Research Article

Vadim Bulavintsev, Alexander Semenov, Oleg Zaikin*, and Stepan Kochemazov

A Bitslice Implementation of Anderson's Attack on A5/1

https://doi.org/10.1515/eng-2018-0002
Received Oct 02, 2017; accepted Nov 26, 2017

Abstract: The A5/1 keystream generator is a part of Global System for Mobile Communications (GSM) protocol, employed in cellular networks all over the world. Its cryptographic resistance was extensively analyzed in dozens of papers. However, almost all corresponding methods either employ a specific hardware or require an extensive preprocessing stage and significant amounts of memory. In the present study, a bitslice variant of Anderson's Attack on A5/1 is implemented. It requires very little computer memory and no preprocessing. Moreover, the attack can be made even more efficient by harnessing the computing power of modern Graphics Processing Units (GPUs). As a result, using commonly available GPUs this method can quite efficiently recover the secret key using only 64 bits of keystream. To test the performance of the implementation, a volunteer computing project was launched. 10 instances of A5/1 cryptanalysis have been successfully solved in this project in a single week.

Keywords: keystream generator, A5/1, Anderson's attack, GPU, volunteer computing, BOINC

1 Introduction

The A5/1 keystream generator has a key length of 64 bits. It is used to encrypt voice and SMS traffic in 2nd generation (2G) GSM networks. The 3rd generation Global System for Mobile Communications networks (3G GSM) can use the 2G communication protocol to preserve the backward compatibility. The exact authorship of this algorithm is unknown. Its design was first leaked to the general public in 1994. Later, in 1999 the A5/1 algorithm was completely reverse-engineered from a GSM phone.

The A5/1 keystream generator is one of the most well-studied cryptographic algorithms, and it is still actively used. That is why the development of new attacks on A5/1, as well as fast implementations of already known attacks, are relevant. The cryptographic resistance of A5/1 was thoroughly analyzed using various cryptographic methods. One of the first attacks on a non-weakened variant of the algorithm was proposed by R. Anderson [5]. Essentially, Anderson's attack is a guess-and-determine attack [7], based on meticulous analysis of the generator design. Its basic idea is that to determine if a candidate secret key produces a particular keystream fragment, it is sufficient to use only 53 bits out of 64-bit secret key, thus reducing the search space from 2^{64} to 2^{53}. The attack was implemented in 2008 with the help of the special computational platform COPACOBANA [17] based on Field-Programmable Gate Arrays (FPGAs). Using COPACOBANA, it was possible to solve one problem of A5/1 cryptanalysis in approximately seven hours. The main disadvantage of COPACOBANA is that it is an FPGA-based system, using custom-designed circuit boards and requiring significant engineering proficiency to operate.

The most *practical* method for A5/1 cryptanalysis is based on the use of the so-called *Rainbow tables* [1]. Informally, it implies traversing through the space of all possible secret keys (2^{64}), and applying special reduction functions to the keystream fragments to organize the resulting data in the form of interconnected chains, commonly known as rainbow tables. The resulting tables take several weeks in a special distributed computing system to generate and require about 1.5 Terabytes of disk space. When the tables are ready, the cryptanalysis takes at most several minutes per instance. However, the success rate of the rainbow method greatly depends on the size of a keystream fragment. For example, for 8 bursts (912 bits) of keystream, the success rate is about 88.75%. For shorter fragments, the success rate is significantly smaller.

***Corresponding Author: Oleg Zaikin:** Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia, Email: zaikin.icc@gmail.com
Vadim Bulavintsev: Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia, Email: v.g.bulavintsev@gmail.com
Alexander Semenov: Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia, Email: biclop.rambler@yandex.ru
Stepan Kochemazov: Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia, Email: veinamond@gmail.com

In this situation, it is reasonable to complement the rainbow method with some technique, which makes it possible to solve the problem instances not covered by rainbow tables. Ideally, it should be complete, *i.e.* to have a 100% success rate, and require small amount of keystream. Being able to work on a commonly available hardware and to scale with today's inherently parallel computing architectures would be beneficial as well. Thus, in the present paper, the goal was to implement Anderson's attack using mainstream PCs, to be able to perform cryptanalysis of A5/1 in a reasonable time (say, at most a week per problem on a single PC). To do this, a *bitslice* variant of A5/1 is implemented. Bitslice technique implies executing parallel operations on the data stored in processor's registers. In addition to the bitslice variant of A5/1, a bitslice variant of Anderson's attack is implemented. An advantage of this implementation is that it can be readily adapted for executing on modern Graphics Processing Units (GPUs). In particular, the CUDA (Compute Unified Device Architecture) version of this algorithm showed a level of performance that allows one to perform cryptanalysis of A5/1 in about ten days on a mainstream low-tier GPU (Nvidia GeForce GTX 1050 Ti). Since Anderson's attack allows embarrassing parallelization, this implementation scales to any number of GPUs. To test this approach, the volunteer computing project *Andersonattack@home* was launched. In this project, 10 cryptanalysis instances for A5/1 were successfully solved in a single week.

As a result, a method was proposed that, when complemented with the rainbow tables method, provides a practical toolset for cryptanalysis of A5/1 with 100% success rate. It uses commonly available state-of-the-art PC components and works relatively fast for almost any acceptable keystream fragment size.

