Implementation and performance analysis of Barkan, Biham and Keller's attack on A5/2

Nicolas Paglieri, Olivier Benjamin
Ensimag, Grenoble Institute of Technology, INP

June 8, 2011

Abstract

Barkan, Biham & Keller proposed an instant ciphertext-only attack on the A5/2 cipher used in GSM. This paper presents a concrete and turnkey implementation of a tool simulating this attack on a personal computer. This program is able to recover the secret key that has been used in encryption in a matter of seconds, with precomputations of around 2 hours necessiting 4 GB of memory space.
Keywords: *GSM, A5/2, attack, implementation*

1 Introduction

It's an open secret that the privacy in GSM – *Global System for Mobile communication*, the standard used in 80% of the world's mobile phones – is compromised. The A5 set of ciphers it uses has been proven incapable of preventing data from being decrypted. The architecture of the standard makes it particularily vulnerable to man-in-the-middle attacks, and since all algorithms share a common key, the system is as weak as its weakest algorithm: A5/2. Thus, this article focuses on it[1] because breaking it is equivalent to breaking the entire encryption scheme, and this attack is both very quick and effective. This paper is organized as follows: Section 2 provides some short technical background on GSM; Section 3 explains how GSM encoding/decoding was simulated; Section 4 presents the attack in terms of implementation and performances; Section 5 describes our implementation's general purpose; Finally, Section 6 presents some countermeasures that may be considered.

[1]Attacks on the other ciphers exist [1] [2]

2 Technical Background

GSM is a standard set developed by the ETSI (*European Telecommunications Standards Institute*) to describe technologies for second generation digital cellular networks. Its first specifications were published in 1990, and included some incipient kind of consideration for security threats. This section only expounds technical aspects of GSM that are relevant to the elements presented in this paper.

2.1 Error Correcting Codes

The GSM standard defines numerous communication channels, each dedicated to a particular use. Every channel has its own coding and interleaving scheme. Here we focus on the SACCH – *Slow Associated Control CHannel*, used for other channels' monitoring – on which the attack will be based.
In real GSM systems, data is transmitted over radio waves, so they are more error-prone than most communication channels, implying the need of error correcting codes. In GSM communications, three successive error correction schemes are used. A short description of these schemes can be found in [3] but the explaination proposed in this paper is more explicit. The first one is a Fire code, which is a cyclic code, and the one used in GSM is characterized by the polynomial generator $g(X) = (X^{23} + 1)(X^{17} + X^3 + 1)$. It is used to encode a 184-bit vector into a 224-bit one containing the original vector, to which is appended a 40-bit cyclic redundancy check –CRC–. Four bits equal to zero are then simply added to the tail of this vector to form a 228-bit one. The second correction code used is a $\frac{1}{2}$-rate convolutional code whose

generators are $\begin{cases} G_0(X) &= 1 + X^3 + X^4 \\ G_1(X) &= 1 + X + X^3 + X^4 \end{cases}$.
In other words, a vector $(u_0, u_1, ..., u_{227})$ would be changed into the vector $(v_0, v_1, ..., v_{455})$ such that

$$\begin{cases} \forall k \in [\![0, 227]\!],\ v_{2k} &= v_k \oplus v_{k-3} \oplus v_{k-4} \\ v_{2k+1} &= v_k \oplus v_{k-1} \oplus v_{k-3} \oplus v_{k-4} \\ \forall\ k < 0\ u_k &= 0 \end{cases}.$$

