# Hardware-Based Cryptanalysis of the GSM A5/1 Encryption Algorithm

Timo Gendrullis

May 29th, 2008

Diploma Thesis

Department of Electrical Engineering
& Information Sciences
Ruhr-University Bochum

Chair for Communication Security
Prof. Dr.-Ing. Christof Paar

Tutors: Dipl.-Inf. Andy Rupp
Ing. Martin Novotný

# Eidesstattliche Erklärung

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate als solche kenntlich gemacht habe.

I hereby declare that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by reference to other authors.

| | |
|---|---|
| Datum/Date | Timo Gendrullis |

# Abstract

In this diploma thesis 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 [BB06, BD00, BSW01, EJ03, Gol97, MJB05, PS00]. 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 [KS01] is running on an existing special-purpose hardware device, called COPACOBANA [KPP$^+$06]. 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. Parts of this thesis have been published in [GNR08].

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

# Contents

# List of Figures

# List of Tables

# Nomenclature

| | |
|---|---|
| ASIC | Application Specific Integrated Circuit |
| AuC | Authentication Center |
| BSC | Base Station Controller |
| BSS | Base Station System |
| BTS | Base Transceiver Station |
| CB | Clocking Bit |
| CLB | Configurable Logic Block |
| COPACOBANA | Cost-Optimized Parallel Code Breaker |
| DCM | Digital Clock Manager |
| DFS | Digital Frequency Synthesizer |
| EDA | Electronic Design Automation |
| EIR | Equipment Identity Register |
| FF | Flip-Flop |
| FPGA | Field Programmable Gate Array |
| FSM | Finite State Machine |
| GSM | Global System for Mobile communication |
| HDL | Hardware Description Language |
| HLR | Home Location Register |
| IC | Integrated Circuit |
| IMEI | International Mobile Equipment Identity |
| IMSI | International Mobile Subscriber Identity |
| IOB | Input/Output Block |
| IV | Initialization Vector |
| LFSR | Linear Feedback Shift Register |
| LUT | Look-Up Table |
| MS | Mobile Station |

| | |
|---|---|
| MSB | Most Significant Bit |
| MSC | Mobile Switching Center |
| MUX | Multiplexer |
| OMC | Operations and Maintenance Center |
| OMSS | Operation and Maintenance Subsystem |
| PLD | Programmable Logic Device |
| PSTN | Public Switched Telephone Network |
| RAM | Random Access Memory |
| ROM | Read Only Memory |
| SDF | Standard Delay Format |
| SIM | Subscriber Identity Module |
| SMSS | Switching and Management Subsystem |
| TMDTO | Time-Memory-Data TradeOff |
| VHDL | Very high speed integrated circuit Hardware Description Language |
| VLR | Visitor Location Register |
| VLSI | Very-Large-Scale Integration |

# 1 Introduction

## 1.1 Motivation

The global system for mobile communications (GSM) is considered as the second generation (2G) mobile phone system and made mobile communications accessible for the mass market. In many industrial countries the number of mobile subscribers even exceeds that of the conventional telephone network. GSM and its underlying security architecture were developed back in the 1980s. Because it is widely deployed and became ubiquitous in most countries around the world it is still growing in coverage.

The stream ciphers A5/1 and A5/2 for securing the over the air communication were kept secret at first. This creates the impression that the developers wanted to increase their security by obscurity. But in contrast to that, according to Kerckhoffs' principle, a cipher should be secure even if it is publicly known. Furthermore, a public evaluation process can actually enhance the security of a cipher. In most cases it is just a matter of time until technical details or even the complete algorithm of an undisclosed cipher is leaked. That was the case with A5/1 and A5/2 as well. When the ciphers were reverse engineered in 1999 [BGW99] scientists started cryptanalyzing them and found several cryptographic weaknesses. Despite this fact, both ciphers are still used for encrypting GSM traffic and, thus, to provide confidentiality.

Even though many attacks were proposed (cf. Section 1.2) none of them were, to the best of our knowledge, entirely realized. Thus, the motivation of this work was to fully design, implement, test, and evaluate an attack against the GSM A5/1 encryption algorithm on an existing target platform.

## 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 [And94], Golic [Gol97], and Babbage [Bab95].

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 [KS01] 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 [KS01] 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 3.1 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 [Gol97] 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 $64 \times 64$ LSE (and the verification of the corresponding state candidate).

Pornin and Stern proposed a SW/HW tradeoff attack [PS00] 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 [Gol00]. These guesses create a tree with 4 branches in each node (each branch represents one clocking combination, cf. Table 2.1). 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 [Bab95] and Golic [Gol97]. Recently, Biryukov, Shamir, and Wagner presented an interesting (non-generic) variant of an TMDTO [BSW01] (see also [BS00]) 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 [BBK03] (see also [BBK06]). 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. [BBK06]). 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 [KS01]. 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 Outline

The remainder of this diploma thesis is organized as follows.

In Chapter 2 we give on the one hand some background information on the GSM architecture and the stream cipher A5/1 securing its over the air communication. On the other hand, we introduce the implementation platform for our attack called COPACOBANA. Starting with a description of the *Xilinx Spartan3* FPGA which is the main building block of the FPGA cluster we discuss some technical details of this special-purpose hardware. Finally, this chapter is closed with an overview of the design flow we followed throughout the whole development phase.

Chapter 3 starts with an analysis of the attack algorithm proposed by Keller and Seitz [KS01] which our attack is based on. In the subsequent two sections we present our modification of this algorithm and determine its time complexity. With the described attack we are able to reveal the internal state of the algorithm after what is referred to as the warm-up state (cf. Section 2.1.2). To complete the attack we describe in the last section of this chapter how to extract the session key out of this internal state.

Both the hardware and the software architecture of the attack are presented in Chapter 4. The hardware section of this chapter deals with a plain and an optimized implementation of a *guessing-engine* to search for the internal state of the cipher. In addition, we present a *control-interface* which provides the intercommunication between the guessing-engines and their environment. In the software section we describe how the hardware implementation on the target platform is controlled by the host computer.

The implementation results of the hardware architecture are shown in Chapter 5. Starting with the synthesis estimations for the plain and the optimized guessing-engine of Chapter 4 we state the full post place & route results of the single engines. After a comparison of the efficiencies of the two different engines the designs are maximally utilized for the target platform. Thereafter, the estimations of the computation times are compared to the actually measured running times on COPACOBANA.

Finally, we summarize the results of this thesis in Chapter 6 and draw a conclusion.

# 2 Background

First, this chapter introduces the GSM network in which the stream cipher A5/1 is used and, afterwards, describes the cipher itself. The second focus is to give some details on the target platform of our implementation and to sketch the design flow.

## 2.1 Global System for Mobile Communication

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 and accounts for 82% of the global mobile market. More than three billion ($3 \cdot 10^9$) customers in 218 countries and territories use GSM technology and yield nearly 29% of the global population (see [CE08]).

In the following we will briefly introduce the infrastructure of the GSM network. It is divided into three main subsystems [ETS01]:

- *Base Station System* (BSS)
- *Switching and Management Subsystem* (SMSS)
- *Operation and Maintenance Subsystem* (OMSS)

The BSS supplies a certain area (i.e., cell) with the GSM network over the radio path and contains several *base transceiver stations* (BTS) which are managed by a *base station controller* (BSC). The BTS contains all radio related hardware for transmitting and receiving the radio signal (e.g., antenna, transceiver) and is connected to the BSC either with a cable or a radio link. The BSC generally controls multiple BTS and provides their interconnection to the SMSS.

The SMSS is mainly responsible for connecting the radio network of GSM to the public partner networks (e.g., the *public switched telephone network* (PSTN)). For this purpose, the SMSS contains a *mobile switching center* (MSC) which supplies several BSCs and is their gateway to the PSTN. Additionally, every MSC has its own *visitor location register* (VLR) in which all necessary user related data are stored to identify a subscriber. Therefore, the VLR can request for these data at the *home location register* (HLR) of the subscriber's provider every time the subscriber travels (i.e., roams) from one MSC to another.

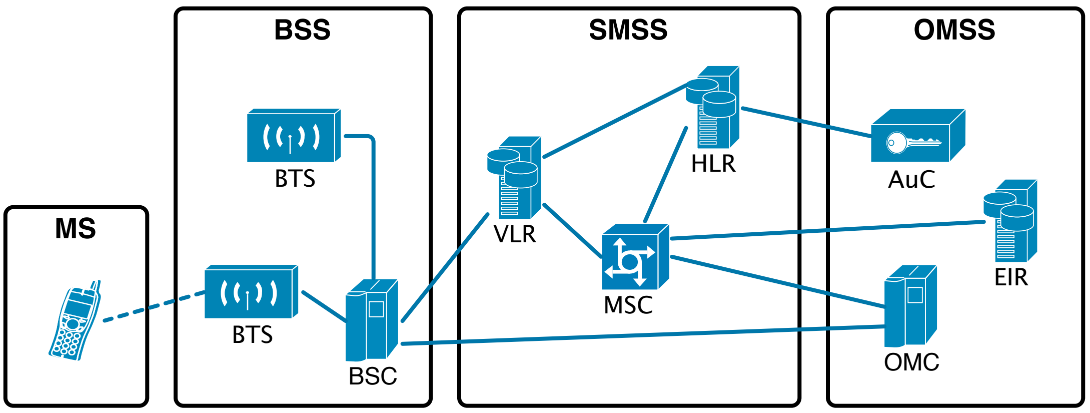


**Figure 2.1:** GSM infrastructure

The OMSS guarantees the operation and maintenance of the GSM network by controlling the other network elements, more precisely the BSCs and MSCs. This is done by the *operations and maintenance center* (OMC). Another important component of the OMSS is the *authentication center* (AuC) which provides the operator's HLR with the necessary data for authenticating a subscriber and protecting his identity (cf. Section 2.1.1). The last network element is the *equipment identity register* (EIR) in which known *international mobile equipment identity* (IMEI) numbers are stored. If a mobile radio station is reported to be stolen its IMEI number can be added to a black list in the EIR and, thus, the equipment will be suspended.

The equipment used by the subscriber to establish a connection to the GSM notwork is called *mobile station* (MS) and is uniquely identified by an IMEI. It contains all necessary hardware and software components for a radio-based communication and additionally a *subscriber identity module* (SIM). The SIM stores all subscriber related personal data (e.g., the *international mobile subscriber identity* (IMSI)) and can compute different algorithms for authentication purposes. Typically, the SIM is manufactured as a smart card.

Figure 2.1 gives an overview of the previously introduced GSM infrastructure and its components.

### 2.1.1 The Security Architecture of the GSM Network

To protect both the subscriber and the provider against violation/misuse, the security architecture of GSM was developed by the Security Expert Group (SEG) founded in 1984. Finally, they stated five essential security features for GSM communication [ETS00, ETS97, Hil01, RWO98, Wal01]:

- **Subscriber identity confidentiality:** It provides protection against tracing the location of a mobile subscriber by listening to the signaling exchanges made on the radio path.
- **Subscriber identity authentication:** It protects the network against unauthorized usage and the mobile subscriber against being impersonated by such an unauthorized user.
- **User data confidentiality on physical connections:** It ensures the privacy of the user information on traffic channels by providing encryption of all voice and non-voice communications.
- **Connectionless user data confidentiality:** It ensures the privacy of the user information transferred in a connectionless packet mode on signaling channels (e.g., short messages).
- **Signaling information element confidentiality:** It ensures the privacy after connection establishment of users related signaling elements, i.e., the IMEI, the IMSI, the calling subscriber directory number, and the callers subscriber directory number. Signaling information needed for connection establishment is not protected.

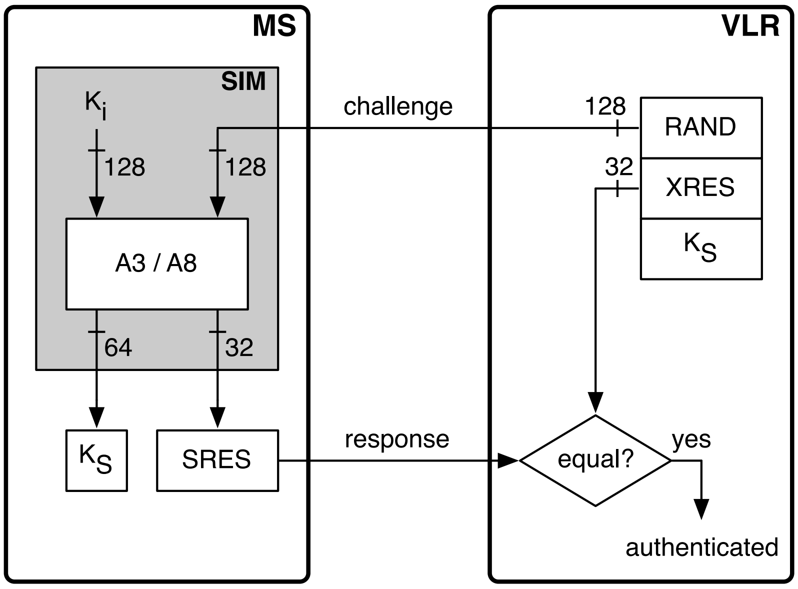


**Figure 2.2:** Challenge response authentication in GSM

The security architecture providing these features is based on symmetric cryptography with a long-term secret. More precisely, the long-term secret is a 128 bit secret key $K_i$ which is uniquely determined for each subscriber. Only the subscriber and his home operator are in possession of this secret. Therefore, $K_i$ is stored in the SIM used in an MS on the subscriber side and only in the AuC of the home operator on the provider side. It is then used in a challenge/response protocol to authenticate the subscriber. The challenge/response protocol works as follows: If a subscriber wants to be authenticated by an operator he first has to identify himself with the help of his IMSI. Afterwards, the operator sends him a 128 bit random number $RAND$ (i.e., the challenge) which he has to respond to with a 32 bit signed response $SRES$. This answer $SRES$ is then compared to the expected response $XRES$ already known by the operator. If the two values match the subscriber is authenticated. The algorithm calculating the signed and the expected response is called A3 and accepts the two 128 bit values $K_i$ and $RAND$ as inputs. Next to this, the algorithm A8 generates a 64 bit session key $K_S$ which is later used for encrypting the communication after a successful authentication. Again, the two values $K_i$ and $RAND$ are the inputs of this algorithm. Together with $K_i$ the IMSI, the authentication algorithm A3, and the key generation algorithm A8 are stored and performed on the SIM. Figure 2.2 summarizes the challenge/response authentication protocol.