A brief outline of the paper follows. Section 2 describes the A5/1 algorithm and Anderson's attack on it. Section 3 introduces bitslice technique, implementations of A5/1 and Anderson's attack with it and additional GPU-related details. Section 4 describes the organization of the volunteer project AndersonAttack@home, that was launched to perform the attack, and the results of experiments held in the project. The remaining sections contain a review of the related works and conclude the findings of the present work.

2 A5/1 keystream generator

The A5/1 keystream generator [18] contains 3 linear feedback shift registers (LFSRs [23]), defined by the following primitive polynomials:

$$\begin{aligned} LFSR1: &\quad x^{19} + x^{18} + x^{17} + x^{14} + 1; \\ LFSR2: &\quad x^{22} + x^{28} + 1; \\ LFSR3: &\quad x^{23} + x^{22} + x^{21} + x^{8} + 1. \end{aligned}$$

The illustration of the A5/1 generator's scheme can be seen

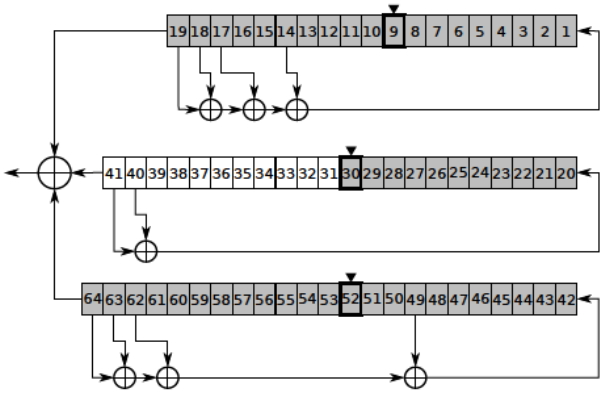

Figure 1: The A5/1 generator scheme. The bits which are guessed in Anderson's attack are greyed out.

in Figure 1. The outputs of the LFSRs are mixed by addition modulo 2 (XOR). The non-linearity of cryptanalysis equations is achieved by asynchronous clocking of individual LFSRs. At each clocking the LFSR with index $j \in \{1, 2, 3\}$ is shifted if the following Boolean function χ_j takes the value of 1:

$$\chi_j = \left(b_j \equiv majority\left(b_1, b_2, b_3\right)\right);$$
$$majority(a, b, c) = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c).$$

Here b_1, b_2, b_3 denote *clocking bits* marked in Figure 1 by black wedges. Thus, if at some moment $\chi_j = 0$, then the corresponding LFSRj is not shifted.

From now on, consider a *known plaintext attack* on A5/1, which is formulated as follows: given a known keystream fragment, to find the state of the generator's LFSRs (64 bits) from which a known keystream fragment was constructed. The length of such fragment should be equal to or greater than 64 bits. The known plaintext attack on A5/1 as used in GSM is possible due to the known vulnerability of the GSM protocol (see [24]).

2.1 Anderson's attack outline

R. Anderson suggested the idea of this attack in 1994 [5]. It is a typical example of a *guess-and-determine attack* (see, for example, [7]).

Suppose that one knows the bits filling LFSR1 and LFSR3, and bits of LFSR2 from the beginning of the register to the clocking bit (bits 20 to 30, see Figure 1). Next, suppose that one knows 64 bits of the keystream. It was shown by R. Anderson, that the remaining 11 unknown bits of LFSR2 can be figured out without any additional guesses. Indeed, the known 53 bits allow computing the clocking schedule for the next 11 shifts of LFSR2. Therefore, using a known clocking schedule, known values of LFSR1 and LFSR3 output bits and known keystream fragment, one can efficiently derive the unknown bits of LFSR2 one by one, by clocking the generator and applying XOR to corresponding keystream bits and output bits of LFSR1 and LFSR3.

The outlined algorithm makes it possible to perform a guess-and-determine attack on the A5/1 generator over the search space of size 2^{53}. The simplicity of the algorithm provides an opportunity to implement it on a specialized computational architecture. One such implementation was built with FPGAs by Gendrullis *et al.* [17]. The following sections describe a variant of implementation of this attack.

3 Bitslice implementations of A5/1 and Anderson's attack

In the present section, the so-called *bitslice* technique is used to construct a fast implementation of Anderson's attack.

First, consider the general ways modern computers are organized. According to the classical taxonomy proposed in [15] there are several primary types of computer architectures. In particular, this study is interested in SISD (Single Instruction Single Data) and SIMD (Single Instruction Multiple Data) entities. As it follows from the name, in SISD architecture a processing unit applies a single instruction to a single piece of data. Conversely, in SIMD architecture a single instruction is applied simultaneously to several pieces of data. It is believed that SISD architecture is the best pick for all-around general purpose computations, while SIMD is useful for applications where most of the calculations are vector-based, such as in multimedia and computer graphics.

