A Hardware-Assisted Realtime Attack on A5/2 without Precomputations

Andrey Bogdanov, Thomas Eisenbarth, Andy Rupp

Horst-Görtz Institute for IT-Security
Ruhr-University Bochum, Germany.
{abogdanov, eisenbarth, arupp}@crypto.rub.de

Abstract. A5/2 is a synchronous stream cipher that is used for protecting GSM communication. Recently, some powerful attacks [2,10] on A5/2 have been proposed. In this contribution we enhance the ciphertext-only attack [2] by Barkan, Biham, and Keller by designing special-purpose hardware for *generating and solving* the required systems of linear equations. For realizing the LSE solver component, we use an approach recently introduced in [5,6] describing a parallelized hardware implementation of the Gauss-Jordan algorithm. Our hardware-only attacker immediately recovers the initial secret state of A5/2 - which is sufficient for decrypting all frames of a session - using a few ciphertext frames *without any precomputations and memory.* More precisely, in contrast to [2] our hardware architecture directly attacks the GSM speech channel (TCH/FS and TCH/EFS). It requires 16 ciphertext frames and completes the attack in about 1 second. With minor changes also input from other GSM channels (e.g., SDCCH/8) can be used to mount the attack.

Key words: A5/2, GSM, SMITH, special-purpose hardware, cryptanalysis, linear systems of equations, Gaussian elimination.

1 Introduction

The Global System for Mobile communications (GSM) was initially developed in Europe in the 1980s. Today it is the most widely deployed digital cellular communication system all over the world. The GSM standard specifies algorithms for data encryption and authentication. The originally specified encryption algorithm in this standard was the stream cipher A5/1. However, due to the export restrictions, for deploying GSM out of Europe a new intentionally weaker version of A5/1 was developed, the stream cipher A5/2. Though the internals of both ciphers were kept secret, their designs were disclosed in 1999 by means of reverse engineering [7].

The security of A5/1 has been extensively analyzed, e.g., in [1,3,4,9,11,14]. In this paper we focus however on the security of the (weaker) A5/2 algorithm. But note that although the use of this algorithm has been officially discouraged in the meantime, its security still has great importance on the security of GSM communication. This is not least due to flaws in the GSM protocols that allow to take advantage of attacks on A5/2 even if a stronger encryption algorithm (e.g., A5/1 or the new A5/3) is used [2]. These flaws can be exploited whenever the mobile phone supports a weak cipher.

A known-plaintext attack on A5/2 was presented in [10]. The actual attack requires only two plaintext frames, but these frames have to be exactly 1326 frames apart to fulfill a certain property. In [15] a weaker attack on A5/2 was proposed which requires 4 arbitrary plaintext frames and allows to decrypt most of the remaining communication. However, this attack does not recover the internal state of A5/2.

Recently, Barkan et al. [2] proposed a guess-and-determine attack that needs four plaintext frames to find an A5/2 session key. The general idea of this attack is to guess 16 bits of the internal state of the cipher and then express the output as a degree-2 function of the remaining unknown 61 initial state bits. Each known plaintext frame yields 114 quadratic equations in this way. Given 4 plaintext frames, one obtains an LSE of dimension 456×655 by linearizing the equations. Though the system is underdetermined, experiments show that this number of equations suffices to resolve the 61 original linear variables. In the same paper the attack is transformed into a ciphertext-only attack. Here, due to the fact that GSM employs error correction before encryption, the attacker knows the values of certain linear combinations of the stream bits. The attack consists of a precomputation phase in which the equation systems for all guesses are computed in advance and an online phase in which this data is used to quickly solve the equations for the specific input frames. It is important to note that these guesses also concern the initialization vectors (aka COUNT values) that are used to setup A5/2 and which are derived from the frame numbers. Thus, as usual for time-memory tradeoffs, depending on the precomputation time, memory and disk space one is willing to spend not all frames may be used in the online phase of the attack. The authors provide estimates for a full-optimized attack against the GSM control channel SDDCH/8. In this case the precomputation can be done in about 11 hours on a PC requiring 1GB of RAM and producing 4GB of data. In the online phase eight consecutive ciphertext frames are needed to recover the session key in about 1 second.

All of the above attacks against A5/2 share the feature that they have been designed for software implementation and so their efficiency has also been assessed for software. To the best of our knowledge the alternative of an efficient hardware implementation of an attack against A5/2 has not been analyzed thoroughly yet. For the case of A5/1, Pornin and Stern [16] discussed the possibility of accelerating attacks using software-hardware trade-offs. It is suggested that software should be used for the exhaustive search over clocking sequences and the generation of affine subspaces containing key candidates. Special-purpose hardware is proposed for the subsequent filtering of these affine subspaces. The hardware remains relatively simple, the software part being responsible for all complex operations including Gaussian elimination. In this paper we show that a *hardware-only attack* against A5/2 leads to significant improvements in terms of time, memory and flexibility compared to current software attacks, although existing attacks are already quite efficient.

Our general approach is similar to the ciphertext-only attack described in [2]. However, no precomputation is required and the ciphertext frames that can be used to mount the attack do not need to satisfy any special properties (e.g., appropriate differences of COUNT values). In contrast to the software implementation in [2], we designed our architecture to directly attack the speech channel. That means, it uses ciphertext frames from the GSM speech traffic channel (TCH/FS and TCH/EFS) instead of a specific control channel (e.g., SDCCH/8 in [2]) to mount the attack. The advantage is that eavesdropping can start immediately at any time during a call (not only at the set-up of the call) without waiting until appropriate data is transmitted over the specific control channel. However, since the proposed architecture is quite generic using minor changes also other GSM channels (e.g., SDCCH/8) can be attacked. Based on our architecture, even a hardware device is conceivable where the target channel can be chosen at runtime.

The basic architecture for attacking the speech channel requires 16 (consecutive) ciphertext frames as input and outputs the recovered secret initial state of A5/2. This initial state is sufficient to decrypt all frames of a session and to recover the key. The core blocks of the architecture are 3 equation generators and the solver for linear systems of equations. As a realization of the latter building block, we have chosen the SMITH-LSE-Solver recently proposed in [5,6]. In every iteration, each equation generator produces one linear equation with the secret state bits as variables. After 185 iterations (when 555 equations have been produced and loaded), the LSE solver performs parallelized Gauss-Jordan elimination. The output of the LSE solver suggests the secret state candidate that needs to be checked. The right candidate is found in this way after about 2^{28} clock cycles on average.

To have a proof of concept and a basis for evaluating the requirements on chip size and average power consumption of an ASIC implementation, we implemented all critical parts of our design in VHDL and synthesized it. Based on these results we estimate about 9.3 million gate equivalents for the whole architecture. Assuming a moderate operating speed of 256 MHz for the main chip component and 512 MHz for the rest, the architecture consumes roughly 12.8 Watts of energy and completes an attack in about 1 second on average. In comparison with a recent desktop PC CPU, the Core 2 Duo "Conroe" processor, our design requires less than 15% of the area and consumes less than one third of the power. Note that these estimates are based on our rather *unoptimized* design and we believe that there is still room for significant improvements in speed and chip area. For instance, the experiments in [2] suggest that there are only 450 equations needed (as opposed to 555) to determine a candidate. Having less equations reduces both required clock cycles and occupied chip area. Moreover, for both reasons also replacing SMITH by a systolic-array LSE solver as described in [12] seems to be a promising approach.

