Embedded Syndrome-Based Hashing

Ingo von Maurich and Tim Güneysu*

Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
{ingo.vonmaurich,tim.gueneysu}@rub.de

Abstract. We present novel implementations of the syndrome-based hash function RFSB on an Atmel ATxmega128A1 microcontroller and a low-cost Xilinx Spartan-6 FPGA. We explore several trade-offs between speed and area/code size on both platforms and show that RFSB is extremely versatile with applications ranging from lightweight to high performance. Our lightweight microcontroller implementation requires just 732 byte of ROM while still achieving a competitive performance with respect to other established hash functions. Our fastest FPGA implementation is based on embedded block memories available in Xilinx Spartan-6 devices and runs at 0.21 cycles/byte, with a throughput of 5.35 Gbit/s. To the best of our knowledge, this is the first time the RFSB hash function is implemented on either of these wide-spread platforms.

Keywords: RFSB, hash function, code-based cryptography, microcontroller, hardware, FPGA.

1 Introduction

Cryptographic hash functions are used in a broad range of applications where mapping an arbitrary amount of data to a fixed length bit string is required. Examples are digital signatures, messages authentication codes, data integrity checks, and password protection. Prominent and widely deployed hash functions such as MD5 [38], SHA-1 [37], and the SHA-2 family [37] are used in various products and implementations whose security depends on the collision resistance of those hash functions. However, over the last years (chosen-prefix) collision attacks have been published for MD5 [42] [43] and SHA-1 [30] and are already exploited in the real-world. Recently, a major attack based on MD5 collisions was performed by the Flame espionage malware which injects itself into the Microsoft Windows operating system. The malware code is signed by a rogue Microsoft certificate and disguises itself as a Microsoft Windows update. The rogue certificate was obtained using a previously unknown chosen-prefix collision attack on a Microsoft Terminal Server Licensing Service certificate which still used the MD5 algorithm [34].

* This work was partially supported by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

Although the SHA-2 family withstands these attacks so far, its similar structure to SHA-1 raised concerns about its long term security. Therefore, the National Institute of Standards and Technology (NIST) announced the public SHA-3 competition in the end of 2007 [36]. A total of 64 candidates entered the competition, of which 14 advanced to the second round, and the last round still has 5 competing candidates. Apart from their security the main selection criteria are hardware and software speed as well as scalability. The SHA-3 competition announcement explicitly demands efficiency in 8-bit microcontrollers and in hardware.

Embedded 8-bit microcontrollers are a common representative of low-cost and energy efficient computation units used in automotive applications, digital signature smart cards, wireless sensor networks and many more. Field-Programmable Gate Arrays (FPGA) on the other hand allow reconfigurable implementations of cryptography in hardware, usually yielding a much better performance than achievable with 8-bit microcontrollers or PCs. FPGA device classes range from low-cost (e.g., Xilinx Spartan family) to high-end high-speed (e.g., Xilinx Virtex family). Since both microcontrollers and FPGAs are used for applications handling sensitive data, efficiently computable cryptographic primitives are essential for successful real-world applications.

Code-based cryptography offers a variety of cryptographic primitives that are built upon the hardness of well-known NP-complete problems in coding theory. The Fast Syndrome Based (FSB) hash function is a code-based hash function that entered the SHA-3 competition but due to its inefficiency compared to other candidates, FSB did not advance to the second round. The Really Fast Syndrome-Based (RFSB) hash function is an improved version of FSB that aims to overcome this problem.

Other cryptographic primitives based on codes are the McEliece [31], the Niederreiter [35] or the Hybrid McEliece (HyMES) [11] asymmetric encryption scheme. Digital signatures based on McEliece can be computed using CFS [14], Parallel-CFS [17], or quasi-dyadic CFS [5] and even one-time signatures are possible using the BMS-OTS scheme [6]. Code-based stream ciphers such as SYND [20] and 2SC [32] also offer security reductions to the syndrome decoding problem.

McEliece and Niederreiter have been reported to be efficiently implementable in 8-bit microcontrollers [12][16][25][26] as well as in reconfigurable hardware [24] [27][41]. Software and hardware implementations of the code-based signature scheme CFS have been published as well [10][29].

1.1 Contribution

With code-based encryption and signature schemes proven to be feasible in hard- and software, it still is an open question how code-based hash functions perform on these platforms. In this paper we set out to answer this question by evaluating the feasibility and achievable performance of RFSB-509 in embedded systems. We explore different design choices for embedded microcontrollers and reconfigurable hardware using the wide-spread 8-bit microcontroller Atmel

ATxmega128A1 and a Xilinx Spartan-6 FPGA. We show that RFSB-509 can be efficiently implemented on both platforms and that RFSB can, in contrast to its predecessor FSB, keep up with current SHA-3 candidates and hash standards. Source code for both platforms is made publicly available in order to allow other researchers to use and evaluate our implementations[1]. To the best of our knowledge this is the first work of its kind.

1.2 Organization

This paper is organized as follows. We present related work on code-based hash functions and the history that led to the development of RFSB in Section 2. After shortly recalling general specifications of the RFSB hash function, we detail on the concrete parameter proposal RFSB-509 and give an implementors' view on RFSB-509 in Section 3. Next, design considerations for implementations on embedded microcontrollers and on reconfigurable hardware are evaluated in Section 4 and Section 5. We present our result in Section 6 before we draw a conclusion in Section 7.

2 Related Work on Code-based Hash Functions

Augot, Finiasz, Gaborit, Manuel, and Sendrier entered the SHA-3 competition with the Fast Syndrome Based (FSB) hash function [2] that relies on the syndrome decoding problem for linear codes. Previous attempts to build such a hash function by Augot, Finiasz, and Sendrier [3][4], and Finiasz, Gaborit, and Sendrier [18] turned out to be flawed and were consequently broken by Coron and Joux [13], Saarinen [40], and Fouque and Leurent [19]. Hence, the FSB parameters were adjusted to withstand these attacks for the SHA-3 submission and to date this parameter set remains unbroken. However, FSB did not advance to the second round of the SHA-3 competition mainly because of its lack in efficiency compared to other submissions.