Then comes the third correction scheme: the interleaving. Its aim is to reorder the bits of the vector, so that two bits that were originally close – and hence contained redundant information about one another – are then far apart. It is done by constructing the table page 98 in [3], that we will call T here. This lookup table is built so that $\forall\ k \in [\![0, 456]\!],\ T(i, j) = u_k$, where $i = k\ mod\ 8$ and $j = 2((49 \times k)\ mod\ 57) + ((k\ mod\ 8)\ div\ 4)$. After the table is created, the new vector is constructed. To do so, the table is separated into two subtables: one for the even-numbered bits – the first 4 columns –, and one for the odd-numbered ones – the last 4. Both of them are then read alternatively, top to bottom, left colum to right colum, and the corresponding value of u_k is placed into the current bit of the vector. The resulting vector is of the form:
$(v_0, v_1, v_2, v_3, ..., v_{114}, v_{115}, v_{116}, v_{117}, ..., v_{227}, v_{228}, v_{229}, v_{230}, ..., v_{340}, v_{341}, v_{342}, v_{343}, ..., v_{455})$ $= (u_0, u_{228}, u_{64}, u_{292}, ..., u_{57}, u_{285}, u_{121}, u_{349}, ..., u_{114}, u_{342}, u_{178}, u_{406}, ..., u_{171}, u_{399}, u_{235}, u_7, .., u_{335})$
If an error should happen over a few bits, for instance bits v_{114} and v_{115} are flipped, it has no impact, since they were not initialized by elements that were in no way related.

```
The test vector (hex representation, MSB first):
5E89B2D7F5C04D54D54D64D96A17AD54DC55D62AABA416
would be fully encoded into:
52027BF6524C84804F41F32D3418A063F3C826FCFE40C9
A37AF076007101896CC0F76000DA356ED8A836F6CAA2F7
BC0B6028A2DE8B013764AB
```

2.2 A5/2 cipher

A5/2 is a stream cipher using 4 different LFSRs (*Linear Feedback Shift Registers*). It is a weaker version of A5/1 (much stronger, but also already cracked [1]), and was developped in 1999 to conform to cryptographic export policies outside the European Union (strong cryptography was seriously restricted at this time).
The LFSRs are initialized from a 64-bit secret

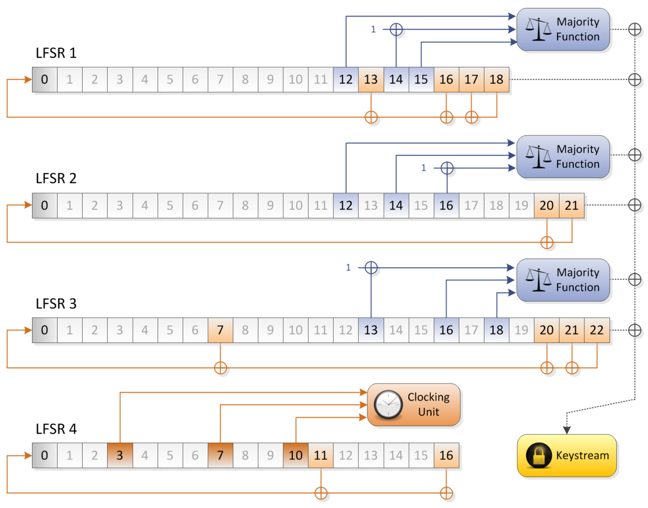


Figure 1: A5/2 cipher with its 4 LFSRs

key K_c and a public 22-bit initial value f (GSM Frame Identifier) according to a procedure called Key Setup:

```
LFSR1 = LFSR2 = LFSR3 = LFSR4 = 0
for (int i=0 ; i<63 ; ++i) {
  clockAllRegisters();
  LFSR1[i] ^= Kc[i]; LFSR2[i] ^= Kc[i];
  LFSR3[i] ^= Kc[i]; LFSR4[i] ^= Kc[i];
}
for (int i=0 ; i<22 ; ++i) {
  clockAllRegisters();
  LFSR1[i] ^= f[i]; LFSR2[i] ^= f[i];
  LFSR3[i] ^= f[i]; LFSR4[i] ^= f[i];
}
LFSR1[15] = LFSR2[16] = LFSR3[18] = LFSR4[10] = 1;
```

Majority functions return 1 if at least two entry bits are non-zero. The clocking unit (only used after Key Setup phase) also includes a majority function: at each cycle, LFSR1 (resp. LFSR2, LFSR3) is clocked if LFSR4[10] (resp. LFSR4[3], LFSR4[7]) is the same as the majority bit. Thus, a new bit of keystream is produced at each cycle. The first 99 bits of generated keystream are simply discarded. Then the following chunks of 228 bits are used to encrypt messages: the first half of 114 bits is dedicated to network-to-phone encryption, and the next 114 bits are dedicated to phone-to-network encryption. The encryption itself consists of a bitwise XOR of the message (divided into 114-bit frames) with the 114-bit related keystream.