Strictly speaking, modern Central Processing Units (CPUs) and Graphics Processing Units (GPUs) should be classified as Multiple Instruction Multiple Data (MIMD) devices. Both device classes have many cores, each core outfitted with several Arithmetic Logic Units (ALUs), capable of processing several data elements simultaneously. However, the programming model of a CPU is SISD-based, with its SIMD capabilities exposed through the special extensions of the instruction set, and its multicore capabilities abstracted through the concept of independent computational threads. The acceptance of SISD model greatly simplifies the development of CPU programs. The actual scheduling of data and instructions are left for compiler and CPU itself. The cores of a modern GPU (Graphics Processing Units) are typical examples of SIMD devices. But thanks to advanced hardware schedulers, each GPU is able to run several programs concurrently. However, GPU programming model directly exposes its SIMD capabilities, which makes building an efficient program for a GPU a considerably harder task. Thus, for the rest of the paper CPUs will be referred as SISD devices, and GPUs as SIMD devices, according to their programming models.

The *bitslice* technique is one method that makes it possible to use general purpose computing units, such as modern CPU cores, as SIMD devices, while staying within SISD programming model. This technique is often referred to as *SIMD Within A Register* (SWAR). The basic building blocks of bitslice technique are the widely used bitwise operations. An outline of how SISD, SIMD and bitslice compare to each other is presented in Figure 2. In this table on a simple example, it is shown that by rearranging the way data is stored in general purpose registers (GPRs), it is possible to use SISD processor core as a SIMD device. However, it comes with a price of packing and unpacking data. In general case, the overhead of such transitions outweighs the potential increase of efficiency from expressing an algorithm in bitslice form. However, for many algorithms, it is possible to avoid such transitions almost completely.

To distinguish operations over single memory cells from *parallel* operations with data, hereinafter the lowercase variables will always stand for individual variables/memory cells, and uppercase variables – for vectors of such variables/memory cells.

The number of algorithms that can benefit from being implemented in bitslice form is, in fact, quite limited due to a lot of reasons. Nevertheless, it is possible to outline quite a large class of such algorithms, which includes the A5/1 algorithm and Anderson's attack on it. Informally speaking, to benefit from bitslice implementation an algorithm must have major parts which natively employ operations from B and various bit shifts. Fortunately, a lot of cryptographic algorithms are natively formulated in operations from B. The bitslice implementation of Data Encryption Standard (DES) [10] is a good example of the usefulness of this technique for cryptography.

Method		Implementation	Operations
SISD		$XOR([0\,0\,0\,x_1],[0\,0\,0\,y_1]) = [0\,0\,0\,z_1]$ $XOR([0\,0\,0\,x_2],[0\,0\,0\,y_2]) = [0\,0\,0\,z_2]$ $XOR([0\,0\,0\,x_3],[0\,0\,0\,y_3]) = [0\,0\,0\,z_3]$ $XOR([0\,0\,0\,x_4],[0\,0\,0\,y_4]) = [0\,0\,0\,z_4]$	4
SIMD		$XOR_{simd}\left(\begin{matrix}[0\,0\,0\,x_1]\\[0\,0\,0\,x_2]\\[0\,0\,0\,x_3]\\[0\,0\,0\,x_4]\end{matrix}, \begin{matrix}[0\,0\,0\,y_1]\\[0\,0\,0\,y_2]\\[0\,0\,0\,y_3]\\[0\,0\,0\,y_4]\end{matrix}\right) = \begin{matrix}[0\,0\,0\,z_1]\\[0\,0\,0\,z_2]\\[0\,0\,0\,z_3]\\[0\,0\,0\,z_4]\end{matrix}$	1
Bitslice	Step 1	$\begin{matrix}[0\,0\,0\,x_1]\\[0\,0\,0\,x_2]\\[0\,0\,0\,x_3]\\[0\,0\,0\,x_4]\end{matrix} \Longrightarrow [x_1\,x_2\,x_3\,x_4],\ \begin{matrix}[0\,0\,0\,y_1]\\[0\,0\,0\,y_2]\\[0\,0\,0\,y_3]\\[0\,0\,0\,y_4]\end{matrix} \Longrightarrow [y_1\,y_2\,y_3\,y_4]$	2*
	Step 2	$XOR_{bitwise}([x_1\,x_2\,x_3\,x_4],[y_1\,y_2\,y_3\,y_4]) = [z_1\,z_2\,z_3\,z_4]$	1
	Step 3	$[z_1\,z_2\,z_3\,z_4] \Longrightarrow \begin{matrix}[0\,0\,0\,z_1]\\[0\,0\,0\,z_2]\\[0\,0\,0\,z_3]\\[0\,0\,0\,z_4]\end{matrix}$	1*

Figure 2: Comparison of SISD, SIMD and Bitslice.

Problem: Compute the result of XOR for 4 pairs of Boolean variables (X_i, Y_i), $X_i, Y_i \in \{True, False\}$, $i = 1, \ldots, 4$ and put it into Boolean variables $Z_i \in \{True, False\}$, $i = 1, \ldots, 4$.

Assumption 1. A General Purpose Register (GPR) of both SISD and SIMD devices stores 4-bit words.

Assumption 2. Boolean variable $R = True$ is stored in GPR as [0 0 0 1] and $R = False$ as [0 0 0 0]. Here, 0 and 1 are individual bits. I.e. without the loss of generality a Boolean variable $R \in \{True, False\}$ is represented in GPR as [0 0 0 r], $r \in \{0, 1\}$.

