Experimentally Verifying a Complex Algebraic Attack on the Grain-128 Cipher Using Dedicated Reconfigurable Hardware

Itai Dinur[1], Tim Güneysu[2], Christof Paar[2],
Adi Shamir[1], and Ralf Zimmermann[2]

[1] Computer Science department, The Weizmann Institute, Rehovot, Israel
[2] Horst Görtz Institute for IT Security, Ruhr-University Bochum, Germany

Abstract. In this work, we describe the first single-key attack on the full version of Grain-128 that can recover arbitrary keys. Our attack is based on a new version of a cube tester, which is a factor of about 2^{38} faster than exhaustive search. To practically verify our results, we implemented the attack on the reconfigurable hardware cluster RIVYERA and tested the main components of the attack for dozens of random keys. Our experiments successfully demonstrated the correctness and expected complexity of the attack by finding a very significant bias in our new cube tester for about 7.5% of the tested keys. This is the first time that the main components of a complex analytical attack against a digital full-size cipher were successfully implemented using special-purpose hardware, truly exploiting the reconfigurable nature of an FPGA-based cryptanalytical device.

Keywords: Special-purpose hardware, Grain-128, stream cipher, cryptanalysis, cube attacks, cube testers.

1 Introduction

Special-purpose hardware, i. e., computing machines dedicated to cryptanalytical problems, have a long tradition in code-breaking, including attacks against the Enigma cipher during WWII [3]. Their use is promising if two conditions are fulfilled. First, the complexity of the cryptanalytical problem must be - as of today - in the range of approximately $2^{50} \ldots 2^{64}$ operations. For problems with a lower complexity, conventional computer clusters are typically sufficient, such as the linear cryptanalysis attack against DES [4] (which required 2^{43} DES evaluations), and more than 2^{64} operations are difficult to achieve with today's technology unless extremely large budgets are available. The second condition is that the computations involved are suited for customized hardware architectures, which is often the case in symmetric cryptanalysis. Both conditions are fulfilled for the building blocks of the Grain-128 attack described in this paper.

Grain-128 [1] is a 128-bit variant of the Grain scheme which was selected by the eSTREAM project in 2008 as one of the three recommended hardware-efficient stream ciphers. The only single-key attacks published so far on this scheme which were substantially faster than exhaustive search were either on a reduced number of rounds or on a specific class of weak keys which contains about one in a thousand keys.

Contribution of this work: In this paper, we describe the first attack which can be applied to the full scheme of Grain-128 with arbitrary keys. It uses an improved cube distinguisher with new dynamic variables, which makes it possible to attack Grain-128 with no restriction on the key. Its main components were experimentally verified by running a 50-dimensional cube tester for 107 random keys and discovering a very strong bias (of 50 zeroes out of 51 bits) in about 7.5% of these keys. For these keys, we expect the running time of our new attack to be about 2^{38} times faster than exhaustive search, using 2^{63} bits of memory. Our attack is thus both faster and more general than the best previous attack on Grain-128 [2], which was a weak-key attack on one in a thousand keys which was only 2^{15} times faster than exhaustive search. However, our attack does not seem to threaten the security of the original 80-bit Grain scheme.

In order to develop and experimentally verify the main components of the attack, we had to run thousands of summations over cubes of dimension 49 and 50 for dozens of randomly chosen keys, where each summation required the evaluation of 2^{49} or 2^{50} output bits of Grain-128 (running the time-consuming initialization phase of Grain-128 for about 2^{56} different key and IV values). This process is hardware-oriented, highly

parallelizable, and well beyond the capabilities of a standard cluster of PC's. We thus decided to implement the attack on a special-purpose hardware cluster.

Even though it is widely speculated that government organizations have been using special-purpose hardware for a long time, there are only few confirmed reports about cryptanalytical machines in the open literature. In 1998, Deep Crack, an ASIC-based machine dedicated to brute-forcing DES, was introduced [5]. In 2006, COPACOBANA also allowed exhaustive key searches of DES, and in addition cryptanalysis of other ciphers [6]. However, in the latter case often only very small-scale versions of the ciphers are vulnerable. The paper at hand extends the previous work with respect to cryptanalysis with dedicated hardware in several ways. Our work is the first time that the main components of a complex analytical attack, i. e., not merely an exhaustive search, are successfully realized in a public way against a full-size cipher by using a special-purpose machine (previous attacks were either a simple exhaustive search sped up by a special-purpose hardware, or advanced attacks such as linear cryptanalysis which were realized in software on multiple workstations). Also, this is the first attack which makes use of the reconfigurable nature of the hardware. Our RIVYERA computer, consisting of 128 large FPGAs, is the most powerful cryptanalytical machine available outside government agencies (possessing more than four times as many logic resources as the COPACOBANA machine). This makes our attack an interesting case study about what type of cryptanalysis can be done with "university budgets" (as opposed to government budgets). As a final remark, it is worth noting that the same attack implemented on GPU clusters would require an extremely large number of graphic cards, which would not only require a very high budget but would consume considerably more electric energy to perform the same computations.

Outline: In Section 2, we give the necessary background regarding Grain-128, dynamic cube attacks and describe the new attack on Grain-128. Then we present our implementation of the attack in Section 3, before we present our results and conclusions in Sections 4 and 5.

2 Background

In this section we briefly discuss the background required in the remainder of this work.

2.1 Grain-128 Stream Cipher

The state of Grain-128 consists of a 128-bit LFSR and a 128-bit NFSR. The feedback functions of the LFSR and NFSR are respectively defined to be
$s_{i+128} = s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96}$
$b_{i+128} = s_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + b_{i+68}b_{i+84}$
The output function is defined as
$z_i = \sum_{j\in\mathcal{A}} b_{i+j} + h(x) + s_{i+93}$, where $\mathcal{A} = \{2, 15, 36, 45, 64, 73, 89\}$.
$h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8$
where the variables x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 and x_8 correspond to the tap positions b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+60}, s_{i+79} and s_{i+95} respectively.

Grain-128 is initialized with a 128-bit key that is loaded into the NFSR, and with a 96-bit IV that is loaded into the LFSR, while the remaining 32 LFSR bits are filled with 1's. The state is then clocked through 256 initialization rounds without producing an output, feeding the output back into the input of both registers.