3 Simulation of GSM transmission

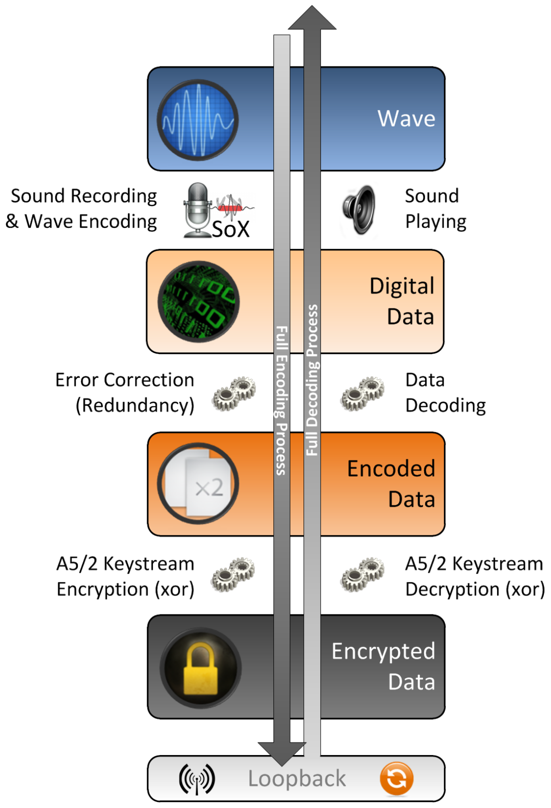


Figure 2: Full Simulator Process

3.1 From Wave to Digital Data

GSM is usually used for voice communication. Even if the aim of the channel considered here isn't actually to carry voice signal, it is more spectacular to prove that one can eavesdrop on calls, so the first step is to convert sound waves into GSM-encoded sound files. One easy way to proceed is to use a microphone to record the sound as a wave file, then to use SoX (Sound eXchange[4]) libraries to correctly compress the data for GSM use. The digital data must then be split into 114-bits packets to simulate SACCH frames ; the rest of the chain is applied to each packet separately. The packets are finally recombined only during the last transformation (from digital data to wave, when playing the sound).

3.2 Error-correcting codes

Given that every error-correcting code described in section 2.1 is linear, they can all be expressed as transformation matrices. We provide all the undermentioned matrices of this section in a separate file `const_matrices.h`, available on our website [5].

Fire Code

The Fire code is a CRC, so it is easy to construct the corresponding 184×224 binary matrix: each line of the matrix is the polynomial (see section 2.1) in reversed binary representation (10010000000000000100000100100000000000001) that is shifted 1 bit to the right at each new line, starting on the upper left corner. Then a Gauss Elimination is performed to get the matrix in systematic form: $G'_{Fire} = \left(\; Id_{184} \; \middle| \; T_{Fire} \; \right)$ where T_{Fire} is a 184×40 matrix processing the CRC. We finally append four columns of zeroes at the end of G'_{Fire}, to form a 184×228 matrix, noted G_{Fire}.

