Don't Trust Satellite Phones: A Security Analysis of Two Satphone Standards

Benedikt Driessen, Ralf Hund, Carsten Willems, Christof Paar, Thorsten Holz
Horst-Goertz Institute for IT Security
Ruhr-University Bochum, Germany
{benedikt.driessen, ralf.hund, carsten.willems, christof.paar, thorsten.holz}@rub.de

***Abstract*—There is a rich body of work related to the security aspects of cellular mobile phones, in particular with respect to the GSM and UMTS systems. To the best of our knowledge, however, there has been no investigation of the security of satellite phones (abbr. *satphones*). Even though a niche market compared to the G2 and G3 mobile systems, there are several 100,000 satphone subscribers worldwide. Given the sensitive nature of some of their application domains (e.g., natural disaster areas or military campaigns), security plays a particularly important role for satphones.**

In this paper, we analyze the encryption systems used in the two existing (and competing) satphone standards, GMR-1 and GMR-2. The first main contribution is that we were able to completely reverse engineer the encryption algorithms employed. Both ciphers had not been publicly known previously. We describe the details of the recovery of the two algorithms from freely available DSP-firmware updates for satphones, which included the development of a custom disassembler and tools to analyze the code, and extending prior work on binary analysis to efficiently identify cryptographic code. We note that these steps had to be repeated for both systems, because the available binaries were from two entirely different DSP processors. Perhaps somewhat surprisingly, we found that the GMR-1 cipher can be considered a proprietary variant of the GSM A5/2 algorithm, whereas the GMR-2 cipher is an entirely new design. The second main contribution lies in the cryptanalysis of the two proprietary stream ciphers. We were able to adopt known A5/2 ciphertext-only attacks to the GMR-1 algorithm with an average case complexity of 2^{32} steps. With respect to the GMR-2 cipher, we developed a new attack which is powerful in a known-plaintext setting. In this situation, the encryption key for one session, i.e., one phone call, can be recovered with approximately 50–65 bytes of key stream and a moderate computational complexity. A major finding of our work is that the stream ciphers of the two existing satellite phone systems are considerably weaker than what is state-of-the-art in symmetric cryptography.

***Keywords*-Mobile Security; Satellite Phone Systems; Cryptanalysis; Binary Analysis**

I. Introduction

Mobile communication systems have revolutionized the way we interact with each other. Instead of depending on landlines, we can talk to other people wherever we are and also send data from (almost) arbitrary locations. Especially the *Global System for Mobile Communications* (GSM) has attracted quite a lot of attention and with more than four billion subscribers in 2011, it is the most widely deployed standard for cellular networks. Many other cellular network standards like *Universal Mobile Telecommunications System* (UMTS), *CDMA2000* (also known as *IMT Multi-Carrier* (IMT-MC)), or *3GPP Long Term Evolution* (LTE) exist and are continuously enhanced to meet the growing customer demands.

Cellular mobile networks require a so called *cell site* to create a cell within the network. The cell site provides all the necessary equipment for transmitting and receiving radio signals from mobile handsets and the radio network. For example, the cell site contains one or more sets of transmitter/receivers, the necessary antennas, digital signal processors to perform all computations, a GPS receiver for timing, and other control electronics. Within GSM, the cell site is called *Base Transceiver Station* (BTS) and other cellular networks also require this kind of equipment. The cells within a network have only a limited operating distance and thus a certain proximity to a cell site is 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, people living in remote areas or areas that are affected by a natural disaster, or certain military and governmental systems are a few of many uses cases where terrestrial cellular networks are not available. To overcome this limitation, satellite telecommunication systems were introduced that provide telephony and data services based on telecommunications satellites. In such systems, the mobile handset (typically called satellite phone, abbr. *satphone*) communicates directly with satellites in orbit and thus coverage can be provided without the need of an infrastructure on the Earth's surface.

At this point, there are two satphone standards that were both developed in the past few years:

- *Geostationary Earth Orbit (GEO) Mobile Radio Interface* (better known as GMR-1) is a family of ETSI standards that were derived from the terrestrial cellular standard GSM. In fact, the specifications of GMR 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 and the de-facto standard in this area and has undergone several revisions to support a broader range of services.
- The *GMR-2* family is also an ETSI standard that is even closer to GSM. It deviates from the GMR-1 specifications in numerous ways, most notably the network architecture is different.

The specifications of GMR-1 and GMR-2 are available since both are ETSI standards. However, the specifications do not provide any information about implementation details of security aspects. More precisely, it is for example not publicly known which encryption algorithm is actually used to secure the communication channel between a satphone and a satellite. This implies that the security aspects of both standards are poorly documented and proprietary. This is problematic due to the fact that an attacker can easily eavesdrop on the communication channel since the radio signal can be captured with antennas even at some distance to the satphone. At this point, it is thus unclear what effort would be needed by an attacker to actually intercept telephony and data services for common satphone systems.

In this paper, we address this problem and perform a security analysis of the two satphone standards GMR-1 and GMR-2. More specifically, we are interested in the stream ciphers A5-GMR-1 and A5-GMR-2 implemented on the satphones since they are responsible for providing confidential communication channels. To assess the attack surface, we analyzed two popular satellite phones that represent typical handsets:

1) The Thuraya SO-2510 phone implements the GMR-1 standard. It was released in November 2006 and one of the most popular handsets sold by Thuraya.
2) The Inmarsat IsatPhone Pro implements the GMR-2 standard[1] and supports functions such as voice telephony and text/email messaging. It was introduced in June 2010 by Inmarsat.

In principle, satphones implement a hardware architecture similar to typical mobile phones used in terrestrial networks. However, since satphones operate according to different standards, we had to reverse-engineer the phones in detail to understand the algorithms implemented in them. More specifically, we developed our own set of tools to disassemble the machine code and to perform typical binary analysis tasks such as control and data flow analysis. This is challenging since satphones do not use the Intel x86 instruction set architecture but typically a combination of an ARM-based CPU and a digital signal processor (DSP). Recently, several techniques were introduced to detect cryptographic code within a binary in a generic way [1]–[3]. While such techniques can also be leveraged for our analysis, it turned out that these methods have certain limitations in practice since signal processing code and speech encoding algorithms exhibit patterns similar to crypto code. To overcome this problem, we developed our own set of heuristics and program analysis techniques to isolate the cryptographic algorithms of the GMR-1 and GMR-2 standards implemented in the phones. We are the first to publish these algorithms, thus disclosing which crypto algorithms are actually used to secure the communication channel in satphone systems.

Based on our analysis results, we also performed cryptanalysis of both algorithms. We introduce different variants of known and novel attacks and successfully break both algorithms. More specifically, GMR-1 uses a stream-cipher that is a modified version of the A5/2 cipher used in GSM, for which we develop both a known-keystream and a ciphertext-only attack. Furthermore, we propose an attack against a specific GMR-1 channel which has some properties that we can take advantage of. In contrast, GMR-2 uses a proprietary cipher, for which we present a known-plaintext attack whose parameters can be tuned with a time/keystream trade-off. Effectively, we thus demonstrate that current satphone systems are vulnerable to eavesdropping attacks and the results of this paper can be used to build an interceptor for satellite telecommunication systems.

In summary, we make the following contributions:

- We are the first to perform an empirical security analysis of the satellite phone standards GMR-1 and GMR-2, focusing on the encryption algorithms implemented in the handsets. This includes reverse-engineering of firmware images to understand the inner working of the phones, developing our own disassembler and tools to analyze the code, and extending prior work on binary analysis to efficiently identify cryptographic code.
- We perform a formal cryptanalysis of the extracted algorithms and extend a known attack on GSM for GMR-1 and introduce an entirely new attack for GMR-2. Thus we are able to break the encryption in both standards. The attacks can be tuned by a time/ciphertext trade-off for GMR-1 and a time/keystream trade-off for GMR-2. We thus demonstrate that the current satphone standards are vulnerable to eavesdropping attacks.

II. Background and Related Work

We now introduce the necessary background information to understand the basics of satellite telephone systems, their security mechanisms, and the architecture of the mobile handsets. More information about these topics can be found in the literature [4]–[8]. Furthermore, we discuss related work in this area.

A. Satellite Telecommunication Systems

A geostationary orbit telephone network consists of a set of satellites and terrestrial gateway/control stations, as depicted in Figure 1. Gateway stations provide the connectivity to any tethered networks, e.g., telephone calls to

[1]Inmarsat actually refers to their standard as GMR-2+.

a landline are forwarded to the public switched telephone network (PSTN). Satellite operators also run additional control facilities for maintenance and configuration purposes. Both types of transmissions employ conventional wavelength (C-Band) signals. Each satellite serves a specific region, with each region being further subdivided by several spot beams. This mainly allows to transfer multiple signals from different regions on equal frequencies. The system uses long wavelength transmission (L-Band) for spotbeams.

