $\mathbb{V}$
$c_{l-1}c_{l-2}...c_0.d_0d_1d_2...d_{m-1}$	Denotes the binary expansion of a rational number with $c_i, d_j \in \{0,1\}$.
$\mathbf{A}$	A matrix
$\|\mathbf{A}\|$	The determinant of $\mathbf{A}$

$\mathbf{I}_n$	The $n \times n$ identity matrix
$\mathbf{1}_n$	An $n \times n$ matrix with all coefficients being 1
$\vec{b}$	A vector
$\vec{0}_n$	A vector of n elements, all being 0
$(x_0, x_1, ..., x_{n-1})_2$	Description of x as string of n bits
$(X_0, X_1, ..., X_{n-1})_{2^m}$	Description of X as string of n elements with m bits each, for $m > 1$
$x_{\langle a \rangle}$	Denotes a single bit of the bitstring x at index a
$x_{\langle a..b \rangle}$	Denotes a substring of the bitstring x, ranging from index a to b
$x\|\|y$	The concatenation of two bitstrings x and y
0^n	A string of n zero bits
$\alpha, \beta, \gamma, ...$	Variables with Greek letters typically denote a hypothesis/guessed value
$\psi(), \xi()$	Mappings with Greek letters are typically introduced to simplify descriptions
$\mathscr{F}(), \mathscr{G}(), ...$	These mappings typically denote entire function blocks (e.g., a DES)
$X = \mathscr{F}(K; Y)$	This typically denotes encryption of Y to X under key K
$x \ggg a$	Shift the bitstring x by a bits to the right

Table 1.1: Notation

CHAPTER 2

SOLVING LINEAR EQUATIONS WITH ANALOG HARDWARE

This chapter presents our results on building a dedicated hardware device based on Operational Amplifiers (OPAMPs), which is capable of solving certain equation systems over $\mathbb{F}_2$. Additionally, we introduce a method to convert a rational solution for a binary equation system into a solution over $\mathbb{F}_2$.

2.1 Motivation

Solving binary Linear Equation System (LES) of the form

$$\mathbf{A}\vec{x} = \vec{b} \quad \text{with} \quad \mathbf{A} \in \mathbb{F}_2^{n\times n} \quad \text{and} \quad \vec{b}, \vec{x} \in \mathbb{F}_2^n$$

in n unknowns is a common problem and appears in numerous research and technical disciplines. In the field of cryptography, a special form of this problem arises when attacking stream ciphers. Certain attacks, such as attacks on A5/1 and A5/2 [Gol97, PFS00] (which are used for voice encryption in GSM), require solving of a very large number (approx. 2^{40}) of LES over $\mathbb{F}_2$. These LES can be solved with the help of Gaussian elimination (and more sophisticated variants), which can easily be implemented in software and hardware. However, solving an equation system typically has cubic complexity, which is unsatisfying in practice when the size of the set of LES is considerable.

To (potentially dramatically) speed up the process of solving these important equation systems, we have experimented with an analog hardware architecture that can solve certain instances of the mentioned problems in a very fast manner with very limited resources. While it is unclear, what the actual time complexity of this architecture is, a single step towards the solution promises to be orders of magnitude faster than any conventional implementation.

A second contribution of this chapter is motivated by intermediate results. Based on the assumption that we are given a rational solution to $\mathbf{A}\vec{u} = \vec{b}$ with $\vec{u} \in \mathbb{Q}$ (either as a result of applying our device, or by other means), we have developed a method to convert this solution into a binary representation, which allows us to solve the LES over $\mathbb{F}_2$.

2.2 Related Work

Our work was inspired by previous attempts at using exotic hardware architectures which exploit certain physical properties to solve computationally intense problems. Most notably, concepts such as TWIRL and TWINKLE have been proposed a decade ago [Sha99, LS00, ST03]. They can be used for the sieving step of the Number Field Sieve (NFS) [LLMP93] and are—in combination with “classical hardware”—assumed to be able to factor 512-bit RSA moduli. Although these devices have never been built (at least not to the author’s knowledge), it is supposedly possible to do so for $10 000 000.

The idea of the hypothetical devices is to shift the most expensive part of the NFS method from digital hardware into the analog domain. More specifically, finding appropriate smooth numbers for the sieving step is done with the help of an array of LEDs and a light sensor. In TWINKLE, the LEDs, which are switched on and off in a specific manner, are emitting light proportional to the logarithms of successive but small prime numbers. The sensor operates as instantaneous adder of these intensities and signals when a certain threshold has been reached. The LEDs switched on in a particular moment, together with the product of their associated primes, indicate a smooth number.

2.3 A Circuit to Solve Linear Equations

The advantage of TWINKLE/TWIRL is that they exploit a physical property to offload computation, which inspired us to experiment with an electrical device we call Analog Solver (ASOL). The core idea of our device is to use a network of switched OPAMPs to solve binary linear equation systems. The basic principle of our circuit is given by the observation that OPAMPs can be used as inverting adders for input voltages u_i, i.e.,

$$u_{\mathtt{out}} = -R_{\mathtt{add}} \left(\frac{u_1}{R_1} + \frac{u_2}{R_2} + \frac{u_3}{R_3} + \cdots + \frac{u_n}{R_n} \right). \tag{2.1}$$

See Figure 2.1 for the corresponding circuit. By choosing all resistors to be equal and inverting the polarity of the input voltages, we can simplify Equation 2.1 to the following form,

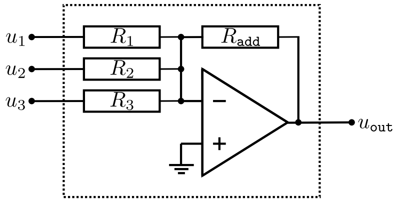


Figure 2.1: Inverting adder with single OPAMP

$$-u_{\mathtt{out}} = u_1 + u_2 + u_3 + \cdots + u_n$$

which is a simple addition of the input voltages. Based on this idea, we can construct a circuit of n OPAMPs with a switched feedback network, which computes a solution to the equation system

$$\mathbf{A}\vec{u} = -\vec{b}U_{\mathtt{in}}, \tag{2.2}$$

where $U_{\mathtt{in}}$ is the input voltage of the circuit.

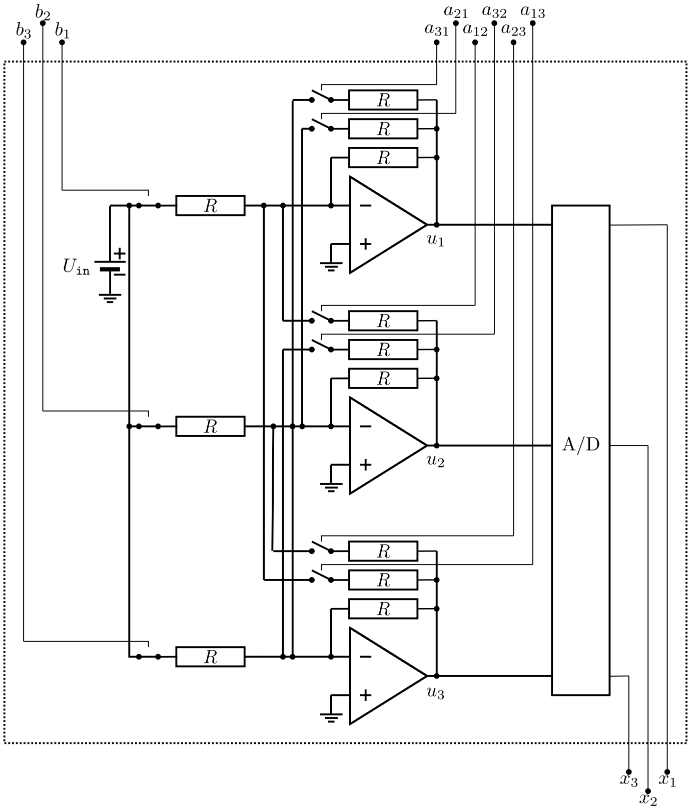


Figure 2.2: Principle of constructing basic ASOL for three unknowns

In this network, feedback loops between the OPAMPs are closed according to the coefficients of an equation system which is to be solved—the solution to the equation system can be measured (after some oscillation) as voltage output of the OPAMPs. An example circuit for $n = 3$ is shown in Figure 2.2. The setting of the switches is determined by the corresponding binary coefficients of $\mathbf{A}$ and $\vec{b}$ (an open switch represents the binary value 0, and a closed one the value 1). Looking at each of the OPAMPs separately, we can write down equations for the expected output voltage of each OPAMP:

$$\begin{aligned} u_1 &= -b_1 U_{\mathtt{in}} - a_{1,2}u_2 - a_{1,3}u_3 \\ u_2 &= -a_{2,1}u_1 - b_2 U_{\mathtt{in}} - a_{2,3}u_3 \\ u_3 &= -a_{3,1}u_1 - a_{3,2}u_2 - b_3 U_{\mathtt{in}} \end{aligned}$$

By re-arranging each equation accordingly, we easily see that the circuit represents an LES of quadratic form, i.e.,

$$\begin{aligned} u_1 + a_{1,2}u_2 + a_{1,3}u_3 &= -b_1 U_{\mathtt{in}} \\ a_{2,1}u_1 + u_2 + a_{2,3}u_3 &= -b_2 U_{\mathtt{in}} \\ a_{3,1}u_1 + a_{3,2}u_2 + u_3 &= -b_3 U_{\mathtt{in}} \end{aligned}$$

where the matrix $\mathbf{A}$ has a non-zero diagonal (cf. Section 2.4). The example circuit solves a particular[1] equa-

---

[1] The configuration of the switches indicates that the LES solved by ASOL shown in the figure is $\mathbf{I}_3\vec{u} = (1, 1, 1)^T$, where $\mathbf{I}_3$ is the 3×3 identity matrix.

tion system with three unknowns and binary coefficients which is exactly what we stated in Equation 2.2. When the circuit has converged to a stable operating point after all relevant switches have been set, the output voltages of the OPAMPs will approximate the solution of the LES, i.e.,

$$\vec{u} = -U_{\mathtt{in}} \left(\mathbf{A}^{-1} \vec{b} \right).$$

Without delving further into the specifics of this device, and postponing a discussion of its limits until Section 2.6, we will assume for the remainder of this chapter that the device will eventually be able to solve equations over the rationals. Consequently, we developed a way to convert a rational solution to a solution over $\mathbb{F}_2$, which will be presented after making a short detour into a discussion about solving matrices where the diagonal elements are not 1.

2.4 Transforming Matrices

Using the device, as discussed in the previous section, imposes a certain constraint on the form of the matrix $\mathbf{A}$. In order to trivially assign the rows of the matrix to the OPAMPs of ASOL, the matrix must be in a form where each diagonal element is 1. This form can always be found for quadratic matrices with full rank. However, since our aim is to ultimately avoid any variant of Gaussian elimination, we decided to look into computationally trivial alternatives.

The idea of our approach is to embed an $n \times n$ matrix $\mathbf{A}$ which has full rank, but is not of the desired form, into a matrix $\mathbf{B}$ of size $2n \times 2n$. This matrix is derived from $\mathbf{A}$ in a way such that the desired form is guaranteed. Constructing $\mathbf{B}$ is straightforward and presented in the following lemma.

Lemma 1. *Let $\mathbf{1}_n$ be the $n \times n$ matrix where all coefficients are 1, $\vec{0}_n$ is a vector of n zeros and $\mathbf{I}_n$ the $n \times n$ identity matrix. If the LES $\mathbf{A}\vec{x} = \vec{b}$, $\mathbf{A} \in \mathbb{F}_2^{n \times n}, \vec{x}, \vec{b} \in \mathbb{F}_2^n$ is uniquely solvable over $\mathbb{F}_2$, constructing an equation system $\mathbf{B}\vec{y} = \vec{c}$ with*

$$\mathbf{B} = \begin{pmatrix} \mathbf{1}_n & \mathbf{A} \oplus \mathbf{1}_n \\ \mathbf{I}_n & \mathbf{I}_n \end{pmatrix} \quad \textit{and} \quad \vec{c} = \begin{pmatrix} \vec{b} \\ \vec{0}_n \end{pmatrix} \tag{2.3}$$

yields a uniquely solvable system, which adheres to the restraint of having a non-zero diagonal. More importantly, the solution $\vec{y}$ to this equation system contains the solution $\vec{x}$ to the original problem in a trivial manner, i.e.,

$$\vec{x} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Proof. In the following, all coefficient additions (and subtractions) are performed in $\mathbb{F}_2$. By construction, the diagonal of $\mathbf{B}$ consists only of 1s. It is easy to see that the equation system can be brought into the following form

$$\begin{pmatrix} \mathbf{0}_n & \mathbf{A} \\ \mathbf{I}_n & \mathbf{I}_n \end{pmatrix} \vec{y} = \begin{pmatrix} \vec{b} \\ \vec{0}_n \end{pmatrix}$$

by adding the lower n rows to the upper n rows in an appropriate manner. Using the same lower rows, the next step gives this form

$$\begin{pmatrix} \mathbf{A} & \mathbf{0}_n \\ \mathbf{I}_n & \mathbf{I}_n \end{pmatrix} \vec{y} = \begin{pmatrix} \vec{b} \\ \vec{0}_n \end{pmatrix},$$

since addition and subtraction over $\mathbb{F}_2$ are similar. In this form it is obvious that the top n equations represent the original equation system, which can be solved completely independent of the artificially introduced, new variables $y_n, ..., y_{2n-1}$. Thus, the top n elements of $\vec{y}$ will be the solution to $\mathbf{A}\vec{x} = \vec{b}$. □

Based on the previously shown technique of embedding the matrix **A**, we can extend the original ASOL design so that it is possible to apply matrix configurations, which do not have a non-zero diagonal. Figure 2.3 shows the extended design (for three unknowns), which, naturally, now relies on six OPAMPs. The basic operating principle is the same; the difference is, however, that the diagonal elements a_{11}, a_{22}, a_{33} can also be set. Please note that all matrix coefficients need to be inverted (to reflect the addition of **A** and $\mathbf{I}_3$).

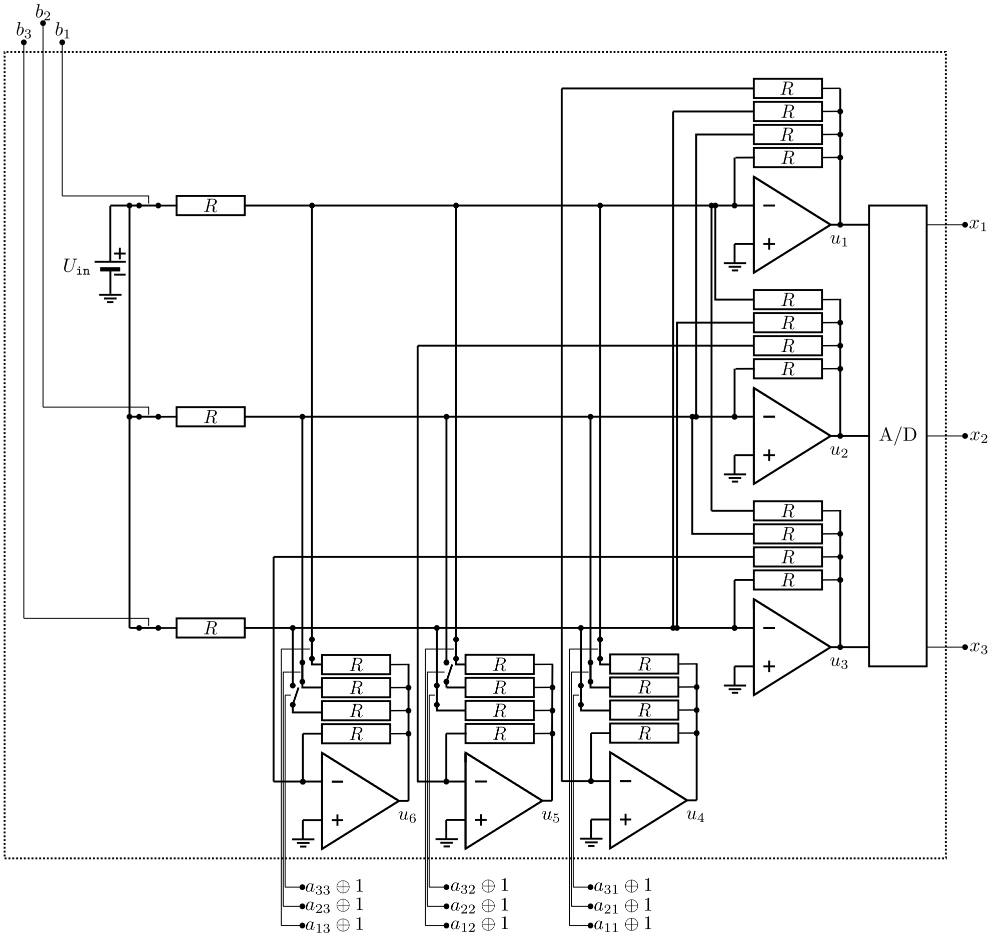


Figure 2.3: Principle of constructing augmented ASOL for three unknowns

It should be noted that due to the way **A** is embedded into **B**, any matrix **B** has an undesired property: due to the $\mathbf{1}_n$ matrix in the upper left of **B**, the first n OPAMPs are heavily interconnected, forming cycles which lead to undesirable behavior of the circuit. Please refer to Section 2.6 for a more detailed discussion.

2.5 Converting Rational Solutions

In this section we present a method to "interpret" a rational solution $\vec{u} = \mathbf{A}^{-1}\vec{b}$ with $\mathbf{A} \in \mathbb{F}_2^{n\times n}, \vec{b} \in \mathbb{F}_2^n$ and $\vec{u} \in \mathbb{Q}^n$ in order to find a binary solution for the same equation system. The rational solution $\vec{u}$ can be obtained either as voltages measured in our hypothetical device or by other means. The only thing that matters is that the solution is "sufficiently" precise, a requirement that will become clear in Section 2.6. Furthermore, we assume that the LES is uniquely solvable over $\mathbb{F}_2$, therefore the determinant of $\mathbf{A}$ must be $|\mathbf{A}| = 1$ over $\mathbb{F}_2$, and thus $|\mathbf{A}| \equiv 1 \bmod 2$ when computing the determinant over the rationals. The latter fact is important and will be used extensively in the remainder of this section.

Now we will describe how we can convert the rational solution to a binary vector $\vec{x} \in \mathbb{F}_2^n$ which satisfies $\mathbf{A}\vec{x} = \vec{b}$ over $\mathbb{F}_2$. We will do this with the help of three lemmata which will lead to the conversion method.

Lemma 2. *If the LES $\mathbf{A}\vec{u} = \vec{b}$, $\mathbf{A} \in \mathbb{F}_2^{n\times n}, \vec{b} \in \mathbb{F}_2^n$ is uniquely solvable over $\mathbb{F}_2$ and $\mathbb{Q}$, then computing the solution over $\mathbb{Q}$ yields a vector $\vec{u} \in \mathbb{Q}_\triangledown^n$ with*

$$\mathbb{Q}_\triangledown = \left\{ \frac{p}{q} : p, q \in \mathbb{Z}, q \equiv 1 \bmod 2 \right\}.$$

Proof. According to Cramer's Rule, we know that a rational solution $\vec{u} = \mathbf{A}^{-1}\vec{b}$ can be computed by the use of determinants, i.e.,

$$\vec{u} = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{pmatrix} = \frac{1}{|\mathbf{A}|} \begin{pmatrix} |\mathbf{A}_1| \\ |\mathbf{A}_2| \\ \vdots \\ |\mathbf{A}_n| \end{pmatrix}$$

where $|\mathbf{A}| \in \mathbb{Z}_+$ denotes the determinant of the binary matrix $\mathbf{A}$ and $|\mathbf{A}_i| \in \mathbb{Z}$ denotes the determinant of $\mathbf{A}$ where the i-th column has been replaced by $\vec{b}$. Given the condition of unique solvability of the equation system over $\mathbb{F}_2$, computing $|\mathbf{A}|$ over $\mathbb{F}_2$ cannot be zero. This must also hold when computing $|\mathbf{A}|$ over $\mathbb{Z}$ and applying the modulo operator only as last step—since both procedures must yield the same result.

Therefore $|\mathbf{A}| \equiv 1 \bmod 2$ holds and thus all potential rational solutions $\vec{u}$ must be in the set $\mathbb{Q}_\triangledown^n$. ☐

Considering Lemma 2 and with the help of the ring homomorphism φ, which is defined as

$$\varphi : \mathbb{Q}_\triangledown \mapsto \mathbb{F}_2 \quad \text{with} \quad \varphi\left(\frac{p}{q}\right) = \frac{p \bmod 2}{q \bmod 2} = p \bmod 2$$

we could directly deduce a solution $\vec{x} \in \mathbb{F}_2^n$ for a particular solution $\vec{u} \in \mathbb{Q}_\triangledown^n$ by simply applying the φ operator to the vector of quotients component-wise, i.e.,

$$\vec{x} = \varphi(\vec{u}) = \begin{pmatrix} \varphi(|\mathbf{A}_1|) \\ \varphi(|\mathbf{A}_2|) \\ \vdots \\ \varphi(|\mathbf{A}_n|) \end{pmatrix} \equiv \begin{pmatrix} |\mathbf{A}_1| \bmod 2 \\ |\mathbf{A}_2| \bmod 2 \\ \vdots \\ |\mathbf{A}_n| \bmod 2 \end{pmatrix}.$$

However, when measuring the voltage output of ASOL, we do only have fixed-point number representations of the solution vector $\vec{u}$ and no information about its quotients. Therefore no direct modulo reduction is possible and we have to find another way to compute $\varphi(\vec{u})$.

Let us now consider how to convert $u_i \in \mathbb{Q}_\nabla$, which is the i-th element of $\vec{u}$ given as rational number in base-2, to a representation $x_i \in \mathbb{F}_2$. For some $l \in \mathbb{N}_+$ let u_i be given in the following form

$$u_i = c.d_0d_1d_2d_3\cdots \quad \text{with} \quad d_j \in \{0,1\} \quad \text{and} \quad c = \sum_{k=0}^{l-1} 2^k c_k \in \mathbb{N} \quad \text{with} \quad c_k \in \{0,1\} \tag{2.4}$$

which we will call the *binary expansion* of u_i. If, for some fixed $m \in \mathbb{N}_+$, we have

$$d_0 = d_{jm}, \quad d_1 = d_{jm+1}, \quad d_2 = d_{jm+2}, \quad \dots, \quad d_{m-1} = d_{jm+(m-1)} \quad \text{for all} \quad j \in \mathbb{N}_+$$

we call the binary expansion of u_i *purely periodic* (with a period length of m) and can re-write Equation 2.4 to

$$u_i = c.\overline{d_0d_1d_2d_3\cdots d_{m-1}},$$

as there are no non-periodic digit-patterns after the decimal point.

Lemma 3. *If the LES* $\mathbf{A}\vec{u} = \vec{b}$, $\mathbf{A} \in \mathbb{F}_2^{n\times n}, \vec{b} \in \mathbb{F}_2^n$ *is uniquely solvable over* $\mathbb{F}_2$ *and* $\mathbb{Q}$*, then the binary expansion of any element* u_i *of the rational solution* $\vec{u} \in \mathbb{Q}_\nabla^n$ *is purely periodic.*

Proof. This proof follows the argumentation found in [YP04]. Suppose we have $u_i = p/q \in \mathbb{Q}_\nabla$ where $p = |\mathbf{A}_i| = cq + r_0$ and $q = |\mathbf{A}|$ is odd. Since q is odd, it holds that $q \mid (2^m - 1)$ for some $m \in \mathbb{N}_+$ and hence

$$q = \frac{2^m - 1}{l} \quad \text{with} \quad m, l \in \mathbb{N}_+.$$

Since

$$0 \le \frac{r_0}{q} \le 1 \quad \text{and} \quad 0 \le (2^m - 1)\frac{r_0}{q} \le 2^m - 1 \quad \text{with} \quad (2^m - 1)\frac{r_0}{q} = lr_0$$

where $d = lr_0$ is an integer we can write

$$d = \sum_{j=1}^{m} 2^{m-j} d_{j-1} = d_0d_1d_2\cdots d_{m-1} \quad \text{with} \quad d_j \in \{0,1\}$$

as a bit-string with m bits. Since

$$\begin{aligned} & d = (2^m - 1)\frac{r_0}{q} \\ \Leftrightarrow \quad & \frac{r_0}{q} = 2^{-m}d + 2^{-m}\frac{r_0}{q} \quad \text{and} \quad 2^{-m}d = 0.d_0d_1d_2\cdots d_{m-1} \end{aligned}$$

we easily see that the quotient of r_0 and q exhibits a recursive behavior, i.e.,

$$\begin{aligned} \frac{r_0}{q} &= 0.d_0d_1d_2\cdots d_{m-1} + 2^{-m}\frac{r_0}{q} \\ &= 0.d_0d_1d_2\cdots d_{m-1}d_0d_1d_2\cdots d_{m-1} + 2^{-2m}\frac{r_0}{q} \\ &= 0.d_0d_1d_2\cdots d_{m-1}d_0d_1d_2\cdots d_{m-1}d_0d_1d_2\cdots d_{m-1} + 2^{-2m}\frac{r_0}{q}, +2^{-3m}\frac{r_0}{q} \\ &= \dots \end{aligned}$$

and therefore the binary expansion of

$$u_i = \frac{|\mathbf{A}_i|}{|\mathbf{A}|} \quad \text{with all} \quad u_i \in \mathbb{Q}_\triangledown$$

is purely periodic. $\square$

If we actually measure a voltage representation of the solution for a given LES over the rationals and the results are given as purely periodic binary expansions of the form

$$c_{l-1}c_{l-2}\cdots c_0.d_0d_1d_2d_3\cdots d_{m-1}d_0d_1d_2d_3\cdots d_{m-1}\cdots,$$

we “only” need to recover the c_0 and d_{m-1} bit of all u_i in order to interpret $\vec{u} = \mathbf{A}^{-1}\vec{b}$ as a solution over $\mathbb{F}_2^n$. Given c_0 and d_{m-1} of a particular u_i we have an alternative way to compute $\varphi(u_i) = \varphi(p/q) \equiv p \bmod 2$, which will be discussed now.

Lemma 4. *Given $\mathbf{A}\vec{u} = \vec{b}$ and $\mathbf{A}\vec{x} = \vec{b}$ with $\mathbf{A} \in \mathbb{F}_2^{n\times n}, \vec{b} \in \mathbb{F}_2^n$, a rational solution $\vec{u} \in \mathbb{Q}_\triangledown^n$ and also a binary solution $\vec{x} \in \mathbb{F}_2$. Consider the binary expansion of any arbitrarily chosen element $u_i \in \mathbb{Q}_\triangledown$ with $0 \le i < n$ where, for some $l, m \in \mathbb{N}_+$, we find that*

$$u_i = c_{l-1}c_{l-2}\cdots c_0.\overline{d_0d_1d_2d_3\cdots d_{m-1}} \quad \textit{with all} \quad c_j, d_k \in \{0,1\} \quad \textit{for} \quad 0 \le j < l, 0 \le k < m$$

is purely periodic. For the corresponding entry $x_i \in \{0,1\}$ in the binary solution vector $\vec{x}$ the following relation holds:

$$x_i = \varphi(u_i) = \varphi(p/q) \equiv p \bmod 2 = c_0 \oplus d_{m-1}.$$

Proof. Suppose we have $u_i \in \mathbb{Q}_\triangledown$ in purely periodic form with

$$u_i = c_{l-1}\cdots c_0.\overline{d_0\cdots d_{m-1}} \quad \text{for some} \quad l, m \in \mathbb{N}_+$$

which we convert to a quotient via a schoolbook trick, i.e.,

$$\begin{aligned} & 2^m u_i = c_{l-1}\cdots c_0d_0\cdots d_{m-1}.\overline{d_0\cdots d_{m-1}} \\ \Leftrightarrow\quad & 2^m u_i - u_i = c_{l-1}\cdots c_0d_0\cdots d_{m-1} - c_{l-1}\cdots c_0 \\ \Leftrightarrow\quad & u_i = \frac{c_{l-1}\cdots c_0d_0\cdots d_{m-1} - c_{l-1}\cdots c_0}{2^m - 1}. \end{aligned}$$

Now we know that

$$u_i = \frac{c_{l-1}\cdots c_0d_0\cdots d_{m-1} - c_{l-1}\cdots c_0}{2^m - 1} \quad \text{but also} \quad u_i = \frac{|\mathbf{A}_i|}{|\mathbf{A}|}.$$

In both cases, the denominators are odd (cf. Lemma 2) and therefore

$$\varphi(u_i) \equiv |\mathbf{A}_i| \bmod 2 = c_{l-1}\cdots c_0d_0\cdots d_{m-1} - c_{l-1}\cdots c_0 \bmod 2 = c_0 \oplus d_{m-1}$$

holds. $\square$

There is also a special case where deducing $\vec{x}$ is easy: if $\mathbf{A} \in \mathbb{F}_2^{n\times n}$ is in upper triangular form, we know that the determinant is the product of its diagonal elements. When the corresponding equation system is uniquely solvable over $\mathbb{F}_2$, it holds that

$$|\mathbf{A}| = 1.$$

Therefore, for any $u_i \in \mathbb{Z}$ we can compute $\varphi(u_i)$ by looking at the least significant bit of u_i, which is an integer, i.e.,

$$x_i = \varphi(|\mathbf{A}_i|) \equiv u_i \bmod 2.$$

2.6 Discussion and Limitations of the Approach

We consider the conversion trick to be an interesting curiosity, for which—besides its intended use as part of ASOL—there might be other uses as well, i.e., in cases where computing over the rationals with sufficient precision is more desirable, than directly over the binaries. In that sense, this is a result that has merits on its own, possibly in a wider scope, even when not directly applicable as intended.

That said, the basic design of ASOL, as it is presented in Section 2.3, was initially simulated with Quite Universal Circuit Simulator (QUCS), an open-source[2] program for Linux. At a later stage, we used Personal Simulation Program with Integrated Circuit Emphasis (PSPICE), a more professional program. During these simulations, we discovered that the basic design suffers from two problems:

1. If the matrix contains cycles (see below), the circuit will never converge to a solution and oscillate "forever".
2. In order to apply our conversion as presented in Section 2.5, the output precision of the circuit needs to be sufficiently high.

Additionally, it is currently not known whether the inherent oscillatory nature of the proposed architecture ultimately leads to a cubic number of oscillations before a solution is obtained and therefore is, in that sense, en par with traditional implementations. However, even without an advantage in complexity of steps/oscillations, it is likely that a *fully working ASOL* would outperform classical architectures such as CPUs.

