A Real-World Attack Breaking A5/1 within Hours

Timo Gendrullis, Martin Novotný, Andy Rupp

Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany.
{gendrullis, arupp}@crypto.rub.de, novotnym@fel.cvut.cz

Abstract. In this paper we present a real-world hardware-assisted attack on the well-known A5/1 stream cipher which is (still) used to secure GSM communication in most countries all over the world. During the last ten years A5/1 has been intensively analyzed [1,2,3,4,5,6,7]. However, most of the proposed attacks are just of theoretical interest since they lack from practicability — due to strong preconditions, high computational demands and/or huge storage requirements — or have never been fully implemented.
In contrast to these attacks, our attack which is based on the work by Keller and Seitz [8] is running on an existing special-purpose hardware device, called COPACOBANA [9]. With the knowledge of only 64 bits of keystream the machine is able to reveal the corresponding internal 64-bit state of the cipher in about 6 hours on average. We provide a detailed description of our attack architecture as well as implementation results.

Keywords. A5/1, GSM, special-purpose hardware, COPACOBANA.

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. A5/1 and A5/2 are the two encryption algorithms stipulated by this standard, where the stream cipher A5/1 is used within Europe and most other countries. A5/2 is the intentionally weaker version of A5/1 which has been developed — due to the export restrictions — for deploying GSM outside of Europe. Though the internals of both ciphers were kept secret, their designs were disclosed in 1999 by means of reverse engineering [10]. In this work we focus on the stronger GSM cipher A5/1.

1.1 The A5/1 Stream Cipher

A5/1 is a synchronous stream cipher accepting a 64-bit session key $K_S = (k_0, \ldots, k_{63}) \in GF(2)^{64}$ and a 22-bit initial vector $IV = (v_0, \ldots, v_{21}) \in GF(2)^{22}$ derived from the 22-bit frame number which is publicly known. It uses three

linear feedback shift registers (LFSRs) $R1$, $R2$, and $R3$ of lengths 19, 22, and 23 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 values of the clocking bits (CBs) which are bits 8, 10, and 10 of registers $R1$, $R2$, and $R3$, respectively.

The A5/1 keystream generator works as follows. First, an *initialization phase* is run. At the beginning of this phase all registers are set to 0. Then the *key setup* and the *IV setup* are performed. In the initialization phase all three registers are clocked regularly and the key bits followed by IV bits are xored with the least significant bits of all three registers. Thus, after $64 + 22 = 86$ clock-cycles the state S^i is achieved.

Based on this initial state S^i the *warm-up phase* is performed where the generator is clocked for 100 clock-cycles and the output is discarded. This results directly in the state S^w producing the first output bit 101 clock-cycles after the initialization phase. Note that already during the warm-up 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 $R1[8]$, $R2[10]$, and $R3[10]$ 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 $maj(a, b, c) = ab + ac + bc$. $R1$ is clocked iff $R1[8]$ agrees with the majority. $R2$ is clocked iff $R2[10]$ agrees with the majority. $R3$ is clocked iff $R3[10]$ agrees with the majority. Regarding to Table 1 in each cycle at least two of the three registers are clocked. After these clockings, an output bit is generated from the values of $R1$, $R2$, and $R3$ by xoring their most significant bits.

Table 1. Clockcontrol of *A5/1*

CB of $R1$: $R1[8]$	0	0	0	1	0	1	1	1
CB of $R2$: $R2[10]$	0	0	1	0	1	0	1	1
CB of $R3$: $R3[10]$	0	1	0	0	1	1	0	1
Majority	0	0	0	0	1	1	1	1
Clock $R1$?	√	√	√	–	–	√	√	√
Clock $R2$?	√	√	–	√	√	–	√	√
Clock $R3$?	√	–	√	√	√	√	–	√

After warm-up A5/1 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 assume that we are given at least 64 consecutive bits of such a 228 bit keystream.

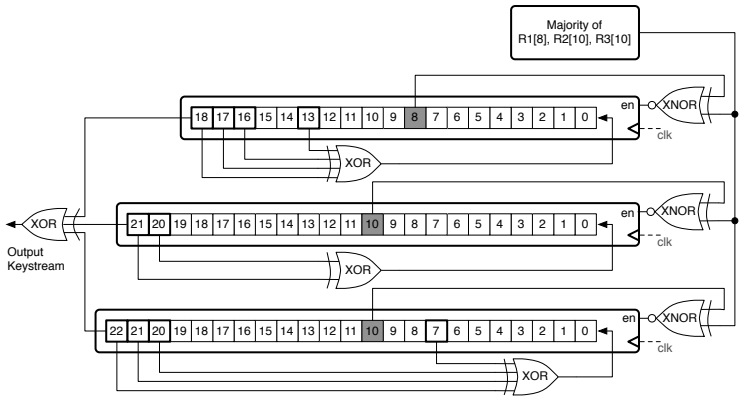

Fig. 1. Design of A5/1

1.2 Related Work

During the last decade the security of A5/1 has been extensively analyzed. Pioneering work in this field was done by Anderson [11], Golic [5], and Babbage [12].

Anderson's basic idea was to guess the complete content of the registers $R1$ and $R2$ and about half of the register $R3$. In this way the clocking of all three registers is determined and the second half of $R3$ can be derived given 64 bits of keystream. In the worst-case each of the 2^{52} determined state candidates (i.e., candidates for S^w) needs to be verified against the keystream which imposes a high workload when done in software.