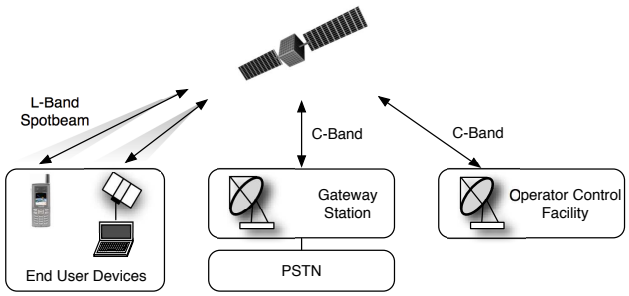

Figure 1. Layout of a geostationary orbit telephone network [8]

In this paper, we focus on the communication channel between end user devices and satellites. This is the only part of the system that is (partially) publicly documented by the GMR specification [6], [9]. The official specification discusses topics relevant to signaling, encoding, and similar aspects of the system. However, no implementation details about the actual cryptographic algorithms that are used between satphones and satellites are disclosed.

The frequency band in GMR systems is divided into different channels and just like in GSM the Time Division Multiple Access (TDMA) time slot architecture is employed which divides each channel into TDMA frames with multiple time slots. Several *logical channels* (called *channels* from now on) are mapped on these time slots. There are different types of channels, but all are either traffic channels (TCH) for voice- or fax-data, or control channels (CCH). Data which is sent over any channel is encoded to add redundancy and protect against transmission failures. For some channels, the encoded data is subsequently encrypted. The encoded (and encrypted) data is finally modulated accordingly before it is transmitted via the phone's antenna. The encoding scheme differs from channel to channel and is dependent on the respective reliability requirements as defined in the various standards.

Figure 2 provides a highly abstract sketch of the authentication and encryption protocol used by GMR-1 and GMR-2. In this protocol, the satellite initiates authentication by sending a request to the phone shortly after the phone has established a connection. This request contains a random number *RAND*, which is sent to the phone's SIM card. On the SIM card, the A3 and A8 algorithm are implemented

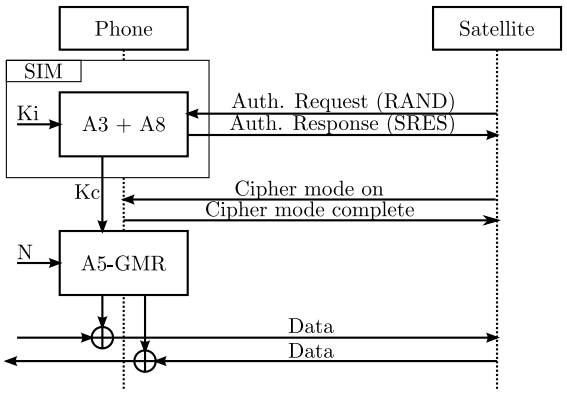

Figure 2. Authentication and encryption in GMR systems (simplified)

that generate a session key *Kc* and an authentication token *SRES*. Both algorithms are not disclosed in the specification and use the SIM card specific key *Ki* as second input. After authentication, the encryption is switched on and all subsequent communication on the relevant channels is encrypted by a stream cipher denoted as A5-GMR, which is also not publicly documented. Due to the limited computing power of the SIM card and the limited bandwidth of the physical connection, the stream cipher is typically implemented on the satphone[2]. The cipher is used to generate a keystream specific for blocks of data, which we will denote as *frames*. In this protocol, the keystream is dependent on the frame's number *N* and *Kc*, which was derived from *RAND*. Due to this architecture, only the stream cipher is responsible for confidentiality which is why we focus on this algorithm in the rest of this paper.

B. Satellite Telephone Architecture

We now briefly elaborate on the general architectural structure of satellite phones and the hardware behind such devices. In a later section, we provide more details on the specific phones we studied during our analysis, including a discussion of the actual processors used in typical satphones.

In general, the architecture of satellite phones is similar to the architecture of cellular phones [10]. Both types of phones have to perform a lot of signal processing due to speech processing and wireless communication, thus they typically ship with a dedicated digital signal processor (DSP) for such purposes. Consequently, complex mathematical operations like for example data compression or speech encoding are outsourced to the DSP where the actual computations are performed. More relevant for our purpose is the fact that DSPs are also suitable for executing cryptographic algorithms, which makes DSP code a prime candidate for holding GMR cipher code.

[2]Page 37 of the specification [9] actually states that the encryption algorithm has to be stored on the SIM card.

The core of the phone is a standard microprocessor (usually an ARM-based CPU) that serves as the central control unit within the system. This CPU initializes the DSP during the boot process. Furthermore, both processors share at least 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 also need to analyze the communication between the two processors.

The operating system running on a phone is typically a specialized embedded operating system that is designed with respects to the special requirements of a phone system (e.g., limited resources, reliability, real-time constraints, etc.). All of the software is deployed as one large, statically linked firmware binary. For our analysis, we were especially interested in the inter-processor communication functionality provided by the operating system as well as the DSP initialization routine. This is due to the fact that cipher code will likely be implemented in the DSP for performance reasons. Our interest for the DSP initialization routine arises from the fact that it typically reveals where DSP code is located in the firmware and how it is mapped to memory.

C. Related Work

Satellite telecommunication systems are related to terrestrial cellular systems, the GMR-1 standard is for example derived from the GSM standard. We can thus leverage work on the analysis of cellular systems for our security analysis as we discuss in the following. Briceno et al. published in 1999 an implementation of the GSM A5/1 and A5/2 algorithms, which they apparently obtained by reverse engineering an actual GSM handset [11]. However, no actual details about the analysis process were ever published and it remains unclear how they actually derived the algorithms. Our analysis is also based on actual satellite phones, we discuss the general approach in Section III and provide analysis details in later sections.

There has been lots of work on the security analysis of the ciphers used within GSM [12]–[20]. A5-GMR-1 is related to the A5/2 algorithm used within GSM, but the configuration of the cipher is different. Our attack for this algorithm builds on the ideas of Petrovic and Fuster-Sabater [13] and Barkan *et. Al.* [18] which we extended to enable a time/ciphertext trade-off.

Up to now, there has been no work on the security aspects of satellite telecommunication systems that we are aware of and we are the first to explore this topic.

III. General Approach

In this section, we outline the general methodology we used for identifying and extracting cipher algorithms from satellite phones. Furthermore, we also discuss the assumptions that helped us during the analysis phase and provide an overview of our target satphones.

We analyzed two representative phones that use the two different standards we are interested in. More precisely, we analyzed the firmwares of the following two phones:

- Thuraya SO-2510 satphone that implements the GMR-1 specification
- Inmarsat IsatPhone Pro satphone that implements the GMR-2 specification

The starting point of our analysis was the publicly available firmware upgrade of each of these two devices. The entire analysis was performed purely statically since we did not have a real satellite phone at our disposal that we could instrument to perform a dynamic analysis. Furthermore, we did not have access to a whole device simulator that enables debugging of arbitrary firmware image, thus we had to develop our own set of analysis tools. However, the ARM code for the main microprocessor (used by both phones) can be partially executed and debugged in a CPU emulator such as QEMU.

The approach we followed to analyze both satphones can be separated into the following five phases:

1) Obtain the firmware installer (usually a Windows setup program).
2) Extract the firmware image from the installer.
3) Reconstruct the correct memory mappings of the code and data sections in the firmware image.
4) Identify the DSP initialization procedure in order to extract the DSP code/mapping.
5) Search for the cipher algorithms in the DSP code using specific heuristics as well as control and data flow analysis techniques.

Several steps can be automated, but some manual analysis is nevertheless required. We successfully applied this method to the two phones we analyzed. In addition, we speculate that also other kinds of satphones can be analyzed in this way. Two assumptions also helped us to find the relevant pieces of code in a shorter amount of time:

1) The key length of the stream cipher algorithms is known.
2) The frame length is equal to the key length.
3) Since the GMR standards are derived from GSM, the ciphers bear at least some resemblance to the well-known, LFSR-based A5 algorithms.

The first two assumptions can be derived from the publicly available parts of the GMR specification [9]. The third assumption was conjectured by us. Note that the standard only specifies the general parameters of the crypto algorithms, but no details about the actual algorithm are publicly available. Nevertheless, these assumptions enabled us to decrease the search space of potential code. The last assumption is rather speculative, but helped us in finding one of the two algorithms.

