Really fast syndrome-based hashing

Daniel J. Bernstein[1], Tanja Lange[2], Christiane Peters[2], and Peter Schwabe[3]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
tanja@hyperelliptic.org, c.p.peters@tue.nl
[3] Institute of Information Science
Academia Sinica, 128 Section 2 Academia Road, Taipei 115-29, Taiwan
peter@cryptojedi.org

Abstract. The FSB (fast syndrome-based) hash function was submitted to the SHA-3 competition by Augot, Finiasz, Gaborit, Manuel, and Sendrier in 2008, after preliminary designs proposed in 2003, 2005, and 2007. Many FSB parameter choices were broken by Coron and Joux in 2004, Saarinen in 2007, and Fouque and Leurent in 2008, but the basic FSB idea appears to be secure, and the FSB submission remains unbroken. On the other hand, the FSB submission is also quite slow, and was not selected for the second round of the competition.

This paper introduces RFSB, an enhancement to FSB. In particular, this paper introduces the RFSB-509 compression function, RFSB with a particular set of parameters. RFSB-509, like the FSB-256 compression function, is designed to be used inside a 256-bit collision-resistant hash function: all known attack strategies cost more than 2^{128} to find collisions in RFSB-509. However, RFSB-509 is an order of magnitude faster than FSB-256. On a single core of a Core 2 Quad Q9550 CPU, RFSB-509 runs at 10.67 cycles/byte: faster than SHA-256, faster than 7 of the 14 second-round SHA-3 candidates, and faster than 3 of the 5 SHA-3 finalists.

Key words: compression functions, collision resistance, linearization, generalized birthday attacks, information-set decoding, tight reduction to L1 cache.

1 Introduction

Finding collisions in a very simple compression function of the form

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

This work was supported by the National Science Foundation under grant 0716498, by the European Commission under Contract ICT-2007-216499 CACE, by the European Commission under Contract ICT-2007-216646 ECRYPT II, and by the National Science Council, National Taiwan University and Intel Corporation under Grant NSC99-2911-I-002-001. Part of this work was carried out when Peter Schwabe was employed by Technische Universiteit Eindhoven. Permanent ID of this document: 067a9e99992a54f43b7f859c81b25d16. Date: 2011.05.08.

turns out to be surprisingly difficult. As an illustration of this difficulty we challenge the reader to break the following parameters:

- w, the weight of the sum, is 112. (Sum here means exclusive-or; we do not bother saying "modulo 2" everywhere.)
- The input chunks $m_1, m_2, \ldots, m_{112}$ range over $\{0, 1, \ldots, 255\}$. The compression function therefore has 896 bits of input.
- Each of the 28672 constants $c_1[0], \ldots, c_1[255], \ldots, c_{112}[0], \ldots, c_{112}[255]$ is an independent uniform random 509-bit vector. The compression function therefore has 509 bits of output.

At first one might think that linear algebra instantaneously finds preimages in this function, with collisions as a trivial side effect. Select 509 of these 28672 constants; there is a good chance that those 509 are linearly independent, guaranteeing that linear algebra modulo 2 will reveal a subset adding up to the target. The reason that this attack does not work is that the resulting subset is extraordinarily unlikely to have the form $c_1[m_1], c_2[m_2], \ldots, c_{112}[m_{112}]$: in particular, the subset will normally have size close to $509/2$, much larger than 112. In other words, linear algebra easily finds random codewords in the linear code defined by the matrix of constants, but it does not find *low-weight* codewords, a classic problem in coding theory.

One can also try to find collisions directly, without a detour through preimages. Select, for example, the 510 constants $c_i[j]$ with $j \in \{0, 1, 2, 3\}$ for $1 \leq i \leq 50$ and $c_i[j]$ with $j \in \{0, 1, 2, 3, 4\}$ for $51 \leq i \leq 112$. Use linear algebra to find a nonempty subset adding up to 0, and try to split the subset into 224 constants $c_1[m_1], c_2[m_2], \ldots, c_{112}[m_{112}]$ and $c_1[m'_1], c_2[m'_2], \ldots, c_{112}[m'_{112}]$. Low weight is no longer an obstacle: the subset has about a 2^{-10} chance of having size exactly 224. The reason that this attack does not work is that the subset has chance only about $(6/16)^{50}(10/32)^{62} \approx 2^{-175}$ of containing exactly two $c_1[\cdots]$, exactly two $c_2[\cdots]$, etc.