The hardware-assisted attack by Keller and Seitz [8] is based on Anderson's idea. However, they proposed a way to exclude a significant fraction of possible candidates at a very early stage of the verification process. The authors claim that their approach reduces the attack complexity to $2^{41} \cdot (\frac{3}{2})^{11}$ with an expected computing time of 14 clock-cycles per guess. This results in a worst-case complexity of $2^{51.24}$ clock cycles. They implemented the attack on a Xilinx XC4062 FPGA. The FPGA is hosting seven instances of the guessing algorithm and operates at a frequency of 18.65 MHz leading to an attack time of about 236 days. Unfortunately, the approach given in [8] does not only immediately discard wrong candidates but a priori *restricts* the search for candidates to a certain subspace. This fact is not explicitly mentioned in the paper. Moreover, no complete analysis of the attack is given. Our analyses in Section 2 show that the success probability of their attack is only about 18% and the expected computing time for a guess is slightly higher than the stated one.

The key idea of Golic's attack [5] is to guess the lower half of each register (these bits determine the register clocking in the first few clock-cycles) and clock the cipher until the guessed bits "run-out". Each output bit immediately yields a linear equation in terms of the internal state bits belonging to the upper halves of three registers. Then we continue guessing the clocking sequence yielding

again other linear equations that describe the output of the majority function. Whenever 64 linearly independent equations are obtained in this way the system is solved using Gaussian elimination. The complexity of this attack is $O(2^{40})$ steps. However, each step is fairly complex since it comprises to compute the solution of an 64×64 LSE (and the verification of the corresponding state candidate).

Pornin and Stern proposed a SW/HW tradeoff attack [7] that is based on Golic's approach but in contrast to Golic they are guessing the clocking sequence from the very first step, similarly to [13]. These guesses create a tree with 4 branches in each node (each branch represents one clocking combination, cf. Table 1). While traversing a path down the tree, three equations are obtained at each node (similarly to the second phase of Golic's method), namely two equations describing the clocking and one equation describing the output. Hence, after n steps (in depth) one collected $3n$ equations. The tradeoff parameter n is chosen such that $3n < 64$. Thus, each path in the tree leads to an underdetermined LSE that is solved in software resulting in a parametric solution on the internal state. The basis of the corresponding linear subspace containing all solutions to such an LSE consists of $(64 - 3n + 1)$ 64-bit vectors. These vectors are sent to the hardware, where a brute force attack is performed, i.e., each of the 2^{64-3n} elements of the subspace is generated and loaded to the A5/1 instance. The instance is run after each load to verify the obtained output keystream against the given keystream. The authors estimated an average running time of 2.5 days when using an XP-1000 Alpha station for the software part and two Pamettes 4010E for the hardware part of the attack (where $n = 18$).

The authors consider to place twelve A5/1 instances into one Xilinx 4010E FPGA, occupying $12 \times 36 = 432$ CLBs out of 576 (75% of the FPGA). Unfortunately, any details (especially the area) of the unit generating 2^{64-3n} internal states are missing which makes it hard to verify the stated figures. However, these figures do not seem to be based on real measurements and we consider them as too optimistic; we expect that the generator unit occupies a relatively large area. For instance, when choosing $n = 18$ the transmitted basis consists of 11 vectors, i.e., $11 \times 64 = 704$ bits. Since the deployed Xilinx 4010E FPGA contains only 1152 flip-flops, more than 60% of them would be used just for holding the coefficients of the basis. So there seems not to be enough space to place twelve A5/1 units (needing further $12 \times 64 = 768$ flip-flops) on the FPGA as stated in the paper.

Finally, there is a whole class of time-memory-data tradeoff (TMDTO) attacks on A5/1 which share the common feature that a large amount of known keystream must be available and/or huge amounts of data must be precomputed and stored in order to achieve reasonable success rates and workloads for the online phase of these attacks. Simple forms of such attacks have been independently proposed by Babbage [12] and Golic [5]. Recently, Biryukov, Shamir, and Wagner presented an interesting (non-generic) variant of an TMDTO [3] (see also [14]) utilizing a certain property of A5/1 (low sampling resistance). The precomputation phase of this attack exhibits a complexity of 2^{48} and memory

requirements of only about 300 GB, where the online phase can be executed within minutes with a success probability of 60%. However, 2 seconds of known keystream (i.e., about 25000 bits) are required to mount the attack making it impractical. Another important contribution in this field is due to Barkan, Biham, and Keller [15] (see also [16]). They exploit the fact that GSM employs error correction before encryption — which reveals the values of certain linear combinations of stream bits by observing the ciphertext — to mount a ciphertext-only TMDTO. However, in the precomputation phase of such an attack huge amounts of data need to be computed and stored; even more than for known-keystream TMDTOs. For instance, if we assume that 3 minutes of ciphertext (from the GSM SACCH channel) are available in the online phase, one needs to precompute about 50 TB of data to achieve a success probability of about 60% (cf. [16]). There are 2800 contemporary PCs required to perform the precomputation within one year. These are practical obstacles making actual implementations of such attacks very difficult. In fact, to the best of our knowledge no full implementation of TMDTO attack against A5/1 has been reported yet.

1.3 Our Contribution

As seen in the previous section most of the proposed attacks against A5/1 lack from practicability and/or have never been fully implemented. In contrast to these attacks, we present a real-world attack revealing the internal state of A5/1 in about 6 hours on average (and about 12 hours in the worst-case) using an existing low-cost (about US$ 10,000) special-purpose hardware device. To mount the attack only 64 consecutive bits of a known keystream are required and we do not need any precomputed data. Also the communication requirements with the host computer are relatively small.

On the theoretical side, we present a modification and analysis of the approach sketched in [8]. Furthermore, we propose an optimization of the attack implementation leading to an improvement of about 13% in computation time compared to a plain implementation. Both plain and optimized version of the attack have been fully implemented and tested on our target platform.