Meziani, Dagdelen, Cayrel, and Yousfi Alaoui use the ideas of FSB and combine them with a sponge construction instead of the Merkle-Damgård principle to construct the S-FSB hash function [33]. Their main goal is to improve the performance compared to FSB and they report a C implementation of S-FSB-256 on an Intel Core 2 Duo CPU running at 183 cycles/byte. Compared to FSB requiring 264 cycles/byte on the same CPU, S-FSB is about 30% faster but when looking at the overall picture S-FSB is still an order of magnitude slower than the current hash standard SHA-256 which runs at 15.49 cycles/byte on a similar CPU according to eBASH[2] [8].

Bernstein, Lange, Peters, and Schwabe introduce the Really Fast Syndrome-Based (RFSB) hash function as an improved version of FSB and propose concrete parameters (RFSB-509) in [9]. The authors report an implementation of

[1] See our web page at `http://www.sha.rub.de/research/projects/code/`

[2] (6fd); 2007 Intel Core 2 Duo E4600; 2 x 2400MHz; cobra, supercop-20111120

RFSB-509 that outperforms the current hash standard SHA-256 on Intel Core 2 Quad Q9550 CPUs at 13.62 vs. 15.26 cycles/byte. According to the latest measurements on eBASH[3], new implementations allow to compute RFSB-509 even faster at 10.64 cycles/byte while SHA-256 remains at 15.31 cycles/byte on the same CPU.

Furthermore, a PC implementation of RFSB in Java and C is reported by Rothamel and Weiel [39]. In addition to RFSB-509, the authors suggest parameter sets RFSB-227, RFSB-379, and RFSB-1019 and provide performance measurements for all four sets. However, their results do not come anywhere close to the speeds reported in the original RFSB paper (e.g., they report RFSB-509 to run at 120.5 cycles/byte on an Intel i7 CPU).

3 The RFSB Hash Function

The RFSB [9] hash function is constructed very similar to the FSB hash function [2], both are designed to be used inside a collision resistant hash function. A fixed length compression function is combined with the Merkle-Damgård domain extender [15] to enable processing of an arbitrary amount of data. An initialization vector (IV) is compressed together with the first message block. The output is used as chaining value together with the second message block and is again fed into the compression function. This continues until the second to last message block has been processed. The last block is padded by appending a single 1 bit followed by sufficiently many zeros and a 64-bit message length counter. After all input has been processed a final output filter (called final compression function in FSB terms) is applied. In case of FSB Whirlpool is used as final compression function, the authors of RFSB suggests to use SHA-256 or an AES-based output filter. The basic hashing principle of FSB and RFSB is illustrated in Figure 1.

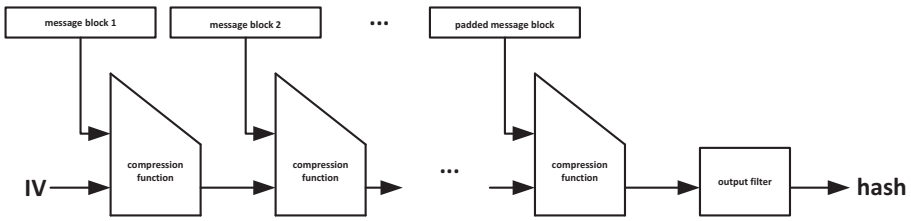

Fig. 1. Illustration of the hashing principle based on the Merkle-Damgård domain extender used by FSB and RFSB. The initialization vector (IV) is set to zero in RFSB and the output filter is defined to be SHA-256.

[3] (10677); 2008 Intel Core 2 Quad Q9550; 4 x 2833MHz; berlekamp, supercop-20120704

3.1 The RFSB Compression Function

The RFSB compression function is defined by four parameters: an odd prime r, positive integers b and w, and a compressed matrix of size $2^b \times r$-bit. The compression function takes a bw-bit string as input which is interpreted as a sequence of $\lceil bw/8 \rceil$ bytes $(m_1, m_2, \ldots, m_w)$ where each $m_i \in \{0, 1, \ldots, 2^b - 1\}$. The output is a r-bit string that is interpreted as a sequence of $\lceil r/8 \rceil$ bytes. Both input and output are interpreted in little-endian format. The compressed matrix consists of constants $c[0], c[1], \ldots, c\left[2^b - 1\right]$ where each constant has a length of r-bit. The uncompressed RFSB matrix is derived from these constants by defining

$$c_i[j] = c[j]\, x^{128(w-i)}, 1 \leq i \leq w,\ 0 \leq j \leq 2^b - 1$$

in the ring $\mathbb{F}_2[x]/(x^r - 1)$ which essentially are rotations of the compressed matrix constants.

The input is mapped to the output using the message bytes m_i as indices of the uncompress matrix constants c_i. The constants are summed up by exclusive-or addition to form the output as follows:

$$(m_1, m_2, \ldots, m_w) \mapsto c_1[m_1] \oplus c_2[m_2] \oplus \cdots \oplus c_w[m_w].$$

When using the compressed matrix notation the mapping from input to output is given by:

$$(m_1, m_2, \ldots, m_w) \mapsto c[m_1]\, x^{128(w-1)} \oplus c[m_2]\, x^{128(w-2)} \oplus \cdots \oplus c[m_w]$$

in $\mathbb{F}_2[x]/(x^r - 1)$.

3.2 A Concrete Proposal: RFSB-509

RFSB-509 is a concrete parameter proposal by the designers of RFSB which achieves good software speed. In the original paper RFSB-509 is assumed to provide a collision resistance of more than 2^{128}. The proposed parameters are $r = 509, b = 8$, and $w = 112$. Hence, the RFSB-509 input message size is 896 bit (112 byte) and the output size is 509 bit. The compressed matrix is of size $2^b \times r = 256 \times 509$ bit which roughly amounts to 16 kByte. A recent result by Kirchner [28] suggests an improved generalized birthday attack which claims to lower the complexity to about 2^{79}. Hence, the parameters need to be adjusted if a collision resistance of 128-bit is required.