2.2 Dynamic Cube Attacks

Cube Testers In almost any cryptographic scheme, each output bit can be described by a multivariate master polynomial $p(x_1, .., x_n, v_1, .., v_m)$ over GF(2) of secret variables x_i (key bits), and public variables v_j (plaintext bits in block ciphers and MACs, IV bits in stream ciphers). The cryptanalyst is allowed to tweak this master polynomial by assigning chosen values to the public variables (which result in multiple derived polynomials).

To simplify our notation, we ignore in the rest of this subsection the distinction between public and private variables. Given a multivariate master polynomial with n variables $p(x_1, .., x_n)$ over GF(2) in algebraic normal form (ANF), and a term t_I containing variables from an index subset I that are multiplied together, the polynomial can be written as the sum of terms which are supersets of I and terms that miss at least one variable from I:

$$p(x_1, .., x_n) \equiv t_I \cdot p_{S(I)} + q(x_1, .., x_n)$$

$p_{S(I)}$ is called the *superpoly* of I in p. Compared to p, the algebraic degree of the superpoly is reduced by at least the number of variables in t_I, and its number of terms is smaller.

Cube testers [7] are related to high order differential attacks [8]. The basic idea behind them is that the symbolic sum over GF(2) of all the derived polynomials obtained from the master polynomial by assigning all the possible 0/1 values to the subset of variables in the term t_I is exactly $p_{S(I)}$ which is the superpoly of t_I in $p(x_1, .., x_n)$. This simplified polynomial is more likely to exhibit non-random properties than the original polynomial P.

Cube testers work by evaluating superpolys of carefully selected terms t_I which are products of public variables, and trying to distinguish them from a random function. One of the natural properties that can be tested is balance: A random function is expected to contain as many zeroes as ones in its truth table. A superpoly that has a strongly unbalanced truth table can thus be used to distinguish the cryptosystem from a random polynomial.

Dynamic Cube Attacks Dynamic Cube Attacks exploit distinguishers obtained from cube testers to recover some secret key bits. In static cube testers (and other related attacks such as the original cube attack [10], and AIDA [9]), the values of all the public variables that are not summed over are fixed to a constant (usually zero), and thus they are called static variables. However, in dynamic cube attacks the values of some of the public variables that are not part of the cube are not fixed. Instead, each one of these variables (called dynamic variables) is assigned a function that depends on some of the cube public variables and on some private variables. Each such function is carefully chosen in order to simplify the resultant superpoly and thus to amplify the expected bias (or the non-randomness in general) of the cube tester.

The basic steps of the attack are briefly summarized below (for more details refer to [2], where the notion of dynamic cube attacks was introduced).

Preprocessing Phase We first choose some polynomials that we want to set to zero at all the vertices of the cube, and show how to nullify them by setting certain dynamic variables to appropriate expressions in terms of the other public and secret variables. To minimize the number of evaluations of the cryptosystem, we choose a big cube of dimension d and a set of subcubes to sum over during the online phase. We usually choose the subcubes of the highest dimension (namely d and $d-1$), which are the most likely to give a biased sum. We then determine a set of e expressions in the private variables that need to be guessed by the attacker in order to calculate the values of the dynamic variables during the cube summations.

Online Phase The online phase of the attack has two steps that are described in the following.

Step 1: The first step consists again of two substeps:

1. For each possible vector of values for the e secret expressions, sum modulo 2 the output bits over the subcubes chosen during preprocessing with the dynamic variables set accordingly, and obtain a list of sums (one bit per subcube).
2. Given the list of sums, calculate its score by measuring the non-randomness in the subcube sums. The output of this step is a sequence of lists sorted from the lowest score to the highest (in our notation the list with the lowest score has the largest bias, and is thus the most likely to be correct in our attack).

Given that the dimension of our big cube is d, the complexity of summing over all its subcubes is bounded by $d2^d$ (using the Moebius transform [11]). Assuming that we have to guess the values of e secret expressions in order to determine the values of the dynamic variables, the complexity of this step is bounded by $d2^{d+e}$ bit operations. Assuming that we have y dynamic variables, both the data and memory complexities are bounded by 2^{d+y} (since it is sufficient to obtain an output bit for every possible vertex of the cube and for every possible value of the dynamic variables).

Step 2: Given the sorted guess score list, we determine the most likely values for the secret expressions, for a subset of the secret expressions, or for the entire key. The specific details of this step vary according to the attack.

Partial Simulation Phase The complexity of executing online step 1 of the attack for a single key is $d2^{d+e}$ bit operations and 2^{d+y} cipher executions. In the case of Grain-128, these complexities are too high and thus we have to experimentally verify our attack with a simpler procedure. Our solution is to calculate the cube summations in online step 1 only for the correct guess of the e secret expressions. We then calculate the score of the correct guess and estimate its expected position g in the sorted list of score values by assuming that incorrect guesses will make the scheme behave as a random function. Consequently, if the cube sums for the correct guess detect a property that is satisfied by a random cipher with probability p, we estimate that the location of the correct guess in the sorted list will be $g \approx \max\{p \times 2^e, 1\}$ (as justified in [2]).

2.3 Dynamic Cube Attacks on Grain-128

The parameter set we use for our attack is described — and its choice justified — in [12], and is given here again in Table 1 for the sake of completeness. However, since we focus on the implementation of the attack, we omit it from this paper. Moreover, we specify only the details of the online phase of the attack that are most relevant to our hardware implementation. For the complete details, refer to [12].

Table 1. Parameter set for the attack on the full Grain-128, given output bit 257.