1.4 Implementation Platform

The COPACOBANA (Cost-Optimized Parallel Code Breaker) machine [9] is a high-performance, low-cost cluster consisting of 120 Xilinx Spartan3-XC3S1000 FPGAs. Currently, COPACOBANA appears to be the only such reconfigurable parallel FPGA machine optimized for code breaking tasks reported in the open literature. Depending on the actual algorithm, the parallel hardware architecture can outperform conventional computers by several orders of magnitude. COPACOBANA has been designed under the assumptions that (i) computationally costly operations are parallelizable, (ii) parallel instances have only a very limited need to communicate with each other, (iii) the demand for data transfers

between host and nodes is low due to the fact that computations usually dominate communication requirements and (iv) typical crypto algorithms and their corresponding hardware nodes demand very little local memory which can be provided by the on-chip RAM modules of an FPGA. Considering these characteristics COPACOBANA appeared to be perfectly tailored for simple guess-and-determine attacks on A5/1 like the one described in the next section.

2 Analysis and Modification of Keller and Seitz's Approach

The approach is based on a simple guess-and-determine attack proposed by R. Anderson in 1994 where the shorter registers $R1$ and $R2$ are guessed and the longer register $R3$ is to be determined. But because Anderson neglected the asynchronous clocking of the registers at first, only the 12 most significant bits of $R3$ can be determined from the known keystream while the remaining bits have to be guessed as well.

Keller and Seitz's attack can be divided into two phases, into the *determination phase* in which a possible state candidate consisting of the three registers of A5/1 after its warm-up phase is generated and into a subsequent *postprocessing phase* in which the state candidate is checked for consistency.

2.1 Analysis

In the determination phase, Keller and Seitz try to reduce the complexity of the simple guess-and-determine attack further by early recognizing contradictions that can occur by guessing the clocking bit (CB) of $R3$ such that $R3$ will not be clocked. Therefore, they first completely guess the registers $R1$ and $R2$ and then derive register $R3$ in the following manner. Let $Ri^{(t)}[n]$ denote the n-th bit of register Ri at a time t, where $t = 0$ is immediately after the warm-up phase of A5/1 and increases by 1 every clock-cycle. Then, foremost compute the first most significant bit (MSB) of $R3$, which is $R3^{(0)}[22]$, immediately out of $R1^{(0)}[18]$ and $R2^{(0)}[21]$ and the first bit of the known keystream (KS). Then inspect the clocking bits of registers $R1$ and $R2$, which are $R1^{(0)}[8]$ and $R2^{(0)}[10]$, and guess the first clocking bit of $R3$, namely $R3^{(0)}[10]$. If $R1^{(0)}[8]$ and $R2^{(0)}[10]$ are not equal, $R3$ will be clocked in either way and so both possibilities for $R3^{(0)}[10]$ have to be checked. But if the CBs of $R1$ and $R2$ are identical then at least these two registers will be clocked. Assume now the CB of $R3$ is chosen to be different from the ones of $R1$ and $R2$, i.e., $R3^{(0)}[10] \neq R1^{(0)}[8]$, and as a consequence $R3$ will not be clocked. Now in one half of these cases the generated output bit of the MSBs of all three registers (which are $R1^{(1)}[18] = R1^{(0)}[17]$, $R2^{(1)}[21] = R2^{(0)}[20]$, $R3^{(1)}[22] = R3^{(0)}[22]$) does not match the given keystream bit and a contradiction occurs. As a consequence the CB of $R3$ has to be guessed in a way that $R3$ will be clocked together with $R1$ and $R2$, i.e., the CB of $R3$ is to be chosen equal to the CBs of $R1$ and $R2$, so that a new MSB can be computed.

By early recognizing this possible contradiction while guessing $R3^{(t)}[10]$, all arising states of this contradictory guess neither need to be computed further on nor checked afterwards. To further reduce the complexity of the attack they do not only discard these described wrong possibilities for the CB of $R3$ in case of a contradiction but they also limit the number of choices to the one of not-clocking $R3$ if this is possible without any contradiction. After having computed the first MSB of $R3$ the process of guessing a CB and computing another MSB of $R3$ is repeated until $R3$ is completely determined which is after having clocked $R3$ for 11 times.

This heuristic reduces the number of possibilities for $R3^{(t)}[10]$ in one half of all cases from two to one. The number of possible state candidates to be checked decreases thus from 2^{11} to $(2-\frac{1}{2})^{11} = (\frac{3}{2})^{11} \approx 2^{6.43} \approx 86$ for every fixed guess of registers $R1$ and $R2$ in general. This results in $2^{41} \cdot 2^{6.43} = 2^{47.43}$ possible state candidates. But because they discard some valid states as well as states leading to a contradiction they have only a low success probability. The number of all valid state candidates for one fixed guess of $R1$ and $R2$ is $(2-\frac{1}{4})^{11} = (\frac{7}{4})^{11} \approx 2^{8.88} \approx 471$. Thus, the number of state candidates inspected by Keller and Seitz in proportion to the number of valid state candidates results in a success probability of only $\frac{86}{471} \approx 0.18 = 18\%$.

Immediately after the determination phase, the A5/1 is performed with the generated state candidate in the postprocessing phase and the generated output bits are checked against the remaining bits of the 64 bit known keystream. Keller and Seitz just state that this consistency check in the postprocessing phase will proceed fast and that both, determining a state candidate and checking it against the known keystream, will take $14 \approx 2^{3.81}$ clock-cycles. This leads to a complexity of $2^{47.43} \cdot 2^{3.81} = 2^{51.24}$ clock-cycles. But with this expected amount of clock-cycles they underestimated the time complexity as will be shown in Section 2.2.