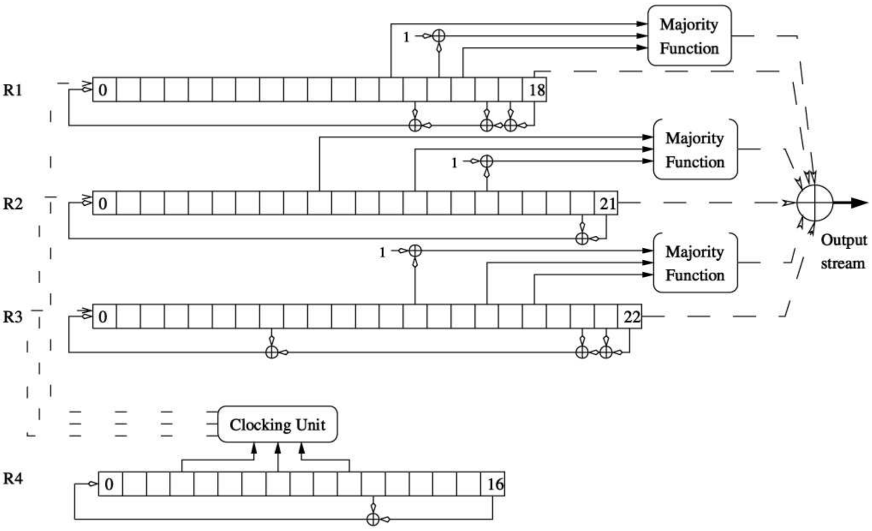


Fig. 1. Design of A5/2 (the Figure is due to [2])

2 The A5/2 Stream Cipher

A5/2 is a synchronous stream cipher accepting a 64-bit key $K = (k_0, \ldots, k_{63}) \in \mathrm{GF}(2)^{64}$ and a 22-bit initial vector $IV = (v_0, \ldots, v_{21}) \in \mathrm{GF}(2)^{22}$ derived from the 22-bit frame number which is publicly known. It uses four linear feedback shift registers (LFSRs) $R1$, $R2$, $R3$ and $R4$ of lengths 19, 22, 23 and 17 bits, respectively, as its main building blocks (see Figure 1). The taps of the LFSRs correspond to primitive polynomials and, therefore, the registers produce sequences of maximal periods. $R1$, $R2$ and $R3$ are clocked irregularly based on the current state of $R4$.

The A5/2 keystream generator works as follows. First, an *initialization phase* is run (see Figure 2). At the beginning of this phase all registers are set to 0. Then the *key setup* and the *IV setup* are performed. Here the key resp. IV bits are cyclically added to the registers modulo 2. At the end of the initialization phase one bit in each register is set to 1.

Then the *warm-up phase* is performed where R4 is clocked 99 times and and the output is discarded. Note that already during this phase and also during the stream generation phase which starts afterwards, the registers $R1$, $R2$ and $R3$ are clocked irregularly. More precisely, the stop/go clocking is determined by the bits $R4[3]$, $R4[7]$ and $R4[10]$ in each clock cycle as follows: the majority of the three bits is computed, where the majority of three bits a, b, c is defined by $maj(a, b, c) = ab \oplus ac \oplus bc$. $R1$ is clocked iff $R4[10]$ agrees with the majority. $R2$ is clocked iff $R4[3]$ agrees with the majority. $R3$ is clocked iff $R4[7]$ agrees with the majority. In each cycle at least two of the three registers are clocked. After these clockings, $R4$ is (regularly) clocked, and an output bit is generated from

```
$R1 \leftarrow 0,\ R2 \leftarrow 0,\ R3 \leftarrow 0,\ R4 \leftarrow 0;$
for $i = 0...63$ do
  -Clock $R1, R2, R3, R4;$
  -$R1[0] \leftarrow R1[0] \oplus k_i, R2[0] \leftarrow R2[0] \oplus k_i, R3[0] \leftarrow R3[0] \oplus k_i, R4[0] \leftarrow R4[0] \oplus k_i;$
for $i = 0...21$ do
  -Clock $R1, R2, R3, R4;$
  -$R1[0] \leftarrow R1[0] \oplus v_i, R2[0] \leftarrow R2[0] \oplus v_i, R3[0] \leftarrow R3[0] \oplus v_i, R4[0] \leftarrow R4[0] \oplus v_i;$
$R1[15] \leftarrow 1,\ R2[16] \leftarrow 1,\ R3[18] \leftarrow 1,\ R4[10] \leftarrow 1;$
```

Fig. 2. Initialization phase of A5/2

the values of $R1$, $R2$, and $R3$ by adding their rightmost bits to three majority values, one for each register (see Figure 1). After warm-up A5/2 produces 228 output bits, one per clock cycle. 114 of them are used to encrypt uplink traffic, while the remaining bits are used to decrypt downlink traffic. In the remainder of this paper we always consider only a fixed half of this keystream used to encrypt the traffic in one direction.

3 Description of the Attack

Our general approach is similar to the ciphertext-only attack described in [2]. However, no precomputation is required and the ciphertext frames that can be used to mount the attack do not need to satisfy special properties like having appropriate frame numbers. The attack requires the ciphertext of any l frames encrypted using the same session key K. The parameter l depends on the channel that should be attacked. For instance, we need about $l = 16$ frames for attacking the speech channel as shown in this paper and about $l = 8$ frames to attack the SDCCH/8 channel of a GSM communication as shown in [2].

The general idea is to guess the internal state of the register $R4$ right after initialization (we have 2^{16} possible states) and write every bit of the generated key stream, that has been used to encrypt the l known ciphertext frames, in terms of the initial states of the registers $R1, R2$ and $R3$. We then use certain information about the key stream bits – which are provided by the error correction coding of the GSM channel – to construct an overdetermined quadratic system of equations. This system is linearized and then solved using Gaussian elimination. Above procedure is repeated for different guesses of $R4$ until the correct solution is found. Using this solution, we can easily construct the internal state of A5/2 after initialization for an arbitrary frame that has been encrypted using K. This is already sufficient to decrypt all frames of a session, since we can construct the respective states and load them into the A5/2 machine. However, by reversing the initialization phase, we can also recover the session key.

In the following we consider the details of the attack. To this end we first introduce the basic notation. We denote the l known ciphertext frames by $C_0, \ldots, C_{l-1}$ and the corresponding (unknown) plaintext frames by $P_0, \ldots, P_{l-1}$. For each frame C_h (or P_h) we denote the respective initialization vector by

$IV_h = (v_{h,0}, \ldots, v_{h,21})$ and the key stream by $S_h = (s_{h,0}, \ldots, s_{h,113})$. Furthermore, let $R1^{(h)}$, $R2^{(h)}$, $R3^{(h)}$ and $R4^{(h)}$ be the internal states of the registers of A5/2 during a certain cycle when generating S_h.

