Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128

Jean-Philippe Aumasson[1,*], Itai Dinur[2], Luca Henzen[3], Willi Meier[1,†], and Adi Shamir[2]

[1] FHNW, Windisch, Switzerland
[2] Weizmann Institute, Rehovot, Israel
[3] ETH Zurich, Switzerland

Abstract. Cube testers are a generic class of methods for building distinguishers, based on cube attacks and on algebraic property-testers. In this paper, we report on an efficient FPGA implementation of cube testers on the stream cipher Grain-128. Our best result (a distinguisher on Grain-128 reduced to 237 rounds, out of 256) was achieved after a computation involving 2^{54} clockings of Grain-128, with a 256×32 parallelization. An extrapolation of our results with standard methods suggests the possibility of a distinguishing attack on the full Grain-128 in time 2^{83}, which is well below the 2^{128} complexity of exhaustive search. We also describe the method used for finding good cubes (a simple evolutionary algorithm), and report preliminary results on Grain-v1 obtained with a bitsliced C implementation. For instance, running a 30-dimensional cube tester on Grain-128 takes 10 seconds with our FPGA machine, against about 45 minutes with our bitsliced C implementation, and more than a day with a straightforward C implementation.

1 Introduction

The stream cipher Grain-128 was proposed by Hell, Johansson, Maximov, and Meier [16] as a variant of Grain-v1 [17,18], to accept keys of up to 128 bits, instead of up to 80 bits. Grain-v1 has been selected in the eSTREAM portfolio[4] of promising stream ciphers for hardware, and Grain-128 was expected to retain the merits of Grain-v1.

Grain-128 takes as input a 128-bit key and a 96-bit IV, and it produces a keystream after 256 rounds of initialization. Each round corresponds to clocking two feedback registers (a linear one, and a nonlinear one). Several attacks on Grain-128 were reported: [22] claims to detect nonrandomness on up to 313 rounds, but these results were not documented, and not confirmed by [9], which used similar methods to find a distinguisher on 192 rounds. Shortcut key-recovery attacks on 180 rounds were presented in [10], while [5] exploited a sliding property to speed up exhaustive search by a factor two. More recently, [21] presented related-key attacks on the full Grain-128. However, the relevance of related-key attacks is disputed, and no attack significantly faster than bruteforce is known for Grain-128 in the standard attack model.

The generic class of methods known as *cube testers* [1], based on cube attacks [8] and on algebraic property-testers, aims to detect non-randomness in cryptographic algorithms, via multiple queries with chosen values for the IV bits (more generally, referred to as public variables). Both cube attacks and cube testers sum the output of a cryptographic function over a *subset* of its inputs. Over GF(2), this summation can be viewed as high-order differentiation with respect to the summed variables. This property was used in [20] to suggest a general measurement for the strength of cryptographic functions of low algebraic degree. Similar summation methods, along with more concrete cryptanalytic ideas, were later used to attack several stream ciphers. For example, Englund, Johansson, and Turan [9] presented a framework to detect non-randomness in stream ciphers, and in [23] Vielhaber developed a key-recovery attack (called AIDA) on reduced versions of Trivium [6]—another cipher in the eSTREAM portfolio. More recently, generalizations of these attacks were proposed: Cube attacks generalize AIDA as a key-recovery attack on a large variety of cryptographic

*Supported by the Swiss National Science Foundation, project no. 113329.
†Supported by GEBERT RÜF STIFTUNG, project no. GRS-069/07.
[4] See `http://www.ecrypt.eu.org/stream`.

schemes. Cube testers use similar techniques to those used in [9], but are more general. Cube testers were previously applied to Trivium [1], and seem relevant to attack Grain-128, since it also builds on low-degree and sparse algebraic equations.

This paper presents an FPGA implementation of cube testers on Grain-128. We ran 256 instances of Grain-128 in parallel, each instance being itself parallelized by a factor 32. Our heaviest experiment involved the computation of 2^{54} clockings of the Grain-128 mechanism, and detected nonrandomness on up to 237 rounds (out of 256). As an aside, we describe some of the other tools we used: a bitsliced C implementation of cube testers on Grain-128, and a simple evolutionary algorithm for searching "good cubes".

Compared to previous works, our attacks are more robust and general. For example, [5] exploits a sliding property that can easily be avoided, as [5, §3.4] explains: "to eliminate the self-similarity of the initialization constant. If the last 16 bits of the LFSR would for example have been initialized with (0,...,0.1), then this would already have significantly reduced the probability of the sliding property."

2 Brief Description of Grain-128

The mechanism of Grain-128 consists of a 128-bit LFSR, a 128-bit NFSR (both over GF(2)), and a Boolean function h. The feedback polynomial of the NFSR has algebraic degree two, and h has degree three (see Fig. 1).

Given a 128-bit key and a 96-bit IV, one initializes Grain-128 by filling the NFSR with the key, and the LFSR with the IV padded with 1 bits. The mechanism is then clocked 256 times without producing output, and feeding the output of h back into both registers. Details can be found in [16].