Each element of the compressed matrix is generated using a concatenation of the ciphertexts output by four AES-128 calls with fixed all-zero key and a plaintext which is a function of the index of the constant. We denote the AES calls by $y = \mathrm{AES}_k(x)$ with y being the 16-byte ciphertext, k being the 16-byte key, and x being the 16-byte plaintext. The plaintext is set to zero except for the last two bytes. The second to last byte is set to j which is the index of the constant and $0 \leq j \leq 255$. The last byte of the plaintext is a counter which

increases with each AES-128 call from 0 to 3. In total this results in a 512-bit string

$$c'[j] = \mathrm{AES}_0(0,\ldots,0,j,0)\,||\mathrm{AES}_0(0,\ldots,0,j,1)\,||\ldots||\mathrm{AES}_0(0,\ldots,0,j,3)$$

which is reduced to

$$c[j] = c'[j] \mod x^{509}-1$$

to stay in the ring $\mathbb{F}_2[x]/(x^{509}-1)$. The 112-byte input block $(m_1, m_2, \ldots, m_{112})$ with each $m_i \in \{0,1,\ldots,255\}$ is mapped to the 509-bit output by computing

$$(m_1, m_2, \ldots, m_{112}) \mapsto c[m_1]\,x^{128(112-1)} \oplus c[m_2]\,x^{128(112-2)} \oplus \cdots \oplus c[m_{112}]$$

in $\mathbb{F}_2[x]/(x^{509}-1)$.

3.3 RFSB-509 from an Implementors' Point-of-View

When designing RFSB-509 for embedded systems a few aspects have to be considered beforehand. In the following we detail considerations and optimizations for implementations of RFSB-509 in embedded devices.

At first, the constant matrix, although compressed, still has a size of 16 kByte which poses a challenge in embedded systems where memory usually is scarce. Due to the computability of the constants there are basically two choices that can be made. Either memory is spent to store the table or each constant is, probably multiple times, generated on-the-fly when needed. For the on-the-fly generation one has to keep in mind that each constant requires four calls to the AES-128 encryption function, thus a total of $4 \times 112 = 448$ AES encryptions are required during one compression.

When compressing a message block the rotations applied to each constant depend on the position of the current message byte. For example the first computation in RFSB-509 is $c[m_1]\,x^{128(112-1)} = c[m_1]\,x^{14208}$ which requires to rotate $c[m_1]$ by 14208 bit positions. When using 512-bit wide registers the amount of different rotations performed during RFSB compression can be reduced to just four since $128(112-i) \equiv 128i \mod 512 \in \{0, 128, 256, 384\}$. Using this the RFSB compression of the first four messages bytes (m_1, m_2, m_3, m_4) can be rewritten as

$$s_1 = \mathrm{ROL}_{384}(c[m_1]) \oplus \mathrm{ROL}_{256}(c[m_2]) \oplus \mathrm{ROL}_{128}(c[m_3]) \oplus c[m_4]$$

where ROL_j denotes a j-bit rotation to the left (towards the most significant bit) of a 512-bit register. The four different rotations and their exclusive-or sum can be seen as *basic compression unit* of RFSB-509, which can be generalized to

$$s_i = \mathrm{ROL}_{384}(c[m_{4i+1}]) \oplus \mathrm{ROL}_{256}(c[m_{4i+2}]) \oplus \mathrm{ROL}_{128}(c[m_{4i+3}]) \oplus c[m_{4i+4}]\,.$$

In order to process all 112 input message bytes this computation has to be repeated 28 times. Adding up all intermediate results s_i then yields the output

of the compression function

$$\begin{aligned}\text{compress}_{509}\left(m_1,\ldots,m_{112}\right) &= \sum_{i=0}^{27} s_i \mod x^{509}-1 \\ &= \sum_{i=0}^{27}\sum_{j=1}^{4} \text{ROL}_{512-128j}\left(c\left[m_{4i+j}\right]\right) \mod x^{509}-1\end{aligned}$$

where the sums are formed using exclusive-or addition. Figure 2 illustrates the tree-like structure of the RFSB-509 compression function and shows the repetitions of the *basic compression unit*.

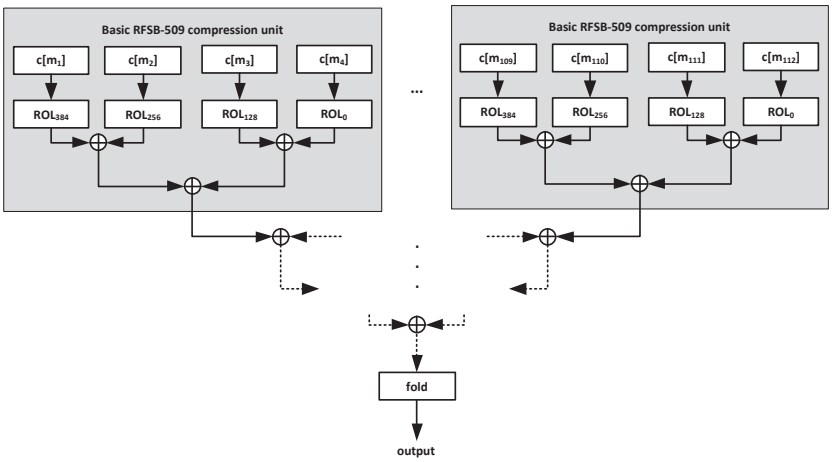

Fig. 2. The basic compression unit in RFSB consists of looking up four constants, rotating them according to their position by either 384, 256, 128, or 0 bits and xoring the result. The *fold* unit represents the reduction modulo $x^{509}-1$ by folding the three most significant bits onto the three least significant bits.

One further important detail is the computation of the reduction modulo $x^{509}-1$ for 512-bit registers. It is achieved by folding the three most significant bits onto the three least significant bits and setting the three most significant bits to zero. Such a reduction does not pose a problem on both platforms and can be realized at minimal cost.

4 Designing RFSB-509 for Embedded Microcontrollers

For our evaluations of RFSB-509 on an embedded microcontroller we use the wide-spread 8-bit ATxmega microcontroller family from Atmel. These microcontrollers are low-cost yet powerful enough for a wide range of cryptographic and

non-cryptographic applications. Apart from the usual features available in this kind of devices (analog to digital converter, digital to analog converter, timers, counters, several communication interfaces, etc.) the ATxmega offers dedicated hardware accelerators for the encryption standards DES and AES-128.