Convolution

The 228×456 convolution binary matrix G_{Conv} may be obtained from the two sets of bits $\begin{cases} S_0 & = 11001 \quad \text{for even indexes} \\ S_1 & = 11011 \quad \text{for odd indexes} \end{cases}$ corresponding to the code polynomials (see section 2.1).

$$\begin{array}{c|cc|cccccccccc c}
 & S_0 & S_1 & & & & & & & & & & & \\
 & 1 & 1 & & & & & & & & & & & \\
 & 1 & 1 & 1 & 1 & & & & & & & & & \\
 & 0 & 0 & 1 & 1 & 1 & 1 & & & & & & & \\
 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & & & & & \\
\hline
\hookrightarrow & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & \hookleftarrow \\
\hline
 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & \\
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 &
\end{array}$$

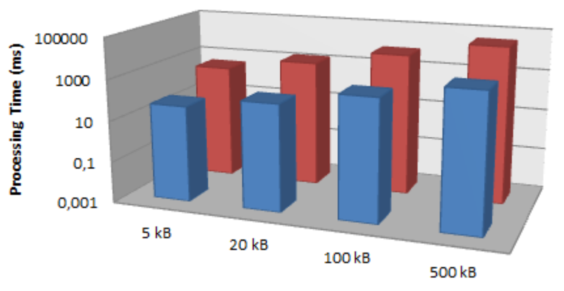
When expressed vertically, S_0 & S_1 form a pattern (designated by $\hookrightarrow$ ... $\hookleftarrow$) that is propagated through G_{Conv}: Each line contains this pattern, starting from the upper-left corner ; then it is shifted 2 bits to the right for each new line.

Interleaving

The interleaving matrix is a simple 456×456 permutation matrix, totally described by $(v_k)_{k \in [\![0,455]\!]}$.

Global Matrix

Our implementation takes advantage of these matrices during the encoding phase. Let $G = G_{Fire} \times G_{Conv} \times G_{Intl}$. This 184×456 binary matrix can be used to directly compute the encoded message c from clear text x: $c = x \times G$. The use of matrices instead of standard functions makes the whole encoding process a lot faster (see performance analysis on figure 3).



	5 kB	20 kB	100 kB	500 kB
🟦 Using Matrix G (µs)	47040	155117	754746	3635380
🟥 Using Functions (µs)	405386	1540888	7721346	38978659

Figure 3: Encoding speed test, logarithmic scale

Decoding

For the simulation to be realistic, it is mandatory to encode the message, thus creating redundancy – that is one of the conditions for the attack to work. Since radio waves are not used in this simulator (there is a simple loopback at the end of the chain), no transmission errors should appear.[2] Hence, it is not necessary to be able to correct any errors. The decoding phase is therefore much simpler. First the encoded message must be de-interleaved (that is made simply by inverting G_{Intl}), then the convolution can be reverted by bitwise XORing each couple of bits (and trimming the additional zeroes at the end), and finally the CRC can just be trimmed to make the message fit the initial data size.

[2]If you really want to correct errors, you can consider using the Viterbi algorithm to revert the convolution step. The Fire code used here is much more a way of detecting residual errors than correcting them.

3.3 Encryption

A A5/2 generator produces keystream (in accordance to section 2.2). The only difference here is that our implementation uses all the bits produced by the cipher with no distinction of purpose (phone-to-network/network-to-phone – still we skip the first 99 ones). It has absolutely no incidence on the global process (except in a conceptual way), nor on the attack, but it simplifies the comprehension of the attack steps. Both encryption and decryption consist of bitwise XORing the keystream with the message.

4 Barkan, Biham & Keller's attack on A5/2

4.1 Overview

GSM uses correcting codes introducing a lot of redundancy in the encoding process. They also diffuse the data throughout the message, so that an error occurring over several bits of data will not tamper with both the original and the redundant value. A5/2 is a stream cipher: it is only used to produce a stream of bits, all related to the key, but in a manner difficult to reverse. This stream is then used to xor-encrypt the bits of the message. Because in the end, the encryption is just xoring, errors would not be propagated, or obfuscated, so an error in the ciphertext means exactly one error on the same bit of the cleartext. This leads to a particular flaw in GSM standard used in this attack: the error correcting codes are applied before encryption, on the cleartext. That means, that the information contained in the cleartext is first duplicated from 184 bits to 456 bits, and then encrypted, so that every one of these 456 bits gives you a different bit of information on the key used in the cipher. Usually (in other more secure systems), the encryption comes first, which means that every one of the 184 bits of cleartext is first encrypted, and the encrypted bits are then duplicated into redundant data to form a final message of 456 bits: contrary to the GSM approach, instead of 456 bits of usable information, one may really only have 184, because that is the quantity of keystream that was used in the encryption process.