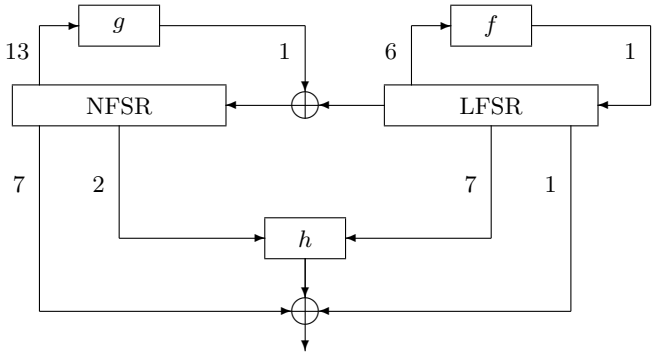

Fig. 1. Schematic view of Grain-128's keystream generation mechanism (numbers designate arities). During initialization, the output bit is fed back into both registers, i.e., added to the output of f and g.

3 Cube Testers

In this section, we briefly explain the principles behind cube testers, and describe the type of cube testers used for attacking Grain-128. More details can be found in [1], and in the article introducing (key-recovery) cube attacks [8].

An important observation regarding cube testers is that for any function $f : \{0,1\}^n \mapsto \{0,1\}$, the sum (XOR) of all entries in the truth table

$$\sum_{x \in \{0,1\}^n} f(x)$$

equals the coefficient of the highest degree monomial $x_1 \cdots x_n$ in the algebraic normal form (ANF) of f. This observation has been used by Englund, Johansson, and Turan [9] for building distinguishers.

For a stream cipher, one may consider as f the function mapping the key and the IV bits to the first bit of keystream. Obviously, evaluating f for each possible key/IV and xoring the values obtained yields the coefficient of the highest degree monomial in the implicit algebraic description of the cipher.

Instead, cube attacks work by summing $f(x)$ over a *subset* of its inputs. For example, if $n = 4$ and

$$f(x) = f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3 + x_1x_2x_4 + x_3 \ ,$$

then summing over the four possible values of (x_1, x_2) yields

$$\sum_{(x_1,x_2)\in\{0,1\}^2} f(x_1, x_2, x_3, x_4) = 4x_1 + 4x_3 + (x_3 + x_4) \equiv x_3 + x_4 \ ,$$

where $(x_3 + x_4)$ is the factor of x_1x_2 in f:

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2(x_3 + x_4) + x_3 \ .$$

Indeed, when x_3 and x_4 are fixed, then the maximum degree monomial becomes x_1x_2 and its coefficient equals the value $(x_3 + x_4)$. In the terminology of cube attacks, the polynomial $(x_3 + x_4)$ is called the *superpoly* of the *cube* x_1x_2. Cube attacks work by detecting linear superpolys, and then explicitly reconstructing them via probabilistic linearity tests [3].

Now, assume that we have a function $f(k_0, \ldots, k_{127}, v_0, \ldots, v_{95})$ that, given a key k and an IV v, returns the first keystream bit produced by Grain-128. For a fixed key $k_0, \ldots, k_{127}$, the sum

$$\sum_{(v_0,\ldots,v_{95})\in\{0,1\}^{96}} f(k_0, \ldots, k_{127}, v_0, v_{95})$$

gives the evaluation of the superpoly of the cube $v_0v_1 \cdots v_{95}$. More generally, one can fix some IV bits, and evaluate the superpoly of the cube formed by the other IV bits (then called the *cube variables*, or CV). Ideally, for a random key, this superpoly should be a uniformly distributed random polynomial. However, when the cipher is constructed with components of low degree, and sparse algebraically, this polynomial is likely to have some property which is efficiently detectable. More details about cube attacks and cube testers can be found in [1, 8].

In our tests below, we measure the *balance* of the superpoly, over 64 instances with distinct random keys.

4 Software Implementation

Since we need to run many independent instances of Grain-128 that operate on bits (rather than bytes or words), a *bitsliced* implementation in software is a natural choice. This technique was originally presented by Biham [2], and can speed up the preprocessing phase of cube attacks (and cube testers) as suggested by Crowley in [7].

To test small cubes, and to perform the search described in §6, we used a bitsliced implementation of Grain-128 that runs 64 instances of Grain-128 in parallel, each with (potentially) different keys and different IV's. We stored the internal states of the 64 instances in two arrays of 128 words of 64 bits, where each bit slice corresponds to an instance of Grain-128, and the i-th word of each array contains the i-th bit in the LFSR (or NFSR) of each instance.

Our bitsliced implementation provides a considerable speedup, compared to the reference implementation of Grain-128. For example, on a PC with an Intel Core 2 Duo processor, evaluating the superpoly of a cube of dimension 30 for 64 distinct instances of Grain-128 with a bitsliced implementation takes approximately 45 minutes, against more than a day with the designers' C implementation. Appendix A gives our C code.

5 Hardware Implementation

Field-programmable gate arrays (FPGA's) are reconfigurable hardware devices widely used in the implementation of cryptographic systems for high-speed or area-constrained applications. The possibility to reprogram the designed core makes FPGA's an attractive evaluation platform to test the hardware performances of selected algorithms. During the eSTREAM competition, many of the candidate stream ciphers were implemented and evaluated, on various FPGA's [4, 11, 13]. Especially for Profile 1 (HW), the FPGA performance in terms of speed, area, and flexibility was a crucial criterion to identify the most efficient candidates.