We will now shortly discuss each of the two mentioned problems, what causes them and how we tried to address them—with "some" success. Future work may naturally begin where we have failed.

2.6.1 Matrices with Cycles

If the graph corresponding to a matrix **A**, which may also be understood as (binary) adjacency matrix, contains cycles, ASOL will oscillate and (at least according to our simulations) never converge to the correct solution. Figure 2.4 shows the directed graph of such an exemplary 4 × 4 matrix (bold coefficients indicate the formed cycle):

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & \mathbf{1} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & \mathbf{1} \\ \mathbf{1} & 0 & 0 & 1 \end{pmatrix}$$

In the figure, each node corresponds to one row of the matrix, i.e., node "1" corresponds to the first row, node "2" to the second row, etc. Incoming vertices of a node are assigned based on whether the respective column is non-zero. If the directed graph contains a cycle, it is possible to start anywhere in the cycle and walk the vertices back to the starting point. Having a cycle in **A** means, in the basic ASOL design, that the output of one OPAMP is fed into another OPAMP whose output feeds another OPAMP etc., until the output of the last OPAMP in the cycle is used as input to the first one. Having one or more feedback loops of this form in the matrix (and thus the circuit) leads to an oscillatory behavior for which, at least in simulation, we did not observe any convergence to the final solution. We have, unsuccessfully, attempted two methods to tame this behavior:

1. Find a transformation of the LES into a different LES (or a set of LES) without cycles. Sadly, we were not able to find such a transformation which is also trivial to compute (we want to avoid any Gaussian elimination and variants thereof).

---

[2] See `http://qucs.sourceforge.net/`.

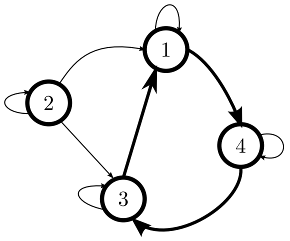


Figure 2.4: Directed graph of the 4 × 4 matrix **A** with cycle of vertices

2. We have experimented with applying the coefficients of the matrix to the circuit by setting the switches in certain timing patterns. The hope here was to set some switches, let the circuit settle to a stable operating point and then, successively, add more and more coefficients while avoiding oscillations. We were unable to find an easy-to-determine (or even universal) pattern of applying the coefficients.

As stated in Section 2.4, the augmented design where **A** is embedded into a matrix of twice the size, invariably suffers from this problem because of the $\mathbf{1}_n$ matrix in the top left corner.

2.6.2 Output Precision

The clear limitation, given a device which is indeed really fast and able to solve all equation systems in n variables, is the requirement of sufficient precision—a notion which has been mentioned a few times already. "Sufficient" precision allows us to obtain the last bit of the binary expansion of each element of the rational vector $\vec{u}$. The overall precision is determined by two factors:

1. The precision of the "computation" depends on the precision of the elements of the circuits, i.e., how close they behave to the ideal case which is expressed by solving the equation system mathematically. Relevant elements are the switches, OPAMPs and resistors but also connections and wires.
2. Once the device has settled on a stable operating point, the resulting voltages at the outputs of the OPAMPs have to be digitized to allow the application of the conversion step.

We have not examined the precision of the actual computation, but the attainable precision of the analog-to-digital conversion is limited by the actual converter used. A typical converter, such as the Texas Instruments (TI) ADS1232 provides 24 bits of digital output precision, while introducing only an insignificant amount of noise by itself. Our argumentation in Section 2.5 implies that the smaller the determinant of **A** (over the rationals), the shorter the length of the binary expansion and hence the lower the required precision of the analog-to-digital conversion.

Since we ultimately want to speed up cryptographic attacks and examine the feasibility of our approach, we have experimentally generated some of the binary linear equation systems which are used to execute Golic's attack [Gol97] against A5/1. For his attack, Golic needs to generate and solve more than 2^{40} equation systems of size $n = 64$ over $\mathbb{F}_2$. From the over 2^{40} equation systems that need to be generated and solved, we have computed (by randomly guessing parts of the key) 100 000 matrices. For all of these matrices, we have counted which determinants occur how often. The result of our examination is depicted in Figure 2.5 in a graph that shows the percentage of determinants (y-axis), which is lower than a specific value (x-axis). Observing the graph quite surprisingly[3] reveals that nearly 50% of all matrices have a determinant below 300, while 90% are still below 3 000. Initially, this analysis was not only surprising but also encouraging.

---

[3] We assume that this result is due to the inherent structure of the generated matrices.

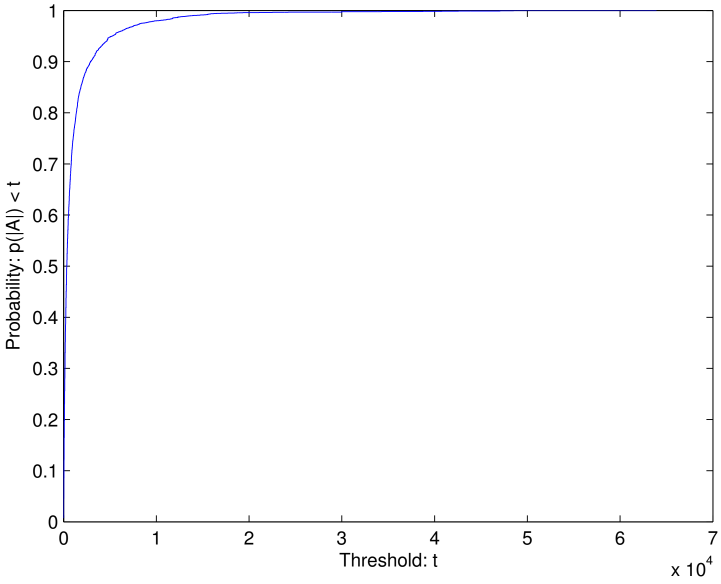


Figure 2.5: Probability the determinant in Golic's A5/1 attack is below a threshold t

Now we assume that, for all output voltages, we can obtain the first 24 bits of the binary expansion. In order to be able to detect the period of the expansion (and thus find the last bit of the periodic sequence, which is required by our conversion method), the period must appear at least once in the first 12 bits. Of those randomly chosen equation systems which we have generated for Figure 2.5, only 86 566 are actually uniquely solvable. A solution is a vector of 64 rational values, but of all those $86\,566 \cdot 64 = 5\,540\,224$ values obtained, only a tiny fraction of 0.16% rational values have such a binary expansion. This implies that the combined approach of using ASOL and the conversion method is not suitable for Golic's attack—although there might be problems (with extremely sparse matrices) which might benefit from our approach.

At this point, we realized that using the purely analog approach for improving attacks on A5/1 may not be feasible. As a result of this, in conjunction with a student from EPFL in Lausanne, a "folder" circuit was developed [Zha11]. This circuit basically performs modulo-2 reductions on the voltage level, by mapping input voltages which are a multiple of 2 to $0V$, and all other voltages to $1V$. Placing this folder at the output of each OPAMP, just before the feedback resistors, basically allows the circuit to compute over the binaries. For more details and results of an initial analysis, the interested reader is referred to the thesis. However, even with the help of these folder elements, the inherent oscillatory nature of the basic design remained as a major problem and ultimately discouraged further research in this area.

CHAPTER 3

SECURITY ANALYSIS OF THE GMR-1 STANDARD

In this chapter we present the results of our analysis of the telecommunications standard used for Thuraya satellite communication systems (and others). We document the entire process from cryptanalysis to actually executing a real-world attack, which allows to receive and decrypt over-the-air communications in negligible time.

3.1 Motivation

Mobile communication systems have revolutionized the way we interact with each other. Instead of depending on landline connections with fixed locations, we can talk to other people wherever we are and also transmit data from (almost) arbitrary locations. Especially GSM has evolved into an extremely large-scale system; with more than four billion subscribers in 2011, it is the most widely deployed standard for cellular networks.

Cellular mobile networks require a so-called *cell site* to create a cell within the network. The cell site provides all the infrastructure necessary for exchanging radio signals between mobile handsets and the provider network. For example, a typical cell site contains one or more sets of transmitters/receivers, antennas, digital signal processors to perform all computations, a Global Positioning System (GPS) receiver for timing and other control electronics. The cells within a network have only a limited operating distance and, thus, a certain proximity to a cell site is always necessary to establish a connection to the mobile network.

In practice, however, it is not always possible to be close to a cell site and there are many use cases in which no coverage is provided. Workers on an oil rig or on board of a ship, researchers on a field trip in a desert or near the poles, aid workers in remote areas or areas that are affected by a natural disaster, journalists working in politically unstable areas, or certain military and governmental undertakings are a few of many uses cases where terrestrial cellular networks are not available. To overcome this limitation, satellite systems were introduced to provide telephony and data services based on telecommunications satellites. In such systems, the mobile handset (typically called *satphone*) communicates directly with satellites in orbit. Thus, coverage can be provided without the need of a highly interconnected infrastructure on the Earth's surface.

The GMR-1 family of European Telecommunications Standards Institute (ETSI) standards for satellite telecommunications were derived from GSM. In fact, their specifications are an extension of the GSM standard, where certain aspects of the specification are adjusted for satphone settings. This protocol family is supported by several providers (e.g., Thuraya, SkyTerra, TerreStar) and has continuously undergone revisions to support a broader range of services.

While the specification documents are available online, no information about security aspects are published. More precisely, it was not publicly known which encryption algorithms are actually used to secure the communication channels between a satphone and a satellite. Since an attacker can easily eavesdrop on the radio signals between satphone and satellite, even at some distance, it is obvious that weak encryption would be a serious threat to confidentiality. At this point, it was thus unclear what effort would be needed by an attacker to actually intercept telephony and data services for common satphone systems, which motivated our research in this area.

3.2 Related Work

Satellite telecommunication systems are related to terrestrial cellular systems since the GMR-1 standards are derived from the GSM standard. We can thus leverage work on the analysis of cellular systems for our security analysis as discussed in the following.

In 1994, an email by Ross Anderson to the `sci.crypt` forum contained the source code of a variant of the A5/1 algorithm. While the comments in the code indicate that the algorithm "arrived anonymously in two brown envelopes", Anderson is the first to propose an attack on the leaked design. In 1997, Golic proposed a different attack, based on solving many linear equation systems (cf. Section 2.1) in the bits of the key [Gol97]. However, since parts of the feedback configuration of the actual cipher were still unknown, the cryptographic community only became "really" interested after the full design was uncovered in 1999. Briceno *et al.* published an implementation of the GSM A5/1 and A5/2 algorithms, which they apparently obtained by reverse engineering a GSM handset [BGW99]. However, no details about the analysis process were ever published and it remains unclear how the algorithms were actually derived. Motivated by the disclosure of the ciphers, there has been much work on the analysis of their security and the implementation of the resulting attacks in hardware and software [BD00, BSW00, EJ03, BER07, NP09, DKS10]. As we discovered (cf. Section 3.5), the cipher used in GMR-1 is related to the A5/2 algorithm, but its configuration is different. Our attack on this algorithm builds on existing ideas for A5/2 [PFS00, BBK08], which we extended to enable a time-ciphertext trade-off.

In addition to the academic community, also the open source movement became interested in GMR-1 and telecommunication technologies in general. The Osmocom project[1] is an umbrella for several sub projects for the diverse standards. More specifically, the OsmocomGMR project[2] set out to re-implement the protocol stack of GMR-1. Besides an initial drop of source code, the project's website documents information gathered about internals of satphones and the configuration of the Thuraya network.

3.3 Technical Background

In this section we introduce the basic background regarding Thuraya's network layout and channel setup. This information is relevant for understanding the description of our real-world attack.

---

[1] See `http://osmocom.org/`.

[2] See `http://gmr.osmocom.org/trac/`.

3.3.1 Network Layout

Thuraya implements the GMR-1 standard and provides satellite telephony for most of Europe, the Middle East, North, Central and East Africa, Asia and Australia. To achieve this coverage, the network consists of two overlapping regions, each handled by a different satellite. Thuraya satellites are operating in Geosynchronous Orbit (GSO), where they do not stay on a position but follow a fixed movement pattern, typically an analemma. Currently, there are two operational[3] satellites named Thuraya-2 and Thuraya-3. The former is relevant here, since it is centered on the Middle East and supplies most of Europe as well as a large portion of the African continent with connectivity, see Figure 3.1[4].

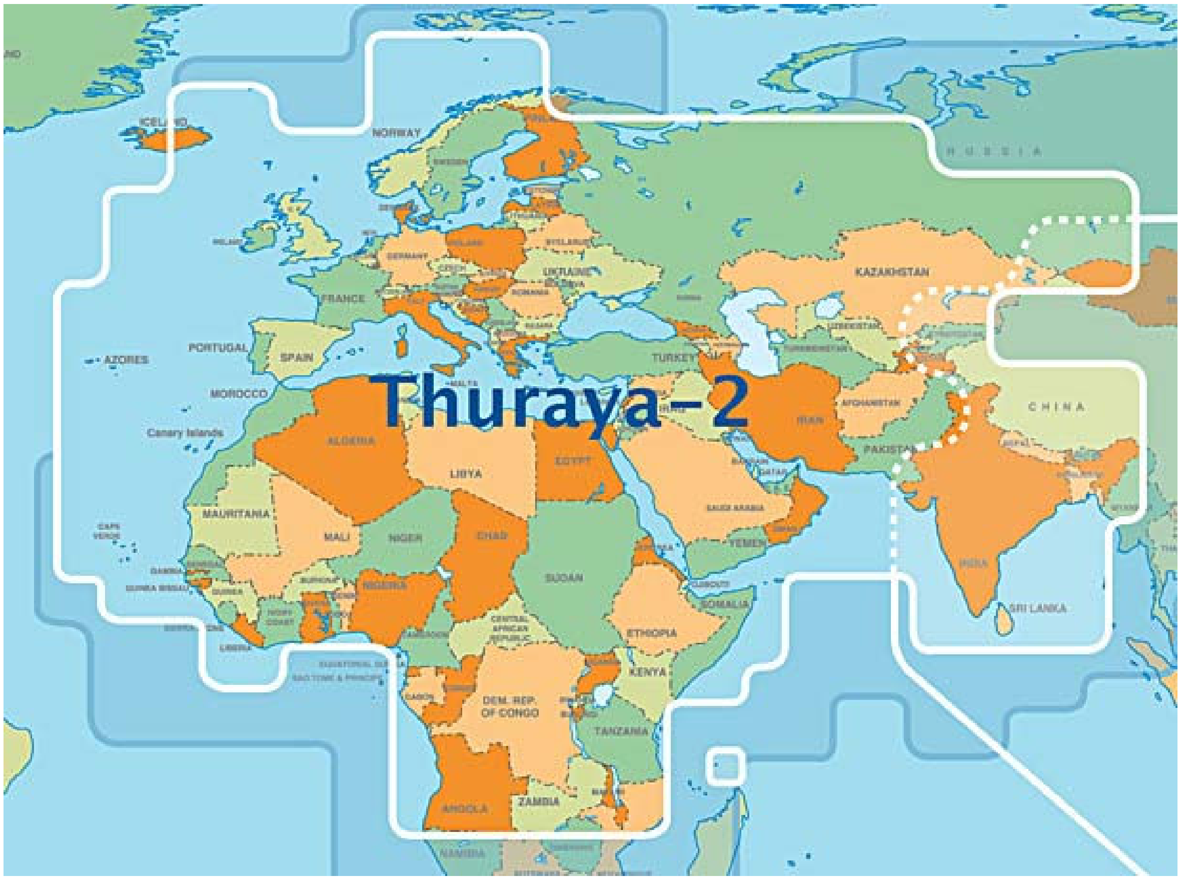


Figure 3.1: Network coverage of the Thuraya-2 satellite

Thuraya offers a diverse range of products for fixed installations, handhelds (i.e., satphones) and even solutions for the maritime environment. With the help of Thuraya, voice, fax and IP-based data can be transmitted where “traditional” infrastructures (e.g., GSM, Universal Mobile Telecommunications System (UMTS), Wireless Local Area Network (WLAN), etc.) are not available. In addition to the satellites, a set of terrestrial gateways and one primary gateway (located in Sharjah, United Arab Emirates) handle the entire network as depicted in Figure 3.2. Gateway stations provide the connectivity to tethered networks, e.g., telephone calls to a landline are forwarded to the Public Switched Telephone Network (PSTN) and

---

[3] Thuraya-1 has ceased to operate in May 2007 and has been moved to “junk orbit”.

[4] See `http://www.peter2000.co.uk/aviation/satcomms/index.html`.

enable maintenance and configuration purposes. For this so-called *ground segment*, conventional frequency bands (3.400 – 3.625 GHz and 6.425 – 6.725 GHz) signals are used. The *user segment* operates on L-band carriers assigned to spotbeams, which are Thuraya's equivalent to cells in GSM (albeit covering far more area). In the L-band, the frequency band from 1.525 to 1.559 GHz is assigned for space-to-earth (downlink) communication while the uplink operates between 1.6265 and 1.6605 GHz. Uplink and downlink are divided into 1087 paired carrier frequencies with a spacing of 31.25 KHz.

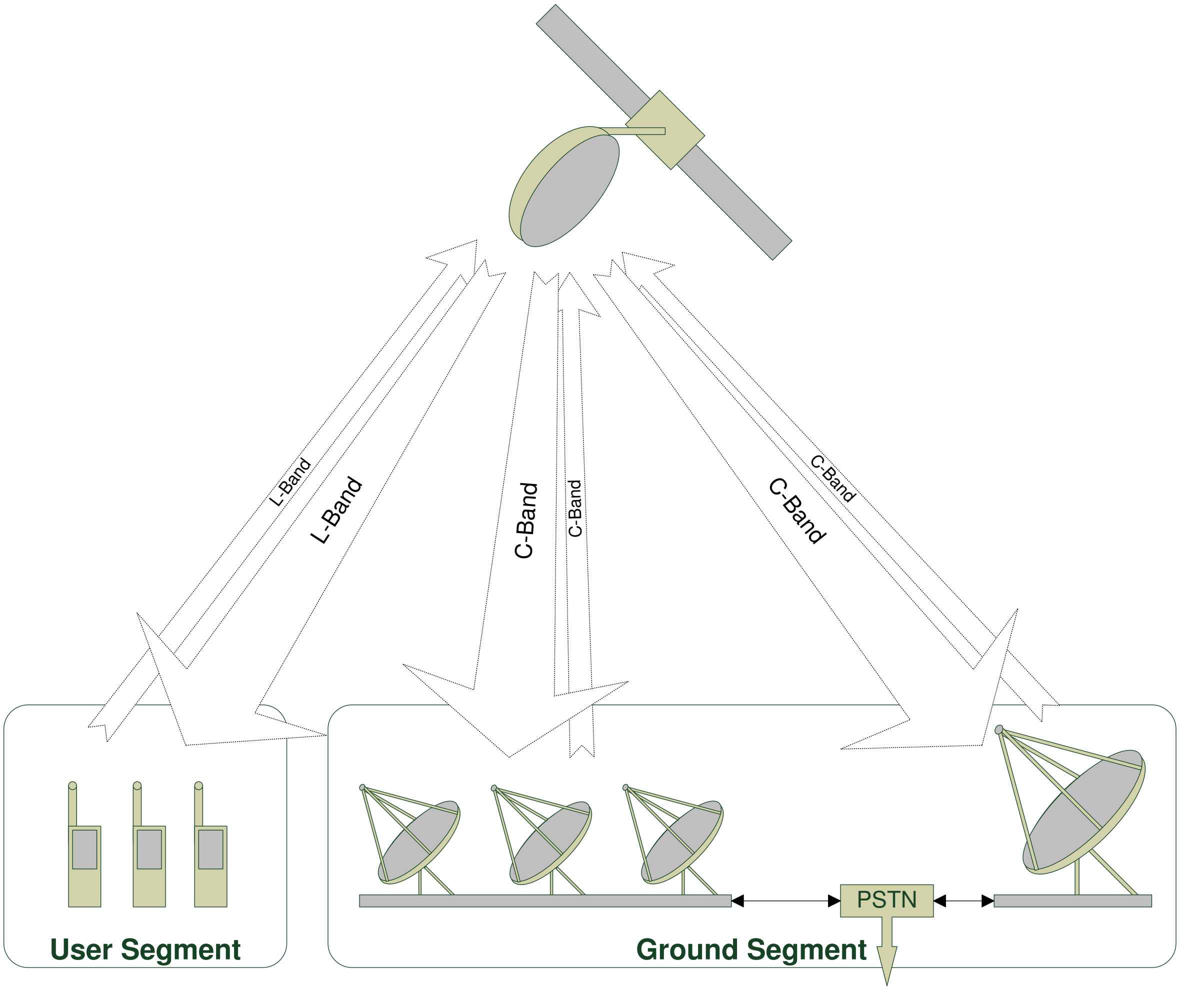


Figure 3.2: Layout of the Thuraya network

3.3.2 Channels

Just like in GSM, the Time Division Multiple Access (TDMA) time slot architecture, which partitions a carrier frequency into disjunct timeslots of a fixed length, is employed in Thuraya. Figure 3.3 shows how a TDMA frame (middle) is split into 24 timeslots (bottom) of $\frac{5}{3}$ ms each. 16 TDMA frames are grouped together into one multiframe (top), which is 640 ms long. Furthermore, multiframes are consolidated into a superframe, of which 4 496 comprise a hyperframe. It should be noted that each TDMA frame has a 19-bit TDMA frame number; numbering starts at 0 and the number is incremented with each new frame.

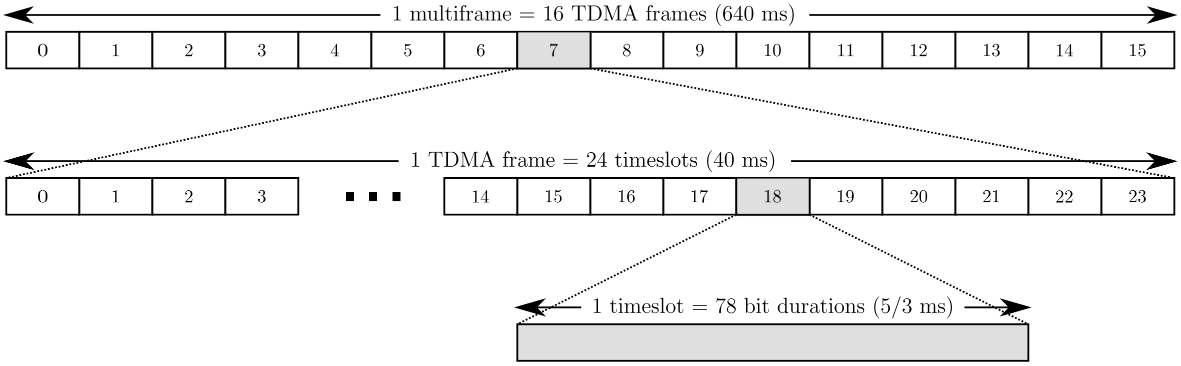

1 multiframe = 16 TDMA frames (640 ms)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
1 TDMA frame = 24 timeslots (40 ms)
0
1
2
3
14
15
16
17
18
19
20
21
22
23
1 timeslot = 78 bit durations (5/3 ms)

Figure 3.3: TDMA architecture of GMR-1 networks

Several *logical channels* (called *channels* from now on) can share a carrier frequency by being mapped to different timeslots. Due to this architecture, a channel is uniquely determined by a frequency and a sequence of Timeslot Numbers (TNs). There are different types of channels, but all are either Traffic Channels (TCH) for voice, fax or IP-based data, or Control Channels (CCH). Data is sent over these channels in the form of frames (i.e., blocks of consecutive bits) that are encoded (cf. Subsection 3.3.3) by adding redundancy to protect against transmission failures. Frames are enumerated by their respective TDMA frame numbers, which we simply call *frame numbers* from now on. For some channels, the encoded data is subsequently encrypted, see Subsection 3.3.3. The encoded (and encrypted) data is finally modulated before it is transmitted via the phone's antenna. The coding scheme differs from channel to channel and is dependent on the respective reliability requirements as defined in the various documents of the standard.

Specific channels relevant for our attack are the Frequency Correction Channel (FCCH), the Common Control Channel (CCCH) and the Traffic Channel-3 (TCH3). The FCCH is initially (e.g., after power-up) used by the satphone to determine its relative time and frequency error in order to synchronize with the satellite. The CCCH is used to send information to the phone when a new channel (e.g., TCH3) needs to be established[5]. These assignment messages contain an Absolute Radio-Frequency Channel Number (ARFCN) and a TN, which is, as explained above, all that is required to use the channel. After TCH3 has been set up on the uplink and downlink, it can be used to transmit speech data.

In Subsection 3.7.1 we will go more into the process of translating ARFCNs into frequencies and how we can actually tune to the TCH3 channel.

3.3.3 Encoding and Encryption

As mentioned before, all data has to be encoded before it is sent to travel the distance of 36 000 km between ground and satellite. Encoding always increases the size of the encoded data, thus adding redundancy

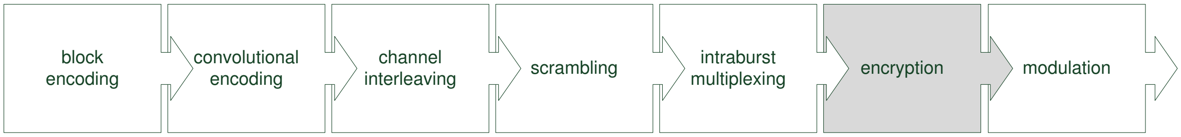


Figure 3.4: Generic encoding (and encryption) scheme for information in the GMR-1 system

---

[5] TCH3 is typically established at the beginning of a call.

which allows error detection and possibly correction. Figure 3.4 shows the pipeline of encoding, encryption and modulation. Please note that, depending on the channel over which data is sent, the actual encoding parameters and modulation schemes may differ. The general encoding procedure is as follows:

> Each channel uses the following sequence and order of operations:
>
> - The information bits are encoded with a systematic block code, i.e., Cyclic Redundancy Check (CRC), building words of information and parity bits;
> - these information and parity bits are encoded with a convolutional code, building the coded bits;
> - the coded bits are reordered and potentially interleaved over multiple bursts;
> - the interleaved bits are scrambled and, in some cases, multiplexed with other bits (before or after encryption);
>
> — [ETS02, p. 11]

To protect against eavesdropping of data sent over the air, the encoded bits are encrypted with a proprietary cipher. However, doing it the way it is done in GMR-1 leads to a property we exploit for our real-world attack (cf. Section 3.6).

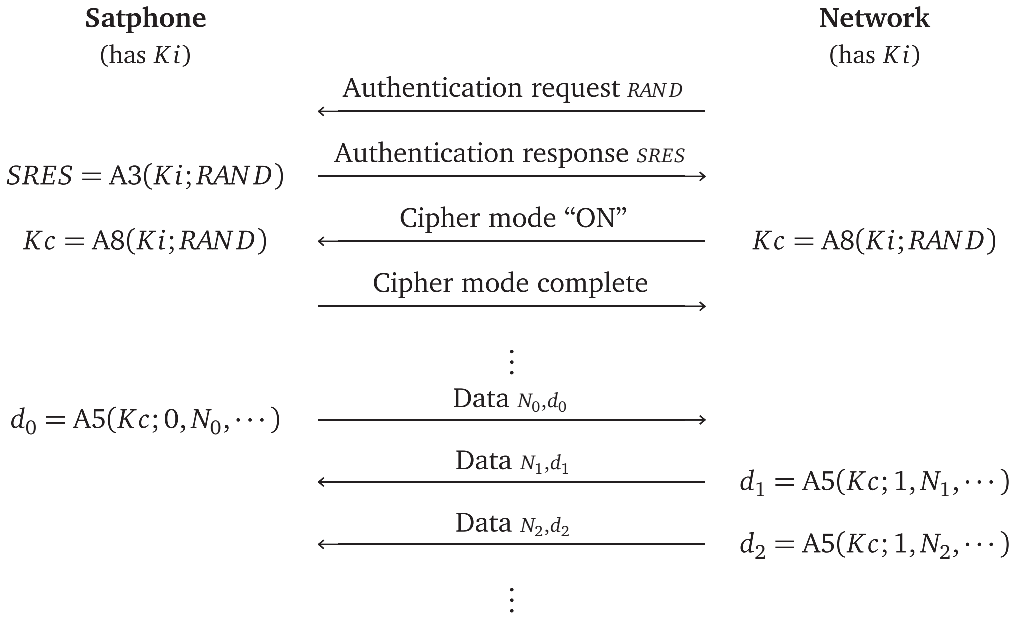

Satphone
(has Ki)
Network
(has Ki)
Authentication request $RAND$
$SRES = \mathrm{A3}(Ki; RAND)$
Authentication response $SRES$
Cipher mode “ON”
$Kc = \mathrm{A8}(Ki; RAND)$
$Kc = \mathrm{A8}(Ki; RAND)$
Cipher mode complete
$\vdots$
$d_0 = \mathrm{A5}(Kc; 0, N_0, \cdots)$
Data N_0, d_0
Data N_1, d_1
$d_1 = \mathrm{A5}(Kc; 1, N_1, \cdots)$
Data N_2, d_2
$d_2 = \mathrm{A5}(Kc; 1, N_2, \cdots)$
$\vdots$

Figure 3.5: Protocol for establishing a session key Kc between satphone and provider network

Encryption in GMR-1 is performed on a per-session basis, i.e., for the duration of one call a session key Kc is established (see Figure 3.5 for a sketch of the respective protocol). This key is derived from a challenge $RAND$ sent by the network and a long-term key Ki, known only to the satphone and network. In the specifications, the key derivation algorithm is denoted as A8, which serves the same role as the A8 function in GSM. On the handheld side it is implemented on the phone's SIM card, where also its long-term key is stored.

With the help of the session key, data can be encrypted with an algorithm denoted as A5, or, A5-GMR-1. This algorithm is a stream cipher which encrypts data based on the session key and its TDMA frame number and direction (i.e., whether it is received or sent by a satphone). A second property of the protocol is that it simultaneously authenticates the phone against the network—with the help of the A3 algorithm. Overall, this protocol is strikingly similar to what is specified for GSM.

3.4 Reverse Engineering

The stream cipher used in GMR-1 was reverse engineered from a software update package for the Thuraya SO-2510 by a colleague from our research institute, Ralf Hund. The firmware was disassembled with a commercial disassembler; the cipher was found with the help of a heuristic: subfunctions in the disassembly were ranked according to their percentage of XOR/SHIFT instructions, which revealed the four routines, each implementing one Linear Feedback Shift Register (LFSR).