There is a long history of proposals of compression functions of this type (see Section 2) and also a long history of attacks (see Section 4). Many of the proposals are motivated by speed: the additions are very fast; the structure $c_1[m_1] \oplus c_2[m_2] \oplus \cdots \oplus c_w[m_w]$ also has obvious virtues of incrementality and parallelizability. However, the large matrix of random constants makes small hardware implementations impossible, and software implementations end up spending far longer waiting for memory access than actually performing useful computations. This problem was already highlighted five years ago by Augot, Finiasz, and Sendrier in [2, Section 6].

Obtaining very high speed requires reducing memory-access costs, which in turn requires compressing the matrix. This is impossible for a uniform random matrix, but security does not seem to require a uniform random matrix. Finiasz, Gaborit, and Sendrier in [26] proposed using a quasi-cyclic matrix: each block of the matrix is a block of rotations of a single vector. In [26, Section 4.2] they suggested choosing the vector length r so that the polynomial $(x^r - 1)/(x - 1)$ is irreducible in $\mathbf{F}_2[x]$. They argued, under this assumption on the vector length, that finding a low-weight codeword for a random quasi-cyclic matrix is

a well-known hard problem in coding theory, as hard as the generic low-weight-codeword problem.

Most of the specific parameters proposed in [26] were promptly broken in two different ways, showing two mistakes in the parameter selection. The first mistake, exploited by Saarinen in [40], is that [26] chose w too large compared to the vector length r; security against linearization requires w to be considerably smaller than $r/2$. The second mistake, exploited by Fouque and Leurent in [27], is that [26, Section 6] ignored [26, Section 4.2] and chose powers of 2 for r, violating the irreducibility of $(x^r-1)/(x-1)$ and allowing the attacker to concentrate on small factors of $(x^r-1)/(x-1)$.

Both of these mistakes were fixed in FSB [3], a first-round SHA-3 submission by Augot, Finiasz, Gaborit, Manuel, and Sendrier. FSB resists the previous attack strategies and remains unbroken today. Bernstein, Lange, Niederhagen, Peters, and Schwabe in [9] needed days on an 8-computer cluster (using 64GB of RAM and 5.5TB of disk) to find collisions in the scaled-down FSB-48 compression function by a streamlined generalized birthday attack; for comparison, an unoptimized attack on the FSB-48 *hash* function finds collisions in about a minute on just one core on one of these computers with negligible memory usage. The compression function has vector length $r = 197$ (subsequently truncated to 192 bits, but the rotations are of 197-bit vectors), weight $w = 24$, and 14 bits in each m_i. Scaling the same attack to the 1024-bit FSB-256 compression function would cost far more than 2^{128}. Other attacks also do not seem to pose a threat.

However, FSB is quite slow, and was not selected for the second round of the SHA-3 competition. The best speed reported in eBASH [8] for FSB-256 on an Intel Core 2 Quad Q9550 (10677) (`berlekamp`) is 95.53 cycles/byte (using an assembly-language implementation by Schwabe). SHA-256 takes just 15.26 cycles/byte on the same computer.

Contents of this paper. We introduce the RFSB ("really fast syndrome-based") compression function, an improved version of FSB. In particular, we introduce RFSB-509, a compression function that reaches higher speeds than SHA-256 on a Core 2 Quad CPU, while maintaining higher collision security than SHA-256 against every known attack. See Section 2 for the definitions of RFSB and RFSB-509.

The FSB-to-RFSB improvements come from two sources. First, the design of RFSB pays much closer attention to the efficiency of the computation of $c_1[m_1] \oplus c_2[m_2] \oplus \cdots \oplus c_w[m_w]$. The most important result of this analysis is that RFSB permutes the vectors in the FSB matrix. This permutation has no effect on the best attacks known, and might also seem irrelevant to speed, but we show that it eliminates a critical inefficiency in FSB. See Section 3 for a detailed explanation of our algorithm for computing RFSB.

Second, the design of RFSB pays much closer attention to the cost of attacks. This allows the RFSB parameters to be tuned much more tightly than the FSB parameters were, while still keeping all known attacks safely above our 2^{128} security target. Our attack survey in Section 4 corrects several algorithm-analysis errors in the literature, and incorporates some new improvements that we found.

Like FSB and earlier designs of this type, RFSB offers incremental hashing and parallelizable hashing. Unlike FSB, RFSB allows fast on-demand matrix generation, making it implementable in very small hardware.