IV. Security Analysis of GMR-1

We used the Thuraya SO-2510 phone as an example for a handset that operates according to the GMR-1 standard. This decision was solely driven by the fact that the firmware of this satphone is publically available from the vendor's website. In fact, we did not analyze any other GMR-1 satellite phone, but since the protocol is standardized we are confident that our analysis results apply to all other GMR-1 phones as well.

A. Hardware Architecture

The Thuraya SO-2510 runs on a Texas Instruments OMAP 1510 platform. The core of the platform is an ARM CPU along with a TI C55x DSP processor. This information can be deduced from corresponding strings in the binary and from pictures of the actual components soldered on the circuit board [21]. Figure 3 provides a high-level overview of the architecture.

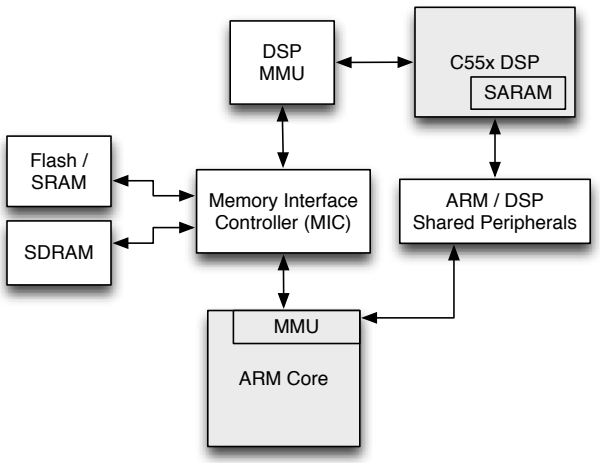

Figure 3. The OMAP1510 Platform [22]

Both processors can communicate with each other using a special shared peripherals bus. Furthermore, they share the same RAM and can access additional memory (e.g., SRAM or Flash) on equal terms. Initially, DSP code or data has to be loaded by the ARM CPU into the specific memory regions of the DSP. The DSP code can be located in either the on-chip SARAM (which holds 96 KB of memory) or in the SRAM, which is accessed through the memory interface controller (MIC). Writes to the SARAM region of the DSP are especially interesting for extracting the corresponding DSP code. The official OMAP1510 documents suggest predefined memory regions to be used by the ARM-MMU for mapping this memory area [22]. During our analysis, we could confirm that the firmware uses exactly the same mappings.

B. Finding the Crypto Code

The firmware of the Thuraya SO-2510 is publically available as a 16 MB sized binary file from the vendor's website. The firmware file is neither (partially) packed nor encrypted and thus the ARM code can be analyzed directly. In the very beginning of the ARM CPU initialization routine, the code sets up the virtual memory system by enabling the MMU with a static translation table. Using this translation table, we deduced the correct memory mapping at runtime. By searching for accesses to the DSP SARAM memory and through string references within the ARM code, we were able to determine the DSP setup code that copies the DSP code from the firmware into the SARAM before resetting the DSP. Briefly speaking, the code is extracted and byte-swapped (due to the differing endianness of both processors) from a number of separate chunks from the firmware image. For convenience, we ran the related ARM code in the QEMU emulator and dumped the resulting DSP code afterwards. This yields approximately 240 KB of DSP code (located in both SARAM and SRAM) that can be readily disassembled by tools such as *IDA Pro*.

Since GMR-1 is derived from GSM, we speculate that the cipher algorithm employed in GMR-1 bears at least some resemblance to the A5/2 cipher from GSM. Due to the nature of this algorithm (e.g., the presence of feedback shift registers), the cipher code is bound to contain a lot of bit shift and XOR operations — unless it is somehow obfuscated. We thus implemented an analysis tool within IDA Pro that counts the occurrences of such instructions in each function and sets them in relation to the total number of instructions in the function. Similar ideas to spot cryptographic primitives have already been published in the literature [1]–[3]. Table I lists the six top-rated functions found when using this heuristic. The four topmost functions are rather short sequences of code that bear striking resemblance to feedback register shift operators; a disassembly is depicted in Figure 12 in the Appendix. Further analyzing the call sites of these four functions revealed an exposed memory region holding variables which equal

- the assumed key length,
- the assumed number and length of the feedback registers, and
- the assumed frame-number lengths (see Section III).

These were all strong indicators that we have spotted the correct region of the code. Starting from this code area, we reverse-engineered the relevant code portions to obtain the cryptographic algorithm employed in the DSP.

C. Structure of the Cipher

The cipher used in GMR-1 is a typical stream-cipher. Its design is a modification of the A5/2 cipher [13], [18], which is used in GSM networks. The cipher uses four linear feedback shift registers (LFSR) which are clocked

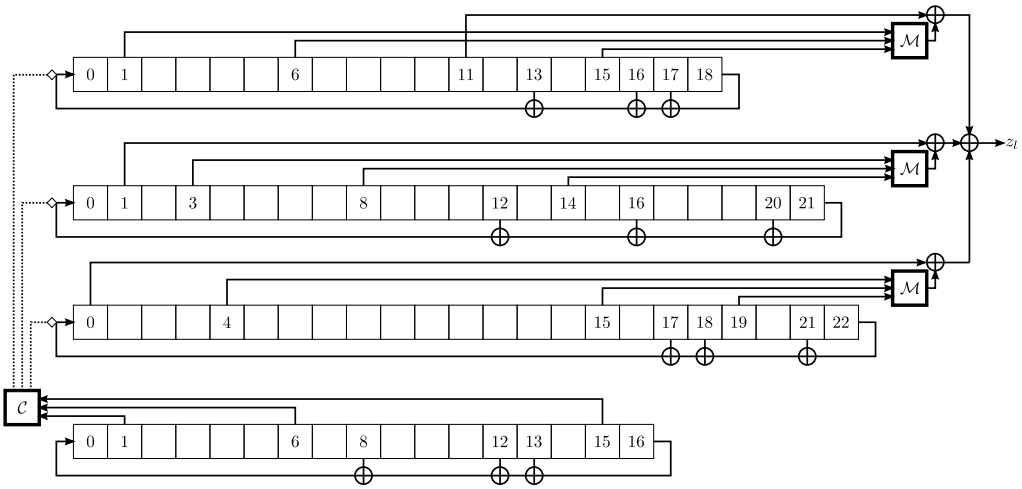

Figure 4. The A5-GMR-1 cipher

Function address	% relevant instr.
0001D038	43%
0001CFC8	41%
0001D000	41%
0001D064	37%
00014C9C	25%
00014CAC	25%

Table I
FUNCTIONS RATED BY PERCENTAGE OF BIT-LEVEL INSTRUCTIONS

	Size	Feedback polynomial	Taps	Final
R_1	19	$x^{19}+x^{18}+x^{17}+x^{14}+1$	1,6,15	11
R_2	22	$x^{22}+x^{21}+x^{17}+x^{13}+1$	3,8,14	1
R_3	23	$x^{23}+x^{22}+x^{19}+x^{18}+1$	4,15,19	0
R_4	17	$x^{17}+x^{14}+x^{13}+x^{9}+1$	1,6,15	-

Table II
CONFIGURATION OF THE LFSRS

irregularly. We call these registers R_1, R_2, R_3 and R_4, see Fig. 4 for a schematic of the structure.

Comparing A5/2 and A5-GMR-1, we see that for most registers the feedback polynomials and also the selection of input taps for the non-linear majority-function $\mathcal{M}$ with

$$\mathcal{M} : \{0,1\}^3 \mapsto \{0,1\}$$
$$x \mapsto x_2x_1 \oplus x_2x_0 \oplus x_0x_1$$

were changed, see Tab. II for details. Also, the positions of the bits that are xor'ed with the respective outputs of the majority functions are different. For curious reasons, all feedback-polynomials have five monomials.

D. 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 $\mathcal{M}(R_{4,1}, R_{4,6}, R_{4,15}) = R_{4,15}$, register R_1 is clocked.
   b) Iff $\mathcal{M}(R_{4,1}, R_{4,6}, R_{4,15}) = R_{4,6}$, register R_2 is clocked.
   c) Iff $\mathcal{M}(R_{4,1}, R_{4,6}, R_{4,15}) = R_{4,1}$, 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.,
$$z_l = \mathcal{M}(R_{1,1}, R_{1,6}, R_{1,15}) \oplus \mathcal{M}(R_{2,3}, R_{2,8}, R_{2,14}) \oplus \mathcal{M}(R_{3,4}, R_{4,15}, R_{3,19}) \oplus R_{1,11} \oplus R_{2,1} \oplus R_{3,0}$$
   is generated.
3) R_4 is clocked.