Comments. Operations marked with a star (*) for bitslice correspond to *packing* data into one register and then *unpacking it*.

Assume that one has some algorithm computing a function f, implemented as a Boolean circuit $C(f)$. Suppose that he needs to calculate the values of the corresponding function $f : \{0, 1\}^n \to \{0, 1\}$ for each of 2^n possible inputs. For each input $X \in \{0, 1\}^n$ the output is obtained as a superposition of the basis functions, according to the circuit $C(f)$. Then if d is the device's GPR capacity, one can simultaneously process d instances of the circuit $C(f)$ in bitslice technique. For the rest of the paper, the process of computation of the function f (represented by the circuit $C(f)$) on a single input from $\{0, 1\}^n$ will be called a *thread*, similar to the computational threads in a SIMD device. To avoid confusion with threads of execution, to refer to the proposed *threads*, the word will always be written in italics. Thus, with the use of bitslice technique, the single SISD processing unit simultaneously executes d *threads*.

3.1 Bitslice Implementation of A5/1

In recent years, the bitslice technique has been used several times to implement the A5/1 generator [1, 9]. However, these implementations were not used for the brute-force cryptanalysis, and the authors of the present work do not know about any attempts to combine Anderson's attack with bitslice technique. The main points of implementing this combination will be described further.

As most of the contemporary computational platforms contain GPRs of width at least 32 bits, from now on assume $d = 32$. Thus, within the present section, if not specified otherwise, the variables with names in uppercase correspond to 32-bit words. Similarly, the operations with these variables are assumed to be bitwise. A computational device is assumed to be capable of performing bitwise variants of functions from the basis $B = \{\wedge, \vee, \neg, \oplus\}$.

The description of the details of bitslice implementation of A5/1 generator follows. Each generator's cell with number n, $n \in \{1, \ldots, 64\}$ is associated with a corresponding word $W_n \in \{0, 1\}^{32}$:

$$\begin{aligned} LFSR1 &: \quad W_1, \ldots, W_{19}; \\ LFSR2 &: \quad W_{20}, \ldots, W_{42}; \\ LFSR3 &: \quad W_{43}, \ldots, W_{64}. \end{aligned}$$

From now on, assume that $W^{'}$ is the value of the word W at the next time moment, for example after the LFSR was shifted or a keystream bit was produced. Then, in bitslice technique, the shifting of the LFSR register (LFSR1 in this example) will take the following form:

$$W_1^{'} = W_{19} \oplus W_{18} \oplus W_{17} \oplus W_{14},$$
$$W_n^{'} = W_{n-1}, \ n \in \{2, \dots 19\},$$

where $\oplus$ is the bitwise addition modulo 2 of 32-bit vectors. The calculation of the 32-bit word $W_{out} \in \{0,1\}^{32}$ containing corresponsing keystream bits will look as follows:

$$W_{out} = W_{19} \oplus W_{41} \oplus W_{64}.$$

The conditional clocking is somewhat more complicated to implement in bitslice technique. First, to know if the LFSRs should be shifted or not, one needs to calculate the corresponding shifting flags F_1, F_2, F_3 using the majority function:

$$W_{maj} = majority\,(W_9, W_{30}, W_{52}),$$
$$F_1 = W_9 \oplus \neg W_{maj},$$
$$F_2 = \oplus \neg W_{maj},$$
$$F_3 = \oplus \neg W_{maj}.$$

If *majority* is not implemented as a single bitwise instruction, it can be represented via a superposition of bitwise instructions corresponding to that from B.

To implement the conditional shifting of an LFSR one can use the bitwise counterpart of the *bitselect* function of arity 3:

$$bitselect(x, y, z) = \begin{cases} y, x = 1 \\ z, x = 0 \end{cases}, x, y, z, \in \{0, 1\}.$$

If the computational architecture lacks the hardware implementation of this function, it can be emulated via standard bitwise functions corresponding to the matching functions from the basis B:

$$bitselect(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z).$$

Shifting LFSR1 using the bitwise version of *bitselect* and the corresponding shifting flag F_1 looks the following way:

$$W_1^{'} = bitselect\,(F_1, (W_{19} \oplus W_{18} \oplus W_{17} \oplus W_{14}), W_1);$$
$$W_n^{'} = bitselect\,(F_1, W_{n-1}, W_n), \ n \in \{2, \dots 19\},$$

The shifting procedures for LFSR2-3 are conducted similarly. Now consider the Anderson's attack in the context of bitslice technique.

3.2 Bitslice implementation of Anderson's attack on A5/1

The attack consists of 2 stages:

1. Clock the generator until LFSR2 shifts 11 times, by using the information from the guessed 53 bits and the known keystream to recover the unknown bits of LFSR2 and consequently calculate its new bits;
2. Clock the generator as usual to check if the guessed initial state of the generator matches the known keystream.

The irregular clocking of the A5/1 generator makes it impossible to predict how many clockings of the generator (bits of keystream) would be needed to shift LFSR2 11 times to complete Stage 1. In general, the LFRS2 shift count will be different for each *thread*. Therefore, the *bitselect* function is again put to use to implement the separation of the attack into 2 stages.

