A Framework for the Design of One-Way Hash Functions Including Cryptanalysis of Damgård's One-Way Function Based on a Cellular Automaton

Joan Daemen, René Govaerts and Joos Vandewalle

Katholieke Universiteit Leuven, Laboratorium ESAT,
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium.

Abstract

At Crypto '89 Ivan Damgård [1] presented a method that allows one to construct a computationally collision free hash function that has provably the same level of security as the computationally collision free function with input of constant length that it is based upon. He also gave three examples of collision free functions to use in this construction. For two of these examples collisions have been found[2] [3], and the third one is attacked in this paper. Furthermore it is argued that his construction and proof, in spite of their theoretical importance, encourage inefficient designs in the case of *practical* hash functions. A framework is presented for the *direct* design of collision free hash functions. Finally a concrete proposal is presented named Cellhash.

1 Introduction

A *collision free* hash function h is an easily computable function that maps strings of arbitrary length to strings of some fixed length in such a way that finding two strings x and x' , with $h(x) = h(x')$ is computationally infeasible. The pair x, x' is called a collision for h . A *collision free function* f from m bits to t bits with $m > t$ is an easily computable function that maps strings of length m to strings of length t , in such a way that finding a pair of strings x, x' of length m , with $f(x) = f(x')$ is computationally infeasible.

The presence of the term 'computationally infeasible' makes the previous definitions somewhat fuzzy. By interpreting this term in different ways, we get different definitions of collision free functions or collision free hash functions. At one end of

the spectrum there is the pragmatic point of view and at the other the complexity theoretic framework.

For readability a collision free hash function will be denoted by ‘hash function’ and a collision free function with an output shorter than its input by ‘CF function’ in the sequel. A function that takes an input of fixed length to an output of fixed length that is shorter than the input will be denoted by ‘FI function’. A CF function is a special case of a FI function.

In [1] a construction is given where finding a collision for the hash function yields a collision for the underlying CF function. Hence the hash function is provably as secure as the CF function. This reduces the design to that of the CF function. It is widely believed that this is much easier. In our opinion the direct design of a hash function is not harder and leads to more efficient, i.e. faster hash functions.

With the breaking of the Damgård function based on cellular automata in the following section, all three CF functions proposed in [1] are broken. This does not necessarily mean that collisions have been found for the corresponding hash functions built on these CF functions. However, because the underlying FI functions are shown not to be computationally collision free, the proof of Damgård does not apply. The hash functions are robbed of their *provable* collision free-ness. For the hash function based on cellular automata we show that the probability of the hash value being unaffected by the complementation of certain messagebits is large. Hence collisions for the hash function can be created ad libitum.

We give a general framework that allows the direct design of hash functions. Finally a concrete proposal for a hardware oriented high speed hash function is described. It is named Cellhash and its design follows this framework. Like Damgård’s CF function it makes use of cellular automata, but this time in a well-considered way.

2 Cryptanalysis of Damgård’s CF Function

Damgård’s CF function $g_c()$ computes a 128-bit string h starting from a 256-bit string x . We have $h = g_c(x)$. In the following we will explain in an informal way how it operates and how to create collisions. The function is based on a cellular automaton thoroughly investigated by Wolfram in [4]. When using results of Wolfram the reader is referred to [4] for a proof or further explanation.

A constant 256-bit string z is appended to x resulting in a 512-bit string. This is taken to be the initial state $a^0 = a_0^0 a_1^0 \dots a_{511}^0$ of a 512-bit binary cellular automaton(CA) with periodic boundary conditions[4]. When the CA is iterated the bits of

Figure 1: Space-time diagrams of the cellular automaton operation of the CF function. **a:** Propagation of information. **b:** Scheme for creating collisions. **c:** Upper left corner of b, black denotes 1, white denotes 0. **d:** bitwise XOR of two space-time diagrams for arguments of the function with the same output h .

the state are modified according to the local updating rule ($\oplus$ denotes XOR, $\vee$ denotes OR)

$$a_i^{t+1} = a_{i-1}^t \oplus (a_i^t \vee a_{i+1}^t)$$

where a_i^t denotes the i -th bit of the state of the CA after t iterations. The cellular automaton performs 384 iterations and the output h consists of the concatenation of the first bits of the strings a^{257} to a^{384} , hence $h = h_0 h_1 \dots h_{127}$ with $h_i = a_0^{257+i}$. h can be seen as the concatenation of subsequent values in time of one bit of the state, i.e. a temporal sequence. The state of the CA at a certain moment can be seen as an array of values in space, i.e. a spatial sequence. Figure 1.a gives a space-time diagram [4] of the computation. Bit a_i^t is at the position i units to the right and t units downwards from the upper left corner of the diagram. The top line of the diagram is the initial state a^0 containing x and z . Because there are periodic boundary conditions the right and left edge are in fact adjacent and the diagram can be seen as an unfolded cylinder. Mathematically this implies that the lower indices should be taken modulo 512.