All following designs are split into three basic functions *init*, *update*, and *final*. During *init* we reset the internal state, output and counter to zero. The *update* function implements the Merkle-Damgård domain extender, processes new message blocks and updates the internal state accordingly until the last message block is reached. The last message block is processed by the finalization function which pads the message, appends the length counter, compresses the message and sets the output when finished.

When designing RFSB-509 for embedded microcontrollers there are basically two different ways of realizing the RFSB compression function. Either the constants are stored in a table or the constants are generated on-the-fly when needed. One can also think of a hybrid mode, where the constants are not stored in the program memory but are generated and stored in volatile SRAM when starting the device. We explore all three possibilities an give details about the design approach for each version in the following. The AES- and ROM-based implementations are done on an Atmel ATxmega128A1 microcontroller while the SRAM-based implementation is done on an Atmel ATxmega384C3 microcontroller.

4.1 On-the-fly Constant Generation

When designing a small memory footprint version of RFSB-509, on-the-fly constant generation is required since the compressed constant table would consume 16 kByte of program memory which would render a lightweight implementation impossible. Especially the hardware AES-128 offered in ATxmega devices is useful in such a setting. The AES-128 crypto module runs concurrently to the CPU and takes 375 clock cycles after loading the key and the plaintext block into the module to en-/decrypt a 128-bit block. When taking loading and storing of key, plaintext and ciphertext into account, an AES-128 encryption takes about 500 clock cycles or 31.25 cycles/byte. Thus when running at its maximum frequency of 32 MHz the ATxmega is able to achieve a AES-128 encryption throughput of about 8 Mbit/s.

Our small footprint implementation of the RFSB-509 hash function is built around a parameterizable constant generation function that is capable of providing rotation widths of 0, 128, 256, and 384 bit. In each iteration the constant generation function computes four AES encryptions. After each encryption the ciphertext is transferred into 16 general purpose registers and immediately afterwards the next plaintext and key (which is the all-zero key for all encryptions but has to be reloaded before every encryption nevertheless) are loaded into the AES module and the next encryption is triggered. While waiting for the next encryption to finish, we concurrently process the previous ciphertext by accumulating it to the output and reducing the computed constant modulo $x^{509} - 1$. Thus, we make use of otherwise wasted cycles while the next encryption is running in

parallel. In order to maintain a reasonable performance, parts of the implementation are unrolled, e.g., storing and loading data to and from the AES crypto module is unrolled since this part is critical for the overall runtime.

If the constants would be generated using DES instead of AES-128, the performance of the on-the-fly constant generation could be further improved. Since the output length of DES is half the output length of AES-128, twice the amount of DES calls would be required. However, at 16 cycles per DES encryption after loading the key and plaintext to the corresponding registers, this would still be an order of magnitude faster than AES-128 encryption on an ATxmega microcontroller. Since the performance of the encryption function is the limiting factor in such an implementation, the overall performance would greatly benefit from such an improvement.

Note, the short key length of DES and its vulnerability to brute-force attacks does not pose a thread to the security of RFSB-509 since all plaintexts and keys are already known by definition. As stated in the original RFSB paper: "The full security of AES is certainly not required for RFSB: all that we need is a function generating a few elements of $\mathbb{F}_2[x]/(x^r - 1)$ without any obvious linear structure" [9].

4.2 ROM-Based Lookup Table

A total of 16 kByte of program memory is required when storing the precomputed constant table in the ROM of the microcontroller. Each of the 256 entries in the table consists of 64 byte, thus we multiply each message byte by 2^6 (shifting the message byte six times to the left) to compute the index of the required constant. Instead of first reading out the constant and then rotating it according to the position of the current message byte, we adjust the table pointer beforehand to directly read out the rotated constant. This is possible since all rotation widths are a multiple of 8 and the basic addressable unit in our 8-bit microcontroller is a byte. For example if a constant has to be rotated by 384 bit, we add $\frac{384}{8} = 48$ to the current index, read out 16 constant bytes, then subtract 64 from the current index and read out the remaining 48 constant bytes. Thus we achieve nearly free rotations by only adjusting the table pointer.

This process is repeated for all message bytes and rotation widths, and after all constants have been read out and accumulated, the result is reduced modulo $x^{509} - 1$.

In our evaluation we explore two different approaches, a rolled and an unrolled version. In the unrolled version we remove every loop inside the computation of the basic compression unit which computes the intermediate output of four consecutive message bytes with four different rotation widths applied to the read out constants (cf. Figure 2).

4.3 RAM-Based Lookup Table

In order to estimate the maximum achievable performance in embedded microcontrollers, we move the constants from the program memory into the faster

SRAM. Accessing a byte in the program memory of the ATxmega takes 3 clock cycles while accessing the internal SRAM takes 2 clock cycles. Since $112 \times 64 = 7168$ byte have to be looked up when hashing one message block, this small difference can have a larger impact on the overall runtime than one might expect on first sight. The hashing itself is constructed similar to the previously described setup, with some minor adjustments taking care of the modified memory locations.

For this evaluation we use the Atmel ATxmega384C3 microcontroller since it offers 32 kByte of SRAM. Devices offering 8 or 16 kByte of SRAM do not suffice in this scenario since in addition to the constant table also the current state and the next message block have to be held in the SRAM.

A remaining question is how to place the RFSB-509 constants into the SRAM. Since SRAM is volatile memory, its content has to be reloaded after every power cycle. As designers we are left with two choices. Either we store the constants in the program memory as done in Section 4.2 and copy them into SRAM at every power up, or we generate the constants once at every power up and directly store them in the SRAM. The decision which of the proposed methods to use basically depends on two factors. Firstly, it has to be considered how much time is available after a power cycle before the hash function has to be used for the first time. Generating the constants takes longer then just copying them from program memory to SRAM. Secondly, it depends on the available program memory. The generation function takes up less space in program memory compared to a 16 kByte table. In our implementation, we generate the constants after each power up, thus avoiding redundant tables in RAM and ROM.

Again we explore two approaches: a rolled and an unrolled version similar to the previously described ROM-table design.