To allow each *thread* to be able to advance from Stage 1 to Stage 2 independently of other *threads*, the special Boolean vector $\Phi = (\phi_1, \dots, \phi_d)$ is introduced. It is called the *attack stage flag*. For the *thread* with number $i, i \in \{1, \dots, d\}$, Stage 1 of the attack corresponds to $\varphi_i = 0$, and Stage 2 corresponds to $\varphi_i = 1$. The transition from Stage 1 to Stage 2 happens when LFSR2 have been shifted 11 times. To count the number of times LFSR2 was shifted, the incremental counter (also implemented in bitslice technique) is used.

Let y be the currently analyzed bit of the keystream. Assume, that $Y = (y, \dots, y)$ is a vector of 32 ones if $y = 1$ and of 32 zeros otherwise. Now the shifting of LFSR2 takes into account the stage of the attack through the use of the attack stage flag:

$$W_{41}^{\star} = bitselect\,(\Phi, W_{41}, (Y \oplus W_{19} \oplus W_{64}));$$
$$W_{20}^{'} = bitselect\left(F_2, \left(W_{41}^{\star} \oplus W_{40}\right) W_{20}\right);$$
$$W_n^{'} = bitselect\,(F_2, W_{n-1}, W_n), \ n \in \{21, \dots 41\}.$$

Here $W_{41}^{\star}$ is a helper vector holding temporary value of W_{41}.

When the attack flag $\varphi_i = 0$, the *thread i* is at Stage 1 of the attack. In this case, at each clocking of the generator, the auxiliary variable $W_{41}^{\star}$ associated with the output bit of LFSR2 is computed first. For this purpose the algorithm computes $Y \oplus W_{19} \oplus W_{64}$ – the XOR of the current known keystream bit and the output bits of other two LFSRs. Remind, that the value of attack flag F_2 is known beforehand because of the structure of Anderson's attack. After computing the value of output bit $W_{41}^{\star}$ the LFSR2 is shifted as usual. However, to track when one needs to transition to

Stage 2 of the attack if LFSR2 is shifted (because of clocking flag $F_2 = 1$), the incremental counter's value is increased by 1. When the counter's value reaches 11, the attack stage flag is set to 1, meaning that (for *thread i*) the attack proceeds to Stage 2. At Stage 2 the incremental counter is never increased.

Description of details regarding this implementation of the incremental counter follows. As number 11 in binary form looks like 1011, the bitslice-based incremental counter uses 4 variables (words of length 32) to hold its current state for 32 *threads*. The counter is implemented as a standard serial binary adder. At Stage 2 the whole generator state is known, and the generator is clocked as usual.It is worth mentioning that in case when LFSR2 is not shifted two times in a row, there could arise a conflict when assigning W_{20}. This situation provides an opportunity for early rejection of the key checked in the *thread*, but requires explicit handling in the code to ensure protection from false keys.

During Stage 2, at each clocking of the generator its output W_{out} is compared with the corresponding bit of the known keystream Y, and if they do not match, the *dead thread flag* E is set to 1:

$$E^{'} = E \vee (W_{out} \oplus Y)$$

When all 32 bits of the word E become 1, the calculations stops – this means that all the *threads* in the current bitslice-batch contain wrong keys. If after 64 clockings at least one bit in E is 0, then the corresponding *thread* contains the correct key.

3.3 GPU-related issues

One significant advantage of this implementation of the bitslice variant of Anderson's attack is that it, informally speaking, is quite undemanding. It can work on one CPU core, and it can be implemented for several CPU cores (via data parallelism) or even for several GPU cores. Thankfully, state-of-the-art GPUs have hardware support for bitselect operation and sufficient number of GPRs to run the algorithm.

The variant of Anderson's attack described above is implemented on an NVIDIA GPU with the use of CUDA SDK 8.0 [13]. The comparison of the performance of GPU and CPU bitslice implementations of Anderson's attack is shown in Table 1. The last row of Table 1 corresponds to the case, in which bitselect function was implemented using the LOP3.LUT instruction (LOP3.LUT is a special instruction that performs arbitrary bitwise functions of arity 3 in hardware). GPU implementation uses 32-bit data type, while CPU uses 64-bit data type. CPU application runs 8 threads to make use of all available physical and virtual cores.

Table 1: Performance of bitslice implementation of Anderson's attack (search space size is 2^{53}) on a CPU and a GPU, measured in millions of partial keys (53 bits out of 64) per second.

Computational device	Partial keys per second $\times 10^6$
CPU Intel Core i7 3770	368
GPU NVIDIA GTX 1050 Ti	9180
GPU NVIDIA GTX 1050 Ti (LOP3.LUT)	11950

Data provided in Table 1 shows that even one midrange consumer GPU is enough to make runtime of Anderson's attack practical (it would take around 250 hours). The speed-up obtained with the shown GPU in comparison to used CPU is about 32.5 times (using LOP3.LUT). A modern computational cluster outfitted with GPUs would perform the attack in mere minutes.

4 Implementation of Anderson's Attack in a GPU-based volunteer computing project

To test in practice the CUDA implementation described in Section 3, 10 cryptanalysis instances for the A5/1 keystream generator were constructed by randomly generating 10 initial states and producing the corresponding keystream fragments of size 64 bits.