A5-GMR-1 is operated in two modes, *initialization* and *generation* mode. Running the cipher in 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 value I is computed by xor'ing the 19-bit frame-number N and 64-bit key K in a certain way. However, the specific mapping is not relevant in the remainder.
3) I is clocked into all four registers, i.e., R_1 is clocked and one bit of I is xor'ed with the feedback-bit, R_2 is clocked and xor'ed with the same bit of I, etc. While doing this, no irregular clocking takes place, i.e., the taps of R_4 are not evaluated.
4) The least-significant bits of all four registers are set to 1, i.e., $R_{1,0} = R_{2,0} = R_{3,0} = R_{4,0} = 1$.

After all registers are initialized, irregular clocking is activated and the cipher is clocked for 250 times. The resulting output bits are discarded.

Now the cipher is switched into generation mode and clocked for $2 \cdot m$ times, generating one bit of keystream at a time. We denote the l-th keystream-bit by $z_l^{(N)}$, where $250 \leq l \leq 250 + 2 \cdot m$ is the number of irregular clockings and N the frame-number that was used for initialization.

The number of keystream-bits depends on the type of channel for which data is encrypted or decrypted, see Tab. III. Note that the encoded blocks on channels TCH6, FACCH6, TCH9, and FACCH9 are always multiplexed with ten bits of the SACCH channel. After $2 \cdot m$ bits have been generated, the cipher is re-initialized with the next frame-number.

Channel	n: size of raw data	m: size of encoded data
TCH3/SDCCH	80/84 bits	208 bits
TCH6/FACCH6+SACCH	144/188+10 bits	420+10 bits
TCH9/FACCH9+SACCH	480/300+10 bits	648+10 bits

Table III
PAYLOAD SIZES IN GMR-1 [9]

Depending on the direction bit, either the first or the second half of the keystream is used. Here, we assume that only the first[3] half of each block with only m bits is used, therefore we define z' as the actual keystream with

$$z' = (z_{250}^{(0)}, \ldots, z_{250+m}^{(0)}, z_{250}^{(1)}, \ldots, z_{250+m}^{(1)}, z_{250}^{(2)}, \ldots)_2.$$

E. Cryptanalysis

Since A5-GMR-1 is similar to A5/2, known attacks against that cipher [13], [17], [18] apply. We build on the ideas of Petrovic and Fuster-Sabater [13] and present a known-keystream attack which systematically guesses all possible 2^{16} initial states of R_4. Knowing R_4 completely determines the clocking behavior of the cipher and allows us to express the known keystream as binary quadratic equation system in variables which determine the state of R_1, R_2, and R_3 before warm-up. Taking the fixed bits in these LFSRs as well as symmetries in the quadratic terms into account, the equation system can be linearized by replacing quadratic terms, thus yielding a linear equation system

$$\mathbf{A} \cdot x = z' \tag{1}$$

in v variables with

$$v = \underbrace{\binom{18}{2} + \binom{21}{2} + \binom{22}{2}}_{\text{linearized variables}} + \underbrace{(18 + 21 + 22)}_{\text{original variables}} = 655$$

for each guess of R_4. After obtaining 655 linearly independent equations, each of these systems can be solved, yielding a potential initialization-state for R_1, R_2, and R_3. To test the candidate, the registers are initialized with the guessed value of R_4 and the state candidate, then the cipher is clocked for several times and the resulting keystream is compared with the known keystream. If both keystreams match, we have most likely found the correct initial state. Given the frame-number N that was used to initialize the registers, K can be derived efficiently.

The known-keystream attack can be modified in order to obtain a time/keystream trade-off which enables a ciphertext-only attack. The number of keystream bits required to solve the equation system is directly dependent on the number of unknowns. We can reduce the size of the equation system by guessing parts of R_1, R_2 and R_3 as well. By k_1, k_2, k_3 we denote the number of bits we guess for each of the respective registers, thus the number of variables is reduced to

$$v = \binom{18 - k_1}{2} + \binom{21 - k_2}{2} + \binom{22 - k_3}{2} + (18 - k_1) + (21 - k_2) + (22 - k_3)$$

On the downside, the number of guesses increases and becomes $2^{16} \cdot 2^{k_1 + k_2 + k_3}$ in the worst case. However, this makes it possible to mount a direct ciphertext-only attack on any of the channels (see Tab. II) which exploits the facts that encryption is applied *after* encoding and encoding is linear. This attack was inspired by a work of Barkan *et. al.* in 2003 [23]. Given a block of raw data $d^{(N)}$ of n bits which is transmitted in the N-th frame on one of the relevant channels in GMR-1, encoding is basically[4] a matrix multiplication, i.e.,

$$c'^{(N)} = (d^{(N)} \cdot \mathbf{G}) \quad \text{and} \quad c^{(N)} = c'^{(N)} \oplus z^{(N)}$$

where $\mathbf{G}$ is called the $n \times m$ generator-matrix of the code. By $z^{(N)}$ we denote a *key-frame* (i.e., a block of keystream bits to encrypt one frame of equal size) for one direction, $c'^{(N)}$ is the encoded and $c^{(N)}$ the encoded and encrypted block of m bits. A property of the encoding scheme is that there exists a corresponding parity-check matrix $\mathbf{H}$ of $(m-n) \times m$ size with

$$\mathbf{H} \cdot c'^{(N)} = 0 \quad \text{and therefore} \quad \mathbf{H} \cdot c^{(N)} = \mathbf{H} \cdot z^{(N)}$$

if $c'^{(N)}$ is a valid code-word (and was received without bit-errors). In this case, we can set up an equation system in the variables $z_i^{(N)}$ with $250 \leq i \leq 250 + m$ by computing the syndrome $r = \mathbf{H} \cdot c^{(N)}$, i.e.,

$$\mathbf{H} \cdot z^{(N)} = r. \tag{2}$$

[3]The first m bits are used on the handset's side for decryption, on the satellite side for encryption

[4]After encoding, a pseudo-random sequence is xor'ed with the encoded block "[...] to randomize the number of 0s and 1s in the output bit stream." [24]. This process is called *scrambling* and not to be confused with the subsequent encryption-step, which does basically the same. However, since the parameters of the LFSR used to generate the scrambling-sequence are publicly known, scrambling can be inverted easily which is why we ignore it completely.

This equation system has $(m-n)$ equations and is therefore underdetermined but completely independent from the encoded plaintext. Given that $v \leq (m-n)$ holds due to a proper choice of k_1, k_2, and k_3 and combining Eq. 1 and Eq. 2, we can set up a uniquely solvable equation system

$$\mathbf{H} \cdot (\mathbf{A} \cdot x) = \mathbf{S} \cdot x = r$$

where $\mathbf{A}$ is a $m \times v$ matrix for a particular guess of R_4 (and parts of R_1, R_2, R_3) and $\mathbf{S}$ a $(m-n) \times v$ matrix. Solving the right-hand system results in a potential initialization-state. This can be tested by deriving a key candidate which can then be used to generate the keystream for a different frame-number M and test whether the following relation holds:

$$\mathbf{H} \cdot (c^{(M)} \oplus z^{(M)}) \stackrel{?}{=} 0 \quad \text{with} \quad N-1 > M > N+1.$$