3.1 Expressing Stream Bits as Register States

Let us consider the stream generation for a frame C_h. At the beginning of the initialization phase the registers $R1^{(h)}$, $R2^{(h)}$, $R3^{(h)}$ and $R4^{(h)}$ are all set to zero. Then the key setup is performed for 64 clock cycles, where in each cycle the first bit of each LFSR is set to the sum of the respective feedback value and one of the key bits (see Section 2). After that, due to the linearity of the feedback functions the bits of the three registers can be written as certain linear combinations of K, e.g., $R1^{(h)}[0] = k_0 \oplus k_{19} \oplus k_{38} \oplus k_{47}$. In the subsequent initialization step, the IV setup, the initialization vector IV_h is added to the content of the registers in an analogous manner. Thus, the resulting register bits are (known) linear combinations of the key bits and the IV bits. Finally, certain bits of the registers are set to 1. More precisely, after initialization the registers $R1^{(h)}$ to $R4^{(h)}$ can be written as

$$\begin{aligned}
R1^{(h)} &= (\alpha_0 \oplus \sigma_{h,0}, \ldots, \alpha_{14} \oplus \sigma_{h,14}, 1, \alpha_{15} \oplus \sigma_{h,15}, \ldots, \alpha_{17} \oplus \sigma_{h,17}),\\
R2^{(h)} &= (\alpha_{18} \oplus \sigma_{h,18}, \ldots, \alpha_{33} \oplus \sigma_{h,33}, 1, \alpha_{34} \oplus \sigma_{h,34}, \ldots, \alpha_{38} \oplus \sigma_{h,38}),\\
R3^{(h)} &= (\alpha_{39} \oplus \sigma_{h,39}, \ldots, \alpha_{56} \oplus \sigma_{h,56}, 1, \alpha_{57} \oplus \sigma_{h,57}, \ldots, \alpha_{60} \oplus \sigma_{h,60}),\\
R4^{(h)} &= (\alpha_{61} \oplus \sigma_{h,61}, \ldots, \alpha_{70} \oplus \sigma_{h,70}, 1, \alpha_{71} \oplus \sigma_{h,71}, \ldots, \alpha_{76} \oplus \sigma_{h,76}),
\end{aligned} \tag{1}$$

where $\alpha_i \in \mathsf{span}(k_0, \ldots, k_{63})$ and $\sigma_{h,i} \in \mathsf{span}(v_{h,0}, \ldots, v_{h,21})$.

This is the starting point of our attack. First observe that since IV_h is known, the values $\sigma_{h,0}$ to $\sigma_{h,76}$ can be considered as known constants. So only the α_i values are unknowns. Note that we have the *same* α_i's for *all* frames C_h. In the following, we guess the values of $\alpha_{61}, \ldots, \alpha_{76}$, determine the initial secret state $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{60}) \in \mathrm{GF}(2)^{61}$ and verify this solution.[1] We have to repeat this procedure at most 2^{16} times until $\alpha_{61}, \ldots, \alpha_{76}$ take on the correct values.

In order to determine α, we have to write the bits of the key stream S_h for each frame C_h in terms of α and use certain information about these bits to construct a linear system of equations which is then solved by Gaussian elimination. Let us now see how this can be done. Remember that after initialization, irregular clocking is performed in each cycle as described in Section 2. Before the first stream bit for C_h is generated the warm-up phase is executed running for 99 cycles. After warm-up, a stream bit is generated from the current internal states of $R1^{(h)}, R2^{(h)}$ and $R3^{(h)}$ every cycle. In an *arbitrary* cycle of A5/2 (after initialization), these states can be written as

$$\begin{aligned}
R1^{(h)} &= (\beta_{h,0} \oplus \delta_{h,0}, \ldots, \beta_{h,18} \oplus \delta_{h,18}),\\
R2^{(h)} &= (\beta_{h,19} \oplus \delta_{h,19}, \ldots, \beta_{h,40} \oplus \delta_{h,40}),\\
R3^{(h)} &= (\beta_{h,41} \oplus \delta_{h,41}, \ldots, \beta_{h,63} \oplus \delta_{h,63}),
\end{aligned} \tag{2}$$

[1] Since the registers $R1^{(h)}, R2^{(h)}$ and $R3^{(h)}$ are clocked irregularly after initialization based on certain bits of $R4^{(h)}$ by guessing α_{61} to α_{76} the clocking of these registers are fully determined.

where $\beta_{h,0},\dots,\beta_{h,18} \in \mathsf{span}(\alpha_0,\dots,\alpha_{17})$, $\beta_{h,19},\dots,\beta_{h,40} \in \mathsf{span}(\alpha_{18},\dots,\alpha_{38})$, $\beta_{h,41},\dots,\beta_{h,63} \in \mathsf{span}(\alpha_{39},\dots,\alpha_{60})$, and $\delta_{h,i} \in \mathsf{span}(v_{h,0},\dots,v_{h,21},1)$. Note that the linear combinations $\beta_{h,i}$ depend on the specific frame C_h, since the clocking of the registers now depends on IV_h. (Certainly, $\beta_{h,i}$ and $\delta_{h,i}$ also depend on the specific clock cycle.) However, it is important to observe that we know the specific linear combination of α_j's each $\beta_{h,i}$ is composed of as well as the concrete value of each $\delta_{h,i}$, since we know IV_h and fix some values for $\alpha_{61},\dots,\alpha_{76}$.

A stream bit $s_{h,k}$ ($k \in \{0,\dots,113\}$) is generated by summing up the output of the three majority functions and the rightmost bits of the registers $R1^{(h)}$, $R2^{(h)}$ and $R3^{(h)}$ (see Fig. 1). More precisely, in terms of the current state (k clock cycles after warm-up) of these registers the output bit can be written as

$$\begin{aligned} s_{h,k} = {} & \mathit{maj}(\beta_{h,12} \oplus \delta_{h,12}, \beta_{h,14} \oplus \delta_{h,14} \oplus 1, \beta_{h,15} \oplus \delta_{h,15}) \\ & \oplus \mathit{maj}(\beta_{h,28} \oplus \delta_{h,28}, \beta_{h,32} \oplus \delta_{h,32}, \beta_{h,35} \oplus \delta_{h,35} \oplus 1) \\ & \oplus \mathit{maj}(\beta_{h,54} \oplus \delta_{h,54} \oplus 1, \beta_{h,57} \oplus \delta_{h,57}, \beta_{h,59} \oplus \delta_{h,59}) \\ & \oplus \beta_{h,18} \oplus \delta_{h,18} \oplus \beta_{h,40} \oplus \delta_{h,40} \oplus \beta_{h,63} \oplus \delta_{h,63} \, . \end{aligned} \tag{3}$$

It is important to note that due to the majority function, each output bit is a quadratic function in $\alpha_0,\dots,\alpha_{60}$. More precisely, it has the general form

$$\begin{aligned} s_{h,k} = {} & \sum_{0\le i<j\le 17} b_{i,j}\alpha_i\alpha_j \oplus \sum_{18\le i<j\le 38} b_{i,j}\alpha_i\alpha_j \\ & \oplus \sum_{39\le i<j\le 60} b_{i,j}\alpha_i\alpha_j \oplus \sum_{0\le i\le 60} a_i\alpha_i \oplus c, \end{aligned} \tag{4}$$

for some $b_{i,j}, a_i, c \in \{0,1\}$.

To linearize above relations we simply replace each quadratic term $\alpha_i\alpha_j$ by a new variable $\gamma_{i,j}$. In this way we obtain $\frac{18\cdot 17}{2} + \frac{21\cdot 20}{2} + \frac{22\cdot 21}{2} = 594$ new variables. Thus, each stream bit can be described by at most 655 variables (and a constant).