4.2 Implementation

This attack focuses on the misuse of the error-correcting codes in GSM. The matrices considered here are the ones mentioned in section 3.2. In [6], the authors introduce another important matrix: the parity-check matrix, which is the matrix H such that: c is a codeword (i.e. $\exists x$ cleartext, $c = x \times G$) $\iff$ the syndrome $S = H \times c = 0$. This matrix is easy to compute when G is in systematic form G':

$$G' = \left(\begin{array}{ccccc|c} 1 & 0 & \cdots & & 0 & \\ 0 & 1 & \ddots & & & \\ \vdots & & \ddots & & \vdots & T \\ & & \ddots & 1 & 0 & \\ 0 & & \cdots & 0 & 1 & \end{array}\right)$$

The matrix H is then given by:

$$H = \left(\begin{array}{c|ccccc} & 1 & 0 & \cdots & & 0 \\ & 0 & 1 & \ddots & & \\ T^t & \vdots & & \ddots & & \vdots \\ & & & \ddots & 1 & 0 \\ & 0 & & \cdots & 0 & 1 \end{array}\right)$$

Considering GSM, G would be a 184×456 matrix, whereas H would be a $456 - 184 = 272$-line, 456-column matrix. Unfortunately, the matrix G used for the whole encoding in GSM is not in this convenient systematic form, and [6] does not explain how to handle this case. With a Gauss elimination, however, one can find a 184×184 invertible matrix L and a 456×456 permutation matrix P so that $G' = L \times G \times P$. P being a premutation matrix, it is invertible, thus $G = L^{-1} \times G' \times P^{-1}$. We will here show that the matrix $H \times P^{-1}$ is actually a parity-check matrix for G in its non-systematic form. Let G be then entire code matrix, G' the matrix for the same code in systematic form, and L, P the afforementioned matrices. Let c be a codeword for G, i.e. $\exists x / c = x \times G$. If we call $x' = x \times L^{-1}$, $c = x' \times L \times G$, so $c \times P = x' \times L \times G \times P = x' \times G'$. Hence, $c \times P$ is a codeword for the matrix G', which is in systematic form. P being a permutation matrix, $P^{-1} = P^t$, and c being a vector, $c \times P = P^{-1} \times c^t$. We now compute the product with H, and $c \times P$ being a codeword for the matrix G', $H \times c \times P = 0$, so $H \times P^{-1} \times c^t = 0$. The matrix $H \times P^{-1}$ is indeed a parity-check matrix for the code in its non-systematic form. Our implementation of the tool computes every one of these matrices in order to compute the real syndrome of the given ciphertext. When given a ciphertext C, using part of the demonstration in [6], H can be used to compute a syndrom giving information on the value of the keystream used k. Contrary to their model, the encoding does not use any g to our knowledge, and can be modeled with only a matrix G so the equations are not exactly the same. Let C be the ciphertext encoded from cleartext x, and then xored with the keystream k: $C = x \times G \oplus k$. So, $H \times P^{-1} \times C = H \times (x \times G \oplus k) = H \times P^{-1} \times (x \times G) \oplus H \times P^{-1} \times k = H \times P^{-1} \times k$.