The attack can be greatly accelerated by pre-computing and storing all possible $\mathbf{S}$ matrices in advance. Even better, if $\mathbf{S}$ is quadratic, we can store the LU-decompositions of all possible matrices, which reduces solving the equation systems to a forward- and backward-substitution step. Typically, $\mathbf{S}$ is not quadratic, but can be made quadratic by dropping an appropriate selection of rows while maintaining full rank of the matrix, i.e.,

$$\mathbf{S}' = \mathbf{L} \cdot \mathbf{U} \quad \text{with} \quad \mathbf{S}' = \mathbf{E} \cdot \mathbf{S}$$

where $\mathbf{S}'$ is a $v \times v$ matrix, $\mathbf{U}$ is an upper triangular matrix and $\mathbf{L}$ a lower triangular matrix. For each guess, we need to pre-compute and store the tuple $(\mathbf{L}, \mathbf{U}, \mathbf{E})$ in order to solve $\mathbf{S} \cdot x = r$ in two steps. First, we compute $r' = \mathbf{E} \cdot (\mathbf{H} \cdot c^{(N)})$ which is then used to solve the equation system with the LU-decomposition of $\mathbf{S}'$, i.e., solve

$$\mathbf{L} \cdot r' = y \quad \text{and} \quad \mathbf{U} \cdot y = x.$$

Finally, we propose an attack against the FACCH9 channel which has some properties that we can take advantage of: one encoded and encrypted block $c^{(N)}$ has 658 bits. This attack requires only one of these blocks, which must be received without a single bit-error, otherwise $\mathbf{H} \cdot c^{(N)} \neq \mathbf{H} \cdot z^{(N)}$. The FACCH9 channel is always multiplexed with ten bits from SACCH which can be removed, but then we have to account[5] for this when generating the $\mathbf{A}$ matrices. By choosing $k_1 = 3, k_2 = 6$ and $k_3 = 8$ the number of variables representing the remaining unknown bits of the LFSRs is $v = 345$. Attacking the demultiplexed FACCH9 channel directly by utilizing only one block of encrypted data thus requires solving

$$(2^{16} \cdot 2^{17})/2 = 2^{32}$$

equation systems on average. One of these systems has the form

$$\mathbf{S} \cdot x = r \quad \text{with} \quad \mathbf{S} = \mathbf{H} \cdot \mathbf{A}, \quad r = \mathbf{H} \cdot (\mathbf{D} \cdot c^{(N)})$$

where $\mathbf{D}$ is a 648×658 matrix responsible for demultiplexing FACCH9/SACCH by removing ten bits from c, $\mathbf{A}$ is a 648×345 matrix describing the state of the cipher for 648 of 658 clockings, $\mathbf{H}$ is the 348×648 syndrome matrix for this particular channel and $\mathbf{S}$ a slightly overdefined 348×345 matrix. As stated above, pre-computations and LU-decomposition can be used if enough memory is available.

Evidently, there are several variants of this attack possible on different channels, even more so when multiple frames are used. These attacks have a lower computational complexity but require more ciphertext. To justify our approach, we argue that being able to intercept a single block without errors is a more reasonable assumption than receiving multiple consecutive blocks correctly.

V. Security Analysis of GMR-2

To obtain the code responsible for implementing the cipher according to the GMR-2 standard, we analyzed the latest publically available firmware image of the Inmarsat IsatPhone Pro, which was released in June 2010. Only Inmarsat handsets support the GMR-2 standard at this point and we are confident that our analysis results apply to all of these satphones.

A. Hardware Architecture

The Inmarsat IsatPhone Pro runs on an Analog Devices LeMans AD6900 platform. The core of the platform is an ARM 926EJ-S CPU, which is supplemented by a Blackfin DSP (see Figure 5 for a schematic overview). This architecture can be deduced from plain text strings within the firmware image. We identified an operating system function that returns information on the underlying hardware of the system and this function returns the platform name as a static string.

Both CPUs 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.). The system is initialized by the boot ROM code of the ARM CPU. The ARM CPU then has the task to initialize the DSP for further operations.

B. Finding the Crypto Code

The firmware file is delivered in a special, proprietary format. When loading it into the update program from Inmarsat, the program extracts[6] three files, with each file having a header of 80 bytes. After removing the header, two resulting binary files can be ARM-disassembled by setting

[5]Multiplexing is done by taking 52 encoded bits from FACCH9, concatenating ten bits from SACCH and then appending the remaining 596 encoded bits from FACCH9. We account for this by clocking the cipher ten times between the generation of the 52-nd and 53-rd equation.

[6]This was discovered while disassembling the Dynamic Link Libraries (DLLs) of the update program.

the correct base address, which can be deduced from the respective headers. The third file is seemingly used in a standardized Device Firmware Upgrade (DFU) protocol and we discarded it from the analysis process since it is not relevant for our security analysis.

The two relevant firmware files contain plenty of strings. Based on these strings, we can make the educated guess that there are indeed two separate platforms combining an ARM926EJ-S CPU and a Blackfin DSP, where both processors share the same memory for their code and data. The second file also contains string references related to encryption, which lead us to focus on this part of the whole firmware image. The ARM part of this file can be disassembled using existing tools such as IDA Pro. In contrast, the memory mapping of the DSP code cannot be inferred without additional efforts. However, a correct mapping of the DSP code and data section is required for our analysis since correct references in subroutine calls or string references from within the code are crucial to disassemble and understand the code.

Therefore, we reverse-engineered the very first initialization routines in the Blackfin code, which turned out to hold a DSP memory initialization routine that builds the DSP code and data from the firmware image into another memory region (presumably RAM). In the firmware image, the actual DSP code and data regions are stored as multiple chunks of data that are either uninitialized (i.e., filled with null bytes) or initialized. Initialized blocks are repeated consecutively in memory multiple times. The meta information for each data chunk (i.e., chunk type, block length, etc.) is prepended as a header. The first chunk starts at a fixed address and each header also contains an offset to the next chunk in memory. As no encryption or compression for the DSP code and data is used within the firmware, the corresponding firmware regions can be extracted directly. Using this information, we were able to reconstruct the actual memory layout of the DSP in RAM.

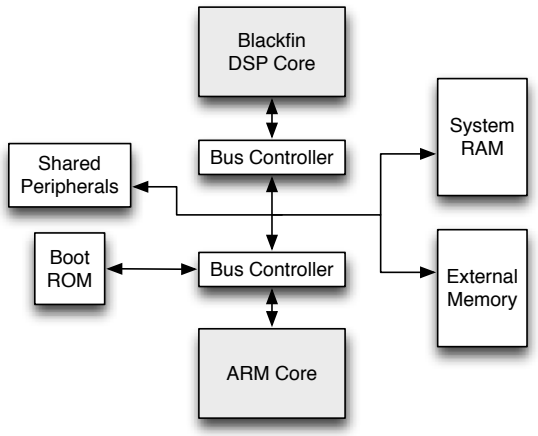

Figure 5. The LeMans AD6900 Platform [25]

As there is no support for Blackfin assembler in IDA Pro, we developed our own disassembler with the help of the official Blackfin reference that disassembles the DSP firmware image by following the program flow using a *recursive traversal* approach:

1) The disassembler performs a linear sweep over the image to obtain target adresses of *call* instructions.
2) In a second step, the disassembler analyzes all adresses identified in the first step and starts to recursively disassemble these locations by following the program flow. Each subsequent *call* and *jump* instruction is taken and *switch* statements are resolved on-the-fly. By following this approach and assuming that no obfuscation is used within the firmware image, we can be sure that only code that is actually reachable by the program flow is interpreted as code.
3) In a third step, all gaps between valid code blocks (i.e., functions) are disassembled recursively to obtain functions that are accessed by indirect means.

Applying our custom disassembler on the reconstructed DSP-image yielded more than 300,000 lines of assembler code, where cross-references are also annotated. An example of a disassembly is shown in the Appendix in Figure 11. Due to the large amount of DSP code, an extensive manual analysis is unfeasible in a reasonable amount of time. Hence, we again applied the same heuristics used in the previous analysis to spot cryptographic code, i.e., we searched for subroutines holding a significant percentage of mathematical operations one would expect from an encryption algorithm [1]–[3]. Unfortunately, this approach did not reveal any code region that could be attributed to keystream generation. This is explained by the fact that the DSP code also contains plenty of demodulation and speech encoding algorithms that naturally bear some resemblance to cryptographic algorithms in that they make extensive use of mathematical operations.

Hence, we decided to follow another approach. The Blackfin code contains a number of debug messages 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 where the debug message contains a reference to a source file with the name `..\..\modem\internal\Gmr2p_modem_ApplyCipher.c`. Hence, we named this function *ApplyCipher()* and we found out that it takes two 120-bit inputs which are then xor'ed. Since we deal with cryptographic code, we assumed that one of these parameters is the output of a stream cipher because the lengths match the expected frame size of 120 bits according to the GMR-2 specification [26]. Starting from this subroutine, we identified the cipher code by applying a number of different techniques

that we explain in the following. All these techniques aim at narrowing down the potentially relevant code base in the disassembly. This is an essential and inevitable step in the analysis process since the keystream generation 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. Figure 6 depicts the reverse call graph of *ApplyCipher()*, where root nodes (shown in grey) constitute thread start routines. Subroutine names, if present, were extracted from debug string within the corresponding subroutine code. Accordingly, in a *forward call graph*, an edge from one node to another indicates that the source node's subroutine calls the destination node's subroutine. The forward call graph is built recursively starting from a given root node.