One instance of Keller and Seitz's guessing algorithm occupies 313 out of the 2304 configurable logic blocks (CLBs) of the XC4062 FPGA. It is hard to estimate how fast the original Keller-Seitz attack would be when implemented on COPACOBANA, since the architecture and the performance of the XC4062 [17] and the Spartan-3 XC3S1000 [18] FPGAs are different. For example, one XC4000 CLB only roughly corresponds to one Spartan-3 slice, because it contains two 4-input look-up tables (LUT), one 3-input LUT and two flip-flops (FF), while a Spartan-3 slice contains only two 4-input LUTs and two FFs. Because the available number of slices on a Spartan-3 XC3S1000 FPGA is 7680 and if we assume that one instance of the guessing algorithm would occupy 313 slices, a maximum number of 24 instances could be implemented on one FPGA. This leaves just 168 slices for other circuits for controlling the instances. According to the datasheets the "internal performance of XC4000 family chips can exceed 150 MHz" while the "maximum toggle frequency of Spartan-3 chips is 630 MHz". That represents a performance ratio of less than 4.2. Out of these figures we estimate that the attack would not be faster than $\frac{24}{7} \times 4.2 \times 120 = 1728$ times when run on COPACOBANA. This yields to a minimum of 3.27 hours to perform the search of Keller and Seitz. But if we recall again that (i) the attack searches

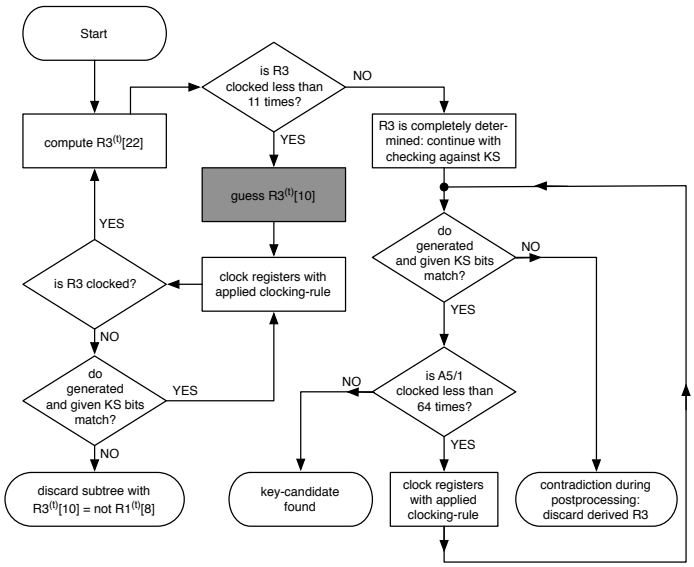

Fig. 2. Flowchart of the FSM of a guessing-engine

only through 18% of the valid states, the search through all valid states would take at least 18.19 hours, (ii) the number of guessing instances implemented in one FPGA would be less than 24 since at least an additional control logic has to be implemented, and (iii) Keller and Seitz underestimate the time complexity as will be shown in Section 2.2, the computation time is expected to increase significantly.

2.2 A Slight Modification

Our algorithm is similar to the one proposed by Keller and Seitz except that we only discard wrong possibilities for $R3^{(t)}[10]$ that would immediately lead into a contradiction. But if no contradiction appears we still check both possibilities for $R3^{(t)}[10]$, which means clocking and not-clocking $R3$. Because of this, we take every possible state candidate into account and therefore will find unlike Keller and Seitz the correct state candidate in any case. This reduces only in $\frac{1}{4}$ of all cases the number of choices from two to one and, hence, the expected number of possibilities for $R3$ that need to be checked is approximately 471 for every fixed guess of registers $R1$ and $R2$ (cf. Section 2.1).

A flowchart of the decisions during the determination phase and the post-processing phase shows Figure 2. A more detailed overview of how $R3^{(t)}[10]$ is guessed and how certain subtrees are discarded is given in Figure 3.

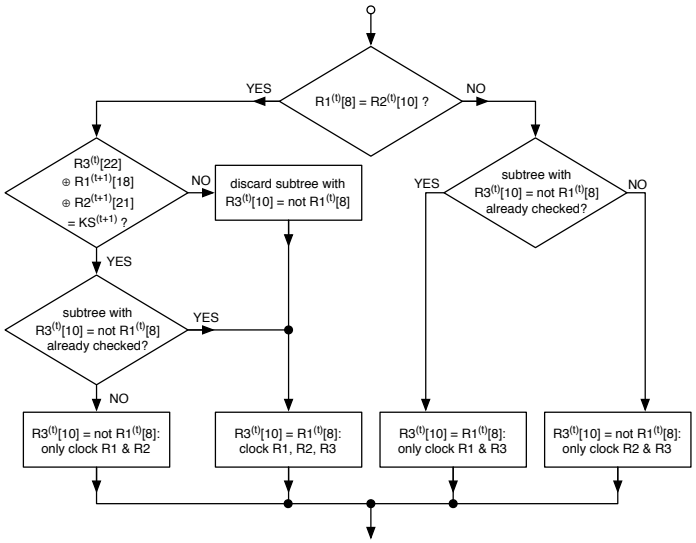

Fig. 3. Guessing the clocking bit of $R3$ in detail

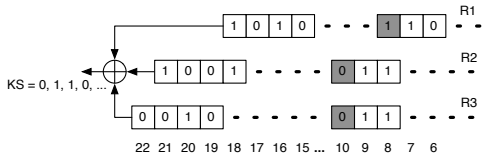

Fig. 4. An example for a generated state candidate after 3 times guessing $R3^{(t)}[10]$

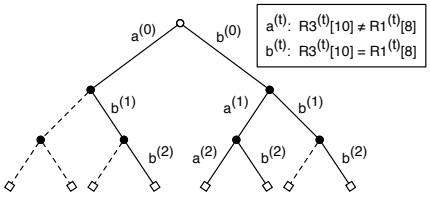

Fig. 5. An example for a reduced binary decision tree of $R3^{(t)}[10]$

Example. An example for the first steps of the reduction of possibilities performed by the algorithm is given in Figure 4. It shows next to the first 4 bits of a known keystream the first 4 MSBs and the first 3 CBs of the guessed registers $R1$ and $R2$ and of the derived register $R3$. The algorithm proceeds as follows.