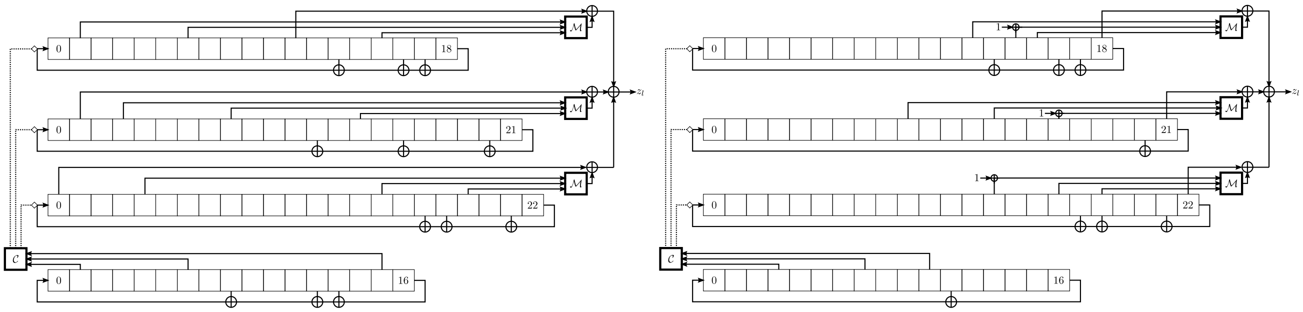
More details on this procedure and the general methodology, which is related to the approach presented in Section 4.4, can be found in our joint paper [DHW+12] and the respective, upcoming Ph.D. thesis.

3.5 The A5-GMR-1 Stream Cipher

In this section we present the cipher used in the GMR-1 standard. The cipher is termed A5-GMR-1 and surprisingly similar to a known design, used in GSM.

3.5.1 Structure

The cipher used in GMR-1 is a typical stream cipher (cf. Figure 3.6a); its design is a modification of the A5/2 cipher (cf. Figure 3.6b), which is used in GSM networks. The cipher uses four LFSRs which are clocked irregularly. We call these registers, starting with the topmost one, R_1, R_2, R_3 and R_4.



(a) A5-GMR-1 (b) A5/2

Figure 3.6: Comparison of stream ciphers used in GMR-1 and GSM

Comparing A5/2 and A5-GMR-1, we see that the structure of the registers is basically the same, i.e., the LFSRs have the same length and only three of them are connected to the output. However, the feedback

		A5-GMR-1			A5/2		
	Size	**Feedback polynomial**	**Taps**	**Final**	**Feedback polynomial**	**Taps**	**Final**
R_1	19	$x^{19}+x^{18}+x^{17}+x^{14}+1$	1,6,15	11	$x^{19}+x^{5}+x^{2}+x+1$	12,14,15	18
R_2	22	$x^{22}+x^{21}+x^{17}+x^{13}+1$	3,8,14	1	$x^{22}+x+1$	9,13,16	21
R_3	23	$x^{23}+x^{22}+x^{19}+x^{18}+1$	4,15,19	0	$x^{23}+x^{15}+x^{2}+x+1$	13,16,18	22
R_4	17	$x^{17}+x^{14}+x^{13}+x^{9}+1$	1,6,15	-	$x^{17}+x^{5}+1$	3,7,10	-

Table 3.1: Configuration of the LFSRs in A5-GMR-1 and A5/2

polynomials and also the selection of input taps for the non-linear majority-function $\mathcal{M}$ with

$$\mathcal{M} : \{0,1\}^3 \mapsto \{0,1\}$$
$$\left(x_{\langle 0\rangle}, x_{\langle 1\rangle}, x_{\langle 2\rangle}\right)_2 \mapsto x_{\langle 2\rangle}x_{\langle 1\rangle} \oplus x_{\langle 2\rangle}x_{\langle 0\rangle} \oplus x_{\langle 0\rangle}x_{\langle 1\rangle}$$

were changed. Also, the positions of the bits that are XORed with the respective outputs of the majority functions are different. In A5-GMR-1, all feedback polynomials are pentanomials, which is not the case for A5/2, as shown in Table 3.1.

3.5.2 Mode of Operation

Next we focus on the mode of operation. Clocking a single LFSR means evaluating its respective feedback polynomial and using the resulting bit to overwrite the leftmost position of the LFSR, after shifting its current state by one bit to the right. When the cipher is clocked for the l-th time with irregular clocking active, the following happens:

1. The irregular clocking component $\mathcal{C}$ evaluates all taps of R_4, the remaining registers are clocked accordingly, i.e.,
    a) Iff $\mathcal{M}(R_{4_{\langle 1\rangle}}, R_{4_{\langle 6\rangle}}, R_{4_{\langle 15\rangle}}) = R_{4_{\langle 15\rangle}}$, register R_1 is clocked.
    b) Iff $\mathcal{M}(R_{4_{\langle 1\rangle}}, R_{4_{\langle 6\rangle}}, R_{4_{\langle 15\rangle}}) = R_{4_{\langle 6\rangle}}$, register R_2 is clocked.
    c) Iff $\mathcal{M}(R_{4_{\langle 1\rangle}}, R_{4_{\langle 6\rangle}}, R_{4_{\langle 15\rangle}}) = R_{4_{\langle 1\rangle}}$, register R_3 is clocked.
2. The taps of R_1, R_2 and R_3 are evaluated and one bit of keystream is output accordingly, i.e.,

$$z_l = \mathcal{M}\left(R_{1_{\langle 1\rangle}}, R_{1_{\langle 6\rangle}}, R_{1_{\langle 15\rangle}}\right) \oplus \mathcal{M}\left(R_{2_{\langle 3\rangle}}, R_{2_{\langle 8\rangle}}, R_{2_{\langle 14\rangle}}\right) \oplus \mathcal{M}\left(R_{3_{\langle 4\rangle}}, R_{4_{\langle 15\rangle}}, R_{3_{\langle 19\rangle}}\right) \oplus R_{1_{\langle 11\rangle}} \oplus R_{2_{\langle 1\rangle}} \oplus R_{3_{\langle 0\rangle}}$$

   is generated.
3. R_4 is clocked.

The cipher is operated in two modes, *initialization* and *generation* mode. Running the cipher in the former mode includes setting the initial state of the cipher, which is done in the following way:

1. All four registers are set to zero.
2. A 64-bit initialization vector $\alpha = \left(\alpha_{\langle 0\rangle}, \ldots, \alpha_{\langle 63\rangle}\right)$ is computed by XORing the bits of the 19-bit frame number N and 64-bit session key K, i.e.,

$$\begin{aligned}\alpha = f(K,N) = & K_{\langle 0..2\rangle} || K_{\langle 3\rangle} \oplus N_{\langle 6\rangle} || K_{\langle 4\rangle} \oplus N_{\langle 7\rangle} || K_{\langle 5\rangle} \oplus N_{\langle 8\rangle} || K_{\langle 6\rangle} \oplus N_{\langle 9\rangle} || K_{\langle 7\rangle} \oplus N_{\langle 10\rangle} || K_{\langle 8\rangle} \oplus N_{\langle 11\rangle} || \\ & K_{\langle 9\rangle} \oplus N_{\langle 12\rangle} || K_{\langle 10\rangle} \oplus N_{\langle 13\rangle} || K_{\langle 11\rangle} \oplus N_{\langle 14\rangle} || K_{\langle 12\rangle} \oplus N_{\langle 15\rangle} || K_{\langle 13\rangle} \oplus N_{\langle 16\rangle} || K_{\langle 14\rangle} \oplus N_{\langle 17\rangle} || \\ & K_{\langle 15\rangle} \oplus N_{\langle 18\rangle} || K_{\langle 16..21\rangle} || K_{\langle 22\rangle} \oplus N_{\langle 4\rangle} || K_{\langle 23\rangle} \oplus N_{\langle 5\rangle} || K_{\langle 24..59\rangle} || K_{\langle 60\rangle} \oplus N_{\langle 0\rangle} || K_{\langle 61\rangle} \oplus N_{\langle 1\rangle} || \\ & K_{\langle 62\rangle} \oplus N_{\langle 2\rangle} || K_{\langle 63\rangle} \oplus N_{\langle 3\rangle}\end{aligned}$$

3. The bits of α are re-ordered to α' with

$$\alpha' = \left(\alpha_{\langle 15\rangle}, \alpha_{\langle 14\rangle}, ..., \alpha_{\langle 0\rangle}, \alpha_{\langle 31\rangle}, \alpha_{\langle 30\rangle}, ..., \alpha_{\langle 16\rangle}, \alpha_{\langle 47\rangle}, ..., \alpha_{\langle 32\rangle}, \alpha_{\langle 63\rangle}, ..., \alpha_{\langle 48\rangle}\right)$$

   and clocked into all four registers in this order. To clock one bit of α' into R_1, its feedback polynomial is evaluated and the resulting bit then clocked into R_1, *after* XORing it with the α' bit. The same bit of α' is also clocked into R_2, R_3 and R_4. Then, the second bit of α' is clocked into all four registers in this manner and so on. While doing this, irregular clocking is deactivated, i.e., all registers are clocked for each bit of α'.
4. The least-significant bits of all four registers are set to 1, i.e., $R_{1_{\langle 0\rangle}} = R_{2_{\langle 0\rangle}} = R_{3_{\langle 0\rangle}} = R_{4_{\langle 0\rangle}} = 1$.

We denote the whole initialization process by g, where

$$\beta = \underbrace{\beta_{\langle 0..18\rangle}}_{R_1} || \underbrace{\beta_{\langle 19..40\rangle}}_{R_2} || \underbrace{\beta_{\langle 41..63\rangle}}_{R_3} || \underbrace{\beta_{\langle 64..80\rangle}}_{R_4} = g(K, N),$$

is a 81-bit string, comprised of the consecutive bits of the four initialized registers. After all registers are initialized, irregular clocking is activated and the cipher is clocked 250 times. The resulting output bits are discarded.

Now the cipher is switched into generation mode and clocked for $2m$ times, generating one bit of keystream at a time. Here, m is the length of an encrypted frame. Depending on the direction[6] bit, either the first half or the second half of the $2m$ keystream bits is used for encryption/decryption. We denote the l-th keystream bit by $z^{(N)}_{\langle l\rangle}$, where $0 \leq l < 2m$ is the number of irregular clockings (after warm-up) and N the frame number that was used for initialization. Since our cryptanalysis will focus on the downlink, we denote the continuous keystream for frames $N, N+1, ...$ (as decrypted by the phone) by z, where

$$z = z^{(N)}_{\langle 0..m-1\rangle} || z^{(N+1)}_{\langle 0..m-1\rangle} || z^{(N+2)}_{\langle 0..m-1\rangle} || z^{(N+3)}_{\langle 0..m-1\rangle} || z^{(N+4)}_{\langle 0..m-1\rangle} || ...$$

is the concatenation of the first halves of $z^{(N)}, z^{(N+1)}, ...$ respectively. The choice of m depends on the type of channel, for which the data is encrypted or decrypted. For the TCH3 channel, each frame has a length of $m = 208$ bits.

3.6 Cryptanalysis

The attack we present here is inspired by previous attacks [PFS00, BBK03] on A5/2. Please note that we treat bitstrings as column vectors[7] and vice versa. We now briefly review the several weaknesses (which are either due to the design of the cipher or due to the use of the cipher) which we exploit for our attack on GMR-1:

1. Given R_4, the clocking behavior of A5-GMR-1 is uniquely determined.
2. Since the inputs to each majority component are only from one register, one bit of keystream can always be expressed as an easy-to-linearize quadratic equation over $\mathbb{F}_2$.
3. In GMR-1, encryption is applied after encoding (which is entirely linear in $\mathbb{F}_2$) and scrambling[8].

---

[6] The first m bits are used on the handset's side for decryption, on the provider network side for encryption

[7] Coding theory traditionally uses row vectors, which is why the equations we obtain for encoding and decoding look slightly different.

[8] While encoding adds redundancy, scrambling is used to "[...] randomize the number of 0s and 1s in the output bit stream." [ETS02, p. 22].

4. For each two keystreams generated by the same session key but different frame numbers, the respective initial states are linearly related by the XOR-differences of the frame numbers.

Due to the first and second observation and given enough keystream bits for a particular frame N, we can guess R_4. Based on the guess, we clock the entire cipher several times and generate a linearized system of equations over $\mathbb{F}_2$, i.e.,

$$\mathbf{A}x = z^{(N)}.$$

These equations describe keystream bits as linear combinations of terms which either are individual bits or products of two bits from the initial state of R_1, R_2 and R_3. If we guess R_4 correctly and $\mathbf{A}$ has full rank, solving the equation system gives the correct initial state which can easily be used to obtain the session key. Please note that, even if the session key is fixed, for different frame numbers not only the keystream but also the initial state and the matrix describing its relation to the keystream will be different. The required number of linearly independent equations, and hence the minimum[9] number of known keystream bits, is denoted as v with

$$v = \underbrace{\binom{18-k_1}{2} + \binom{21-k_2}{2} + \binom{22-k_3}{2}}_{\#\text{ linearized variables}} + \underbrace{(18-k_1)+(21-k_2)+(22-k_3)}_{\#\text{ original variables}},$$

where k_1, k_2 and k_3 are the number of bits we may additionally guess for R_1, R_2 and R_3 respectively. Fixing variables helps to decrease the size of the equation systems and the number of required keystream bits, but also increases the average amount of bits to guess for the whole attack to $2^{15+k_1+k_2+k_3}$.

We now use the principle we have outlined above (and the fact that encryption is applied to encoded data) for a ciphertext-only attack which explicitly targets the TCH3 channel in GMR-1. Encoding, scrambling and encrypting a 160-bit speech-frame $d^{(N)}$ with frame number N can be expressed as

$$c^{(N)} = \mathbf{G}^t d^{(N)} \oplus s \oplus z^{(N)},$$

where $\mathbf{G}^t$ is the transpose of the 160×208 generator matrix $\mathbf{G}$ of the code, s is a 208-bit pseudo-random scrambling sequence, $z^{(N)}$ the keystream generated for this frame and $c^{(N)}$ the resulting 208-bit codeword. $\mathbf{G}$ can be determined and is known (cf. Subsection 3.3.3), therefore a parity-check matrix $\mathbf{H}$ can be derived from $\mathbf{G}$ with $\mathbf{H}c = 0$ iff $c = \mathbf{G}^t d$. Due to this property, if we invert scrambling for a codeword $c^{(N)}$, we get

$$\mathbf{H}\left(c^{(N)} \oplus s\right) = \mathbf{H}\left(\mathbf{G}^t d^{(N)} \oplus z^{(N)}\right) = \mathbf{H}z^{(N)}.$$

Given a syndrome (i.e., a bit vector indicating whether decoding was completed without errors, potentially enabling error correction) vector $r^{(N)} = \mathbf{H}\left(c^{(N)} \oplus s\right)$, we can again set up an equation system in variables $x_0, x_1, ..., x_{v-1}$ of the initial state by guessing R_4, clocking the cipher 250 times (to account for the warm-up phase) and another 208 times, i.e.,

$$\mathbf{H}(\mathbf{A}x) = \mathbf{S}x = r^{(N)}.$$

Here, $\mathbf{A}$ is the $208 \times v$ matrix that describes the linear relation between x and the bits $z_0^{(N)}, z_1^{(N)} ..., z_{207}^{(N)}$ generated by the cipher. Please note that $\mathbf{H}$ is a 48×208 matrix and subsequently $\mathbf{S}$ is a $48 \times v$ matrix which implies that for $v > 48$ this system is not uniquely solvable. In order to obtain an equation system where $\mathbf{S}$ has full rank, we need to generate and collect equations from several encrypted frames for consecutive frame numbers $N, N+1, ...$ For a fixed session key, the initial states for different frame numbers are

---

[9] We need at least as many keystream bits as we have variables and thus equations. However, since not all equations we obtain by clocking the cipher based on R_4 are necessarily linearly independent, we may need even more keystream bits.

linearly related by the XOR-differences of the frame numbers. Taking these differences into account when generating equations allows to build a uniquely solvable equation systems and solving this equation system gives a potential initial state which could have generated $z^{(N)}$.

Now we describe the actual steps of our attack for which we assume that we are in possession of n 48-bit syndromes

$$r^{(N_0)} = \mathbf{H}\left(c^{(N_0)} \oplus s\right), ..., r^{(N_{n-1})} = \mathbf{H}\left(c^{(N_{n-1})} \oplus s\right)$$

which correspond to TCH3 downlink data encrypted under the same session key. Our attack is parameterized by n, k_1, k_2, k_3 and N_0 and recovers the initial state $\beta = g(K, N_0)$. Before we proceed, we need to introduce a helper method:

$\psi(\delta)$ Depending on the configuration of the attack, the result will be a string of 81 bits. It will contain only the bits of δ which are at positions in the state of the cipher, whose bits we have guessed (i.e., R_4 and parts of the other registers); all others will be 0.

The attack works by iterating over all possible values for the parts of the cipher we guess:

1. Systematically guess the bitstring γ which has $20 + k_1 + k_2 + k_3$ bits (also incorporating the fixed bit per LFSR). For each syndrome $0 \leq i < n$ do the following:
   a) Compute the 81-bit difference $\delta = g(0, N_0) \oplus g(0, N_i)$ in the initialization state for frame number N_0 and N_i.
   b) Modify γ by XORing it with the corresponding positions of δ, i.e., $\gamma' = \gamma \oplus \psi(\delta)$.
   c) Based on γ' and δ generate a linearized $458 \times v$ matrix $\mathbf{B}$ (and vector y for the one constant per equation) which describes the linear relation between the initial state for N_0 and the 458 keystream bits generated for $r^{(N_i)}$.
   d) Take the warm-up phase into account by discarding the first 250 rows of $\mathbf{B}$ to obtain a $208 \times v$ matrix $\mathbf{B}'$ and also discarding the first 250 elements of y to obtain y'.
   e) Compute the $48 \times v$ matrix $\mathbf{S}'$ and vector r' such that

   $$\begin{aligned} \mathbf{S}' &= \mathbf{H}\mathbf{B}' \qquad \text{and} \\ r' &= \mathbf{H}y' \oplus r^{(N_i)} = \mathbf{H}\left(y' \oplus c^{(N_i)}\right) \end{aligned}$$

   and add those rows of $\mathbf{S}'$ (and the corresponding bits from r') to the equation system $\mathbf{S}x = r$, which are linearly independent from all previously existing rows of $\mathbf{S}$.
   f) Abort if $\mathbf{S}$ has full rank.
2. Solve the equation system by computing $x = \mathbf{S}^{-1}r$ and combine the guessed bits and x appropriately to obtain the 81-bit initialization state candidate β.
3. Initialize A5-GMR-1 with β and clock it to obtain 208 bits of keystream $z'^{(N_0)}$ for frame number N_0 and test whether

   $$\mathbf{H}\left(c^{(N_0)} \oplus s \oplus z'^{(N_0)}\right) = 0.$$

   If this equation holds, applying the obtained keystream produces a valid codeword. This implies we have produced the correct keystream and therefore (most likely) the correct initial state.

Once we have $\beta = \mathscr{G}(K, N_0)$, we can set up another equation system

$$\mathbf{L}\alpha = \beta \oplus \epsilon \qquad \text{with} \qquad \alpha = \mathbf{L}^{-1}\left(\beta \oplus \epsilon\right) = \mathscr{F}(K, N_0)$$

where $\mathbf{L}$ describes the process of clocking α into all four LFSRs (and setting the lowest bit per LFSR to 1 which is expressed by ϵ). Solving the equation system resolves the initialization vector α from which we can easily derive the session key $K = \mathscr{F}(\alpha, N_0)$.

3.7 A Real-World Attack

In this section, we describe the details of our real-world attack on the TCH3 channel in the Thuraya network.

3.7.1 Recording TCH3 Data

Executing the attack requires acquiring and setting up appropriate hardware to generate and receive real-world data in the Thuraya network.

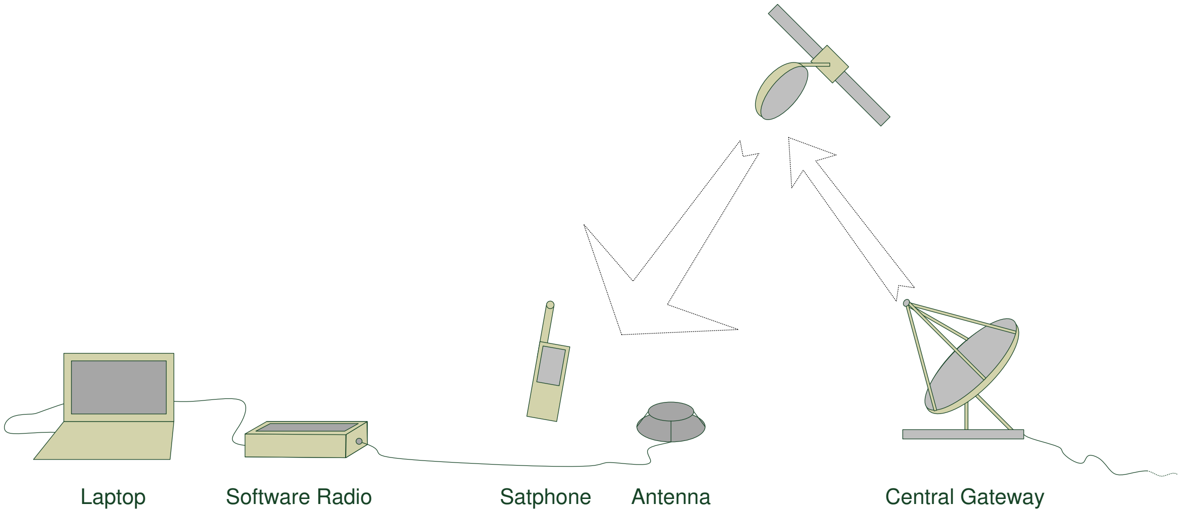


Figure 3.7: Schematic of the attack set-up

Figure 3.7 shows a schematic of our attack set-up: we use a satphone to establish a call in the Thuraya network and place an antenna nearby, thus receiving all downlink transmissions. Attached to the antenna is a Software Defined Radio (SDR) system. With the help of the SDR hardware and some software running on the laptop, we can demodulate and decode received transmissions. It is important to note here that we only receive the downlink and not the uplink. We focus on this part of the communication for two reasons:

1. Demodulation of downlink transmissions is (mostly) readily available as part of OsmocomGMR, while this is not true for the uplink.
2. The downlink can be received and demodulated (at least) in the entire area, which is assigned to one spotbeam (see below).

Furthermore, if we can decrypt the downlink, we can also decrypt the uplink—both share the same session key for encryption.

The software we use here is based on the OsmocomGMR[10] project which is maintained by Sylvain Munaut. The aim of the Osmocom project family is to establish open source implementations of a wide range of communication standards, e.g., GSM, TETRA and even GMR-1. OsmocomGMR itself uses the GNURadio project[11], which is an open source framework and provides signal processing functionalities. Although the implementation of OsmocomGMR is still in its infancy it is evolved enough for our purposes; we were able to use it with only a few tweaks—although not in a completely automated fashion.

---

[10] See `http://gmr.osmocom.org/trac/`.

[11] See `http://gnuradio.org/`.

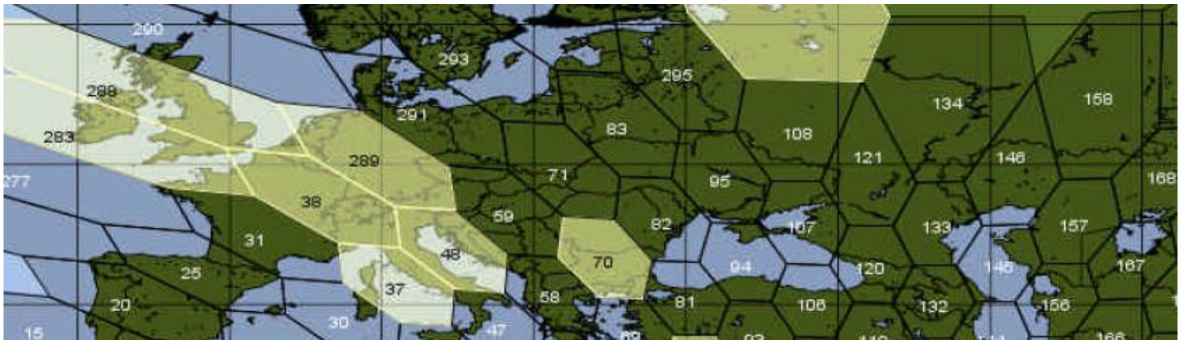


Figure 3.8: Thuraya spotbeams (with numeric IDs) over Europe

In addition to software, the OsmocomGMR project also provides information regarding the configuration of the Thuraya network. As shown in Figure 3.8[12], there are two spotbeams assigned to Germany, they have the IDs 289 and 291. Since our experiments were performed in Bochum, we picked the spotbeam with the former ID, to which (as stated on the same website) the ARFCN frequency identifier number 1007 is assigned. Given the fact that the downlink frequency band is divided into 1087 physical channels (starting at 1.525 GHz) with a spectral bandwidth of 31.25 KHz, ARFCN 1007 translates into a radio frequency of

$$f_{RF} = 1.525 \text{ GHz} + \frac{31.25}{2} \text{ KHz} + 1007 \cdot 31.25 \text{ KHz} = 1.556484375 \text{ GHz}$$

on the downlink.

To obtain real-world data for our attack, we used our Thuraya satphone and established some calls to a landline. By simultaneously tuning the SDR to f_{RF}, we were able to intercept TCH3 assignments that were sent to our phone via the CCH (cf. Subsection 3.3.2). After some experimentation we found that TCH3 is typically assigned to one of these three ARFCNs: 1008, 1009 and 1011—which is now also documented on the website. Upon observing the ARFCN assignment, we could tune to the newly assigned frequency and capture most of the encrypted downlink speech data (missing only a fraction at the beginning of the call). All subsequent data (including frame numbers) was stored on a harddisk, and could directly be used in our cryptanalysis.

3.7.2 Parity-check Matrix

As stated in the previous sections, a key step to move from a known-plaintext to a ciphertext-only scenario is collapsing all linear encoding steps into a single matrix **G** and deriving the respective parity-check matrix **H**. Obtaining **G** is straightforward: all relevant encoding steps for the TCH3 channel can be found in the respective document of the specification [ETS01b]. These steps include:

1. Block encoding
2. Convolutional encoding
3. Interleaving
4. (Scrambling)
5. Multiplexing

---

[12] See `http://gmr.osmocom.org/trac/wiki/Thuraya_Beams/`.

Each step—except for scrambling—can be modeled as multiplication of an information vector with an appropriately constructed matrix $\mathbf{M}_i$ with $0 \leq i \leq 3$. Given these matrices, their product is the 160×208 encoding matrix

$$\mathbf{G} = \prod_{i=0}^{3} \mathbf{M}_i .$$

The corresponding parity-check matrix $\mathbf{H}$ with

$$\mathbf{H}(\mathbf{G}^t d) = 0 \quad \text{for all} \quad d \in \{0,1\}^{160}$$

can be obtained with these steps:

1. Use Gaussian elimination to find a permutation matrix $\mathbf{P}$ with

$$\mathbf{LGP} = \mathbf{G}' = \left(\mathbf{I}_{160} | \mathbf{T}\right)$$

for some $\mathbf{L}$, where the left hand side of $\mathbf{G}'$ is the 160×160 identity matrix. Here, $\mathbf{G}'$ is the *systematic form* of the encoding matrix.

2. From the systematic form of $\mathbf{G}$, $\mathbf{H}'$ can be obtained easily, i.e.,

$$\mathbf{H}' = \left(\mathbf{T}^t | \mathbf{I}_{48}\right) .$$

The result is a 48×208 matrix, which is appropriate for code words encoded with $\mathbf{G}'$.

3. From $\mathbf{H}'$ the parity-check matrix $\mathbf{H}$ for the actual form of $\mathbf{G}$ can be obtained via another matrix multiplication, i.e.,

$$\mathbf{H} = \mathbf{H}'\mathbf{P}^{-1}.$$

The resulting matrix is given (rows encoded hexadecimally) in Figure 3.9.

3.7.3 Parameterization

As described in Section 3.6, our ciphertext-only attack on TCH3 is parameterized by the tuple (n, k_1, k_2, k_3). To actually execute the attack, we have experimentally established a working set of parameters.

First of all, we have determined how many streaks of TCH3 frames with consecutive frame numbers of a certain length we can expect to obtain with our eavesdropping setup. By *streak* we denote a set of received frames with numbers $N_0, N_1, N_2, \ldots, N_{n-1}$ with $N_i = N_{i-1} + 1$ for all $0 < i < n$, where n denotes the length of a streak. We have analyzed the TCH3 data of several 10 second calls and plotted the percentage of observed streak lengths in Figure 3.10. The longest streak we have observed consists of 56 TCH3 frames, which served as an upper bound for the next step.

In order to determine how many bits of the LFSRs R_1, R_2 and R_3 we need to guess[13] (in addition to completely guessing R_4), we have performed experiments: we have systematically evaluated all combinations of k_1, k_2 and k_3 with the aim to minimize $k_1 + k_2 + k_3$. We found that we need to guess at least 6 bits of R_1 to R_3, in order to always achieve full rank from 56 TCH3 frames. Of the 28 possibilities, only three were found to always guarantee full rank of the obtained matrices, see Table 3.2. Looking at the results, we picked $k_1 = 0, k_2 = 2, k_3 = 4$ because the average and maximum number of frames required to achieve full rank is lowest. Also, we can be certain that 33% of the frame number streaks (cf. Figure 3.10) are at least 24 frames long. This is helpful, because now we can pick multiple, different subsets of 24 TCH3 frames for our attack, which allows to validate any session key we may find.