3.2 Setting Up an LSE Using Speech Channel Data

Now, we describe how a ciphertext-only attack using data from the speech traffic channel can be mounted. In the case of a ciphertext-only attack the direct output stream of A5/2 is not (partly) known. However, we have access to certain linear combinations of the output bits. This is due to the fact that A5/2 is encrypting linearly in the plaintext (as any synchronous stream cipher) and to the linear error-correction coding procedure that is performed *before* encryption. The applied error-correction procedure is however specific to the GSM channel (see [13] for details on GSM channel coding). How this procedure can be exploited in the case of the GSM control channel SDCCH/8 is sketched in [2]. We analyze how this can be done in the case of the full-rate speech traffic channel (TCH/FS and TCH/EFS) where a different interleaving procedure is used. We like to point out that our description is more constructive and detailed compared to the one in [2], making it especially useful with regard to an actual implementation.

To protect a 260-bit block of speech data produced by the speech coder against transmission errors a multi-stage error-correction procedure is performed.

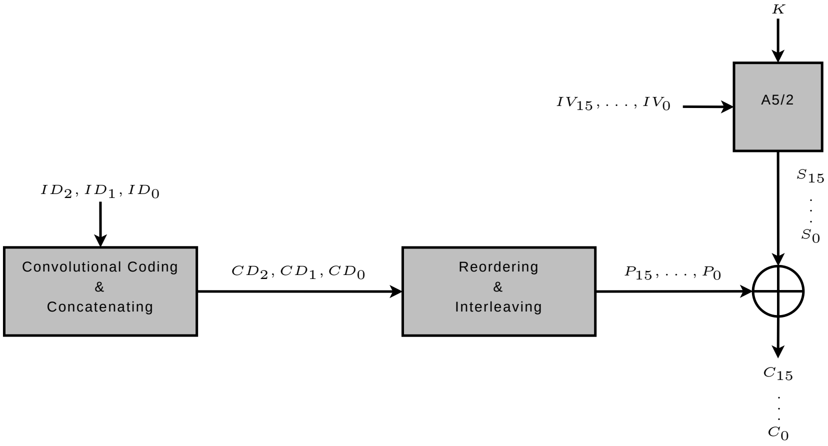


Fig. 3. Simplified view on the GSM convolutional coding, interleaving and A5/2 encryption process

This procedure increases the data size by adding redundant data in each stage and also reorders bits. We are interested in the last two stages of this procedure which are depicted in Figure 3. Here the 267-bit blocks ID_i containing some intermediate data are input to a so-called non-recursive binary convolutional encoder (of rate 1/2 with memory length 4 and constant length 5). The outputs of the convolutional coder are the 456-bit blocks CD_i. The function CC computed by the convolution encoder can be described as follows:

$$CC : ID_i = (id_{i,0}, \ldots, id_{i,266}) \mapsto (cd_{i,0}, \ldots, cd_{i,455}) = CD_i \text{ , where}$$
$$cd_{i,j} = \begin{cases} id_{i,k} \oplus id_{i,k-3} \oplus id_{i,k-4}, & 0 \leq j \leq 377 \text{ and } j = 2k \\ id_{i,k} \oplus id_{i,k-1} \oplus id_{i,k-3} \oplus id_{i,k-4}, & 0 \leq j \leq 377 \text{ and } j = 2k+1 \\ id_{i,182+(j-378)}, & 378 \leq j \leq 455 \end{cases} \quad (5)$$

Note that the last 78 bits of ID_i are actually not protected by a convolutional code. Rather these bits are just copied unchanged to the tail of CD_i. The important property of the convolutional code bits of an arbitrary block CD_i (bits 0-377) - that is exploited later on - are the following linear dependencies that hold for $1 \leq j \leq 184$:

$$cd_{i,2j} \oplus cd_{i,2j+1} \oplus cd_{i,2j+2} \oplus cd_{i,2j+3} \oplus cd_{i,2j+6} \oplus cd_{i,2j+8} \oplus cd_{i,2j+9} = 0 \quad (6)$$

As we can see in Figure 3, the blocks CD_i are not directly encrypted. Prior to encryption, they are first reordered and interleaved “block diagonal”. The result of the interleaving is a distribution of the reordered 456 bits of a given data block CD_i over the eight 114-bit blocks $P_{4i+0}, \ldots, P_{4i+7}$ using the even numbered bits of the first 4 blocks and odd numbered bits of the last 4 blocks. The reordered bits of the next data block CD_{i+1}, use the even numbered bits of the blocks

$P_{4i+4}, \dots, P_{4i+7}$ and the odd numbered bits of the blocks $P_{4i+8}, \dots, P_{4i+11}$. The interleaving of CD_{i+2} and subsequent blocks is done analogously. So new data starts every 4th block and is distributed over 8 blocks. Considering the example in Figure 3, this means that each of the blocks $P_0, \dots, P_3$ contains 57 bits of data from CD_0, $P_4, \dots, P_7$ each contains 57 bits from CD_0 and 57 bits from CD_1, $P_8, \dots, P_{11}$ each contains 57 bits from CD_1 and 57 bits from CD_2 and finally each of the blocks $P_{12}, \dots, P_{15}$ contains 57 bits of CD_2.

More precisely, the following function can be used to describe the reordering and interleaving of data blocks:

$$\begin{gathered} f : \mathbb{N} \times \{0, \dots, 455\} \to \mathbb{N} \times \{0, \dots, 113\} \\ (i, j) \mapsto (4i + (j \bmod 8),\, 2(49j \bmod 57) + (j \bmod 8) \operatorname{div} 4) \end{gathered} \tag{7}$$

Then we have the following relation between the bits CD_i and the output blocks $P_{(4i+0)}, \dots, P_{(4i+7)}$:

$$cd_{i,j} = p_{f(i,j)}, \tag{8}$$

where the right-hand side denotes the bit with index $f(i, j)$ belonging to block $P_{(4i+(j \bmod 8))}$.

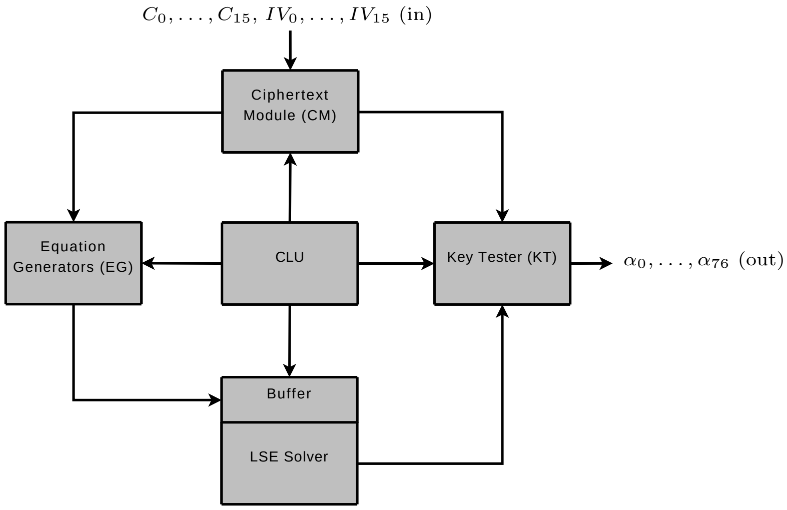
A 114-bit block P_i produced by the interleaver is then encrypted by computing the bitwise XOR with the output stream S_i resulting in the ciphertext frame C_i. The linear dependencies of the convolutional code bits seen in Equation 6 also propagate to the ciphertext because the encryption is linear in the plaintext and the keystream. So taking the interleaving and reordering into account, we can exploit this property to obtain equations of the form