Cube Indexes	{0,2,4,11,12,13,16,19,21,23,24,27,29,33,35,37,38,41,43,44,46, 47,49,52,53,54,55, 57,58,59,61,63,65,66,67,69,72,75,76,78,79,81,82,84,85,87,89,90,92,93}
Dynamic Variables	{31,3,5,6,8,9,10,15,7,25,42,83,1}
State Bits Nullified	$\{b_{159}, b_{131}, b_{133}, b_{134}, b_{136}, b_{137}, b_{138}, b_{145}, s_{135}, b_{153}, b_{170}, b_{176}, b_{203}\}$

Before executing the online phase of the attack, one should take some preparation steps (given in [12])in order to determine the list of $e = 39$ secret expressions in the key variables we have to guess during the actual attack. Online step 1 of the attack is given below:

1. Obtain the first output bit produced by Grain-128 (after the full 256 initialization steps) with the fixed secret key and all the possible values of the variables of the big cube and the dynamic variables given in Table 1 (the remaining public variables are set to zero). The dimension of the big cube is 50 and we have 13 dynamic variables and thus the total amount of data and memory required is $2^{50+13} = 2^{63}$ bits.
2. We have 2^{39} possible guesses for the secret expressions. Allocate a guess score array of 2^{39} entries (an entry per guess). For each possible value (guess) of the secret expressions:
   (a) Plug the values of these expressions into the dynamic variables (which thus become a function of the cube variables, but not the secret variables).
   (b) Our big cube in Table 1 is of dimension 50. Allocate an array of 2^{50} bit entries. For each possible assignment to the cube variables:
      i. Calculate the values of the dynamic variables and obtain the corresponding output bit of Grain-128 from the data.
      ii. Copy the value of the output bit to the array entry whose index corresponds to the assignment of the cube variables.
   (c) Given the 2^{50}-bit array, sum over all the entry values that correspond to the 51 subcubes of the big cube which are of dimension 49 and 50. When summing over 49-dimensional cubes, keep the cube variable that is not summed over to zero. This step gives a list of 51 bits (subcube sums).
   (d) Given the 51 sums, calculate the score of the guess by measuring the fraction of bits which are equal to 1. Copy the score to the appropriate entry in the guess score array and continue to the next guess (item 2). If no more guesses remain go to the next step.

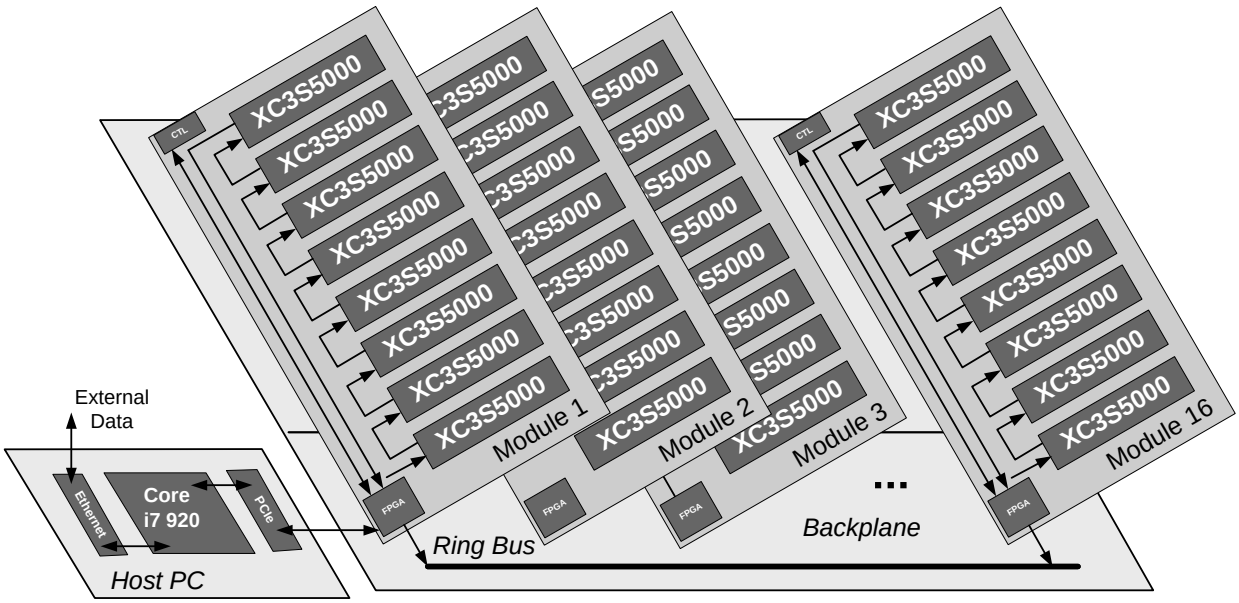

Fig. 1. Architecture of the RIVYERA cluster system

3. Sort the 2^{39} guess scores from the lowest score to the highest.

The total complexity of algorithm above is about $50 \times 2^{50+39} < 2^{95}$ bit operations (it is dominated by item 2.c, which is performed once for each of the 2^{39} possible secret expression guesses).

Given the sorted guess array which is the output of online step 1, we are now ready to perform online step 2 of the attack (which recovers the secret key without going through the difficult step of solving the large system of polynomial equations). The details of online step 2 are given in [12], which also shows that the total complexity the algorithm is equivalent to about $g \times 2^{90}$ cipher evaluations.

The attack is worse than exhaustive search if we have to try all the 2^{39} possible values of g, and thus it is crucial to provide strong experimental evidence that g is relatively small for a large fraction of keys. In order to estimate g, we executed the online part of the attack by calculating the score for the correct guess of the 39 expression values, and estimating how likely it is to get such a bias for incorrect guesses if we assume that they behave as random functions.

The simulation algorithm is a simplified version of item 2 in online step 1 (performed only for the correct key), and is described in Algorithm 2 as shown in the appendix. We performed this simulation for 107 randomly chosen keys, out of which 8 gave a very significant bias in which at least 50 of the 51 cubes sums were zero. This is expected to occur in a random function with probability $p < 2^{-45}$, and thus we estimate that for about 7.5% of the keys, $g \approx \max\{2^{-45} \times 2^{39}, 1\} = 1$ and thus the correct guess of the 39 secret expressions will be the first in the sorted score list (additional keys among those we tested had smaller biases, and thus a larger g). The complexity of online step 2 of the attack is thus expected to be about 2^{90} cipher executions, which dominates the complexity of the attack (the complexity of online step 1 is about 2^{95} bit operations, which we estimate as $2^{95-10} = 2^{85}$ cipher executions). This gives an improvement factor of 2^{38} over the 2^{128} complexity of exhaustive search for a non-negligible fraction of keys, which is significantly better than the improvement factor of 2^{15} announced in [2] for the small subset of weak keys considered in that attack. We note that for most additional keys there is a continuous tradeoff between the fraction of keys that we can attack and the complexity of the attack on these keys.