Because of the local nature of the CA updating each bit a_i^t can only affect bits a_{i-1}^{t+1} to a_{i+1}^{t+1} of the next state. Hence each bit a_i^t can only depend on bits a_{i-1}^{t-1} to a_{i+1}^{t-1} of the previous state. Applying this recursively implies that a_i^t depends only on a_{i-k}^{t-k} to a_{i+k}^{t-k} of the state k iterations ago. The union of these sets for all k is called the dependence area of a_i^t , denoted by $\mathcal{A}_i^t$. In words: if all bits in $\mathcal{A}_i^t$ are left unchanged, a_i^t cannot change, whatever happens outside of $\mathcal{A}_i^t$. In Figure 1.a the borders of the dependence area of the last bit of h are indicated. The left border is line ℓ with slope $+1$ and the right border is r_1 with slope -1 . This comprises the dependence areas of all other bits of h that are mere translations upwards of the indicated area. Hence the indicated area can be seen as the dependence area of all h and will be denoted by $\mathcal{H}$.

However, the updating rule is asymmetric, and so is the propagation of information [4]. It turns out that for *almost all* initial states a^0 , $\mathcal{H}$ is bounded at the right by a border that is much steeper than r_1 . In Figure 1.a the actual border is approximated by a diagonal r_2 with slope -3 such that all of a_0 lies in $\mathcal{H}$. Of course we can't just choose this slope. In reality it is fixed by the initial state a_0 . For most a_0 however r_2 can be fairly well approximated by a straight line with slope -4 , hence we've been rather pessimistic. This implies that for these states some bits of a_0 lie outside of $\mathcal{H}$, providing a first type of collisions. For most inputs the output is independent of the bits a_{126-k}^0 to a_{126}^0 for some i . Hence *for the majority of inputs x there exists an integer $k > 0$ such that all x' that only differ from x in bit positions $126 - k$ to 126 have the same output h , leading to 2^k simultaneous collisions.*

But even *larger sets of simultaneous collisions can be found.* If the initial state

contains a substring that exhibits the alternating pattern (010101...), a remarkable observation can be made. As can be seen in Figure 1.c the left edge of the pattern shifts one bit the right each timestep and the right edge follows an irregular path that moves on the average one bit to the *right* per three iterations[4]. This pattern is stable with respect to the updating rule, in that it expands, albeit slowly, to the right. With respect to information propagation, this pattern acts as a buffer, as can be seen in Figure 1.d. Here the bitwise XOR of Figure 1.c with another space-time diagram is shown. The initial state of the second space-time diagram differs only from that in Figure 1.c in the 64 bits on the right from the alternating pattern. It can be seen that the differences in the subsequent states propagate in the cone bounded at the left by the alternating area. This can be used to construct collisions as illustrated in Figure 1.b.

The description in [1] does not specify the constant z . However our attack works independently of the particular value of z . x is the concatenation of x_s (64 bits), x_v (62 bits) and x_c (130 bits). x_s contains the alternating pattern. This results in the division of Figure 1.b. Area II contains the alternating pattern that ‘buffers’ changes in area III from area I. The dependence area for this configuration is formed by area I. Figure 1.c and 1.d are close ups of the upper left corner of Figure 1.b. Figure 1.c clearly shows area II and in Figure 1.d the difference pattern originating from two initial states that differ only in x_v is seen to be limited to area III. In conclusion: all x that have the same x_c and for which x_s contains the alternating pattern have the same output. This yields 2^{130} different sets, each containing 2^{62} inputs with the same output h . Finding these sets requires no computations whatsoever.