---

[13] Please note that, for simplicity, we always guess the Least Significant Bits (LSBs) of each LFSR. It is certainly an interesting question which bits should be guessed for optimal performance.

```
20080202008020000280000a0202020a0808000000800000000000
04100140100010000040501444501400441044440400000000000
080200808020080000a00028080808282020000002000000000000
44150440004004050050000444544414004010001000000000000
80200800200802000220008880808088080800000080000000000
00040100504401000010505440001014545410400400000000000
40140400405000050000001040504000105010000000000000000
8028080080a0000a0000002080a0800020a0200000000000000000

a0280802002020020080a0a020800080808000000008000000000
44000540104010040040501404101440041044000040000000000
280a0200800808008020282808200020202000000002000000000
44140400104004040010140054545010045414000010000000000
88220080208002080200888880080008080800000000800000000
04050540500001040000145440445054545410400004000000000
00000800008000088020880008808088080000080000000000000
000002000020000a0200a000802020a08000008000000000000000

880a0a8280080808828 0a00088a8a8200008000000000080000000
00000000000000400004000504050004010105040000040000000
a0220a80a080020a00a02800a0a8a808008000000000020000000
04110001004001050050145404540004544054000000010000000
28280a82202020028220880028a8a880002000000000008000000
44050141404410000010141404001410545414400000004000000
04110441000400000010444000044444444444004000000000000
200a020020280002800000082028200008280800000000000000000

a802088000a828008280200088282820808800000000000800000
04100040500014010040401454405440140014400000000400000
a8200a0000a80a0800a00800a088880820a000000000000200000
40040441100405010050044444145004540044000000000100000
a808028000a82202022080000 28a0a0800828000000000000080000
00040501504411040010105400401410101454400000000040000
04110041004400044000004004440400404440000000000000000
00000080000800028080280020080828200000200000000000000

282a00828088000a800020008828a8200008000000000000008000
44040541100000001000010141450041014401400000000004000
882a0800a0a0000a80000800a088a808008000000000000002000
00100041104400040000045014501004505450000000000001000
a02a02022028000a8000800028a0a880002000000000000000800
40100001000000040000400404444440040404000000000000400
54000541500000000000000545400540054005400000000000000
04110001004000050040144014400014504440100000000000000

00080002808028000200202 0a080000080000000000000000000080
40100501000014000000400050005050105010400000000000040
00020000a0200a0000800808282000002000000000000000000020
40150400104005050040041454404000005010000000000000010
00200002200822000020808088080000080000000000000000008
00100001100411000010404044040000040000000000000000004
40140500404000000100501000405050505010400000000000000
08220082008800088000008008880800808880000000000000000
```

Figure 3.9: Encoded rows of the parity-check matrix of the TCH3 channel in Thuraya

k_1	k_2	k_3	**#Variables**	**#Frames (avg.)**	**#Frames (max.)**
0	2	4	532	12.42	24
0	3	3	532	12.73	25
1	2	3	533	13.01	25

Table 3.2: Guessed bits of the LFSRs R_1, R_2 and R_3

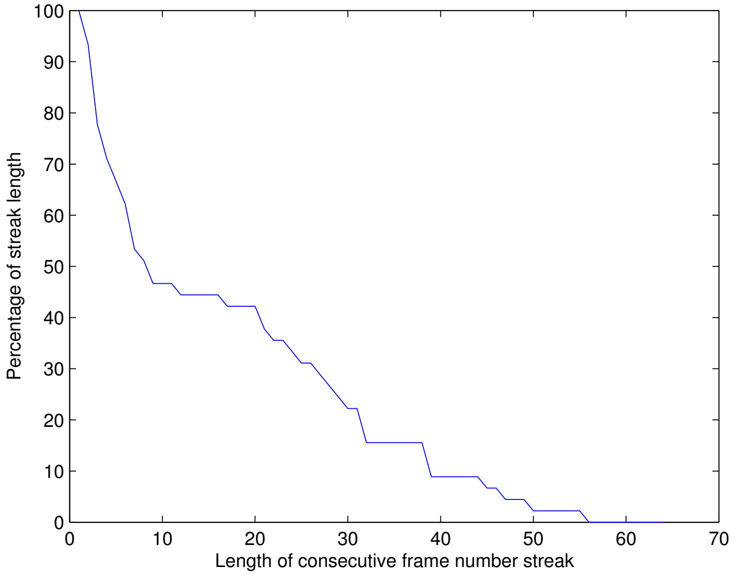


Figure 3.10: Proportion of frame number streak lengths

3.7.4 Implementation

With the parameters we have established in the previous section, we need to generate and solve on average 2^{21} equation systems with matrices of dimension 532×532 and consequentially test as many state candidates (by decoding via another matrix operation) to obtain one session key. To speed up the actual attack, we exploit the fact that the matrices describing the linear relation between internal state of the cipher and keystream depend only on the bits we guess. These matrices are simply multiplied with $\mathbf{H}$, which is fixed and static (cf. Subsection 3.3.3 and Subsection 3.7.2). Thus, once we have fixed a set of parameters we are going to use in the attack, we need to generate and store the resulting $\mathbf{S}$ matrices only once. This effectively splits the execution of the attack into a *pre-computation* and *recovery* phase:

- In the pre-computation phase, a parameter set is chosen and for each possible guess we generate and store the respective matrix. This step has to be done only once.
- In the recovery phase, the generated matrices are read from disk and subsequently used to build and solve equation systems for actual TCH3 channel data. This step has to be repeated for every new GMR-1 TCH3 session.

To further optimize the execution time of the recovery phase, we apply two more tricks:

1. In the pre-computation phase, we already test the matrices for linear dependencies and bring them in upper triangular form. Since we have to apply the resulting row operations also to the syndromes (i.e., the results of multiplying ciphertexts with $\mathbf{H}$) in the recovery phase, we have to track and store them, too. While this requires some more storage space, not having to do linearity testing in the recovery phase and knowing that the lower tridiagonal part of each matrix is zero is a considerable improvement in terms of computation time and storage space.
2. We use a freely available[14] and very fast implementation of the Lempel-Ziv (LZ) compression algo-

---

[14] See `http://www.quicklz.com/`.

rithm [ZL77] in order to minimize the required storage space and the performance impact of reading and writing matrices (from/to a standard harddisk). Although compression introduces some computational overhead in the pre-computation phase, decompression is fast enough to significantly speed-up our attack (when compared to an attack using uncompressed files).

Although our attack only uses TCH3 data, our implementation is also able to generate and handle mixtures of Fast Associated Control Channel-3 (FACCH3), TCH3 and keystream frames for an attack, which makes it considerably more complex. However, more details on the implementation are not relevant here, which is why the results of our attack will be presented next.

3.7.5 Results

In this section we shortly subsume the results of executing our proposed attack and the hardware we have used. Figure 3.11 shows the components we have used to perform the attack:

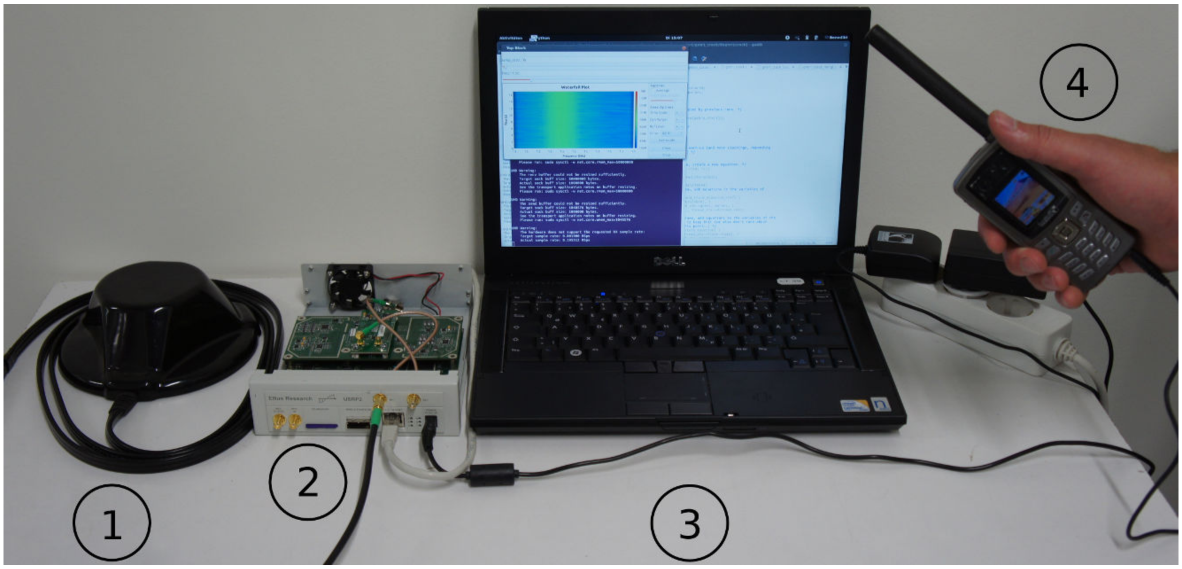


Figure 3.11: The attack setup: (1) antenna, (2) software radio, (3) laptop, (4) satphone

1. **Antenna**.
   An accessory[15] antenna (typically used for installing satphones in cars) was used to receive Thuraya traffic.
2. **USRP-2**.
   An Ettus USRP-2 device was used to digitize received EM transmissions (coming from the antenna) and send them to an attached laptop.
3. **Laptop**.
   A laptop is used to control the USRP-2, apply demodulation steps and execute the implemented attack.
4. **Satphone**.
   A Thuraya SO-2510 satphone was used as handheld device for communicating over the Thuraya network.

---

[15] Earlier attempts at building such an antenna failed; assembling helical antennas is a very delicate process and requires experience and very high precision.

In the pre-computation phase, we have generated approx. 400 GB of system matrices in a negligible amount of time. We have executed a 30 second call between satphone and landline, of which more[16] than 27 seconds of TCH3 data could be saved to disk. Given the eavesdropped data and pre-computed matrices, we were able to find the session key for multiple subsets of 24 frames in 32.1 minutes (on average).

It must be stressed here that—since the speech codecs of Thuraya still have not been reverse engineered—actually listening to a conversation is not possible for us. However, since the codecs can be reverse engineered too (potentially even by applying similar heuristics as used by us) this is no real obstacle.

3.8 Discussion and Future Work

Based on the reverse engineered details of the A5-GMR-1 stream cipher, we have have formulated an efficient and effective ciphertext-only attack. We have detailed the hardware and software components used to capture and decode downlink speech data which ultimately allowed us to execute the proposed attack with a non-optimized implementation. Given only a handful of encrypted TCH3 frames and some precomputed data, we were able to reveal the encryption key in *half an hour* of computation, demonstrating that our attack is not only feasible but quite practical.

Our attack and its implementation is specifically tailored for a real-world attack on Thuraya, but the implications most likely apply to all communication systems based on the GMR-1 standard. Given the fact that the downlink can be received at least[17] everywhere in the assigned spotbeam, we issue the explicit warning that solely *GMR-1 based systems should not be relied on* if strong privacy is required.

In the following sections we will discuss two aspects necessary to be solved (additionally to reverse engineering the audio codec) in order to extend our work to a fully working eavesdropper.

3.8.1 Uplink Interception

We currently intercept downlink communication in the L-Band, which gives us access to only one half of a call. In contrast to the downlink, which is broadcast to a vast area, the uplink is sent with a directional antenna utilizing only a minimum of power. The Thuraya SO-2510 has a small, helical antenna which not only radiates where the satphone is pointed at, but also to the sides—although with lower power. In conjunction with Prof. Kronberger from Cologne University of Applied Science, we have determined this level of power by establishing a call on the roof of the university while holding the phone strictly vertical and measuring the side radiation from a fixed distance. Using a signal analyzer and a horizontally polarized antenna with 5.85 dB gain, the uplink signal was detected at 1.65 GHz and determined to have a power of −33 dBm at a distance of 15 m. This implies that, assuming a free path loss of 110 dB and further propagation losses of 20–30 dB, it is entirely possible to directly receive the uplink signal at distances of 5 Km and more[18], given a direct line of sight.

We speculate that a more indirect approach might exploit the fact that Thuraya-2 and Thuraya-3 operate according to the “bent pipe” principle. In this setup, the satellite just acts as a redirector of incoming data, i.e., uplink data sent by a satphone is simply redirected to the ground segment (although shifted to a different frequency band). This implies that uplink data of mobile devices in the user segment can be intercepted by placing a satellite dish “closely” to the central gateway. However, implementing this method

---

[16] The first 3 seconds were lost due to the time it took for extracting the assigned ARFCN and manually re-tuning the SDR hardware accordingly.

[17] The Europe-based OsmocomGMR project was even able to receive and identify spotbeams assigned to South-Africa.

[18] Using a high gain antenna and/or a more sensitive receiver should significantly boost reception levels.

of interception is hindered by the lack of public specifications for the C-Band, which is used to transmit data between ground segment and satellite. Another difficulty stems from the fact that several concurrent communications are sent over this link to the gateway in parallel, therefore the L-Band downlink and C-Band uplink data need to be matched. To us, it seems nonetheless reasonable to assume that this can be done, when considering that Thuraya handles "only" 13 750 calls simultaneously, up- and downlink share frame numbers (and we can get frame numbers from the downlink) and timing of uplink data in the C-Band can probably be predicted quite accurately.

3.8.2 Real-time Decryption

Our approach represents a proof-of-concept implementation with a fairly complex (but flexible) program-library, written in plain C. Our only optimization is to use multi-threading, which distributes the search space across all cores of Intel's Xeon E5540. With eight concurrent threads, our attack requires 32.1 minutes (on average) to derive a session key. Real-time decryption is typically (e.g., in GSM) claimed when a session key can be found in less than one second. This can be achieved by distributing our attack across $30 \cdot 60$ Xeon processors, which might not be the best choice after all. Instead, it might be smarter to offload solving the equation system to Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs), since, as we have learned by profiling our implementation, this step is most demanding but also easy to parallelize as was shown in previous publications [GGHM05, BMPP06].

However, a month after the publication of a pre-print of this work, an email, claiming an implementation capable of revealing the A5-GMR-1 encryption key in real-time, was received by us. The author of this email claims to have implemented a variant of our attack, which targets the FACCH3[19] channel, requires fewer frames (but some plaintext) and allegedly uses only 8 GB of precomputed data. To the best of our knowledge, no description of this variant nor any source code have been published yet. Nevertheless, lacking the possibility to verify this specific implementation, it is quite reasonable to assume that incorporating more knowledge about the different channels and the structure of plaintexts in GMR-1 is very likely to lead to a much more efficient attack than what we have demonstrated so far.

---

[19] Data sent on the FACCH3 channel has much more redundancies added, which results in more linear equations per FACCH3 frame.

CHAPTER 4

SECURITY ANALYSIS OF THE GMR-2 STANDARD

This chapter documents our work on reverse engineering the cipher used in the GMR-2 satellite phone standard from a firmware update of an actual handheld device. Furthermore, we present a practical attack on the recovered scheme and show that it can be broken in negligible time with given plaintext.

4.1 Motivation

Since GMR-2 is the second of the two ETSI standards for satellite communications, it came into focus after analyzing GMR-1 and due to our general interest in the security of satphone telecommunication (cf. Section 3.1).

4.2 Related Work

In addition to the related work we have mentioned in the previous chapter (cf. Section 3.2), the analysis of DECT, which is another ETSI standard, comes to mind. DECT is a standard for connecting cordless telephones to a fixed telecommunications network over a short range. Cryptographic methods to ensure privacy of the wireless link were not publicly available until reverse engineered and cryptanalyzed by Stefan Lucks *et al.* in 2009 [LST+09]. While they do not describe the process of reverse engineering, they present a very detailed analysis of the multiple algorithms and protocols involved in DECT, which are all entirely proprietary designs.

4.3 Technical Background

In this section we shortly introduce the basic technical background which is necessary to understand our method of reverse engineering the stream cipher used in GMR-2 from Inmarsat's IsatPhone Pro.

4.3.1 Satphone Hardware

We now briefly introduce the general architectural structure of satellite phones and the hardware typically found in such devices. In general, the architecture of satellite phones is similar to the architecture of cellular phones, as documented[1] by Harald Welte. Both types of handsets have to perform a lot of real-tome processing of speech and signal data, thus they typically ship with a Digital Signal Processor (DSP), dedicated for this purpose. More relevant for our purpose are the facts that DSPs are also suitable for executing cryptographic algorithms, and that encryption is part of the encoding process (cf. Subsection 3.3.3), which makes the DSP's code a prime candidate for locating the stream cipher used in GMR-2.

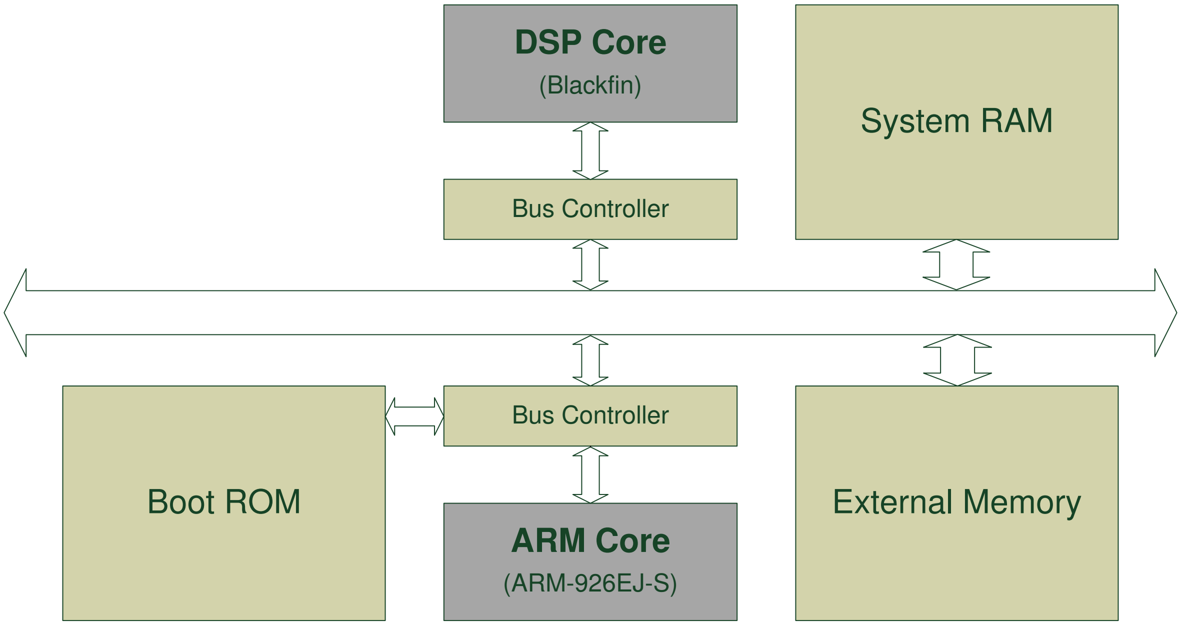


Figure 4.1: Essential building blocks of the "LeMans" AD6900 platform

Nevertheless, the core of a satphone is a standard microprocessor (usually an ARM-based Central Processing Unit (CPU)) that serves as the central control unit within the system. This CPU initializes the DSP during the boot process. Furthermore, both processors share parts of the main memory or other peripheral devices to implement inter-processor communication. To understand the flow of code and data on a phone, we thus needed to analyze the communication between the two processors.

All of the general characteristics of a satphone can be found when opening an Inmarsat IsatPhone Pro, which (probably) runs on an Analog Devices LeMans AD6900 platform[2]. The core of the platform is an ARM-926EJ-S CPU, which is supplemented by a Blackfin DSP (see Figure 4.1 for a schematic overview). We conjecture that this platform is used, since it is indicated by text fragments found in the phone's firmware. In the phone, both processors connect to the same bus interface, which is attached to the system RAM, any external memory that might be present as well as the shared peripherals (e.g., SIM card, keypad, SD/MMC slots, etc.).

4.3.2 The Blackfin DSP

Most of our reverse engineering work involved code running on the DSP used in the Inmarsat phone, which is why we will give some background information on this chip as well. It is not clear, which Blackfin variant

---

[1] See `http://laforge.gnumonks.org/papers/gsm_phone-anatomy-latest.pdf`

[2] See `http://www.eetimes.com/General/DisplayPrintViewContent?contentItemId=4016202`.

of Analog Devices' vast palette of DSPs is exactly used in the IsatPhone Pro, we will give some general information about the purpose and common architecture features of this class of processors.

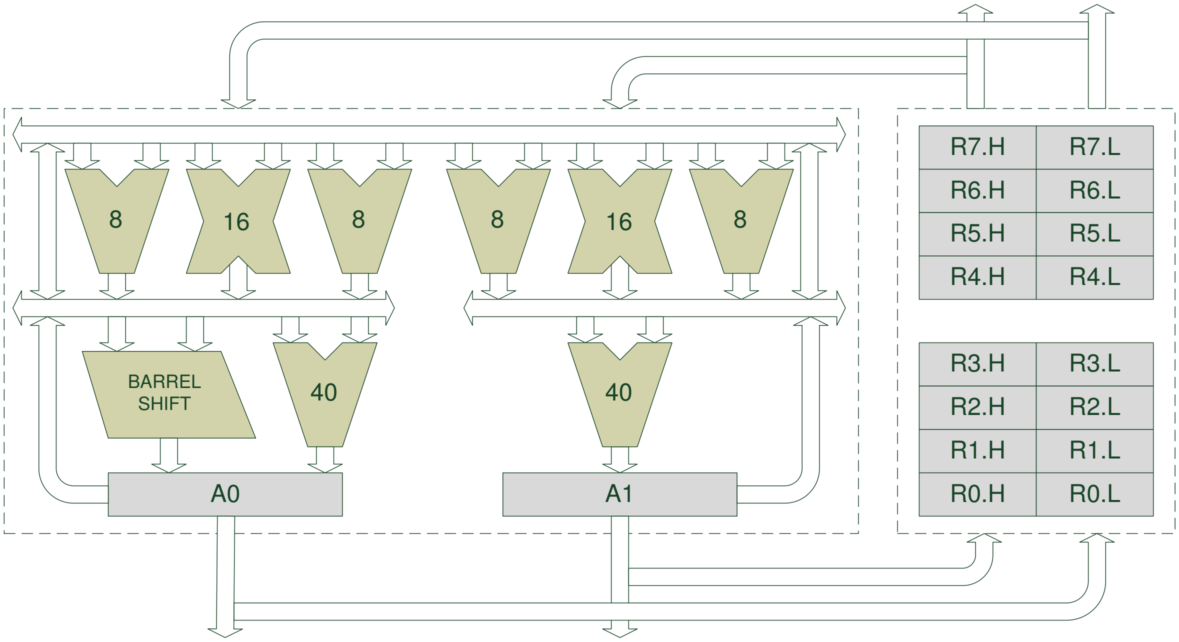


Figure 4.2: (Simplified) Data Arithmetic Unit of Blackfin DSPs

The Blackfin architecture represents a specialized 16/32-bit microprocessor, jointly developed by Intel and Analog Devices. The basic design is now over a decade old and was presented in 2000/2001. In general, a DSP is specifically designed for the operational needs of signal processing. Here, computing the convolution of two time-series is a typical task for many processing methods and these require many multiply-accumulate (MAC) operations. A MAC operation is defined as computing

$$a := a + (b \cdot c).$$

The Blackfin allows b and c to be of 16-bit size, while a can have up to 40 bits.

Figure 4.2 shows the "core" of a Blackfin DSP—its Data Arithmetic Unit (DAU). The two X-shaped, central elements are 16-bit multipliers, each connected to a 40-bit Arithmetic Logic Unit (ALU) which writes to the 40-bit accumulator registers A0 (and A1 respectively). The presence of 40-bit accumulators and an equally wide ALU represents one of the optimizations of this processor for MAC operations: although the output of a 16-bit multiplication only has a maximum of 32 bits, having a 40-bit ALU and accumulator allows multiple accumulation steps (i.e., additions of 32-bit values), before an overflow can occur. The accumulators can be fed back into the ALU, which enables the DSP to perform two 16-bit MAC operations simultaneously. Additionally, the Blackfin has four 8-bit ALUs, which are optimized for video encoding. The DSP has 16 16-bit data registers, of which two always form a pair. Additionally, the processor has registers specifically designed for the implementation of "zero-overhead loops", which means that the processor can execute loops without consuming additional cycles for the loop itself. Typically, a loop would require the CPU to test the value of the loop counter, perform a conditional branch to the top of the loop and decrement the loop counter. However, more complex loops, i.e., loops spanning many instructions instead of just a few, are still constructed in this manner—even on the Blackfin.

```
sub_20000000: x-refs 204fe02c
  20000000 LINK 0x18;
  20000004 [--SP] = (R7:6, P5:3);
  20000006 P5 = R1;
  20000008 P4 = R0;
  2000000a P0.L = 0x2034;     /* P0=0x00002034 */
  2000000e P0.H = 0x2046;     /* @P0(0x20462034)=hex:0x00000001 */
  20000012 R2.L = 0x7410;     /* R2=0x00007410 */
  20000016 R2.H = 0x2054;     /* @R2(0x20547410)=str:'pDecTemp0' */
  ...
  20000034 IF CC JUMP 0x2000015c;
  ...
loc_2000015c: x-refs 20000034
  2000015c SP += 0x24;
  2000015e (R7:6, P5:3) = [SP++];
  20000160 UNLINK;
  20000164 RTS;
...
sub_20000530: x-refs 200008bc
  ...
  20000538 P2.L = 0x5470;     /* P2=0x00005470 */
  2000053c P0 = 0x104 (X);    /* P0=0x00000104 */
  20000540 P2.H = 0x2000;     /* @P2(0x20005470)=hex:0x002f0000 */
  20000544 LSETUP(0x20000548, 0x20000550) LC0 = P0;
  20000548  R0 = W[P2++] (X);
  2000054a  P0 = R0;
  2000054c  R0.L = W[I0++];
  2000054e  P0 = P1 + (P0 << 0x1);
  20000550  W[P0 + 0x0] = R0;
  20000552 UNLINK;
  20000556 RTS;
```

Figure 4.3: Excerpt from the actual disassembly of the Blackfin's firmware

Figure 4.3 shows a (slightly redacted) excerpt from the output of our disassembler. As can be seen, the mnemonics of the Blackfin's assembly language are quite intuitive, even sporting simple conditional statements as indicated by the `IF` code word. Also, a zero-overhead loop, which is defined by the `LSETUP` command, can be found at address `0x20000544`.

4.3.3 Software and Operating System

The operating system running on the main CPU is a highly specialized system that is designed with respect to the special requirements of a phone system (e.g., limited resources, reliability, real-time constraints, etc.). We assume that what we found is a variant of the AMX 4-Thumb Kernel from Kadak Inc., which is also referred to as "Aos-AMX Version 0.29". Features of this Real Time Operating System (RTOS) include the following:

> - Compact, ROMable real time operating system
> - Rapid task context switching
> - Fast interrupt response
> - Nested interrupts with priority ordering
> - Preemptive, priority based task scheduler
> - Timing support for delays, timeouts, periodic events
> - Time slicing option with adjustable slices
> - Message passing with configurable message length
> - Dynamic task creation and dynamic task priorities
> - Protection against task priority inversion
>
> — `http://www.kadak.com/rtos/rtos.htm`

On the side of the Blackfin, we found strings referring to "Aos-BF Version 2.106.1", which is probably only a runtime framework and no complete operating system (in the sense of the AMX Kernel).

All of the software is deployed as *one* large, statically linked firmware binary which contains ARM code mixed with DSP code. Therefore, for our analysis, we were specifically interested in how the main CPU initializes the DSP, i.e., how the DSPs firmware is extracted from the firmware binary, at which memory address it is loaded into the DSP and how DSP and CPU communicate.

4.4 Reverse Engineering

Here, we describe the separate steps we followed to obtain a high-level description of the stream cipher A5-GMR-2. The general observation here is that, since any phone interacting with Inmarsat's network must be able to encrypt and decrypt, the respective implementation can be found in the firmware of a satphone. We describe the process of unpacking the firmware, locating the portion of code which is relevant and how we eventually found the algorithm with the help of a custom disassembler.

4.4.1 Obtaining the DSP Firmware

Obtaining the entire firmware of the satphone was easy, it could be downloaded as a firmware-update from Inmarsat's website. The firmware-updates of Inmarsat's satphones come as `.fpk` files in a proprietary format. To extract the firmware of the DSP, we had to follow these steps:

1. Understand the update package and locate the actual firmware.
2. Find the initialization routine in the ARM code.
3. Mimic the initialization process in order to obtain the DSP's code, as it is actually run by the DSP.

To understand the format, we have partially reverse engineered Inmarsat's firmware update program with the help of Interactive Disassembler (IDA), a general purpose disassembler which supports many binary formats and processor architectures. In the process of analyzing the update functionality, we discovered that the updater splits the original file into three files named `File1.bin`, `File2.bin` and `File3.bin` and writes these to the directory for temporary files. Examining strings in these three files revealed that `File2.bin` is the most interesting file, since it contained strings with `ApplyCipher` as substring. Furthermore, we were able to understand that the first two files have a header of 80 bytes each, which is immediately followed by ARM-code. Among data which we could not identify, the header includes this information:

- Address of the image, presumably in RAM.
- Size of the firmware image in bytes (i.e., file size without header).
- A string describing the current version of the firmware.

Particularly the first bit of information was very helpful, because in order to correctly resolve references to static addresses in the code, IDA requires the correct base-address. For the file we are interested in, we had to set the base-address to `0x0480000`, after which IDA produced a nice disassembly of the ARM firmware (which includes the DSP firmware at address `0x04a00000`). The third file seems to be used in a standardized firmware-update protocol called Device Firmware Upgrade (DFU), as indicated by its header (and more DFU related information at the end of the file).

As stated before (cf. Subsection 4.3.1), in the tandem of DSP and CPU, the ARM processor is responsible for starting the initialization process on the DSP. Analysis of the beginning of the DSP's firmware found that, after the DSP is powered up by the ARM, the DSP itself performs a "decompression" operation on its part of the firmware. A second effect is that this process also transfers the firmware from address `0x04a0000` (which we assume is FLASH memory) to address `0x20000000` (presumably RAM). In order to do this, the DSP reads its configuration from a table (starting at `0x04A00578`) and repeatedly does one of these three operations[3]:

- `copy_block(srcAddr, dstAddr, nBytes)`
  Here, `nBytes` bytes are simply copied from address `srcAddr` to `dstAddr`.
- `copy_block_repeatedly(srcAddr, dstAddr, nBytes, nTimes)`
  Here, `nBytes` are read from `srcAddr` and consecutively written for `nTimes`, beginning at `dstAddr`.
- `null_block(dstAddr, nBytes)`
  Here, `nBytes` null bytes are written to `dstAddr`.

After reverse engineering and understanding this initialization routine, we have used a script for IDA which evaluates the table and extracts order and parameters of the executed operations. A second program was then used to evaluate the output of the script, reproduce this part of the initialization procedure and consequently isolate and expand the DSP's code from the firmware. The result is a file, representing the DSP's firmware in the way it is actually found in RAM after decompression.