Building a hash function from a compression function. We emphasize that our goal in this paper is the traditional goal of building a collision-resistant compression function F for fixed-length messages. RFSB, specifically RFSB-509, is our proposal for F. Merkle–Damgård iteration then produces a collision-resistant compression function $\overline{F}$ for longer messages; see, e.g., [22, Theorem 3.1]. Our discussion of speed focuses on the speed of this iterated function $\overline{F}$ for long messages.

Many, perhaps most, papers on hash-function design use the iteration mode as an argument for weakening their collision-resistance goals. If the compression function F has input (v, m), where v is the previous chaining value (or initialization vector) and m is an attacker-controlled block, then these papers say that $F(v, m) = F(v', m')$ with $(v, m) \neq (v', m')$ is merely a "pseudo-collision" and that it qualifies as a "collision" only if $v = v'$. However, many papers on hash-function cryptanalysis say that finding a pseudo-collision is a "certificational attack" even if $v \neq v'$. To avoid this debate we have designed RFSB to stop all pseudo-collisions.

One interesting consequence of incrementality is that RFSB can precompute the v-dependent part of its output before m is available. The preliminary FSB designs in [1], [2], and [26] had the same feature, but the FSB SHA-3 submission does not, because it permutes the bits of (v, m). According to [25], this permutation was added in reaction to [27, Section 3], in which Fouque and Leurent object to the following "IV weakness" in the preliminary FSB designs: a collision of the form $\overline{F}(m) = \overline{F}(m')$, where m and m' are distinct single-block messages, implies $\overline{F}(p, m) = \overline{F}(p, m')$ for every prefix p. We do not see why this is any more troubling than the following "weakness" in the compression functions of SHA-1, SHA-2, and every SHA-3 candidate: a collision of the form $\overline{F}(m) = \overline{F}(m')$, where m and m' are distinct identical-length block-aligned messages, implies $\overline{F}(m, q) = \overline{F}(m', q)$ for every suffix q. Our goal is to prevent these collisions from occurring in the first place.

To build a full-fledged cryptographic hash function, suitable for use in message authentication, commitment protocols, etc., we can add any reasonably strong output filter to RFSB-509. One reasonable choice of output filter is SHA-256; of course, the 256-bit output length of SHA-256 then reduces collision resistance to 2^{128}. We emphasize that an output filter adds only a small constant overhead to the cost of hashing; the speed of hashing a long message is the speed of our compression function.

2 Design of RFSB

This section defines the RFSB family of compression functions. In particular, this section defines the RFSB-509 compression function. This section then reviews

the literature, showing in particular how RFSB differs from FSB and explaining why we introduced these differences.

Specification of RFSB. 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 unit group $\mathbf{F}_r^*$; i.e., so that the cyclotomic polynomial $(x^r-1)/(x-1)$ in $\mathbf{F}_2[x]$ is irreducible.

The RFSB output is an r-bit string, represented as a sequence of $\lceil r/8 \rceil$ bytes in little-endian form. The string $(s_0, s_1, \ldots, s_{r-1})$ represents the polynomial $s_0 + s_1x + \cdots + s_{r-1}x^{r-1}$ in the ring $\mathbf{F}_2[x]/(x^r-1)$. For example, for $r = 13$, the byte string $(12, 16)$ represents the bit string $(0,0,1,1,0,0,0,0,0,0,0,0,1)$, which in turn represents the polynomial $x^2 + x^3 + x^{12}$ in $\mathbf{F}_2[x]/(x^{13}-1)$.

The RFSB input is a bw-bit string, represented as a sequence of $\lceil bw/8 \rceil$ bytes in little-endian form. This string represents a sequence $(m_1, m_2, \ldots, m_w)$, where each m_i is an element of $\{0, 1, \ldots, 2^b-1\}$.

The compressed RFSB matrix is a sequence of r-bit strings $c[0], c[1], \ldots, c[2^b-1]$. We define $c_i[j] = c[j]x^{128(w-i)}$ in the ring $\mathbf{F}_2[x]/(x^r-1)$; in other words, $c_i[j]$ is a $128(w-i)$-bit rotation of $c[j]$. This matrix specifies the relationship between the RFSB input and the RFSB output: RFSB is the function

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

i.e., the function that maps an input $(m_1, m_2, \ldots, m_{w-1}, m_w)$ to the output $x^{128(w-1)}c[m_1] \oplus x^{128(w-2)}c[m_2] \oplus \cdots \oplus x^{128}c[m_{w-1}] \oplus c[m_w]$ in $\mathbf{F}_2[x]/(x^r-1)$.