3 Collisions for the Hash Function Itself

The input to the hash function is a message m that has been padded in a one-to-one way to a multiple of 128 bits. Suppose $m = m_0 \parallel m_1 \dots \parallel m_n$, with every m_i a 128 bit string. The hash value h is the result of applying the hash function $h_c()$ to m . We have $h = h_c(m)$. This function is defined in terms of the CF function g_c .

$$h^i = g_c(h^{i-1} \parallel m_{i-1}) \quad \text{for} \quad i = 1, \dots, n$$

h^0 is equal to IV , a fixed initial value that is part of the specification of $h_c()$. The hashvalue h is given by h^n . Suppose we have an intermediate hash value h^i . h^{i+1} is then computed by $h^{i+1} = g_c(h^i, m_i)$. In figure 2.a it can be seen that changes in the first k bits of m_i (denoted by $\leftrightarrow$) influence only the $k + 1$ last bits of h^{i+1} (denoted by $\Leftrightarrow$) if the border β is steep enough. This is the case for almost all initial configurations. Now suppose the last bit of h^{i+1} is not affected. This happens with

Figure 2: Space-time diagrams of two subsequent applications of $g_c()$. **a:** The first k bits of m_i influence only the last $k + 1$ bits of h^{i+1} **b:** h^{i+2} is independent of the $k + 1$ last bits of h^{i+1}

probability $1/2$. In the next step h^{i+2} is computed from h^{i+1} and m_{i+1} . In figure 2.b it can be seen that, for small k there is a large probability that h^{i+2} will be independent of the k bits of h^{i+1} that were affected. From this it can be concluded that changes in bits that are situated at the beginning of the blocks m_i have a fair probability not to influence the hash value h .

4 Provably secure hashing vs. hashing in practice

Suppose $g()$ is a CF function for which it can be proven that finding a collision is equivalent to solving a hard problem. With Damgård's method a hash function $h()$ can now be constructed based on $g()$. Finding a collision for $h()$ is equivalent to solving the hard problem. However it cannot be proven that a certain problem is hard, i.e. that there is no efficient (e.g. polynomial) algorithm to solve it. Hence the security of $h()$ ultimately depends on the unproven assumption of the hardness of the problem. If the assumption is widely accepted, the hash function is widely considered to be secure. Complexity theory provides a fundament to a variety of widespread assumptions. Furthermore, there is emerging a vast literature treating hash functions in a complexity theoretic setting.

In practice, very fast hash functions are required. However, CF functions that are based on some widespread complexity assumption require a lot of calculation and are generally slow. Hence for the sake of speed practical hash function have been proposed that are based on functions for which finding collisions *seems* hard. These functions often incorporate elements that *remind of* hard problems. Damgård's[1] three CF functions are of this type. They are 'based' respectively on modular

squaring, knapsack and cellular automata. The inventors of this type of functions assume that the use of the calculations that are related to some hard problem will provide the needed security. The fact that most of these functions have been broken, suggests that this is not a sound argument.

A more rational approach is the construction of a hash function using a FI function that is *designed to be collision free*. If the correct construction is used the Damgård proof applies. Observe that the hash function is not computationally collision free, but that finding a collision for the hash function is at least as hard as finding a collision for the underlying FI function. The distinction between these functions and the ones described in the previous paragraph lies in the design of the presumed CF functions. The ones meant in this paragraph are those designed using sound principles of diffusion and confusion. The ones in the previous paragraph are not really designed but merely ‘based’ on hard problems.

Another class of hash functions makes use of an FI function that is not especially designed to be collision free. The security of the hash function is based on the fact that collisions for the FI function are of no use in the construction of a collision for the hash function. Damgård’s proof does not apply in this case.

The last two classes comprise most of the practical hash function designs. There are numerous designs where the FI function is realized by a block cipher. Because a block cipher is designed for another purpose, a dedicated function is likely to provide a more efficient solution. Two recent examples of dedicated hash functions are Snefru[6] and MD4[7]. The design of the underlying functions is based on considerations of diffusion and confusion.

To our knowledge all proposed hash functions can be described by the following general scheme:

- Segmentation: The input is padded in a prescribed way and subdivided in a number of blocks of equal length.
- Initialization: The initial value of the *intermediate hash result* is fixed by the specification.
- Iteration: For all blocks: the FI is applied to block i and the intermediate hash result. the output is the new intermediate hash result. Observe that each block is used only once.
- Result: The hash result is the output of the last iteration. Eventually a specified compression may take place, e.g. not all bits may be used.

