An Improved Hardware Implementation of the Quark Hash Function

Shohreh Sharif Mansouri(✉) and Elena Dubrova

Department of Electronic Systems, Royal Institute of Technology, Stockholm, Sweden
{shsm, dubrova}@kth.se

Abstract. We present an implementation of U-Quark, the lightest instance of the Quark family of hash functions, which is optimized for throughput. The throughput is increased by converting the Feedback Shift Registers (FSRs) of Quarks permutation block from the original Fibonacci configuration to the Galois configuration. In this way, the complex feedback functions of the FSRs are decomposed into several simpler feedback functions. As a result, the throughput of U-Quark is increased by 34 % on average without any area penalty. The power consumption of the hash function also improves by 19 %.

1 Introduction

The Quark family of cryptographic hash functions [1] is based on a *sponge construction*, an architecture that minimizes memory requirements and targets the implementation of cryptographic algorithms in highly-constrained environments such as RFID systems [2]. As a sponge construction, Quark can be used for message authentication, stream encryption or authenticated encryption.

The permutation block of Quark is based on shift registers and is inspired by two low-weight ciphers: the stream cipher Grain [3–5] and the block cipher KATAN [6].

Three Quark instances have been proposed [1]: U-Quark, which is the smallest design and provides at least 64-bit security against crypto-attacks such as collisions, multicollisions, distinguishers, etc.; D-Quark, which is the second lightest instance of Quark and provides at least 80-bit security against the same attacks; S-Quark, which is the heaviest instance and provides at least 112-bits security against the attacks. For all three instances, the level of security against pre-image attacks is double compared to the other attacks (i.e. it is 128 bits for U-Quark, 160 bits for D-Quark and 224 bits for S-Quark).

In this work we optimize the implementation of Quark in terms of throughput. To do so, we modify Quark's permutation block without changing its functionality (and thus also without losing any of its security properties). We transform the Non-Linear Feedback Shift Registers (NLFSRs) of the permutation block, originally given in Fibonacci configuration, into Galois NLFSRs. Fibonacci NLFSRs have feedback only on the input state bit while Galois NLFSRs have

M. Hutter and J.-M. Schmidt (Eds.): RFIDsec 2013, LNCS 8262, pp. 113–127, 2013.
DOI: 10.1007/978-3-642-41332-2_8,
© Springer-Verlag Berlin Heidelberg 2013

several simpler feedbacks on different state bits: the latter are therefore normally characterized by better throughput. A Fibonacci-to-Galois transformation for NLFSRs was described in [7]. Here we extend this original transformation so that it allows dealing with external inputs, multiple outputs and parallel loading, all features that are needed for U-Quark. Due to the characteristics of U-Quark, we also need to modify the structure of the hash function for the transformation to be successful. This modification does not modify the functionality of the hash function but introduces a 9 cycles latency for the generation of the hash in case the initial state of the hash function is not fixed and can change from run to run.

We limit our analysis and our experiments to U-Quark. However, since all three Quark instances have the same hardware structure, the presented technique can be applied also to D-Quark and S-Quark.

The hash function throughput is increased by 34 %, without any area overhead, while its power consumption is also reduced by 19 %.

2 The Quark Hash Function

2.1 General Structure

The operation of a sponge construction is shown in Fig. 1.

A sponge construction is characterized by different parameters: the output length n, the rate (or block size) r and the capacity c [1,2].

The size of the internal state of a sponge construction is given by the width $b = r + c$. The state bits are denoted as:

$$s_0, s_1, ..., s_{b-1}$$

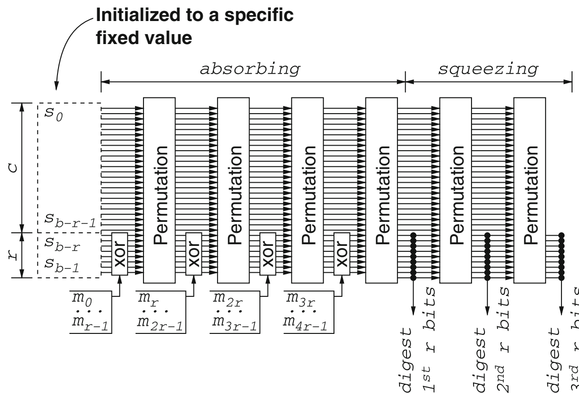

Fig. 1. Operation of a sponge construction

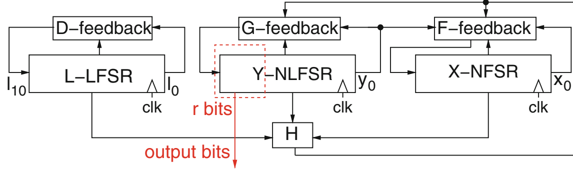

Fig. 2. Quark's permutation block

A sponge construction goes through three phases:

- *Initialization*: The initial state of the hash function is set to a fixed value, which is specific for every Quark instance; the message is padded by appending a '1' bit followed by the minimal number of '0' bits, so that the length of the message is a multiple of r.
- *Absorbing phase*: One r-bit message block is XORed with the last r bits of the state $(s_{b-r}, ..., s_{b-1})$. Then, a permutation P is applied to all the state bits. The process is repeated until all message bits have all been "absorbed".
- *Squeezing phase*: the last r bits $s_{b-r}, ..., s_{b-1}$ of the state are returned as output, then the permutation P is applied, and the process is repeated until n bits have been "squeezed out".