2.4 RIVYERA Special-Purpose Hardware Cluster

In this work, we employ an enhanced version of the COPACOBANA special-purpose hardware cluster that was specifically designed for the task of cryptanalysis [6]. This enhanced cluster (also known as RIVYERA [13]) is populated with 128 Spartan-3 XC3S5000 FPGAs, each tightly coupled with 32MB memory. Each Spartan-3 XC3S5000 FPGA provides a sea of logic resources consisting of 33,280 slices and 104 BRAMs enabling the implementation even of complex functions in reconfigurable hardware. Eight FPGAs are soldered on individual card modules that are plugged into a backplane which implements a global systolic ring bus for high-performance communication. The internal ring bus is further connected via PCI Express to a host PC which is also installed in the same 19" housing of the cluster. Figure 1 provides an overview of the architecture of the RIVYERA special-purpose cluster.

3 Implementation

To get a better understanding of our implementation and the design decisions, we need to examine the steps of the algebraic attack from Section 2.3. We start by formulating the steps from an implementation point-of-view and discuss the workflow in more detail, briefly discuss the different implementation options and finally describe our implementation on an FPGA cluster.

3.1 Analysis of the Algorithm

Algorithm 1 describes the attack with respect to its implementation in hardware.

Algorithm 1 Dynamic Cube Attack Simulation (Algorithm 2), Optimized for Implementation

Input: 96 bit integer *baseIV*, cube dimension d, cube $C = \{C_0, \ldots, C_d\}$ with $0 \leq C_i < 96\ \forall C_i \in C$, number of polynomials m, dynamic variable indices $D = \{D_0, \ldots, D_m\}$ with $0 \leq C_i < 96\ \forall D_i \in D$, state bit indices $S = \{S_0, \ldots, S_m\}$ with $0 \leq S_i < 96\ \forall S_i \in S$.
Output: $(d+1)$ bit cubesum s

IV $\leftarrow$ *baseIV*
$s \leftarrow 0$.

Key Selection
Choose random 128 bit key K.
Choose key-dependent polynomials $P_j(X)$ nullifying state bits S_j.

Computation
for $i \leftarrow 0$ **to** $2^d - 1$ **do**
    for $j \leftarrow 0$ **to** $d - 1$ **do**
        setBit(IV, C_j, getBit(i, j))
    end for
    for $j \leftarrow 0$ **to** $m - 1$ **do**
        setBit(IV, D_j, $P_j(i)$)
    end for
    $ks \leftarrow$ first bit of Grain-128(IV, K) keystream
    if $ks = 1$ **then**
        $s \leftarrow s \oplus (1|\text{not}(i))$
    end if
end for
return s.

To simplify the description, we use the function `getBit(int, pos)`, returning the bit at position `pos` of the integer `int`, and `setBit(int, pos, val)`, setting the bit at position `pos` of integer `int` to `val`.

All input arguments are included in the parameter set in Table 1. While they are fixed by the Table, we have to keep in mind that the attack is also an experimental verification of this parameter set. Thus it is important that the implementation should allow changes and pose as few restrictions as possible to these values. The algorithm will compute the cube sum of $d + 1$ bits, i. e., of size 51 bits with our parameters.

After selecting the key we wish to attack, we need to choose the polynomials, which nullify certain state bits and reset the IV to a default value. In the loop starting at line 5, we iterate over all 2^d combinations. Each time, we modify the IV by spreading the current iteration value over the cube positions (line 7) and evaluate the polynomials - boolean functions depending on these changing positions - and store the resulting bit per function at the dynamic variable positions (line 10). Now that the IV is prepared, we run a full initialization (256 rounds) of Grain-128 (line 12 and - in case the first keystream bit is not zero - we XOR the current sum with the inverse of the d bit iteration count, prefixed by a 1 (line 14).

Figure 2 describes the basic workflow of an implementation: It uses a parameter set as input, e. g., the cube dimension, the cube itself, a base IV and the number of keys to attack. It selects a random key to attack and divides the big cube into smaller worker cubes and distributes them to worker threads running in parallel. Please note that for simplicity the figure shows only one worker. If 2^w workers are used in parallel, the iterations per worker are reduced from 2^d to 2^{d-w}.

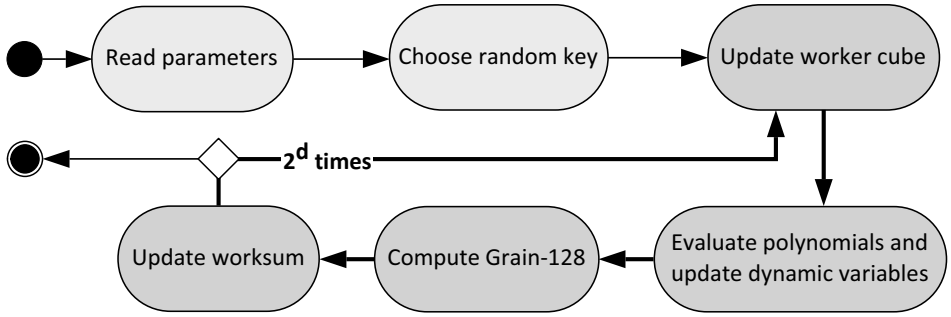

Fig. 2. Cube Attack — Program flow for cube dimension d.

The darker nodes and the bold path show the steps of each independent thread: As each worker iterates over a distinct subset of the cube, it evaluates polynomials on the worker cube (dynamic variables) and updates the IV input to Grain-128. Using this generated IV and the random key, it computes the output of Grain-128 after the initialization phase. With this output, the thread updates an intermediate value — the worker sum — and starts the next iteration. In the end, the software combines all worker sums, evaluates the result and may chose a new random key to start again.

We can see that the algorithm is split into three parts: First, we manipulate the worker cube positions and derive an IV from it. Then, we compute the output of the Grain-128 stream cipher using the given key, data and derived IV. Before we start the next iteration, the worksum is updated.

Grain-128: The second part is straight-forward and seems to be the main computational task. It concerns only the Grain implementation: With a cube of dimension d, the attack on one key computes the first output bit of Grain-128 2^d times. As we already need 2^{50} iterations with the current parameter set, it is necessary to implement Grain-128 as efficiently as possible in order to speed up the attack.