According to the runtime estimation, presented in Section 3, it would take about 3-4 months to solve all these instances on a single GPU. Meanwhile, any supercomputer equipped with state-of-the-art GPUs can cope with them in a reasonable time. However, at that moment the authors did not have access to such a supercomputer. Therefore, it was decided to employ volunteer computing [4] to run the experiment.

4.1 Volunteer Computing and BOINC

Being a form of distributed computing, volunteer computing suits well to solving computationally hard problems that can be decomposed into independent subproblems (*i.e.* the problems that are embarrassingly parallel [16]).

A volunteer computing project must have its own website and a server, which distributes tasks via the Internet. These tasks are downloaded and processed by computers (hosts) of private persons. Volunteer computing is cheap – to maintain a project one only needs an online server working 24/7. The most popular platform for organizing volunteer computing projects is BOINC Berkley Open Infrastructure for Network Computing [3]).

Some volunteers' hosts can return incorrect results. It could be done by volunteers intentionally. Another possible reason is a hardware malfunction, which could be a consequence of overclocking. To achieve the reliability of calculations, several replication policies are implemented in BOINC. Under the default replication policy, several (at least 2) identical tasks ("results" according to the BOINC terminology) are created for each workunit. Here workunit stands for a unit of work to be performed on hosts. Tasks which belong to the same workunit are required to be processed on different hosts. The results are compared on the project server, and become accepted only when a consensus is reached (the sufficient number of successful results is called a *quorum*). The adaptive replication avoids replicating jobs that are sent to highly reliable hosts.

There are several actions which can boost a BOINCbased project's performance [21]. The majority of these actions deal with credits – a numerical measure of performed calculations. If a task was successfully processed and validated, then credits are granted to the corresponding host (and user). If a result is returned past the given deadline, or it turned out to be invalid, then no credits are granted, and a new task is generated for the workunit.

The scheme of a BOINC-based project with the quorum of 2 is shown in Figure 3. In this figure several server daemons are mentioned: work_generator produces workunits and corresponding tasks; validator checks results provided by volunteers, assimilator constructs solutions of the original problems from checked results.

4.2 AndersonAttack@home project

In order to solve the 10 instances mentioned above, the BOINC-based volunteer computing project AndersonAttack@home [6] was launched. The client application of the project was based on the CUDA implementation (with LOP3.LUT instruction), which was described in Section 3. The Windows and Linux versions of client application were deployed. The default replication policy with quorum of 2 was used. Therefore, the scheme of the project is similar to the one shown in Figure 3.

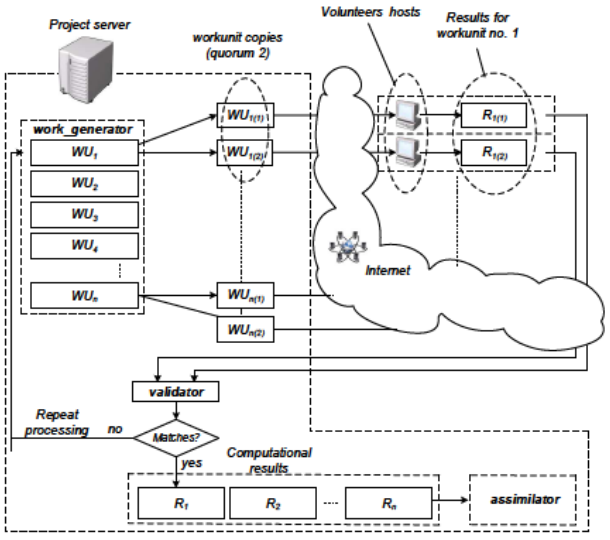

Figure 3: The scheme of a BOINC-based project with quorum of 2

In the first stage of the experiment, a family of workunits was generated on the project server. In each workunit the values of 12 out of 53 bits (see Figure 1 in Section 2) were fixed. Thus, 40960 workunits were generated for 10 instances in total. In the next stage, for every workunit there were constructed 2 tasks (since quorum size was 2), so 81920 tasks were initially generated. The experiment took 7 days. In the course of this experiment 108485 tasks were processed in total. 81920 tasks had the "success" status, while the remaining 26565 tasks had various error statuses. Two most common error statuses were "aborted by user" (15239 tasks) and "computational error" (6505 tasks). On average, it took 248 seconds to successfully process a single task on a single GPU, while tasks with error results required very few resources. Therefore, the experiment could be performed on a dedicated computer equipped with a single modern GPU in about 118 days (without any replication). Taking into account that the experiment's duration was 7 days, one can conclude that the project's performance was comparable to that of a computational cluster equipped with about 15 modern GPUs.

To perform the calculations in the project quickly, an improved task schedule could have been used (*e.g.*, [22]), but another strategy was chosen instead. Usually, the tasks deadline in BOINC projects is about 10-14 days. The deadline was set to 1 day instead. As a result, the volunteer computing project was successfully used in an unusual way – to quickly perform a simple experiment. Thankfully, according to the obtained feedback, it was quite convenient for the volunteers.

As a result of the experiment, solutions for 10 considered cryptanalysis instances were successfully found (see Table 2).