All instances of Quark are parametrized by different values of n, r and c. For all instances of Quark, n is constrained to be equal to $b = c + r$.

The sponge construction can be implemented serially, with a single permutation block. In alternative, to increase throughput, it can be parallelized by introducing more than one permutation block. This last solution is not taken into account in this work.

2.2 Permutation

Quark's permutation block is inspired by Grain [3–5] and KATAN [6]. Its structure is shown in Fig. 2.

The permutation block contains two $b/2 = m$ NLFSRs, called X-NLFSR and Y-NLFSR, whose state bits are respectively denoted as:

$$x_0, x_1, ..., x_{m-1}$$

and

$$y_0, y_1, ..., y_{m-1}$$

It also contains a $k = \lceil log_2 (4b) \rceil$ LFSR, called L-LFSR, whose state bits are denoted as:

$$l_0, l_1, ..., l_{k-1}$$

Thus, the permutation block contains $b + k$ state bits split between the three feedback shift registers. These should not be confused with the b state bits $s_0, s_1, ..., s_{b-1}$ of Quark described in Sect. 2.1.

A permutation P takes $4b$ cycles to complete. The permutation is split between three phases (not to be confused with the sponge construction phases described in Sect. 2.1):

- *Initialization*: The X-NLFSR is initialized with the $b/2$ lowest-grade state bits of Quark ($x_0 = s_0, ..., x_{m-1} = s_{m-1}$). The Y-NLFSR is initialized with the $b/2$ second lowest-grade state bits ($y_0 = s_m, ..., y_{m-1} = s_{2m-1}$). The L-LFSR is initialized with all ones ($l_0 = 1, ..., l_{k-1} = 1$).
- *State Update*: The permutation block is clocked for $4b$ cycles. For the X-NLFSR, elements x_i are updated to x_{i+1} for $0 \leq i < m-1$; x_{m-1} is updated to $f(x,y) \oplus h(x,y,l)$. For the Y-NLFSR, elements y_i are updated to y_{i+1} for $0 \leq i < m-1$; y_{m-1} is updated to $g(y) \oplus h(x,y,l)$. For the L-LFSR, l_i is updated to l_{i+1} for $0 \leq i < k-1$; l_{k-1} is updated to $d(l)$. The functions $f(x,y)$, $g(y)$, $d(l)$ and $h(x,y,l)$ vary between the three Quark instances. U-Quark's functions are reported in Sect. 2.3.
- *Output Write-Back*: The updated state bits of Quark are read from the X-NLFSR and the Y-NLFSR. The $b/2$ lowest-grade state bits are read from the X-NLFSR ($s_0 = x_0, ..., s_{m-1} = x_{m-1}$). The $b/2$ second lowest-grade state bits are read from the Y-NLFSR ($s_m = y_0, ..., s_{2m-1} = y_{m-1}$).

2.3 U-Quark

For U-Quark, $r = 8$, $c = 128$ and $n = b = 136$. The LFSR contains $k = \lceil log_2\,(4b) \rceil = 10$ state bits. The functions $f(x,y)$, $g(y)$, $d(l)$ and $h(x,y,l)$ are respectively equal to:

$$
\begin{aligned}
g(y) = {} & y_0 \oplus y_7 \oplus y_{16} \oplus y_{20} \oplus y_{30} \oplus y_{35} \oplus y_{37} \oplus y_{42} \oplus y_{51} \oplus \\
& y_{54} \oplus y_{49} \oplus y_{58}y_{54} \oplus y_{37}y_{35} \oplus y_{15}y_7 \oplus y_{54}y_{51}y_{42} \oplus \\
& y_{35}y_{30}y_{20} \oplus y_{58}y_{42}y_{30}y_7 \oplus y_{54}y_{51}y_{37}y_{35} \oplus y_{58}y_{54}y_{20}y_{15} \oplus \\
& y_{58}y_{54}y_{51}y_{42}y_{37} \oplus y_{35}y_{30}y_{20}y_{15}y_7 \oplus y_{51}y_{42}y_{37}y_{35}y_{30}y_{20}
\end{aligned}
$$