Sometimes we refer to the uncompressed RFSB matrix. This is a $2^b w \times r$-bit matrix containing the strings $c_i[j]$, for $i \in \{1, 2, \ldots, w\}$ and $j \in \{0, 1, \ldots, 2^b-1\}$. We do not mean to suggest that implementations need to compute this matrix.

Specification of RFSB-509. Our RFSB-509 proposal has $r = 509$, $b = 8$, and $w = 112$. In other words, RFSB-509 maps $(m_1, m_2, \ldots, m_{112})$, where each m_i is an 8-bit string, to

$$x^{128(112-1)}c[m_1] \oplus x^{128(112-2)}c[m_2] \oplus \cdots \oplus x^{128}c[m_{111}] \oplus c[m_{112}]$$

in $\mathbf{F}_2[x]/(x^{509}-1)$. We chose the parameters $(509, 8, 112)$ to maximize the software speed of RFSB (see Section 3) while keeping the cost of all known attacks above 2^{128} (see Section 4).

The compressed RFSB-509 matrix is defined as a concatenation of AES outputs. Specifically, each 509-bit $c[j]$ is obtained by encrypting the four 16-byte strings $(0, j, 0, \ldots, 0, 0)$, $(1, j, 0, \ldots, 0, 0)$, $(2, j, 0, \ldots, 0, 0)$, $(3, j, 0, \ldots, 0, 0)$ with AES, concatenating the 128-bit outputs into a 512-bit string, and reducing modulo $x^{509}-1$ (i.e., folding the last three bits onto the first three bits). The AES key is a 128-bit all-0 key.

We comment that implementors can trade space for time by computing each $c[j]$ when it is used, rather than precomputing and storing the AES outputs. The hardware area required for RFSB-509 (and an AES-based output filter) is then not much larger than the hardware area required for AES. The regular input structure also allows "counter-mode caching", a sharing of work in the first

two rounds of AES; see [11]. We also comment that varying the AES key is a natural way to "salt" RFSB-509, converting RFSB-509 into a keyed compression function.

History and credits. In a 1970 technical report [46], Zobrist introduced the compression function $(m_1, m_2, \ldots, m_w) \mapsto c_1[m_1] \oplus c_2[m_2] \oplus \cdots \oplus c_w[m_w]$, with random choices of matrix entries $c_i[j]$, as a non-cryptographic hash function. Zobrist's parameter choices were too small to be of cryptographic interest.

The same compression function was reintroduced and discarded in a Eurocrypt 1997 paper [6] by Bellare and Micciancio. The only difference between Zobrist's hash and "XHASH" in [6, Section 1] is that $c_i[m_i]$ is replaced by $H(i, m_i)$, allowing much longer input chunks m_i while raising security questions and efficiency questions for the underlying function H. Bellare and Micciancio described "XHASH" as insecure, independently of H, because they were able to find collisions by linearization for large w. They instead proposed various slower alternatives to $\oplus$, such as modular multiplication. They did not consider small values of w.

A very similar compression function with limited w had been introduced a decade earlier by Damgård at Crypto 1989 [22, Section 4.3]. Damgård used addition rather than $\oplus$, took $w = 256$ for 128-bit output (or more generally $w \approx 2r$ for r-bit output), and took $m_i \in \{0, 1\}$. Camion and Patarin introduced generalized birthday attacks (without giving them that name) at Eurocrypt 1991 [16] and showed that Damgård's function is breakable in subexponential time.

As far as we know, the first proposal with limited w and several bits in each m_i was the preliminary version of FSB by Augot, Finiasz, and Sendrier appearing in [1] and [2]. The larger range of m_i appears to allow a security level exponential in r with a polynomial-size matrix, specifically a matrix containing $\Theta(r^2)$ bits. However, the implicit constant in $\Theta(r^2)$ is quite large, and the time to access the matrix is quite troublesome.

FSB with a quasi-cyclic matrix was introduced by Finiasz, Gaborit, and Sendrier in [26]. FSB with a *truncated* quasi-cyclic matrix was introduced by Augot, Finiasz, Gaborit, Manuel, and Sendrier in [3] and submitted to the SHA-3 competition. These proposals appear to allow a security level exponential in r with a compressed matrix containing $\Theta(r)$ bits, although the implicit constant in $\Theta(r)$ is still quite large.