To attack Grain-128, we used a Xilinx Virtex-5 LX330 FPGA to run the first reported implementation of cube testers in hardware. This FPGA offers a large number of embedded programmable logic blocks, memories and clock managers, and is an excellent platform for large scale parallel computations.

Note that FPGA's have already been used for cryptanalytic purposes, most remarkably with COPACOBANA [15,19], a machine with 120 FPGA's that can be programmed for exhaustive search of small keys, or for parallel computation of discrete logarithms.

5.1 Implementation of Grain-128

The Grain ciphers (Grain-128 and Grain-v1) are particularly suitable for resource-limited hardware environments. Low-area implementations of Grain-v1 are indeed able to fill just a few slices in various types of FPGA's [14]. Using only shift registers combined with XOR and AND gates, the simplicity of the Grain's construction could also be easily translated into high-speed architectures. Throughput and circuit's efficiency (area/speed ratio) are indeed the two main characteristics that have been used as guidelines to design our Grain-128 module for the Virtex-5 chip. The relatively small degree of optimization for Grain allows the choice of different datapath widths, resulting in the possibility of a speedup by a factor 32 (see [16]).

We selected a 32-bit datapath to get the fastest and most efficient design in terms of area and speed. Fig. 2 depicts our module, where both sides of the diagram contain four 32-bit register blocks. During the setup cycle, the key and the IV are stored inside these memory blocks. In normal functioning, they behave like shift register units, i.e., at each clock cycle the 32-bit vectors stored in the lower blocks are sent to the upper blocks. For the two lowest register blocks (indices between 96 and 127), the input vectors are generated by specific functions, according to the algorithm definition. The g' module executes the same computations of the function g plus the addition of the smallest index coming from the LFSR, while the output bits are entirely computed inside the h' module. Table 1 summarizes the overall structure of our 32×Grain-128 architecture.

Table 1. Performance results of our Grain-128 implementation.

	Frequency [MHz]	Throughput [Mbps]	Size [Slices]	Available area [Slices]
Grain-128 module	200	6,400	180	51,840

5.2 Implementation of Cube Testers

Besides the intrinsic speed improvement from software to hardware implementations of Grain-128, the main benefit resides in the possibility to parallelize the computations of the IV queries necessary for the cube tester. With 2^m instances of Grain-128 in parallel, running a cube tester with a $(n+m)$-dimensional cube will be as fast as with an n-dimensional cube on a single instance.

In addition to the array of Grain-128 modules, we designed three other components: the first provides the pseudorandom key and the 2^n IV's for each instance, the second collects and sums the outputs, and the last component is a controller unit. Fig. 3 illustrates the architecture of our cube tester implementation fitted in

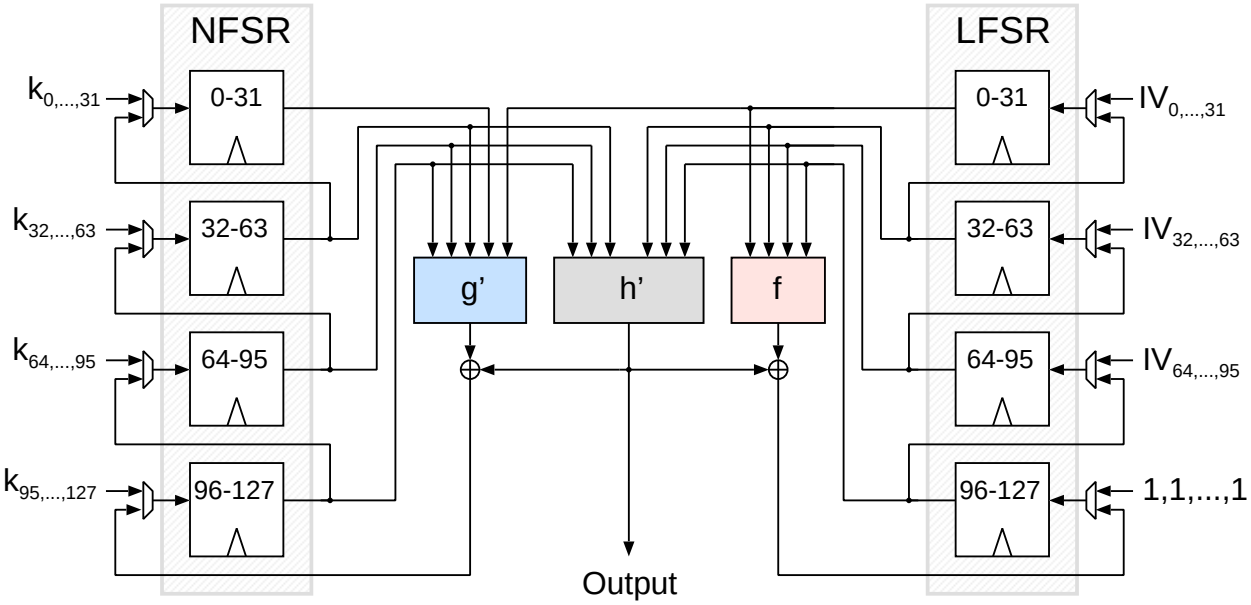

Fig. 2. Overview of our Grain-128 architecture. At the beginning of the simulation, the key and the IV are directly stored in the NFSR and LFSR register blocks. All connections are 32-bit wide.