1. Compute $R3^{(0)}[22] = R1^{(0)}[18] \oplus R2^{(0)}[21] \oplus KS[0] = 0$.
2. $R1^{(0)}[8] \neq R2^{(0)}[10]$: Choose $R3^{(0)}[10] = 0 \neq R1^{(0)}[8]$ first and clock registers $R2$ and $R3$.
3. Compute $R3^{(1)}[22] = R3^{(0)}[21] = R1^{(0)}[18] \oplus R2^{(0)}[20] \oplus KS[1] = 0$.
4. $R1^{(0)}[8] = R2^{(0)}[9]$: Not clocking register $R3$ would result in a contradiction because $R1^{(0)}[17] \oplus R2^{(0)}[19] \oplus R3^{(0)}[21] \neq KS[2]$.
   Hence, discard the possibility $R3^{(1)}[10] = 0 = R3^{(0)}[9] \neq R1^{(1)}[8]$, instead choose $R3^{(1)}[10] = 1 = R3^{(0)}[9] = R1^{(0)}[8]$, and clock all registers $R1$, $R2$, $R3$.
5. Compute $R3^{(2)}[22] = R3^{(0)}[20] = R1^{(0)}[17] \oplus R2^{(0)}[19] \oplus KS[2] = 1$.

6. ...

The example ends here because it is apparent from Figure 5, which shows the binary decision tree for $R3^{(t)}[10]$ up to a depth of 3, that discarding possibilities for $R3^{(t)}[10]$ results in cutting whole subtrees. In the example above we chose edge $a^{(0)} = R3^{(0)}[10] = 0 \neq R1^{(1)}[8]$ at the root node first and then discarded the possibility $a^{(1)} = R3^{(1)}[10] = 0 \neq R1^{(1)}[8]$ at the corresponding node of depth 1.

Time Complexity of the Attack. Generating one possible state candidate during determination phase takes one clock-cycle for deriving $R3^{(0)}[22]$ and then eleven times clocking register $R3$ to determine the remaining MSBs of the register. With a probability of $P_{clk} = \frac{3}{4}$ for clocking a register of A5/1 it takes an expected number of $1 + \frac{4}{3} \cdot 11 = 15\frac{2}{3}$ clock-cycles to generate the state candidate for fixed registers $R1$ and $R2$ and the known keystream. Because every clock-cycle one bit of the known keystream is inspected, the expected number of needed known keystream bits to generate a state candidate corresponds to the number of clock-cycles needed for this process.

After having generated one state candidate it needs to be checked in the postprocessing phase further on against the remaining bits of the known keystream. To be able to perform this check immediately after the determination phase we additionally compute the feedback bits of register $R3$ with its linear feedback function. We start with this computation from the time when $R3^{(3)}[10] = R3^{(0)}[7]$ is guessed. So we already computed 8 of the 11 feedback bits of $R3$ when the state candidate is generated. The remaining 3 feedback bits are computed in parallel and we continue with performing A5/1. Now, the produced output is compared to the known keystream. A contradiction between the generated output and a known keystream bit is expected to occur with a probability of $\alpha = \frac{1}{2}$ in the first clock-cycle of postprocessing. Every cycle the algorithm is clocked further on, the probability of a contradiction is again $\frac{1}{2}$. Generally speaking, it is $\alpha_n = \frac{1}{2^n}$ for the n-th cycle after the determination phase and the algorithm will clock on with an expected value of $\frac{1}{\alpha} = 2$ further needed clock-cycles to inspect the output. If it is clocked without any contradiction up to the 64-th bit of the known keystream we found a valid state candidate for reconstructing the session key. Although there might be more than just one state candidate generating the same 64 bit of output, the probability for this event is negligible.

So, we get an expected number of $T = 15\frac{2}{3} + 2 = 17\frac{2}{3}$ clock-cycles to determine a state candidate and check it for consistency with the given keystream instead of just 14 clock-cycles as stated by Keller and Seitz. Thus, the time complexity of our whole attack is $C \approx 2^{41} \cdot (\frac{7}{4})^{11} \cdot 17\frac{2}{3} \approx 2^{54.02}$.

3 Breaking A5/1 on COPACOBANA

3.1 Our Hardware Architecture

This section presents an efficient implementation of a *guessing-engine* in hardware which performs the determination phase and the postprocessing phase

of the attack. On every FPGA, several instances of this guessing-engine will be implemented. Therefore, we will additionally introduce a hardware-software-interface controlling these instances and providing intercommunication.

The Guessing-Engine. Figure 6 shows an overview of the guessing-engine with its different components. A large part of the architecture for implementing this guessing-engine consists of flip-flops (FFs) for storing the content of different registers. This is in detail the *state candidate register*, storing the computed register $R3$ and the fixed guess of registers $R1$ and $R2$ in 64 bits. Additionally, we need FFs to store the 64 bits of known keystream and an additional simple shift register to evaluate a different known keystream bit every clock-cycle. To perform the consistency check in the postprocessing phase, all three *A5/1 LFSRs* have to be implemented, too. But the most important part of this architecture is the finite state machine (*FSM*) performing the determination phase and the postprocessing phase. Its functionality was already presented in Figures 2 and 3. The shown process is repeated until all possible state candidates, i.e., the whole binary decision tree of $R3^{(t)}[10]$, for one fixed guess of registers $R1$ and $R2$ have been checked. The fact, that the guess $R3^{(t)}[10] \neq R1^{(t)}[8]$ is always checked first corresponds to the binary decision tree of Figure 5. This binary decision tree storing the discarded or already checked possibilities is mapped into the *branching state register*.