$$
\begin{aligned}
f(x,y) = {} & y_0 \oplus x_0 \oplus x_9 \oplus x_{14} \oplus x_{21} \oplus x_{28} \oplus x_{33} \oplus x_{37} \oplus \\
& x_{45} \oplus x_{52} \oplus x_{55} \oplus x_{50} \oplus x_{59}x_{55} \oplus x_{37}x_{33} \oplus x_{15}x_9 \oplus \\
& x_{55}x_{52}x_{45} \oplus x_{33}x_{28}x_{21} \oplus x_{59}x_{45}x_{28}x_9 \oplus x_{55}x_{52}x_{37}x_{33} \oplus \\
& x_{59}x_{55}x_{21}x_{15} \oplus x_{59}x_{55}x_{52}x_{45}x_{37} \oplus x_{33}x_{28}x_{21}x_{15}x_9 \oplus \\
& x_{52}x_{45}x_{37}x_{33}x_{28}x_{21}
\end{aligned}
$$

$$
\begin{aligned}
h(x,y,l) = {} & l_0 \oplus x_1 \oplus y_2 \oplus x_4 \oplus y_{10} \oplus x_{25} \oplus x_{31} \oplus y_{43} \oplus x_{56} \oplus \\
& y_{59} \oplus y_3x_{55} \oplus x_{46}x_{55} \oplus x_{55}y_{59} \oplus y_3x_{25}x_{46} \oplus y_3x_{46}x_{55} \oplus \\
& y_3x_{46}y_{59} \oplus l_0x_{25}x_{46}y_{59} \oplus l_0x_{25}
\end{aligned}
$$

$$
d(l) = l_0 \oplus l_3
$$

3 Intuitive Idea

To improve the throughput of Quark's hash function, we need to identify the location of the critical path in the synthesized design, i.e. the longest combinational propagation delay which determines the throughput of the system.

The longest combinational delays in U-Quark are all located within the NLFSRs in the permutation block, i.e. they are all paths starting from a flip-flop in the Y-NLFSR or the X-NLFSR, passing through the $f(x, y)$, the $g(y)$ and the $h(x, y, l)$ functions and ending on a flip-flop of the X-NLFSR or the Y-NLFSR. If these paths can be made faster, Quark's performances will improve.

To speed up the paths, in Sect. 5 we transform the Fibonacci NLFSRs of the hash function into Galois NLFSRs, while at the same time transforming the $h(x, y, l)$ function so that it has lower propagation delays compared to the original function. We use an extended version of the Fibonacci-to-Galois FSR transformation proposed in [8], which also supports external input signals, multiple outputs and efficient parallel loading. We also modify the structure of Quark (without any functional modification) due to the impossibility of retrieving the highest-grade bits of the Y register in the Galois hash function. This transformation adds a 9 cycles overhead in the number of cycles required to calculate the hash in case the initial state is not fixed and can change from run to run, but does not modify the functionality of the system and thus does not have any effect on its security properties.

4 NLFSR Preliminaries

In this paper we define as Feedback Shift Register (FSR) a register composed of a sequence of n state bits x_i (note that the notation used in this section has no relation with the notation used in Sect. 2) that has certain properties. The next value f_i of every state bit in the FSR is calculated as a function of the current FSR state (the sequence of all x_i) and, possibly, some external inputs y_i. Functions f_i are said to be *update functions*.

4.1 Fibonacci and Galois FSRs

FSRs can be categorized into Fibonacci FSRs and Galois FSRs.

In a Fibonacci FSR, all update functions have the form $f_i = x_{i-1}$ except the first, for which $f_{n-1} = g(x, y)$:

$$
\begin{aligned}
f_{n-1} &= g(x, y) \\
f_{n-2} &= x_{n-1} \\
&\ldots \\
f_1 &= x_2 \\
f_0 &= x_1
\end{aligned}
$$

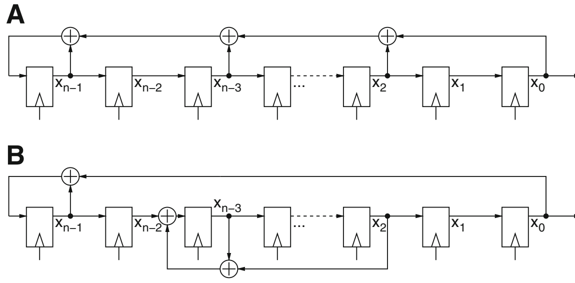

Fig. 3. (A) A Fibonacci FSR; (B) A Galois FSR

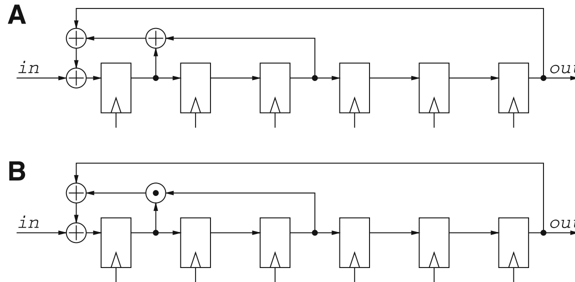

Fig. 4. (A) LFSR; (B) NLFSR

In a Galois FSR the update functions are in the form:

$$
\begin{aligned}
f_{n-1} &= g_{n-1}(x, y) \\
f_{n-2} &= x_{n-1} + g_{n-2}(x, y) \\
&\ldots \\
f_1 &= x_1 \\
f_0 &= x_0
\end{aligned}
$$

Example Fibonacci and Galois FSRs are shown in Fig. 3. Fibonacci FSRs are thus special cases of Galois FSRs.