a Virtex-5 chip. No special macro blocks has been used, we just tried to exploit all the available space to fit the largest Grain-128 array. Below we describe the mode of operation of each component:

- **Simulation controller**: This unit manages the IO interface to control the cube tester core. Through the signal `s_inst`, a new instance is started. After the complete evaluation of the cube over the Grain-128 array, the `u_inst` signal is asserted and later a new instantiation with a different key is started. This operation mode works differently from the software implementation, where the 256 instances run in parallel.
- **Input generator**: After each run of the cipher array, the $(n\text{-}m)$-bit partial IV is incremented by one. This vector is then combined with different m-bit offset vectors to generate the 2^m IVs. The key distribution is also managed here. A single key is given to the parallel Grain-128 modules and is updated only when the partial IV is equal to zero.
- **Output collector**: The outcoming 32-bit vectors from the parallel Grain-128 modules are xored, and the result is xored again with the intermediate results of the previous runs. The updated intermediate results are then stored until the `u_inst` signal is asserted. This causes a reset of the 32-bit intermediate vector and an update of an internal counter.

The m-bit binary representations of the numbers in $0, \ldots, 2^m - 1$ are stored in offset vectors. These vectors are given to the Grain-128 modules as the last cube bits inside the IV. The correct allocation of the CV bits inside the IV is performed by the CV routers. These blocks take the partial IV and the offset vectors to form a 96-bit IV, where the remaining bits are set to zero. When the cube is updated, the offset bits are reallocated, varying the composition of the IV's.

In the input generator, the key is also provided by a LSFR with (primitive) feedback polynomial $x^{128} + x^{29} + x^{27} + x^2 + 1$. This guarantees a period of $2^{128} - 1$, thus ensuring that no key is repeated.

The evaluation of the superpoly for all 256 instances with different pseudorandom keys is performed inside the output collection module. After the 2^{n-m} queries, the intermediate vector contains the final evaluation of the superpoly for a single instance. The implementation of a modified Grain-128 architecture with $\times 32$ speedup allows us to evaluate the same cube for 32 subsequent rounds. That is, after the exhaustive simulation of all possible values of the superpoly, we get the results for the same simulation done with an increasing