The most straightforward way of mapping such a binary decision tree with a certain height h into hardware, is to use an h-bit wide binary counter. In our case all leaves are at a depth of $d = h = 11$. Turning left at a node of the tree, i.e., $R3^{(t)}[10] \neq R1^{(t)}[8]$, is represented by 0 in the corresponding counter bit and turning right at a node, i.e., $R3^{(t)}[10] = R1^{(t)}[8]$, is represented by 1. Now, to reach all leaves from the leftmost unto the rightmost one by one, we initialize the 11-bit wide counter to all 0 and read it in 11 clock-cycles bit by bit from the most significant bit (MSB) to the least significant bit (LSB). When having reached the leftmost leaf in such a manner, we increase the register by one and restart reading bit by bit at the MSB again. This will lead us to the second leaf from the left. To reach the rest of the leaves we count through this 11-bit wide register up to all bits being 1. Now it is claimed by the attack that certain subtrees of the binary decision tree are discarded (cf. Section 2.2). To be able to do that while passing through the tree, we have to set the corresponding bits of the 11-bit wide counter manually to 1 with an 1-to-11 bit demultiplexer. The FSM does this with bit number b every time a contradiction is detected at a node of depth $d = b+1$ and a possibility of $R3^{(t)}[10]$ is discarded. This results in the reduced number of leaves of the binary decision tree of $(\frac{7}{4})^{11} \approx 471$ meaning the amount of possible state candidates for a fixed guess of $R1$ and $R2$.

The Control-Interface. Because several instances of the guessing-engine are implemented on one FPGA they need to be controlled continuously. This is done by the *control-interface* and there is exactly one instance of it implemented on each FPGA of COPACOBANA. It accepts the 64 bit known keystream and a

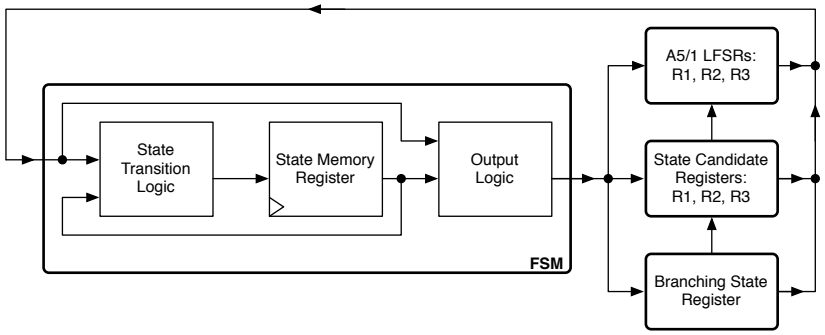

Fig. 6. An overview of the guessing-engine

sub-searchspace which has to be searched by the FPGA. By sub-searchspace we mean a certain amount of fixed guesses for registers $R1$ and $R2$. Therefore, a software divides the *searchspace* consisting of the 2^{41} possibilities into these sub-searchspaces and transmits to each FPGA another one of them together with the known keystream. The control-interface of the FPGA then counts through this sub-searchspace and provides each guessing-engine with a fixed guess of registers $R1$ and $R2$ to be searched. Every time a guessing-engine finishes its search it sends a report to the control-interface whether it was successful or not on finding a state candidate and requests for another fixed guess of registers $R1$ and $R2$ out of the current sub-searchspace. In case of success the valid state candidate is propagated to the software. This is repeated until the whole sub-searchspace is searched by the FPGA. During the search, the software retrieves regularly at reasonable intervals the status information of each FPGA and assigns a new sub-searchspace to an FPGA if requested. The search is finished when all state candidates that can be generated with the 2^{41} possibilities for guessing $R1$ and $R2$, i.e., the whole searchspace, are checked for consistency.

3.2 Optimization: Storing Intermediate States

When completely passing through a binary decision tree, edges near the root node are traversed much more often than edges near the leaf nodes. The number of cycles $R3$ needs to be clocked to reach any leaf of the tree is 11 (cf. Section 3.1). For example, when inspecting the two leftmost leaves we have to go bit by bit through the states 00000000000 and 00000000001 of the 11-bit wide counter corresponding to the tree. Apparently, the first ten edges up to the node of depth 10 for both leaves are identical. Therefore, we can create *recovery points* at some depth in the search tree. More precisely, it is possible to store the intermediate state (i.e., the content of all A5/1 registers) at such a point (node of tree) and search the subtree starting at this recovery point instead of starting at the root node. This apparently demands a larger area, but saves a certain amount of clock-cycles.

Let us assume that reloading takes exactly one clock-cycle. If we store and reload the intermediate states at depth $d = 10$, then the number of clock-cycles for $R3$ reduces from 11 to $\frac{11+1+1}{2} = 6.5$ on average: 11 times clocking $R3$ to reach the first leaf, one clock-cycle reloading the intermediate state, and one time clocking $R3$ to reach the next leaf from the reloaded state. If we store the intermediate states at depth $d = 9$, the corresponding subtree has 4 leaves. To reach the leftmost one takes 11 clock-cycles, but to reach the other 3 leaves will take just $1 + 2 = 3$ clock-cycles each. Therefore, the average number of times $R3$ needs to be clocked is in this case only $\frac{11+3+3+3}{4} = \frac{8+3\cdot 4}{4} = 5$.

Generalizing this approach of storing and reloading intermediate states at a depth of $d = 10$ or $d = 9$ to a depth of $d = b + 1$, where b denotes the number of the bit in the 11-bit wide counter consecutively numbered from 0 to 10, we need to clock $R3$

$$f(b) = \frac{b + (11 - b) \cdot 2^{(10-b)}}{2^{(10-b)}} \tag{1}$$

times on average to reach one leaf. The function has a minimum of 4.875 times clocking $R3$ on average to reach a leaf for storing and reloading intermediate states at a depth of $b_{min} = 7$ for $b \in \mathbb{N}$.