4.2 LFSRs and NLFSRs

FSRs can be categorized into Linear FSRs (LFSRs) and Non-Linear FSRs (NLFSRs) [9].

In a linear FSR, all update functions are sums (XORs) of the state bits and the external input bits.

In a Non-Linear Feedback Shift Registers (NLFSRs), the update functions are sums of product terms, with the product terms being products (ANDs) of state bits and external input bits.

Example LFSRs and NLFSRs are shown in Fig. 4.

4.3 Equivalent FSRs

Two FSRs are said to be equivalent if the sequence of all their *output bits* are always identical [7]. Often the only output bit of an FSR is bit x_0, i.e. the last bit in the FSR chain. However, when FSRs are used in cryptographic systems such as stream ciphers or hash functions, often many of their state bits are used as output bits, i.e. inputs for external functions.

5 NLFSRs Transformation

A Galois FSR has usually better timing compared to an equivalent Fibonacci FSR, due to the fact that it has several simple update functions instead of a single, complex one [7,8,10].

There exist standard and well-known techniques to transform a Fibonacci LFSR into an equivalent Galois LFSR [7]. A transformation for NLFSRs has been proposed in [7].

5.1 Transformation Overview

The transformation in [7] considers only FSRs in which the output corresponds to the last state bit x_0 only.

The transformation in [7] consists in "moving" a set of product terms P from any update function f_i to any update function f_j with $j < i$. The transformation is restricted to product terms containing only variables x_i and no external inputs y_i. When moving a product term, the indexes of each variable x_k in each product term in P is changed to x_{k-i+j}. The transformation can be applied multiple times, i.e. it is possible to move some product terms from update function f_i to update function f_j and then move some other products from update function f_i to update function f_k.

To guarantee the equivalence of a Fibonacci NLFSR to a Galois NLFSR, it is sufficient that no product term is shifted to an update function of grade lower than the *minimum terminal bit* τ_{min}, which is calculated as:

$$\tau_{min} = \max_{p_i \in P_T} \left(max_index\,(p_i) - min_index\,(p_i) \right)$$

where P_T is the set of all product terms; $min_index(p_i)$ and $max_index(p_i)$ denote respectively the minimum and maximum index of the variables x_k in product term p_i.

Note that an "implicit" constraint when moving the products is that no product term p_i can be moved to an update function of grade lower than $n-1-min_index(p_i)$ (which would result into at least one variable having a negative index).

Proof of equivalence between the Fibonacci and the Galois FSRs can be found in [7].

Figure 5 shows a Fibonacci FSR and an equivalent Galois FSR in which one product term has been moved.

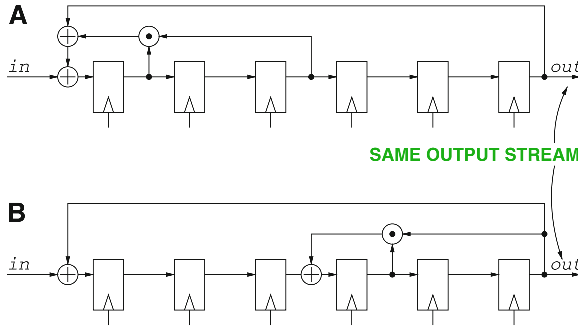

Fig. 5. (A) Original Fibonacci NLFSR; (B) Galois NLFSR

5.2 Multiple Outputs

If an FSR has multiple outputs, i.e. some of its internal state bits except the last are used as inputs for some external functions, then the transformation proposed in [7] cannot guarantee the equivalence between the original and the transformed FSRs.

However, we note that the equivalence can be guaranteed as long as no product term is moved to an update function having grade higher than Mo, where Mo denotes the grade of the highest-grade state bit x_{Mo} used as an input for an external block. In other words, the lowest-grade non-trivial update function must have a higher grade than that of the highest-grade state bit which is an output of the FSR.

Figure 6 shows a correct and an incorrect Fibonacci-to-Galois transformation.

5.3 Moving External Inputs

If an FSR receives as input some external bits y_i which are part of another FSR composed of a cascade of flip-flops, then we note that it is possible to move product terms containing a combination of x and y bits from update function i to update function j if, at the same time as the indexes of the x terms are modified, the indexes of the y_i bits are also decreased from y_i to y_{i-j}.

Figure 7 shows how a product term containing an input coming from an external FSR can be moved.