4.4.2 Developing a Blackfin Disassembler

As stated before, at the time of our analysis (early 2011), IDA did not support the disassembly of Blackfin code. However, what was available at that time was the GNU Toolchain[4] for Blackfin, now developed in cooperation between the open source community and Analog Devices. Part of this toolchain is `objdump`, a linear disassembler for Blackfin object files in the Executable and Linking Format (ELF) format. By *linear* we denote a disassembler that maps a sequence of bytes to their respective mnemonics, treating all bytes in the sequence as code and proceeding in a linear fashion from start to beginning. However, since the firmware does not contain only code, but also data, a non-negligible amount of the output of `objdump` is actually invalid (i.e., data interpreted as code) and thus worthless.

At this point, we saw two options: the first option was extending IDA with a custom processor module (in IDA's scripting language) for Blackfin, thus being able to leverage all of IDA's capabilities for navigating

---

[3] Please note that there are no actual functions with these names in the code; we break down the functionality into these three functions for the sake of clarity.

[4] See `http://blackfin.uclinux.org/gf/project/toolchain`.

in disassemblies. Alternatively, we could build our own disassembler around the already existing `objdump` from the GNU Toolchain. We decided to go with the second approach, which was motivated by three facts:

1. The Application Programming Interface (API) of IDA was (at the time, i.e., mid-2011) in a confusing state, which expressed itself in inconsistent naming conventions and a general lack of documentation.
2. The most error-prone process in the implementation is translating a sequence of bytes into their respective mnemonics. This code did already exist as part of `objdump`, but would—potentially—have to be re-written to be used with IDA.
3. The fact that the Blackfin firmware exhibited promising strings (i.e., `ApplyCipher`) led us to believe that finding the cipher from the function referencing this string would be easy, thus the disassembler need not be very sophisticated or complex.

The general design idea for our disassembler was using the `objdump` code to disassemble single bytes, while building additional logic around it in order to reach these goals (in comparison to the unmodified, linear disassembler):

1. Distinguish between data and code.
2. Resolve references to strings (and 32-bit values) and mark them visibly.
3. Properly annotate cross-references, i.e., resolve and mark targets and destinations of instructions that alter the control flow in the program (see below).

In order to distinguish between data and code, the idea is to follow the control flow of the program. There are two classes of instructions: there are instructions that do not alter the control flow and those that do (although sometimes some conditions need to be met). In the case of the Blackfin DSP, these instructions belong to the second class:

- `JUMP addr;`
  Unconditional, direct jump to code at another address.
- `JUMP (P0);`
  `JUMP (P1);`
  Unconditional, indirect jump to address stored in `P0` or `P1`.
- `IF CC JUMP addr;`
  `IF !CC JUMP addr;`
  Conditional jump to an address, iff the carry register `CC` is set (or not set).
- `CALL addr;`
  Direct call of a subroutine at a specific address; program flow will resume after this instruction after subroutine has finished.
- `CALL (P0);`
  `CALL (P1);`
  Indirect call of an address, which is stored in `P0` or `P1`. Control flow will resume at call site after returning from subroutine.
- `RET;`
  Return to the caller of the current subroutine.

By disassembling only the portions of the firmware that are reachable from valid code, we can make sure that—at least[5]—all parts of the firmware we reach are "real" code. The general strategy here is to perform an initial discovery of potential code by doing a linear disassembly of the entire firmware, thus identifying all `CALL` instructions, i.e., calls into subroutines. For each entry in the list of potential subroutines, we do

---

[5] On the other hand, there might still be code that is not directly reachable by `CALL` instructions, which would thus be treated as data.

a *speculative, recursive* disassembly, i.e., we follow the call, assume that it leads to a valid subroutine and disassemble it. If, for a top-level `CALL` we have followed, the disassembler comes across invalid opcodes, we know that the originating `CALL` site was invalid itself (i.e., we treated data as code, which is what leads to calling our strategy a "speculative" disassembly) and thus discard all code with that origin. The "recursive" part of our disassembler refers to the fact that we follow all `CALL` instructions in a recursive manner: assume that the disassembler implements the routine `analyze_code(addr)`, which starts disassembling at an address and subsequently follows the control flow. Assume further that we start disassembly at address `addr1` by calling the analysis routine via `analyze_code(addr1)`. When, in the course of the execution of this function, another call like `CALL addr2` is encountered, this function calls itself recursively via `analyze_code(addr2)`. When, in the course of analyzing the code at `addr2`, the disassembler does not encounter any of the instructions which alter the control flow, but only a `RET` at the end of the subroutine, `analyze_code(addr2)` also returns to its caller, which was `analyze_code(addr1)`. Here, analysis is continued at the next instruction. In summary, this is how we treat each of the control flow altering instructions:

- **Unconditional jumps**.
  Always and immediately continue disassembly at the new address.
- **Conditional jumps**.
  Follow the jump *once* until a `RET` is reached. Then continue disassembly after the conditional jump.
- **Calls.**
  Follow the call *once*, return to instruction after `CALL` when `RET` is encountered in subroutine.

Obviously, since execution can continue after a `CALL` or a conditional jump (i.e., if the condition is not met), both are treated similarly.

While following the control flow is suitable to distinguish between code and data, it introduces a problem: it becomes necessary to detect infinite loops, where the disassembler follows the same sequence of instructions over and over. Constructions like this are often the result of concurrent computing and can vary in complexity: in the process of developing the disassembler, we have found quasi-empty loops (i.e., a `JUMP` instruction jumping to its own address) but also very complex constructions with conditional jumps. In order to detect these loops, we have implemented a simple circular buffer, which stores the last N addresses we have disassembled. If we find a sequence of addresses twice in this buffer (both being adjacent to each other), we assume that we are in an infinite loop and stop disassembly of the respective subroutine.

```c
switch(value)
{
  case 0: ...
  case 1: ...
  case 2: ...
  case 3: ...
}
```

Figure 4.4: A `switch/case` statement in C code

Annotating references to data and code is straightforward, while resolving indirect jumps involves a bit more of processing. Typically, indirect jumps are used to implement `switch/case` statements, which are quite common across a variety of programming languages; Figure 4.4 shows a variant implemented in the C programming language. The compiler translates such a construct into an indirect jump, for which the target address is found in a table of 32-bit addresses that is indexed by `value`. Figure 4.5 shows some

```
 1      ...
 2      2049e77c P0 = R0;
 3      2049e77e CC = R0 < 0x0;
 4      2049e780 [FP -0x10] = R0;
 5      2049e782 IF CC JUMP 0x2049e9d8;
 6      2049e784 CC = R0 <= 0x3;
 7      2049e786 IF !CC JUMP 0x2049e9d8;
 8      2049e788 P1.L = 0x5a4;              /* P1=0x000005a4 */
 9      2049e78c P1.H = 0x2054;             /* @P1(0x205405a4)=hex:0x2049e796 */
10      2049e790 P0 = P1 + (P0 << 0x2);
11      2049e792 P0 = [P0 + 0x0];
12      2049e794 JUMP (P0);                 /* switch */
13    loc_2049e796: x-refs 2049e794;        /* switch case 0 */
14      2049e796 P0 = [FP -0x28];
15      ...
16      2054059c 3e ea 49 20 ec e9 49 20 96 e7 49 20 c4 e8 49 20  >.I ..I ..I ..I
17      205405ac 66 e8 49 20 1a e8 49 20 02 be 4d 20 fa bd 4d 20  f.I ..I ..M ..M
18      ...
```

Figure 4.5: A `switch/case` statement in Blackfin assembler

Blackfin code implementing a switch statement with four different cases, i.e., what is represented by the C code above. It can be observed that the address for the `JUMP` instruction is computed from `P1` and `P0` (line 10). Here, `P1` points (lines 8, 9) to the beginning of a table (lines 16, 17) while it is verified that $0 \leq \mathtt{P0} \leq 3$ (lines 3, 6). A simple and efficient heuristic to detect `switch/case` implementations is to look for a construction such as found in line 10 and 11 (might also occur with register `P1`). Starting from there and following the control flow backwards quickly discovers the remaining information, i.e., the start of the table and the range of the other register. Given this information, all jump targets can be resolved from the addresses stored in the table and are annotated as shown in line 13.

4.4.3 Finding the Cipher

Our custom disassembler is able to resolve cross-references, which eases understanding of the obtained disassembly significantly. Still, applying the disassembler on the reconstructed DSP firmware yields more than 300,000 lines of assembler code and we decided that purely manual analysis is too inefficient. Hence, we again applied the same heuristics we successfully used to find the cipher in GMR-1, i.e., we searched for subroutines holding a significant percentage of mathematical operations one would expect from an encryption algorithm [DHW+12]. Unfortunately, this approach did not reveal any code regions that could be attributed to LFSR-based keystream generation.

Hence, we decided to follow another approach. The Blackfin code contains a number of `ASSERT()` statements which include the name of the source files of the respective code. This allowed us to directly infer preliminary function names and to derive the purpose of several functions. More specifically, we identified one subroutine that referenced a source file with the name `..\..\modem\internal\Gmr2p_modem_ApplyCipher.c` and named it `ApplyCipher()`. We found that the function does what the name implies, it takes two 120-bit inputs and simply XORs them. We assumed that one of these parameters is the output of a stream cipher because the length matches the expected frame size of 120 bits according to the GMR-2 specification [ETS01b]. Starting from this subroutine, we identified the code for generating the keystream by applying a number of different techniques that we explain in the following. All of these techniques aim at narrowing down the potentially relevant code base in the disassembly. This was an essential and inevitable step in the analysis process since the stream cipher code is located in an entirely different part within the DSP code than `ApplyCipher()`.

First, we created the *reverse call graph* of the `ApplyCipher()` function, i.e., we recursively identified all call sites of this subroutine. Each call site is represented as a node in the graph and an edge from one node to another node indicates that the destination node's subroutine is called from the source node's subroutine. This process is repeated until there is no caller left. It turned out that this graph had ten topmost functions, which are not called directly because they were thread functions. We started by manually tracking the data flow of the keystream parameter, starting at `ApplyCipher()`, following the reverse call graph. Unfortunately, this did not turn out to be promising since a myriad of additional functions are being called in between the topmost functions in the graph and `ApplyCipher()`. However, in each of the topmost functions, we were able to identify a subroutine (denoted by us as `CreateChannelInstance()`) that allocates the memory region of the keystream buffer before initializing it with zeros. What was missing was the piece of code that fills the buffer.

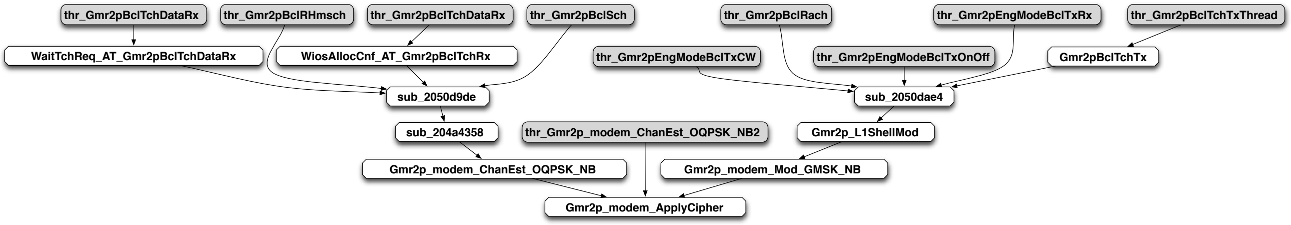


Figure 4.6: Reverse call graph of `ApplyCipher()` (the ten gray nodes are root nodes)

An analysis of the control flow graphs of the ten topmost routines in the reverse callgraph suggests that each routine implements a state machine using one `switch/case` statement. We generated the *forward call graph* for each case in the `switch` statement, where, in analogy to the reverse call graph, an edge from one node to another indicates that the source node's subroutine calls the destination node's subroutine. Given this new graph, we were able to derive which functions are called in each corresponding state. Most notably, this allows us to identify the points at which the keystream buffer is created (by calling `CreateChannelInstance()`) and the encryption of the plain text happens (by calling `ApplyCipher()`). The code responsible for generating the keystream naturally has to be called in between these two points.

The remaining code (approximately 140 subroutines) was still too large for a manual analysis. In order to further narrow down the relevant code parts, we created the forward call graphs of all ten thread routines and computed the intersection of all the nodes in the graphs. The idea behind this approach is that in every case the stream cipher has to be called eventually, regardless of the actual purpose of the thread. The intersection greatly reduces the candidate set of code regions from about 140 subroutines to only 13 functions shared by all threads (not including further nested subroutine calls). In the last step, we analyzed these remaining functions manually. At first, this analysis revealed the subroutine, which encodes the TDMA-frame counters into a 22-bit frame number. Shortly after this function, the actual cipher code is called. The algorithm itself, as explained in the next section, is completely dissimilar to A5/2, which also explains why we were not able to spot the cipher with the same methods as in the analysis of GMR-1.

4.5 The A5-GMR-2 Stream Cipher

After having obtained the cipher's assembler code, we had to find a more abstract description in order to enhance intuitive understanding of its way of functioning. We arbitrarily chose to split the cipher into several distinct components which emerged after examining its functionality. Note that, for the sake of symmetry, we denote the cipher as A5-GMR-2, although it shows no resemblance to any of the A5-type ciphers and is called GMR-2-A5 in the respective specification [ETS01a].

4.5.1 Structure

The cipher uses a 64-bit encryption-key and operates on bytes. When the cipher is clocked, it generates one byte of keystream, which we denote by Z_l, where l represents the number of clockings. The cipher exhibits an eight byte state register $S = (S_0, S_1, ..., S_7)_{2^8}$ and three major components we call $\mathscr{F}$, $\mathscr{G}$ and $\mathscr{H}$. Additionally, there is a 1-bit register T that outputs the so-called "toggle-bit" and a 3-bit register C that implements a counter. Figure 4.7 provides a schematic overview of the cipher structure. In the following, we detail the inner workings of each of the three major components.

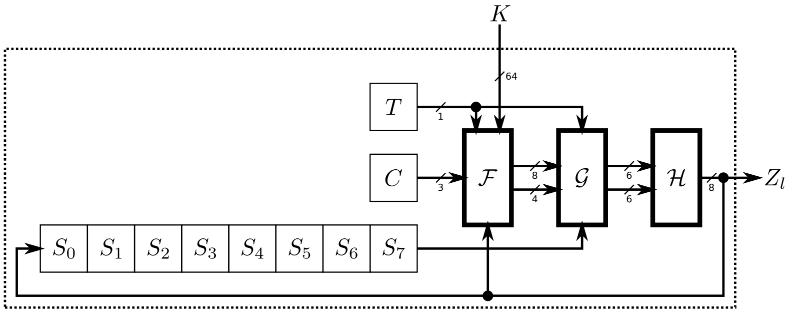
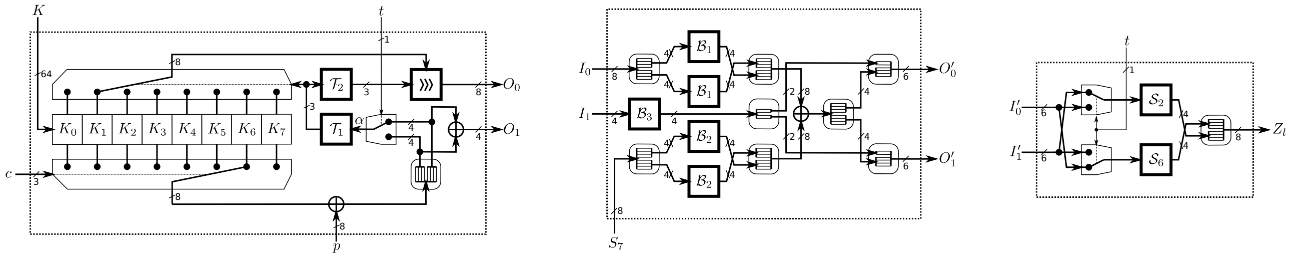


Figure 4.7: The A5-GMR-2 cipher

We begin with the $\mathscr{F}$-component, which is certainly the most interesting part of this cipher—Figure 4.8a shows its internal structure. On the left we see another 64-bit register split into eight bytes $(K_0, K_1, ..., K_7)_{2^8}$. The register is read from two sides, on the lower side one byte is extracted according to the value of c, i.e., the output of the lower multiplexer is K_c. The upper multiplexer outputs another byte, but this one is



(a) $\mathscr{F}$-component

(b) $\mathscr{G}$-component

(c) $\mathscr{H}$-component

Figure 4.8: Components of the A5-GMR-1 cipher

determined by a 4-bit value, which we will call α. On the right side, two smaller sub-components

$$\mathcal{T}_1 : \{0,1\}^4 \mapsto \{0,1\}^3$$
$$\mathcal{T}_2 : \{0,1\}^3 \mapsto \{0,1\}^3$$

are implemented via table-lookups (see Table 4.1). Also, a 4-bit and an 8-bit XOR are used. The input of $\mathcal{T}_1$ is determined by p, K_c and the toggle-bit t. Note that we use $p = Z_{l-1}$ as a shorthand to denote one byte of keystream that was already generated. We model the behavior of the small vertical multiplexer by $\mathcal{N}(\cdot)$, which we define as

x	$\mathcal{T}_1(x)$	$\mathcal{T}_2(x)$	$\mathcal{T}_2(\mathcal{T}_1(x))$
$(0,0,0,0)_2$	2	4	6
$(0,0,0,1)_2$	5	5	3
$(0,0,1,0)_2$	0	6	4 *
$(0,0,1,1)_2$	6	7	2
$(0,1,0,0)_2$	3	4	7
$(0,1,0,1)_2$	7	3	1
$(0,1,1,0)_2$	4	2	4 *
$(0,1,1,1)_2$	1	1	5
$(1,0,0,0)_2$	3	-	7
$(1,0,0,1)_2$	0	-	4 *
$(1,0,1,0)_2$	6	-	2
$(1,0,1,1)_2$	1	-	5
$(1,1,0,0)_2$	5	-	3
$(1,1,0,1)_2$	7	-	1
$(1,1,1,0)_2$	4	-	4 *
$(1,1,1,1)_2$	2	-	6

Table 4.1: $\mathcal{T}_1$ and $\mathcal{T}_2$ as lookup-table

$$\mathcal{N} : \{0,1\} \times \{0,1\}^8 \mapsto \{0,1\}^4$$
$$(t, x_{\langle 7\rangle}, x_{\langle 6\rangle}, ..., x_{\langle 0\rangle})_2 \mapsto \begin{cases} (x_{\langle 3\rangle}, x_{\langle 2\rangle}, x_{\langle 1\rangle}, x_{\langle 0\rangle})_2 & \text{if } t = 0, \\ (x_{\langle 7\rangle}, x_{\langle 6\rangle}, x_{\langle 5\rangle}, x_{\langle 4\rangle})_2 & \text{if } t = 1. \end{cases}$$

With the help of $\mathcal{N}$, which returns either the higher or lower nibble of its second input, the following holds for the output of the mentioned multiplexer

$$\alpha = \mathcal{N}(t, K_c \oplus p) = \mathcal{N}(c \bmod 2, K_c \oplus p).$$

The output of the upper multiplexer is rotated to the right by as many positions as indicated by the output of $\mathcal{T}_2$, therefore the 8-bit output O_0 and the 4-bit value O_1 are of the following form,

$$O_0 = \left(K_{\mathcal{T}_1(\alpha)} \ggg \mathcal{T}_2\left(\mathcal{T}_1(\alpha)\right)\right)_{2^8} \quad \text{and}$$
$$O_1 = \left(K_{c_{\langle 7\rangle}} \oplus p_{\langle 7\rangle} \oplus K_{c_{\langle 3\rangle}} \oplus p_{\langle 3\rangle}, K_{c_{\langle 6\rangle}} \oplus p_{\langle 6\rangle} \oplus K_{c_{\langle 2\rangle}} \oplus p_{\langle 2\rangle}, K_{c_{\langle 5\rangle}} \oplus p_{\langle 5\rangle} \oplus K_{c_{\langle 1\rangle}} \oplus p_{\langle 1\rangle}, K_{c_{\langle 4\rangle}} \oplus p_{\langle 4\rangle} \oplus K_{c_{\langle 0\rangle}} \oplus p_{\langle 0\rangle}\right)_2.$$

The $\mathcal{G}$-component gets the outputs of the $\mathcal{F}$-component as inputs, i.e., $I_0 = O_0, I_1 = O_1$. Additionally, the one byte S_7 of the state is used as input. As can be seen in Figure 4.8b, three sub-components, denoted as $\mathscr{B}_1, \mathscr{B}_2, \mathscr{B}_3$, are employed—again, they are implemented in the form of lookup-tables. Each of these components works on 4-bit inputs and equally returns 4-bit. After analyzing the tables, we found that all three simply implement linear boolean arithmetic, i.e.,

$$\begin{aligned}
\mathscr{B}_1 : \{0,1\}^4 &\mapsto \{0,1\}^4 \\
\left(x_{\langle 3\rangle}, x_{\langle 2\rangle}, x_{\langle 1\rangle}, x_{\langle 0\rangle}\right)_2 &\mapsto \left(x_{\langle 3\rangle} \oplus x_{\langle 0\rangle}, x_{\langle 3\rangle} \oplus x_{\langle 2\rangle} \oplus x_{\langle 0\rangle}, x_{\langle 3\rangle}, x_{\langle 1\rangle}\right)_2, \\
\mathscr{B}_2 : \{0,1\}^4 &\mapsto \{0,1\}^4 \\
\left(x_{\langle 3\rangle}, x_{\langle 2\rangle}, x_{\langle 1\rangle}, x_{\langle 0\rangle}\right)_2 &\mapsto \left(x_{\langle 1\rangle}, x_{\langle 3\rangle}, x_{\langle 0\rangle}, x_{\langle 2\rangle}\right)_2, \\
\mathscr{B}_3 : \{0,1\}^4 &\mapsto \{0,1\}^4 \\
\left(x_{\langle 3\rangle}, x_{\langle 2\rangle}, x_{\langle 1\rangle}, x_{\langle 0\rangle}\right)_2 &\mapsto \left(x_{\langle 2\rangle}, x_{\langle 0\rangle}, x_{\langle 3\rangle} \oplus x_{\langle 1\rangle} \oplus x_{\langle 0\rangle}, x_{\langle 3\rangle} \oplus x_{\langle 0\rangle}\right)_2.
\end{aligned}$$

Since these sub-components and the modulo-2 addition are linear and all other operations on single bits just amount to permutations, the $\mathscr{G}$-component is entirely linear. Therefore, we can write the 6-bit outputs O'_0, O'_1 as linear functions of the inputs I_0, I_1 and S_7, i.e.,

$$O'_0 = \left(I_{0_{\langle 7\rangle}} \oplus I_{0_{\langle 4\rangle}} \oplus S_{7_{\langle 5\rangle}}, I_{0_{\langle 7\rangle}} \oplus I_{0_{\langle 6\rangle}} \oplus I_{0_{\langle 4\rangle}} \oplus S_{7_{\langle 7\rangle}}, I_{0_{\langle 7\rangle}} \oplus S_{7_{\langle 4\rangle}}, I_{0_{\langle 5\rangle}} \oplus S_{7_{\langle 6\rangle}}, I_{1_{\langle 3\rangle}} \oplus I_{1_{\langle 1\rangle}} \oplus I_{1_{\langle 0\rangle}}, I_{1_{\langle 3\rangle}} \oplus I_{1_{\langle 0\rangle}}\right)_2 \quad \text{and}$$

$$O'_1 = \left(I_{0_{\langle 3\rangle}} \oplus I_{0_{\langle 0\rangle}} \oplus S_{7_{\langle 1\rangle}}, I_{0_{\langle 3\rangle}} \oplus I_{0_{\langle 2\rangle}} \oplus I_{0_{\langle 0\rangle}} \oplus S_{7_{\langle 3\rangle}}, I_{0_{\langle 3\rangle}} \oplus S_{7_{\langle 0\rangle}}, I_{0_{\langle 1\rangle}} \oplus S_{7_{\langle 2\rangle}}, I_{1_{\langle 2\rangle}}, I_{1_{\langle 0\rangle}}\right)_2.$$

Finally, the $\mathscr{H}$-component gets $I'_0 = O'_0$ and $I'_1 = O'_1$ as input and constitutes the non-linear “filter” of the cipher (see Figure 4.8c). Here, two new sub-components

$$\begin{aligned}
\mathscr{S}_2 &: \{0,1\}^6 \mapsto \{0,1\}^4 \\
\mathscr{S}_6 &: \{0,1\}^6 \mapsto \{0,1\}^4
\end{aligned}$$

are used and implemented via lookup-tables. Interestingly, these tables were taken from the DES, i.e., $\mathscr{S}_2$ is the second S-box and $\mathscr{S}_6$ represents the sixth S-box of DES. However, in this cipher, the S-boxes have been reordered to account for the different addressing, i.e., the four most-significant bits of the inputs to $\mathscr{S}_2$ and $\mathscr{S}_6$ select the S-box-column, the two least-significant bits select the row. Note that this is crucial for the security of the cipher. The inputs to the S-boxes are swapped with the help of two multiplexers, depending on the value of t. Given the inputs I'_0, I'_1 and t we can express the l-th byte of keystream as

$$Z_l = \begin{cases} (\mathscr{S}_2(I'_1), \mathscr{S}_6(I'_0))_{2^4} & \text{if } t = 0, \\ (\mathscr{S}_2(I'_0), \mathscr{S}_6(I'_1))_{2^4} & \text{if } t = 1. \end{cases}$$

4.5.2 Mode of Operation

Next we describe the mode of operation. When the cipher is clocked for the l-th time, the following happens:

1. Based on the current state of the S-, C- and T-register, the cipher generates one byte Z_l of keystream.
2. The T-register is toggled, i.e., if it was 1 previously, it is set to 0 and vice versa.
3. The C-register is incremented by one, when 8 is reached the register is reset to 0.
4. The S-register is shifted by 8 bits to the right, i.e., $S_7 := S_6, S_6 := S_5$ etc. The previous value of S_7 is fed into the $\mathscr{G}$-component, the subsequent output Z_l of $\mathscr{H}$ is written back to S_0, i.e., $S_0 := Z_l$. This value is also passed to the $\mathscr{F}$-component as input for the next iteration.

The cipher is operated in two modes, *initialization* and *generation*. In the initialization phase, the following steps are performed:

1. The T- and C-register are set to 0.
2. The 64-bit encryption-key is written into the K-register in the $\mathscr{F}$-component.
3. The state-register S is initialized with the 22-bit frame number N, this procedure is dependent on the "direction bit" but not detailed here as it is irrelevant for the remainder.

After C, T and S have been initialized, the cipher is clocked eight times, but the resulting keystream is discarded.

After initialization is done, the cipher is clocked to generate and output actual keystream bytes. By $Z_l^{(N)}$ we denote the l-th $(0 \leq l \leq 14)$ byte of keystream generated after initialization (and warm-up) with frame number N. In GMR-2, the frame number is always incremented after 15 bytes of keystream, which forces a re-initialization of the cipher. Therefore, the keystream that is actually used is the concatenation of these 15-byte blocks. We denote continuous keystream for frames $N, N+1, \ldots$ by Z, with

$$Z = \left(Z_0^{(N)}, \ldots, Z_{14}^{(N)}, Z_0^{(N+1)}, \ldots, Z_{14}^{(N+1)}, Z_0^{(N+2)}, \ldots\right)_{2^8}.$$

4.6 Cryptanalysis

In this section, we present a known-plaintext attack that is based on several observations that can be made when carefully examining the $\mathscr{F}$-component (and the starred rows in Table 4.1):

1. If $\alpha \in \{(0,0,1,0)_2, (1,0,0,1)_2\}$ then $\mathscr{T}_1(\alpha) = 0$ and $\mathscr{T}_2(\mathscr{T}_1(\alpha)) = 4$, thus $O_0 = (\mathscr{N}(0, K_0), \mathscr{N}(1, K_0))_{2^4}$.
2. If $\alpha \in \{(0,1,1,0)_2, (1,1,1,0)_2\}$ then $\mathscr{T}_1(\alpha) = 4$ and $\mathscr{T}_2(\mathscr{T}_1(\alpha)) = 4$, thus $O_0 = (\mathscr{N}(0, K_4), \mathscr{N}(1, K_4))_{2^4}$.
3. If $\mathscr{T}_1(\alpha) = c$, both multiplexers select the same key byte. We call this a *read-collision* in K_c.

In the following, we describe how to obtain K_0 and K_4 with high probability, which is then leveraged in a second step in order to guess the remaining 48 bits of K in an efficient way.

The key idea to derive K_0 is to examine keystream bytes $(Z_i, Z_{i-1}, Z_{i-8})_{2^8}$ with $i \in \{8, 23, 38, \ldots\}$ in order to detect when a read-collision in K_0 has happened during the generation of Z_i. Please note that due to our choice of i this

$$Z_8 = Z_8^{(N)}, Z_{23} = Z_8^{(N+1)}, Z_{38} = Z_8^{(N+2)}, \ldots$$

holds, i.e., for each i we already know that the lower multiplexer has selected K_0. In general, if the desired read-collision has happened in the $\mathscr{F}$-component, the outputs of the $\mathscr{F}$-component are

$$O_0 = \left(p_{\langle 3\rangle} \oplus \alpha_{\langle 3\rangle}, p_{\langle 2\rangle} \oplus \alpha_{\langle 2\rangle}, p_{\langle 1\rangle} \oplus \alpha_{\langle 1\rangle}, p_{\langle 0\rangle} \oplus \alpha_{\langle 0\rangle}, K_{0_{\langle 7\rangle}}, K_{0_{\langle 6\rangle}}, K_{0_{\langle 5\rangle}}, K_{0_{\langle 4\rangle}}\right)_2 \quad \text{and}$$
$$O_1 = \left(K_{0_{\langle 7\rangle}} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle}, K_{0_{\langle 6\rangle}} \oplus p_{\langle 6\rangle} \oplus \alpha_{\langle 2\rangle}, K_{0_{\langle 5\rangle}} \oplus p_{\langle 5\rangle} \oplus \alpha_{\langle 1\rangle}, K_{0_{\langle 4\rangle}} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle}\right)_2,$$

and the subsequent outputs of $\mathscr{G}$ are

$$\begin{aligned}O_0' = (&p_{\langle 3\rangle} \oplus \alpha_{\langle 3\rangle} \oplus p_{\langle 0\rangle} \oplus \alpha_{\langle 0\rangle} \oplus S_{7_{\langle 5\rangle}}, p_{\langle 3\rangle} \oplus \alpha_{\langle 3\rangle} \oplus p_{\langle 2\rangle} \oplus \alpha_{\langle 2\rangle} \oplus p_{\langle 0\rangle} \oplus \alpha_{\langle 0\rangle} \oplus S_{7_{\langle 7\rangle}}, p_{\langle 3\rangle} \oplus \alpha_{\langle 3\rangle} \oplus S_{7_{\langle 4\rangle}}, p_{\langle 1\rangle} \oplus \alpha_{\langle 1\rangle} \oplus S_{7_{\langle 6\rangle}},\\ &K_{0_{\langle 7\rangle}} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle} \oplus K_{0_{\langle 5\rangle}} \oplus p_{\langle 5\rangle} \oplus \alpha_{\langle 1\rangle} \oplus K_{0_{\langle 4\rangle}} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle}, K_{0_{\langle 7\rangle}} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle} \oplus K_{0_{\langle 4\rangle}} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle})_2 \quad \text{and}\\ O_1' = (&K_{0_{\langle 7\rangle}} \oplus K_{0_{\langle 4\rangle}} \oplus S_{7_{\langle 1\rangle}}, K_{0_{\langle 7\rangle}} \oplus K_{0_{\langle 6\rangle}} \oplus K_{0_{\langle 4\rangle}} \oplus S_{7_{\langle 3\rangle}}, K_{0_{\langle 7\rangle}} \oplus S_{7_{\langle 0\rangle}}, K_{0_{\langle 5\rangle}} \oplus S_{7_{\langle 2\rangle}}, K_{0_{\langle 6\rangle}} \oplus p_{\langle 6\rangle} \oplus \alpha_{\langle 2\rangle}, K_{0_{\langle 4\rangle}} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle})_2.\end{aligned}$$

Considering the $\mathscr{H}$-component, we also know that

$$Z_i = \left(\mathscr{S}_2\left(O_1'\right), \mathscr{S}_6\left(O_0'\right)\right)_{2^4}$$

holds.