In the following section we will present a framework that is even more general. It allows the design of hash functions that don’t have certain restrictions imposed by

this description. Because all proposed hash functions can be modeled to fit in our framework, it can also be considered as an alternative viewpoint. The framework is partly inspired by the design of MD4[7].

5 A Framework for the Direct Design of Hash Functions

The security of the hash function should be guaranteed by the diffusion and confusion of the messagebits in a hash result. In our framework this is realized in a way similar to the diffusion and confusion of the key in a symmetric block cipher, e.g. DES. The security of the overall scheme is based on executing a number of rounds each performing easily implementable functions. This is exactly what happens in the FI function in MD4. It is however different in that the number of rounds in our framework is not constant but depends on the length of the input, i.e. the message. In practical designs the number of rounds will be proportional to the length of the message, with a lower bound for short messages.

The scheme of the framework is given in Figure 2. It can be seen as an ‘encryption’ of the initial value IV that is a constant for a particular scheme. The number of rounds is $k(n)$. Each round consists of the application of a function $f_{j,n}$ on the intermediate hash result H^{j-1} . Each $f_{j,n}$ is ‘keyed’ by a string m_j , consisting of messagebits specified by a selection window $s_{j,n}$. The output after $k(n)$ rounds is the hash result. A concrete design has to define the padding, the function $k(n)$, the functions $f_{j,n}$ and the selection windows $s_{j,n}$ for all n and $1 \leq j < k(n)$.

In using this framework to design a hash function, a number of problems arise because of the variability of the number of rounds. It is for instance not clear how to devise a general selection schedule s for a variable length input. This problem *can* be tackled by subdividing the (possibly padded) message M in a number of equal blocks and designing the scheme for one such block. This comes down to designing an IF function that inputs one block and the intermediate hash result. This is what happens in *all* proposed hash functions thus far.

This is however not *the* solution, but just *one of the many*. Instead of being a design *principle*, this blocking is a design *option*. When blocking is applied and the designer wants to apply the Damgård proof, he has to construct the FI functions to be collision free. This restricts these FI functions in an unnecessary way. Seen within our framework, the advantage of having a FI function that is designed to be collision free is no longer self evident. What is relevant here is the contribution of the FI function to the purpose of the hash function related to its ‘computational cost’, i.e. its effectiveness and efficiency.

A global approach can lead to more elegant solutions and a better use of the

Figure 3: Schematic overview of the framework.

available computing resources, both in a hardware as in software oriented designs. Although we believe that the Damgård proof is of no value here, it keeps on being a valuable and necessary step in the construction of computationally collision free hash functions based on CF functions for which finding a collision is equivalent to solving some (generally assumed) hard problem.

It is impossible to give a complete list of design criteria for hash functions in our framework. However, in the rest of this section we will informally describe some general criteria for the components of the scheme. This will be illustrated by the design of our concrete proposal Cellhash. A more theoretical treatment is in preparation.

- Size: To prevent a birthday attack the hash result has to be at least 128 bits.
- Diffusion: The functions $f_{j,n}$ have to guarantee diffusion of information. With equal keys, two inputs H^{j-1} and $\tilde{H}^{j-1}$ that differ in only a *few* bits must give rise to two outputs H^j and $\tilde{H}^j$ differing in *substantially more* bits even in worst case. If the $f_{j,n}$ are invertible for a fixed key m_j it is guaranteed that no two inputs give the same output. The key-dependence of the functions has to maximize the diffusion of the messagebits into the hash result. Therefore it is required that each $f_{j,n}$, with the input fixed and seen as a function from key to output, is an injection for all possible inputs.
- Confusion: The hash result has to depend on the messagebits in an involved and complicated way. Therefore it is necessary that the $f_{j,n}$ provide the indispensable nonlinearity (in relation to all thinkable algebras). To promote confusion the selection schedule has to use each and every messagebit several times. Preferably a messagebit would be selected in a number of rounds that are *mutually distant in time*. The rounds that precede and succeed these would preferably use as many different messagebits as possible.
- Feasibility: The goal is a hash function actually working at *high speed*. A succinct and clear description is an advantage both in software, leading to a short program, and in hardware, possibly leading to a simple and compact device. Moreover, a long or complicated description enlarges the probability of implementation errors.
 - Software: Specific widespread instructions of processors can be advantageously used in the $f_{j,n}$. Restrictions on the selection schedule are imposed by the *relatively* slow memory access in contrast to the access of on-chip registers or cache.