Our first approach was to manually back track the data flow of the keystream parameter of *ApplyCipher()*. Unfortunately, this did not turn out to be promising since a myriad of additional functions are being called in between the thread creation and *ApplyCipher()*. We were only able to identify a subroutine (denoted by us as *CreateChannelInstance()*) that allocates the memory region of the key data before initializing it with zeros. However, we needed to find the piece of code that fills the keystream buffer with the actual key data. Additional techniques were needed that enable us to exclude those large portions of code that are irrelevant for this purpose.

An analysis of the control flow graphs of the thread start routines (the nine grey nodes in Figure 6) suggests that each routine implements a state machine using one *switch* statement. By generating the forward call graph for each case in the *switch* statement, we can 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 nine 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.

C. Structure of the 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, althtough it shows no resemblance to any of the A5-type ciphers and is called GMR-2-A5 in the respective specification [27].

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 $S = (S_0, S_1, \ldots, S_7)_{2^8}$ and three major components we call $\mathcal{F}, \mathcal{G}$, and $\mathcal{H}$. 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 7 provides a schematic overview of the cipher structure. In the following, we detail the inner workings of each of the three major components.

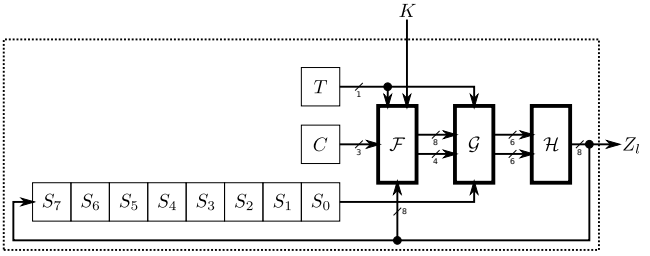

Figure 7. The A5-GMR-2 cipher

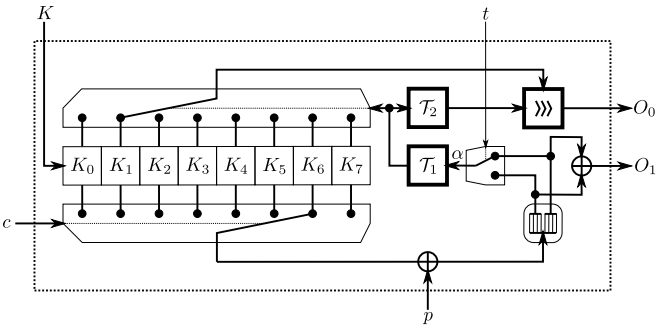

Figure 8. $\mathcal{F}$-component of A5-GMR-2

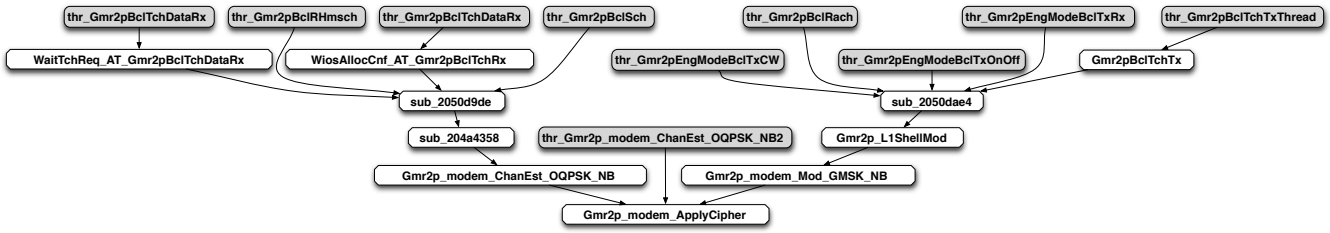

Figure 6. Reverse call graph of *ApplyCipher()* (the ten grey nodes are root nodes)

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 IV
$\mathcal{T}_1$ AND $\mathcal{T}_2$ AS LOOKUP-TABLE

We begin with the $\mathcal{F}$-component, which is certainly the most interesting part of this cipher – Fig. 8 shows its internal structure. On the left we see another 64-bit register split into eight bytes $(K_0, K_1, \ldots, K_7)_{2^8}$. 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 determined by a 4-bit value 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 Tab. IV). Also two bit-wise modulo-2 additions 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

$$\mathcal{N} : \{0,1\} \times \{0,1\}^8 \mapsto \{0,1\}^4$$
$$(t, x) \mapsto \begin{cases} (x_3, x_2, x_1, x_0)_2 & \text{if } t = 0, \\ (x_7, x_6, x_5, x_4)_2 & \text{if } t = 1. \end{cases}$$

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,

$$\begin{aligned} O_0 &= (K_{\mathcal{T}_1(\alpha)} \ggg \mathcal{T}_2(\mathcal{T}_1(\alpha)))_{2^8} \\ O_1 &= (K_{c,7} \oplus p_7 \oplus K_{c,3} \oplus p_3, \\ &\quad K_{c,6} \oplus p_6 \oplus K_{c,2} \oplus p_2, \\ &\quad K_{c,5} \oplus p_5 \oplus K_{c,1} \oplus p_1, \\ &\quad K_{c,4} \oplus p_4 \oplus K_{c,0} \oplus p_0)_2. \end{aligned}$$

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, one byte S_0 of the state is used as input. As can be seen in Figure 9,

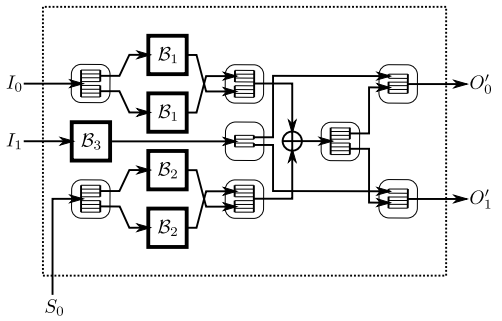

Figure 9. $\mathcal{G}$-component of A5-GMR-2

three sub-components, denoted as $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$, 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.,

$$\begin{aligned} \mathcal{B}_1 &: \{0,1\}^4 \mapsto \{0,1\}^4 \\ &\quad x \mapsto (x_3 \oplus x_0, x_3 \oplus x_2 \oplus x_0, x_3, x_1)_2, \\ \mathcal{B}_2 &: \{0,1\}^4 \mapsto \{0,1\}^4 \\ &\quad x \mapsto (x_1, x_3, x_0, x_2)_2, \\ \mathcal{B}_3 &: \{0,1\}^4 \mapsto \{0,1\}^4 \\ &\quad x \mapsto (x_2, x_0, x_3 \oplus x_1 \oplus x_0, x_3 \oplus x_0)_2. \end{aligned}$$

Since these sub-components and the modulo-2 addition are linear and all other operations on single bits just amount to permutations, the $\mathcal{G}$-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_c, i.e.,

$$\begin{aligned} O'_0 = (&I_{0,7} \oplus I_{0,4} \oplus S_{0,5}, \\ &I_{0,7} \oplus I_{0,6} \oplus I_{0,4} \oplus S_{0,7}, \\ &I_{0,7} \oplus S_{0,4}, \\ &I_{0,5} \oplus S_{0,6}, \\ &I_{1,3} \oplus I_{1,1} \oplus I_{1,0}, \\ &I_{1,3} \oplus I_{1,0})_2, \\ O'_1 = (&I_{0,3} \oplus I_{0,0} \oplus S_{0,1}, \\ &I_{0,3} \oplus I_{0,2} \oplus I_{0,0} \oplus S_{0,3}, \\ &I_{0,3} \oplus S_{0,0}, \\ &I_{0,1} \oplus S_{0,2}, \\ &I_{1,2}, \\ &I_{1,0})_2. \end{aligned}$$

Finally, the $\mathcal{H}$-component gets $I'_0 = O'_0$ and $I'_1 = O'_1$ as input and constitutes the non-linear "filter" of the cipher (see Figure 10). Here, two new sub-components

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

are used and implemented via lookup-tables. Interestingly, these tables were taken from the DES, i.e., $\mathcal{S}_2$ is the second S-box and $\mathcal{S}_6$ 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 $\mathcal{S}_2$ and $\mathcal{S}_6$ 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

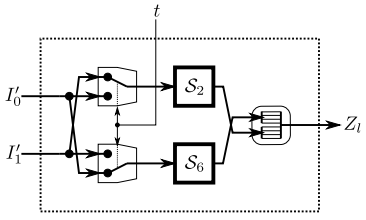

Figure 10. $\mathcal{H}$-component of A5-GMR-2

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

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

D. 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 $\mathcal{G}$-component, the subsequent output Z_l of $\mathcal{H}$ is written back to S_0, i.e., $S_0 := Z_l$. This value is also passed to the $\mathcal{F}$-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 $\mathcal{F}$-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 of this paper.