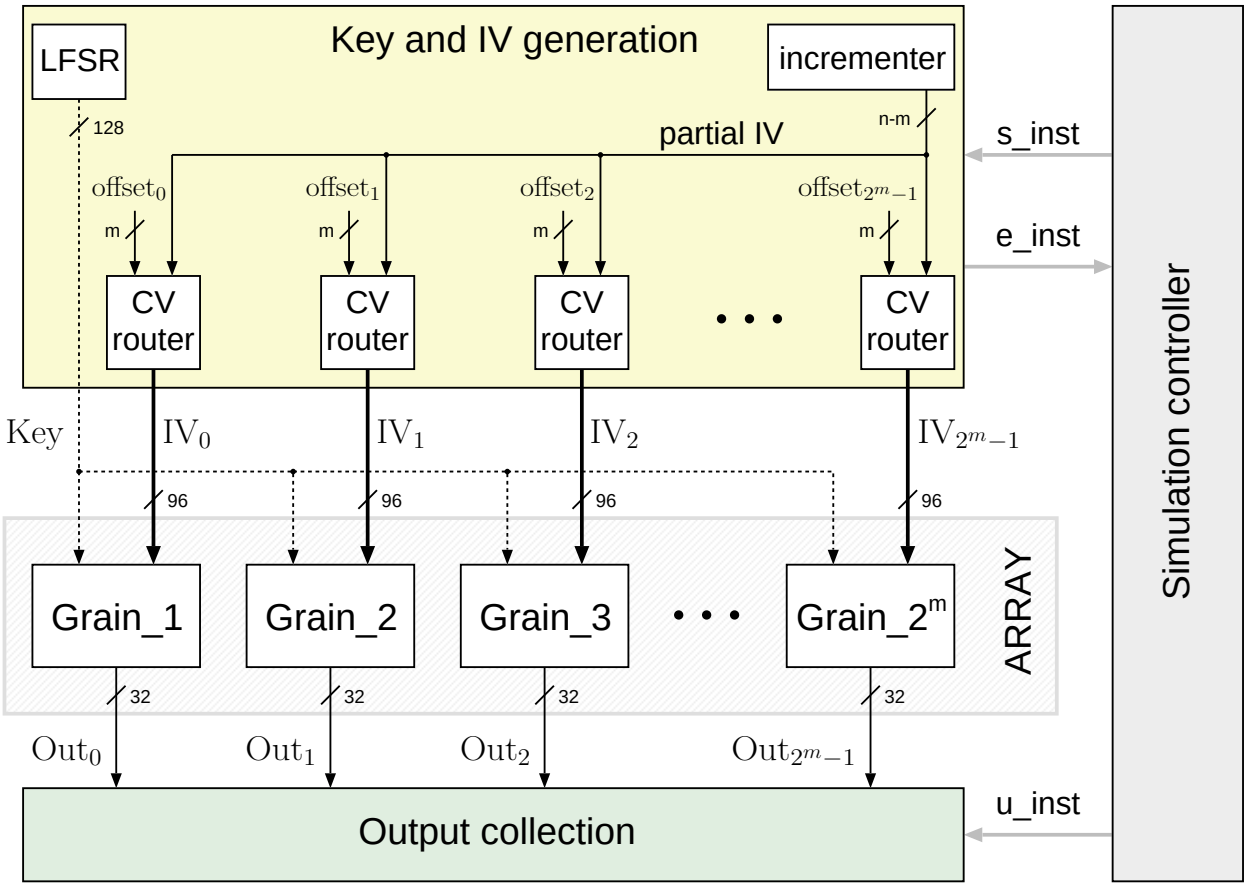

Fig. 3. Architecture of the FPGA cube module. The width of all signals is written out, except for the control signals in grey.

number of initialization rounds r, $32i \leq r < 32(i+1)$ and $i \in [1, 7]$. This is particularly useful to test the maximal number of rounds attackable with a specific cube (we don't have to run the same initialization rounds 32 times to test 32 distinct round numbers).

Finally, 32 dedicated counters are incremented if the values of the according bit inside the intermediate result vector is zero or one, respectively. At the end of the repetitions, the counters indicate the proportion between zeros and ones for 32 different values of increasing rounds. This proportion vector can be constantly monitored using an IO logic analyzer.

Since the required size of a single Grain-128 core is 180 slices, up to 256 parallel ciphers can be implemented inside a Virtex-5 LX330 chip (cf. Table 1). This gives $m = 8$, hence decreasing the number of queries to 2^{n-8}. Table 2 presents the evaluation time for cubes up to dimension 50. The critical path has been kept inside the Grain-128 modules, so the working frequency of the cube machine is 200 MHz.

Estimate for an ASIC Implementation The utilization of an application-specific integrated circuit (ASIC) is a further solution to enhance the performances of cube testers on Grain-128. Like in the FPGA, several parallel cipher modules should run at the same time, decreasing the evaluation period of a cube. Using the ASIC results presented in [11, 14], we could estimate a speed increase up to 400 MHz for a 90 nm CMOS technology. Evaluating a related area cost of about 10 kGE for a single Grain-128 module (broad estimate), we could take into account a single chip design of 4 mm×4 mm size, hosting the same number of Grain-128 elements of 256. This leads to a similar ASIC cube tester implementation, which is able to compute a cube

Table 2. FPGA evaluation time for cubes of different dimension with $2^m = 2^8$ parallel Grain-128 modules. Note that detecting nonrandomness requires the calculation of statistics on several trials, e.g., our experiments involved 64 trials with a 40-bit cube.

Cube dimension	30	35	37	40	44	46	50
Nb. of queries	2^{22}	2^{27}	2^{29}	2^{32}	2^{36}	2^{38}	2^{42}
Time	0.17 sec	5.4 sec	21 sec	3 min	45 min	3 h	2 days

in half the time of the FPGA. However, in this rough estimate we omitted several problematics related to ASIC design, like the expensive fabrication costs or the development of an interface to communicate the cube indices inside the chip.

6 Search for Good Cubes

To search for cubes that maximize the number of rounds after which the superpoly is still not balanced, we programmed a simple *evolutionary algorithm* (EA). Metaheuristic optimization methods like EA's seem relevant for searching good cubes, since they are generic, highly parametrizable, and are often the best choice when the topology of the search space is unknown. In short, EA's aim to maximize a *fitness function*, by updating a set of points in the search space according to some evolutionary operators, the goal being to converge towards a (local) optimum in the search space.

We implemented in C a simple EA that adapts the evolutionary notions of selection, reproduction, and mutation to cubes, which are then seen as individuals of a population. Our EA returns a set of cubes, and is parametrized by

- σ, the cube dimension, in bits.
- μ, the maximal number of mutations.
- π, the (constant) population size.
- χ, the number of individuals in the offspring.
- γ, the number of generations.

Algorithm 1 gives the pseudocode of our EA, where lines 3 to 5 correspond to the *reproduction*, lines 6 and 7 correspond to the *mutation*, while lines 8 and 9 correspond to the *selection*.

Algorithm 1 uses as fitness function a procedure that returns the highest number of rounds for which it yields a *constant* superpoly. We chose to evaluate the constantness rather than the balance because it reduces the number of parameters, thus simplifying the configuration of the search.

Algorithm 1 Evolutionary algorithm for searching good cubes.

