Report Cryptography Lab SS2011 Implementation of the RFSB hash function

Lucas Rothamel, Manuel Weiel
Technische Universität Darmstadt, Fachbereich Informatik
Fachgebiet Theoretische Informatik - Kryptographie und Computeralgebra
Hochschulstraße 10, D-64289 Darmstadt, Germany
{uw19avax,id79xaba}@informatik.tu-darmstadt.de

***Abstract*—In his paper "Really fast syndrome-based hashing"[1], Daniel J. Bernstein et. al. introduces an enhancement to FSB, a code based hash function. No implementation of this has been published so far however. The goal of this Cryptography Lab is therefore to produce efficient C and Java based implementations of this hash function.**

***Index Terms*—RFSB, RFSB-509, code based cryptography, syndrome-based hash function**

I. Introduction to SB and FSB

The Syndrome Based hash function (SB) proposed in 2003 by Augot et al. [2] is the foundation of this work. It was the first code-based hash family presented to our knowledge, using a Merkle-Damgård [3],[4] construction based on a compression function to create a cryptographic hash function. Code-Based hash functions represent one promising option in the strive for secure hash functions in the event quantum computers start breaking current crypto systems based problems like integer factorisation or the discrete logarithm.

Fast Syndrome Based Hashing (FSB) was proposed by Finiasz et al. [5] in 2007. It introduces quasi-cyclic codes instead of random codes, which reduces memory consumption whilst increasing its speed. FSB was then a SHA-3 candidate, but was still too slow to be selected, so further improvements are necessary. For further details on FSB the reader is recommended reading "Hash Functions Based on Coding Theory" by Meziani et al. [6] as an overview on Code-Based Hash Functions.

II. Introduction to RFSB

Really Fast Syndrobe Based Hashing (RFSB) was introduced by Bernstein et al. [1]. RFSB also uses the Merkle-Damgård [3],[4] contruction for secure hash functions to build a parameterized code based hash function.

Bernstein[1] et all note: There are four RFSB parameters: an odd prime number r, a positive integer b, a positive integer w and a $2^b \times r$-bit compressed matrix. The prime r is chosen so that 2 has order $r-1$ in the group F_r^*; i.e. so that the cyclotomic polynomial $(x^r-1)/(x-1)$ in $F_2[x]$ is irreducible.

III. Specification of RFSB-509

RFSB-509 is suggested by Daniel J. Bernstein[1] by choosing $r = 509$, $b = 8$, $w = 112$ and a matrix A as below.

For $k = 0, ..., 3$ and $j = 0, ..., 255$ the following functions

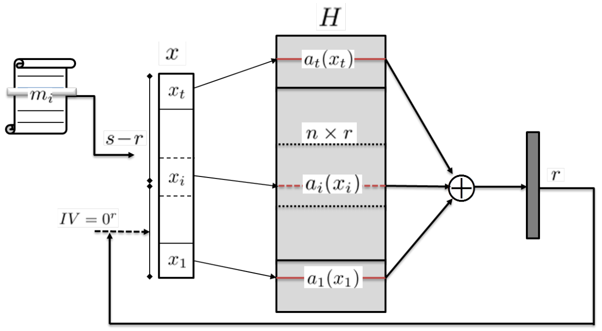

Figure 1. The RFSB Hash Function [6]

are defined:

$$P_k(j) = (0, ..., 0, j, k)$$

such that their output is always a 16 Byte sequence. Also let $AES(k)$ be the AES Encryption of k using the encryption key zero. Then $P(j)$ is defined as the concatenation:

$$P(j) = P_0(j).P_1(j).P_2(j).P_3(j)$$

and $a(j)$ as

$$a(j) = AES(P_0(j)).AES(P_1(j)).AES(P_2(j)).AES(P_3(j))$$

being the concatenation of the AES encryptions and applying the reduction to $x^{509} - 1$. We then define $a_i(j)$ as

$$a_i(j) = a(j) << 128(w-i)$$

where $<<$ notates the binary left shift function.

Thereafter all $a_j(x_i)$ are XORed to give a result of length r. This will be used again as an input value for the next cycle as in the Merkle-Damgård construction, in the final round giving the hash result. This can be seen in Figure 1 as graphical representation of the algorithm RFSB.

IV. Suggesting RFSB-227, RFSB-379 and RFSB-1019

As stated above, the parameter r needs to be chosen such that 2 has order $r-1$ in the group F_r^*. This is equivalent to 2 being a primitive root of r. Modern computers are able to work with 64 bits of data per instruction, suggesting to use an r close to a multiple of but smaller than 64. RFSB-509 suggests $r = 509$, which is very close to $512 = 64 \times 8$ with only three bits wasted per 509 computation bits. 379 is the best alternative we have found that is smaller than 509 and still reasonably close to a multiple of 64 (384), the difference being 5. As a value smaller 256, 227 was implemented, however this wastes a total of 29. Another more suitable value appears to be 1019, which is just below $1024 = 64 \times 16$, the difference being just five. We therefore suggest *RFSB-227*, *RFSB-379* and *RFSB-1019* as implementations with r chosen accordingly, and leaving b, w and A as suggested by Bernstein et al. [1].