$$\begin{aligned} &c_{f(i,2j)} \oplus c_{f(i,2j+1)} \oplus c_{f(i,2j+2)} \oplus c_{f(i,2j+3)} \oplus c_{f(i,2j+6)} \oplus c_{f(i,2j+8)} \oplus c_{f(i,2j+9)} \\ &\oplus s_{f(i,2j)} \oplus s_{f(i,2j+1)} \oplus s_{f(i,2j+2)} \oplus s_{f(i,2j+3)} \oplus s_{f(i,2j+6)} \oplus s_{f(i,2j+8)} \oplus s_{f(i,2j+9)} \\ &= p_{f(i,2j)} \oplus p_{f(i,2j+1)} \oplus p_{f(i,2j+2)} \oplus p_{f(i,2j+3)} \oplus p_{f(i,2j+6)} \oplus p_{f(i,2j+8)} \oplus p_{f(i,2j+9)} \\ &= cd_{i,2j} \oplus cd_{i,2j+1} \oplus cd_{i,2j+2} \oplus cd_{i,2j+3} \oplus cd_{i,2j+6} \oplus cd_{i,2j+8} \oplus cd_{i,2j+9} = 0 \end{aligned} \tag{9}$$

for $0 \leq j \leq 184$. It is important to note that the ciphertext and stream bits in above equation do not belong to a *single* ciphertext block respectively stream. Rather, for fixed i eight consecutive ciphertext blocks and corresponding streams are involved in the 185 equations. A single equation involves bits from 5 different blocks and streams. These effects are due to the interleaving and reordering (and make an efficient hardware implementation somewhat tricky).

Hence, given 16 consecutive ciphertext blocks we can setup LSEs with 555 equations and 655 unknowns using the results from the previous section[2]. Though the LSEs are underdetermined, we found out by experiments (similar to [2]) that this number of equations is always sufficient to determine the 61 original linear variables α using Gaussian elimination. Having determined the values of these variables, merely the consistency with the quadratic equations needs to be checked to identify the correct secret initial state.

[2] Assuming that the first 8 blocks contain the encrypted data of a whole convolutional code block.



$C_0, \ldots, C_{15}$, $IV_0, \ldots, IV_{15}$ (in)
Ciphertext Module (CM)
Equation Generators (EG)
CLU
Key Tester (KT)
$\alpha_0, \ldots, \alpha_{76}$ (out)
Buffer
LSE Solver

Fig. 4. Overview of the proposed architecture

4 A Hardware Architecture for Attacking A5/2

4.1 Overview

Our architecture is sketched in Figure 4. It accepts 16 ciphertext frames and the 16 corresponding IVs as input. The hardware calculates and outputs the recovered 77-bit state $\alpha_0, \ldots, \alpha_{76}$.

The given ciphertext frames and IVs are stored in the Ciphertext Module (CM). Each of the three Equation Generators (EGs) generates 185 linear equations with the secret state bits α_i $(0 \leq i \leq 60)$ as variables (cf. Eq. (9)). The EGs receive the required IVs and ciphertext bits from the CM. A generated equation is passed to the buffer of the LSE Solver. This buffer is needed because the LSE Solver accepts only one equation per clock cycle, but the three EGs produce their equations simultaneously. After the LSE Solver is filled with 555 equations, it proceeds to the solving step and produces a candidate for the secret state. The secret state candidate is sent from the LSE Solver to the Key Tester (KT) that verifies whether the correct state has been found. This verification process is done in parallel to the determination of a new candidate. More precisely, while equations for the j-th candidate are generated by the EGs the $(j-1)$-th candidate is tested by the KT. All processes are controlled by the Control Logic Unit (CLU) that performs synchronization and clocking for the CM, EGs, the LSE Solver, and the KT. Its main task is to ensure that the right stream and ciphertext bits are combined (within the EGs and also the KT) to form the desired equations as described in Section 3.2 in Eq. (9).

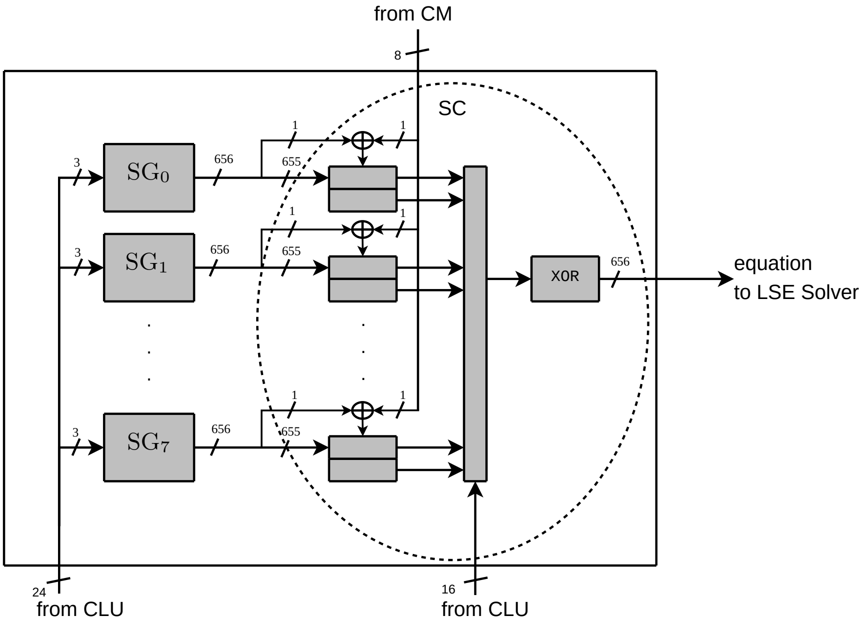
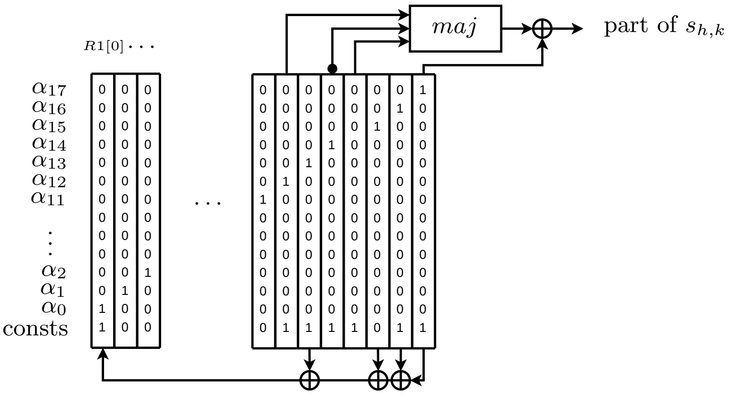


Fig. 5. Equation Generator with its components Stream Generator (SG) and Stream Combiner (SC)

4.2 Equation Generators (EGs)

Three EGs are used to generate the system of linear equations for the LSE Solver. Each EG is associated with one of the 3 convolutional code blocks $CD_0, \ldots, CD_2$ whose data is spread - due to interleaving - over 8 of the 16 given ciphertext blocks C_h (cf. Section 3.2). By means of the CM an EG has access to the required 8 ciphertext blocks and the corresponding IVs and generates 185 equations from this data.