1. initialize a population of π random σ-bit cubes
2. **repeat** γ times
3. **repeat** χ times
4. pick two random cubes $\square_1$ and $\square_2$ in the population of π cubes
5. create a new cube with each index chosen randomly from $\square_1$ or $\square_2$
6. choose a random number i in $\{1, \ldots, \mu\}$
7. choose i random indices in the new cube, replace them by random indices
8. evaluate the fitness of the population and of the offspring
9. replace population by the π best-ranking individuals
10. **return** the π cubes in the population

In practice, we optimized Algorithm 1 with ad hoc tweaks, like initializing cubes with particular "weak" indices, e.g., 33, 66, and 68; we indeed observed that these indices appeared frequently in the cubes found by

a vanilla version of our EA, which suggests that the distribution of monomials containing the corresponding bits tends to be lesser than that of random monomials. We later initialized the population by forcing the use of alleged weak indices in certain individuals, and experimental results did not contradict our conjecture.

Note that EA's can be significantly more complex, notably by using more complicated selection and mutation rules (see [12] for an overview of the topic).

The choice of parameters depends on the cube dimension considered. In our algorithm, the quality of the final result is determined by the population size, the offspring size, the number of generations, and the type of mutation. In particular, increasing the number of mutations favors the exploration of the search space, but too much mutation slows down the convergence to a local optimum. The population size, offspring size, and number of generations are always better when higher, but too large values make the search too slow.

For example, we could find our best 6-dimensional cubes ($\sigma = 6$) by setting $\mu = 3$, $\pi = 40$, $\chi = 80$, and $\gamma = 100$. The search then takes a few minutes. Slower searches did not give significantly better results.

7 Experimental Results

Table 3 summarizes the maximum number of initialization rounds after which we could detect imbalance in the superpoly corresponding to the first keystream bit. It follows that one can mount a distinguisher for 195-round Grain-128 in time 2^{10}, and for 237-round Grain-128 in time 2^{40}. The cubes used are given in Appendix B.

Table 3. Best results for various cube dimensions on Grain-128.

Cube dimension	6	10	14	18	22	26	30	37	40
Rounds	180	195	203	208	215	222	227	233	237

8 Discussion

8.1 Extrapolation

We used standard methods to extrapolate our results, using the generalized linear model fitting of the Matlab tool. We selected the Poisson regression in the "log" value, i.e. logarithm as canonical function and the Poisson distribution, since the achieved results suggested a logarithmic behavior between cube size and number of round. The obtained extrapolation, depicted on Fig. 4, suggests that cubes of dimension 77 may be sufficient to construct successful cube testers on the full Grain-128, i.e., with 256 initialization rounds.

If this extrapolation is correct, then a cube tester making $64 \times 2^{77} = 2^{83}$ chosen-IV queries can distinguish the full Grain-128 from an ideal stream cipher, against 2^{128} ideally. We add the factor 64 because our extrapolation is done with respect to results obtained with statistic over 64 random keys. That complexity excludes the precomputation required for finding a good cube; based on our experiments with 40-dimensional cubes, less than 2^5 trials would be sufficient to find a good cube (based on the finding of good small cubes, e.g., using our evolutionary algorithm). That is, precomputation would be less than 2^{88} initializations of Grain-128.

8.2 The Possibility of Key-Recovery Attacks

To apply key-recovery cube attacks on Grain-128, one must find IV terms with a linear superpoly in the key bits (or maxterms). In general, it is more difficult to find maxterms than terms with a biased superpoly, since one searches for a very specific structure in the superpoly. Moreover, the internal structure of Grain-128 seems to make the search for maxterms particularly difficult for reduced variants of the cipher: Initially, the

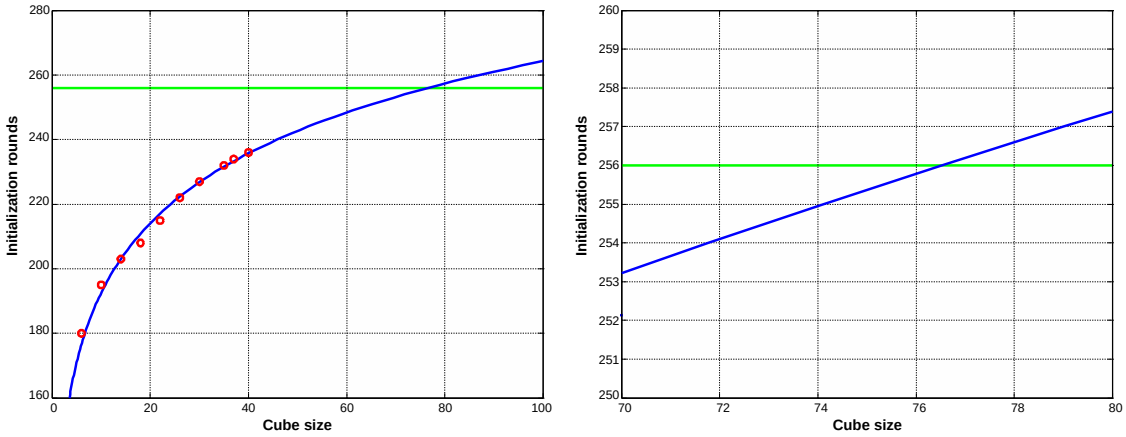

Fig. 4. Extrapolation of our cube testers on Grain-128, obtained by general linear regression using the Matlab software, in the "poisson-log" model. The required dimension for the full Grain-128 version is 77 (see zoom on the right).

key and IV are placed in different registers, and the key bits mix together extensively and non-linearly before mixing with the IV bits. Thus, the output bit polynomials of Grain-128 in the key and IV variables contain very few IV terms whose superpoly is linear in the key bits. A natural way to deal with these polynomials is to apply linearization by replacing non-linear products of key bits with new variables. The linearization techniques are more complicated than the basic cube attack techniques and thus we leave key-recovery attacks on Grain-128 as future work.