Taking a closer look at the design of the stream cipher (see Section 2.1), it yields much potential for an efficient hardware implementation: It consists mainly of two shift registers and some logic cells. While using bit-slicing techniques potentially decreases the overhead of CPUs when computing expensive bit-manipulations, Aumasson *et al.*already proposed a fast and small FPGA implementation as a good choice when implementing cube testers on Grain-128 in [14].

IV Generation: To create an independent worker, it also needs to include the IV generation. This process takes the default IV and modifies $d + m$ bits, which is easily done in software by storing the generated IV as an array and accessing the positions directly. Changing the parameters to compute larger cube dimensions d, to allow more than m polynomials poses no problem either.

Considering a possible hardware implementation, this increases the complexity a lot. In contrast to the software design, we cannot create a generic layout, which reads the parameter set: We need multiplexers for all IV bits to allow dynamic choices and even more multiplexers to allow all possible combinations of boolean functions to support all possible polynomials.

As this problem seems very easy in software and difficult in hardware, a software approach seems more reasonable. Nevertheless, the communication overhead to supply many workers with new IVs every few clock cycles explodes. To estimate the effort of building an independent worker in hardware, we need to know how many dynamic inputs we have to consider in the IV generation process, as these modifications are very inefficient in hardware: In order to compute the cipher, we need a key and an IV. The value of the key varies, as it is chosen at random.

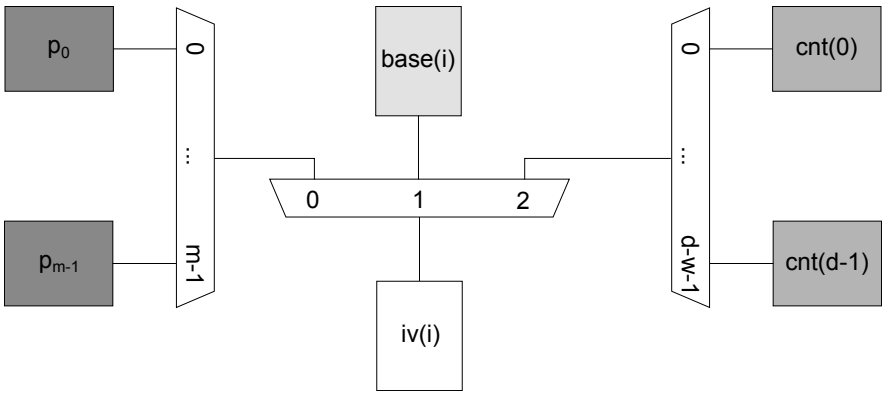

Fig. 3. Necessary Multiplexers for each IV bit (without Optimizations) of a worker with worker cube size $d - w$ and m different polynomials. This is an $(m + d - w + 1)$-to-1 bit multiplexer, i. e., with the current parameter set a $(64 - w)$-to-1 bit multiplexer.

The IV is a 96 bit value, where each bit utilizes one of three functions as Figure 3 shows: it is either a value given by the base IV (light grey) provided by the parameter set, part of the current counter spread across the worker cube (grey) or a dynamic variable (dark grey). As the function of each bit differs not only per parameter set, but also when assigning partial cubes to different workers, this input also varies and we need to create an $(m + d - w + 1)$-to-1 bit multiplexer for each bit, resulting in $96 \times (64 - w)$-to-1 bit multiplexers for our current parameter set.

The first two functionalities are both restricted and can be realized by simple multiplexers in hardware. The dynamic variable on the other hand stores the result of a polynomial. As we have no set of pre-defined polynomials and they are derived at runtime, every possible combination of boolean functions over the worker cubes must be realized. Even with tight restrictions, i. e., a maximum number of terms per monomial and monomials per polynomial, it is impossible to provide such a reconfigurable structure in hardware. As a consequence, a fully dynamic approach leads to extremely large multiplexers and thus to very high area consumption on the FPGA, which is prohibitively slow. Therefore, we need to choose a different strategy to implement this part in hardware.

Worksum update: In order to finish one iteration, the worksum is modified. To simplify the synchronization between the different threads, each worker updates a local intermediate value. In order to generate $d + 1$ bit intermediate values from the $d - w$ bit sums, we precede the number not by a constant 1 but also with

the w bit number of the worker thread. Please note that the actual implementation, we do not use $d+1$ bit XOR operations: If the number of XORs is even, we need to prefix the constant, otherwise, we need to prefix zeroes. Thus, a simple 1 bit value is sufficient to choose between these two values. When all workers are finished, the real result needs to be computed by a XOR operation over all results.

Overall, the complexity of the algebraic attack is too high for a single machine and a cluster of some kind is necessary. As the most cost-intensive operation concerns the 2^d computations of the 256 step Grain initialization, the use of a PC cluster is only marginally feasible, as CPUs are ill-suited for performing computations relying heavily on bit-permutations.

We thus decided to implement and experimentally verify the complex attack on dedicated reconfigurable hardware using the RIVYERA special-purpose hardware cluster, as described in Section 2.4. For the following design decisions, we remark that RIVYERA provides 128 powerful Spartan-3 FPGAs, which are tightly connected to an integrated server system powered by an Intel Core i7 920 with 8 logical CPU cores. This allows us to utilize dedicated hardware and use a multi-core architecture for the software part.

3.2 FPGA Design

In this section, we give an overview of the hardware implementation. As the total number of iterations for one attack (for the correct guess of the 39 secret expression values) is 2^d, the number of workers for an optimal setup has to be a power of two to decrease control logic and communication overhead.