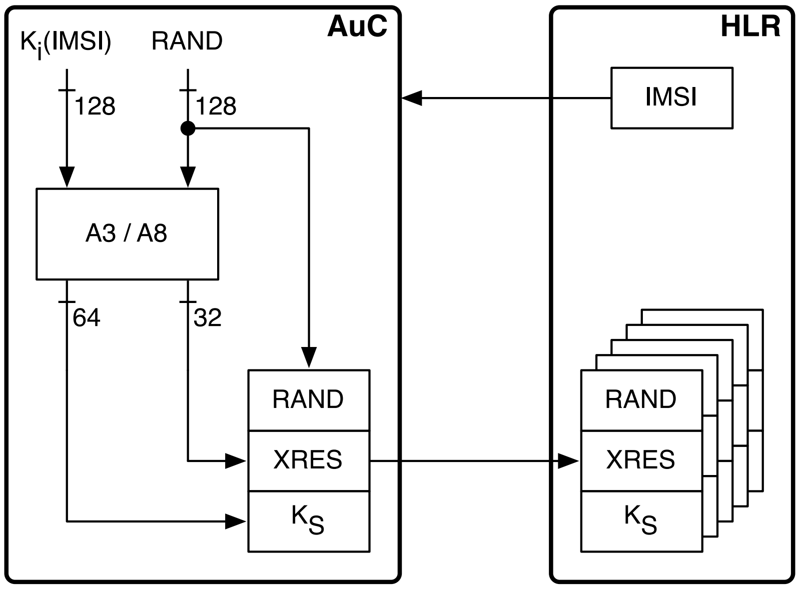


**Figure 2.3:** Security triplet generation in the AuC

The secret key is not to be shared with a third party, especially to be mentioned not with another operator even if the subscriber roams to that network. To still be able to authenticate a subscriber of another network operator the VLR can request at the subscriber's HLR for a set of precomputed triplets of the values $RAND$, $XRES$, and $K_S$ belonging to the IMSI of a subscriber. The triplets at the HLR were previously computed by the respective AuC which is solely knowing the secret key $K_i$ on the operator side. When an HLR requests for a new batch of data triplets for a certain IMSI the AuC looks up the appropriate $K_i$ and generates several random numbers $RAND$. It then uses the algorithms A3/A8 to compute $XRES$ and $K_S$ just like the subscriber's SIM does. Finally, the values are stored in the HLR where they can be requested by a VLR (see Figure 2.3).

After being successfully authenticated the subscriber and the operator can both be sure of having generated the same session key $K_S$ for a subsequent encryption. Because the whole authentication process up to now was done unencrypted the last message sent in *clear text mode* is now "`ciph mod cmd`" sent by the operator. This switches the MS to the *ciphered mode* and it responds with the determined encrypted message "`ciph mod com`". From now on, the whole following communication is done encrypted. The Algorithm used for the encryption is the A5/1 algorithm described in the next section.

### 2.1.2 The GSM A5/1 Encryption Algorithm

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 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 [BGW99]. In this work we focus on the stronger GSM cipher A5/1.

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 2.4). 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. During the initialization phase all three registers are clocked regularly and the key bits followed by the IV bits are xored with the least significant bits of all three registers. Thus, the initialization phase takes an overall of $64 + 22 = 86$ clock-cycles after which 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

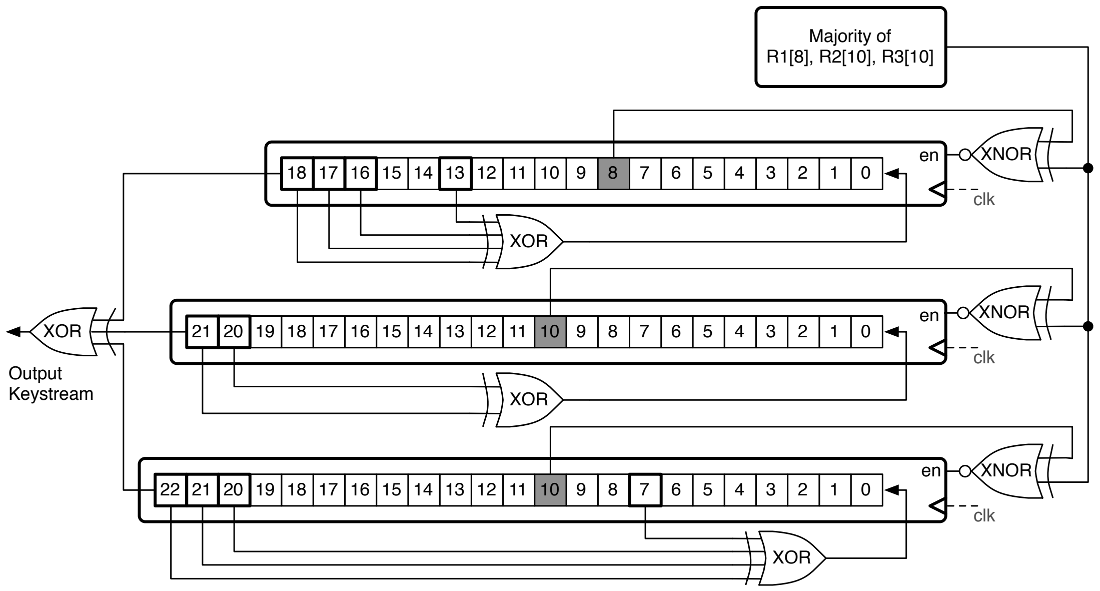


**Figure 2.4:** Design of A5/1

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 2.1 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 (MSBs).

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 diploma thesis we assume that we are given at least 64 consecutive bits of such a 228 bit keystream.

## 2.2 Implementation Platform

As the target platform for our implementation we chose a special purpose hardware called COPACOBANA. Thus, in this section we describe the properties of this hardware device and give an outline of the underlying core element: the *Xilinx Spartan3-XC3S1000* FPGA.

**Table 2.1:** Majority based clock-control of *A5/1*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| CB of $R1$ | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| CB of $R2$ | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| CB of $R3$ | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| Majority | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Clock $R1$? | √ | √ | √ | – | – | √ | √ | √ |
| Clock $R2$? | √ | √ | – | √ | √ | – | √ | √ |
| Clock $R3$? | √ | – | √ | √ | √ | √ | – | √ |

### 2.2.1 Field Programmable Gate Arrays

Field programmable gate arrays (FPGA) are programmable logic devices (PLD) and belong to the class of integrated circuits (IC). More precisely, they are at least from a customer's point of view application specific integrated circuits (ASIC). The main advantages of FPGAs are (i) the wide field of application because (ii) the devices are free configurable and reprogrammable, (iii) the possibility to correct errors while in use, and (iv) the relatively short time to market of a design. But it should also be mentioned that the costs for a design on an FPGA do only stay moderate at low quantities. Furthermore, FPGAs are limited in their resources and, thus, in the complexity of the circuits which can be implemented. Compared to other ASICs (e.g., full-custom or standard cell design) FPGAs achieve slower operating frequencies and demand for more power.

In this section we will focus on the *Xilinx Spartan3-XC3S1000* FPGA as it is the main component of our special purpose hardware described later in Section 2.2.2. Figure 2.5 shows a cut-out of an overview of a *Spartan-3* family FPGA.

The communication between the FPGA and its environment is managed over input/output blocks (IOB). All signals can enter and exit the FPGA only through these IOBs. A set of different standard specifications concerning the signal voltage, current, and buffering makes it easy to integrate the FPGA into a large variety of backplanes with different interface standards. To enable communication of multiple FPGAs on one bus the IOBs provide a three-state output logic which means a possible output of $0, 1$, or $Z$. With the output of the high impedance state $Z$ the FPGA is disconnected from the bus, allowing other resources to communicate on it without affecting them.

The configurable logic block (CLB) is the main component of an FPGA providing the resources to implement the designated functionality. Each CLB consists of four internally connected slices. All four slices consist of three further elements

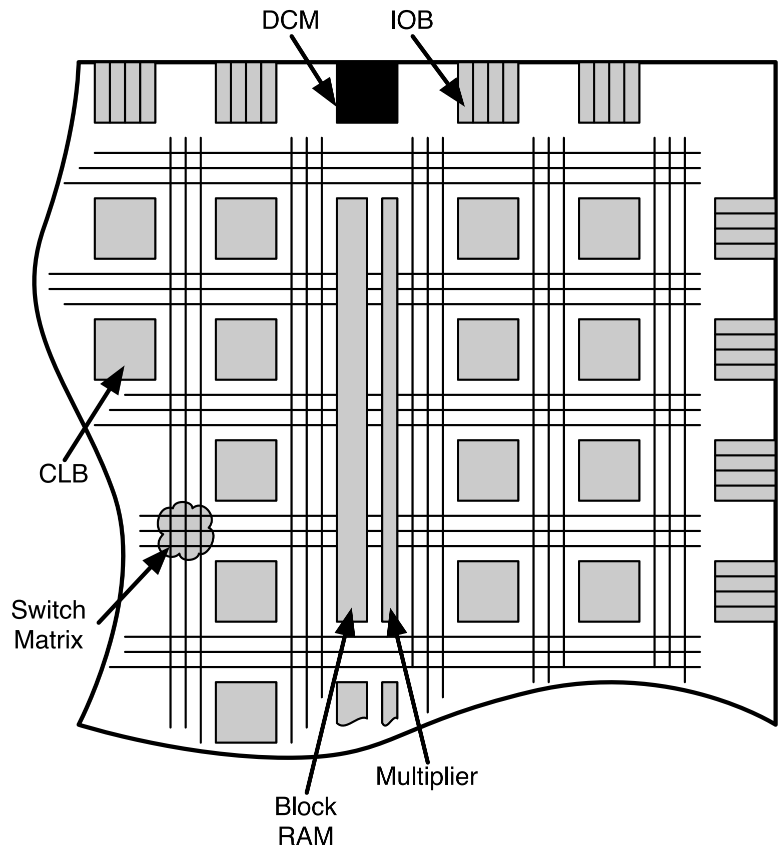


**Figure 2.5:** Cut-out of a schematic overview of a *Spartan-3* FPGA

in general: two flip-flops (FF) as storage elements, two look-up tables (LUT) as logic function generators, and two dedicated multiplexers (MUX). These three elements are shown in Figure 2.6. On the *Xilinx Spartan3-XC3S1000* FPGA 1920 CLBs are placed in a matrix of 48 rows and 40 columns and are connected with a global programmable network of signal pathways. The single signal pathways are linked with each other by programmable switch matrices.

The standard storage element of an FPGA is a D flip-flop which has inputs for data (D), synchronous reset (R), set (S), clock enable (CE), and the clocking signal (C). Generally, the output Q of a D-FF adopts the data value D and delays it for one clock cycle if and only if a rising edge occurs at the clocking input C. Subsequent changes of the data line are ignored until the next rising edge. This non-transparency of edge triggered FFs is the greatest advantage over state triggered latches which are transparent for changes of the data line for one whole state of the clocking signal. Thus, a D-FF can store a binary value for one clock cycle and synchronizes it to the clocking signal. The CE signal allows to store values for more than one clock cycle as the output Q does not change

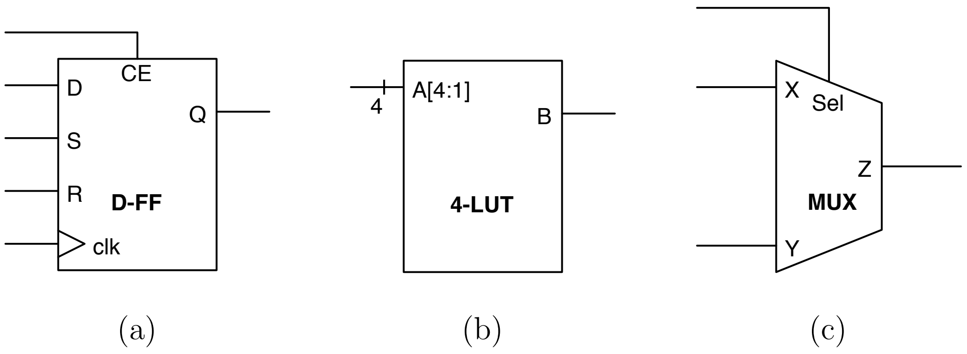


**Figure 2.6:** A flip-flop with synchronous reset, set, and clock enable (a), a 4 input look-up table (b), and a multiplexer (c)

as long as CE is low. Setting CE back to high re-enables the FF and updates the output Q as described. With the synchronous set or reset input being high, the output Q is set to high respectively low at the moment of a rising clock edge regardless of the data input D. This functionality is mostly used for initialization issues. Table 2.2 summarizes the characteristics of the FFs used in *Spartan-3* FPGAs and shows the preferences of the input signals.

**Table 2.2:** Functionality of a D flip-flop with synchronous reset, set, and clock enable

| **Reset (R)** | **Set (S)** | **Clock Enable (CE)** | **Data (D)** | **Clock (C)** | **Output (Q)** |
|---|---|---|---|---|---|
| 1 | X | X | X | rising edge | 0 |
| 0 | 1 | X | X | rising edge | 1 |
| 0 | 0 | 0 | X | X | Q (no change) |
| 0 | 0 | 1 | 0 | rising edge | 1 |
| 0 | 0 | 1 | 1 | rising edge | 0 |

The LUT is a RAM-based function generator and has four logic inputs $A4$, $A3$, $A2$, $A1$ and a single output $B$. It permits to implement any Boolean logic function with up to four variables. For this purpose, it stores every possible linear combination of the input bits with its according output bit. When asked to calculate the solution of a certain input value it just looks up the table entry for this input with the corresponding output bit. To implement Boolean functions with more than four variables it is necessary to cascade multiple LUTs. The number of cascaded LUTs is called the level of logic. The more logic levels are used to implement a function the more difficult it is to interconnect the single components. This is because of limited global interconnection resources of every FPGA. Thus, it is reasonable to use the dedicated MUXs inside the slices first before enhancing functionality with higher levels of logic.

With the dedicated MUXs it is possible to effectively combine several LUTs of either one slice or even more slices of different CLBs. In order to that, more complex operations can be implemented with less LUTs and reducing the amount of logic levels. A MUX has three logic inputs $X, Y, Sel$ and a single output $Z$. With the input bit $Sel$ it can be selected which of the other two inputs, $X$ or $Y$, is passed through to the output $Z$. Now, it is for example possible to implement any possible Boolean function with five variables into one slice by only using two LUTs and one MUX. Thereto, both LUTs contain independent functions of the same four input bits and with the input $Sel$ of the MUX it is decided which of the two LUT outputs is the correct solution. Thus, the MUX provides the fifth input bit of the function with its input $Sel$.