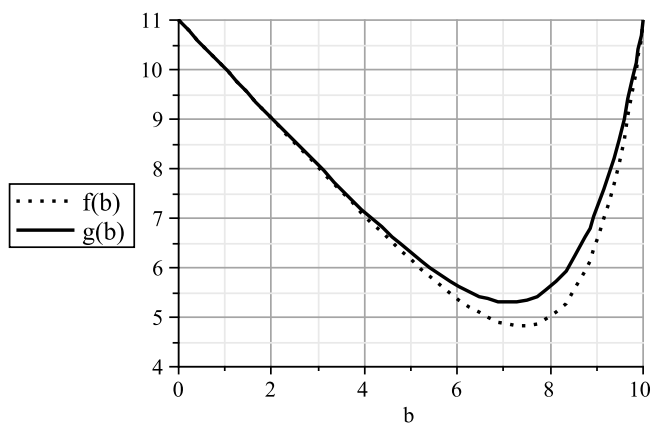

Fig. 7. Functions $f(b), g(b)$: The average number of cycles clocking $R3$ to generate a state candidate with reloading intermediate states at recovery position b

Taking also into account that some subtrees are discarded while passing through the tree (cf. Section 2.2) and the number of possibilities is reduced from 2 to $\frac{7}{4}$ for every guess, the function needs to be adapted:

$$g(b) = \frac{b + (11 - b) \cdot (\frac{7}{4})^{(10-b)}}{(\frac{7}{4})^{(10-b)}}. \tag{2}$$

Both functions $f(b)$ and $g(b)$ are shown in Figure 7. The value for the minimum of the function $g(b)$ now changes to approximately 5.31 at $b_{min} = 7$ for $b \in \mathbb{N}$. Therefore, the expected number of clock-cycles for generating and checking one state candidate is now

$$T_{opt} = 1 + \frac{4}{3} \cdot 5.31 + 2 \approx 10.10 \approx 2^{3.33}$$

instead of $T = 17\frac{2}{3}$ (cf. Section 2.2). This results in an optimized time complexity of

$$C_{opt} \approx 2^{41} \cdot 2^{8.88} \cdot 2^{3.33} \approx 2^{53.21}$$

and reduces the previous complexity of $C \approx 2^{54.02}$ by 0.81 bit. But when comparing the time complexities of the standard and the optimized guessing-engine we additionally have to take the required area into account. The optimized guessing-engine is expected to occupy a larger area because of the storing elements for intermediate states of several registers. Hence, we will be able to place less instances on one FPGA. This comparison of time-area products is done after the implementation process and will be discussed in Section 3.3.

3.3 Implementation Results for COPACOBANA

We used Xilinx ISE Foundation 9.2i to synthesize and implement all components for a Xilinx Spartan3-XC3S1000-FT256 FPGA used in COPACOBANA. The simulation of the hardware model was done in MentorGraphics ModelSim SE 6.3d.

First, we implemented and tested one single instance of the standard and optimized guessing engine together with the control-interface for one instance. Therefore, Table 2 shows the post place & route results of the implementation process for a single instance of the control-interface and both guessing-engines.

Table 2. Implementation results for the control-interface and the guessing-engines

	slices	**flip-flops**	**look-up tables**	$\mathbf{f_{max}}$ **[MHz]**
control-interface	371	304	254	123.19
standard guessing-engine	202	179	256	112.84
optimized guessing-engine	311	312	412	115.01

To decide whether it is worth or not implementing the optimized guessing-engine in spite of the increased area consumption we calculated the *time-area product*. Table 3 shows a comparison of the computing time T and T_{opt} in *clock-cycles* (cf. Sections 2.2 and 3.2), the number of slices needed, and the time-area

product in *clock-cycles·slices* for our standard and optimized implementation of the guessing-engine. The last row shows the quotient of the values of both designs. The quotient of the time-area products shows an overall improvement of about 12% for one single optimized guessing-engine compared to the standard one. We omitted considering the operating frequencies in the time-area product because both implementations run at nearly the same speed.

Table 3. Comparison of the implementation results of both guessing-engines

	computing-time [clock-cycles]	**slices**	**time-area product [clock-cycles · slices]**
optimized	10.10	311	3,141.10
standard	17.67	202	3,568.73
$\frac{\textbf{optimized}}{\textbf{standard}}$	0.57	1.54	0.88

After having tested a single instance of each guessing-engine together with the control-interface on one of the *Spartan3-XC3S1000* FPGAs we attempted to maximize the utilization ratio of the available hardware resources. For this purpose, we implemented as many instances as possible of both types of guessing-engines with one instance of the control-interface. We were able to place & route 36 instances of the standard engine on one of the target FPGAs. However, the complexity of the control-interface grows with the number of guessing-engines. For 36 such engines the critical path was transfered to the control-interface creating the bottle-neck of the design. Therefore, the achieved maximum frequency of 81.13 MHz was relatively low. So we decided to implement less engines at a higher frequency instead. The best trade-off for the standard guessing-engine was to implement 32 instances at a maximum frequency of 102.42 MHz. In case of the optimized guessing-engine we were able to implement 23 instances running at 104.65 MHz. The implementation results of both complete designs are shown in Table 4. Additionally, the available resources of one FPGA are listed, too.

Table 4. Implementation results of the maximally utilized designs

	slices	**FFs**	**LUTs**	$\mathbf{f_{max}}$ **[MHz]**	$\mathbf{f_{test}}$ **[MHz]**
1 control-engine &					
◦ **36 standard**	6,953 (91 %)	10,730	10,576	81.85	72.00
◦ **32 standard**	6,614 (86 %)	9,636	9,417	102.42	92.00
◦ **23 optimized**	7,494 (98 %)	10,141	10,562	104.65	92.00
guessing-engines					
Spartan3-XC3S1000	7,680 (100 %)	15,360	15,360	300.00	—

Table 4 also shows the frequencies the designs were tested with. Thus, we can calculate a preliminary estimation of the computation time to determine and check all possible state candidates. For the slow design with the standard guessing-engine and a time complexity of $C = 2^{54.02}$ (cf. Section 2.2) we expect a computation time of

$$t_{est} = \frac{2^{54.02}}{120 \cdot 36 \cdot 72 \cdot 10^6} \cdot \frac{1}{3600}\,\mathrm{h} \approx 16.31\,\mathrm{h}.$$

This is an estimation for a fully equipped COPACOBANA with 120 FPGAs. In accordance to the previous calculation, the preliminary estimation of the computation time for the smaller but faster standard design (32 instances @ 92 MHz) is $t'_{est} \approx 14.36\,\mathrm{h}$. For the optimized guessing-engine (23 optimized instances @ 92 MHz) with a time complexity of $C_{opt} = 2^{53.21}$ we expect a computation time of $t''_{est} \approx 11.40\,\mathrm{h}$.