5.4 Parallel Loading

For the original and the transformed FSRs to be equivalent (having the same output stream), care must be taken to how the FSRs are loaded. If the FSRs are loaded serially, then no modification is needed. If the FSRs are loaded in parallel, then it is necessary to modify the initial value that is loaded in the Galois FSR (the initial values of the Fibonacci and Galois FSRs must be different for the output streams to be identical). This solution was described in [11]. The

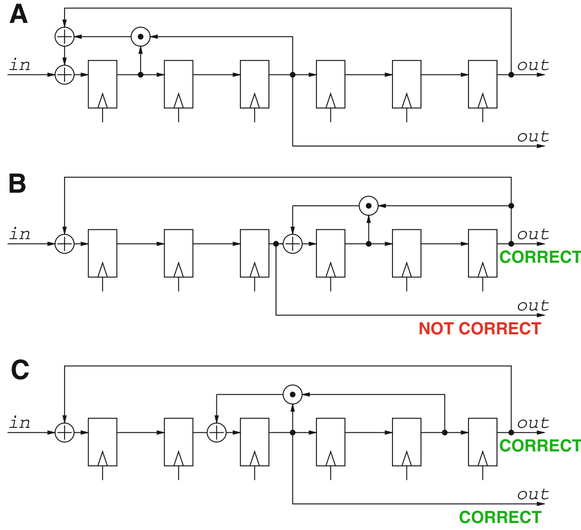

Fig. 6. (A) Original Fibonacci FSR; (B) Incorrectly transformed FSR (a non-trivial update function is righter than where the first output bit is taken); (C) Correctly transformed FSR (the last non-trivial output function is lefter than where the first output bit is taken)

computation is well-suited for fixed initialization values but is too costly in terms of hardware to do on-the-fly for values that are unknown before run time.

We here note that it is possible to modify the structure of the Galois FSR so that it can be loaded in parallel with the same value of the Fibonacci FSR, while preserving the equivalence between the FSRs. The Fibonacci FSR and the Galois FSR are loaded in parallel with the same value, but the update functions of the Galois FSR are "turned on" one by one: in the first cycle, all update functions except the highest-graded are forced to be trivial, in the second all update function except the first two are forced to be trivial, and so on until all update functions are turned on. The outputs of the two FSRs are then identical.

An example Fibonacci FSR and an equivalent parallel-loading Galois FSR are shown in Fig. 8. The enable signals for the update functions are indicated in red. The two FSRs are loaded with the same initial value. The update functions of the Galois FSR are turned on one by one. The outputs of the FSRs are identical as long as the transformation satisfies the other constraints given in this section.

5.5 Design Space Exploration

More than one Galois NLFSR equivalent to a given Fibonacci NLFSR can in general be obtained. In general, the performances will vary between the different solutions because some will have longer feedback functions while others will have shorter ones. To explore the design space and find the best solution in terms of throughput, we used a modified version of the heuristic algorithm developed

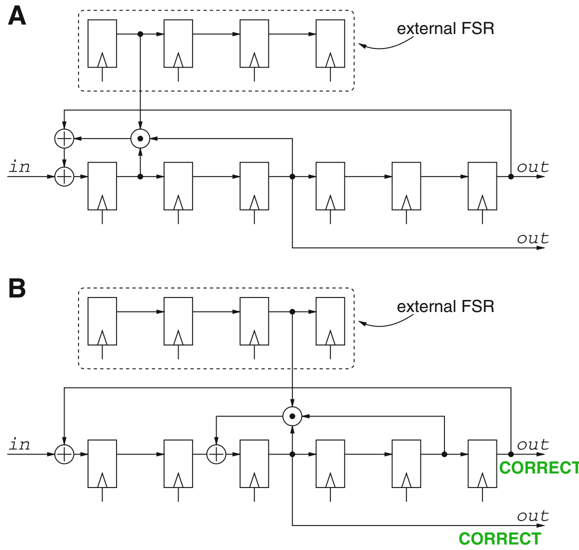

Fig. 7. FSR with external inputs: (A) Original Fibonacci FSR; (B) Transformed FSR

in [12]. The algorithm takes as input a Fibonacci FSR and a list of allowed update functions, i.e. update functions where the products can be placed. In output, the algorithm generates an efficient (high-throughput) Galois FSR. The algorithm was modified so that it supports shifting external signals along with internal ones.

The algorithm tries to place the product terms into the update functions in an efficient way, trying to minimize a *cost function*, which is an approximation of the critical path of the system.

6 U-Quark's Transformation

6.1 Structural Modification

The operation of U-Quark can be described as follows: the permutation block is initialized with the initial value and runs continuously. A counter counts continuously from 0 to 543 and then loops back to 0. When the counter value is 0 the L-LFSR state bits are resetted at the all-ones state. If the hash function is in the absorbing phase, the update functions $g_{67}, ..., g_{60}$ of the Y-NLFSR state bits $y_{67}, ..., y_{60}$ are substituted with the update functions $g^*_{67}, ..., g^*_{60}$, where the functions $g^*_{67}, ..., g^*_{60}$ are obtained by XORing the values of $g_{67}, ..., g_{60}$ with the next 8 message bits. The outputs are taken from the update functions $g_{67}, ..., g_{60}$ when the value of the counter is 0 and the message bits are finished (during the squeezing phase).