For more complex arithmetic operations all *Spartan-3* FPGAs provide embedded multipliers with only little use of the general purpose resources. One such multiplier accepts two 18 bit wide binary numbers as inputs and produces a 36 bit wide product as output. To multiply more than two numbers or numbers wider than 18 bit several multipliers can be cascaded. The *Spartan3-XC3S1000* FPGA contains 24 of these dedicated multipliers each matched to one block of random access memory (RAM) for fast and efficient data handling. Each block RAM stores $18\,\text{Kbit} = 18,432\,\text{bit}$ of data making a total amount of 432 Kbit on the FPGA. It can be used as a RAM, read only memory (ROM), large LUT, or large shift register.

If the design demands more than one clock signal with different frequencies or if the frequency of the externally provided clock signal is not suitable for the design, new clock signals can be synthesized with the four dedicated digital clock managers (DCM). The digital frequency synthesizer (DFS) inside each DCM can synthesize frequencies between $f_{fx} = 18...307\,\text{MHz}$ out of a given frequency $f_{in} = 1...280\,\text{MHz}$. This is done by multiplying $f_{in}$ with $m = 2...32$ and dividing it by $d = 1...32$. Generally, up to eight different global clock signals can be propagated throughout the FPGA over a designated clocking infrastructure.

Table 2.3 lists an overview of the different components and their amount on the considered *Spartan3-XC3S1000* FPGA as described in this section.

### 2.2.2 The Special Purpose Hardware: COPACOBANA

The COPACOBANA (Cost-Optimized Parallel Code Breaker) machine [KPP$^+$06] is a high-performance, low-cost cluster consisting of 120 *Xilinx Spartan3-XC3S1000* FPGAs. Currently, COPACOBANA appears to be the only such reconfigurable

**Table 2.3:** Summary of *Spartan3-XC3S1000* FPGA elements

| **Element** | **Amount** |
|---|---:|
| Input/Output Blocks (IOBs) | 175 |
| Configurable Logic Blocks (CLBs) | 1,920 |
| Slices | 7,680 |
| Flip-Flops (FFs) | 15,360 |
| Look-Up Tables (LUTs) | 15,360 |
| Digital Clock Managers (DCMs) | 4 |
| Dedicated Multipliers | 24 |
| Block RAM [Kbit] | 432 |

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 developed in this diploma thesis.

**Figure 2.7:** A front view of COPACOBANA (taken from [cop06])

The FPGAs are located on custom made DIMM modules each housing six *Xilinx Spartan3-XC3S1000-4FT256*. A separate power module generates the 1.2 V core voltage for the FPGAs. Up to 20 of such DIMM modules are hosted on the backplane which makes a maximum of 120 FPGAs for one COPACOBANA. The DIMM modules are connected on the backplane by a 64 bit data bus and a 12 bit address bus with an operating frequency of $f_{Bus} = 20\,\text{MHz}$. With 11 of the 12 address bits the array of the 120 FPGAs is addressed. The 20 DIMM module slots are encoded with 5 address bits and another 6 address bits are used for the one-hot encoding of the 6 FPGAs per DIMM module. The one remaining address bit can be used to choose between two different 64 bit data registers on each FPGA. The whole communication with the host computer is provided by a controller card based on an FPGA with a *Xilinx MicroBlaze* softcore. It is connected via ethernet and supports the connection oriented TCP/IP protocol. Figure 2.8 shows the backplane with one DIMM module with its 6 FPGAs and the controller card behind it.

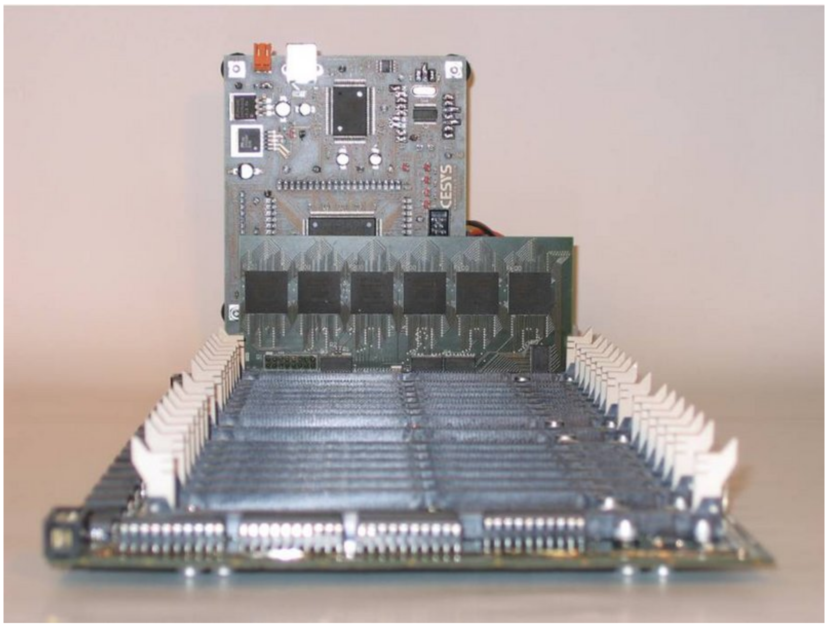


**Figure 2.8:** COPACOBANA backplane with one DIMM module and the controller card (taken from [cop06])

The global clock is distributed by the controller card over the bus on the backplane to the FPGAs on the DIMM modules. To implement designs running at a frequency higher than $f_{Bus} = 20\,\text{MHz}$ the clock signal is synthesized on each FPGA with the embedded DCMs (cf. Section 2.2.1). Therewith, operating frequencies up to 300 MHz can be achieved on the FPGAs.

The whole machine fits into a standard 19" rack of three height units (i.e., 45 cm width, 49.5 cm depth, 13.5 cm height) and makes even stacking of multiple COPACOBANA units feasible. When nearly using all hardware resources of one COPACOBANA and clocking the FPGAs internally with 100 MHz the maximum power consumption is around 600 Watts. The meanwhile moderate costs for the single components, for example only approximately US$ 65 for one *Xilinx Spartan3-XC3S1000* FPGA, make it possible to produce COPACOBANA for less than US$ $10,000$.

### 2.2.3 The Design Flow

According to [Lan06], [Jan03], [Xil07a], and [Vac06] we defined the following top-down design flow which we followed throughout the whole design and development process. Figure 2.9 gives an overview of this design flow which is divided into three sections: the system model design, the system implementation, and the system verification. Due to the very high density of modern ICs with up to millions of transistors we are currently in the era of very-large-scale integration (VLSI) of electronic circuits. To be still capable of efficiently developing such circuits electronic design automation (EDA) became an essential part of this process. Thus, throughout the whole design flow different design tools for each implementation step are available supporting the engineer.

After having determined the system specifications in terms of its behavior, ability to communicate with its environment, and timing constraints a software model of the whole system is created first. This algorithmic model is used to proof the feasibility of the system and to redefine the specifications if necessary. Furthermore, it is also used to generate test vectors for a later verification of both the hardware and the software design. During the subsequent partitioning process it is decided due to the simulation results of the system model which components are realized in hardware and which in software.

Once having partitioned the design we proceed with the system implementation phase. Designing hardware consisting of several thousand gates with only Boolean equations is almost impossible. Because of this, as the hardware design entry, the system components are described in a high level language. By using such a hardware description language (HDL) we remain flexible to choose between as well a functional description of our system at a high level of abstraction as detailed gate-level constructs. For that purpose, we chose the language *VHDL* (very high speed integrated circuit hardware description language) but there are other possibilities such as *Verilog* or *SystemC*. Another important advantage of using an HDL is to be able to immediately simulate the behavior of the so far designed system and thus to verify it at this early design stage with the test vectors created by the system model.

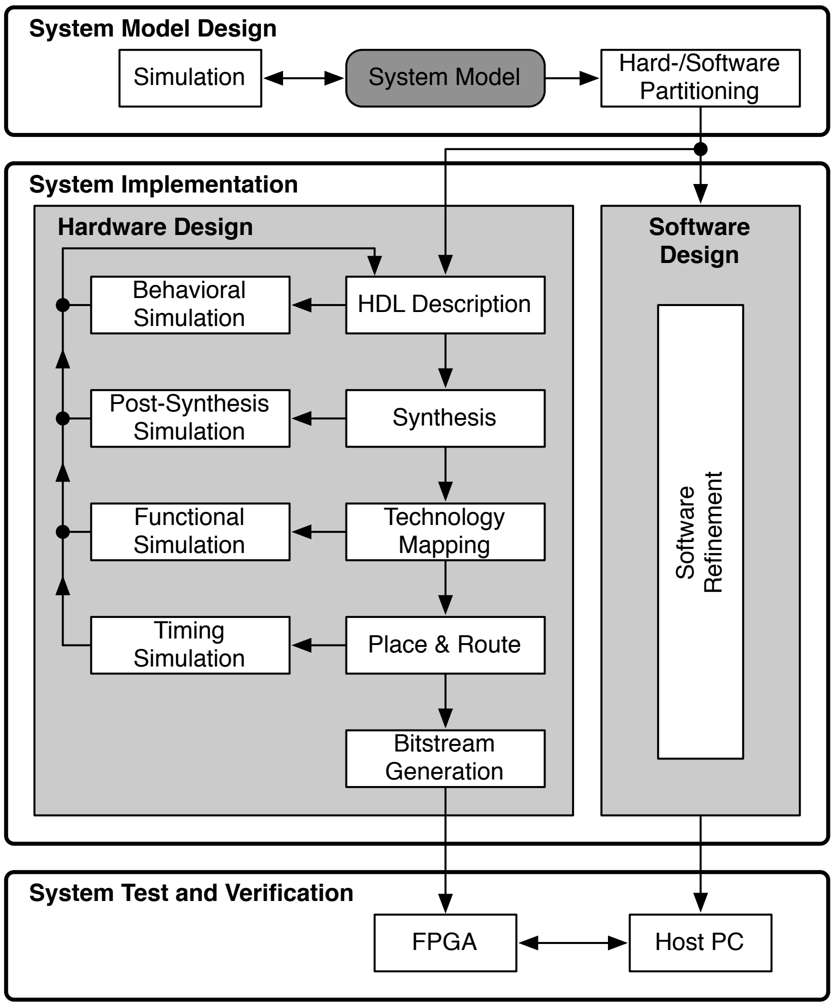


**Figure 2.9:** The design flow for FPGA implementation

Now, this very abstract behavioral model needs to be described more technically. Thus, the next implementation step is the synthesis process. It generates a circuit consisting of logic gates and flip-flops out of the existing HDL description and consequently reduces the level of abstraction. This gate-level netlist can be simulated again to check if it still matches the functional description.

The following mapping process adapts the circuit to the target platform with its different resources. Therefore, the logic gates of the circuit are translated into modules of the FPGA, such as LUTs and FFs. It is also possible to test if the design can be mapped (i.e., if it “fits”) better into different FPGAs. Hence, the mapping process checks if the logic resources of the target platform satisfy the circuit's requirements. As before, this implementation step is again a reduction of the level of abstraction and it needs to be verified by simulation that it does not cause any unintended or erroneous behavior.

After having mapped the circuit into the target platform it needs to be placed and routed. Therefore, the design software searches for a physical place on the FPGA for each logic element first. Then, it tries to route (i.e., wire) the several elements according to the circuit specifications and the user constraints (e.g., the operating frequency, a certain ambient temperature, or placing restrictions). Due to the limited resources of an FPGA routings can differ significantly in their lengths or components can even stay unrouted. In the latter case the design needs to be altered or implemented into a bigger FPGA. If just one given constraint is not met the design can be replaced and rerouted. After the place & route process is finished successfully the timing information about gate delays and signal propagation delays are stored in the standard delay format (SDF). While the gate delays depend on the number of levels of logic each routing has its own signal propagation delay according to its length. Together with this timing information the gap between the system model and the physically implemented design is reduced to a minimum. Therefore, the design needs to be simulated one last time. If the functionality is verified we continue to generate a bitstream file with which the FPGA can finally be configured.

The software design of the system is done in parallel to the hardware design and depends significantly on it. This is because it can be changed much more easily than the hardware design. Every time specifications are redefined in one of its previously described iteration steps they are afterwards realized in the software design. Because the software solves in our design just tasks which are not time-critical such as calculations with minor complexity, controlling, and intercommunicating with the FPGA its design is just mentioned that briefly.

Finally, the FPGA is configured with the bitstream file of the hardware design and tested together with the software design on a host computer. During this system test and verification phase it needs to be checked if the physically implemented design meets all previous simulation results. Errors occurring now but not during the different simulation steps are both difficult to find and to resolve.

# 3 The Attack Algorithm

This chapter deals with an analysis of the approach our attack algorithm is based on and presents our modification. After this, we analyze the time complexity and discuss some methods to recover the session key out of the revealed internal state of the cipher.

## 3.1 Analysis 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.

In the determination phase, Keller and Seitz try to reduce the complexity of the simple guess-and-determine attack by early recognizing contradictions that can occur by guessing the clocking bit (CB) of $R3$. Such a contradiction can occur every time register $R3$ is not clocked. Recognizing and avoiding these contradictions early instead of running into them reduces the number of guesses significantly. Therefore, Keller and Seitz 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 global clock-cycle. Then, foremost compute the first most significant bit (MSB) of $R3$ (i.e., $R3^{(0)}[22]$) 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 $R2$ (i.e., $R1^{(0)}[8]$ 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 (i.e., $R1^{(1)}[18] = R1^{(0)}[17]$, $R2^{(1)}[21] = R2^{(0)}[20]$, $R3^{(1)}[22] = R3^{(0)}[22]$) does not match the given keystreambit 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. Thus, in this case they completely neglect the second choice of clocking $R3$ which could as well lead to a valid state candidate. As a consequence of this they discard twice as many states as the invalid ones. 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 indeed 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

$$N_{KS} = \left(2 - \frac{1}{2}\right)^{11} = \left(\frac{3}{2}\right)^{11} \approx 2^{6.43} \approx 86$$

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 altogether. But because they discard in $\frac{1}{4}$ of all cases 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

$$N = \left(2 - \frac{1}{4}\right)^{11} = \left(\frac{7}{4}\right)^{11} \approx 2^{8.88} \approx 471.$$

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

$$P_{KS} = \frac{N_{KS}}{N} \approx \frac{86}{471} \approx 0.18 = 18\%.$$

Immediately after the determination phase, A5/1 is executed 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

$$C_{KS} \approx 2^{47.43} \cdot 2^{3.81} = 2^{51.24}$$

clock-cycles of the Keller-Seitz-Attack. But with this expected amount of clock-cycles they underestimated the time complexity as will be shown in Section 3.3.