As shown in Figure 5, an EG consists of eight Stream Generators (SGs) and one Stream Combiner (SC) which are all controlled by the CLU. Each of the eight SG is associated with one of the eight ciphertext frames C_h related to its EG. More precisely, Stream Generator SG$_j$ $(0 \leq j \leq 7)$ belonging to Equation Generator EG$_i$ $(0 \leq i \leq 2)$ is associated with frame $C_{(4i+j)}$. The SG for a frame C_h consists of an expanded A5/2 engine and can produce linearized terms for the 114 stream bits $s_{h,k}$ (cf. Eq. (4)). For instance, SG$_0$ in EG$_1$ is associated with C_4 and is able to generate linear terms for the stream bits $s_{4,0}, \ldots s_{4,113}$. Each EG also contains another type of component, the Stream Combiner, that takes care of adding the right stream bit terms and the right ciphertext bits together in order to get the final equations that are then passed to the LSE Solver.



α_{17}	0	0	0		0	0	0	0	0	0	0	1
α_{16}	0	0	0		0	0	0	0	0	0	1	0
α_{15}	0	0	0		0	0	0	0	0	1	0	0
α_{14}	0	0	0		0	0	0	1	0	0	0	0
α_{13}	0	0	0		0	0	1	0	0	0	0	0
α_{12}	0	0	0		0	1	0	0	0	0	0	0
α_{11}	0	0	0	$\cdots$	1	0	0	0	0	0	0	0
	0	0	0		0	0	0	0	0	0	0	0
$\vdots$	0	0	0		0	0	0	0	0	0	0	0
	0	0	0		0	0	0	0	0	0	0	0
α_2	0	0	1		0	0	0	0	0	0	0	0
α_1	0	1	0		0	0	0	0	0	0	0	0
α_0	1	0	0		0	0	0	0	0	0	0	0
consts	1	0	0		0	1	1	1	1	0	1	1

Fig. 6. Detailed view of R1 represented within a SG after initialization.

Stream Generator (SG). The SG unit consists of an A5/2 engine where the states of the LFSRs R1, R2 and R3 are represented by vectors of α_i's instead of single bits. We implemented the R*i*'s as *vector LFSRs* instead of standard scalar LFSRs to obtain linear expressions in the variables α_i (every LFSR binary addition and shift operation is applied to a linear combination of α_i's). Figure 6 shows an example of this representation for R1 right after the initialization phase. Each column vector gives the dependence of the corresponding bit of the simple LFSR on the α_i's and a constant (Eq. (1) describes exactly the same state of the vector LFSR). Hence the rows of the matrix indicate the dependence on the corresponding α_i while the columns indicate the position in the LFSR. The row at the bottom corresponds to the constants $\sigma_{h,i}$. Right after the initialization phase, each column (ignoring the bottom row) only contains a single 1, because before warm-up each position of each *Ri* depends only on a single α_i (cf. Eq. (1)). The only exception are the three positions in R1 through R3 that are set to one. Here the respective positions do not depend on any α_i but the respective constant part is 1. Note that no clock cycles need to be wasted to actually perform the initialization phase for vector LFSRs, since we can precalculate the IV's influence on each LFSR position

Each SG performs the warm-up phase where its vector LFSRs are clocked 99 times. Every time a vector LFSR is clocked (forward), all columns are shifted one position to the right, the last column is dropped and the first column is calculated as an XOR of the columns according to the feedback term. After warm-up, the CLU can query one of its 114 outputs. To produce this output, the SG is clocked as many times as necessary to reach the desired linear expression for $s_{h,k}$. An SG can be clocked forward as well as backward[3], resulting in an average of 36 clock cycles required for generating one output equation. The output is generated by XORing the result of the majority function as described in Eq. (4).

[3] Figure 6 depicts only the control logic for forward clocking. For simplicity reasons we also omitted certain other control and data signals in the figure.

The majority function performs a pairwise "multiplication" of all three input vectors and binary adds the intermediate results and the vector that directly enters the equation (e.g, R1[18] in Figure 6). The multiplication of two column vectors is done by binary multiplying each element of one vector with each element of the other vector. The resulting term for one key bit $s_{h,k}$ is linearized by assigning each quadratic variable to a new variable (represented by a "new" signal line). We implemented the multiplication of Eq. (4) to be performed in one clock cycle. Instead of rotating a number of registers several times, we directly tap and combine all input bits of these registers that are required for the computation of a specific output bit.

Since the domain of each instance of the majority function is restricted to one register, its action is local and one obtains three smaller quadratic equations with disjunct variables (except for the constant term) before the final XOR. That is, the results of the different vector LFSRs do not have to be XORed and can be directly output (the first 655 data lines). The only operation one has to perform to compute this XOR is to add the three constant bits (the last single-bit output of each SG). Note that one does to have to linearize the local quadratic equations, since we already use the linearized form to represent quadratic equations in hardware (each linear or quadratic term is represented by one bit).

Each of the 8 SGs accepts 3 control signals from the CLU indicating the clocking direction (forward or backward), the stop-go command (as the vector LFSRs belonging to different SGs need to be clocked a different number of times), and the initialization command (increment R4 and perform warm-up).

Stream Combiner (SC). The SC combines the results of the SGs with the right ciphertext bits from the CM to produce the equations for the LSE Solver. More precisely, it works as follows: the output of an SG are 656 signals representing a certain stream bit $s_{h,k}$. The signal representing the constant value c of $s_{h,k}$ (cf. Eq. (4)) is then XORed with the respective ciphertext bit $c_{h,k}$ provided by the CM. By having a closer look at Eq. (9) and the involved function f, we can see that this 656-bit result is sometimes needed for the generation of two consecutive equations. Moreover, note that sometimes also the current and the previous result of an SG are required at the same time to build an equation. To this end the result of an SG is buffered in the SC (see Figure 6). A signal of the CLU is used to decide which of the 8 previous and 8 current results are XORed together. The resulting equation is now passed as new equation to the buffer of the LSE Solver.

4.3 Ciphertext Module (CM)

The CM stores the 16 ciphertext blocks and IVs and provides them to the SCs and the KT in the required order. It consists of 24 memory blocks for storing ciphertexts and 16 memory blocks for storing the IVs. The content of the ciphertext memory blocks can be cyclicly shifted in both directions. The ciphertexts $C_0, \ldots, C_{15}$ are initially stored in the bit order as they are recorded from air. $C_0, \ldots, C_7$ is put in the first 8 memory blocks, $C_4, \ldots, C_{11}$ is put in the next 8 memory blocks and $C_8, \ldots, C_{15}$ is put in the last 8 blocks.

Each EG and the KT has parallel access to the 8 required IVs. Each of the SCs needs only access to 8 of the 24 ciphertext memory blocks. More precisely, the SC belonging to EG_i is provided with the first bit of memory block $4i + 0$ to $4i + 7$ respectively (i.e, the positions where the bits $c_{4i+j,0}$ are initially stored). The content of these memory blocks needs to be rotated in the same way as the vector LFSRs within the 8 SGs of EG_i. To this end the CM receives the same control signals from the CLU as the SGs. Finally, the KT accesses the first bit of the ciphertext memory blocks 0 to 7 respectively, i.e., the same bits as EG_0.

4.4 LSE Solver

The LSE Solver is controlled by the CLU. Each time an equation in an Equation Generator is ready, the LSE Solver obtains a signal for loading the equation. As all orders have been processed and all equations loaded into the LSE Solver, it receives a command from the CLU to start Gaussian elimination. When the LSE Solver is ready, it writes the 61-bit result into a buffer. After this it signals that a solution is ready. The CLU informs the Key Tester that the secret state candidate is in the buffer. It is then read out by the Key Tester. We decided to use the SMITH architecture presented in [5,6] to realize the LSE Solver module.