We will show that the functions on which U-Quark's Y-NLFSR product terms can be moved have grade between 67 and 59 and preliminarily we decide to use

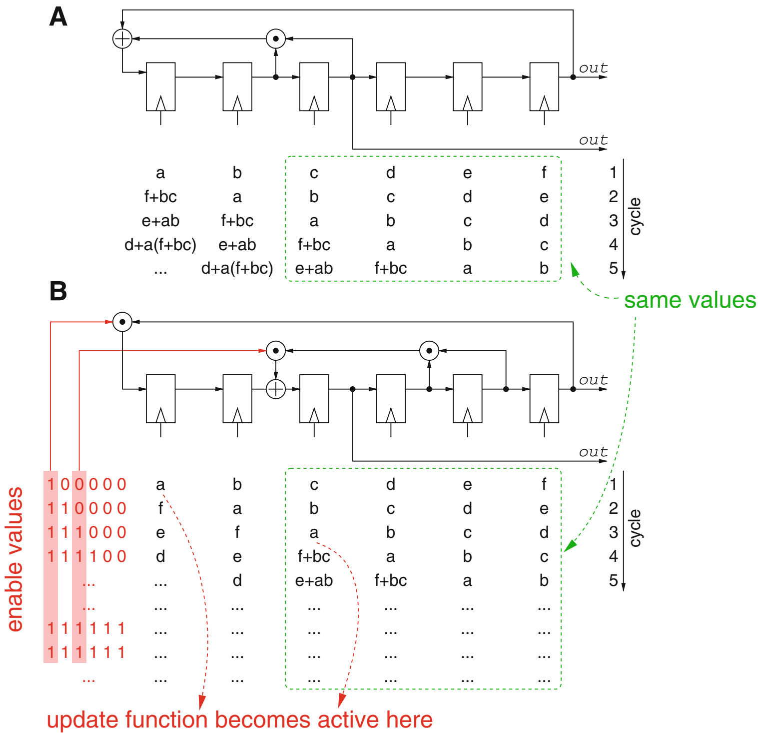

Fig. 8. (A) Original Fibonacci FSR; (B) Transformed parallel-loading FSR. In red are indicated the enable signals for the update functions. In this example, the only non-trivial update functions become active at cycles 1 and 3 respectively

all of the available feedback functions for product placements. This means that the lowest-grade non-trivial feedback function of the Y-NLFSR will be g_{59} and the original values of $g_{67}, ..., g_{60}$ are not available due to the considerations in Sect. 5 (only the state bits having grade lower than 59 match in the original and the transformed FSRs). However we observe that the values of $g_{67}, ..., g_{60}$ appear on the values $g_{58}, ..., g_{51}$ with a 9-cycle delay.

If the initialization value is programmable and can change from run to run, we transform U-Quark into the structure of Fig. 9: the hash function is initialized with the same value as for the original hash and then the system is clocked for 9 cycles turning on the feedback functions one-by-one, as discussed in Sect. 5. During the absorbing phase, when the counter value is 9, the update functions $g_{58}, ..., g_{51}$ of the Y-NLFSR state bits $y_{58}, ..., y_{51}$ are substituted with the update functions $g^*_{58}, ..., g^*_{51}$, where the functions $g^*_{58}, ..., g^*_{51}$ are obtained by XORing the values of $g_{58}, ..., g_{51}$ with the next 8 message bits. During the squeezing phase, the outputs are taken from the update functions $g_{58}, ..., g_{51}$ when the value of the counter is 9. The L-LFSR is reset to the all-ones state when the value of the counter is 0.