**Expected Performance on COPACOBANA.** 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 [Xil99] and the Spartan-3 XC3S1000 [Xil07b] 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 (cf. Section 2.2.1). 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 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 3.2, the computation time is expected to increase significantly.

## 3.2 Modification of Keller and Seitz's Approach

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 to a contradiction in the next clock-cycle. We call this a *first order contradiction.* But in case of no contradiction we still check for both possibilities of $R3^{(t)}[10]$ which means clocking and not clocking $R3$. Such a first order contradiction can occur only if the clocking bits of registers $R1$ and $R2$ are equal which happens in $\frac{1}{2}$ of all cases. In this case the given and the generated keystream bit of the next round do not agree again with a probability of $\frac{1}{2}$. This reduces in $\frac{1}{4}$ of all cases the number of choices from two to one. Hence, the expected number of possibilities for $R3$ that remain to be checked is approximately 471 for every fixed guess of registers $R1$ and $R2$ (cf. Section 3.1). When proceeding in this way we take every possible state candidate into account and therefore will find unlike Keller and Seitz the correct state candidate in any case.

For a better understanding we describe the process of guessing the eleven clocking bits of $R3$ as a binary decision tree with a height of $h = 11$. The root node of the tree corresponds to the first guess of $R3^{(t)}[10]$. The two edges leading to the depth of $d = 1$ represent the two possible choices for $R3(0)[10]$ (i.e., $R3^{(0)}[10] \neq R1^{(0)}[8]$ and $R3^{(0)}[10] = R1^{(0)}[8]$). Nodes at a depth of $d$ correspond to the guess of $R3^{(d)}[10]$, respectively. Apparently, guessing one clocking bit (i.e., clocking register $R3$ one time) equals traversing a path of the tree by one edge. At the end of the determination phase we have reached one leaf of the tree at the depth of $d = 11$ and have thus fully determined one possible state candidate for a fixed guess of registers $R1$ and $R2$. Now, we can check it for consistency in the postprocessing phase. After this, we start again at the root node to reach the next leaf of the tree. Every time we discard one possibility on our way by early recognizing and avoiding a first order contradiction, we *prune* the binary decision tree by a whole subtree. Figure 3.1 shows such a reduced binary decision tree up to a depth of 3. In Example 1 later on in this section we will go through the steps which led to this tree in detail.

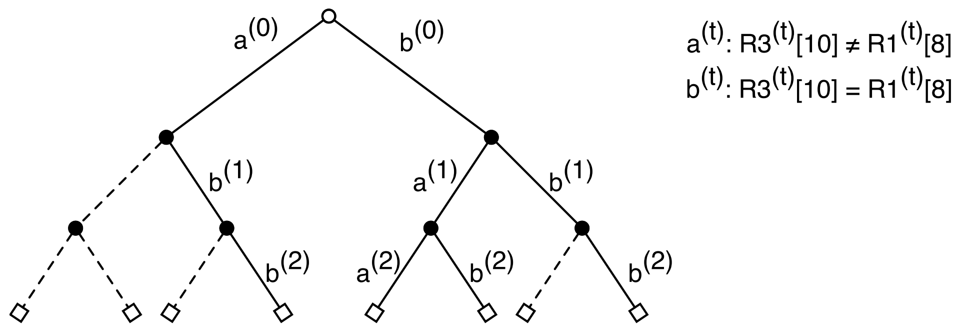


**Figure 3.1:** An example for a reduced binary decision tree of $R3^{(t)}[10]$

Algorithm 1 describes in more detail the determination phase and Algorithm 2 the subsequent postprocessing phase based on the idea of pruning the decision tree as described above. Because of the irregular clocking manner of the A5/1 registers $R1$, $R2$, $R3$ (cf. Section 2.1.2) $t_i$ denotes the number of times register $Ri$ was clocked. This is necessary because we need to guess eleven clocking bits of register $R3$ and, thus, run Algorithm 1 until register $R3$ was clocked eleven times. As both algorithms are performed successively they are illustrated together in a flowchart in Figure 3.2.

**Algorithm 1** Determination phase: generating $R3$

```
Require: fixed guess for registers $R1$ and $R2$, 64 bit of known keystream $KS$
Ensure: a possible state candidate $R1$, $R2$, $R3$
 1: $t \Leftarrow 0$
 2: $t_3 \Leftarrow 0$
 3: compute $R3^{(0)}[22] = R1^{(0)}[18] \oplus R2^{(0)}[21] \oplus KS^{(0)}[0]$
 4: while $t_3 \neq 11$ do
 5:    guess $R3^{(t)}[10]$
 6:    $t \Leftarrow t + 1$
 7:    apply clocking rule
 8:    if $R3$ is clocked then
 9:       $t_3 \Leftarrow t_3 + 1$
10:    else
11:       if $R1^{(t)}[18] \oplus R2^{(t)}[21] \oplus R3^{(t)}[22] \neq KS^{(t)}[0]$ then
12:          discard subtree {higher order contradiction occurred}
13:          return fail
14:       end if
15:    end if
16:    compute $R3^{(t)}[22] = R3^{(0)}[22 - t_3] = R1^{(t)}[18] \oplus R2^{(t)}[21] \oplus KS(t)[0]$
17: end while {$R3$ is completely determined}
```

**Algorithm 2** Postprocessing phase: checking $R3$

```
Require: clock-cycle $t$ and state candidate $R1^{(t)}$, $R2^{(t)}$, $R3^{(t)}$ of Algorithm 1
    after determination phase
Ensure: consistency checking of state candidate
 1: while $t \neq 63$ do
 2:    $t \Leftarrow t + 1$
 3:    apply clocking rule
 4:    if $R1^{(t)}[18] \oplus R2^{(t)}[21] \oplus R3^{(t)}[22] \neq KS^{(t)}[0]$ then
 5:       return fail {contradiction occurred during postprocessing phase}
 6:    end if
 7: end while
 8: return success {state candidate validated}
```

Although the clocking bit of $R3$ is always guessed in a way that contradictions of first order are avoided we still need to check if the generated and the given keystream bit of the next round will coincide (see Algorithm 1, Line 11). If by a certain guess of the clocking bit register $R3$ was stopped for one round the probability that it will not be clocked again in the next round is $\frac{1}{4}$: $\frac{1}{2}$ for the clocking bits of registers $R1$ and $R2$ being equal and $\frac{1}{2}$ for the clocking bit of

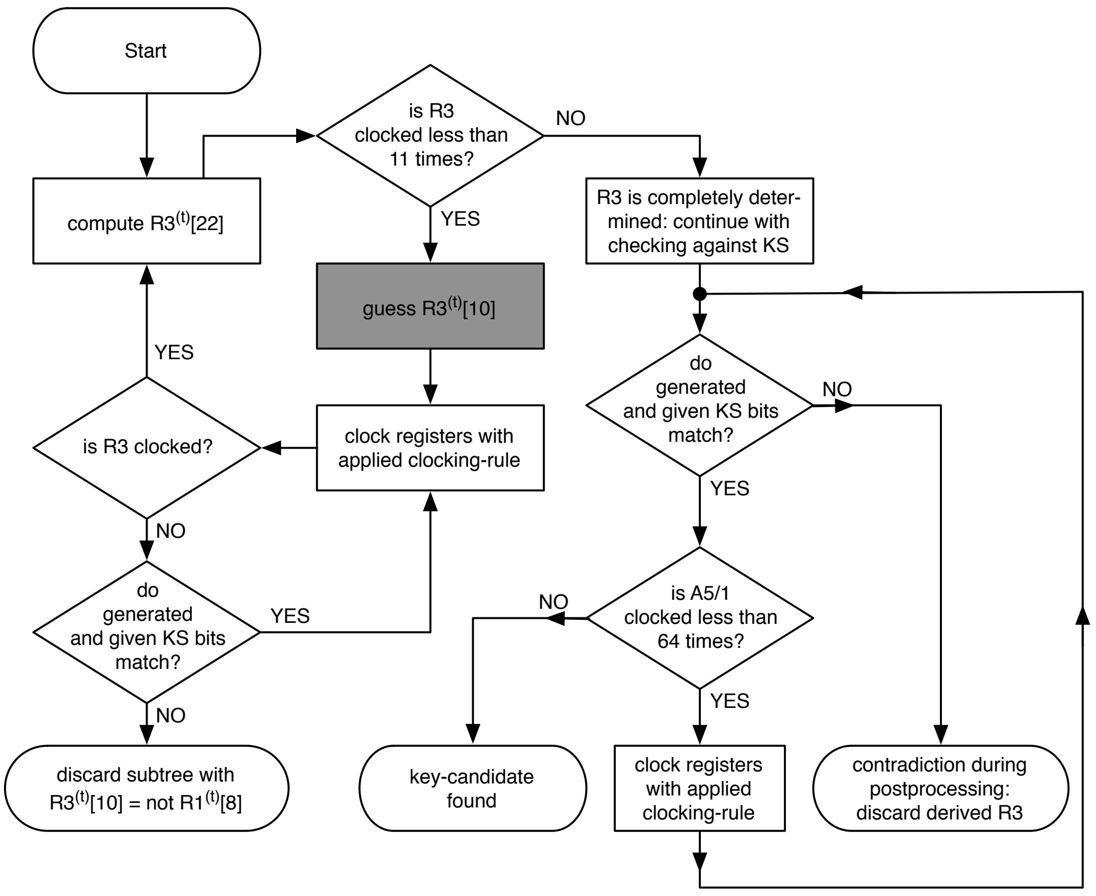

Start
compute R3(t)[22]
is R3 clocked less than 11 times?
NO
YES
R3 is completely determined: continue with checking against KS
guess R3(t)[10]
YES
is R3 clocked?
clock registers with applied clocking-rule
NO
do generated and given KS bits match?
YES
NO
discard subtree with R3(t)[10] = not R1(t)[8]
do generated and given KS bits match?
NO
YES
is A5/1 clocked less than 64 times?
NO
YES
key-candidate found
clock registers with applied clocking-rule
contradiction during postprocessing: discard derived R3

**Figure 3.2:** Flowchart of the determination phase and the postprocessing phase

register $R3$ being different. In this case it can happen that the generated and the given keystream bit disagree. We call this a *higher order contradiction* because it occurred after register $R3$ was not clocked for more than one round. These later contradictions cannot easily be recognized and avoided early unlike those of first order. Every time one of the two algorithms fails or a key candidate could be validated in the postprocessing phase the process restarts with Algorithm 1. This is repeated until the whole decision tree for one fixed guess of registers $R1$ and $R2$ is searched.

A more detailed description of how the clocking bit $R3^{(t)}[10]$ is guessed in Line 5 of Algorithm 1 and how this discards certain subtrees is given in Algorithm 3. Figure 3.3 shows a flowchart of this algorithm.

When asked to guess a clocking bit for register $R3$ Algorithm 3 first chooses $R3^{(t)}[10] \neq R1^{(t)}[8]$ until the whole subtree of this node is completely checked or discarded. This leads in the first iteration to the leftmost leaf of the decision tree. But when guessing a clocking bit at a node with one fully searched subtree or

**Algorithm 3** Guess clocking bit of $R3$

```
Require: clock-cycle $t$ and registers $R1^{(t)}$, $R2^{(t)}$, $R3^{(t)}$
Ensure: guessed clocking bit $R3^{(t)}[10] = R3^{(0)}[10 - t_3]$
 1: if $R1^{(t)}[8] = R2^{(t)}[10]$ then {at least $R1$ and $R2$ will be clocked}
 2:    if $R1^{(t+1)}[18] \oplus R2^{(t+1)}[21] \oplus R3^{(t)}[22] \neq KS^{(t)}[0]$ then
 3:       discard subtree {avoiding first order contradiction}
 4:       guess $R3^{(t)}[10] = R1^{(t)}[8]$ {all registers $R1$, $R2$, $R3$ will be clocked}
 5:    else {no first order contradiction when not clocking $R3$}
 6:       if subtree for $R3^{(t)}[10] \neq R1^{(t)}[8]$ is completely checked then
 7:          return $R3^{(t)}[10] = R1^{(t)}[8]$ {all registers $R1$, $R2$, $R3$ will be clocked}
 8:       else
 9:          return $R3^{(t)}[10] \neq R1^{(t)}[8]$ {register $R3$ will not be clocked}
10:       end if
11:    end if
12: else {CBs of $R1$ and $R2$ disagree: $R3$ will be clocked in any way}
13:    if subtree for $R3^{(t)}[10] \neq R1^{(t)}[8]$ is completely checked then
14:       return $R3^{(t)}[10] = R1^{(t)}[8]$ {registers $R3$ and $R1$ will be clocked}
15:    else
16:       return $R3^{(t)}[10] \neq R1^{(t)}[8]$ {registers $R3$ and $R2$ will be clocked}
17:    end if
18: end if
```

when discarding a subtree the algorithm returns $R3^{(t)}[10] = R1^{(t)}[8]$ instead. This leads in turn to the next leave on the right in the second iteration of determining one possible state candidate.

Example 1 performs the first steps of Algorithms 1-3 guessing the clocking bits and discarding subtrees. The necessary content of registers $R1$ and $R2$ is shown in Figure 3.4. It shows next to the first 4 bits of a known keystream the first 4 MSBs and the first 3 CBs of a possible fixed guess of registers $R1$ and $R2$. The bits of $R3$ derived by this example are displayed, too.

**Example 1** Deriving register $R3$

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 $R1$, $R2$, $R3$.