In order to determine K_0, we examine the inputs and outputs of $\mathscr{S}_6$ and $\mathscr{S}_2$ in the $\mathscr{H}$-component, starting with $\mathscr{S}_6$. Due to the reordering of the DES S-boxes, the column of $\mathscr{S}_6$ is selected by the four most-significant bits of O'_0. If we assume a collision in K_0 has happened while generating Z_i, we can compute these most-significant bits due to the fact that $S_7 = Z_{i-8}$ and $p = Z_{i-1}$ are also known for all of our choices of i. If, for $\alpha \in \{(0,0,1,0)_2, (1,0,0,1)_2\}$ the lower nibble of Z_i is found in the row with index β, a collision may indeed have happened and the lower two bits of O'_0 must be $(\beta_{\langle 1\rangle}, \beta_{\langle 0\rangle})_2$, which implies

$$\begin{aligned} K_{0_{\langle 7\rangle}} \oplus K_{0_{\langle 5\rangle}} \oplus K_{0_{\langle 4\rangle}} &= \beta_{\langle 1\rangle} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle} \oplus p_{\langle 5\rangle} \oplus \alpha_{\langle 1\rangle} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle}, \\ K_{0_{\langle 7\rangle}} \oplus K_{0_{\langle 4\rangle}} &= \beta_{\langle 0\rangle} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle}. \end{aligned}$$

Here we gain information about the bits of K_0; $K_{0_{\langle 5\rangle}}$ can even be computed. We can then use the output of $\mathscr{S}_2$ to verify whether a collision has happened for the particular α we used above. Due to the structure of the S-box, there are only four 6-bit inputs γ with

$$\mathscr{S}_2(\gamma) = \left(Z_{i_{\langle 7\rangle}}, Z_{i_{\langle 6\rangle}}, Z_{i_{\langle 5\rangle}}, Z_{i_{\langle 4\rangle}}\right)_2.$$

Due to our partial knowledge about $\left(K_{0_{\langle 4\rangle}}, K_{0_{\langle 5\rangle}}, K_{0_{\langle 7\rangle}}\right)_2$ we can test for each γ whether the following relations hold:

$$\begin{aligned} \gamma_{\langle 5\rangle} &\overset{?}{=} \beta_{\langle 0\rangle} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle} \oplus S_{7_{\langle 1\rangle}}, \\ \gamma_{\langle 4\rangle} \oplus \gamma_{\langle 1\rangle} &\overset{?}{=} \beta_{\langle 0\rangle} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle} \oplus S_{7_{\langle 3\rangle}} \oplus p_{\langle 6\rangle} \oplus \alpha_{\langle 2\rangle}, \\ \gamma_{\langle 3\rangle} \oplus \gamma_{\langle 0\rangle} &\overset{?}{=} \beta_{\langle 0\rangle} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle} \oplus S_{7_{\langle 0\rangle}}, \\ \gamma_{\langle 2\rangle} \oplus \gamma_{\langle 5\rangle} &\overset{?}{=} \beta_{\langle 1\rangle} \oplus p_{\langle 7\rangle} \oplus \alpha_{\langle 3\rangle} \oplus p_{\langle 5\rangle} \oplus \alpha_{\langle 1\rangle} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle} \oplus S_{7_{\langle 1\rangle}} \oplus S_{7_{\langle 2\rangle}}. \end{aligned}$$

If all of these relations hold for one γ, we can be sure with sufficiently high probability that a read-collision has indeed happened. A probable hypothesis for K_0 is now given by

$$\gamma_{\langle 3\rangle} \oplus S_{7_{\langle 0\rangle}} || \gamma_{\langle 1\rangle} \oplus p_{\langle 6\rangle} \oplus \alpha_{\langle 2\rangle} || \gamma_{\langle 2\rangle} \oplus S_{7_{\langle 2\rangle}} || \gamma_{\langle 0\rangle} \oplus p_{\langle 4\rangle} \oplus \alpha_{\langle 0\rangle} || p_{\langle 3\rangle} \oplus \alpha_{\langle 3\rangle} || p_{\langle 2\rangle} \oplus \alpha_{\langle 2\rangle} || p_{\langle 1\rangle} \oplus \alpha_{\langle 1\rangle} || p_{\langle 0\rangle} \oplus \alpha_{\langle 0\rangle}.$$

Our method detects all read-collisions, but there may also be false positives, therefore the process described above must be iterated for a few times for different portions of the keystream. Typically, over time, one or two hypotheses occur more often than others and distinguish themselves quite fast from the rest. Experiments show that about a dozen key-frames are usually enough so that the correct key byte is among the first two hypotheses. The principle we outlined above not only works for K_0, it also allows to recover the value of K_4 when $\alpha \in \{(0,1,1,0)_2, (1,1,1,0)_2\}$, $i \in \{12 + 15x | x \in \mathbb{N}\}$ are chosen appropriately.

In the following we assume that we have obtained a set of hypotheses for K_0—we might also have K_4, but this improves the efficiency of the remainder of the attack only slightly. Based on these hypotheses, starting with the most plausible one, we can brute-force the remaining key bytes separately. Please note that the following process will only produce the correct key, if our hypothesis for K_0 was correct. To obtain $K_1, \ldots, K_7$, we examine a few keystream-bytes for a second time, while focusing on the $\mathscr{F}$-component. For each K_j with $j \in \{0, 1, \ldots, 7\}$ for which we already have a hypothesis, we can use the corresponding key-stream bytes $\left(Z_{i+j}, Z_{i+j-1}, Z_{i+j-8}\right)_{2^8}$ with $i \in \{8 + 15x | x \in \mathbb{N}\}$ to compute

$$\alpha = \mathscr{N}(j \bmod 2, K_j \oplus Z_{i+j-1}).$$

If we do not already have a plausible hypothesis for K_k with $k = \mathcal{T}_1(\alpha)$, we can simply try out all possible values $\delta \in \{0, 1, ..., 255\}$ and compute the output of the cipher. If we find for one value that the output equals Z_{i+j} we keep δ as hypothesis for K_k. This can be repeated for a few different i until a hypothesis for the full key has been recovered. Since the validity of the full hypothesis solely depends on the correctness of K_0, we must verify each key candidate by generating and comparing keystream.

The overall complexity of this attack depends on how many hypotheses for K_0 are used to derive the remaining key. Given 15-20 key-frames, the correct byte for K_0 is usually ranked as best hypothesis so deriving the complete key means testing

$$(7 \cdot 2^8)/2 \approx 2^{10}$$

single byte hypotheses for the missing bytes (on average). Clearly, a keystream/time trade-off is possible: the more key-frames are available to test hypotheses for K_0, the more the right hypothesis distinguishes itself from all others. As a matter of fact, the most extreme trade-off is simply trying all 2^8 possible values for K_0 (without even ranking them), which reduces the required amount of known keystream to about 400–500 bits but increases the computational complexity to

$$(7 \cdot 2^8 \cdot 2^8)/2 \approx 2^{18}$$

guesses on average.

4.7 Discussion and Future Work

The effort to find the cipher in the firmware was quite considerable. It started with understanding proprietary hardware, software and firmware formats, went over developing a “poor man’s version” of IDA for the Blackfin DSP and culminated in locating a fraction of code within a 300 000 line text file. However, without much effort on the side of Inmarsat, the reverse engineering process could have been considerably harder:

1. Due to the fact that the firmware was available online, the reverse engineering process involved only software. Obtaining the firmware from an actual device would have added the need for expertise in the area of hardware and possibly hardware reverse engineering.
2. Although our initial assumption that the code referring to `ApplyCipher` implements the cipher was wrong, leaving assertions in the binary provided unnecessary clues. Removing these strings would have been easy enough for Inmarsat, while leaving us with no way to start our analysis of the firmware.

We have successfully recovered the stream cipher used in the GMR-2 network and constructed a very efficient known-plaintext attack on A5-GMR-2. To analyze the availability of known plaintext, and make our results more practical, we would need access to live data from the Inmarsat network. However, there are currently no initiatives similar to OsmocomGMR (which targets GMR-1 only) thus obtaining data requires implementing the modulation schemes (for instance as part of Osmocom). However, a recent publication at a conference in South America [OM12], indicates that it is possible to extract live data from Inmarsat’s network. The authors describe how they have patched the firmware of a satphone in order to be able to read out internal buffers with the help of a serial console. To us, it seems entirely possible that parts of the encrypted data in GMR-2 are predictable, since similar behavior was observed in the case of GMR-1.

CHAPTER 5

SECURITY ANALYSIS OF THE SIMONSVOSS 3060 LOCKING SYSTEM

This chapter presents our work on the cryptanalysis of a proprietary authentication scheme, used in the second generation of the SimonsVoss digital locking system.

5.1 Motivation

In the world of access control by electronic means, various manufacturers have been inventing their own cryptographic primitives and protocols. Often, the resulting designs are driven by requirements for low cost and overall footprint. In order to further improve security, manufacturers regularly choose to resort to obscurity and not reveal the results of their creativity to an educated audience. The past decade has shown that the vast majority of these schemes is flawed and that once the ciphers have been reverse engineered and become public, they can be broken with low to modest efforts.

The system we describe and analyze in this chapter was manufactured by SimonsVoss Technologies AG. SimonsVoss is a market leader for electronic locking and access control systems. According to recent information[1], the company installed its one millionth digital locking cylinder in April 2012 and has sold more than three million corresponding transponders. The list of customers and objects secured with this technology in Europe, USA, and Asia, as listed on the official website[2], is very impressive: it includes residential buildings, tourist apartments, hospitals, universities, embassies, major banks, airports, buildings of the German armed forces and the US army, factory sites of well-known brands, police stations, stadiums, town halls, prisons and many others.

Motivated by the fact that a system manufactured by SimonsVoss was recently installed in buildings of the Ruhr-University Bochum, we set out to reverse engineer and analyze the employed authentication scheme. After all, it was not only a distant academic interest in the provided security guarantees (motivated by widespread adoption of this technology), but also a personal quest to understand the safety of one's own belongings.

---

[1] See http://www.simons-voss.us/Record-sales-in-2011.1112.0.html?&L=6

[2] See http://www.simons-voss.com/References.1163.0.html?&L=1

5.2 Related Work

One of the first examples of a proprietary cryptographic system in the context of this chapter is the DST40 cipher. It was used in Texas Instrument's Digital Signature Transponder (DST) which has been reverse engineered [BGS$^+$05] in 2005: knowing at least two challenge/response pairs, the 40-bit secret key of a corresponding transponder can be revealed by means of a brute-force attack in less than one day. Likewise, following the reverse engineering of NXP's Mifare Classic cards [NESP08] through analyzing the silicon die, the used Crypto1 cipher was found to be weak, relying on a state of only 48 bits. Further mathematical weaknesses of the cipher and implementations flaws, e. g., a weak random number generator, enable to reveal all secret keys and practically circumvent the protection mechanisms with a card-only attack in minutes [GvRVWS09, Cou09].

The Hitag 2 transponders of the same manufacturer—widely used for car immobilizers and Remote Keyless Entry (RKE) systems—were found to be flawed after the cipher became public [CONQ09]. Based on the latest results [VGB12], their secret keys can be extracted in six minutes. Further insecure products for access control include HID Global iClass and Legic Prime cards, both based on highly ineffective cryptographic measures [GdKGVM12, PN09].

5.3 Technical Background

Here we will shortly introduce the elements commonly found in an installation of the SimonsVoss 3060 locking system. Since this entire chapter focuses on the cryptanalysis of the system, many of the low level details are left out. They can be found in our publication [SDK$^+$13].

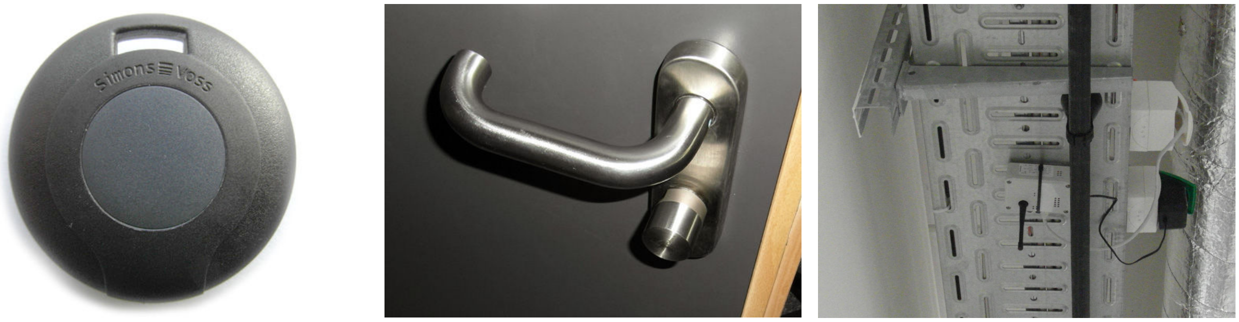

(a) Transponder 3064 (b) Door lock cylinder 3061 (c) WaveNet node 3065

Figure 5.1: Components of the SimonsVoss 3060 digital locking system

The visible part of the digital locking system 3060 installed in a building of the Ruhr-University Bochum consists of three parts, see also Figure 5.1.

1. **Transponder 3064.**
   Transponders serve as replacements for traditional keys and have only one central button, which allows the transponder to start authentication with a nearby door lock.
2. **Lock 3061.**
   RF-enabled locking cylinders 3061 are mounted on doors. When authentication with a transponder was successful, the door cylinder beeps twice, indicating that the lock can be opened or closed during the next few seconds (with manual force, by turning a knob that is attached to the cylinder).

3. **WaveNet node 3065**.
   Several locks are commonly assigned to so called WaveNet nodes, which allow communication between a lock and the central backend system.

The widespread digital locking system 3060 is also termed “Generation 2” or “G2-based” system by the manufacturer. It supports up to 64,000 digital locking cylinders of type 3061 per installation, up to 64,000 transponders 3064 per lock, and the storage of up to 1,000 access instances on the transponder.

Transponders and locks communicate on a wireless link at 25 KHz and use a proprietary protocol, as well as undisclosed authentication primitives. It is this link between transponders and doors that we analyze here. Due to the low frequency and minimal output power of the transponders, this wireless connection works only when lock and transponder are in close proximity (i.e., for distances of up to a few centimeters).

In the “online” version of the 3060 system, locks can have a permanent connection to a central server through multiple WaveNet nodes at 868 MHz. The (successful or unsuccessful) opening attempt of any transponder at any door lock in the system can be monitored and stored in logfiles. Additionally, this feature allows to maintain and configure locks individually from a central location, which is important for large scale installations. Please note that this link is not considered in this chapter.

5.4 Reverse Engineering

The protocol responsible for authenticating locks and transponders, as well as the cryptographic primitives, were reverse engineered from actual hardware by Daehyun Strobel, Falk Schellenberg and David Oswald (all members of the Chair for Embedded Security).

In order to obtain the full authentication protocol, the Integrated Circuit (IC) of a transponder and a lock was opened with fuming acid. In both cases, a fuse-bit had to be erased with UV-C light, which then enabled reading out the respective firmwares. Disassembly of the firmware with IDA Pro allowed to understand the protocol, as well as the involved cryptographic primitives and constructions.

More details on the actual process and the involved tools can be found in our joint publication [SDK+13] and the respective, upcoming Ph.D. theses.

5.5 The G2 Authentication System

In this section we present the results of reverse engineering the G2 authentication system. We present the protocol used for mutual authentication, as well as the primitives used in this protocol. Please note that some facts have been left undisclosed, to enable SimonsVoss to respond to the attacks we present in Section 5.6.

5.5.1 Keys & Protocol

In the following, we present the essential results of reverse engineering the software running on the transponder and the lock. We focus on the keys, protocol, and the cryptographic primitives used to mutually authenticate transponders and locks.

For a successful execution of the authentication protocol, transponder and lock must be in possession of a shared secret. For this purpose, each transponder has a (unique) 128-bit long-term secret $K_T \in \{0,1\}^{128}$. On the other hand, each lock stores a set of four 128-bit keys that are *identical for every lock in the entire* *installation*. In the following, we refer to the set of the keys as the *system key*:

$$K_L = \left(K_{L_0}, K_{L_1}, K_{L_2}, K_{L_3}\right)_{2^{128}} \quad \text{with} \quad K_{L_i} \in \{0,1\}^{128} \quad \text{for all} \quad 0 \le i \le 3.$$

Based on this key, the lock can derive any transponder key, as will be explained in the course of the protocol.

The system uses an 11-step challenge-response protocol to achieve mutual authentication between transponder and lock. A protocol run is initiated by the transponder when the central button is pressed in proximity of a lock. In the course of this authentication step, a multitude of messages is exchanged, see Figure 5.2.

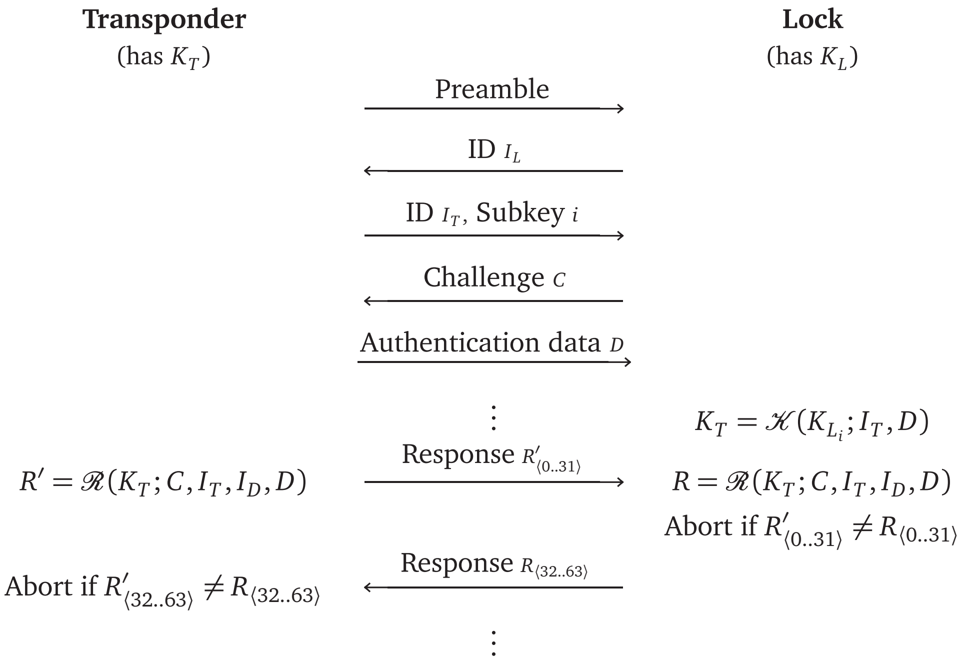

Transponder
(has K_T)
Lock
(has K_L)
Preamble
ID I_L
ID I_T, Subkey i
Challenge C
Authentication data D
$K_T = \mathscr{K}(K_{L_i}; I_T, D)$
Response $R'_{\langle 0..31\rangle}$
$R' = \mathscr{R}(K_T; C, I_T, I_D, D)$
$R = \mathscr{R}(K_T; C, I_T, I_D, D)$
Abort if $R'_{\langle 0..31\rangle} \neq R_{\langle 0..31\rangle}$
Response $R_{\langle 32..63\rangle}$
Abort if $R'_{\langle 32..63\rangle} \neq R_{\langle 32..63\rangle}$

Figure 5.2: Protocol for mutual authentication between transponder 3064 and cylinder 3061

Here we only focus on a subset of the messages that we identified to be relevant for our security analysis. Please also note that the descriptions of the respective functions used in the protocol follow after this section. By observing several protocol runs, we have identified these messages:

I_L Each lock has a 24-bit ID that is transmitted to the transponder.
I_T Each transponder has a 32-bit ID that is transmitted to the door.
C After the ID exchange, the lock sends an 88-bit challenge C to the transponder.
D The transponder sends 80 bits of authentication data to the lock. This includes the result of its ID verification and transponder-specific data.

In the first four steps of the protocol, most of the messages are fixed for a transponder/lock combination. Only C (and conversely the responses $R'_{\langle 0..31\rangle}$ and $R_{\langle 32..63\rangle}$) change between protocol runs—and of this 88-bit value, only 40 bits are actually random. The remaining bits are either fixed or change infrequently. In answer to such a challenge, both transponder and lock derive the same 64-bit response R using the data exchanged in previous messages and the 128-bit long-term secret K_T of the transponder. We denote the function to compute the response as $\mathscr{R}$, with

$$R = R_{\langle 0..31\rangle} || R_{\langle 32..63\rangle} = \mathscr{R}(K_T; C, I_T, I_D, D).$$

The main part of the authentication is then accomplished by exchanging the following two messages:

$R'_{\langle 0..32\rangle}$ The transponder sends the first 32-bit half of R' as the first response to the lock.

$R_{\langle 32..63\rangle}$ If $R_{\langle 0..31\rangle}$ matches the first half of R' computed by the lock, it sends the second 32-bit half of R to the transponder.

As each party computes the full 64-bit output of $\mathscr{R}$, both can verify the response of the other party and mutually authenticate each other on the basis of K_T. Instead of storing the key K_T for each transponder, a lock is able to derive K_T from a transponder's ID I_T, the authentication data D, and part of the long-term system key. A key derivation function $\mathscr{K}$ is used for this purpose, i.e.,

$$K_T = \mathscr{K}(K_{L_i}; I_T, D) \quad \text{for some} \quad 0 \leq i \leq 3.$$

Please note that the index i, selecting one of the four subkeys of the system key, is selected during the execution of the authentication protocol.

5.5.2 Cryptographic Primitives

In the authentication protocol, the two basic functions $\mathscr{K}$ (for key derivation on the door's side) and $\mathscr{R}$ (for response computation) are used. These functions are proprietary constructions and share two building blocks we denote as $\mathscr{O}$ and $\mathscr{D}$. While it turned out that $\mathscr{D}$ is simply a modified Data Encryption Standard (DES) [Nat77], what we call "the obscurity function" $\mathscr{O}$ is a more intricate design, which we are describing in the following. Please note that, in order to give SimonsVoss time to upgrade their security, we do not disclose the modifications that were applied to the DES at this time.

The $\mathscr{O}$ function takes two 128-bit inputs (a plaintext and a key) and returns a 128-bit output (the ciphertext). Figure 5.3 shows the internal structure of $\mathscr{O}$. This function operates byte-wise on two registers with 16 8-bit cells. The upper registers are continuously updated while the lower registers remain constant.

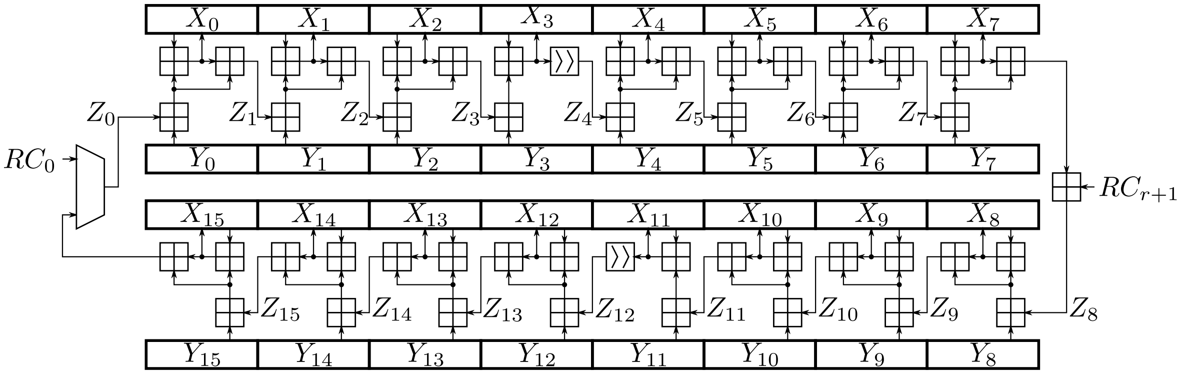

X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7
Z_0 Z_1 Z_2 Z_3 Z_4 Z_5 Z_6 Z_7
Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7
RC_0
RC_{r+1}
X_{15} X_{14} X_{13} X_{12} X_{11} X_{10} X_9 X_8
Z_{15} Z_{14} Z_{13} Z_{12} Z_{11} Z_{10} Z_9 Z_8
Y_{15} Y_{14} Y_{13} Y_{12} Y_{11} Y_{10} Y_9 Y_8
⟩⟩⟩

Figure 5.3: Structure of the obscurity function

To compute the output of $\mathscr{O}$ the upper X registers are initialized with the plaintext (we denote this state as $X^{(0)}$), while the lower Y registers are set to the key. After that, the registers are updated successively, all in all each of the X registers is updated for 8 times, according to the following scheme: updates start with X_0, then X_1, etc., and are computed mostly as sums of 8-bit values modulo 256. Additionally, each cell update incorporates an 8-bit chaining value Z_i, which is the result of the update of the preceding cell. The update equation for the successive state $X_i^{(r+1)}$ from $X_i^{(r)}$ for all $0 \leq i \leq 15$ is given as

$$X_i^{(r+1)} = X_i^{(r)} + Y_i + Z_i^{(r)} + \bmod 2^8 \quad \text{with} \quad 0 \leq r < 8. \tag{5.1}$$

There are basically three ways the chaining value is computed for any round r,

$$Z_{i+1}^{(r)} = \begin{cases} X_i^{(r)} + 2\left(Y_i + Z_i^{(r)}\right) \bmod 2^8 & \text{if } i \in \{0,\dots,14\} \setminus \{3,7,12\}, \\ X_i^{(r)} + 2\left(Y_i + Z_i^{(r)}\right) + RC_{r+1} \bmod 2^8 & \text{if } i = 7, \\ \left(Y_i + Z_i^{(r)} \bmod 2^8\right) + X_i^{(r)} \ggg 1 \bmod 2^8 & \text{if } i \in \{3,12\}. \end{cases} \tag{5.2}$$

with the first Z_0 being assigned depending on r, i.e.,

$$Z_0^{(r)} = \begin{cases} RC_0 & \text{if } r = 0, \\ X_{15}^{(r-1)} + 2\left(Y_{15} + Z_{15}^{(r-1)}\right) \bmod 2^8 & \text{if } 0 < r \leq 8. \end{cases}$$

Please note that all RC_r are 8-bit round constants, which we do not disclose at this time (cf. Subsection 5.7.2). The function's output is given by the contents of the X cells after 8 rounds, i.e.,

$$X^{(8)} = \mathscr{O}\left(Y; X^{(0)}\right).$$

$\mathscr{K}$: Key Derivation Function

In the following we describe how $\mathscr{D}$ and $\mathscr{O}$ are combined to construct the key derivation function, which is used only in the door. This function can be decomposed into three blocks, $\mathscr{D}$ and two instances of the aforementioned obscurity function $\mathscr{O}$, as illustrated in Figure 5.4.

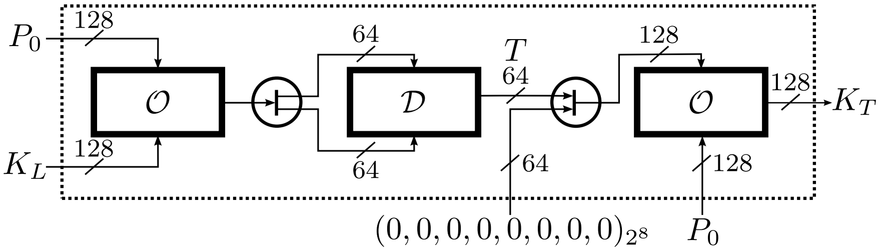


Figure 5.4: Construction of the key derivation function

The construction has four inputs, one fixed to 64 zero bits and two other 128-bit inputs that are exchanged in the authentication protocol: the value P_0, used twice during key derivation, is composed of the first three bytes $I_{T_0}, I_{T_1}, I_{T_2}$ of the transponder ID I_T and the first three bytes D_0, D_1, D_2 of the authentication data D. The last of each of these bytes is masked by a Boolean AND-operation with the fixed constant `0xC7` or `0x3F`, respectively, thus selecting only certain bits. All other bytes are filled with zeros, i. e.,

$$P_0 = \left(I_{T_0}, I_{T_1}, I_{T_{2\langle 0..2\rangle}} || 0^3 || I_{T_{2\langle 6\rangle}} || I_{T_{2\langle 7\rangle}}, D_0, D_1, D_{2\langle 0..5\rangle} || 0^2, 0, \dots, 0\right)_{2^8}.$$

Only one input of $\mathscr{K}$ is secret: one of the four 128-bit keys of the system key set is selected according to the two most significant bits of the third byte of I_T. This 128-bit key is used as key K_L for the first instance of $\mathscr{O}$ to encrypt P_0. The output of this operation is split into two 64-bit halves, which are used as plaintext and key for $\mathscr{D}$. The output of $\mathscr{D}$, denoted by T, is then concatenated with 64 zero bits, and the result is encrypted with $\mathscr{O}$—using P_0 as the key. The resulting 128-bit value is the transponder's key K_T, i. e.,

$$\mathscr{K}(K_L; P_0) = \mathscr{O}(P_0; \mathscr{D}\left(\mathscr{O}(K_L; P_0)_{\langle 64..127\rangle}; \mathscr{O}(K_L; P_0)_{\langle 0..63\rangle}\right) || 0^{64}).$$

$\mathscr{R}$: Response Computation Function

The structure of the response computation $\mathscr{R}$ is very similar to the key derivation function $\mathscr{K}$. However, the way the building blocks are combined is different. Figure 5.5 shows the internal structure of $\mathscr{R}$.