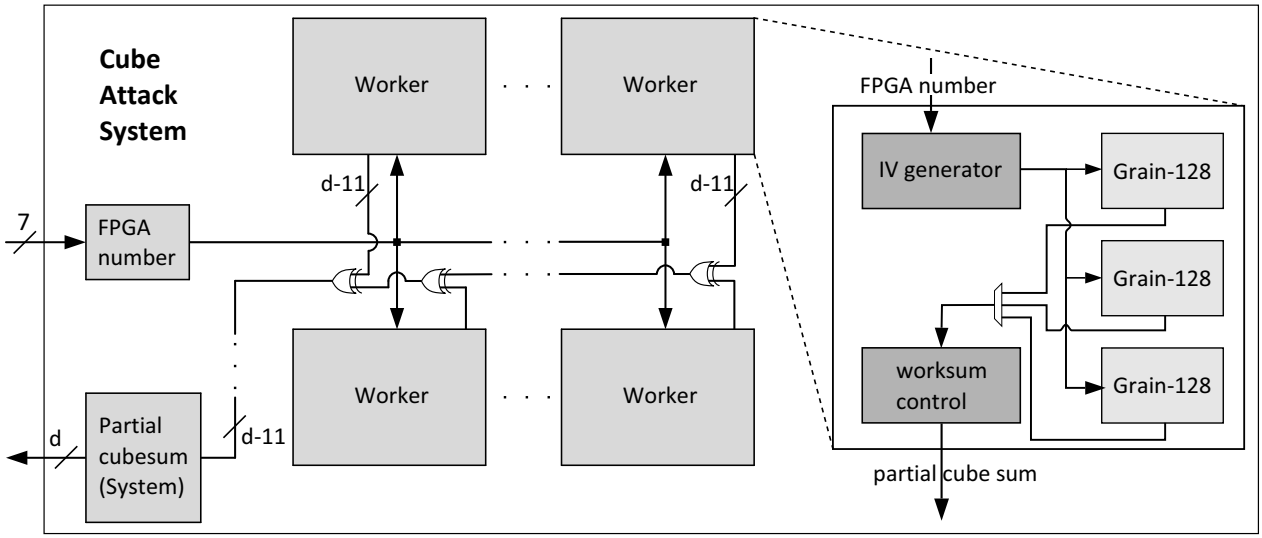

Fig. 4. FPGA Implementation of the online phase for cube dimension d.

Figure 4 shows the top level overview. Each of the 128 Spartan-3 5000 FPGAs features 2^4 independant workers and each of these workers consists of its own IV generator to control multiple Grain-128 instances.

The IV generator needs three clock cycles per IV and we need a corresponding number of Grain instances to process the output directly. As it is possible to run more than one initialization step per clock cycle in parallel, we had to find the most suitable time/area trade-off for the cipher implementation.

Table 2 shows the synthesis results of our Grain implementation. In comparison, Aumasson *et al.* also used 2^5 parallel steps, which is the maximum number of supported parallel steps without additional overhead, on the large Virtex-5 LX330 FPGA used in [14]. By using three Grain instances, we do not lose clock cycles where IV generation or cipher computation idle and - analyzing the critical path of the full design - the Grain module is not the limiting factor.

Table 2. Synthesis results of Grain-128 implementation on the Spartan-3 5000 FPGA with different numbers of parallel steps per clock cycle.

Parallel Steps	2^0	2^1	2^2	2^3	2^4	2^5
Clock Cycles (Init)	256	128	64	32	16	8
Max. Frequency (MHz)	227.169	226.706	236.016	234.357	178.444	159.210
Max. Frequency (MHz)	227	226	236	234	178	159
FPGA Resources (Slices)	165	170	197	239	311	418
Flip Flops	278	285	310	339	393	371
4 input LUTs	288	297	345	420	583	804

The results of the cipher instances are gathered in the worksum control, which updates the partial cubesum per worker, which is the output of all worker instances. The FPGA computes a partial cubesum of all workers on the FPGA and returns it upon request.

As mentioned before, it is not possible to create an unrestricted IV generation. To circumvent this problem, we locally fix certain values per key. This enables us to reduce the complexity of the system, as dynamic inputs are changed to constants. The drawback is that we need to generate a bitstream, which is dependent on the parameter set and - more important - on the key we wish to attack.

By looking at the discussion on the dynamic input to the IV generation, we can see that by fixing the parameter set, we already gain an advantage on the iteration over the cube itself: By sorting these positions and a smart distribution among the FPGAs, we reduce the complexity of the first part of the IV generation. By setting the base IV constant, we can optimize the design automatically and with the constant key, we remove the need to transfer any data to the FPGAs after programming them.

Nevertheless, the most important unknown are the polynomials. While we do have some restrictions from the way these polynomials (consisting of `and` and `xor` operations) are generated, we cannot forecast the impact of them: Remember that we use 13 different boolean functions in this parameter set. Each of these can have up to 50 monomials, where every monomial can - in theory - use all d positions of the cube. Luckily, on average, most polynomials depend on less than 5 variables.

3.3 Software Design

Now that we described the FPGA design and the need of key-dependent configurations, we will go into detail on the software side of the attack. In order to successfully implement and run the attack on the RIVYERA cluster and benefit from its massive computing power, we propose the following implementation. Figure 5 shows the design of the modified attack.

The software design is split into two parts: We use all but one core of the i7 CPU to generate attack specific bitstreams, i. e., configuration files for the FPGAs, in parallel to prepare the computation on the FPGA cluster. Each of these generated designs configures the RIVYERA for a complete attack on one random key. As soon as one bitstream was generated and waits in the queue, the remaining core programs all 128 FPGAs with it, starts the attack, waits for the computation to finish and fetches the results. With the partial cubesums per FPGA, the software computes the final result and evaluates the attack on the chosen key to measure the effectiveness of the attack.

In contrast to the first approach, which uses the generic structure realizable in software and needed a lot of communication, we generate custom VHDL code containing constant settings and fixed boolean functions of the polynomials derived from the parameter set and the provided key. Building specific configuration files for each attack setup allows us to implement as many fully functional, independent, parallel workers as possible without the area consumption of complex control structures. In addition, this allows us to strip down the communication interface and data paths to a minimum: only a single 7 bit parameter, distributing the workspace between all 128 FPGAs, is necessary at runtime to start the computation and receive a d bit return value. This efficiently circumvents all of the problems and overhead of a generic hardware design at the cost of rerunning the FPGA design flow for each parameter/key pair.