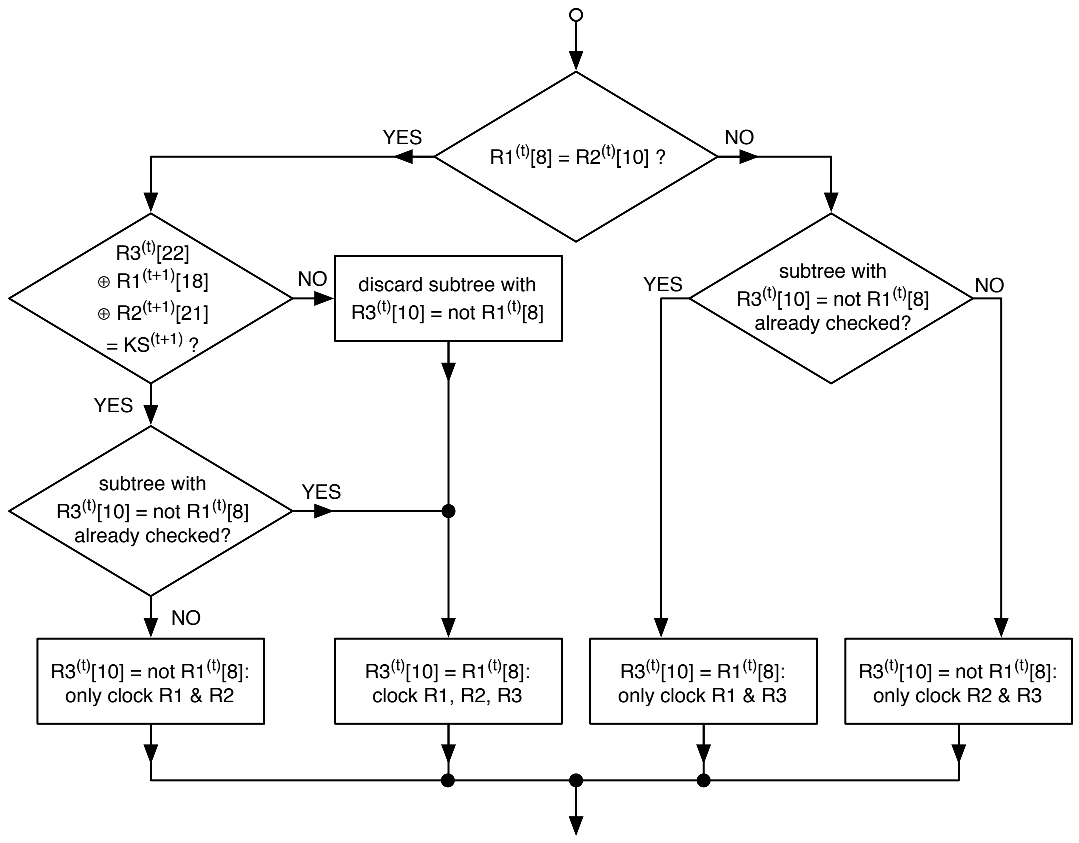

R1(t)[8] = R2(t)[10] ?
YES
NO
R3(t)[22] ⊕ R1(t+1)[18] ⊕ R2(t+1)[21] = KS(t+1) ?
NO
discard subtree with R3(t)[10] = not R1(t)[8]
YES
subtree with R3(t)[10] = not R1(t)[8] already checked?
NO
YES
subtree with R3(t)[10] = not R1(t)[8] already checked?
YES
NO
R3(t)[10] = not R1(t)[8]: only clock R1 & R2
R3(t)[10] = R1(t)[8]: clock R1, R2, R3
R3(t)[10] = R1(t)[8]: only clock R1 & R3
R3(t)[10] = not R1(t)[8]: only clock R2 & R3

**Figure 3.3:** Flowchart of guessing the clocking bit of $R3$ in detail

```
5: Compute $R3^{(2)}[22] = R3^{(0)}[20] = R1^{(0)}[17] \oplus R2^{(0)}[19] \oplus KS[2] = 1$.
6: ...
```

The example ends at this point because it is apparent from Figure 3.1, which shows the binary decision tree for $R3^{(t)}[10]$ up to a depth of 3 corresponding to the example, 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.

## 3.3 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. Because of the irregular clocking

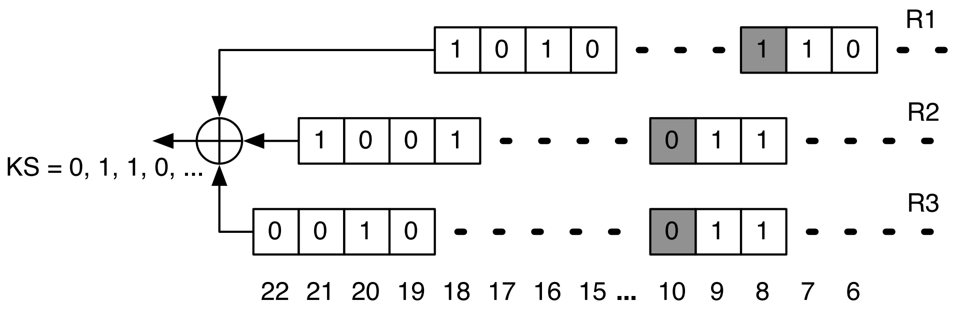


**Figure 3.4:** An example for a generated state candidate after guessing $R3^{(t)}[10]$ three times

rule applied to the A5/1 registers, the probability for each register $R1$, $R2$, $R3$ of being clocked is $P_{clk} = \frac{3}{4}$ every clock-cycle. Thus, the determination phase takes an expected number of

$$T_{dp} = 1 + \frac{4}{3} \cdot 11 = 15\frac{2}{3}$$

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 afterwards 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 the fourth clocking bit of register R3 (i.e., $R3^{(3)}[10] = R3^{(0)}[7]$) 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, each clock-cycle the produced output bit 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 spoken, it is

$$\alpha_n = \frac{1}{2^n}$$

for the $n$-th cycle after the determination phase and the algorithm will clock on during the postprocessing phase with an expected value of

$$T_{pp} = \frac{1}{\alpha} = 2$$

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

$$T = T_{dp} + T_{pp} = 15\frac{2}{3} + 2 = 17\frac{2}{3}$$

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

$$C \approx 2^{41} \cdot \left(\frac{7}{4}\right)^{11} \cdot 17\frac{2}{3} \approx 2^{54.02}.$$

## 3.4 Deriving the Initial State of the A5/1 Registers

After having found a possible state candidate, i.e., the content of the internal registers $R1$, $R2$, and $R3$ in the state $S^w$ of A5/1 after the warm-up phase (cf. Section 2.1.2), we have to derive the state $S^i$ 101 clock-cycles earlier. The difficulty of reconstructing $S^i$ is that of reversing the irregular clocking manner during the warm-up phase. After having found the corresponding state $S^i$ of the possible state candidate the initialization vector $IV$ and the session key $K_S$ can be extracted easily. This is because the A5/1 is clocked regularly during the initialization phase as described in Section 2.1.2.

Due to the irregular clocking of the registers during the warm-up phase every previous global A5/1 clock-cycle has up to four predecessor states and backtracking the algorithm would be unnecessary complex. Instead, we simply compute the 101 previous MSBs of each of the three registers $R1$, $R2$, and $R3$ because they can be clocked between 0 and 101 times during this warm-up phase. Afterwards we guess the number of cycles each register was actually clocked between the states $S^i$ and $S^w$. Therefore, we can generate at most $102^3 \approx 2^{20}$ possible linear combinations of initial values of the three registers as possible candidates for $S^i$. Now, we have to check which state of them results in $S^w$ after $101 \approx 2^7$ clock-cycles and is the correct $S^i$. Apparently, even this simple approach has only an overall worst-case complexity of less than $2^{20} \cdot 2^7 = 2^{27}$ clock-cycles and should be performed fast in either software or hardware.

As some values will occur much more likely than others starting to guess and check these ones will lead to the correct solution even faster. Therefore we assume that the number of cycles $n$ each register is clocked is binomially distributed with the probability distribution

$$P(X = n) = B(n \mid p, N) = \binom{N}{n} p^n (1-p)^{N-n} \tag{3.1}$$

and the corresponding distribution function

$$F_X(x) = P(X \le x) = \sum_{n=0}^{\lfloor x \rfloor} \binom{N}{n} p^n (1-p)^{N-n} \tag{3.2}$$

where $N = 101$ denotes the number of clock-cycles of the A5/1 warm-up phase, $p = \frac{3}{4}$ the probability for a register being clocked, $E = p \cdot N = \frac{3}{4} \cdot 101 \approx 76$ the expectation value, $V = Np(1-p) = 101 \cdot \frac{3}{4} \cdot \frac{1}{4} \approx 19$ the variance, and $\sigma = \sqrt{V} = \sqrt{Np(1-p)} \approx 4.35$ the standard deviation. The probability distribution $P(X = n)$ with the afore denoted characteristics is shown in Figure 3.5.

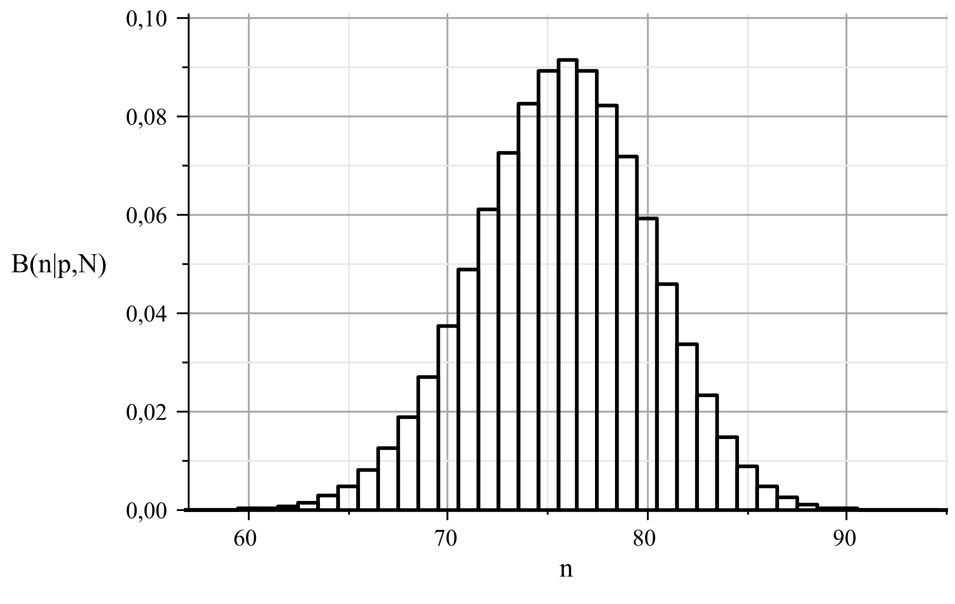


**Figure 3.5:** The probability distribution of the binomially distributed number of clock-cycles of a register

The probability $P$ of $X$ being in a certain interval $n^- \le X \le n^+$ is

$$P(n^- \leq X \leq n^+) = P(X \leq n^+) - P(X \leq n^-). \tag{3.3}$$

If we choose an interval of twice the standard deviation around the expectation value, i.e., choosing the lower and upper bound of the interval to be

$$\begin{aligned} n^- &= E - \lceil 2\sigma \rceil = 76 - 9 = 67, \\ n^+ &= E + \lceil 2\sigma \rceil = 76 + 9 = 85, \end{aligned}$$

the probability that the number of cycles the register was clocked during the $N = 101$ clock-cycles of the A5/1 warm-up phase is inside this interval is greater than 95%. To be exact it is

$$P(67 \leq X \leq 85) = P(X \leq 85) - P(X \leq 67) \approx 0.9903 - 0.0318 \approx 0.9585.$$

This means that we need to perform only

$$(n^+ - n^- + 1)^3 = (85 - 67 + 1)^3 = 19^3 \approx 2^{13}$$

tests in approximately $2^7$ clock-cycles to find the corresponding state $S^i$ with a probability of more than 95%. Only in those few cases the number of cycles the register was clocked is not inside this interval it takes more tests. Comparing the complexity of the determination phase and the postprocessing phase to the one of this part of the attack it is obvious that deriving the initial state of the A5/1 registers is not the bottleneck of the attack.

But we can reduce the complexity of deriving the initial state $S^i$ out of $S^w$ further more. According to the majority function controlling the register clocking, at least two registers are clocked every clock-cycle. The sum of cycles each of the three registers were clocked is between 202 and 303. Because of this, all linear combinations of initial values of $R1$, $R2$, and $R3$ with a cumulated number of cycles being clocked of less than 202 do not need to be checked at all.

# 4 Architecture of the Attack

The architecture of the attack is divided into a *hardware architecture* and a *software architecture*. Both are described in this chapter.

## 4.1 The 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 *control-interface* interconnecting these instances and providing communication to the backplane bus. On each FPGA one of the dedicated DCM units (cf. Section 2.2.1) synthesizes the internal clock out of the global clock of the backplane bus. This is necessary to run the architecture at a frequency higher than $f_{Bus} = 20\,\text{MHz}$. Figure 4.1 gives a top-level overview of the hardware architecture on one FPGA.

Each FPGA is connected to the backplane bus of COPACOBANA and accepts the 64 bit known keystream and a *sub-searchspace* which has to be searched. By sub-searchspace we mean a certain amount of fixed guesses for registers $R1$ and $R2$. Therefore, the software on the host computer divides the *searchspace* consisting of the $2^{41}$ possibilities into these sub-searchspaces and transmits them sequentially together with the known keystream to the FPGAs. One sub-searchspace contains $2^{28}$ possibilities of the whole searchspace and is thus determined by the first 13 MSBs of register $R1$. A 28 bit wide counter of the control-interface counts through the remaining bits and provides each guessing-engine with a fixed guess of registers $R1$ and $R2$ to search on. Every time a guessing-engine finishes its search it sends a status report to the control-interface whether it was successful or not in finding a state candidate. In case of success the valid state candidate is propagated over the control-interface and the backplane bus to the host computer. Afterwards, the guessing-engine requests for another fixed guess of registers $R1$ and $R2$ out of the current sub-searchspace. This is repeated until the whole sub-searchspace was processed by the FPGA. During this, the host computer retrieves regularly at reasonable intervals information on the progress of each FPGA and assigns a new sub-searchspace if requested. The search is finished when all state candidates that can be generated with the $2^{41}$ possibilities for registers $R1$ and $R2$ (i.e., the whole searchspace) are checked for consistency.