- Hardware: For speed, the implementation should be on one chip, with the traffic onto and off the chip minimized. The $f_{j,n}$ must be decomposable in a limited number of primitives, which are actually implemented. By parallel computation optimal use can be made of available storage and time.

6 Cellhash: A Hardware oriented Hash Function

Cellhash is a hash function capable of very high speeds if implemented in hardware. In applications where speed is important and very large amounts of data have to be authenticated (e.g. electronic financial transactions, software integrity) hardware implementations are the natural solution. The core of Cellhash is formed by two cellular automata operations and a permutation. These guarantee the confusion and diffusion properties and allow for an efficient one chip hardware implementation by well established methods[11]. By adoption of Cellhash (or a similar algorithm) as a standard, costs would be reduced even further because of the high quantities of chips produced. This low cost opens up the possibility of the installation of these IC's in a standard way on computers.

6.1 The Global Scheme

The hash result of a message M of length n is computed in two phases :

- Preparation of the message

The message is extended with the minimum number of 0's so that its length in bits is at least 248 and congruent to 24 modulo 32. The number of bits added is represented in a byte that is subsequently appended, most significant bit first. At this point the resulting message has a length that is an exact multiple of 32 bits. The resulting message can be written as $M_0M_1 \dots M_{N-1}$, i.e. the concatenation of N (32-bit) words M_i .

- Application of the hash function

$F_c(H, A)$ is a function with argument H a bitstring of length 257 and A a bitstring of length 256. It returns a bitstring of length 257. IV is the all-zero bitstring of length 257.

$$H^0 = IV$$

$$H^j = F_c(H^{j-1}, M_{j-1}M_{j \bmod N} \dots M_{j+6 \bmod N}), j = 1 \dots N$$

H^N is the hash result

Hence the number of rounds $k(n) = \max(8, \lceil \frac{n+8}{32} \rceil)$, All functions $f_{j,n}$ are the same, namely F_c , and each selection window s_j selects messagewords M_{j-1} to M_{j+6} . By

reaching the end of the message the first 224 bits can be seen as repeated (e.g. if j is $N - 3$, s_j selects $M_{N-4}M_{N-3}M_{N-2}M_{N-1}M_0M_1M_2M_3$). In this way every bit of the message is used exactly 8 times in the hashing process.

6.2 Description of $F_c(H, A)$

The computation can be considered as a 5-step transformation of H . The calculations in each step are done simultaneously on all bits of H . Let $h_0h_1 \dots h_{256}$ denote the bits of H and $a_0a_1 \dots a_{255}$ the bits of A . All indices should be taken modulo 257, $\vee$ means OR and $\oplus$ means XOR.

$$\begin{aligned} \text{Step 1 : } & h_i = h_i \oplus (h_{i+1} \vee \bar{h}_{i+2}), & 0 \leq i < 257 \\ \text{Step 2 : } & h_0 = \bar{h}_0 \\ \text{Step 3 : } & h_i = h_{i-3} \oplus h_i \oplus h_{i+3}, & 0 \leq i < 257 \\ \text{Step 4 : } & h_i = h_i \oplus a_{i-1}, & 1 \leq i < 257 \\ \text{Step 5 : } & h_i = h_{10*i}, & 0 \leq i < 257 \end{aligned}$$

The five steps are further clarified in Figure 3. Step 1 is a nonlinear cellular automaton (CA) operation where each bitvalue h_i is updated according to the bitvalues in its neighborhood (in this step and step 3 periodic boundary conditions apply). The nonlinearity of the updating rule has to guarantee the needed confusion. This particular CA operation is invertible if the length of H is odd [9]. Step 2 consists merely of complementing 1 bit to eliminate circular symmetry in case bitstring A consists of only 0's. Step 3 is a linear CA operation that greatly increases the diffusion. This step is invertible if the length of H is no multiple of 9 [9]. In step 4 the actual messagebits are injected in H , to be diffused and confused during subsequent rounds. Step 5 is a bit permutation where bits are placed away from their previous neighbors. This step is needed to maintain a permanent diffusion during the hashing process. The length of H is 257 (a prime) to make step 1 and 3 invertible and to avoid circular symmetric patterns in H .

6.3 Properties

- **Size**

The hash result is 257 bits long. If it is more convenient to use only 256 bits of the hash result, its first or last bit can easily be omitted without jeopardizing the security significantly.