Please note that in this approach the host software modifies a basic design by hard-coding conditions and adjusting internal bus and memory sizes for each attack. We optimized this basic layout as much as

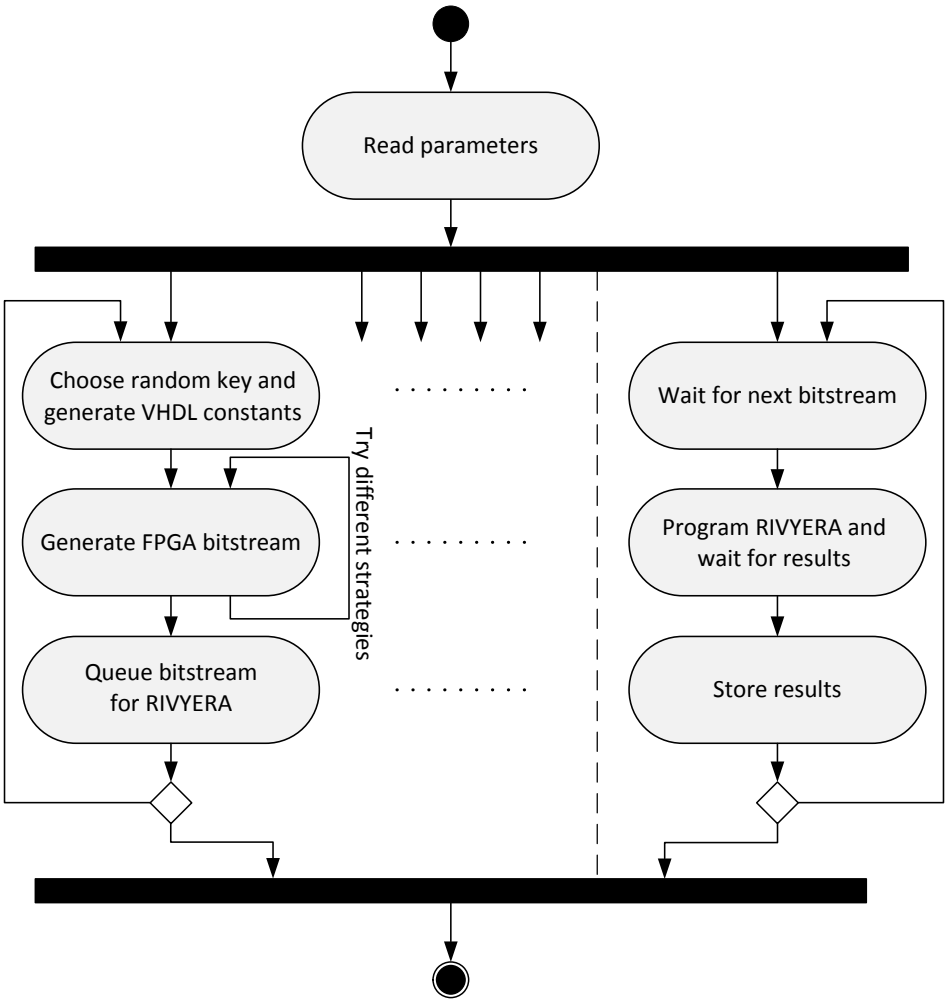

Fig. 5. Cube Attack Implementation on Special-Purpose Hardware

possible for average sized boolean functions, but the different choices of the polynomial functions lead to different combinatorial logic paths and routing decisions, which is bound to change the critical path in the hardware design. As the clock frequency is directly linked to the critical path, we implemented different design strategies as well as multiple fall-back options to modify the clock frequency constraints in order to prevent parameter/key pairs from resulting in an invalid hardware configurations. As a result, you can see a fallback path in Figure 2, which tries different design strategies automatically if the generated reports indicate failures during the process or timing violations after the analysis phase.

4 Results

In this section, we present the results of our implementation. The hardware results are based on Xilinx ISE Foundation 13 for synthesis and place and route. We compiled the software part using GCC 4.1.2 and the OpenMP library for multi-threading and ran the tests on the i7 CPU integrated in the RIVYERA cluster.

The hardware design was used to test different parameter sets and chose the most promising parameters. The resulting attack system for the online phase — consisting of the software and the RIVYERA cluster — uses 16 workers per FPGA and 128 FPGAs on the cluster in parallel. This means that the number of Grain computations per worker is reduced to 2^{d-11}, i. e., 2^{39} with the current cube dimension. The design ensures

that each key can be attacked at the highest possible clock frequency, while it tries to keep the building time per configuration moderate.

Table 3. Strategy Overview for the automated build process. The strategies are sorted from top to bottom. In the worst case, all 16 combinations may be executed.

Global Settings	Worker Clock (MHz)			
2.4× Input Clk	120			
2.2× Input Clk	110			
2.0× Input Clk	100			
RIVYERA Input Clk	50			
Map Settings	Placer Effort	Placer Extra Effort	Register Duplication	Cover Mode
Speed	Normal	None	On	Speed
Area	High	Normal	Off	Area
Place and Route Settings	Overall Effort	Placer Effort	Router Effort	Extra Effort
Fast Build	High	Normal	Normal	None
High Effort	High	High	High	Normal

Table 3 explains the different strategies in more detail: Each line represents one choice of settings, while the three blocks represent the impact on the subsequent build process. The design is synthesized with one of the four clock frequency settings. When the build process reaches the mapping stage, it tries first the speed optimized settings and runs the fast place and route. In case this fails, it tries the high effort place and route setting. If this fails, it tries the Area settings for the mapping and may fall back to a lower clock frequency setting, repeating the complete build process again.

As the user clock frequency of the RIVYERA architecture is 50 MHz, the Xilinx Tools will easily analyze the paths for a scaling factor 1.0 and 2.0. As the success rate when routing the design with 2.5 times the input clock frequency (125 MHz) was too low, we removed this setting due to the high impact on the building time.

Table 4. Results of the generation process for cubes of dimension 46, 47 and 50. The duration is the time required for the RIVYERA cluster to complete the online phase. The Percentage row gives the percentage of configurations built with the given clock frequency out of the total number of configurations built with cubes of the same dimension.