The discussion of obtained results follows. For 7 out of 10 instances, several (up to 3) solutions were found. It was shown in [18], that different initial states of A5/1 can produce identical output sequences. The problem of finding all solutions for the considered instances was not studied. In particular, the application stopped the processing of a task, when it encountered a solution. However, the application could be easily modified to solve the problem mentioned above.

According to the server statistics, 143 active hosts belonging to 90 volunteers participated in the experiment. Here, the *active host* means a host which processed at least one task.

Table 2: The solutions found for 10 cryptanalysis instances of the A5/1 generator (in hexadecimal format).

	Keystream	Initial state	
1	0x770c0410869366f1	no. 1	0x11b8e4340276c4ee
		no. 2	0x42634f3266d302a3
2	0xae9590560c26e9ed	no. 1	0x4c656fd73e59ab9b
		no. 2	0xcf23e4722e3cfb68
3	0xdd4b3ab7f6cf8224	no. 1	0x09429d158555f4b3
		no. 2	0x09429d158553e967
		no. 3	0x40e5f2c8128a1781
4	0x93cd42d97eb75fd9	no. 1	0xfa386a338355aafd
		no. 2	0xf9e81096bb4d0aad
		no. 3	0xf9e81096bb4a8556
5	0x925e423c98121152	no. 1	0xe5cf81035ce5fbe2
6	0x3b3464bd6e377b87	no. 1	0x9625e9d810b46248
		no. 2	0xf5aa1be2d6c36e18
7	0x0367d29121dd1677	no. 1	0xd1b8b06086edf162
8	0x6b49230b7fc0249d	no. 1	0xbe81a896968c486b
9	0xc65847556752d14c	no. 1	0xb6f65d2855a211c0
		no. 2	0xb6f65d2855a508e0
10	0x07bb7f83d26072ec	no. 1	0x122a1a2955286b9f
		no. 2	0xd5151aaa50490012

5 Related work

R. Anderson's note (a mailing list message) written in 1994 was one of the first works on the A5/1 cryptanalysis. Shortly afterward J. Golic suggested the attack on "Alleged" A5/1, based on linearization of the equations describing A5/1 [18]. The Golic's attack estimated complexity is $C \cdot 2^{40}$, where C denotes the complexity of solving a system of linear equations over $GF(2)$. The next widely known attack on the A5/1 was presented in [12]. To speed up the A5/1 generator the authors of that paper used the technique of precomputation of LFSRs. Note, that the attack presented in [12], as well as those from [8, 11, 14], requires a keystream fragment of a considerable length (corresponding to at least several seconds of a conversation).

Another class of attacks uses a small amount of keystream, but perform a lot of computations. One of the first attacks of that kind was presented in [17]. Its authors implemented the optimized variant of Anderson's attack on a specialized computational device of their own design, assembled from 120 'Xilinx Spartan 3' FPGAs. They state that the attack took about 6 hours. It is worth mentioning that the COPACOBANA architecture was used to analyze several other ciphers, *e.g.*, DES [19].

A different attack on A5/1 that uses small fragment of keystream reduced the problem of cryptanalysis of the generator to Boolean Satisfiability problem (SAT) [25–29]. This reduction made it possible to construct a guess-and-determine attack with a much smaller set of bits to guess (30-33 variables) compared to Anderson's attack. In particular, 20 cryptanalysis instances of the A5/1 generator were solved in the volunteer computing project SAT@home using the CluBORun tool [2].

There exist other attacks on A5/1 that use specific methods to reduce the search space [20]. The work [19] includes the estimates of time and disk space required to create the rainbow-tables that would make it possible to "break" A5/1 on an average PC in several minutes. [19] estimated that these tables would require about 7 Tb of disk space. The A5/1 Cracking Project put the significantly more compact (about 2 Tb) tables into the public domain at the end of 2009 [24]. By analyzing 2 frames (912 bits) of known keystream with the help of these tables one can restore the secret key with the probability of success over 85%. This method of cryptanalysis of the A5/1 algorithm could be assumed to be the most practical one.

6 Conclusion

The main contribution of the present study is a bitslice implementation of Anderson's attack on the A5/1 keystream generator that runs on a consumer-grade hardware (GPUs and CPUs). This implementation is thoroughly adapted to the SIMD architecture. Using this implementation, several cryptanalysis problems were solved in a GPU-based volunteer computing project in a single week. In the future, the authors are planning to use volunteer computing for cryptanalysis of several other keystream generators.

Acknowledgement: The research was funded by Russian Science Foundation (project No. 16-11-10046). Authors thank all volunteers, whose computers took part in the experiment.

References

[1] A5/1 cracking project, https://opensource.srlabs.de/projects/a51-decrypt.

[2] A. P. Afanasiev, I. V. Bychkov, O. S. Zaikin, M. O. Manzyuk, M. A. Posypkin, and A. A. Semenov. Concept of a multitask grid system with a flexible allocation of idle computational resources of supercomputers. Journal of Computer and Systems Sciences International, 56(4):701–707, Jul 2017.

[3] David P. Anderson. BOINC: A system for public-resource computing and storage. In Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, GRID'04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[4] David P. Anderson and Gilles Fedak. The computational and storage potential of volunteer computing. In Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), 16-19 May 2006, Singapore, pages 73–80. IEEE Computer Society, 2006.