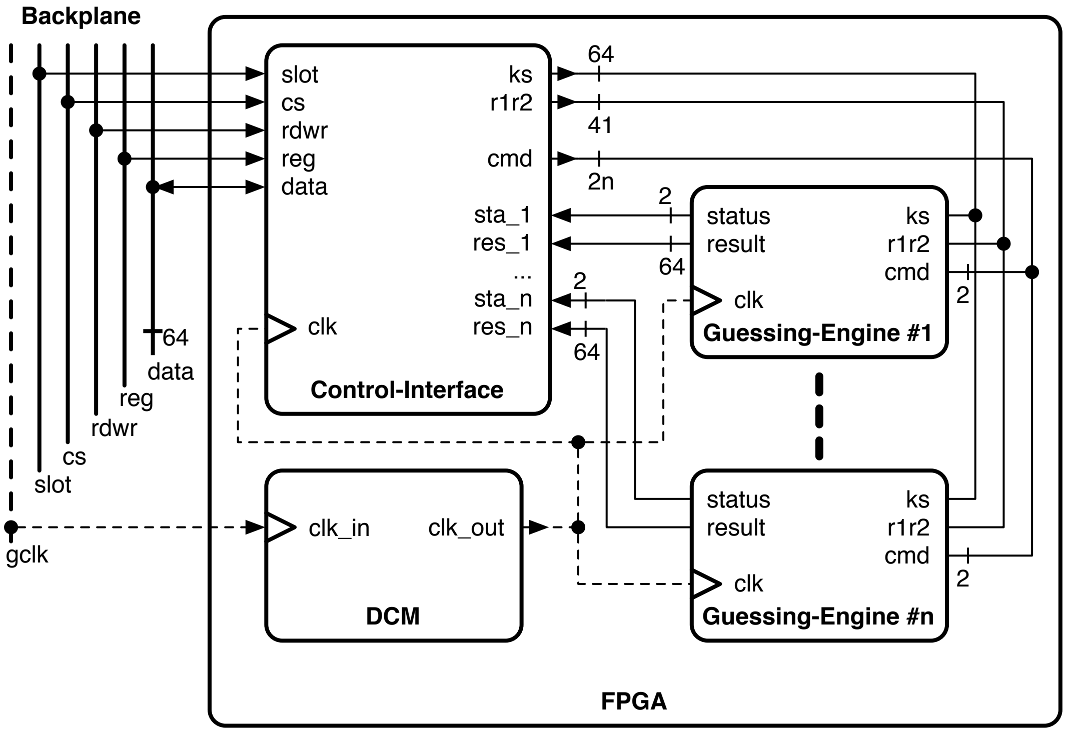


**Figure 4.1:** A top-level overview of the backplane bus, the control-interface, $n$ instances of the guessing-engine, and a dedicated DCM on one FPGA

### 4.1.1 The Guessing-Engine

Figure 4.2 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 initial values of the 64 bit known keystream and of the 41 bit fixed guess of registers $R1$ and $R2$ both coming from the control-interface. Together with a 2 bit status word they are stored in the *communication interface* of the guessing-engine. Additionally, we need all three *A5/1 LFSRs* and a simple 64 bit shift register to evaluate a different known keystream bit every clock-cycle to perform the consistency check in the determination and postprocessing phase. But the most important part of this architecture is the *guessing FSM* (finite state machine) controlling the other components during the two search phases. Its general functionality was already presented with the flowcharts in Figures 3.2 and 3.3. This 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 3.1. This binary decision tree storing the discarded or already checked possibilities is mapped into the *branching state register*. The derived bits of register $R3$ are stored in there, too. Together with the initial values of registers $R1$ and $R2$ they can be put out to the control interface over the 64 bit output `result` in case of a successfully validated state candidate.

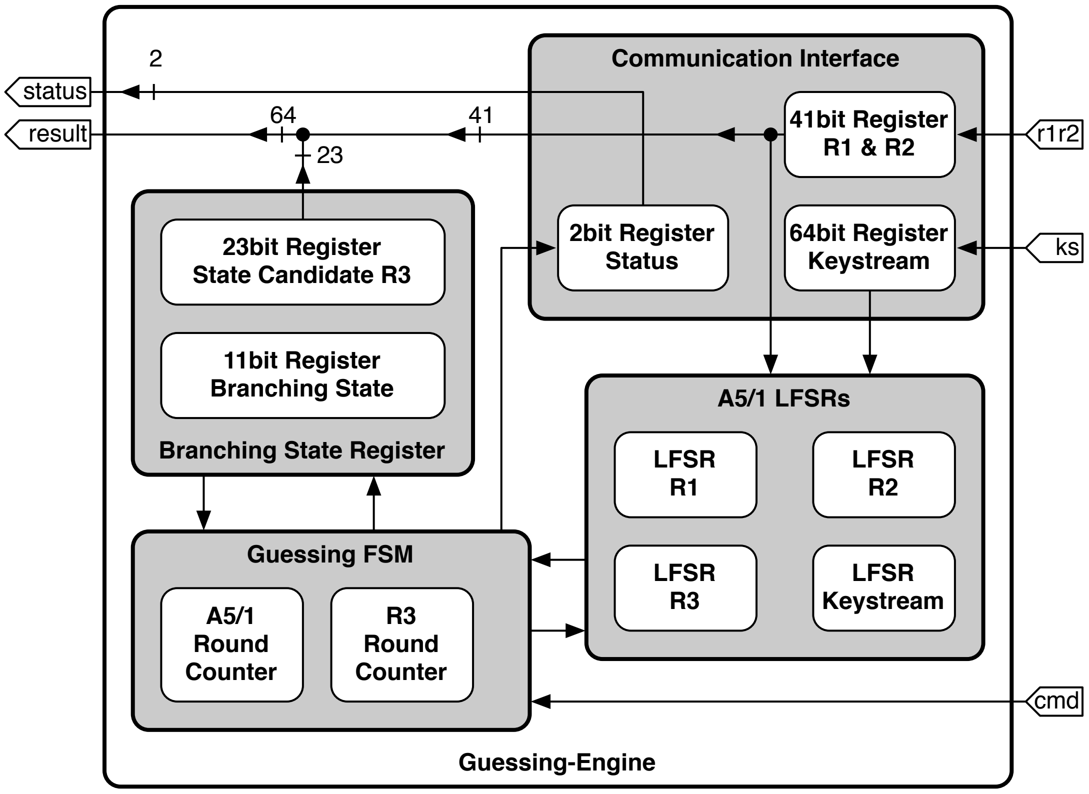


**Figure 4.2:** An overview of the guessing-engine

The most straightforward way of mapping 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., guessing $R3^{(t)}[10] \neq R1^{(t)}[8]$) is represented by 0 in the corresponding counter bit and turning right at a node (i.e., guessing $R3^{(t)}[10] = R1^{(t)}[8]$) 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 demanded by the attack that certain subtrees of the binary decision tree are discarded (cf. Section 3.2). 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 for 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 guessing FSM is designed as a Mealy type FSM which means that the input does not only affect the state transition but also the state output. In our case the input coming from the A5/1 LFSRs influences the guessing of the clocking bits and thus its output. The FSM is divided into three main building blocks: the *state memory*, the *state transition*, and the *state output*. Only the state memory consists of synchronously clocked FFs. The two remaining components are just designed as combinatorial logic. This is possible because all inputs of the FSM come from and all outputs go to synchronously clocked components. Figure 4.3 shows such a three process Mealy type FSM.

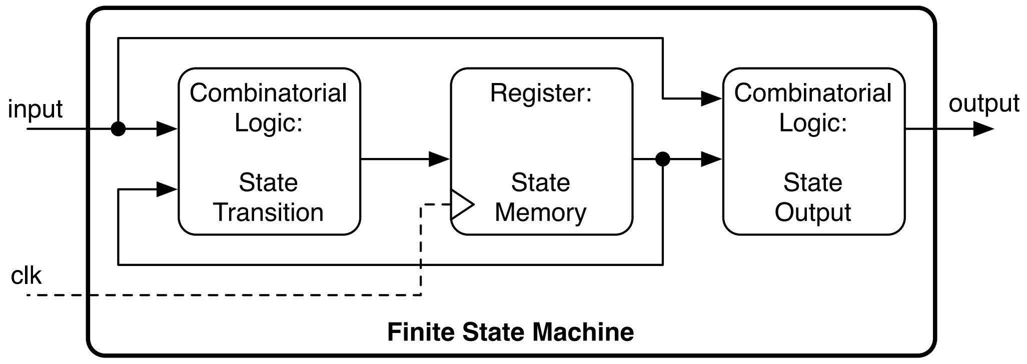


**Figure 4.3:** A finite state machine with three processes

Additionally, the guessing FSM contains two counters. One 6 bit wide synchronous counter to recognize when the whole 64 bit known keystream was evaluated (*A5/1 round counter*) and one 4 bit wide counter to keep track of the number of times register $R3$ was clocked (*R3 round counter*). The latter counter is increased every time register $R3$ is clocked and triggers the transition from the determination phase to the postprocessing phase when reaching a value of 11. The guessing-engine can be controlled by the control-interface over a set of 2 bit wide commands `cmd`. For example, it is possible to reset the guessing-engine or to instruct it to store a new fixed guess of registers $R1$ and $R2$ to search on. These commands are evaluated by the guessing FSM as well. Furthermore, it creates a 2 bit wide status word `status` reporting if the guessing-engine has already finished searching, found a solution, or is waiting for new data.

### 4.1.2 Optimization of the Guessing-Engine: 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. Sections 3.2 and 4.1.1). 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 $R3$

$$f(b) = \frac{b + (11 - b) \cdot 2^{(10-b)}}{2^{(10-b)}} \tag{4.1}$$

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 $b \in \mathbb{N}$.

Taking also into account that some subtrees are discarded while passing through the tree (cf. Section 3.2) and the number of possibilities for guessing one clocking bit of $R3$ is reduced from 2 to $\frac{7}{4}$, the function needs to be adapted:

$$g(b) = \frac{b + (11 - b) \cdot (\frac{7}{4})^{(10-b)}}{(\frac{7}{4})^{(10-b)}}. \tag{4.2}$$

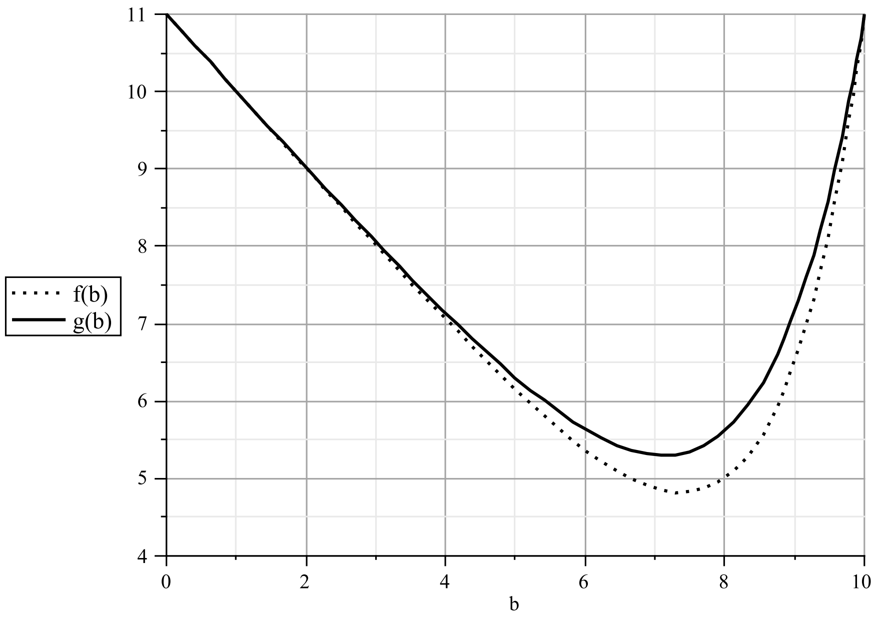


**Figure 4.4:** Functions $f(b), g(b)$: The average number of cycles clocking $R3$ to generate a state candidate with reloading intermediate states at recovery position $b$

Both functions $f(b)$ and $g(b)$ are shown in Figure 4.4. 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

$$T_{opt} = 1 + \frac{4}{3} \cdot 5.31 + 2 \approx 10.10 \approx 2^{3.33} \tag{4.3}$$

instead of $T = 17\frac{2}{3}$ (cf. Section 3.3). This results in an optimized time complexity of

$$C_{opt} \approx 2^{41} \cdot 2^{8.88} \cdot 2^{3.33} \approx 2^{53.21} \tag{4.4}$$

and reduces the previous complexity of $C \approx 54.02$ 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 5.

### 4.1.3 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 shown in Figure 4.5. There is exactly one instance of it implemented on each FPGA of COPACOBANA.

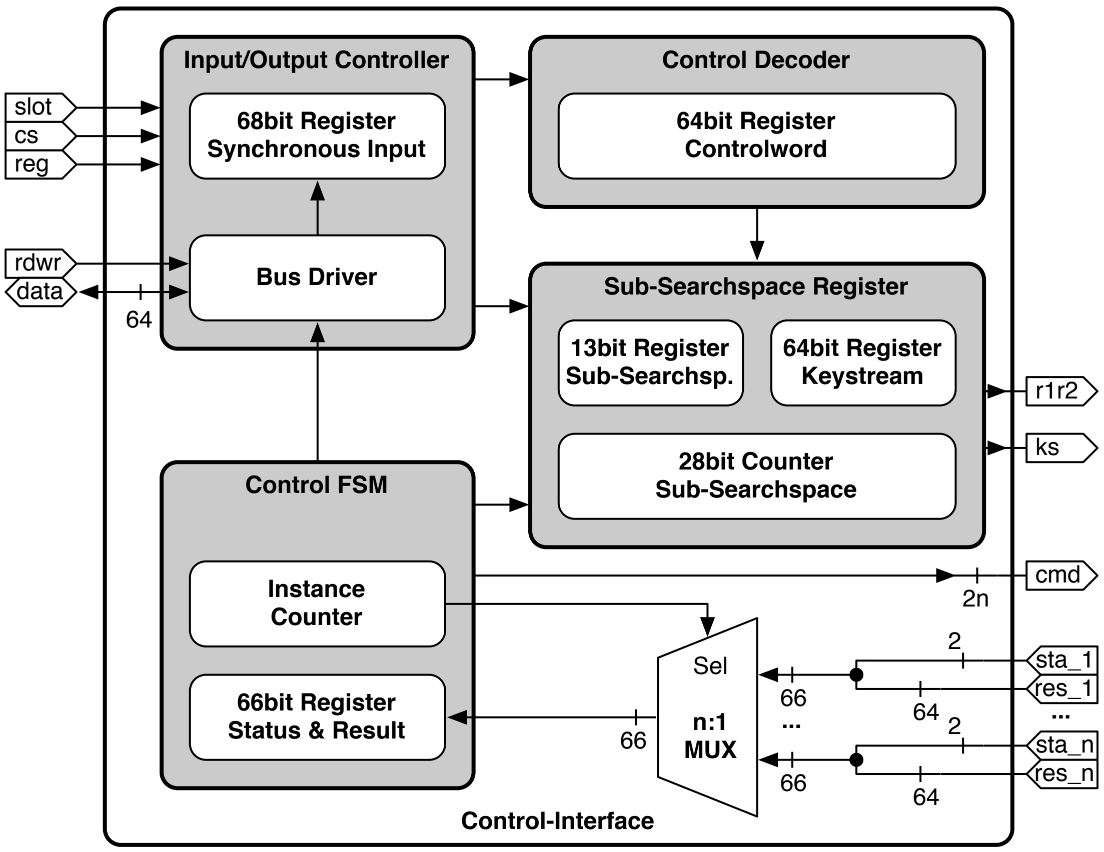


**Figure 4.5:** An overview of the control-interface

To communicate with the backplane it can as well read from as write to the 64 bit data bus over the bidirectional connection `data`. Further inputs `slot`, `cs`, `reg`, and `rdwr` are for controlling this communication. All these inputs coming from the backplane are synchronized to the internal clock by storing them into appropriate registers of the *input/output controller*. The *bus driver* manages the bidirectional communication on the data bus. If the enable signal `rdwr` is set to '1' the bus driver writes the data coming from the *control FSM* to the data bus. Otherwise, if it is set to '0' the output is switched to a high-impedance state and the host computer is allowed to write to the bus. In both cases the signal on the data bus is stored into the *synchronous input* register. This tri-state bus driver allows writing to the data bus from 'both sides'.