V. Implementation in C

A. Used Libraries

The AES Library of Brian Gladman[11] was used for matrix generation. It provides a fast and long standing AES implementation for generation of the A matrix. It is provided in the package *aes-src-11-01-11.zip*.

B. File and Function Description

1) main.c:

a) CPUFREQUENCY: This represents the CPU Frequency in GHz, used to calculate the cycles per byte value.

b) main(): This is the main function of the implementation, first checking the passed parameters, selecting the RFSB variant to be used, calculating the A matrix, calculating the hash and calculating the performance measurements at the end.

2) RFSB.c: This file contains the main RFSB functions in a library style.

a) GenerateMatrix(): This function calculates the complete compressed matrix.

b) GenerateColumn(): This function calculates one column of the compressed matrix.

c) ComputeRound(): This function hashes one round with the given matrix.

d) ComputeFile(): This function implements the Merkle-Damgård [3],[4] construction of RFSB, looping over the input file to incrementally read in the file and hash it.

3) Tools.c:

a) rotateAndxor(): Rotates a given array by a given amount and xors it with another array which is then returned.

b) fold(): Folds the first n bits onto the last ones.

c) printHash(): Prints a given hash to the command line.

C. Usage

GNU Make may be used to compile the C Implementation. Type *make* in the folder's command line to start compilation. The resulting executable is named *rfsb*.

As a first parameter, provide the file to be hashed. An optional Parameter *-r* $\langle number \rangle$ may be appended, where $\langle < number > \rangle$ represents one of 227, 379, 509 or 1019, to indicate the r value to be used. If no r value is specified, r is taken to be 509, starting a computation of *RFSB-509*.

After start of the program, the file that is being hashed, its size in bytes as well as the RFSB variant used are printed. When ready, the hash value is printed, together with the elapsed cpu time in seconds, and the resulting megabytes per second. A cycles per byte value is calculated, be aware that the cpu frequency must be entered correctly in constant *CPUFREQUENCY* of $main.c$ before compilation to ensure a correct calculation.

VI. Implementation in Java

The Java implementation is a standalone application. It operates significantly slower then the C implementation, part of which is due to the fact that Java by default knows no unsigned data types apart from *char*, requiring overhead that the C implementation does not worry about. In actual fact, for small files (approx. 5MB) the C implementation is finished with all calculations before the Java implementation has finished the generation of the

Only the RFSB-509 specification by Bernstein[1] et al. has been implemented in the Java version, as that way at least some optimisations were possible.

A. Used Libraries

The Java implementation makes use of the FlexiProvider[12] Library for AES. Due to the experienced performance constraint in Java, no FlexiProvider integration is provided.

B. File and Function Description

1) RFSB_Impl: This class represents the main class of the implementation, containing the following functions:

a) main(): The first executable function, primarily handling time measurement as well as file input reading.

b) computeAll(): This function implements the Merkle-Damgård [3],[4] construction of RFSB, by allocating memory and handling the looping over the input data and previous results.

c) computeRound(): This function implements one single round, calling the *rorateAndxor()* in each round for calculation of the round.

d) rotateAndxor(): In every round of the RFSB-Construction, a rotation by a few bits and subsequent xor are necessary. This function performs the rotation required.

e) ComputeMatrix(): Calculates the A matrix entries using the *CompressedMatrix()* function for each row.

f) CompressedMatrix(): Calculates each row of the A matrix, using the AES library with an all-zero key.

2) Tools: The Tools class provides static access to a few generic functions not special to RFSB.

a) getbytesFromFile(): Reads the content of a file, returning it as a byte array.

b) UnsignedByteAt(): Returns a single Byte at the given index of a byte array.

c) fold3(): Executes a single fold operation on a byte array by 3 bits. This translates into an x-or of the first 3 bits onto the last 3 bits, and then setting the first 3 bits to zero, corresponding the folding from a 512 bit number to a 509 bit one.

C. Usage

As the only parameter, provide the file name to be hashed. When finished, the hash is returned, together with the time taken (in seconds) and the resulting megabytes per second rating.

Scheme	Speed (Cycles/byte)
FSB	257 [5] / 95.53 [7]
S-FSB	160 [8]
SWIFFTX	57 [9]
RFSB	13.62 [1]
Keccack	12.6 [10]

Table I
PERFORMANCE COMPARISON BETWEEN CODE-BASED HASH FAMILIES AND SOME OTHER HASH FUNCTIONS.[6]

Variant	Speed (Cycles/byte)
RFSB-227	42.4
RFSB-379	62.8
RFSB-509	120.5
RFSB-1019	152.8

Table II
PERFORMANCE COMPARISON OF OUR C RFSB IMPLEMENTATION WITH DIFFERENT R VALUES.