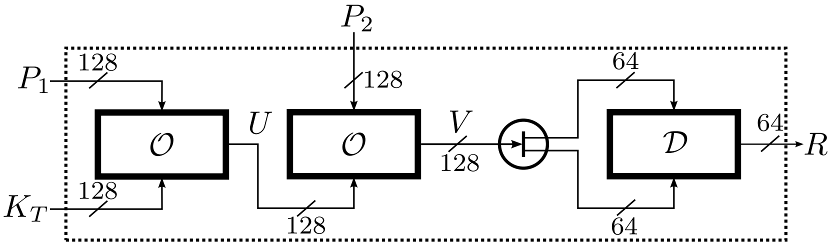


Figure 5.5: Construction of the response computation function

Again two instances of the proprietary obscurity function $\mathscr{O}$ are used along with the modified DES $\mathscr{D}$. The 128-bit input P_1 to $\mathscr{R}$ is the concatenation of the challenge C and part of the authentication data D, i. e.,

$$P_1 = (C_0, C_1, \ldots, C_{10}, D_6, D_7, D_8, D_9, 0)_{2^8}.$$

The output of the first instance of $\mathscr{O}$ is used as key for the second iteration of $\mathscr{O}$. The 128-bit input P_2 is fixed for every transponder/lock combination and is composed of more bytes taken from the IDs of lock and door, i. e.,

$$P_2 = (I_{L_2}, I_{T_2}, I_{T_3}, D_3, D_4, D_5, 0, \ldots, 0)_{2^8}.$$

The output of this operation is split into two 64-bit halves, whereas the first half is used as plaintext for $\mathscr{D}$ and the second as the respective key. The two halves of the 64-bit result R form the responses R_0 and R_1 used in the protocol, i. e.,

$$\mathscr{R}\left(K_T; P_1, P_2\right) = \mathscr{D}\left(\mathscr{O}\left(\mathscr{O}\left(K_T; P_1\right); P_2\right)_{\langle 64..127\rangle}; \mathscr{O}\left(\mathscr{O}\left(K_T; P_1\right); P_2\right)_{\langle 0..63\rangle}\right)$$

5.6 Cryptanalysis

In the following we present three non-invasive attacks that allow to recover the longterm key K_T of a particular transponder. The presented attacks exploit a subset of the following weaknesses we have discovered in the design of the authentication scheme:

1. When the door computes

   $$R^{(t)} = \mathscr{R}\left(K_T; P_1^{(t)}, P_2^{(t)}\right)$$

   to verify the transponder's response, 40 bits of the internally computed DES key are used as part of the next challenge, i. e.,

   $$P^{(t+1)}_{1_{\langle 16..55\rangle}} = \mathscr{O}\left(\mathscr{O}\left(K_T; P_1^{(t)}\right); P_2^{(t)}\right)_{\langle 64..103\rangle}$$

   In the following denote this string of 40 leaked bits as $V_L^{(t)}$.
2. Looking at one instance of $\mathscr{O}$ and the equations describing the LSBs of each X cell after 8 rounds, these bits are only dependent on 32 bits of the key. More specifically, the equations reveal that the LSBs of the Y cells occur in non-linear combinations, while the bits next to the LSBs occur only in linear combinations.

This observation can be generalized for any bit b in the 16 output bytes, for cases where M instances of $\mathscr{O}$ are chained (like in $\mathscr{R}$). Here, the $M + b$ lower bits per byte of the key are found in non-linear combinations in the output bits at position b, while bits at position $M + b + 1$ are only appearing in linear combinations.

Thus, in a sense even multiple instances of $\mathscr{O}$ resemble a T-function (a concept we introduce and explain in Section 5.6.1), making it significantly easier to invert them.

3. The $\mathscr{O}$ construction is quite predictable when trying to estimate XOR differences in output pairs when given a specific input difference. For instance, for the first two blocks of the response computation function, i.e., $V = \mathscr{O}(\mathscr{O}(K_T; P_1); P_2)$, we easily found some input differences (for the first application of $\mathscr{O}$)
$$\Delta P_1 = P_1 \oplus P_1' \quad \text{with} \quad K_T, P_1, P_1', P_2 \in \{0,1\}^{128}$$
for which the corresponding output difference
$$\Delta V = \mathscr{O}\left(\mathscr{O}(K_T; P_1); P_2\right) \oplus \mathscr{O}(\mathscr{O}(K_T; P_1 \oplus \Delta P_1); P_2)$$
is highly predictable.
4. The output of $\mathscr{K}$, and thus every transponder key, has actually only 64 bits of entropy (the output of $\mathscr{D}$). We denote this 64-bit value as T, which—if recovered—allows to compute the full 128-bit key K_T, if the corresponding P_0 is known. Note that this fact alone allows to break the scheme in practice using dedicated hardware.

Especially the first item is a crucial flaw in the protocol. However, it is a well-known fact that obtaining “good” random numbers in (constrained) embedded systems is hard and costly, therefore it is not entirely surprising that these seemingly random looking bits are re-used as challenge.

5.6.1 Differential Attack

The following attack is—chronologically speaking—the first attack to break the system in a practical way. It exploits the first three of the four listed weaknesses, i.e., the 40-bit leak of the internal state, the resemblance of a T-function of $\mathscr{O}$ and the distribution of XOR differences. The naive version of this attack works in two stages:

1. From the observation of two full authentication runs, we know inputs P_1, P_2 and output R of $\mathscr{R}$, as well as 40 bits V_L of its internal state. Obtain the set of all potential outputs $\mathbb{V}$ of the second instance of $\mathscr{O}$ by inverting $\mathscr{D}$ for all possible keys.
2. Given any pair of inputs $P_2, V \in \mathbb{V}$ for $\mathscr{O}$, we have to obtain its “key” U, with $V = \mathscr{O}(U; P_2)$, which is the output of the first instance of $\mathscr{O}$. Given U, P_1, we have to find candidates for K_T with $U = \mathscr{O}(K_T; P_1)$ and verify them accordingly.

Assuming that inverting $\mathscr{O}$ with two arbitrary 128-bit inputs always yields only one key candidate (which is not the case), we would have to perform

$$\underbrace{2^{21}}_{\#\text{DES keys}} \cdot \underbrace{2^{3}}_{\#\text{outputs/key}} \cdot \underbrace{2}_{\#\text{inversions/output}} = 2^{25}$$

inversions of $\mathscr{O}$ in the worst case. In order to reduce this and enable a more efficient attack, we exploit the weak distribution of XOR differences, which we will describe now. After that, we will describe how to invert $\mathscr{O}$, once one of its inputs and the output is known. From the latter description, the entire attack follows canonically.

Finding and Filtering Outputs

Here we describe, how we can use publicly observable data of several, successful authentication runs to effectively reduce the set of all potential outputs of the second instance of $\mathscr{O}$ in $\mathscr{R}$. The general idea is to use highly predictable XOR differences after $\mathscr{O}$ and before $\mathscr{D}$ to distinguish right guesses for the key of the DES from wrong ones.

ΔP_1	$\#\Delta V$
0x00000000800000000000000000000000	64
0x00000000000000000000000080800000	64
0x00000000808000000000000000000000	128

Table 5.1: Number of occurrences of different XOR differences ΔV

We will shortly explain what we mean by “highly predictable XOR differences”: for a fixed set of $P_2, K \in_R \{0,1\}^{128}$ and L random choices of $P_1^{(i)} \in_R \{0,1\}^{128}$ we have counted how many different output differences

$$\Delta V^{(i)} = \mathscr{O}\left(\mathscr{O}\left(K; P_1^{(i)}\right); P_2\right) \oplus \mathscr{O}\left(\mathscr{O}\left(K; P_1^{(i)} \oplus \Delta P_1\right); P_2\right) \quad \text{with} \quad 0 \le i < L \tag{5.3}$$

we observe. Table 5.1 shows some of the results we have obtained[3] by experimenting with different input XOR differences. The first row indicates that, if we choose

$$\Delta P_1 = \text{0x00000000800000000000000000000000} \quad \text{and} \quad L \gg 64,$$

we will observe only 64 different output differences as described by Equation 5.3. From these 64 output differences we can derive the *expected difference* which we denote as a string of 128 characters, i.e., $\Delta_E V \in \{\text{'0'},\text{'1'},\text{'?'}\}^{128}$. A ’0’ here means the output differences always have a 0 at the respective position, a ’1’ means it is always 1. These bits we call the *known difference bits* since they are fixed, independent of an actual key and can be determined in advance. A ’?’ indicates that at these positions the output differences assume different values, depending on the key. Conversely, we call these bits the *unknown difference bits*. The first row of the table thus indicates that we can expect a $\Delta_E V$ with 122 known and 6 unknown difference bits.

Assuming that, by observing N successive authentications between a genuine transponder and lock, we have obtained a set

$$\mathbb{T} := \left\{\left(P_1^{(0)}, P_2, R^{(0)}\right), \ldots, \left(P_1^{(N-1)}, P_2, R^{(N-1)}\right)\right\}$$

of input/output tuples (P_2 is always constant), we use the following method to find a tuple that promises to most dramatically reduce the number of potential outputs.

1. Successively pick input/output tuples $\left(P_1^{(i)}, P_2, R^{(i)}\right)$ with $0 \le i < N-1$.
    a) For a given $\left(P_1^{(i)}, P_2, R^{(i)}\right)$ successively pick pairs $\left(P_1^{(j)}, P_2, R^{(j)}\right)$ for all $i < j < N-1$.
        i. For $P_1^{(i)}, P_1^{(j)}$ compute the XOR difference in the output of $\mathscr{O}$ for a set of L randomly chosen keys $K^{(0)}, \ldots, K^{(L-1)} \in_R \{0,1\}^{128}$, i.e.,

$$\Delta V^{(l)} = \mathscr{O}\left(\mathscr{O}\left(K^{(l)}; P_1^{(i)}\right); P_2\right) \oplus \mathscr{O}\left(\mathscr{O}\left(K^{(l)}; P_1^{(j)}\right); P_2\right)$$

with $0 \le l < L$.

---

[3] To find the presented input differences (and more), we have used a simple genetic algorithm which explores XOR differences in the input.

ii. From the observed output differences $\Delta V^{(l)}$ with $0 \leq l < L$ derive the expected difference $\Delta_E V$.
iii. Keep track of (i, j) with XOR difference $\Delta P_1 = P_1^{(i)} \oplus P_1^{(j)}$ for which the expected output difference $\Delta_E V$ has the least amount of unknown difference bits. A more refined expression of this criterion which will be described later.

The execution time of this method to determine a good filter depends on the choice of N and L, i.e.,

$$\left(\frac{(N-1)^2 + (N-1)}{2}\right) \cdot L \cdot 4$$

is the number of inner computations of $\mathcal{O}$ in the filter generation algorithm. However, for all of our experiments, it turned out that this step is negligible with regard to its computation time.

Given a selection (i, j) of $\mathbb{T}$ as output of above algorithm, we continue to use

$$P_1^{(i)}, P_2, R^{(i)}, V_L^{(i)} = P_{1_{\langle 16..55\rangle}}^{(i+1)} \quad \text{and} \quad P_1^{(j)}, P_2, R^{(j)}, V_L^{(j)} = P_{1_{\langle 16..55\rangle}}^{(j+1)} \quad \text{with} \quad \Delta P_1 = P_1^{(i)} \oplus P_1^{(j)}$$

and the corresponding expected output difference $\Delta_E V$ as filter. We proceed with finding the set $\mathbb{V}$ of potential outputs with

$$\mathcal{O}\left(\mathcal{O}\left(K_T; P_1^{(i)}\right); P_2\right) \in \mathbb{V},$$

where K_T is unknown. It has to be kept in mind that each $V \in \mathbb{V}$ is the concatenation of the DES plaintext V_P and the used key V_K, i.e., $V = V_P || V_K$. Of the 64-bit key, we know the first 40 bits because they are re-used as succeeding challenge, but 24 bits remain unknown. Of these 24 bits, only 21 are effectively used by the DES. Before we proceed with describing the actual algorithm, we need to introduce two helper methods which operate on the expected XOR difference (or parts thereof):

$\psi\left(\Delta_E V\right)$ Returns the number of unknown difference bits, but ignores the positions belonging to the parity bits of the DES key.

$\xi\left(\Delta_E V, \beta\right)$ Replaces the unknown positions in the expected difference string with single bits taken from β. The second input is expected to be a string of as many bits as the first input has '?'. Also ignores parity bits.

Our attack iterates over the 21 unknown DES key bits (the parity bits are initially set to zero) in order to obtain a minimal set $\mathbb{V}$:

1. For $P_1^{(i)}, P_1^{(j)}$ compute the expected difference $\Delta_E V$ in the output of $\mathcal{O}$ and input of $\mathcal{D}$ respectively.
2. Let $\mathbb{V}$ be the empty set, $\mathbb{V} := \emptyset$.
3. Iterate over all 2^{21} possible 21-bit strings α and do the following:
    a) For $R^{(i)}, R^{(j)}$ construct the respective key hypotheses $\gamma^{(i)}, \gamma^{(j)}$ from the leaked parts of the key and the guessed value α, i.e.,

    $$\begin{array}{rcl} \gamma^{(i)} & = & V_L^{(i)} || \alpha_{\langle 0..6\rangle} || 0 || \alpha_{\langle 7..13\rangle} || 0 || \alpha_{\langle 14..20\rangle} || 0 \quad \text{and} \\ \gamma^{(j)} & = & V_L^{(j)} || \alpha_{\langle 0..6\rangle} || 0 || \alpha_{\langle 7..13\rangle} || 0 || \alpha_{\langle 14..20\rangle} || 0. \end{array}$$

    b) Decrypt $R^{(i)}$ with the modified DES accordingly, i.e.,

    $$\delta^{(i)} = \mathcal{D}^{-1}\left(\gamma^{(i)}; R^{(i)}\right),$$

    thus obtaining a hypothesis for the corresponding DES plaintext.

c) Determine $n = \psi\left(\Delta_E V_{\langle 104..127\rangle}\right)$ and iterate over all 2^n possible n-bit strings β and do the following:
   i. Modify the original hypothesis for $R^{(j)}$ according to the guessed value β for the unknown difference bits and decrypt accordingly, i.e.,

   $$\delta^{(j)} = \mathscr{D}^{-1}\left(\gamma^{(j)} \oplus \left(0^{40} \| \xi\left(\Delta_E V_{\langle 104..127\rangle}, \beta\right)\right); R^{(j)}\right),$$

   thus obtaining a hypothesis for the DES plaintext (based on the guessed values α and β).
   ii. Test whether $\delta^{(i)} \oplus \delta^{(j)}$ matches the known difference bits of $\Delta_E V_{\langle 0..63\rangle}$, which indicates that $\gamma^{(i)}$ might be correct. If this is the case, add the eight[4] possible variants of $\delta^{(i)} \| \gamma^{(i)}$ as potential outputs of $\mathscr{O}$ to $\mathbb{V}$.

If $\psi\left(\Delta_E V_{\langle 104..127\rangle}\right)$ is the number of unknown difference bits in the guessed part of the DES key, the number of DES decryptions and tests of this algorithm is $2^{21+\psi\left(\Delta_E V_{\langle 104..127\rangle}\right)}$ (in the worst case). Consequently, depending on the amount of unknown bits $\psi\left(\Delta_E V_{\langle 0..63\rangle}\right)$ in the plaintext part of the DES inputs, the size of $\mathbb{V}$ can be estimated by

$$|\mathbb{V}| \approx 8 \cdot \begin{cases} 2^{\psi\left(\Delta_E V_{\langle 104..127\rangle}\right)+\psi\left(\Delta_E V_{\langle 0..63\rangle}\right)-43} & \text{if } 21+\psi\left(\Delta_E V_{\langle 104..127\rangle}\right) > 64-\psi\left(\Delta_E V_{\langle 0..63\rangle}\right), \\ 1 & \text{if } 21+\psi\left(\Delta_E V_{\langle 104..127\rangle}\right) \le 64-\psi\left(\Delta_E V_{\langle 0..63\rangle}\right). \end{cases}$$

The intuition behind this approximation is that we need one known bit in the expected difference to validate (or filter) one bit we guess; we guess $21+\psi\left(\Delta_E V_{\langle 104..127\rangle}\right)$ bits in total while $64-\psi\left(\Delta_E V_{\langle 0..63\rangle}\right)$ is the amount of known bits in the plaintext part of the XOR differences. If we guess less bits than we know, our filter is most efficient and thus the size of $\mathbb{V}$ is minimal. Please also note that it only makes sense to filter $\mathbb{V}$ in this way, when it can be expected that $|\mathbb{V}| \ll 2^{24}$. In any other case, directly trying out all 2^{24} possibilities for the DES plaintext (without filtering via XOR differences) would be more efficient.

Inverting the Obscurity Function

The proprietary construction we call the “obscurity function” resembles a T-function, a concept presented by Alexander Klimov in 2002 [KS02, KS03]. A T-function is a mapping that updates each bit of the state of the cipher based on a linear combination of the same bit and some function f of all less significant bits. Assuming we have a cipher with an n-bit state we denote as $x = \left(x_{\langle 0\rangle}, x_{\langle 1\rangle}, \ldots, x_{\langle n-1\rangle}\right)_2$, the corresponding description of the update function in the sense of a T-function would be as follows:

$$x^{(t+1)}_{\langle i\rangle} = x^{(t)}_{\langle i\rangle} \oplus f\left(x^{(t)}_{\langle 0..i-1\rangle}\right) \quad \text{for all} \quad 0 \le i < n \quad \text{and} \quad t > 0.$$

If we consider the update function of $\mathscr{O}$ (cf. Equation 5.1) of each 8-bit cell X_i with $0 \le i \le 15$ over 8 rounds we get

$$X^{(r+1)}_{i_{\langle j\rangle}} = X^{(r)}_{i_{\langle j\rangle}} \oplus Y_{i_{\langle j\rangle}} \oplus Z^{(r)}_{i_{\langle j\rangle}} \oplus f_U\left(X^{(r)}_{i_{\langle 0..j-1\rangle}}, Y_{i_{\langle 0..j-1\rangle}}, Z^{(r)}_{i_{\langle 0..j-1\rangle}}\right) \tag{5.4}$$

for all $0 \le j \le 7$ and $0 \le r < 8$. However, it should be noted that all Z_i themselves depend on the preceding X and Y values in a non-linear way, i.e.,

$$Z^{(r)}_i = f_Z\left(X^{(r)}_0, \ldots, X^{(r)}_{i-1}, Y_0, \ldots, Y_{i-1}\right).$$

---

[4] DES uses only 21 of the 24 unknown bits, so three bits are still not uniquely determined by matching the expected difference.

Note that f_U is a non-linear function that computes the respective carries of the addition modulo 256, while f_Z is just another expression of Equation 5.2. Comparing the update function of $\mathcal{O}$ in Equation 5.4 with the previously given definition of a T-function, we see that the resemblance is quite obvious not only for updates of 8-bit cells, but also for the entire 128-bit state.

We have studied the resulting equations in more detail, which we will describe now. Given a plaintext $X^{(0)}$ and a ciphertext $X^{(8)}$, the task is to find a possible 128-bit key Y with

$$X^{(8)} = \mathcal{O}\left(Y; X^{(0)}\right).$$

We have generated the equation system describing the LSBs of the ciphertext in relation to the key and plaintext:

$$\begin{aligned}
Y_{9_{\langle 1\rangle}} \oplus f_0\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{0_{\langle 0\rangle}} \\
Y_{10_{\langle 1\rangle}} \oplus f_1\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{1_{\langle 0\rangle}} \\
Y_{11_{\langle 1\rangle}} \oplus f_2\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{2_{\langle 0\rangle}} \\
f_3\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{3_{\langle 0\rangle}} \\
Y_{11_{\langle 1\rangle}} \oplus Y_{13_{\langle 1\rangle}} \oplus f_4\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{4_{\langle 0\rangle}} \\
Y_{14_{\langle 1\rangle}} \oplus f_5\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{5_{\langle 0\rangle}} \\
Y_{15_{\langle 1\rangle}} \oplus f_6\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{6_{\langle 0\rangle}} \\
Y_{0_{\langle 1\rangle}} \oplus f_7\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{7_{\langle 0\rangle}} \\
Y_{0_{\langle 1\rangle}} \oplus Y_{1_{\langle 1\rangle}} \oplus f_8\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{8_{\langle 0\rangle}} \\
Y_{2_{\langle 1\rangle}} \oplus Y_{1_{\langle 1\rangle}} \oplus f_9\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{9_{\langle 0\rangle}} \\
Y_{2_{\langle 1\rangle}} \oplus Y_{3_{\langle 1\rangle}} \oplus f_{10}\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{10_{\langle 0\rangle}} \\
Y_{3_{\langle 1\rangle}} \oplus f_{11}\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{11_{\langle 0\rangle}} \\
g\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) Y_{3_{\langle 1\rangle}} \oplus Y_{5_{\langle 1\rangle}} \oplus Y_{4_{\langle 1\rangle}} \oplus Y_{3_{\langle 2\rangle}} \oplus f_{12}\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{12_{\langle 0\rangle}} \\
Y_{6_{\langle 1\rangle}} \oplus Y_{5_{\langle 1\rangle}} \oplus Y_{3_{\langle 1\rangle}} \oplus f_{13}\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{13_{\langle 0\rangle}} \\
Y_{1_{\langle 1\rangle}} \oplus Y_{6_{\langle 1\rangle}} \oplus f_{14}\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{14_{\langle 0\rangle}} \\
Y_{8_{\langle 1\rangle}} \oplus Y_{7_{\langle 1\rangle}} \oplus f_{15}\left(Y_{0_{\langle 0\rangle}}, \dots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) &= X^{(8)}_{15_{\langle 0\rangle}}
\end{aligned}$$

What we can observe, initially, is that the LSBs of the ciphertext are only dependent on a small subset of key bits (and the plaintext). Apparently only the two LSBs per key byte are involved in the computation. We can obtain the values of the non-linear functions f_i with $0 \le i \le 15$ by guessing one LSB per key byte and setting all remaining bits to 0. This produces a key hypothesis α, which, if used to "encrypt" the known plaintext, yields

$$\beta = \beta_{\langle 0\rangle} || \beta_{\langle 1\rangle} || \dots || \beta_{\langle 128\rangle} = \mathcal{O}\left(\alpha; X^{(0)}\right) \quad \text{with} \quad f_i(\dots) = \beta_{\langle 8i\rangle} \quad \text{for all} \quad 0 \le i \le 15.$$

The value of g can be computed explicitly—based on the guessed key—with the help of this equation:

$$g(\ldots) = X^{(0)}_{3_{\langle 0\rangle}}\alpha_{3_{\langle 0\rangle}} \oplus X^{(0)}_{2_{\langle 0\rangle}}\alpha_{3_{\langle 0\rangle}} \oplus X^{(0)}_{2_{\langle 0\rangle}}X^{(0)}_{3_{\langle 0\rangle}} \oplus X^{(0)}_{11_{\langle 0\rangle}} \oplus X^{(0)}_{3_{\langle 1\rangle}} \oplus X^{(0)}_{2_{\langle 1\rangle}} \oplus X^{(0)}_{1_{\langle 0\rangle}} \oplus \alpha_{4_{\langle 0\rangle}} \oplus \alpha_{2_{\langle 0\rangle}}.$$

With this information, the equation system becomes entirely linear, although it does not have full rank. The upside is that we can exploit some of the obtained equations to filter invalid guesses, e.g., here we know that

$$f_3\left(Y_{0_{\langle 0\rangle}}, \ldots, Y_{15_{\langle 0\rangle}}, X^{(0)}\right) = X^{(8)}_{3_{\langle 0\rangle}}.$$

However, for all four solutions solving the resulting system, we obtain more bits of the key (based on our initial guess). If we use them to generate equations for the next bits of the ciphertext, i.e., $X^{(8)}_{0_{\langle 1\rangle}}, \ldots, X^{(8)}_{15_{\langle 1\rangle}}$, we observe a phenomenon quite similar to what we have seen for the LSBs: the ciphertext bits are only dependent on three bits per key byte and the describing equations are *already linear*. If we combine above observations with the fact that one key bit can easily be derived from plaintext and ciphertext via XORs, i.e.,

$$Y_{4_{\langle 0\rangle}} = X^{(8)}_{3_{\langle 1\rangle}} \oplus \bigoplus_{i=4}^{11}\left(X^{(8)}_{0_{\langle i\rangle}} \oplus X^{(0)}_{0_{\langle i\rangle}}\right) \tag{5.5}$$

we can formulate an attack, which requires us to initially guess only 15 bits of Y.

Denote for a subset $\mathbb{S} \subset \{0, \ldots, 127\}$ by $Y_{\mathbb{S}} \in \{0,1\}^{128}$ the projection of Y to $\mathbb{S}$, i. e.,

$$\left(Y_{\mathbb{S}}\right)_{\langle i\rangle} = \begin{cases} Y_{\langle i\rangle} & \text{if } i \in \mathbb{S}, \\ 0 & \text{if } i \notin \mathbb{S}, \end{cases}$$

and let $\mathcal{O}_i\left(Y; X^{(0)}\right)$ be the i-th LSBs of the ciphertext, i.e.,

$$\mathcal{O}_i\left(Y; X^{(0)}\right) = \left(\mathcal{O}\left(Y; X^{(0)}\right)_{\langle i\rangle}, \mathcal{O}\left(Y; X^{(0)}\right)_{\langle 8+i\rangle}, \ldots, \mathcal{O}\left(Y; X^{(0)}\right)_{\langle 120+i\rangle}\right)_2 \quad \text{for all} \quad 0 \leq i \leq 7$$

for which we find

$$\mathcal{O}_i\left(Y; X^{(0)}\right) = \mathcal{O}_i\left(Y_{\mathbb{S}_i}; X^{(0)}\right) + l_i\left(Y_{\mathbb{S}_{i+1}\setminus\mathbb{S}_i}\right)$$

where l_i are linear mappings of the respective key bits for these sets:

$$\begin{aligned}
\mathbb{S}_0 &= \{0,8,16,24,32,40,48,56,64,72,80,88,96,104,112,120\},\\
\mathbb{S}_1 = \mathbb{S}_0 &\cup \{1,9,17,25,26,33,41,49,57,65,73,81,89,105,113,121\},\\
\mathbb{S}_2 = \mathbb{S}_1 &\cup \{2,10,18,27,34,42,50,58,66,74,82,90,97,106,114,122\},\\
\mathbb{S}_3 = \mathbb{S}_2 &\cup \{3,11,19,28,35,43,51,59,67,75,83,91,98,107,115,123\},\\
\mathbb{S}_4 = \mathbb{S}_3 &\cup \{4,12,20,29,36,44,52,60,68,76,84,92,99,108,116,124\},\\
\mathbb{S}_5 = \mathbb{S}_4 &\cup \{5,13,21,30,37,45,53,61,69,77,85,93,100,109,117,125\},\\
\mathbb{S}_6 = \mathbb{S}_5 &\cup \{6,14,22,31,38,46,54,62,70,78,86,94,101,110,118,126\},\\
\mathbb{S}_7 = \mathbb{S}_6 &\cup \{7,15,23,39,47,55,63,71,79,87,95,102,111,119,127\},\\
\mathbb{S}_8 = \mathbb{S}_7 &\cup \{103\} = \{0,\ldots,127\}.
\end{aligned}$$

Given a plaintext/ciphertext tuple $\left(X^{(0)}, X^{(8)}\right)$, this structure of $\mathcal{O}$ immediately leads to a recursive attack: one first determines $Y_{\langle 32\rangle}$ (cf. Equation 5.5) and guesses the remaining 15 key bits in $\mathbb{S}_0$ and sets up a system of linear equation for the bits in $\mathbb{S}_1$ using $\mathcal{O}_0$. For each solution, one recursively sets up and solves the corresponding linear system for the bits in $\mathbb{S}_i$ using the values of $\mathcal{O}_i$ for $1 \leq i \leq 6$. Finally, $Y_{\langle 103 \rangle}$ can be determined from $\mathcal{O}_7$, where it is the last unknown key bit. Please note that at each step we obtain some equations that are independent of the key bits we are trying to resolve at that stage, which implies the following:

1. The equation systems are underdetermined, so we obtain multiple solutions at each stage, i.e., 4 solutions for $\mathcal{O}_0, \mathcal{O}_6$ and always 8 solutions for $\mathcal{O}_i$ with $1 \leq i \leq 5$.
2. Equations that are only dependent on the known (i.e., guessed or derived) bits can be used to filter initial guesses (i.e., bits in $\mathbb{S}_0$). Especially the equation system for $\mathcal{O}_7$ provides a very effective filter: in the 16 resulting equations, only one key bit $Y_{\langle 103 \rangle}$ is unknown.

Latter fact leads to the property that, if a guess does not contradict any of the equation systems we obtain, it is most likely the correct key Y.

Results

We have used the free SAGE[5] computer algebra system to implement a symbolic version of the obscurity function. This allowed us to generate equations, depending on which bits of the key and input are known and was thus important in understanding the characteristic of the obscurity function. However, the increasing non-linearity from LSB to Most Significant Bit (MSB) in the output bits did not allow to obtain a full equation system, describing each bit of the output as a function of key bit and plaintext variables.

Based on the described analysis, which was facilitated by our symbolic version of $\mathcal{O}$, we have implemented the proposed two-stage attack in the C programming language. The only optimization we have applied was that we have used multi-threading to distribute the search space for the DES decryptions and the inversion of $\mathcal{O}$ across the Intel Xeon E5540 CPU.