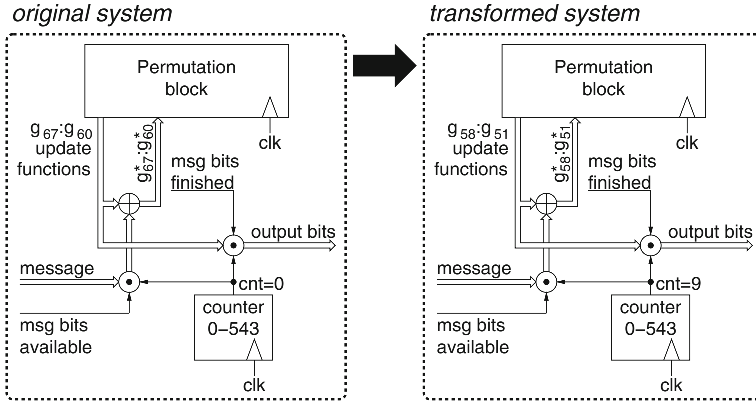

Fig. 9. (left) Original U-Quark; (right) Transformed U-Quark

If the initialization value is fixed and does not change, as is usually the case with low-weight hash functions, then we load the hash function with the initial value, load the L-LFSR with all ones, and run U-Quark offline (in a simulator) for 9 cycles, turning on the feedback functions one by one. The state of the permutation block after the 9th cycle is stored as the initial value into the hash function. The absorption of the new message bits and the squeezing of the digest bits happen both on $g_{58}, ..., g_{51}$ when the value of the counter is 0. This solution allows skipping the overhead of the block necessary to turn on the feedback functions one by one and the 9-cycles latency, since the cipher is clocked offline, in a simulator, for the first 9 cycles.

6.2 Updated X-NLFSR

For the X-NLFSR in U-Quark, the product term with the maximal difference in variable indexes is $x_9x_{28}x_{45}x_{59}$, i.e. $\tau_{min} = 50$. Product terms cannot be allocated to feedback functions x_i of grade $i < 56$ because bit x_{56} is used as an input in function $h(x, y, l)$. We decide to reserve the first three feedback functions for $h(x, y, l)$. Therefore, except for the product term x_0, which cannot be moved, we allocate all other product terms to feedback functions x_i of grade $i \leq 64$.

The following Galois NLFSR is obtained by the algorithm developed in [12] and described in Sect. 5.5 (for the values of h_{67}, h_{66} and h_{65} see Sect. 6.4):

$$
\begin{aligned}
f_{67} &= x_0 \oplus y_0 \oplus h_{67} \\
f_{66} &= x_{67} \oplus h_{66} \\
f_{65} &= x_{66} \oplus h_{65} \\
f_{64} &= x_{65} \oplus x_{30}x_{34} \oplus x_{49} \oplus x_{34} \oplus x_{42} \oplus x_{47} \\
f_{63} &= x_{64} \oplus x_{51} \oplus x_{55}x_{51}x_{17}x_{11} \oplus x_{17} \\
f_{62} &= x_{63} \oplus x_{47}x_{40}x_{32}x_{27}x_{23}x_{16}
\end{aligned}
$$

$$f_{61} = x_{62} \oplus x_{53}x_{49}x_{46}x_{39}x_{31}$$
$$f_{60} = x_{61} \oplus x_{30}x_{26}x_{45}x_{48} \oplus x_{38}x_{45}x_{48} \oplus x_{2}$$
$$f_{59} = x_{60} \oplus x_{1}x_{20}x_{37}x_{51} \oplus x_{6} \oplus x_{1}x_{7}$$
$$f_{58} = x_{59}$$
$$f_{57} = x_{58} \oplus x_{23}x_{18}x_{11} \oplus x_{49}x_{45}$$
$$f_{56} = x_{57} \oplus x_{22}$$

6.3 Updated Y-NLFSR

For the Y-NLFSR, the product term with the maximal difference in variable indexes is $y_7y_{30}y_{42}y_{58}$, i.e. $\tau_{min} = 51$. Product terms cannot be allocated to feedback functions y_i of grade $i < 59$ because bit y_{59} is used as a input in function $h(x, y, l)$. We decide to reserve the first three feedback functions for $h(x, y, l)$. Therefore, we allocate all other product terms to feedback functions x_i of grade $i \leq 64$.

The following Galois NLFSR is obtained by the algorithm developed in [12] and described in Sect. 5.5 (for the values of h_{67}, h_{66} and h_{65} see Sect. 6.4):

$$g_{67} = y_0 \oplus h_{67}$$
$$g_{66} = y_{67} \oplus h_{66}$$
$$g_{65} = y_{66} \oplus h_{65}$$
$$g_{64} = y_{65} \oplus y_{27} \oplus y_{32} \oplus y_{34} \oplus y_{39} \oplus y_{48} \oplus y_{51}$$
$$g_{63} = y_{64} \oplus y_{47}y_{38}y_{33}y_{31}y_{26}y_{16} \oplus y_{50}y_{47}y_{33}y_{31}$$
$$g_{62} = y_{63} \oplus y_{2} \oplus y_{11} \oplus y_{53}y_{49} \oplus y_{32}y_{30} \oplus y_{10}y_{2}$$
$$g_{61} = y_{62} \oplus y_{48}y_{45}y_{36} \oplus y_{29}y_{24}y_{14} \oplus y_{52}y_{36}y_{24}y_{1}$$
$$g_{60} = y_{61} \oplus y_{51}y_{47}y_{13}y_{8} \oplus y_{51}y_{47}y_{44}y_{35}y_{30} \oplus y_{28}y_{23}y_{13}y_{8}y_{0}$$
$$g_{59} = y_{60} \oplus y_{41} \oplus y_{50}y_{46}y_{43}y_{34}y_{29} \oplus y_{12}$$