- **Diffusion**

Each bit of H^i depends on 9 bits of H^{i-1} . This dependence is linear in 3 bits. Because of the bit permutation in step 5, the neighborhood is 'refreshed' each

Figure 4: Schematic overview of the calculation of one output bit using the F_c function. It clearly shows that each output bit depends on 9 input bits and that nearby output bits depend on non-overlapping sets of input bits.

iteration. By this we mean that nearby bits of H^i depend on non-overlapping sets of bits of H^{i-1} . This property assures that *every* bit of an intermediate hash result H^i depends on *all* bits of H^{i-3} , the intermediate hash result three rounds before, in a nonlinear way. Moreover, F_c is invertible if the key A is regarded as a constant.

• Confusion

This paragraph uses ideas of differential cryptanalysis as described in [8]. With the same m_i , a difference pattern in H^i (input XOR's) gives rise to a number Λ of possible difference patterns in H^{i+1} (output XOR's) *each appearing with the same probability* $1/\Lambda$. We have

$$\Lambda = 2^{k+p}$$

where k is the number of 1's and p is the number of 001 patterns in the input XOR. So only input XOR's with few 1's give rise to high probability characteristics. These phenomena are caused by the nonlinearity and the regular structure of the cellular automaton mechanism in step 1. Because of the diffusion caused by step 3 and 5, output XOR's of these characteristics consist of isolated 1's on a zero background. On the input of the next F_c iteration these will give rise to characteristics with high Λ . Hence simple difference patterns in H^i give rise to a vast amount of possible difference patterns, each with very

small probability, only a few rounds later. Moreover every bit of the message is selected 8 times in the hashing process. Because of the diffusion mechanism in F_c described above, the influence of the first injection of a messagebit has already spread out over all bits of the intermediate hash result by the time of the last injection.

- **Hardware Feasibility**

The description allows a straightforward chip implementation[11]. One round comprises one F_c calculation and the parallel input of 32 new messagebits and can be performed in one clockcycle. The total hashing process takes one round per 32 bits and 8 supplementary initial rounds for the installation of the first 256 messagebits. A very conservative guess of about 10 Mhz clockfrequency[11] gives a hashing speed of 0.3 Gbit/sec. for long messages.

- **Provable Security ?**

The designers of Cellhash do not claim that finding a collision for Cellhash is equivalent to any hard problem other than finding a collision for Cellhash.

7 Conclusions

Collisions have been found for Damgård's CF function based on cellular automata. A general framework is presented that allows the direct design of hash functions. A concrete proposal is described named Cellhash. Unlike Snefru and MD4, Cellhash is designed for hardware implementation, making extremely high speed possible.

8 Acknowledgements

The authors would like to thank Ivan Damgård[10]. There has been an extensive revision of this paper in consequence of his comments.

References

- [1] I. Damgård, Design Principles for Hash Functions, in *Advances in Cryptology: Proceedings of Crypto '89*, 416–427, Springer-Verlag, 1990.
- [2] P. Camion, J. Patarin, The Knapsack Hash Function proposed at Crypto '89 can be broken, *Advances in Cryptology–Eurocrypt '91*, Proceedings, Springer-Verlag, to appear.
- [3] Bert Den Boer, *Internal Report RIPE*.

- [4] S. Wolfram, Random Sequence Generation by Cellular Automata, *Advances in Applied Mathematics*, **7** (1986) 123–169.
- [5] W. Meier, O. Staffelbach, Analysis of Pseudo Random Sequences Generated by Cellular Automata, *Advances in Cryptology–Eurocrypt ’91*, Proceedings, Springer-Verlag, to appear.
- [6] R. Merkle, A Fast Software One-Way Hash Function, *Journal of Cryptology*, **3** (1990) 43–58.
- [7] R. Rivest, The MD4 Message Digest Algorithm, *Abstracts of Crypto ’90*, 281–291.
- [8] E. Biham, A. Shamir, Differential Cryptanalysis of DES-like Cryptosystems, *Abstracts of Crypto ’90*, 1–32 .
- [9] J. Daemen, R. Govaerts, J. Vandewalle, Properties of Partially Linear Cellular Automata, *Internal Report ESAT*.
- [10] Ivan Damgard, *personal communication*.
- [11] I. Verbauwhede and Z. Sahraoui of IMEC (Interuniversitair Micro Electronica Centrum), *personal communication*.