5 Designing RFSB-509 for Reconfigurable Hardware

For our evaluation of RFSB-509 in reconfigurable hardware we use the low-cost Xilinx Spartan-6 device family. Spartan-6 devices are powerful, up-to-date FPGAs offering hundreds to (ten-)thousands of slices, where each slice contains four 6-input lookup tables (LUT), eight flip-flops (FF), and surrounding logic. In addition to the general purpose logic, embedded resources such as block memories (BRAM) and digital signal processors (DSP) are available. Yet Spartan-6 devices are about an order of magnitude cheaper than Xilinxs' high-performance devices families Virtex-5 and Virtex-6.

For the design of RFSB-509 in reconfigurable hardware, we again follow two different strategies of implementing the lookup of compressed matrix constants. In one architecture we generate the constants when needed using on-the-fly AES computations, in the other architectures we make use of the available block memories to store the matrix constants.

Since different choices for the constant lookups only affect the compression function of RFSB, all implementations share the same top-level component that

takes care of handling the input and output of data through FIFOs and passes the data and control signals to the Merkle-Damgård construction which is also the same for all hardware implementations. Thus we design a modular system in which the compression function can be easily exchanged. We detail on the different compression function designs in the following.

5.1 Implementing RFSB-509 Using Embedded Block Memories

Spartan-6 FPGAs feature dual-ported block memories (BRAM) each capable of storing up to 18 Kbit of data. They can be configured to represent one of five different memory types. For our purpose we choose to configure the BRAMs as dual-port read-only memory (ROM) since we do not need the write capability. In each clock cycle two separate values can be read from two different memory addresses because of the BRAMs' dual-port layout.

Minimal BRAM Consumption Since the compressed matrix constant table has a size of about 15.9 Kbyte a theoretical minimum of $\frac{15.9 \cdot 8}{18} = 7.07$ BRAMs is required to store the full table. However, a wide-access port of 509 bit for each constant is not natively supported by the BRAM primitives. The maximum natively supported width is 32 bit (36 when using the parity bits) or 64 bit (72 when using the parity bits) when combining both ports. Thus, for achieving a minimal block memory usage, we use eight BRAMs to store the constants as shown in Figure 3.

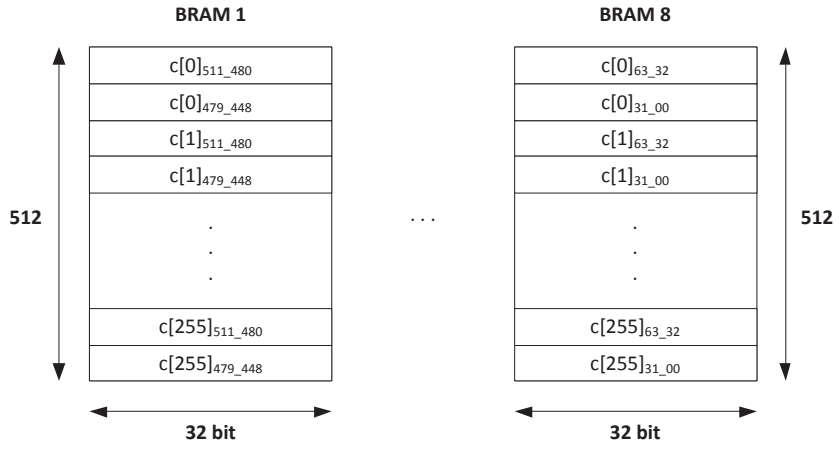

Fig. 3. Our smallest table based FPGA implementation of RFSB-509 requires 8 block memories configured as 512×32 bit dual-port ROM. Every BRAM holds a 64-bit chunk of the 509-bit constants (prepended by three zero bits) split into two 32-bit parts. Since two memory slots of each BRAM can be read out in one clock cycle, one constant can be read out in one clock cycle.

We configure the BRAMs to store 512 values of 32 bit each, which is natively supported. The RFSB constants are divided into eight 64-bit chunks and are distributed to the BRAMs. The 64-bit chunks are again split and stored in two consecutive memory slots. Hence, BRAM_1 holds the topmost 64-bit of all 256 RFSB constants, BRAM_2 the following 64-bit of all RFSB constants and so forth.

The index into the table is the current message byte m_i appended by a zero and a one bit to address both memory slots. Because of the dual-port layout of the block memories, both 32-bit memory slots can be read out simultaneously. This is done for all block memories at the same time and the results are concatenated and rotated to form the demanded constant $\text{ROL}_x\left(c\left[m_i\right]\right)$. Because of this set-up, a complete already rotated RFSB-509 constant is available in one clock cycle.

We sequentially iterate over all input message bytes, accumulate the read out constants and reduce the result after all message byte have been processed.

Due to its tree-like structure, RFSB allows for very scalable designs which allow to process multiple message bytes in one clock cycle since the inputs to the block memories are independent of each other. In the following we make different proposals of how to implement multiple constant lookups in one clock cycle.

Wide-Access Block Memories Our next architecture uses block memories with wide-access ports to provide the matrix constants. Creating a 256×509-bit table using the Xilinx block memory generator results in 15 occupied 18 Kbit BRAMs. With this architecture it is possible to read out two RFSB-509 constants in one clock cycle, thus reducing the necessary cycles spent for table lookups by a factor of 2 to 56 cycles.

The internal compression module now handles two bytes at once and applies two different rotations to the read out constants depending on the position of the message byte in the input string. In the first mode, one constant is rotated by 384-bit, the second by 256-bit, in the second mode the first constant is rotated by 128-bit and the second is not rotated. Both constants are accumulated to the intermediate result, the rotation mode is switched with every input message pair and after the complete input block has been processed, the result is reduced modulo $x^{509} - 1$.

Multiple Table Instances For high-performance applications we explore architectures in which we instantiate multiple of the aforementioned wide-access block memories containing the full constant table. We go only so far that we still stay within reasonable (i.e., realizable on current Spartan-6 devices) resource boundaries.

In the first setting we use two tables which allows to process four message bytes in one clock cycle, essentially representing the basic compression unit introduced in Figure 2. Furthermore, it is now possible to hardwire the rotations applied to the constants because the output of each of the block memory ports

only handles either $c[m_{4i+1}]$, $c[m_{4i+2}]$, $c[m_{4i+3}]$, or $c[m_{4i+4}]$, $0 \leq i \leq 27$. The two tables require 29 block memories and again halve the required clock cycles to 28 for the constant lookups for one 112-byte input block.