4.3 Change of variables

The final aim of the attack is to recover the key. However, the syndromes computed only contain information about the keystream, which is calculated from the values of the variables in the LFSRs R_1, R_2, and R_3, which themselves depend on the key and the LFSR R_4. For the attack to work, the equations on the keystream must be adapted into equations on the variables of the LFSRs first. The dependencies between the LFSRs and the keystream vary greatly with the initial value of R_4, and since the variables of R_4 play no role in the value of the keystream, so it is not possible to find them. The possibilities for the initial value of R_4 must hence be bruteforced. Because R_4 contains 17 bits of data, and the 10^{th} one is always initialized to one, 2^{16} possibilities exist. However, the dependencies between the LFSRs and the keystream are not linear, but quadratic, as explained in [6], so one has to linearize the system, which leads to a description by a matrix of size 456×656. As was mentioned before, 2^{16} can arise, so as many matrices must be calculated and tried, in order to find the right one. That is the process as it was described in [6], but the tool we developed takes a slightly different approach. In that article, the authors mention that they need 8 frames of ciphertext in oreder to recover the key, but they do not describe the method used. The tool described here uses a simple Gauss elimination, but it needs 12 frames of ciphertext, 114-bit long each, that are three 456-bit encoded messages and keystreams. It computes 3 syndromes, and the 2^{16} matrices are not of size 456×656, but

$(3\times 456)\times 656$, i.e. 1368×656. We call one of these S, which can be subdivided into three 456×656 matrices S_1, S_2, S_3. We have the following linear system: $S\times r = K$, where k is the concatenation of 3 unknown keystreams k_1, k_2, and k_3, and r is the vector of unknowns representing the state of the LFSRs. The problem arises from the fact, that k is unkown. On the other hand, $H\times P^{-1}\times k_i$ can be computed, since it has been proven to be equal to $H\times P^{-1}\times C_i$, and all these elements are known. In the software, it is then the system $\begin{cases} H\times P^{-1}\times S_1\times r = H\times P^{-1}\times k_1 \\ H\times P^{-1}\times S_2\times r = H\times P^{-1}\times k_2 \\ H\times P^{-1}\times S_3\times r = H\times P^{-1}\times k_3 \end{cases}$ that is used. The matrices $(H\times P^{-1}\times S_i)_{i\in[\![1,3]\!]}$ are 272×656, so it is more efficient to store the matrices after multiplying them with $H\times P^{-1}$ and regrouping the three matrices together. Once that is done, the original keysetup must be reversed in order to find the key. The keysetup being a farly complicated process, the easiest way to do that, which was chosen in the implementation, is to proceed to the keysetup with variables instead of real values, and then analyze the dependencies in a linear system that can be solved by Gauss Elimination.

4.4 Attack scenario

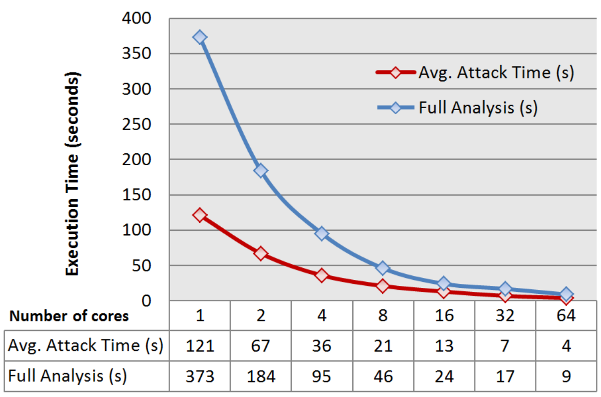
As explained above, this attack needs to exhaustively explore 2^{16} cases, according to the initial value of R_4. That means 2^{16} matrices of size 1368×656. Because that would last a while to calculate, and because they only need to be calculated once – they only depend on the design of A5/2, so they are unique –, it seems like a good solution to precalculate and store them. The average time cost would be 3 minutes 24s, and it would generate 6,85 GB of data, but memory space is cheap nowadays, so that should not be a problem. That was the solution in the article, but the implemented solution computes the matrices of size 816×656 as described above, which only takes 4,08 GB of memory and took 2 hours 23 minutes 44 seconds to run. Because this data can be calculated in advance, it does not matter that it takes a few hours, and it enables the actual attack to be much faster, since the products would have been computed anyways. The most time-consuming process in the attack itself is the exhaustive search through the 2^{16} matrices, so it is important that it takes as few time as possible to eliminate the wrong ones. That is why several filters were implemented. During the attack, the linear system is first trigonalized using Gauss elimination, but since there are 1368 equations for only 656 variables, a lot of these equations are redundants. Fortunately enough, we checked on thousands of cases that the program always has enough equations to solve the problem. That could probably be explained by the fact that the A5/2 cipher aims to diffuse the information, so over a short period of keystream, the least is redundant. However, what that means, is that after the elimination, $816-656=160$ equations are of the form $0=b$, where $b\in\{0,1\}$. Matrices that result in any strictly positive number of equations $0=1$ cannot be solved and should therefore be eliminated. This criterium makes elimination of matrices very fast, but two others can be checked: since the variables are not really independant, but only treated as such, one can check that the quadratic variables are in accordance with the others. The last check is done during the reversing of the keysetup. Actually, however, the first criteria suffices to eliminate 100% of the inadequate matrices on the 50000^{+} random cases we tested. Such numbers would indicate that there is a theoretical reason behind that fact, but we could not take the time to look for it.