6.4 Updated *h* Function

We define the $h(x, y, l)$ function as h_{67} because it is fed to state bits x_{67} and y_{67} of the two NLFSRs. The $h_{x,y,l}$ function is split into the three functions h_{67}, h_{66} and h_{65}:

$$h_{67}(x, y, l) = l_0 \oplus x_1 \oplus y_2 \oplus y_3x_{55} \oplus l_0x_{25}x_{46}y_{59} \oplus l_0x_{25}$$
$$h_{66}(x, y) = y_2x_{45}x_{54} \oplus y_2x_{44}y_{58} \oplus y_2x_{21}x_{45} \oplus x_3 \oplus y_{58}$$
$$h_{65}(x, y) = x_{44}x_{53} \oplus x_{53}y_{57} \oplus y_8 \oplus x_{23} \oplus x_{29} \oplus y_{41} \oplus x_{54}$$

7 Implementation Results

Table 1 reports the maximal operating frequency of U-Quark after applying the transformation described in Sect. 6, as well as area and power figures. The improvements over the original hash function are also reported.

Table 1. Implementation results

	Frequency (GHz)	Area (μm^2)	Power (μW)
Original	1.86	4956	14.8
Optimized	2.50	5029	12.0
Improvement	34 %	0 %	19 %

Results were obtained by designing the hash functions at Register Transfer Level (RTL) in Verilog, and then synthesizing the code for best performances using Cadence RTL Compiler for the TSMC 90 nm ASIC technology. Power results were obtained by running gate-level simulation on the obtained netlist, using random test vectors and a 1 MHz operating frequency, and using the obtained toggle values to estimate the power consumption through the synthesis tool.

The transformed U-Quark has a 34 % higher throughput compared to the original Quark and consumes 19 % less power. We think that the power improvement is due to the shorter combinational paths in the transformed hash function, which allow the synthesis tool better optimization opportunities. The area of the original and the transformed hash functions are very close. The only drawback of the transformation is therefore the 9 cycles latency in the production of the hash, in case the initial value of the cipher is not fixed and can change from run to run.

8 Conclusion

In conclusion, we have shown that it is possible to considerably improve the hardware timing figures of the Quark hash function by applying a Fibonacci-to-Galois transformation of its feedback shift registers. We have extended the NLFSR Fibonacci-to-Galois transformation described in [7] so that it can support U-Quark's FSRs.

We could obtain a 34 % better throughput and a 19 % lower power consumption, without any area overhead. The transformation is very easy to apply because it only requires a modification of the feedback functions at RTL. If the initial state of the hash function is programmable and can change from run to run, a 9 cycles latency in the production of the hash is inserted. If the initial state is fixed, then no additional latency is inserted.

Acknowledgment. This work was supported in part the research grant No 621-2010-4388 from the Swedish Research Council and in part by the research grant No SM12-0005 from the Swedish Foundation for Strategic Research.

References

1. Aumasson, J.-P., Henzen, L., Meier, W., Naya-Plasencia, M.: QUARK: a lightweight hash. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 1–15. Springer, Heidelberg (2010)

2. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferentiability of the sponge construction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008)
3. Hell, M., Johansson, T., Maximov, A., Meier, W.: The grain family of stream ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 179–190. Springer, Heidelberg (2008)
4. Hell, M., Johansson, T., Maximov, A., Meier, W.: A stream cipher proposal: Grain-128. In: 2006 IEEE International Symposium on Information Theory, pp. 1614–1618, July 2006
5. Agren, M., Hell, M., Johansson, T., Meier, W.: Grain-128a: a new version of Grain-128 with optional authentication. Int. J. Wire. Mob. Comput. **5**, 48–59 (2011)
6. De Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN — a family of small and efficient hardware-oriented block ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 272–288. Springer, Heidelberg (2009)
7. Dubrova, E.: A transformation from the Fibonacci to the Galois NLFSRs. IEEE Trans. Inf. Theory **55**(11), 5263–5271 (2009)
8. Mansouri, S.S., Dubrova, E.: An improved hardware implementation of the Grain-128a stream cipher. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 278–292. Springer, Heidelberg (2013)
9. Golomb, S.: Shift Register Sequences. Aegean Park Press, Laguna Hills (1982)
10. Mansouri, S., Dubrova, E.: An improved hardware implementation of the Grain stream cipher. In: 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), pp. 433–440, September 2010
11. Dubrova, E.: Finding matching initial states for equivalent NLFSRs in the Fibonacci to the Galois configurations. IEEE Trans. Inf. Theory **56**(6), 2961–2967 (2010)
12. Chabloz, J.-M., Mansouri, S.S., Dubrova, E.: An algorithm for constructing a fastest Galois NLFSR generating a given sequence. In: Carlet, C., Pott, A. (eds.) SETA 2010. LNCS, vol. 6338, pp. 41–54. Springer, Heidelberg (2010)