Comparison between FSB and RFSB. The FSB-256 proposal from [3] follows Zobrist's formula $(m_1, m_2, \ldots, m_w) \mapsto c_1[m_1] \oplus c_2[m_2] \oplus \cdots \oplus c_w[m_w]$ with $r = 1061$ bits of output (truncated to 1024 bits), weight $w = 128$ in the sum, and $b = 14$ bits in each input chunk m_i. The uncompressed FSB-256 matrix entries $c_i[0], c_i[1], \ldots, c_i[16383]$ are generated modulo $x^{1061} - 1$ as

$$\begin{array}{rrrcr}
c_i[0], & c_i[0]x, & c_i[0]x^2, & \ldots, & c_i[0]x^{1023}, \\
c_i[1024], & c_i[1024]x, & c_i[1024]x^2, & \ldots, & c_i[1024]x^{1023}, \\
\vdots & & & & \\
c_i[15360], & c_i[15360]x, & c_i[15360]x^2, & \ldots, & c_i[15360]x^{1023},
\end{array}$$

where $c_i[0], c_i[1024], \ldots, c_i[15360]$ are generated from digits of π.

FSB-256 handles $14 - 1024/128 = 6$ bits of new input for each 1024-bit addition, while RFSB-509 handles $8 - 512/112 \approx 3.43$ bits of new input for each 512-bit addition. We are comfortable with a smaller $r = 509$, and a larger ratio w/r, because of our tighter security analysis; see Section 4. These changes allow the compressed RFSB-509 matrix to fit into just 16384 bytes, comfortably inside L1 cache on typical CPUs.

FSB-256 uses almost r rotations of each vector, while RFSB-509 uses only $w \approx r/4.5$ rotations of each vector. The number of rotations is important because it is the compression factor for the matrix. We could have allowed further compression as an option in RFSB-509 by modifying the definition of the matrix to use $2w$ or $3w$ or $4w$ rotations of each vector; but this option would not help fast software implementations such as ours, and it would slightly complicate implementations that generate matrix entries on the fly.

The most important difference between FSB and RFSB is the order of matrix entries: FSB defines $c_i[j]$ as $c_i[0]x^j$ (at least for a wide range of j), while RFSB defines $c_i[j]$ as $c[j]x^i$ (or rather $c[j]x^{128(w-i)}$), exchanging the roles of i and j. This change is important because j is unpredictable, a chunk of input, while i is a constant, the position of the chunk. The rotation distances in FSB are therefore input-dependent, making them quite expensive. The rotation distances in RFSB are constant, allowing several optimizations that are not available to FSB.

3 Speed of RFSB-509

We implemented the RFSB-509 compression function in assembly language, targeting the widely deployed 45nm Intel Core 2 line of CPUs, specifically the Core 2 Quad Q9550 (10677). To allow public benchmarking of RFSB-509 we implemented a complete hash function that uses RFSB-509 with Merkle–Damgård iteration, an all-zero initialization vector, and SHA-256 as output filter. We padded the original message to a multiple of 48 bytes as follows: first zero-pad to 40 bytes plus a multiple of 48 bytes; then append 8 bytes contaning, in little-endian form, the number of bytes of the message before padding. We placed the software into the public domain to maximize reusability, and submitted it to eBASH [8] for benchmarking.

The eBASH results show RFSB-509 running at 10.67 cycles/byte on a Core 2 Quad Q9550 named `berlekamp`. For comparison, eBASH reports SHA-256 running at 15.26 cycles/byte on the same machine using the assembly-language implementation of SHA-256 from Wei Dai's Crypto++ library, and reports that the SPHlib and OpenSSL implementations of SHA-256 are slower. The 256-bit SHA-3 finalists run at 7.90 (Skein), 8.82 (BLAKE), 11.69 (Keccak), 17.20 (JH), and 22.32 (Grøstl) cycles/byte.

The algorithm that we use to compute RFSB-509 is explained in this section. This algorithm relies critically on the predictable rotation distances in RFSB; we would not be able to achieve similar speeds for FSB.

This section also describes two additional algorithmic improvements that provide even higher speed for some applications. One improvement, incremental hashing, is well known, while the other improvement, fast batch verification, is less well known. We have not implemented these improvements; we emphasize that RFSB-509 is already quite fast without these improvements. This section concludes by discussing ways to compute RFSB without variable-index table lookups.