From $N = 35$ consecutive runs of the authentication protocol, we have obtained the corresponding set

$$\mathbb{T} := \left\{ \left(P_1^{(0)}, P_2, R^{(0)} \right), \ldots, \left(P_1^{(34)}, P_2, R^{(34)} \right) \right\}.$$

In $\mathbb{T}$ we found one pair of tuples suitable for filtering via XOR differences: for this pair, we found 9 known bits in $\Delta_E V_{\langle 104..127 \rangle}$ (i.e., the part of the DES key we are guessing) and only 12 unknown bits. Additionally, we had 22 known bits in $\Delta_E V_{\langle 0..63 \rangle}$ (i.e., the plaintext part of V) and thus expected to find $|\mathbb{V}| \approx 2048$, for which we had to test 2^{33} hypothesis pairs. We found 1975 potential outputs in $\mathbb{V}$, of which we had to test only the first 5% in order to find K_T after 23.3 minutes of total computation.

5.6.2 Active Attack

Based on the previously presented results, a variant of the attack was developed by Gregor Leander which completely ignores the DES and also exploits all four weaknesses.

For the attack, the targeted transponder's ID has to be known in advance. Additionally, four partial authentications are required. A partial authentication can be initiated by anyone, does not require a valid transponder key and is aborted after the challenge was sent by the door. In this sense, the attack does not require eavesdropping of genuine authentication runs.

The full description of the attack and the trade-offs between available challenges, computation time and number of resulting key candidates can be found in our joint paper [SDK+13].

---

[5] See `http://sagemath.org/`.

5.6.3 Passive Attack

The following attack is designed to work with an absolute minimum of protocol runs and incorporates all four weaknesses listed in the beginning of this section. The attack is basically an extension of the attack on a single instance of $\mathcal{O}$ and works by merging the last instance of $\mathcal{O}$ in $\mathcal{K}$ with the two instances in $\mathcal{R}$. We have observed that even for this merged function the same principles, which allowed us to mount the attack in Section 5.6.1, hold. Subsequently, we denote the iterated application of $\mathcal{O}$ by

$$\begin{aligned} \mathcal{O}^3 : \mathbb{F}_2^{128} \times \mathbb{F}_2^{64} \times \mathbb{F}_2^{128} \times \mathbb{F}_2^{128} &\rightarrow \mathbb{F}_2^{128} \\ \mathcal{O}^3(P_0; T, P_1, P_2) &\rightarrow V, \end{aligned}$$

which is naturally defined as

$$\mathcal{O}^3\left(P_0; T, P_1, P_2\right) = \mathcal{O}\left(\mathcal{O}\left(\mathcal{O}\left(P_0; T || 0^{64}\right); P_1\right); P_2\right) = V.$$

Here, we denote by T the “pre-key”, from which the transponder-key is derived in $\mathcal{K}$. Thus, the goal of this attack is to recover T, from which K_T can be derived via $K_T = \mathcal{O}\left(P_0; T || 0^{64}\right)$ as described in Section 5.5.2.

As for the notation, again, we split the output of $\mathcal{O}^3$ into 8 chunks of 16 bits (with $\mathcal{O}_0^3$ being the LSBs, $\mathcal{O}_1^3$ the next bits per output byte, etc.), i. e.,

$$\mathcal{O}_i^3(P_0; T, P_1, P_2) = \left(\mathcal{O}^3\left(P_0; T, P_1, P_2\right)_{\langle i \rangle}, \mathcal{O}^3\left(P_0; T, P_1, P_2\right)_{\langle 8+i \rangle}, \ldots\right)_2 \quad \text{for all} \quad 0 \leq i \leq 7$$

and denote for a subset $\mathbb{S} \subset \{0, \ldots, 127\}$ by $T_{\mathbb{S}} \in \{0,1\}^{128}$ the projection of T to $\mathbb{S}$, i. e.,

$$\left(T_{\mathbb{S}}\right)_{\langle i \rangle} = \begin{cases} T_{\langle i \rangle} & \text{if } i \in \mathbb{S}, \\ 0 & \text{if } i \notin \mathbb{S}. \end{cases}$$

We have, similarly to the equations we have observed in Section 5.6.1,

$$\mathcal{O}_i^3\left(P_0; T, P_1, P_2\right) = \begin{cases} \mathcal{O}_0^3\left(P_0; T_{\mathbb{S}_0}, P_1, P_2\right) & \text{if } i = 0, \\ \mathcal{O}_i^3\left(P_0; T_{\mathbb{S}_{i-1}}, P_1, P_2\right) + l_i\left(T_{\mathbb{S}_i \setminus \mathbb{S}_{i-1}}\right) & \text{if } 1 \leq i \leq 4, \\ \mathcal{O}_i^3\left(P_0; T_{\mathbb{S}_i}, P_1, P_2\right) & \text{if } 5 \leq i \leq 7, \end{cases}$$

where $l_i\left(T_{\mathbb{S}_i \setminus \mathbb{S}_{i-1}}\right)$ are linear mappings of rank 5 for these sets of key bits:

$$\begin{aligned} \mathbb{S}_0 = & \quad \{0,8,16,24,32,40,48,56\} \\ & \cup \{1,9,17,25,33,41,49,57\} \\ & \cup \{2,10,18,26,34,42,50,58\} \\ & \cup \{3,11,19,27,35,43,51,59\} \\ \mathbb{S}_1 = \mathbb{S}_0 & \cup \{4,12,20,28,36,44,52,60\} \\ \mathbb{S}_2 = \mathbb{S}_1 & \cup \{5,13,21,29,37,45,53,61\} \\ \mathbb{S}_3 = \mathbb{S}_2 & \cup \{6,14,22,30,38,46,54,62\} \\ \mathbb{S}_4 = \mathbb{S}_3 & \cup \{7,15,23,31,39,47,55,63\} \\ \mathbb{S}_5 = \mathbb{S}_6 = \mathbb{S}_7 = \mathbb{S}_4 & \cup \{\} = \{0,\ldots,63\}. \end{aligned}$$

Here, we deliberately choose $\mathbb{S}_0$ to contain 32 key bits, because guessing these actually fixes 25 bits of $\mathscr{O}^3$, which are completely independent of any other key bit. Having this many fixed bits is advantageous, as will be understood in the following description of the actual attack.

It is important to note that we cannot apply the same recursive attack principle (cf. Section 5.6.1) directly, because in order to do this, we would need to be in possession of an actual output V of $\mathscr{O}^3$. This is not the case, however, due to the 40-bit leak, we know some bits of V, which allows to run a meet-in-the-middle attack: if we can observe two consecutive authentication runs at time t and $t+1$, we know a tuple

$$\left(P_0, P_1^{(t)}, P_2, R^{(t)}, V_L^{(t)}\right) \quad \text{with} \quad V_L^{(t)} = P^{(t+1)}_{1_{\langle 16..55\rangle}}.$$

We then try to find a 64-bit value α as pre-key and a full 64-bit DES key β (of which only 56-bits are used) such that

$$\mathscr{O}^3\left(P_0; \alpha, P_1^{(t)}, P_2\right) = \mathscr{D}^{-1}\left(\beta; R^{(t)}\right) || \beta \quad \text{with} \quad \beta_{\langle 0..39\rangle} = V_L^{(t)}. \tag{5.6}$$

In this sense, we literally try to meet with the two (partially) guessed values α and β in the “middle”, which is the output V of the second instance of $\mathscr{O}$ in $\mathscr{R}$. If we find a match, the guessed keys for the two ends of the attacked construction are possibly correct.

We begin as follows: due to $V_L^{(t)}$ we already know 35 bits of the key used for the modified DES in $\mathscr{R}$. We invert $\mathscr{D}$ by decrypting $R^{(t)}$ for all 2^{21} possible keys with the modified DES. For each guess, we obtain 8 different 64-bit key candidates. The 2^{24} results of this operation are stored in a sorted lookup table, each entry consisting of a 128-bit string, being the concatenation of DES plaintext and key (see right hand side of Equation 5.6) and thus a potential output of $\mathscr{O}^3$. The entries in the table are in this way very similar to the set $\mathbb{V}$ which we can obtain via XOR difference filtering (cf. Section 5.6.1), only that here we need only 1.5 authentications.

The remainder of the attack proceeds in a manner quite similar to the first attack. One starts by guessing all 32 bits of T in $\mathbb{S}_0$. This allows to compute $\mathscr{O}_0^3$ and parts of $\mathscr{O}_1^3$ which are then matched against the lookup table to filter wrong candidates. Please note that here we have a very strong filter, i. e., 25 bits for a table with 2^{24} entries. For all remaining candidates one recursively sets up and solves a linear system for the bits of T in $\mathbb{S}_i$ with $1 \leq i \leq 4$. At each step, a partial matching is used to further filter out wrong candidates. After solving equations for key bits in $\mathbb{S}_4$ the entire key is known and thus $\mathscr{O}_5^3, \mathscr{O}_6^3$ and $\mathscr{O}_7^3$ are only used to filter further. The guessed (and derived bits) that pass all filters constitute the correct T, from which K_T can easily be computed with the help of P_0.

Interestingly enough, this attack would even work (in principle) when $V_L^{(t)}$ is no longer leaked, which would be a trivial fix of the authentication protocol. The only difference would be that one would start by guessing the 32 bits of $\mathbb{S}_0$ and build a table based on the resulting, fixed bits.

Results

We have implemented the attack in C and used multi-threading on the Intel Xeon E5540 CPU. We were able to obtain the transponder key after recording one real-world authentication and the succeeding challenge and approx. 15 minutes of computation. The average runtime of the attack is approx. 17 minutes.

5.7 Discussion and Future Work

In this section we shortly compare the presented attacks with regard to their requirements in recorded data and processing time. Furthermore, we shortly elaborate on how the system can be improved in a practical way, i.e., with modifications only on the side of the infrastructure (i.e., backend and locks).

Future work would very likely include an analysis of the authentication between the doors, programming devices and the 868 MHz communication nodes. Attacks on these parts of the system probably offer even more powerful methods to influence the system's behavior, for instance by reprogramming locks with backdoors. Additionally, some ingenious methods for eavesdropping on genuine authentications without a user noticing it could be devised and tested. Of particular interest is finding out, whether it is possible to extract protocol information by using the metal frame of a SimonsVoss equipped door as "antenna" for the near-field of the 25 KHz link between transponder and lock.

5.7.1 Comparison of Attacks

Due to the existence of (at least) three different, very efficient attacks that are practical in a real-world setting, it is obvious that the authentication protocol is severely flawed.

In Table 5.2, we compare the three respective attacks with regard to their requirements. While the active and passive attacks are quite predictable when it comes to required data and processing times, this cannot be said for the differential attack. Here, it solely depends on how predictable to best XOR difference in the set of eavesdropped authentications is. This is influenced by how many authentications are available for filtering: the more data can be recorded, the better the probability for a good filter, the less computation is necessary to recover the whole key. In this sense, the attack offers a time/data trade-off and the given requirements in the table are to be considered as experimentally determined lower bounds. It should be noted that, although the differential attack is typically not very competitive when compared with the other two, it does not rely on the assumption that K_T has only 64 bits of entropy. The differential attack is capable of recovering any 128-bit key, while the other attacks can only find a 128-bit key derived from the 64-bit intermediate value T. This is particularly interesting, because fixing the lack of entropy is one of the more trivial mitigations we are going to suggest in Subsection 5.7.2.

Attack	**A priori knowledge**	**#Auths**	**#Challenges**	**Time**
Differential	*None*	> 2	0	> 1 mins
Active	ID_T	0	4	11.5 secs
Passive	*None*	1	1	17.1 mins

Table 5.2: Comparison of the three attacks on the "Generation 2" authentication scheme

The active and passive attacks are quite comparable; which attack is preferable depends on which scenario is more likely: if it is possible to guess a valid ID_T or obtain it from a transponder by reading it out secretly, the active attack has the advantage because from there on, as little as four interactions with a door only are required and the key can be computed in negligible time. If the ID cannot be guessed or obtained by "borrowing" a transponder, it has to be recorded from a genuine authentication run. In this scenario, the passive attack has the lead, because it requires only one valid authentication (i.e., requires eavesdropping communication between a genuine transponder and a lock) and the following challenge to succeed—although it is quite a bit slower, but still very practical indeed.

5.7.2 Mitigations

Here we are discussing four mitigations, which should *all* be applied, in order to dramatically improve the resistance of the system against the mathematical attacks. The idea of these mitigations is that we want to avoid having to reprogram every transponder that is circulating in an installation. Instead, only doors should need to be reprogrammed, which is apparently even possible via the door's link to the backend.

1. **Randomization of all transponder IDs.**
   The ID of a transponder has 32 bit, and it is currently not clear how they are chosen. Reverse engineering of locks revealed that IDs such as `0x00000005` do exist, while others look more like the result of a random process. It is advisable to choose *all* IDs in the system with the help of a good Random Number Generator (RNG). If IDs cannot be guessed, the active attack becomes less powerful.
2. **Full entropy for transponder keys.**
   Currently, each transponder key is derived from the 64-bit output of the modified DES. This enables the very efficient active and passive attacks. If the key had full entropy, the differential attack would still be possible, but the other two attacks would be thwarted (in the present form).
3. **Decoupling of challenge generation and response computation.**
   By leaking 40 bits of the internal state of $\mathscr{R}$, it becomes possible to very efficiently compute the transponder key. Obtaining challenges in a way that is completely independent from the transponder key eliminates this weakness. However, it should be taken care that the challenges do not become worse in the sense that they improve the predictability of the XOR differences.
4. **Consistency checks in backend**.
   In theory, the "online" version of system 3060 allows to store who opened which door at what time. Therefore, it should be possible to detect protocol anomalies as caused by the active attack (in which the challenges cannot be answered correctly). Since the passive and the differential attack only eavesdrop genuine authentications, this cannot be detected at first. However, when a door is accessed at unusual times (by someone who has managed to make a copy of a transponder, regardless of the method), this could trigger further investigations. It should be noted that this logging feature is often deactivated due to privacy reasons, i.e., in order to avoid creating opportunities to track employees by their use of doors.

Please note that we do not give any recommendation on how these mitigations should be implemented. However, if the construction of the functions is kept in principle, the security of the system will always be upper-bounded by the the costs for breaking DES. If this is desirable, which can be disputed because breaking DES and its 56-bit key is feasible with today's hardware, it is likely that existing building blocks (esp. the DES) can be re-used.

To allow SimonsVoss a reasonable time frame for the implementation of these and other mitigations, we did not disclose the round constants of the obscurity function and also did not reveal how the DES was modified. We did this intentionally, to hinder direct application of our attacks.

CHAPTER 6

CONCLUSION

This document is a testament to an engineer's advances in understanding and applying symmetric cryptanalysis over the course of four projects. While the first project tried to improve existing attacks on A5/1 by actual engineering work (e.g., designing hardware), all subsequent projects focused on more and more novel attacks. It should be stated that all these attacks were enabled by simply being the first to have access to the newly uncovered schemes, which was due to active or passive participation in reverse engineering projects. While, in the case of GMR-1, the ideas for the attacks were already known, they were applied and extended thoroughly and actually executed in practice. In the case of GMR-2, the act of reverse engineering took more work than actually analyzing the cipher. However, since the uncovered design was highly proprietary, the resulting keystream/time trade-off attack is an original contribution. Finally, in the case of the locking system, the focus was entirely on analyzing the construction devised by SimonsVoss. This resulted in two attacks, the first being still a bit unwieldy, while the meet-in-the-middle attack is more elegant—apart from this distinction, both attacks are highly practical.

If there is, in retrospect, an actual hypothesis this document is trying to prove experimentally[1], it can be stated as a variation of an often repeated mantra:

"Security by obscurity is *still* a bad idea."

Our results show that, while overcoming initial hurdles due to obscurity is usually tough, analyzing uncovered security mechanisms usually leads to very satisfactory results—for the attacker. Please note that the cryptanalytic techniques presented here do not require a great degree of sophistication or understanding/application of some underlying theory. However, in an age where cryptography is ubiquitous and well-known primitives of all flavors are publicly available (e.g., due to projects such as eSTREAM[2]), it is an interesting curiosity that such proprietary designs are still in wide circulation.

In the following we will shortly discuss a set of *probable causes* for the situation we have observed in the course of this work. We explicitly stress, that, obviously, we cannot know for sure why a particular design was chosen, so the following is rather speculative:

---

[1] It might be argued that a sample size of $N = 3$ real-world crypto systems is not sufficient or even representative enough to substantiate the stated thesis. However, considering the whole field of real-world security research, there is certainly enough empirical evidence to support it.

[2] See `http://www.ecrypt.eu.org/stream/`.

- **Security as add-on.**
  While it seems to be widely understood that security is a necessity, it is usually not enough in the focus of a product's development cycle. Security is, in this sense, just another requirement—however, one for which no clear engineering approach exists since a design's *effective* security is a function of how many people of a certain expertise were unable to break it in some amount of time.

  An industry that is focused on delivering products (e.g., satphones or locking systems) may rank security behind the actual functionality of a product, but typically even more constraints are imposed: while using the Advanced Encryption Standard (AES) might be a perfect option at times, using it when there is only 256 bytes of FLASH available is simply not possible. Similarly, generating good random numbers on a micro-controller without being able to sample external sources of entropy is equally hard. Also, if the use-case calls for an entirely new paradigm for the design of a cipher, existing standards might not be attractive. The common result of these three examples is the emergence of yet another proprietary construction, possibly with known building blocks such as LFSRs (cf. Section 3.5), the DES (cf. Subsection 5.5.2) or its S-boxes (cf. Section 4.5), born out of a unique set of constraints.
- **Obscurity.**
  The fact that proprietary designs are not available to the cryptographic community for analysis allows potential security weaknesses to remain hidden for quite a while. An effect of this is clearly that manufacturers mostly do not know about security risks (as can be conjectured in the case of SimonsVoss) or do not suffer enough public pressure to improve (as may have been the case for GMR-1 or GMR-2).

  If manufacturers choose to keep details secret, they are likely be successful for a while (depending on incentives for an attacker, for which distribution is a factor) since black-box analysis is often a tedious task. Even worse, the longer obscurity seemingly works (not all breaks are published), the more likely a manufacturer is going to resort to obscurity a second or third time. In this sense (and assuming a growing market penetration over time), longer periods of successful obscurity guarantee a greater degree of devastation once it fails.
- **Duration of deployment.**
  The distribution of global systems is often a function of their deployment time, i.e., the longer the system is in use, the more it spreads out around the globe. Conversely, the more distributed the system is, the more it is exposed to public scrutiny but also, simultaneously, becomes more complex which makes it inherently harder to fix.

  GSM is here another prime example: attacks on the "good" cipher A5/1 are known for over a decade and advances in commercial radio-technology equipment (such as the USRP-2) as well as public computing efforts to compute rainbow tables have put the capability to eavesdrop GSM communication in range of dedicated individuals (as opposed to well-funded intelligence agencies or corporations). Despite all these reasons to abandon GSM, it is not only still in use, but also rapidly conquering developing markets in Africa and Asia.

  Only a switch to a standard that is no longer backward compatible with GSM (i.e., unlike UMTS), and thus is not prone to the same old attacks, can potentially bring an increase in security. Although promising attempts are made by ETSI to publicly evaluate next-generation ciphers [Bab11, ETS10a, ETS10b], a full roll-over to this system is still in the stars.
- **Historic developments.**
  Especially the ciphers used in globally deployed communication standards such as GSM and possibly GMR-1 and GMR-2, were born out of a certain conflict of interest, which is best illustrated by the following:

  In 1982, Europe began to work on a cell phone system. It was the time of the "cold war" and on

> the side of the West, a committee existed to control and limit the export of equipment to the Soviet Union. In line with the general mindset of the US, the Coordinating Committee on Multilateral Export Control (CoCOM) understood that cryptography is a dual-use technology, hence cell phone equipment fell under its regulations. While it was important to have strong cryptography for the domestic market, only weak cryptography should be exported. At that time, 40-bit security was deemed exportable and consequently the A5-type ciphers where invented.
>
> Since A5-GMR-1 is a variant of A5/2, the export version of the "good" A5/1 stream cipher, it can be speculated that similar thoughts influenced the decision to use this cipher to protect over-the-air transmissions in GMR-1.

To put the discussed aspects into perspective: with the advent of "cyber-security" as one of the top concerns of the modern world, it is likely that awareness in the realm of security will increase. Combining this with current trends in cryptographic research, where more and more specialized ciphers are developed and published, it is likely that the average need for proprietary solutions decreases while their usage becomes more robust. In this sense the author hopes that this work can contribute another motivating factor to "get the security right".

BIBLIOGRAPHY

[Bab11] Steve Babbage. The History and Pre-History of ZUC. Technical report, 2011.

[BBK03] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In *International Cryptology Conference (CRYPTO)*. Springer, 2003.

[BBK08] Elad Barkan, Eli Biham, and Nathan Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. *Journal of Cryptology*, 21(3), 2008.

[BD00] Eli Biham and Orr Dunkelman. Cryptanalysis of the A5/1 GSM Stream Cipher. In *International Conference on Cryptology in India (Indocrypt)*, 2000.

[BER07] Andrey Bogdanov, Thomas Eisenbarth, and Andy Rupp. A Hardware-Assisted Realtime Attack on A5/2 Without Precomputations. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[BGS$^+$05] Stephen Bono, Matthew Green, Adam Stubblefield, Ari Juels, Aviel Rubin, and Michael Szydlo. Security analysis of a cryptographically-enabled RFID device. In *USENIX Security Symposium (USENIX)*, 2005.

[BGW99] Marc Briceno, Ian Goldberg, and David Wagner. A pedagogical implementation of the GSM A5/1 and A5/2 “voice privacy” encryption algorithms, 1999.

[BMPP06] Andrey Bogdanov, Marius Mertens, Christof Paar, and Jan Pelzl. A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *International Workshop on Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS)*, 2006.

[BSW00] Alex Biryukov, Adi Shamir, and David Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *International Workshop on Fast Software Encryption (FSE)*, 2000.

[CONQ09] Nicolas Courtois, Sean O Neil, and Jean-Jacques Quisquater. Practical Algebraic Attacks on the Hitag2 Stream Cipher. In *International Conference on Information Security (ISC)*, 2009.

[Cou09] Nicolas Courtois. The Dark Side of Security by Obscurity - and Cloning MiFare Classic Rail and Building Passes, Anywhere, Anytime. In *International Conference on Security and Cryptography (SECRYPT)*, 2009.

[DHW$^+$12] Benedikt Driessen, Ralf Hund, Carsten Willems, Christof Paar, and Thorsten Holz. Dosn't Trust Satellite Phones: A Security Analysis of Two Satphone Standards. In *Symposium on Security and Privacy (Oakland)*, 2012.

[DKS10] Orr Dunkelman, Nathan Keller, and Adi Shamir. A Practical-Time Related-Key Attack on the KASUMI Cryptosystem Used in GSM and 3G Telephony. In *International Crytology Conference (CRYPTO)*, 2010.

[EJ03] Patrik Ekdahl and Thomas Johansson. Another Attack on A5/1. 49(1), 2003.

[ETS01a] ETSI. ETSI TS 101 377-3-10 V1.1.1 (2001-03); GEO-Mobile Radio Interface Specifications; Part 3: Network specifications; Sub-part 9: Security related Network Functions; GMR-2 03.020, 2001.

[ETS01b] ETSI. ETSI TS 101 377-5-3 V1.1.1 (2001-03); GEO-Mobile Radio Interface Specifications; Part 5: Radio interface physical layer specifications; Sub-part 3: Channel Coding; GMR-2 05.003, 2001.

[ETS02] ETSI. ETSI TS 101 376-5-3 V1.2.1 (2002-04); GEO-Mobile Radio Interface Specifications; Part 5: Radio interface physical layer specifications; Sub-part 3: Channel Coding; GMR-1 05.003, 2002.

[ETS10a] ETSI/SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 and 128-EIA3. Document 1: 128-EEA3 and 128-EIA3 Specification. Version 1.4, 2010.

[ETS10b] ETSI/SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 and 128-EIA3. Document 2: ZUC Specification. Version 1.4, 2010.

[GdKGVM12] Flavio Garcia, Gerhard de Koning Gans, Roel Verdult, and Milosch Meriac. Dismantling iClass and iClass Elite. In *European Symposium on Research in Computer Security (ESORICS)*, 2012.

[GGHM05] Nico Galoppo, Naga Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *Conference on Supercomputing (SC)*, 2005.

[Gol97] Jovan Golic. Cryptanalysis of alleged A5 stream cipher. In *International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT)*, 1997.

[GvRVWS09] Flavio Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly Pickpocketing a Mifare Classic Card. In *Symposium on Security and Privacy (Oakland)*, 2009.

[KS02] Alexander Klimov and Adi Shamir. A New Class of Invertible Mappings. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[KS03] Alexander Klimov and Adi Shamir. Cryptographic Applications of T-Functions. In *Conference on Selected Areas in Cryptography (SAC)*, 2003.

[LLMP93] Arjen Lenstra, Hendrik Lenstra, Mark Manasse, and John Pollard. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. 1993.

[LS00] Arjen Lenstra and Adi Shamir. Analysis and optimization of the TWINKLE factoring device. In *International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT)*, 2000.

[LST+09] Stefan Lucks, Andreas Schuler, Erik Tews, Ralf-Philipp Weinmann, and Matthias Wenzel. Attacks on the DECT authentication mechanisms. In *Cryptographer's Track of RSA (CT-RSA)*, 2009.

[Nat77] National Bureau of Standards. Data Encryption Standard. 46, 1977.

[NESP08] Karsten Nohl, David Evans, Starbug, and Henryk Plötz. Reverse-Engineering a Cryptographic RFID Tag. In *USENIX Security Symposium (USENIX)*, 2008.

[NP09] Karsten Nohl and Chris Paget. GSM: SRSLY? In *Chaos Communication Congress*, 2009.

[OM12] Alfredo Ortega and Sebastian Muniz. Satellite baseband mods: Taking control of the InmarSat GMR-2 phone terminal. ekoparty Security Conference, 2012.

[PFS00] Slobodan Petrovic and Amparo Fuster-Sabater. Cryptanalysis of the A5/2 Algorithm. Technical report, 2000.

[PN09] Henrik Plötz and Karsten Nohl. Legic Prime: Obscurity in Depth. In *Chaos Communication Congress*, 2009.

[SDK+13] Daehyun Strobel, Benedikt Driessen, Timo Kasper, David Oswald, Falk Schellenberg, Gregor Leander, and Christof Paar. Fuming Acid and Cryptanalysis: Handy Tools for Overcoming a Digital Locking and Access Control System. In *International Crytology Conference (CRYPTO)*, 2013. To appear.

[Sha99] Adi Shamir. Factoring large numbers with the TWINKLE device. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 1999.

[ST03] Adi Shamir and Eran Tromer. Factoring large numbers with the TWIRL device. In *International Cryptology Conference (CRYPTO)*, 2003.

[VGB12] Roel Verdult, Flavio Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with Hitag2. In *USENIX Security Symposium (USENIX)*, 2012.

[YP04] Thomas Yeung and Eric Poon. Binary decimal numbers and decimal numbers other than base ten. In *International Conference on The Future of Mathematics Education*, 2004.

[Zha11] Mingyi Zhang. Design of a linear equation solver. Master's thesis, EPFL, Switzerland, 2011.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3), 1977.

Curriculum Vitae

Personal

Name	Benedikt Driessen
Born	31.01.1983 in Hamburg, Germany

Education

09/2009–07/2013	**Dr.-Ing.** (PhD), Chair for Embedded Security, Ruhr-Universität Bochum
10/2002–09/2007	**Dipl.-Ing.** (MSc), IT-Security, Ruhr-Universität Bochum

Experience

09/2009–10/2013	**Research Assistant**, Chair for Embedded Security, Ruhr-Universität Bochum
10/2012–12/2012	**Interim Engineering Intern**, QUALCOMM Research, San Diego
10/2007–08/2009	**Security Engineer**, ESCRYPT GmbH, Bochum
01/2007–02/2007	**Intern**, Zynamics GmbH (now Google), Bochum
01/2006–05/2007	**Student trainee**, ESCRYPT GmbH, Bochum

List of Publications

Peer-reviewed Journal Papers

- **Benedikt Driessen**, Ralf Hund, Carsten Willems, Christof Paar, Thorsten Holz (Under minor revision). *An Experimental Security Analysis of Two Satphone Standards*. ACM Transactions on Information and System Security (TISSEC), 2013.

Peer-reviewed Conference Proceedings

- **Benedikt Driessen**, Markus Dürmuth. *Achieving Anonymity Against Major Face Recognition Algorithms*. Conference on Communications and Multimedia Security (CMS), 2013. To appear.
- Daehyun Strobel, **Benedikt Driessen**, Timo Kasper, Gregor Leander, David Oswald, Falk Schellenberg, Christof Paar. *Fuming Acid and Cryptanalysis: Handy Tools for Overcoming a Digital Locking and Access Control System*. Advances in Cryptology (CRYPTO), 2013. To appear.
- **Benedikt Driessen**, Tim Güneysu, Elif Kavun, Oliver Mischke, Christof Paar, Thomas Pöppelmann. *IPSecco: A Lightweight and Reconfigurable IPSec Core*. International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2012.
- **Benedikt Driessen**, Christof Paar. *Solving Binary Linear Equation Systems over the Rationals and Binaries*. International Workshop on the Arithmetic of Finite Fields (WAIFI), 2012.
- **Benedikt Driessen**, Ralf Hund, Carsten Willems, Christof Paar, Thorsten Holz. *Don't Trust Satellite Phones: A Security Analysis of Two Satphone Standards*. IEEE Symposium on Security and Privacy (Oakland), 2012.
- **Benedikt Driessen**, Axel Poschmann, Christof Paar. *Comparison of Innovative Signature Algorithms for WSNs*. Conference on Wireless Network Security (WiSec), 2008.

Other Publications

- **Benedikt Driessen**, Christof Paar. *Angriff auf Thuraya Satellitentelefonie*. Datenschutz und Datensicherheit 12/2012.
- **Benedikt Driessen**, Markus Dürmuth. *Achieving Anonymity Against Major Face Recognition Algorithms*. Cryptology ePrint Archive, Report 2012/878.
- **Benedikt Driessen**. *Eavesdropping on Satellite Telecommunication Systems*. Cryptology ePrint Archive, Report 2012/051.
- Christof Meyer, Juray Somorovsky, Jörg Schwenk, **Benedikt Driessen**, Thang Tran, Christian Wietfeld. *Sec2 – Ein mobiles Nutzer-kontrolliertes Sicherheitskonzept für Cloud-Storage*. D-A-CH Security, 2011.
- Daniel Bussmeyer, **Benedikt Driessen**, André Osterhues, Jan Pelzl, Volker Reiss, Jörg Schwenk, Christof Wegener. *A Generic Architecture and Extension of eCryptfs: Secret Sharing Scheme, Smartcard Integration and a new Linux Security Module*. Linux Kongress, 2009.