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 ($l \geq 8$) byte of keystream generated after initialization with frame-number N. In GMR-2, the frame-number is always incremented after 15 byte of keystream, which forces a re-initialization of the cipher. Therefore, the keystream Z' that is actually used for $N \in \{0, 1, 2 \dots\}$ is made up of blocks of 15 bytes (which we call a key-frame with respect to frame-number N) that are concatenated as follows:

$$Z' = (Z_8^{(0)}, \dots, Z_{22}^{(0)}, Z_8^{(1)}, \dots, Z_{22}^{(1)}, Z_8^{(2)}, \dots, Z_{22}^{(2)}, \dots)_{2^8}$$

E. Cryptanalysis

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

1) If $\alpha \in \{(0,0,1,0)_2, (1,0,0,1)_2\}$ then $\mathcal{T}_1(\alpha) = 0$ and $\mathcal{T}_2(\mathcal{T}_1(\alpha)) = 4$, thus $O_0 = (\mathcal{N}(0, K_0), \mathcal{N}(1, K_0))_{2^4}$.
2) If $\alpha \in \{(0,1,1,0)_2, (1,1,1,0)_2\}$ then $\mathcal{T}_1(\alpha) = 4$ and $\mathcal{T}_2(\mathcal{T}_1(\alpha)) = 4$, thus $O_0 = (\mathcal{N}(0, K_4), \mathcal{N}(1, K_4))_{2^4}$.
3) If $\mathcal{T}_1(\alpha) = c$, 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 $i \in \{8, 23, 38, \dots\}$ 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

$$Z'_8 = Z^{(0)}_{16}, Z'_{23} = Z^{(1)}_{16}, Z'_{38} = Z^{(2)}_{16}, \dots$$

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 $\mathcal{F}$-component, the outputs of the $\mathcal{F}$-component are

$$\begin{aligned} O_0 = (&p_3 \oplus \alpha_3, p_2 \oplus \alpha_2, p_1 \oplus \alpha_1, p_0 \oplus \alpha_0, \\ &K_{0,7}, K_{0,6}, K_{0,5}, K_{0,4})_2, \\ O_1 = (&K_{0,7} \oplus p_7 \oplus \alpha_3, K_{0,6} \oplus p_6 \oplus \alpha_2, \\ &K_{0,5} \oplus p_5 \oplus \alpha_1, K_{0,4} \oplus p_4 \oplus \alpha_0)_2, \end{aligned}$$

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

$$\begin{aligned} O'_0 = (&p_3 \oplus \alpha_3 \oplus p_0 \oplus \alpha_0 \oplus S_{0,5}, \\ &p_3 \oplus \alpha_3 \oplus p_2 \oplus \alpha_2 \oplus p_0 \oplus \alpha_0 \oplus S_{0,7}, \\ &p_3 \oplus \alpha_3 \oplus S_{0,4}, \\ &p_1 \oplus \alpha_1 \oplus S_{0,6}, \\ &K_{0,7} \oplus p_7 \oplus \alpha_3 \oplus K_{0,5} \oplus p_5 \oplus \alpha_1 \oplus K_{0,4} \oplus p_4 \oplus \alpha_0, \\ &K_{0,7} \oplus p_7 \oplus \alpha_3 \oplus K_{0,4} \oplus p_4 \oplus \alpha_0)_2, \\ O'_1 = (&K_{0,7} \oplus K_{0,4} \oplus S_{0,1}, \\ &K_{0,7} \oplus K_{0,6} \oplus K_{0,4} \oplus S_{0,3}, \\ &K_{0,7} \oplus S_{0,0} \\ &K_{0,5} \oplus S_{0,2}, \\ &K_{0,6} \oplus p_6 \oplus \alpha_2, \\ &K_{0,4} \oplus p_4 \oplus \alpha_0)_2. \end{aligned}$$

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

$$Z'_i = (\mathcal{S}_2(O'_1), \mathcal{S}_6(O'_0))_{2^4}$$

holds.

In order to determine K_0, we examine the inputs and outputs of $\mathcal{S}_6$ and $\mathcal{S}_2$ in the $\mathcal{H}$-component, starting with $\mathcal{S}_6$. Due to the reordering of the DES S-boxes, the column of $\mathcal{S}_6$ 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

$$S_0 \stackrel{!}{=} Z'_{i-8} \quad \text{and} \quad p \stackrel{!}{=} Z'_{i-1}$$

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 $(\beta_1, \beta_0)_2$, which implies

$$\begin{aligned} K_{0,7} \oplus K_{0,5} \oplus K_{0,4} &= \beta_1 \oplus p_7 \oplus \alpha_3 \oplus p_5 \oplus \alpha_1 \oplus p_4 \oplus \alpha_0, \\ K_{0,7} \oplus K_{0,4} &= \beta_0 \oplus p_7 \oplus \alpha_3 \oplus p_4 \oplus \alpha_0. \end{aligned}$$

Here we gain "some" information about the bits of K_0, $K_{0,5}$ can even be computed. We can then use the output of $\mathcal{S}_2$ 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

$$\mathcal{S}_2(\gamma) = (Z'_{i,7}, Z'_{i,6}, Z'_{i,5}, Z'_{i,4})_2.$$

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

$$\begin{aligned} \gamma_5 &\stackrel{?}{=} \beta_0 \oplus p_7 \oplus \alpha_3 \oplus p_4 \oplus \alpha_0 \oplus S_{0,1}, \\ \gamma_4 \oplus \gamma_1 &\stackrel{?}{=} \beta_0 \oplus p_7 \oplus \alpha_3 \oplus p_4 \oplus \alpha_0 \oplus S_{0,3} \oplus p_6 \oplus \alpha_2, \\ \gamma_3 \oplus \gamma_0 &\stackrel{?}{=} \beta_0 \oplus p_7 \oplus \alpha_3 \oplus S_{0,0}, \\ \gamma_2 \oplus \gamma_5 &\stackrel{?}{=} \beta_1 \oplus p_7 \oplus \alpha_3 \oplus p_5 \oplus \alpha_1 \oplus p_4 \oplus \alpha_0 \oplus S_{0,1} \oplus S_{0,2}. \end{aligned}$$

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

$$\begin{aligned} (&\gamma_3 \oplus S_{0,0}, \gamma_1 \oplus p_6 \oplus \alpha_2, \gamma_2 \oplus S_{0,2}, \gamma_0 \oplus p_4 \oplus \alpha_0, \\ &p_3 \oplus \alpha_3, p_2 \oplus \alpha_2, p_1 \oplus \alpha_1, p_0 \oplus \alpha_0)_2. \end{aligned}$$

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 $\alpha \in \{(0,1,1,0)_2, (1,1,1,0)_2\}$, $i \in \{12, 27, 42, \dots\}$ 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 theses 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 $K_1, ..., K_7$ we examine a few keystream-bytes for a second time, while focusing on the $\mathcal{F}$-component. For each K_j with $j \in \{0, 1, \dots, 7\}$ for which we already have a hypothesis, we can use the corresponding key-stream bytes $(Z'_{i+j}, Z'_{i+j-1}, Z'_{i+j-8})_{2^8}$ with $i \in \{8, 23, 38, \dots\}$ to compute

$$\alpha = \mathcal{N}(j \bmod 2, K_j \oplus Z'_{i+j-1}).$$

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 $\delta \in \{0, 1, \dots, 255\}$ 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.

VI. Conclusion and Implications

Despite a large body of work related to the security aspects of the GSM, and to a lesser extend, the UMTS system, there had been no scientific or other publicly available investigation of the security mechanisms employed by the two existing satphone standards, GMR-1 and GMR-2. Contrary to the practice recommended in modern security engineering, both standards rely on proprietary algorithms for (voice) encryption. Even though it is impossible for outsiders (like us) to decide whether this is due to historic developments or because secret algorithms were believed to provide a higher level of "security", the findings of our work are not encouraging from a security point of view. GMR-1 relies on a variant of the GSM cipher A5/2, for which serious weakness have been demonstrated for more than a decade. The GMR-2 cipher, which appears to be an entirely new stream cipher, shows even more serious cryptographic weaknesses. In the case of GMR-1, an attacker can mount a successful ciphertext-only attack. With respect to the GMR-2 cipher, in a known-plaintext setting where approximately 50–65 bytes plaintext are known to the attacker, it is possible to recover a session key with a moderate computational complexity, allowing the attack to be easily performed with a current PC. Both algorithms are cryptographically dramatically weaker than would be possible with state-of-the-art ciphers, e.g., AES. Even if AES is considered too "heavy" for realtime encryption on mobile phones, the Phase 2 eSTREAM finalist stream ciphers [28] or lightweight block ciphers such as PRESENT [29], provide considerably better cryptographic protection.