In a second design we use four separate instances of the constant table, which requires 58 BRAMs. It enables us to look up eight message bytes per clock cycle and finish the lookups after 14 clock cycles.

The third design quadruples the amount of block memories to be able to hold eight parallel instances of the constant table. This requires 116 BRAMs and allows to lookup 16 constant at the same time which means all constants are retrieved in just 7 clock cycles.

5.2 Implementing RFSB-509 Using an AES Core

To cover all possible designs, we also include on-the-fly generation of the matrix constants using an hardware AES core. Since the key is always fixed to the all-zero key, the key-schedule does not have to be implemented as the round-keys can be precomputed. This of course is only true if the AES core is not used by other applications that require the key to be adjustable during runtime. The AES core in use is a T-table based implementation that occupies eight block memories for storing the tables.

The constant computation unit is built straightforward. It receives a message byte and starts four consecutive encryptions with the respective input blocks as described in Section 3.2. Each result is xored to an internal output signal and after the fourth encryption is finished, a modular reduction is performed and the constant is output. The higher level unit receives the constant, rotates it according to the position of the current message byte and passes the next message byte to the constant computation unit.

6 Results

All our implementations are verified against testvectors generated using the reference implementation of RFSB-509 by Schwabe which was submitted to the ECRYPT Benchmarking of Cryptographic Systems (eBACS) [7].

The results for embedded microcontrollers are provided by the Atmel AVR Studio 6, and the implementations are additionally tested in real hardware using an AVR XPLAIN board equipped with an ATxmega128A1. In addition, the microcontroller implementations feature a full padding unit.

The FPGA results are achieved using post place-and-route results from Xilinx ISE Design Suite 14.1. As target device we use a Spartan-6 FPGA XC6SLX100, but we stress that for all implementations smaller Spartan-6 FPGAs suffice.

The output filter is currently not implemented because a wide range of SHA-256 implementation is already available in hard- and software. Neglecting the output filter arguably does not effect the performance measurements when hashing long messages since it is only applied once to the output of the RFSB-509 compression function.

In the following we present our microcontroller and FPGA results and compare them to other hash function implementations on similar devices.

6.1 Embedded Microcontrollers

Table 1 shows the results of our implementations of RFSB-509 on the embedded microcontroller ATxmega. The achievable performance is measured in cycles/byte, where the amount of clock cycles required for calling the *update* function is divided by 48 byte although in total 112 byte are hashed. This is due to the fact, that only 48 fresh message bytes enter each compression function because of the Merkle-Damgård construction. Thus, these figures give a realistic performance overview when hashing long messages.

Table 1. Results of RFSB-509 achieved on the embedded microcontroller Atmel ATxmega128A1. *Results for the SRAM table based implementations are measured on an ATxmega384C3.

Design	ROM [byte]	RAM [byte]	Cycles/ Byte	Used ROM	Used RAM
HW-AES	732	232	4753.1	0.5%	2.8%
ROM table	602+16384	232	1573.9	12.2%	2.8%
ROM table unrolled	3100+16384	232	1114.9	14.0%	2.8%
RAM table*	996	232+16384	1424.5	4.2%	50.7%
RAM table unrolled*	3494	232+16384	965.6	4.9%	50.7%

All implementations require 232 byte of RAM, split into 112-byte state, 48-byte input, 64-byte output and an 8-byte counter. An additional 16 kByte of SRAM are used by the SRAM-based table implementations to store the table.

The fastest implementation is running at 965.6 cycles/byte but is so far only realizable in a few 8-bit microcontrollers since only newer devices meet the RAM requirements. The fastest ROM-based implementation computes one RFSB-509 round at 1114.9 cycles/byte. The counterpart to the unrolled version does not seem to be a good choice, since program memory at this size is not a problem for current microcontrollers and spending an additional 2.5 Kbyte of ROM seems to be worth the 460 cycles/byte performance improvement.

Our smallest implementation, the one based on AES encryptions, only requires 732 byte of ROM which falls into the lightweight cryptography category. If ROM memory is scarce, the current version could be implemented even smaller since some loops have been unrolled to improve the performance. Since for every constant the AES encryption is called four times, 448 AES encryptions are needed during compression. When assuming 500 clock cycles for each AES encryption we get a lower bound of 224000 clock cycles or 4666.7 cycles/byte for the encryptions, not counting rotations, modular reductions and the combination of looked-up constants to form the output. Our result of 4753.1 cycles/byte comes very close to this lower bound.

Although RFSB fits well on current embedded microcontrollers and performs at a decent speed, beating implementations of the SHA-3 candidates is not possible due to memory requirements caused by the size of the matrix constant table. When comparing the lightweight AES-based implementation to the results of a ECRYPT initiative that aims to provide a comprehensive collection of lightweight implementations of hash functions [1], RFSB-509 beats well known hash functions such as SHA-256, BLAKE-256, JH-256, and Skein-256 in terms of code size and outperforms JH-256, and sponge-based construction such as PHOTON and SPONGENT. However, it has to be noted that the other implementations do not use the crypto accelerators.

6.2 Reconfigurable Hardware

Table 2 contains our FPGA results taken from the post place-and-route reports which in contrast to post-synthesis figures take actual delays in hardware into account. The different designs using BRAM tables are named RFSB-509_x where x denotes the amount of used block memories.

Table 2. Results of RFSB-509 achieved on a Xilinx Spartan-6 XC6SLX100. We measure the occupied slices, used flip flops(FF), 6-input lookup tables (LUT), and the maximum clock frequency f. From this we compute the cycles/byte, the throughput (Tp), and for comparison the throughput to area ratio (Tp/Area).