SMITH. The SMITH architecture implements a hardware-optimized variant of the Gauss-Jordan[4] algorithm over GF(2). The architecture is described in [5] for LSEs of dimension $m \times n$ where $m \geq n$. Its average running time for systems of dimension $n \times n$ with uniformly distributed coefficients is about $2n$ (clock cycles) as opposed to about $\frac{1}{4}n^3$ in software.

Though SMITH was not originally designed for performing Gauss-Jordan on underdetermined LSEs ($m < n$), using minor modifications it can also handle this type of LSEs which is required for our purposes. Due to page limitations, we omit describing these straightforward adaptions. In the remainder of this section, we sketch a simple enhancement of the architecture that significantly reduces the total number of clock cycles (in our case) for doing Gauss-Jordan elimination.

The required number of clock cycles is determined by two operations which are applied $\min(m, n) = m$ times: pivoting and elimination. Pivoting roughly means the search for a non-zero element in a certain column of the respective coefficient matrix which is then used during the elimination operation to zero-out all other “1” entries in this column. While the elimination operations consume a fixed amount of m clock cycles in total, a variable number of clock cycles for pivoting is required. This number highly depends on the characteristics of the

[4] In this version of Gaussian elimination the backward substitution steps are combined with the elimination steps. In this way one immediately obtains the solution vector without doing any post-processing.

coefficient matrix. Roughly speaking, if the matrix columns contain many zero entries pivoting requires many clock cycles and may dominate the total costs. In our case, we initially have dense matrices that however contain many linearly dependent row vectors resulting in many zero columns while the algorithm proceeds. More precisely, our experiments show that each of our matrices contains about 160 to 190 linearly dependent equations. It is important to note that if the column, the pivoting operation is applied to, is a zero column, no pivoting and no subsequent elimination is actually required. Thus, by performing zero column detection (in an efficient manner), we can save these clock cycles and proceed immediately with the next column. Since, in the case of the SMITH architecture the logic for pivoting is physically located in a *single* column (the 1st column), we could efficiently realize this detection by computing the OR over all entries of this single column. Using this simple adaption, the pivoting operations for the whole matrix consume about 4000-4500 cycles (instead of more than 60000). Thus, a solution is computed after about 5000 cycles.

4.5 Key Tester (KT)

The KT receives the output of the LSE Solver, i.e., the secret state candidate and checks its correctness. The KT is built-up as a modified EG. The determined candidate is written into the SG-engines of the KT, which are normal A5/2 engines that can be clocked in both directions. Hence the size of this modified SGs is much smaller and they produce single bits as output. For the verification of a candidate, the output bits $s_{h,k}$ generated by the SGs are combined with the ciphertext bits according to Eq. (9) like it is done within a regular EG. If all resulting XOR-sums are equal to 0, the correct secret state has been found and is written out.

4.6 Control Logic Unit (CLU)

The CLU controls all other components and manages the data flow. It ensures that the right stream bit expressions are generated, combined with the ciphertext bits and passed to the LSE Solver. Once the LSE Solver is filled with 555 equations, it stops the EGs and starts the LSE Solver. When the LSE is solved, the candidate is passed to the KT. The KT is operated in parallel to the generation of the new LSE.

The CLU generates the same sequence of 24-bit control signals for each of the three EGs and the KT (cf. Figure 5). Remember that each SG belonging to an EG receives 3 of these signals. They determine the direction in which the engine is clocked, whether the engine is clocked at all and whether the R4 register need to be increased (to generate equations for a new candidate). The generation of these signals can be immediately derived from Eq. (9). More precisely, the CLU internally generates an *order* for each of the 185 equations an EG should produce. From such an order the required signals can be easily derived. The orders for the first 21 equations are shown in Table 1. Each order thereby consists of 7 pairs (fr_i, cl_i) $(0 \leq i \leq 6)$. The number fr_i equals the index

Table 1. Orders required for the first 21 equations ($(j, 0)$ means that the current output of Stream Generator j is needed).

Equation	**Orders**						
0	(0, 0),	(0, 100),	(1, 98),	(1, -14),	(2, 82),	(3, 66),	(6, 19)
1	(0, 0),	(2, 0),	(2, -14),	(3, 0),	(3, -14),	(4, 51),	(5, 35)
2	(2, 0),	(4, 0),	(4, -14),	(5, 0),	(5, -14),	(6, 0),	(7, 3)
3	(0, 0),	(1, 0),	(4, 0),	(6, 0),	(6, -14),	(7, 0),	(7, 100)
4	(0, 0),	(0, -14),	(1, 0),	(1, -14),	(2, 0),	(3, 0),	(6, 0)
5	(0, 0),	(2, 0),	(2, -14),	(3, 0),	(3, -14),	(4, 0),	(5, 0)
6	(2, 0),	(4, 0),	(4, -14),	(5, 0),	(5, -14),	(6, 0),	(7, 0)
7	(0, 0),	(1, 0),	(4, 0),	(6, 0),	(6, 100),	(7, 0),	(7, -14)
8	(0, 0),	(0, -14),	(1, 0),	(1, -14),	(2, 0),	(3, 0),	(6, 0)
9	(0, 0),	(2, 0),	(2, -14),	(3, 0),	(3, -14),	(4, 0),	(5, 0)
10	(2, 0),	(4, 0),	(4, -14),	(5, 0),	(5, 100),	(6, 0),	(7, 0)
11	(0, 0),	(1, 0),	(4, 0),	(6, 0),	(6, -14),	(7, 0),	(7, -14)
12	(0, 0),	(0, -14),	(1, 0),	(1, -14),	(2, 0),	(3, 0),	(6, 0)
13	(0, 0),	(2, 0),	(2, -14),	(3, 0),	(3, -14),	(4, 0),	(5, 0)
14	(2, 0),	(4, 0),	(4, 100),	(5, 0),	(5, -14),	(6, 0),	(7, 0)
15	(0, 0),	(1, 0),	(4, 0),	(6, 0),	(6, -14),	(7, 0),	(7, -14)
16	(0, 0),	(0, -14),	(1, 0),	(1, -14),	(2, 0),	(3, 0),	(6, 0)
17	(0, 0),	(2, 0),	(2, -14),	(3, 0),	(3, 100),	(4, 0),	(5, 0)
18	(2, 0),	(4, 0),	(4, -14),	(5, 0),	(5, -14),	(6, 0),	(7, 0)
19	(0, 0),	(1, 0),	(4, 0),	(6, 0),	(6, -14),	(7, 0),	(7, -14)
20	(0, 0),	(0, -14),	(1, 0),	(1, -14),	(2, 0),	(3, 0),	(6, 0)

of a ciphertext/key-stream block modulo 8 (cf. Eq. (9)) required in an equation. So this number addresses one of the 8 SGs belonging to an EG. The number cl_i (which can be negative) is the relative position of the required bit within the fr_i-th ciphertext/key-stream block. "Relative" means that this position is given relatively to the position of the bit of this block that was required just before. This number can be used to signal how often and in which direction an SG should be clocked. Considering the columns of Table 1, we see that these pairs occur (almost) periodically. So orders can be generated easily in hardware.