How to compute RFSB-509. We view the computation of RFSB-509 as having three phases. The first phase loads the input bytes $m_1, m_2, \ldots, m_{112}$ and multiplies their values by 64 to make them usable offsets for loads from the matrix. Specifically, we load an 8-byte chunk of input into a 64-bit integer register and then extract 8 matrix offsets from these eight bytes using 7 copy, 8 shift, and 8 mask instructions. The total cost for 112 bytes of input is 14 loads, 98 copies, 112 shifts, and 112 masks.

The second phase loads the 509-bit matrix entries $c[m_1], c[m_2], \ldots, c[m_{112}]$ and adds them together with appropriate shifts to form the $(128(112-1)+512)$-bit (i.e., 14720-bit) polynomial

$$\begin{aligned}
& x^{128(112-1)}c[m_1] \oplus x^{128(112-2)}c[m_2] \oplus \cdots \oplus x^{128}c[m_{111}] \oplus c[m_{112}] \\
= \quad & x^{128(112+2)}(\qquad\qquad\qquad\qquad\qquad\qquad c[m_1]_3) \\
& \oplus x^{128(112+1)}(\qquad\qquad\qquad\qquad c[m_1]_2 \oplus c[m_2]_3) \\
& \oplus x^{128(112+0)}(\qquad\qquad\qquad c[m_1]_1 \oplus c[m_2]_2 \oplus c[m_3]_3) \\
& \oplus x^{128(112-1)}(c[m_1]_0 \oplus c[m_2]_1 \oplus c[m_3]_2 \oplus c[m_4]_3) \\
& \oplus x^{128(112-2)}(c[m_2]_0 \oplus c[m_3]_1 \oplus c[m_4]_2 \oplus c[m_5]_3) \\
& \oplus \cdots \\
& \oplus x^{128(1)}(c[m_{111}]_0 \oplus c[m_{112}]_1) \\
& \oplus x^{128(0)}(c[m_{112}]_0)
\end{aligned}$$

where $c[m_i] = c[m_i]_0 \oplus x^{128}c[m_i]_1 \oplus x^{256}c[m_i]_2 \oplus x^{384}c[m_i]_3$. We represent this polynomial in radix x^{128}, with 128-bit coefficients such as $c[m_1]_2 \oplus c[m_2]_3$ stored in the 128-bit XMM registers on the Core 2. Loading all the matrix entries takes $112 \cdot 4 = 448$ XMM load instructions, and there are $112 \cdot 3 - 3 = 333$ xors of coefficients shown above. The AMD64 architecture allows offsets to be multiplied by a constant as part of the load operation, and if this constant were allowed to be 16 then we could eliminate some of the shifts and masks mentioned above; but the constant is limited to 8.

The third phase reduces this polynomial modulo $x^{509}-1$. This is an iterative process, clearing more and more bits of the polynomial until only 509 coefficients remain. The most obvious reduction strategy, starting from $\sum_i x^{128i}p_i$ where each p_i is a 128-bit polynomial, is to replace p_j by p_j/x^{509} for a selected $j \geq 4$: in other words, replace p_{j-4} by $p'_{j-4} = p_{j-4} \oplus x^3(p_j \bmod x^{125})$; replace p_{j-3} by $p'_{j-3} = p_{j-3} \oplus \lfloor p_j/x^{125} \rfloor$; and replace p_j by $p'_j = 0$. Using this strategy

to successively eliminate $p_{114}, p_{113}, \ldots, p_4$ produces a 512-bit polynomial $p_0 \oplus x^{128}p_1 \oplus x^{256}p_2 \oplus x^{384}p_3$; replacing p_0 by $p'_0 = p_0 \oplus \lfloor p_3/x^{125} \rfloor$ and replacing p_3 by $p'_3 = p_3 \bmod x^{125}$ then completes the reduction modulo $x^{509} - 1$.