Time measurements of several extended test runs on COPACOBANA showed an average computation time of $t' = 13.58\,\mathrm{h}$ for the small and fast standard design to perform a complete search for a given 64 bit known keystream. Comparing this result to the estimation of the computing time t'_{est} shows that the complexity differs only by 0.08 bit from our measurements. The optimized design took an average computation time of $t'' = 11.78\,\mathrm{h}$ for a full search. This equals a variation of only 0.05 bit between the estimated and the measured computation time. Because these were the computation times for a full search (i.e., the worst case) the expected average time for finding the valid state candidate is 6.79 h for the standard design and 5.89 h for the optimized design, respectively.

4 Conclusion

In this paper we presented a guess-and-determine attack on the A5/1 stream cipher running on the special-purpose hardware device COPACOBANA. It reveals the internal state of the cipher in less than 6 hours on average needing only 64 bits of known keystream. We like to stress that our attack is also very attractive with regard to monetary costs which is a significant factor for the practicability of an attack: The acquisition costs for COPACOBANA are about US$ 10,000. Since COPACOBANA has a maximum power consumption of only 600 W, the attack also features very low operational costs. For instance, assuming 10 cent per kWh the operational costs of an attack are only 36 cents.

We like to note that we just provided a machine efficiently solving the problem of recovering a state of A5/1 after warm-up given 64 bits of known keystream. There is still some work to do in order to obtain a full-fledged practical GSM cracker: To finally recover the session key used for encryption, the cipher still needs to be tracked back from the revealed state to its initial state. Albeit, this backtracking and the extraction of the key can be done efficiently and in a fraction of time on almost any platform. Further technical difficulties will certainly appear when it actually comes to eavesdropping GSM calls. This is due to the frequency hopping method applied by GSM which makes it difficult

to synchronize a receiver to the desired signal. Also the problem of obtaining known plaintext is still under discussion in pertinent news groups and does not seem to be fully solved. However, these are just some technical difficulties that certainly cannot be considered serious barriers for breaking GSM.

References

1. Barkan, E., Biham, E.: Conditional Estimators: An Effective Attack on A5/1. In: Proc. of SAC'05. Volume 3897 of LNCS., Springer-Verlag (2006) 1–19
2. Biham, E., Dunkelman, O.: Cryptanalysis of the A5/1 GSM Stream Cipher. In: Proc. of Indocrypt'00. Volume 1977 of LNCS., Springer-Verlag (2000)
3. Biryukov, A., Shamir, A., Wagner, D.: Real Time Cryptanalysis of A5/1 on a PC. In: Proc. of FSE'00. Volume 1978 of LNCS., Springer-Verlag (2001) 1–18
4. Ekdahl, P., Johansson, T.: Another Attack on A5/1. IEEE Transactions on Information Theory **49**(1) (2003) 284–289
5. Golic, J.: Cryptanalysis of Alleged A5 Stream Cipher. In: Proc. of Eurocrypt'97. Volume 1233 of LNCS., Springer-Verlag (1997) 239–255
6. Maximov, A., Johansson, T., Babbage, S.: An Improved Correlation Attack on A5/1. In: Proc. of SAC'04. Volume 3357 of LNCS., Springer-Verlag (2005) 239–255
7. Pornin, T., Stern, J.: Software-hardware Trade-offs: Application to A5/1 Cryptanalysis. In: Proc. of CHES'00. Volume 1965 of LNCS., Springer-Verlag (2000) 318–327
8. Keller, J., Seitz, B.: A Hardware-Based Attack on the A5/1 Stream Cipher. http://pv.fernuni-hagen.de/docs/apc2001-final.pdf (2001)
9. Kumar, S., Paar, C., Pelzl, J., Pfeiffer, G., Schimmler, M.: Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In: In Proc. of CHES'06. Volume 4249 of LNCS., Springer-Verlag (2006) 101–118
10. Briceno, M., Goldberg, I., Wagner, D.: A Pedagogical Implementation of the GSM A5/1 and A5/2 "voice privacy" Encryption Algorithms. http://cryptome.org/gsm-a512.html (1999)
11. Anderson, R.: A5 (was: Hacking digital phones). sci.crypt (17 June 1994)
12. Babbage, S.: A Space/Time Tradeoff in Exhaustive Search Attacks on Stream Ciphers. In: European Convention on Security and Detection. (May 1995)
13. Golic, J.: Cryptanalysis of three mutually clock-controlled stop/go shift registers. IEEE Transactions on Information Theory **46** (May 2000) 1081–1090
14. Biryukov, A., Shamir, A.: Cryptanalytic time/memory/data tradeoffs for stream ciphers. In: Proc. of Asiacrypt'00. Volume 1976 of LNCS., Springer (2000) 1–13
15. Barkan, E., Biham, E., Keller, N.: Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communications. In: Proc. of Crypto'03. Volume 2729 of LNCS., Springer-Verlag (2003)
16. Barkan, E., Biham, E., Keller, N.: Instant Ciphertext-only Cryptanalysis of GSM Encrypted Communication (full-version). Technical Report CS-2006-07, Technion (2006)
17. Xilinx: XC4000E and XC4000X Series Field Programmable Gate Arrays (May 1999)
18. Xilinx: Spartan-3 FPGA Family: Complete Data Sheet, DS099 (November 2007)