Cube Dimension d	46			47	50	
Clock Frequency (MHz)	100	110	120	120	110	120
Configurations Built	1	7	8	6	60	93
Percentage	6.25	43.75	50	100	39.2	60.8
Online Phase Duration	17.2 min	15.6 min	14.3 min	28.6 min	4h 10 min	3h 49 min

Table 4 reflects the results of the generation process and the distribution of the configurations with respect to the different clock frequencies. It shows that the impact of the unknown parameters is not predictable and that fallback strategies are necessary. Please note that the new attack tries to generate configurations for multiple keys in parallel. This process — if several strategies are tried — may require more than 6 hours before the first configuration becomes available. Smaller cube dimensions, i. e., all cube dimensions lower than 48, result in very fast attacks and should be neglected, as the building time will exceed the duration of the attack in hardware. Further note that the duration of the attack increases exponentially in d, e. g., assuming 100 MHz as achievable for larger cube dimensions, $d = 53$ needs 1.5 days and $d = 54$ needs 3 days.

5 Conclusions

Cube attacks and testers are notoriously difficult to analyze mathematically. To test the attack experimentally, to find and validate suitable parameters and to verify its complexity, we were restricted by the limitations of

CPU clusters, making further evaluations difficult. Due to its high complexity and hardware-oriented nature, the attack was developed and verified using the RIVYERA hardware cluster.

We presented an architecture making use of both the integrated i7 CPU and the 128 FPGAs of the RIVYERA cluster and heavily relying on the reconfigurability of the cluster system. This way, we were able to successfully implement the first attack on Grain-128, which is considerably faster than exhaustive search and, unlike previous attacks, makes no assumptions on the secret key.

While we were unable to conduct the full attack in this work, we can estimate its results by running a partial version. Our experimental results showed that for about 7.5% of the keys we could achieve a signficant improvement by a factor of 2^{38} over exhaustive search.

References

1. M. Hell, T. Johansson, A. Maximov, and W. Meier, "A Stream Cipher Proposal: Grain-128," in *Information Theory, 2006 IEEE International Symposium on*, july 2006, pp. 1614 –1618.
2. I. Dinur and A. Shamir, "Breaking Grain-128 with Dynamic Cube Attacks," in *FSE*, ser. Lecture Notes in Computer Science, A. Joux, Ed., vol. 6733. Springer, 2011, pp. 167–187.
3. S. Budiansky, *Battle of wits: the complete story of codebreaking in World War II*. Free Press, 2000.
4. M. Matsui, "The First Experimental Cryptanalysis of the Data Encryption Standard," in *CRYPTO*, ser. Lecture Notes in Computer Science, Y. Desmedt, Ed., vol. 839. Springer, 1994, pp. 1–11.
5. E. F. Foundation, *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*, M. Loukides and J. Gilmore, Eds. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1998.
6. T. Güneysu, T. Kasper, M. Novotny, and C. Paar, "Cryptanalysis with COPACOBANA," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 57, no. 11, pp. 1498–1513, 2008.
7. J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir, "Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium," in *FSE*, ser. Lecture Notes in Computer Science, O. Dunkelman, Ed., vol. 5665. Springer, 2009, pp. 1–22.
8. X. Lai, "Higher Order Derivatives and Differential Cryptanalysis," *Symposium on Communication, Coding and Cryptography*, pp. 227–233, 1994, Higher Order Derivatives and Differential Cryptanalysis. In *"Symposium on Communication, Coding and Cryptography", in honor of James L. Massey on the occasion of his 60th birthday*, pages1994.
9. M. Vielhaber, "Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack," *IACR Cryptology ePrint Archive*, vol. 2007, p. 413, 2007.
10. I. Dinur and A. Shamir, "Cube Attacks on Tweakable Black Box Polynomials," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, A. Joux, Ed., vol. 5479. Springer, 2009, pp. 278–299.
11. A. Joux, *Algorithmic Cryptanalysis*. Chapman & Hall/CRC, 2009.
12. I. Dinur, T. Güneysu, C. Paar, A. Shamir, and R. Zimmermann, "An Experimentally Verified Attack on Full Grain-128 Using Dedicated Reconfigurable Hardware," in *ASIACRYPT*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds., vol. 7073. Springer, 2011, pp. 327–343.
13. T. Güneysu, G. Pfeiffer, C. Paar, and M. Schimmler, "Three Years of Evolution: Cryptanalysis with COPACOBANA Special-Purpose Hardware for Attacking Cryptographic Systems," *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS*, 2009, "uneysu, Gerd Pfeiffer, Christof Paar, and Manfred Schimmler. Three Years of Evolution: Cryptanalysis with COPACOBANA. In *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2009*, September2009.
14. J.-P. Aumasson, I. Dinur, L. Henzen, W. Meier, and A. Shamir, "Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128," *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS*, September 2009.

A Appendix: Simulation Algorithm

Algorithm 2 The Dynamic Cube Attack Simulation

Input: 128-bit key K.
Input: Expressions $e_1, ..., e_{13}$ and the corresponding indexes of the dynamic variable $i_1, ..., i_{13}$.
Input: Big cube $C = (c_1, ..., c_{50})$ containing the indexes of the 50 cube variables.
Output: The score of K.

$S \leftarrow (0, ..., 0)$ ▷ the 51 cube boolean sums, where $S[51]$ is the sum of the big cube
$IV \leftarrow (0, ..., 0)$ ▷ as the initial 96-bit IV
for $j \leftarrow 1$ **to** 13 **do**
    $e_j \leftarrow eval(e_j, K)$ ▷ Plug the value of the secret key into the expression
end for
for all cube indexes CV **from** 0 to 2^{50} **do**
    for $j \leftarrow 1$ **to** 50 **do**
        $IV[c_j] \leftarrow CV[j]$ ▷ Update IV with the value of the cube variable
    end for
    for $j \leftarrow 1$ **to** 13 **do**
        $IV[i_j] \leftarrow eval(e_j, IV)$ ▷ Update IV with the evaluation of the dynamic variable
    end for
    $b \leftarrow$ Grain-128(IV, K) ▷ Calculate the first output bit of Grain-128
    for $j \leftarrow 1$ **to** 50 **do**
        if $CV[j] = 0$ **then**
            $S[j] \leftarrow S[j] + b \pmod 2$ ▷ Update cube sum
        end if
    end for
    $S[51] \leftarrow S[51] + b \pmod 2$
end for
$HW \leftarrow 0$
for $j \leftarrow 1$ **to** 51 **do**
    if $S[j] = 0$ **then**
        $HW \leftarrow HW + 1$.
    end if
end for
return $HW/51$