8.3 Observations on Grain-v1

Grain-v1 is the predecessor of Grain-128. Its structure is similar to that of Grain-128, but the registers are 80-bit instead of 128-bit, the keys are 80-bit, the IV's are 64-bit, and the initialization clocks the mechanism 160 times (see Appendix C).

The feedback polynomial of Grain-v1's NFSR has degree six, instead of two for Grain-128, and is also less sparse. The filter function h has degree three for both versions of Grain, but that of Grain-v1 is denser than that of Grain-128. These observations suggest that Grain-v1 may have a better resistance than Grain-128 to cube testers, because its algebraic degree and density are likely to converge much faster towards ideal ones.

To support the above hypothesis, we used a bitsliced implementation of Grain-v1 to search for good cubes with the EA presented in §6, and we ran cube testers (still in software) similar to those on Grain-128. Table 4 summarizes our results, showing that one can mount a distinguisher on Grain-v1 with 81 rounds of initialization in 2^{24}. However, even an optimistic (for the attacker) extrapolation of these observations suggests that the full version of Grain-v1 resists cube testers, and the basic cube attack techniques.

Table 4. Best results for various cube dimensions on Grain-v1.

Cube dimension	6	10	14	20	24
Rounds	64	70	73	79	81

9 Conclusion

We developed and implemented a hardware cryptanalytical device for attacking the stream cipher Grain-128 with cube testers (which give distinguishers rather than key recovery). We were able to run our tests on 256 instances of Grain-128 in parallel, each instance being itself parallelized by a factor 32. The heaviest experiment run involved about 2^{54} clockings of the Grain-128 mechanism.

To find good parameters for our experiments in hardware, we first ran light experiments in software with a dedicated bitsliced implementation of Grain-128, using a simple evolutionary algorithm. We were then able to attack reduced versions of Grain with up to 237 rounds. An extrapolation of our results suggests that the full Grain-128 can be attacked in time 2^{83} instead of 2^{128} ideally. Therefore, Grain-128 may not provide full protection when 128-bit security is required.

References

1. Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and Trivium. In Orr Dunkelman, editor, *FSE*, LNCS. Springer, 2009. to appear.
2. Eli Biham. A fast new des implementation in software. In Eli Biham, editor, *FSE*, volume 1267 of *LNCS*, pages 260–272. Springer, 1997.
3. Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. In *STOC*, pages 73–83. ACM, 1990.
4. Philippe Bulens, Kassem Kalach, Francois-Xavier Standaert, and Jean-Jacques Quisquater. FPGA implementations of eSTREAM phase-2 focus candidates with hardware profile. Technical Report 2007/024, ECRYPT eSTREAM, 2007.
5. Christophe De Cannière, Özgül Kücük, and Bart Preneel. Analysis of Grain's initialization algorithm. In *SASC 2008*, 2008.
6. Christophe De Cannière and Bart Preneel. Trivium. In *New Stream Cipher Designs*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008.
7. Paul Crowley. Trivium, SSE2, CorePy, and the "cube attack", 2008. Published on `http://www.lshift.net/blog/`.
8. Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *LNCS*, pages 278–299. Springer, 2009.
9. Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT*, volume 4859 of *LNCS*, pages 268–281. Springer, 2007.
10. Simon Fischer, Shahram Khazaei, and Willi Meier. Chosen IV statistical analysis for key recovery attacks on stream ciphers. In Serge Vaudenay, editor, *AFRICACRYPT*, volume 5023 of *LNCS*, pages 236–245. Springer, 2008.
11. Kris Gaj, Gabriel Southern, and Ramakrishna Bachimanchi. Comparison of hardware performance of selected phase II eSTREAM candidates. Technical Report 2007/026, ECRYPT eSTREAM, 2007.
12. David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.
13. Tim Good and Mohammed Benaissa. Hardware performance of eSTREAM phase-III stream cipher candidates. In *SASC*, 2008.
14. Tim Good, William Chelton, and Mohamed Benaissa. Review of stream cipher candidates from a low resource hardware perspective. Technical Report 2006/016, ECRYPT eSTREAM, 2006.
15. Tim Gueneysu, Timo Kasper, Martin Novotny, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008.
16. Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. A stream cipher proposal: Grain-128. In *IEEE International Symposium on Information Theory (ISIT 2006)*, 2006.
17. Martin Hell, Thomas Johansson, and Willi Meier. Grain - a stream cipher for constrained environments. Technical Report 2005/010, ECRYPT eSTREAM, 2005.
18. Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *IJWMC*, 2(1):86–93, 2007.
19. Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA - a cost-optimized parallel code breaker. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *LNCS*, pages 101–118. Springer, 2006.