Design	**Occ. Slices**	**Slice FFs**	**Slice LUTs**	**18 Kbit BRAM**	f **[MHz]**	**Cycles/ Byte**	**Tp [Mbit/s]**	**Tp/Area** $[\frac{\text{Mbit/s}}{\text{Slices}}]$
AES-based	1526	5793	4920	8	260.2	213.8	9.3	0.01
RFSB-509_8	1402	4621	4316	8	259.4	2.46	805.1	0.57
RFSB-509_15	1381	4106	4277	15	234.7	1.25	1,432.8	1.04
RFSB-509_29	1409	4101	4309	29	223.0	0.65	2,633.9	1.87
RFSB-509_58	1447	4070	3709	58	171.1	0.38	3,480.2	2.41
RFSB-509_116	2112	4071	4690	116	146.2	0.21	5,354.0	2.54

To measure the performance of our implementations we count the clock cycles consumed for loading new message bits into the Merkle-Damgård state, compressing the current state and updating it accordingly. We divide the number of clock cycles by 48 instead of 112 byte since due to the Merkle-Damgård construction only 48 new message bytes enter each 112-byte compression. In addition, we compute the achieved throughput of each implementation as $\text{Tp} = \frac{\text{clock frequency} \times 8}{\text{cycles/byte}}$. In terms of speed the amount of utilized block memories directly correlates with the performance. When using just 8 BRAMs a throughput of 805.1 Mbit/s can be achieved. Our fastest implementation runs at 5.35 Gbit/s and consumes 116 block memories. A designer is thus left with the decision of how many block memory resources he is willing to spend for the hash function

or from a different perspective how many block memories have to be spent for a certain performance goal.

We measure the required area on an FPGA in terms of occupied flip-flops, LUTs, and BRAMs. We also include the number of occupied slices for comparison even though this number has to be considered with care since the slice count itself does not reveal the actual degree of used logic inside the slice and neglects the number of occupied embedded resources (e.g., DSPs and BRAMs). The overall slice count stays on the same level for nearly all of our implementations, only the fastest implementation occupies more slices but the amount of used flip-flops and LUTs does not increase on the same scale. This is due to fact that block memories are spread out over the FPGA and are not located at only one designated area. Usually this leaves more freedom of where to place an implementation on the FPGA but when combining more than just a few BRAMs, the design is spread out leading to partly used slices. It also increases the critical path which explains the decreasing clock frequency for the 58 and 116 BRAM variants.

Note, the performance and size of the AES-based design is inherently depended on the underlying AES core. Nevertheless, using on-the-fly constant generation on an FPGA does not seem to be a good choice since the required resources are nearly the same as in our smallest BRAM implementation plus additional logic for the AES core (393 flip-flops, 326 LUTs, 130 slices, 8 BRAMs, and 21 clock cycles for one encryption). The performance however is two orders of magnitude slower. The only scenario in which an AES-based implementation could be favorable is a setting in which no block memories are present (which of course would also require a non BRAM-based AES implementation).

We compare our results to a recent evaluation of the hardware performance of the five SHA-3 finalists [21] and a recent implementation of the lattice-based hash function SWIFFTX [22] in Table 3. When comparing the plain numbers one has to keep in mind that our implementation results are achieved on low-cost Xilinx Spartan-6 devices while the other results are measured using high-end Xilinx Virtex-5 and Virtex-6 devices. Nevertheless, our implementations keep up with most implementations and get only clearly beaten by the Keccak-256 hardware implementation.

7 Conclusion

In this work, we presented the first implementations of RFSB-509 for embedded microcontrollers and reconfigurable hardware. Different designs from lightweight to high speed implementations have been evaluated and proven to be feasible on both platforms with competitive results in code size/area and performance. Our result show that code-based hash functions are practical and can be used in real-world application involving embedded systems.

References

1. ECRYPT Benchmarking of Lightweight Hash Functions in Atmel AVR devices.

Table 3. In this table our results are compared to other hash functions implemented in hardware. The results of [21] are given for high-end Xilinx Virtex-6 devices, [22] for Xilinx Virtex-5 and our results for the low-cost Xilinx Spartan-6.

Hash Function	Occ. Slices	Tp [Gbit/s]	Tp/Area [$\frac{\text{Mbit/s}}{\text{Slices}}$]	Device [Xilinx]
RFSB-509_58	1,447	3.48	2.41	Spartan-6
RFSB-509_116	2,112	5.34	2.54	Spartan-6
SWIFFTX [22]	16,645	4.85	0.29	Virtex-5
SHA-256 [21]	239	1.63	6.83	Virtex-6
Helion Fast SHA-256 [23]	214	1.5	7.01	Spartan-6
BLAKE-256 [21]	2,530	8.06	3.18	Virtex-6
Grøstl-256 [21]	898	4.20	4.68	Virtex-6
JH-256 [21]	849	5.41	6.37	Virtex-6
Keccak-256 [21]	1,474	18.80	12.76	Virtex-6
Skein-256 [21]	1,628	6.21	3.82	Virtex-6

(accessed 21 July 2012), 2012. `http://perso.uclouvain.be/fstandae/source_codes/hash_atmel/`.

2. D. Augot, M. Finiasz, P. Gaborit, S. Manuel, and N. Sendrier. SHA-3 proposal: FSB, 2008. `http://www.rocq.inria.fr/secret/CBCrypto/fsbdoc.pdf`.
3. D. Augot, M. Finiasz, and N. Sendrier. A Fast Provably Secure Cryptographic Hash Function. Cryptology ePrint Archive, Report 2003/230, 2003. `http://eprint.iacr.org/`.
4. D. Augot, M. Finiasz, and N. Sendrier. A family of fast syndrome based cryptographic hash functions. *Progress in Cryptology–Mycrypt 2005*, pages 64–83, 2005.
5. P. Barreto, P. Cayrel, R. Misoczki, and R. Niebuhr. Quasi-dyadic CFS signatures. In *Information Security and Cryptology*, pages 336–349. Springer, 2011.
6. P. Barreto, R. Misoczki, and M. Simplicio Jr. One-time signature scheme from syndrome decoding over generic error-correcting codes. *Journal of Systems and Software*, 84(2):198–204, 2011.
7. D. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems (accessed 21 July 2012), 2012. `http://bench.cr.yp.to`.
8. D. Bernstein and T. Lange. eBASH: ECRYPT Benchmarking of All Submitted Hashes (accessed 21 July 2012), 2012. `http://bench.cr.yp.to/results-hash.html`.
9. D. J. Bernstein, T. Lange, C. Peters, and P. Schwabe. Really fast syndrome-based hashing. In A. Nitaj and D. Pointcheval, editors, *Progress in Cryptology – AFRICACRYPT 2011*, volume 6737 of *Lecture Notes in Computer Science*, pages 134–152. Springer-Verlag Berlin Heidelberg, 2011.
10. J. Beuchat, N. Sendrier, A. Tisserand, and G. Villard. FPGA Implementation of a Recently Published Signature Scheme. Rapport de recherche RR LIP 2004-14, 2004.
11. B. Biswas and N. Sendrier. McEliece cryptosystem implementation: Theory and practice. *Post-Quantum Cryptography–PQCrypto 2008*, pages 47–62, 2008.
12. P. Cayrel, G. Hoffmann, and E. Persichetti. Efficient implementation of a CCA2-secure variant of McEliece using generalized Srivastava codes. *Public Key Cryptography–PKC 2012*, pages 138–155, 2012.