Because COPACOBANA houses 120 FPGAs each of them has to be able to be selected separately for communication. Setting the inputs `slot` and `cs` (chip select) both to '0' selects the current FPGA to communicate with. Finally, the input `reg` addresses two type of registers to write to: the *sub-searchspace register* for storing the keystream and the sub-searchspace and a *controlword* register for a certain set of commands.

**Table 4.1:** Inputs of the control-interface coming from the backplane

| | **0** | **1** |
|---|---|---|
| `reg` | address data-register | address control-register |
| `rdwr` | read from bus | write to bus |
| `(slot + cs) = fpga_select` | deselect FPGA | select FPGA |

The sub-searchspace register stores the 13 MSBs of register $R1$ defining the 28 bit wide sub-searchspace and the 64 bit known keystream. Additionally, it provides a 28 bit counter which increases the sub-searchspace by '1' every time one of the guessing-engines finishes its search. Altogether, the sub-searchspace register creates $2^{28}$ fixed guesses of registers $R1$ and $R2$ in this manner.

The *control decoder* evaluates commands coming from the host computer addressed to the control-register (`reg = 1`). Valid commands are *reset*, *store keystream*, *store sub-searchspace*, *start*, *propagate*, *readback keystream*, and *readback sub-searchspace*. These seven commands are passed one-hot encoded to the control FSM of the interface. Table 4.2 summarizes all valid control words and shows the equivalent 5 bit wide bit string sent by the host computer. Furthermore, it lists the states of the control FSM during which the commands are accepted.

The main task of the control FSM is to coordinate the $n$ instances of the guessing-engine on one FPGA. Therefore, it supplies every guessing-engine with the 64 bit known keystream and a different fixed guess of registers $R1$ and $R2$ to search on. Figure 4.6 shows the state transition diagram of the control FSM.

After the *reset* state the sub-searchspace registers need to be loaded with the appropriate data during the *initiate* state. Therefore, the data has to be announced with the *store sub-searchspace* and the *store keystream* command, respectively. Afterwards, it can be sent on the data bus (`reg = 0`) to be stored. For verification reasons the data of the sub-searchspace registers can be read out during this state again with the commands *readback sub-searchspace* and *readback*

**Table 4.2:** Valid control words on the data line addressed to the control register

| **control word** | **bit string** `data[4-0]` | **valid during FSM state** |
|---|---|---|
| *propagate* | `00001` | *success* |
| *store sub-searchspace* | `00010` | *initiate* |
| *store keystream* | `00100` | *initiate* |
| *readback keystream* | `01000` | *initiate* |
| *readback sub-searchspace* | `10000` | *initiate* |
| *start* | `10101` | *initiate, propagate solution* |
| *reset* | `11111` | all states |

*keystream*. The command *start* makes the FSM transit from the *initial* state to the state *searching*. In this state the control FSM requests for the status of the guessing-engines (cf. Section 4.1.1) and reacts accordingly. Therefore, it contains an *instance counter* which counts from 0 to $n-1$ to communicate with each of the $n$ guessing-engines one after the other. Depending on the status the FSM either assigns a new sub-searchspace to the current instance, requests for its solution, or passes on to the next instance without doing anything. The solution as well as the status of the current guessing-engine are selected with the instance counter over a 66 bit $n$:1 multiplexer. If the status of the current guessing-engine indicates that a solution was found the next state of the FSM is the state *success*. In here it is published that a possible state candidate was validated and is ready to be read out. The FSM remains in this state until it is requested via the command *propagate* to broadcast the validated state candidate. To do this, it changes to the state *propagate solution*. The host computer confirms the reception of the results with the command *start*. This makes the FSM go back to the *searching* state. When the 28 bit sub-searchspace counter is completely enumerated and all guessing-engines finished searching on their data the FSM goes into its final state *done* and waits in there until it is reset.

In each state the FSM reports about its own status to the host computer when the FPGA is selected to write to the data bus. With the 8 LSBs of `data` the states *reset*, *initiate*, *searching*, *success*, and *done* of the FSM are encoded. In the state *searching* the 28 bit sub-searchspace counter is additionally sent with the subsequent bits of `data` to inform the host computer about the progress of the search. In the state *propagate solution* all 64 bits are used to broadcast the validated state candidate. Table 4.3 shows an overview of the different status reports of the FSM during its single states.

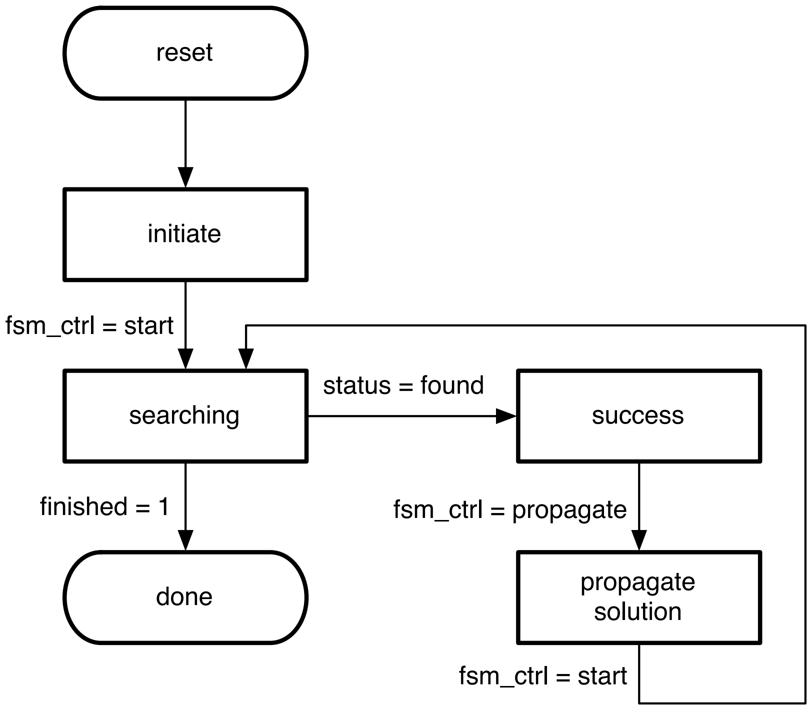


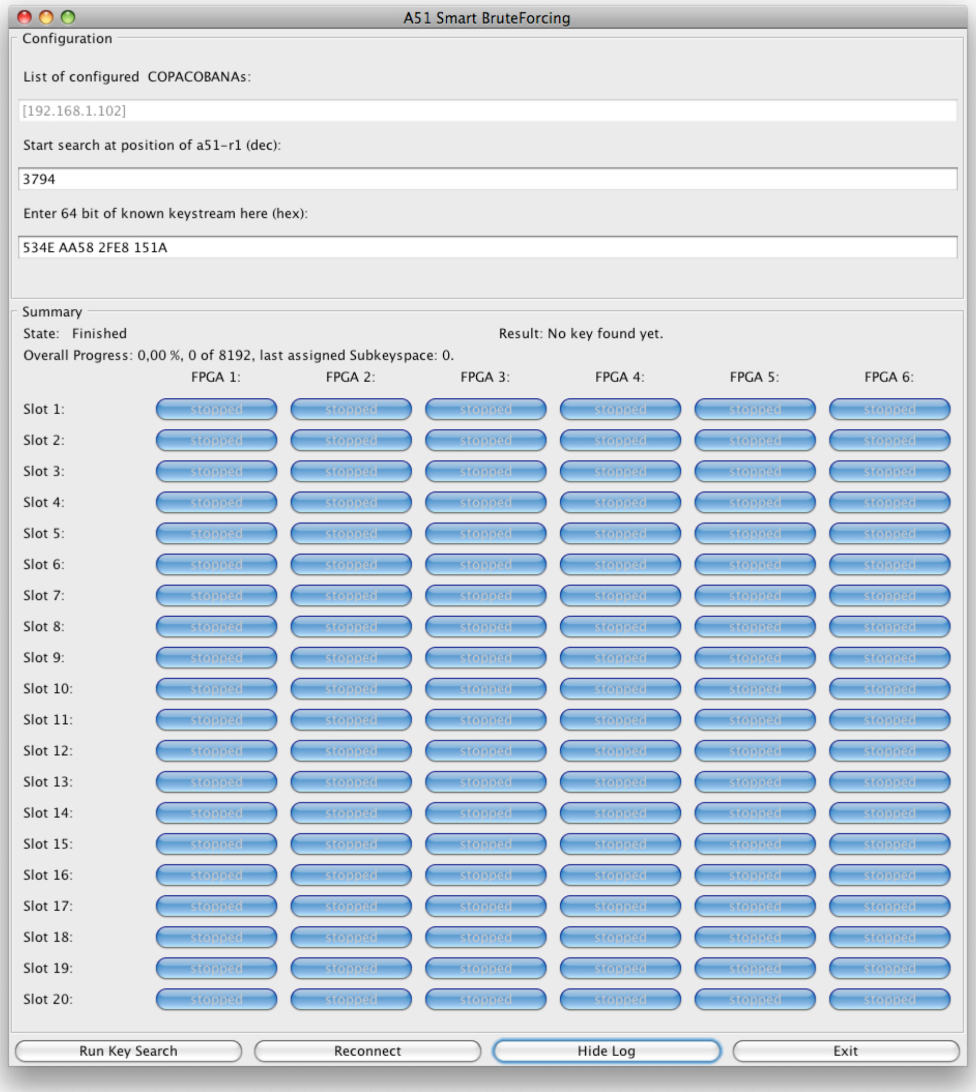
**Figure 4.6:** The state transition diagram of the control FSM

**Table 4.3:** Requested outputs of the FSM on the data line depending on its state

| **FSM state** | `data[63-8]` | `data[7-0]` |
|---|---|---|
| *reset* | `00..00` | `00010001` |
| *initiate* | `00..00` | `00100010` |
| $\hookrightarrow$ *readback keystream* | 64 bit *keystream register* | |
| $\hookrightarrow$ *readback sub-searchspace* | 13 bit *sub-searchspace register* + `00..00` | |
| *searching* | `00..00` + 28 bit *sub-searchspace counter* | `01000100` |
| *success* | `00..00` | `01010101` |
| *propagate solution* | 64 bit validated state candidate | |
| *done* | `00..00` | `10101010` |

## 4.2 The Software Architecture

The task of the software architecture is very similar to the one of the control-interface: it has to control several equally designed engines and supply them with a set of data to search on. More precisely, the software architecture on the host computer controls the 120 FPGAs of COPACOBANA which contain all the same hardware architecture introduced in Section 4.1. The software was implemented in *Java* and Figure 4.7 shows the graphical user interface (GUI) of the architecture on the host computer.



Configuration

List of configured COPACOBANAs:

[192.168.1.102]

Start search at position of a51-r1 (dec):

3794

Enter 64 bit of known keystream here (hex):

534E AA58 2FE8 151A

Summary

State: Finished

Result: No key found yet.

Overall Progress: 0,00 %, 0 of 8192, last assigned Subkeyspace: 0.

| | FPGA 1: | FPGA 2: | FPGA 3: | FPGA 4: | FPGA 5: | FPGA 6: |
|---|---|---|---|---|---|---|
| Slot 1: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 2: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 3: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 4: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 5: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 6: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 7: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 8: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 9: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 10: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 11: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 12: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 13: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 14: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 15: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 16: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 17: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 18: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 19: | stopped | stopped | stopped | stopped | stopped | stopped |
| Slot 20: | stopped | stopped | stopped | stopped | stopped | stopped |

**Figure 4.7:** The GUI of the software architecture

As inputs it accepts an initial value for the sub-searchspace and the 64 bit known keystream. As the 28 bit wide sub-searchspace is defined by the first 13 bits of register $R1$ (cf. Section 4.1) its initial value is supposed to be chosen between 0 and $2^{13} - 1 = 8191$. The keystream is represented by hexadecimal number with 16 digits and LSB first.

When starting a search with the software the FPGAs are first reset, programmed if necessary, and initialized with different but successive sub-searchspaces and the known keystream. Afterwards, all FPGAs are requested for their status one by one. Depending on their status the software either assigns a new sub-searchspace or asks for the solution of the FPGA. If the FPGA reports the status *searching* no interaction is to be performed by the software at all. In this case, only the progress of the FPGA's search, i.e., the value of the 28 bit sub-searchspace counter (cf. Section 4.1.3), is evaluated. In any case the GUI is refreshed accordingly. Every time a new sub-searchspace was assigned it increased by one. The search is finished when all such 8192 subspaces were analyzed by the FPGAs. If a solution was found and propagated by an FPGA it is again displayed as a hexadecimal number of 16 digits consisting of the three concatenated register $R1$, $R2$, and $R3$, all MSB first.