The cipher code inside the firmware was not specifically protected against reverse-engineering efforts. The difficulty in reconstructing both algorithms thus stems from the inherent complexity in analyzing large pieces of code. If software engineers had employed state-of-the art obfuscation schemes, the analysis could have been at least complicated significantly. Furthermore, implementing the ciphers in hardware would also hamper reverse-engineering.

In this paper, we do not address the issue of obtaining ciphertext or plaintext data. However, follow-up work on GMR-1 has done this, executing the attack on GMR-1 and revealing the session key with moderate effort [30]. This clearly demonstrates the impact and validity of our analysis.

Acknowledgment

We would like to thank Sylvain Munaut from the OsmocomGMR project [31] for verifying the reconstructed A5-GMR-1 cipher with real-world data.

This work has been supported by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant IV.5-43-02/2-005-WFBO-009).

References

[1] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "ReFormat: Automatic Reverse Engineering of Encrypted Messages," in *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[2] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[3] F. Gröbert, C. Willems, and T. Holz, "Automated Identification of Cryptographic Primitives in Binary Programs," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[4] D. Wright, "Reaching out to remote and rural areas: Mobile satellite services and the role of Inmarsat," *Telecommunications Policy*, vol. 19, no. 2, pp. 105 – 116, 1995.

[5] D. Matolak, A. Noerpel, R. Goodings, D. Staay, and J. Baldasano, "Recent progress in deployment and standardization of geostationary mobile satellite systems," in *Military Communications Conference (MILCOM)*, 2002.

[6] ETSI, *ETSI TS 101 376-3-2 V1.1.1 (2001-03); GEO-Mobile Radio Interface Specifications; Part 3: Network specifications; Sub-part 2: Network Architecture; GMR-1 03.002*, Std., 2001.

[7] G. Maral and M. Bousquet, *Satellite Communications Systems: Systems, Techniques and Technology*, 5th ed. John Wiley & Sons, 2009.

[8] Jim Geovedi and Raoul Chiesa, "Hacking a Bird in the Sky," in *HITBSecConf, Amsterdam*, 2011.

[9] ETSI, *ETSI TS 101 376-3-9 V1.1.1 (2001-03); GEO-Mobile Radio Interface Specifications; Part 3: Network specifications; Sub-part 9: Security related Network Functions; GMR-1 03.020*, Std., 2001.

[10] H. Welte, "Anatomy of contemporary GSM cellphone hardware," 2010. [Online]. Available: http://laforge.gnumonks.org/papers/gsm_phone-anatomy-latest.pdf

[11] M. Briceno, I. Goldberg, and D. Wagner, "A pedagogical implementation of the GSM A5/1 and A5/2 "voice privacy" encryption algorithms," 1999, originally published at http://www.scard.org, mirror at http://cryptome.org/gsm-a512.htm.

[12] J. D. Golic, "Cryptanalysis of alleged A5 stream cipher," in *Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, ser. EUROCRYPT'97. Springer-Verlag, 1997, pp. 239–255.

[13] S. Petrovic and A. Fuster-Sabater, "Cryptanalysis of the A5/2 Algorithm," Cryptology ePrint Archive, Report 2000/052, Tech. Rep., 2000, http://eprint.iacr.org/.

[14] E. Biham and O. Dunkelman, "Cryptanalysis of the A5/1 GSM Stream Cipher," in *Indocrypt*, 2000.

[15] A. Biryukov, A. Shamir, and D. Wagner, "Real Time Cryptanalysis of A5/1 on a PC," in *Fast Software Encryption (FSE)*, 2000.

[16] P. Ekdahl and T. Johansson, "Another Attack on A5/1," *IEEE Transactions on Information Theory*, vol. 49, no. 1, 2003.

[17] A. Bogdanov, T. Eisenbarth, and A. Rupp, "A Hardware-Assisted Realtime Attack on A5/2 Without Precomputations," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[18] E. Barkan, E. Biham, and N. Keller, "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication," *Journal of Cryptology*, vol. 21, March 2008.

[19] K. Nohl and C. Paget, "GSM: SRSLY?" 2009, 26th Chaos Communication Congress.

[20] O. Dunkelman, N. Keller, and A. Shamir, "A Practical-Time Related-Key Attack on the KASUMI Cryptosystem Used in GSM and 3G Telephony," in *International Crytology Conference (CRYPTO)*, 2010.

[21] OsmocomGMR. Thuraya SO-2510. [Online]. Available: http://gmr.osmocom.org/trac/wiki/Thuraya_SO2510

[22] Texas Instruments. The OMAP 5910 Platform. [Online]. Available: http://www.ti.com/product/omap5910

[23] E. Barkan, E. Biham, and N. Keller, "Instant Ciphertext-Only Cryptanalysis of GSM encrypted communication," in *International Crytology Conference (CRYPTO)*, 2003, pp. 600–616.

[24] 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*, Std., 2002.

[25] Jose Fridman, Analog Devices. How to optimize H.264 video decode on a digital baseband processor. [Online]. Available: http://www.eetimes.com/General/DisplayPrintViewContent?contentItemId=4016202

[26] 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*, Std., 2001.

[27] ——, *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*, Std., 2001.

[28] M. Robshaw and O. Billet, Eds., *New Stream Cipher Designs: The eSTREAM Finalists*, ser. LNCS. Springer, 2008, vol. 4986.

[29] A. Bogdanov, G. Leander, L. R. Knudsen, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT—An Ultra-Lightweight Block Cipher," in *CHES '07: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. LNCS, no. 4727. Springer, 2007, pp. 450–466.

[30] B. Driessen, "Eavesdropping on Satellite Telecommunication Systems," Cryptology ePrint Archive, Report 2012/051, 2012, http://eprint.iacr.org/2012/051.

[31] S. Munaut. (2012, Jan.) OsmocomGMR. [Online]. Available: http://gmr.osmocom.org/

APPENDIX

A. Disassembly of Blackfin Code

```
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=hex:0x00000001 */
 20000012 R2.L = 0x7410;    /* R2=0x00007410 */
 20000016 R2.H = 0x2054;    /* @R2=str:'pDecTemp0' */
 2000001a SP += -0x24;
 2000001c R0 = 0x4a8 (X);   /* R0=0x000004a8 */
 20000020 R1 = B[P0] (Z);
 20000022 [SP + 0xc] = P0;
 20000024 P1.L = 0x6010;    /* P1=0x00006010 */
 20000028 P1.H = 0xc000;    /* P1=0xc0006010 */
 2000002c CALL (P1);
 2000002e P3 = R0;
 20000030 CC = R0 == 0x0;
 20000032 R0 = 0x1 (X);
 20000034 IF CC JUMP 0x2000015c;
 ...
loc_2000015c: x-refs 20000034
 2000015c SP += 0x24;
 2000015e (R7:6, P5:3) = [SP++];
 20000160 UNLINK;
 20000164 RTS;
```

Figure 11. Example for disassembly of Blackfin code

Figure 11 shows an excerpt from the output of our Blackfin disassembler. As can be seen, cross-references (i.e., who calls a function or where a jump originates) are annotated. Additionally, if a register is loaded with an address which points to a memory location which is within the firmware image, the disassembler also reads and interprets the respective data.

B. DSP Code of Feedback Register Shift Subroutine

```
ROM:1D038 update_reg3:
ROM:1D038   mov     dbl(*abs16(#reg3)), AC1
ROM:1D03D   sftl    AC1, #-1, AC2
ROM:1D040   mov     dbl(*abs16(#reg3)), AC1
ROM:1D045   xor     AC2, AC1
ROM:1D047   bfxtr   #0FFF0h, AC1, AR1
ROM:1D04B   and     #1, AR1, AC3
ROM:1D04E   and     #1, AC1, AC1
ROM:1D051   xor     AC3, AC1
ROM:1D053   xor     AC0, AC1
ROM:1D055   sftl    AC1, #22, AC0
ROM:1D058   xor     AC2, AC0
ROM:1D05A   mov     AC0, dbl(*abs16(#reg3))
ROM:1D05F   ret
```

Figure 12. DSP code of a feedback register shift subroutine

Figure 12 depicts the TMS320C55x DSP code of one of the feedback register shift subroutines. Note that the subroutine and variable names were inserted by us afterwards and are not present in the binary.