13. J.-S. Coron and A. Joux. Cryptanalysis of a Provably Secure Cryptographic Hash Function. Cryptology ePrint Archive, Report 2004/013, 2004. http://eprint.iacr.org/.
14. N. Courtois, M. Finiasz, and N. Sendrier. How to achieve a McEliece-based digital signature scheme. *Advances in Cryptology–ASIACRYPT 2001*, pages 157–174, 2001.
15. I. Damgård. A design principle for hash functions. In *Advances in Cryptology–CRYPTO'89 Proceedings*, pages 416–427. Springer, 1990.
16. T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar. MicroEliece: McEliece for embedded devices. *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 49–64, 2009.
17. M. Finiasz. Parallel-CFS: Strengthening the CFS McEliece-based signature scheme. In *Selected Areas in Cryptography*, pages 159–170. Springer, 2011.
18. M. Finiasz, P. Gaborit, and N. Sendrier. Improved fast syndrome based cryptographic hash functions. In *Proceedings of ECRYPT Hash Workshop*, volume 2007, page 155, 2007.
19. P. Fouque and G. Leurent. Cryptanalysis of a hash function based on quasi-cyclic codes. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA conference on Topics in cryptology*, pages 19–35. Springer-Verlag, 2008.
20. P. Gaborit, C. Lauradoux, and N. Sendrier. SYND: a Fast Code-Based Stream Cipher with a Security Reduction. In *Information Theory, 2007. ISIT 2007. IEEE International Symposium on*, pages 186–190, 2007.
21. K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif. Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs. Cryptology ePrint Archive, Report 2012/368, 2012. http://eprint.iacr.org/.
22. T. Gyrfi, O. Cre, G. Hanrot, and N. Brisebarre. High-Throughput Hardware Architecture for the SWIFFT / SWIFFTX Hash Functions. Cryptology ePrint Archive, Report 2012/343, 2012. http://eprint.iacr.org/.
23. Helion. Fast Hash Core Family for Xilinx FPGA (accessed 21 July 2012), 2011. http://heliontech.com/fast_hash.htm.
24. S. Heyse. Code-based cryptography: Implementing the McEliece scheme in reconfigurable hardware. Diploma thesis, 2009.
25. S. Heyse. Low-reiter: Niederreiter encryption scheme for embedded microcontrollers. *Post-Quantum Cryptography–PQCrypto 2010*, pages 165–181, 2010.
26. S. Heyse. Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices. *Post-Quantum Cryptography-PQCrypto 2011*, pages 143–162, 2011.
27. S. Heyse and T. Güneysu. Towards One Cycle per Bit Asymmetric Encryption: Code-Based Cryptography on Reconfigurable Hardware. In *Cryptographic Hardware and Embedded Systems–CHES 2012 (to appear)*. Springer, 2012.
28. P. Kirchner. Improved Generalized Birthday Attack. Cryptology ePrint Archive, Report 2011/377, 2011. http://eprint.iacr.org/.
29. G. Landais and N. Sendrier. CFS Software Implementation. Cryptology ePrint Archive, Report 2012/132, 2012. http://eprint.iacr.org/.
30. S. Manuel. Classification and Generation of Disturbance Vectors for Collision Attacks against SHA-1. Cryptology ePrint Archive, Report 2008/469, 2008. http://eprint.iacr.org/.
31. R. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report*, 42(44):114–116, 1978.

32. M. Meziani, P. Cayrel, and S. Yousfi Alaoui. 2SC: an Efficient Code-based Stream Cipher. *Information Security and Assurance*, pages 111–122, 2011.
33. M. Meziani, Ö. Dagdelen, P. Cayrel, and S. Yousfi Alaoui. S-FSB: An Improved Variant of the FSB Hash Family. *Information Security and Assurance*, pages 132–145, 2011.
34. J. Ness. Microsoft certification authority signing certificates added to the Untrusted Certificate Store (accessed 21 July 2012). Microsoft Security Research and Defense, 2012. `http://blogs.technet.com/b/srd/archive/2012/06/03/microsoft-certification-authority-signing-certificates-added-to-the-untrusted-certificate-store.aspx`.
35. H. Niederreiter. A public-key cryptosystem based on shift register sequences. In *Advances in Cryptology–EUROCRYPT85*, pages 35–39. Springer, 1986.
36. NIST. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA3) Family (accessed 21 July 2012), 2007. `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`.
37. U. D. of Commerce. Secure Hash Standard (SHS). Technical report, National Institute of Standards and Technology, 2008.
38. R. Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992.
39. L. Rothamel and M. Weiel. Report Cryptography Lab SS2011 Implementation of the RFSB hash function, 2011. `http://www.cayrel.net/IMG/pdf/Report.pdf`.
40. M. Saarinen. Linearization attacks against syndrome based hashes. *Progress in Cryptology–INDOCRYPT 2007*, pages 1–9, 2007.
41. A. Shoufan, T. Wink, G. Molter, S. Huss, and F. Strentzke. A novel processor architecture for McEliece cryptosystem and FPGA platforms. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 98–105. IEEE, 2009.
42. M. Stevens. On collisions for MD5. *Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science (June 2007)*, 2007.
43. M. Stevens, A. Lenstra, and B. De Weger. Chosen-prefix collisions for MD5 and colliding X. 509 certificates for different identities. *Advances in Cryptology–EUROCRYPT 2007*, pages 1–22, 2007.