Performances

Our testing environment is a Bull Escala server (i686) composed of four 8-core processors (Intel Xeon 75xx Nehalem EX, with Hyper-threading and TurboBoost) providing us with 32 physical cores (64 virtual cores), running Red Hat Enterprise Linux 5 with 128GB of RAM. Although the coding & decoding process is almost instant[3] (see figure 3), the attack phase requires much more resources, both in terms of memory and processing time. Because we use a dictionary of resolution matrices to realize the attack, the whole attack process is easily parallelizable: on the one hand the dictionary can be calculated in separate processes, and on the other hand the secret key search can be dispatched on several threads using different dictionary subsets. Note that to get even better performances, we chose to load the full dictionary in RAM instead of using the hard drive cache to re-

[3]For very large chunks of data, it can even be done in parallel

trieve data[4]. Figure 4 gathers some average timing results depending on the number of cores used: the blue chart denotes the time needed to analyze all the matrices (thus not stopping the program when the correct key is found), and the red one denotes the average attack time on 10000 random secret keys and frame ids.



Number of cores	1	2	4	8	16	32	64
Avg. Attack Time (s)	121	67	36	21	13	7	4
Full Analysis (s)	373	184	95	46	24	17	9

Figure 4: Attack speed test

5 Contributions

5.1 Description of the tool

We aimed at producing an easy-to-use & turnkey tool allowing direct GSM hacking to demonstrate that A5/2 is not only theoretically weak, but also really easy to crack in a concrete way. We hope this will finally make the public aware of the problem arisen by GSM standard.
The program was written in C language, and the source code is fully available under the terms of the GNU General Public License[7] on our website[5]. We did not operate on a real system for technical, operational, and legal reasons, but we simulated the whole GSM encoding process, and then performed the attack on the resulting data.
The only real implementations related to our work existing on the Internet[5] were an implementation of the A5/2 cipher which seemed inconvenient in the way it handled data, and a generic implementation of blocked convolutional codes and viterbi decoding, that presented what appeared to us as a bug and did not seem to be actively supported since 1999. As a result, we decided to write our own versions of all the modules presented in this paper, apart from the voice encoder (provided by SoX).

[4]We gain in average a factor 2 on matrices lookup time
[5]To our knowledge, as for June 2011

5.2 Adaptation to Physical Attack

In order to transpose the attack to the physical layer, the required equipment only consists of a USRP (*Universal Software Radio Peripheral*[8]) and of an appropriate driver (for example GNURadio[9] or OpenBTS[10]). The attack being focused on the SACCH, there would also be a need to analyze a bit further the full GSM specifications to be able to decode data on other channels. Also (in addition to the fact that breaking ciphers on real data may not be considered to be legal), the legal framework concerning the authorization of using GSM frequency bands may depend of the country you reside in. Chris Paget exposed in a Defcon video recording [11] the US legislation on this matter.