# 5 Implementation Results

In this chapter we present the synthesis and implementation results of the standard and optimized guessing-engine and of the control-interface of our hardware architecture. To finally implement the hardware architecture into the target platform we followed the hardware design process of the design flow we defined in Section 2.2.3. As the design software, we used *Xilinx ISE Foundation 9.2i* to synthesize and implement all components for a *Xilinx Spartan3-XC3S1000-FT256* FPGA as used in COPACOBANA (cf. Section 2.2.2). The simulation of the hardware models of all intermediate steps was done with *MentorGraphics ModelSim SE 6.3d*. To generate test vectors for verification purposes during the simulations we enhanced the *pedagogical implementation of A5/1* [BGW99] written in *C* and adapted it to our needs.

Table 5.1 shows the synthesis results of the standard guessing-engine as described in Section 4.1.1. It lists the amount of slices, flip-flops, look-up tables, and the maximum frequency of the main components and, in summary, of the whole engine. Even though these values are just estimates by the synthesis process at a very early design stage, they help evaluating and optimizing the different components. Furthermore, we can compare the results more easily to those of the optimized guessing-engine.

The synthesis results of the optimized guessing-engine are shown in Table 5.2. Comparing these to the results of the standard version in Table 5.1 shows that it demands for a larger amount of slices (i.e., 'area') of the FPGA. Most of the additional hardware resources needed are flip-flops for storing the intermediate states (cf. Section 4.1.2) of registers $R1$, $R2$, the determined bits of $R3$, and the shifted keystream. Furthermore, the slightly more sophisticated control logic claims for the remaining increased consumption. But because the additional circuits are not located in the critical path, the optimized design is synthesized with nearly the same maximum frequency as the standard version. That it even outperforms the standard version is probably caused by differently applied optimization techniques by the synthesizer.

The next remarkable step of the system implementation is the place & route process. The values for area and speed of the design determined during this implementation step are no longer estimates but correspond to the actually demanded hardware resources. Thus, Table 5.3 shows the results of the fully placed

**Table 5.1:** Synthesis results of the standard guessing-engine

| | slices | | FFs | LUTs | $\mathbf{f_{max}}$ [MHz] |
|---|---|---|---|---|---|
| **A5/1 LFSRs** | | | | | |
| ○ **LFSR Keystream** | 37 | (18 %) | 64 | 64 | 413.39 |
| ○ **LFSR R1** | 12 | (6 %) | 19 | 21 | 232.67 |
| ○ **LFSR R2** | 13 | (6 %) | 22 | 23 | 224.82 |
| ○ **LFSR R3** | 19 | (9 %) | 20 | 34 | 294.29 |
| **Branching State Register** | 27 | (13 %) | 32 | 45 | 222.72 |
| **Guessing FSM** | 37 | (17 %) | 17 | 67 | 177.84 |
| **Communication Interface** | 65 | (31 %) | 113 | 4 | — |
| **standard guessing-engine** | 210 | (100 %) | 287 | 258 | 93.63 |

**Table 5.2:** Synthesis results of the optimized guessing-engine

| | slices | | FFs | LUTs | $\mathbf{f_{max}}$ [MHz] |
|---|---|---|---|---|---|
| **A5/1 LFSRs** | | | | | |
| ○ **LFSR Keystream** | 101 | (30 %) | 128 | 129 | 326.58 |
| ○ **LFSR R1** | 30 | (9 %) | 38 | 40 | 234.41 |
| ○ **LFSR R2** | 35 | (10 %) | 44 | 46 | 226.45 |
| ○ **LFSR R3** | 23 | (7 %) | 27 | 40 | 324.68 |
| **Branching State Register** | 37 | (11 %) | 48 | 62 | 222.72 |
| **Guessing FSM** | 45 | (13 %) | 23 | 77 | 177.84 |
| **Communication Interface** | 69 | (20 %) | 112 | 19 | — |
| **optimized guessing-engine** | 340 | (100 %) | 420 | 413 | 104.02 |

and routed design. First, both guessing-engines, the standard and the optimized one, and the control-interface for one such instance were implemented separately.

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 5.4 shows a comparison of the computing time $T$ and $T_{opt}$ in *clock-cycles* (cf. Sections 3.3 and 4.1.2), 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

**Table 5.3:** Implementation results of the standard guessing-engine and the control-interface

| | slices | FFs | LUTs | $f_{max}$ [MHz] |
|---|---|---|---|---|
| **control-interface** | 371 | 304 | 254 | 123.19 |
| **standard guessing-engine** | 202 | 179 | 256 | 112.84 |
| **optimized guessing-engine** | 311 | 312 | 412 | 115.01 |

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 5.4:** Comparison of the implementation results of the standard and the optimized guessing-engine

| | 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 transferred to the control-interface creating the bottleneck 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 5.5. Additionally, the available resources of one FPGA are listed, too.

**Table 5.5:** 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 5.5 also shows the frequencies the designs were tested with. The first implementation with 36 instances of the standard guessing-engine was tested on COPACOBANA with a system inherent operating frequency of $f = 72\,\text{MHz}$ and the other two implementations with 92 MHz. 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 3.3) we expect a computation time of

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

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

$$t'_{est} = \frac{2^{54.02}}{120 \cdot 32 \cdot 92 \cdot 10^6} \cdot \frac{1}{3600}\,\text{h} \approx 14.36\,\text{h}.$$

For the optimized guessing-engine (23 optimized instances @ 92 MHz) with a time complexity of $C_{opt} = 2^{53.21}$ we expect an computation time of

$$t''_{est} = \frac{2^{53.21}}{120 \cdot 23 \cdot 92 \cdot 10^6} \cdot \frac{1}{3600}\,\text{h} \approx 11.40\,\text{h}.$$

Time measurements of several extended test runs on COPACOBANA showed an average computation time of $t' = 13.58\,\text{h}$ 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 $t'' = 11.78\,\text{h}$ 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. Table 5.6 summarizes the estimated and measured worst case computation time to perform a full search.

**Table 5.6:** Comparison of estimated and measured worst case computation time

| | **computation time [h]** | | **variation** |
|---|---|---|---|
| | **estimated** | **measured** | **[bit]** |
| **1 control-engine &** | | | |
| ○ **36 standard** | 16.31 | — | — |
| ○ **32 standard** | 14.36 | 13.58 | 0.08 |
| ○ **23 optimized** | 11.40 | 11.78 | 0.05 |
| **guessing-engines** | | | |

Albeit, the implementation results shown in this section indicate that there is still potential for further improvements, e.g., reducing the size of the different engines or increasing the operating frequency beyond 92 MHz.

# 6 Conclusions

In this diploma thesis 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 (cf. Section 3.4). 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.

Considering optimization purposes the implementation results shown in Section 5 still leave potential for further improvements of the hardware architecture. Due to the relatively low communication ratio between the guessing-engines and the control-interface it will probably pay off to pipeline the 66 bit $n$:1 multiplexer (cf. Section 4.1.3). Because many guessing-engines work in parallel this communication process is not time-critical. Pipelining it will extend on the one hand the communication process between the engines and their interface by some clock-cycles but will on the other hand reduce the critical path of the design for a maximally utilized FPGA with standard guessing-engines. The second performance limiting component — especially when using the optimized guessing-engine — is the guessing FSM. Putting more effort in mapping the guessing process more efficiently into hardware will lead to a slightly higher performance of the whole design but will probably cause a higher demand for hardware resources.

**Comparison of results.** In Table 6.1 we compare our implementation results of the last section with the estimates of related work already introduced in Section 1.2. Generally, the table lists the necessary amount of known keystream bits (KS) to mount the attack, the success probability, the worst-case computation time, and the costs of the platform the attack was designed for. Additionally, for the last three attacks (TMDTOs), the data that needs to be precomputed is listed as well as the time this precomputation phase takes. Next to the already introduced attacks we include a new TMDTO attack approach by Güneysu, Kasper, Novotný, Paar, and Rupp [GKN+08].

**Table 6.1:** Comparison of attacks against A5/1

| **attack** | **KS** | **succ.** | **precomputation** | | **comp.** | **costs** |
|---|---|---|---|---|---|---|
| | **[bits]** | **prob.** | **data** | **time** | **time** | **[US$]** |
| **our approach:** [**GNR08**] | 64 | 100 % | — | — | 11.78 h | 10,000 |
| [**KS01**] | 64 | 18 % | — | — | 236 d | 100 |
| [**Gol97**], [**PS00**] | 64 | 100 % | — | — | 5 d | 5,000 |
| **TMDTO attacks:** | | | | | | |
| [**BSW01**], [**BS00**] | 25,000 | 60 % | 300 GB | $2^{48}$ clks | minutes | 500 |
| [**BBK03**], [**BBK06**] | — | 60 % | 50 TB | 2800 y | 13.3 m | 500 |
| [**GKN+08**] | 114<br>456 | 55 %<br>96 % | 4.85 TB | 95 d | 5 m – 1.5 h | 10,000 |

In [GKN+08] the authors use thin-rainbow tables together with distinguished points (DP) for their attack to achieve a better time-memory-data tradeoff. Unlike the other TMDTO approaches the precomputation effort is moderate in time and data. Both, the online phase and the precomputation phase are performed on COPACOBANA. During the online phase the attack itself can be performed within a few seconds but it is limited by $2^{20}$ table accesses. Assuming that one table access equals one disk access and thus needs 5 ms the attack time increases to 1.5 h. But parallelizing the table accesses reduces the computation time in the online phase of the attack to 5 minutes. The relatively low success probability of only 55 % can be increased significantly to 96 % if 4 frames of known keystream are available which is still a realistic assumption. This is contrary to our attack where a bigger amount of known keystream has neither an impact on the success probaility nor on the computation time.

Altogether, our attack has next to [GKN$^+$08] the best cost-efficiency also with respect to the full costs including the power consumption during the comparatively short (pre-)computation time.

# Bibliography

[And94] R. Anderson. A5 (was: Hacking digital phones). Newsgroup Communication, 1994.

[Bab95] S. Babbage. A Space/Time Tradeoff in Exhaustive Search Attacks on Stream Ciphers. In *European Convention on Security and Detection*, May 1995.

[BB06] E. Barkan and E. Biham. Conditional Estimators: An Effective Attack on A5/1. In *Proc. of SAC'05*, volume 3897 of *LNCS*, pages 1–19. Springer-Verlag, 2006.

[BBK03] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communications. In *Proc. of Crypto'03*, volume 2729 of *LNCS*. Springer-Verlag, 2003.

[BBK06] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-only Cryptanalysis of GSM Encrypted Communication (full-version). Technical Report CS-2006-07, Technion, 2006.

[BD00] 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.

[BGW99] 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.

[BS00] A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *Proc. of Asiacrypt'00*, volume 1976 of *LNCS*, pages 1–13. Springer-Verlag, 2000.

[BSW01] 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*, pages 1–18. Springer-Verlag, 2001.

[CE08] R. G. Conway and C. Ehrlich. 2008 Corporate Brochure. GSM Association (GSMA), `http://www.gsmworld.com/documents/gsm_brochure.pdf`, 2008.

[cop06] COPACOBANA - Special-Purpose Hardware for Code-Breaking. `http://www.copacobana.org/`, 2006.

[EJ03] P. Ekdahl and T. Johansson. Another Attack on A5/1. *IEEE Transactions on Information Theory*, 49(1):284–289, 2003.

[ETS97] ETSI - European Telecommunications Standards Institute. Digital Cellular Telecommunications System (Phase 2); Security Related Network Functions (GSM 03.20 Version 4.4.1 Release 1997). `http://www.etsi.org`, 1997.

[ETS00] ETSI - European Telecommunications Standards Institute. Digital Cellular Telecommunications System (Phase 2); Security Aspects (GSM 02.09 Version 4.5.1 Release 2000). `http://www.etsi.org`, 2000.

[ETS01] ETSI - European Telecommunications Standards Institute. Digital cellular telecommunications system (Phase 2+); General description of a GSM Public Land Mobile Network (PLMN) (GSM 01.02 Version 6.0.1 Release 1997). `http://www.etsi.org`, 2001.

[GKN+08] T. Güneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers: Special-Purpose Hardware for Cryptography and Cryptanalysis*, to appear, 2008.

[GNR08] T. Gendrullis, M. Novotny, and A. Rupp. A Real-World Attack Breaking A5/1 within Hours. In E. Oswald and P. Rohatgi, editors, *Proc. of CHES'08*, volume 5154 of *LNCS*, pages 266–282. Springer-Verlag, 2008.

[Gol97] J. Golic. Cryptanalysis of Alleged A5 Stream Cipher. In *Proc. of Eurocrypt'97*, volume 1233 of *LNCS*, pages 239–255. Springer-Verlag, 1997.

[Gol00] J. Golic. Cryptanalysis of Three Mutually Clock-Controlled Stop/Go Shift Registers. *IEEE Transactions on Information Theory*, 46:1081–1090, May 2000.

[Hil01] Friedhelm Hillebrand, editor. *GSM and UMTS: The Creation of Global Mobile Communication.* John Wiley & Sons, 2001.

[Jan03] D. Jansen. *The Electronic Design Automation Handbook.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[KPP+06] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In *Proc. of CHES'06*, volume 4249 of *LNCS*, pages 101–118. Springer-Verlag, 2006.

[KS01] J. Keller and B. Seitz. A Hardware-Based Attack on the A5/1 Stream Cipher. `http://pv.fernuni-hagen.de/docs/apc2001-final.pdf`, 2001.

[Lan06] Prof. Dr.-Ing. U. Langmann. Lecture Notes in VLSI Design. Institute of Integrated Systems, Department of Electrical Engineering and Information Sciences, Ruhr-University Bochum, Germany, 2006. `http://www.is.rub.de`.

[MJB05] 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.

[PS00] 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.

[RWO98] Siegmund M. Redl, Matthias K. Weber, and Malcolm W. Oliphant. *GSM and Personal Communications Handbook*. Artech House, 1998.

[Vac06] A. Vachoux. Top-Down Digital Design Flow. Microelectronics System Lab, November 2006.

[Wal01] Bernhard H. Walke. *Mobile Radio Networks*. John Wiley & Sons, 2001.

[Xil99] Xilinx. XC4000E and XC4000X Series Field Programmable Gate Arrays, May 1999.

[Xil07a] Xilinx. Development System Reference Guide. `http://toolbox.xilinx.com/docsan/xilinx92/books/docs/dev/dev.pdf`, 2007.

[Xil07b] Xilinx. Spartan-3 FPGA Family: Complete Data Sheet, DS099. `http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf`, November 2007.