[5] Ross Anderson. A5 (was: Hacking digital phones). http://yarchive.net/phone/gsmcipher.html. Newsgroup Communication, 1994.

[6] AndersonAttack@home: a volunteer computing project aimed at solving A5/1 cryptanalysis problems, http://www.parlea.ru/andersonattack/.

[7] Gregory V. Bard. Algebraic Cryptanalysis. Springer Publishing Company, Incorporated, 1st edition, 2009.

[8] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of gsm encrypted communication. Journal of Cryptology, 21(3):392–429, 2008.

[9] Andreas Beckmann, Jaroslaw Fedorowicz, Jörg Keller, and Ulrich Meyer. A structural analysis of the a5/1 state transition graph. arXiv preprint arXiv:1210.6411, 2012.

[10] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, Fast Software Encryption, 4th Inter- national Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings, volume 1267 of Lecture Notes in Computer Science, pages 260–272. Springer, 1997.

[11] Eli Biham and Orr Dunkelman. Cryptanalysis of the A5/1 GSM stream cipher. In Bimal Roy and Eiji Okamoto, editors, Progress in Cryptology — INDOCRYPT 2000: First International Conference in Cryptology in India Calcutta, India, December 10–13, 2000 Proceedings, pages 43–51, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[12] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In Bruce Schneier, editor, Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings, volume 1978 of Lecture Notes in Computer Science, pages 1–18. Springer, 2000.

[13] CUDA Software Development Kit 8.0, https://developer.nvidia.com/cuda-toolkit.

[14] Patrik Ekdahl and Thomas Johansson. Another attack on A5/1. IEEE Trans. Information Theory, 49(1):284–289, 2003.

[15] Michael J. Flynn. Some computer organizations and their effectiveness. IEEE Trans. Comput., 21(9):948–960, September 1972.

[16] Ian Foster. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. AddisonWesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] Timo Gendrullis, Martin Novotný, and Andy Rupp. A realworld attack breaking A5/1 within hours. In Elisabeth Oswald and Pankaj Rohatgi, editors, Cryptographic Hardware and Embedded Systems CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings, volume 5154 of Lecture Notes in Computer Science, pages 266–282. Springer, 2008.

[18] Jovan Dj. Golic. Cryptanalysis of alleged A5 stream cipher. In Walter Fumy, editor, Advances in Cryptology EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding, volume 1233 of Lecture Notes in Computer Science, pages 239–255. Springer, 1997.

[19] Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. IEEE Trans. Comput., 57(11):1498–1513, November 2008.

[20] S. A. Kiselev and N. N. Tokareva. Reduction of the key space of the cipher A5/1 and invertibility of the next-state function for a stream generator. Journal of Applied and Industrial Mathematics, 6(2):194–202, Apr 2012.

[21] Ilya Kurochkin and Anatoliy Saevskiy. BOINC forks, issues and directions of development1. Procedia Computer Science, 101(Supplement C):369–378, 2016. 5th International Young Scientist Conference on Computational Science, YSC 2016, 26-28 October 2016, Krakow, Poland.

[22] Vladimir V. Mazalov, Natalia N. Nikitina, and Evgeny E. Ivashko. Task scheduling in a desktop grid to minimize the server load. In Proceedings of the 13th International Conference on Parallel Computing Technologies Volume 9251, pages 273–278, New York, NY, USA, 2015. SpringerVerlag New York, Inc.

[23] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[24] Karsten Nohl. Attacking phone privacy. In BlackHat 2010 Lecture Notes, Las-Vegas, USA, July 28-29, 2010, pages 1–6, 2010.

[25] Mikhail Posypkin, Oleg Zaikin, Dmirty Bespalov, and Alexander Semenov. Cryptanalysis of stream ciphers in distributed computing systems (in Russian). Proceedings of ISA RAS, 46:119–137, 2009.

[26] Alexander Semenov and Oleg Zaikin. Using Monte Carlo method for searching partitionings of hard variants of Boolean satisfiability problem. In Victor Malyshkin, editor, Parallel Computing Technologies 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 September 4, 2015, Proceedings, volume 9251 of Lecture Notes in Computer Science, pages 222–230. Springer, 2015.

[27] Alexander Semenov and Oleg Zaikin. Algorithm for finding partitionings of hard variants of Boolean satisfiability problem with application to inversion of some cryptographic functions. SpringerPlus, 5(1):1–16, 2016.

[28] Alexander Semenov, Oleg Zaikin, Dmitry Bespalov, Pavel Burov, and Alexey Hmelnov. Solving discrete functions inversion problems on multiprocessor computing systems (in Russian). In Proceedings on Parallel computing and Control Problems (PACO'2008), Moscow, Russia, October 27-29, 2008, pages

152–176, 2008.

[29] Alexander Semenov, Oleg Zaikin, Dmitry Bespalov, and Mikhail Posypkin. Parallel logical cryptanalysis of the generator A5/1 in bnb-grid system. In Victor Malyshkin, editor, Parallel Computing Technologies - 11th International Conference, PaCT 2011, Kazan, Russia, September 19-23, 2011. Proceedings, volume 6873 of Lecture Notes in Computer Science, pages 473–483. Springer, 2011.