6 Countermeasures

As we commented above, what made this attack possible is a conception flaw inherent to the design of the GSM standard, so a natural reaction would be to change the standard, which means changing both software and hardware (as many functionnalities are directly implemented in chips for performance concern and low energy consumption) of all GSM base stations and handsets. This is a major inconvenience, because GSM is already very widely deployed. The change would be very expensive and very difficult to organize. That is why the industry has answered this threat by introducing other algorithms such as A5/1 and A5/3, which are much more secure, and can be used by the base stations. The problem remaining is, that in order to save room, these 3 algorithms share the same hardware, and the same key.

Knowing that the BTS (*Base Transceiver Station*) has full control over the link securing method used to communicate with handsets, the GSM is therefore very vulnerable to man-in-the-middle attacks, because a malicious GSM base station could impersonate the network to the handset, and demand it use A5/2, recover its secret key, and then impersonate the handset to the network, this way having total control over the traffick from and to the trapped handset. The only solution would be to disable the possibility of using A5/2 in the handset, and that is very hard to do for all the handsets. This solution is not realistically manageable on existing phones.

Our attack is very practical because it only needs some chunks of ciphertext. But because of that it has an important drawback: it cannot work when errors occur. The whole attack is based on the fact that, to take into account the errors that may occur on the air, GSM uses error correcting codes, and in this case not wisely. The problem is, because the correction is applied to the cleartext, it is not possible to correct eventual errors on the ciphertext, without deciphering it first. In the standard process –the way GSM works, not considering any attack– that is not such a big deal because the encryption process only consists of a bitwise xor with the keastream, and no error propagation occur (contrary to block cipher encryption). Thus, if a bit is wrong in the cipher, it can be decrypted excatly the same way and then error correcting codes would then take over from it, successfully recovering the unaltered data. However, considering our attack, the event of bits flipping randomly in the cipher may lead the process to be considerably harder: in fact, each bit of the keystream is part of a system of equations and a single error would lead to inconsistancies, dismissing the tested configuration of R4. Two different kind of errors can happen : Bits may be flipped (some bits that had 1 as their value now have 0, or vice-versa) or erased (some bits have an undetermined value). In the second case [6], it is still possible to correct the errors without exhaustively trying all the possibilities, just by dismissing the equations related to the erased bit. However, in case of flipped bits, the only solution is to try to guess which bits were flipped. The time complexity is exponential, so it can still be manageable for a very low number of errors[7], but it is not a viable solution for any number[8]. For more accurate information about error handling, please refer to section 5 of [6]. Considering the previous analysis, a simple countermeasure could be to randomly inject errors into the ciphertext, so that they can be corrected by the correcting codes in the system that already knows the key, and make it difficult, or even almost impossible for an attacker to use this attack.

[6]And only if the hardware provides the erasures locations

[7]In order to decrypt a message containing only 2 errors agmonst 3 chunks of 456 bits, the time needed on our test environment is approx. 6 months of constant processing.

[8]The only solution in this case is to discard frames that do not lead to a result, and hope of getting unaltered others

References

[1] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In *FSE: Fast Software Encryption*, pages 1–18. Springer-Verlag, 2000.

[2] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time attack on the A5/3 cryptosystem used in third generation GSM telephony. 2010. `http://eprint.iacr.org/`.

[3] ETSI - European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+) (GSM); Channel coding (GSM 05.03 version 8.5.1 Release 1999)*, November 2000.

[4] *SoX*. `http://sox.sourceforge.net`.

[5] Nicolas Paglieri and Olivier Benjamin. *A52HackTool Project Webpage*. `http://www.ni69.info/security-gsm-en.php`.

[6] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. pages 600–616. Springer-Verlag, 2003.

[7] Free Software Foundation. *GNU General Public License, v3*. `http://www.gnu.org/licenses/`.

[8] Ettus Research LLC. *Universal Software Radio Peripheral*. `http://www.ettus.com`.

[9] *GNURadio Software Development Toolkit*. `http://gnuradio.org`.

[10] Kestrel Signal Processing, Inc. *OpenBTS*. `http://openbts.sourceforge.net`.

[11] Chris Paget. *Practical Cellphone Spying*. `http://www.youtube.com/watch?v=DU8hg4FTm0g`.