20. Xuejia Lai. Higher order derivatives and differential cryptanalysis. In *Symposium on Communication, Coding and Cryptography, in honor of James L. Massey on the occasion of his 60'th birthday*, pages 227–233, 1994.
21. Yuseop Lee, Kitae Jeong, Jaechul Sung, and Seokhie Hong. Related-key chosen IV attacks on Grain-v1 and Grain-128. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *ACISP*, volume 5107 of *LNCS*, pages 321–335. Springer, 2008.
22. Sean O'Neil. Algebraic structure defectoscopy. Cryptology ePrint Archive, Report 2007/378, 2007.
23. Michael Vielhaber. Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack. Cryptology ePrint Archive, Report 2007/413, 2007.

A Bitsliced Implementation of Grain-128

We present the C code of a function that, given 64 keys and 64 IV's (already bitsliced), returns the first keystream bit produced by Grain-128 with `rounds` initialization rounds.

```
typedef unsigned long long u64;

u64 grain128_bitsliced64( u64 * key, u64 * iv, int rounds ) {

   u64 l[128+rounds], n[128+rounds], z=0;
   int i,j;

   for(i=0; i<96; i++){
     n[i]= key[i];
     l[i]= iv[i];
   }
   for(i=96; i<128; i++){
     n[i]= key[i];
     l[i]= 0xFFFFFFFFFFFFFFFFULL;
   }
   for(i=0; i<rounds; i++){
     l[i+128] = l[i] ^ l[i+7] ^ l[i+38] ^ l[i+70] ^ l[i+81] ^ l[i+96];
     n[i+128] = l[i] ^ n[i] ^ n[i+26] ^ n[i+56] ^ n[i+91] ^ n[i+96] ^
     (n[i+ 3] & n[i+67]) ^ (n[i+11] & n[i+13]) ^ (n[i+17] & n[i+18]) ^
     (n[i+27] & n[i+59]) ^ (n[i+40] & n[i+48]) ^ (n[i+61] & n[i+65]) ^
     (n[i+68] & n[i+84]);

     z = (n[i+12] & l[i+8]) ^ (l[i+13] & l[i+20]) ^
     (n[i+95] & l[i+42]) ^ (l[i+60] & l[i+79]) ^
     (n[i+12] & n[i+95] & l[i+95]);
     z = n[i + 2] ^ n[i + 15] ^ n[i + 36] ^ n[i + 45] ^ n[i + 64] ^
     n[i + 73] ^ n[i + 89] ^ z ^ l[i + 93];

     l[i+128] ^= z;
     n[i+128] ^= z;
   }

   z = (n[i+12] & l[i+8]) ^ (l[i+13] & l[i+20]) ^
   (n[i+95] & l[i+42]) ^ (l[i+60] & l[i+79]) ^
   (n[i+12] & n[i+95] & l[i+95]);
   z = n[i + 2] ^ n[i + 15] ^ n[i + 36] ^ n[i + 45] ^ n[i + 64] ^
   n[i + 73] ^ n[i + 89] ^ z ^ l[i + 93];

   return z;
}
```

B Cubes for Grain-128

Table 5 gives the indices of the cubes used for finding the results in Table 3.

Table 5. Cubes used for Grain-128.

Cube dimension	Indices
6	33, 36, 61, 64, 67, 69
10	5, 28, 34, 36, 37, 66, 68, 71, 74, 79
14	5, 28, 34, 36, 37, 51, 53, 54, 56, 63, 66, 68, 71, 74
18	5, 28, 30, 32, 34, 36, 37, 62, 63, 64, 65, 66, 67, 68, 69, 71, 73, 74
22	4, 5, 28, 30, 32, 34, 36, 37, 51, 62, 63, 64, 65, 66, 67, 68, 69, 71, 73, 74, 79, 89
26	4, 7, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68
30	4, 7, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 59, 62, 65, 66, 69, 72, 75, 78, 79, 80, 83, 86
37	4, 7, 12, 14, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68, 74, 75, 76, 77, 78, 79, 89, 90, 91
40	4, 7, 12, 14, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68, 74, 75, 76, 77, 78, 79, 86, 87, 88, 89, 90, 91

C Grain-v1

Fig. 5 presents the structure of Grain-v1.

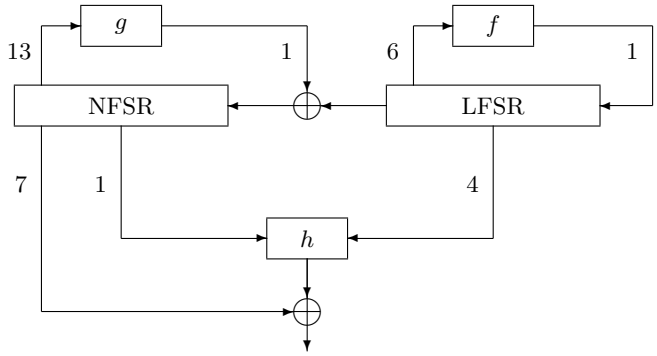

Fig. 5. Schematic view of Grain-v1's keystream generation mechanism (numbers designate arities). During initialization, the output bit is fed back into both registers, i.e., added to the output of f and g.