VII. PERFORMANCE MEASUREMENTS

All performance measurements were done on a Intel Core i7 2.96 GHz CPU running Ubuntu 10.10 and a 1GB random data input file held on a RAM disk. The C implementation was compiled using the GCC 4.4.5 for Ubuntu compiler with -O3 optimisations. All quoted speeds are the average value of 5 successive measurements.

In table I, some performance measurements by Meziani[6] et al. comparing different code-based hash functions can be seen. Table II then shows our own performance measurements, comparing different values of r. Only the implementation in C was tested, as the Java implementation is significantly slower in all cases.

VIII. CONCLUSION

RFSB has been implemented both in C and in Java with decent speeds reached in the C implementation. Further improvements may be made by de-generalising the code for a specific variant of RFSB, as the current code is quite general. It should be easy to implement further RFSB modes in the code by changing a few lines of code to add the required parameters. However, the generalised code is certainly not the fastest possible.

In our implementations, the A matrix is calculated before using it. It would generally be possible to save this statically in the program code, making the executable somewhat larger, but probably also faster. All time measurements in table II include the generation of the A matrix before starting the hashing itself.

The Java implementation unfortunately does not shine through high efficiency. At least to our knowledge, the Java VM does not support bigger unsigned data types, and calculating with only 8 bits at a time already decreases efficiency by a factor 8 when compared to the 64 bits that C is capable of on most modern computers. Writing components that can run natively appears to be more suitable for number crunching like building a hash function.

An Implementation in assembly language could be an option to achieve further speed improvements. Bernstein[1] et al. give suggestions on the use of Horner's Rule for a faster RFSB-509 implementation, and combining this with an assembly language implementation could be a goal when further efficiency improvements are required.

Finally it needs to be noted that no publicly available implementation of RFSB exists to our knowledge, therefore we were unable to compare our results with another implementation. It should not be used in a production system, as correctness cannot be guaranteed.

APPENDIX
SUITABLE VALUES OF r FOR RFSB

For RFSB [1] the parameter r needs to be chosen such that 2 has order $r-1$ in the group F_r^*. The following prime numbers between 128 and 1024 fulfill this criterion:

131, 139, 149, 163, 173, 179, 181, 197, 211, 227, 269, 293, 317, 347, 349, 373, 379, 389, 419, 421, 443, 461, 467, 491, 509, 523, 541, 547, 557, 563, 587, 613, 619, 653, 659, 661, 677, 701, 709, 757, 773, 787, 797, 821, 827, 829, 853, 859, 877, 883, 907, 941, 947, 1019

REFERENCES

[1] Daniel J. Bernstein, Tanja Lange, Christiane Peters, and Peter Schwabe. Really fast syndrome-based hashing. In AFRICACRYPT 2011, Lecture Notes in Computer Science, Vol. 6737, pp. 134–152. Springer-Verlag Berlin Heidelberg, 2011.

[2] D. Augot, M. Finiasz, and N. Sendrier. A family of fast syndrome based cryptographic hash functions. In E. Dawson and S. Vaudenay Eds., MyCrypt 2005. Springer LNCS 3615,64–83 (2005).

[3] R. C. Merkle. One Way Hash Functions and DES. In Gilles Brassard, editor. Advances in Cryptology – CRYPTO'89, Proc. Volume 435:428–446. Springer,1990.

[4] I. Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, Advances in Cryptology – CRYPTO'89, Proc. Volume 435:416–427. Springer, 1990.

[5] D. Augot, M. Finiasz, Ph. Gaborit, S. Manuel, and N. Sendrier. SHA-3 proposal: FSB. Submission to the SHA-3 NIST competition, 2008.

[6] M. Meziani, S.M. El Yousfi Alaoui and P.-L. Cayrel. Hash Functions Based on Coding Theory. 2011.

[7] http://bench.cr.yp.to/ebash.html.

[8] M. Meziani, Ö. Dagdelen, P.-L- Cayrel, and S. M. E. Alaoui. S-FSB: An Improved Variant of the FSB Hash Family. To appear (2011)

[9] Y. Arbitman, G. Dogon, V. Lyubashevsky, D. Micciancio and C. Peikert and A. Rosen. SWIFFTX: A Proposal for the SHA-3 Standard. http://www.eecs.harvard.edu/ alon/PAPERS/lattices/swifftx.pdf. Submission to NIST (2008).

[10] G. Bertoni and J. Daemen and M. Peeters and G. Van Assche. Keccak specifications. http://keccak.noekeon.org/Keccak-specifications-2.pdf. Submission to NIST (Round 2) (2009).

[11] Brian Gladman. AES C Library. http://www.gladman.me.uk/, last checked August 3rd, 2011.
[12] http://www.flexiprovider.de/