Besides the three control signals for each SG, the CLU has to produce a 16-bit mask to control which outputs of the SGs are XORed within an SC (cf. Figure 5). As can be derived from Table 1, only 7 bits of the mask are simultaneously set to 1. Finally, the CLU also "orders" the needed ciphertext bits from the CM which is done in the same way as stream bits are "ordered" from the SGs.

Operating Procedure. During the setup of our attack engine, all components are being initialized and 16 ciphertext frames and 16 corresponding IVs are read into the CM. The R4 registers of the GEs are set to the initialization value 0.

After the initialization the equations are generated, solved, and tested for all different possible states of R4, until the right state is found. Hence the following steps are performed 2^{15} times on average:

1. The registers R4 are incremented and the warm-up is executed in the SGs and in the KT. The SGs are now ready to generate the linearized terms for the stream bits $s_{h,k}$ when queried.
2. The LSE Solver gets filled with 555 equations. The CLU queries each of the three EGs 185 times to receive these equations. The CLU plays an important

role in this, because it controls each SG to provide the right $s_{h,k}$ terms, which are then combined by the SCs and passed to the buffer of the LSE Solver. The SGs inside the EGs need to be clocked 36 times on average to produce the necessary terms.

3. Once all equations are generated, the LSE Solver is started. It takes roughly 5000 cycles until the result is calculated.
4. The determined candidate is fed into the KT and the warm-up is executed.
5. The CLU queries the KT 185 times to generate the output bits. If all parity checks in the KT succeed, the recovered 77-bit state is passed to the output.

Since the KT and the EGs have the same components the warm-up and equation generation of both can be performed in parallel. Hence, steps 1 and 4 as well as 2 and 5 are performed in parallel. Furthermore, setup and warm-up (steps 1 & 4) for the new state candidate can be performed while the LSE Solver is determining the previous candidate (step 3).

5 Implementation Results and ASIC Estimates

Due to the size of the architecture, an ASIC realization seems most realistic. We decided to keep the operating speed at 256 MHz for the LSE Solver to maintain a decent power consumption at still reasonable performance time. Since the remaining components are smaller than the LSE Solver and there are periods where those components are idle, they are clocked at twice the speed (512MHz). This way the LSE Solver still accounts for two thirds of the overall power consumption and heat development. At these clock rates one key is recovered on average in about 1 second.

To evaluate the requirements on chip size and average power consumption, we implemented our design in VHDL and synthesized it using the Virtual Silicon (VST) standard cell library based on the UMC L180 0.18μ 1P6M Logic process. We used Synopsys Design Compiler version Y-2006.06 for synthesis and estimation of power consumption. Mentor Graphics Modelsim SE was used for simulation. Due to the huge size of the whole system, simulation and synthesis were done component-wise. A synthesis of the whole design should further decrease the needed area.

All critical components were implemented and synthesized. Table 2 shows the synthesis results. For each component the area it needs is given as well as the consumed average power. The area is given in gate equivalents (GE). One GE is equal to the area needed by one NAND-gate in the appropriate process. Power is given in μW. The first column shows at which clock frequency a component is operated. For few uncritical components like the Stream Combiner and the Ciphertext Module module, area and power consumption were estimated rather than synthesized. Conservative assumptions on the needed number of flip-flops and area for control-logic were transformed into area and power estimations. Estimated components are indicated by *.

The last value of Table 2 shows the estimated sum for both area and power consumption of a realization of the whole design. Obviously the LSE Solver

Table 2. Simulation results sorted by components used (* indicates estimated values)

Component Name	**Clock Speed** [MHz]	**Area** [kGE]	**Power Consumption** [mW]
Stream Generator (SG)	512	28.9	129.9
Key Tester (KT)	512	0.7	2.9
LSE Solver	256	8,205.3	8,360.8
Stream Combiner* (SC)	512	95.5	431.8
Ciphertext Module* (CM)	512	16.6	27.3
Control Logic* (CLU)	512	4.6	20.1
Full Design*	256/512	9,316.8	12,833.7

accounts for the biggest part of the area needed, almost 90% of the total area of the design. Yet the sum of the other components account for roughly one third of the total power consumption of 12.8 W. This is due to the higher operating frequency of these components. Note that many components appear multiple times in the design. I.e. the EG appears 3 times, resulting in 24 SGs.

The full design needs roughly 9.3 million GEs and consumes roughly 12.8 Watts of energy. For comparison, a recent desktop PC CPU, the Core 2 Duo "Conroe" processor, accumulates roughly 2.9×10^8 transistors (0.065 microns) and draws up to 65 Watts [8]. So we used less than 15% of the area and less than one third of the power.

References

1. E. Barkan and E. Biham. Conditional estimatores: An Effective Attack on A5/1. In *Proc. of SAC'05*, volume 3897 of *LNCS*. Springer-Verlag, 2006.
2. E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communications. In *Proc. of Crypto 2003*, volume 2729 of *LNCS*. Springer-Verlag, 2003.
3. E. Biham and O. Dunkelman. Cryptanalysis of the A5/1 GSM Stream Cipher. In *Proc. of Indocrypt'00*, volume 1977 of *LNCS*. Springer-Verlag, 2000.
4. A. Biryukov, A. Shamir, and D. Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Proc. of FSE'00*, volume 1978 of *LNCS*. Springer-Verlag, 2001.
5. A. Bogdanov, M. Mertens, C. Paar, J. Pelzl, and A. Rupp. A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *Proc. of FCCM'06*, pages 237–248. IEEE Computer Society, 2006.
6. A. Bogdanov, M. Mertens, C. Paar, J. Pelzl, and A. Rupp. SMITH - a Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS 2006), Conference Records*, 2006.
7. M. Briceno, I. Goldberg, and D. Wagner. A Pedagogical Implementation of the GSM A5/1 and A5/2 "voice privacy" Encryption Algorithms. `http://cryptome.org/gsm-a512.html`, 1999.
8. Intel Corporation. Intel Unveils World's Best Processor. Press Release, July 27 2006.
9. P. Ekdahl and T. Johansson. Another Attack on A5/1. *IEEE Transactions on Information Theory*, 49(1):284–289, 2003.
10. I. Goldberg, D. Wagner, and L. Green. The Real-Time Cryptanalysis of A5/2. Presented at the Rump Session of Crypto'99, 1999.
11. J. Golic. Cryptanalysis of Alleged A5 Stream Cipher. In *Proc. of Eurocrypt'97*, volume 1233 of *LNCS*, pages 239–255, 1997.
12. B. Hochet, P. Quintin, and Y. Robert. Systolic Gaussian Elimination Over GF(p) with Partial Pivoting. *IEEE Trans. Comput.*, 38(9):1321–1324, 1989.
13. European Telecommunications Standards Institute. Digital Cellular Telecommunications System (Phase 2+); Channel Coding (GSM 05.03 Version 8.5.1 Release 1999). `http://www.etsi.org`, 1999.
14. A. Maximov, T. Johansson, and S. Babbage. An Improved Correlation Attack on A5/1. In *Proc. of SAC'04*, volume 3357 of *LNCS*, pages 239–255. Springer-Verlag, 2005.
15. S. Petrovic and A. Fuster-Sabater. Cryptanalysis of the A5/2 Algorithm. IACR ePrint Report 200/52, `http://eprint.iacr.org`, 2000.
16. T. Pornin and J. Stern. Software-hardware Trade-offs: Application to A5/1 Cryptanalysis. In *Proc. of CHES'00*, volume 1965 of *LNCS*, pages 318–327. Springer-Verlag, 2000.