This reduction strategy would be satisfactory if the Core 2 had fast instructions to shift p_j down by 125 bits, producing $\lfloor p_j/x^{125} \rfloor$, and to shift p_j up by 3 bits, producing $x^3(p_j \bmod x^{125})$. However, the XMM upwards-shift instruction `psllq` actually operates in parallel on two 64-bit halves of a 128-bit vector: if $p_j = A + x^{61}B + x^{64}C + x^{125}D$ then a 3-bit `psllq` produces $x^3A + x^{67}C$, discarding the desired $x^{64}B$. There is a 128-bit shift-up instruction `pslldq`, but it can only shift by multiples of 8 bits. Similar comments apply to the downwards shifts `psrlq` and `psrldq`. Clearing p_j in this way takes 8 instructions: copy (since `psllq` overwrites its input), `psllq` to obtain $x^3A + x^{67}C$, `psrlq` to obtain $B + x^{64}D$, a shuffle instruction `pshufd` to obtain $x^{64}B$, `pxor` to obtain $x^3A + x^{64}B + x^{67}C$, `psrldq` to obtain D, a `pxor` into p_{j-4}, and a `pxor` into p_{j-3}.

To eliminate some instructions we use a higher-distance reduction strategy, replacing p_j by $p_j/x^{16 \cdot 509}$ for each $j \geq 64$. In other words, we replace p_{j-64} by $p'_{j-64} = p_{j-64} \oplus x^{48}(p_j \bmod x^{80})$; replace p_{j-63} by $p'_{j-63} = p_{j-63} \oplus \lfloor p_j/x^{80} \rfloor$; and replace p_j by $p'_j = 0$. The shift distances 48 and 80 here are multiples of 8 bits, so clearing p_j in this way takes only 5 instructions: copy, `pslldq`, `psrldq`, `pxor`, `pxor`. We could similarly replace p_j by $p_j/x^{8 \cdot 509}$ for $64 > j \geq 32$, but this would have a smaller benefit and would complicate the data flow discussed below.

This third phase involves $5(115 - 64) + 8(64 - 4) + 5 = 740$ arithmetic instructions. In total the three phases involve 462 loads and 1395 arithmetic instructions. We have checked these totals against our implementation.

There are two obvious bottlenecks in this computation. First, the Core 2 can perform at most 1 load per cycle, so 462 loads take at least 462 cycles. Second, the Core 2 can perform at most 3 arithmetic instructions per cycle, so 1395 arithmetic instructions take at least 465 cycles. With Merkle–Damgård iteration each call to the compression function processes 48 bytes of input (together with the previous output), so 465 cycles yield a lower bound of 9.6875 cycles per byte. There are several ways to rebalance loads and arithmetic (loading smaller chunks of input, for example, or replacing chaining-variable loads with arithmetic), but the balance here is already very good.

The above description ignores all questions of instruction scheduling and register allocation. There is data flow connecting each input chunk to 8 consecutive offsets, connecting each offset to 4 consecutive polynomial coefficients, connecting p_j to p_{j-64} and p_{j-63} for $j \geq 64$, and connecting p_j to p_{j-4} and p_{j-3} for $64 > j \geq 4$. We organize the computation as successive elimination of $p_{63}, p_{62}, \ldots, p_4$, preceded "just in time" by each relevant elimination of $p_{114}, p_{113}, \ldots, p_{64}$. The polynomial coefficients active at each moment then fit into the 16 available XMM registers on the AMD64 architecture, and the offsets, pointers, etc. fit into the 15 available integer registers.

There is enough parallelism in the computation to overcome most latency problems and come close to the 9.6875-cycle-per-byte lower bound, if instructions are scheduled carefully. As mentioned above, our software actually uses 10.67

cycles per byte, about 10% above the lower bound. Note that 448 cycles (97% of the lower bound and 87% of our actual time) are explained by the 112 512-bit matrix loads, so there is very little room for improvement on this CPU.

Extra speed: incremental hashing. Zobrist in [46, page 6] emphasized the incremental nature of his hash, i.e., the ability to quickly update the hash output for a small change to the input: "moves will typically involve two XOR operations." For example, changing m_2 to m_2' simply adds $c_2[m_2] \oplus c_2[m_2']$ to the output. Bellare and Micciancio in [6] advertised the same feature, with various generalizations of $\oplus$ and without credit to Zobrist; their paper title was "A new paradigm for collision-free hashing: incrementality at reduced cost."

Chaining an incremental compression function such as RFSB produces a somewhat incremental hash function for long messages. (We say "somewhat" to emphasize, as in [6, Section 1.1], that this requires storage of all intermediate compression-function outputs, not merely the final output.) The block that changes can be recomputed incrementally at very high speed. Each subsequent block must be recomputed, but RFSB allows some of this computation to be skipped, since the only change to the input is in the chaining value. Note that Damgård's tree hash [22, Theorem 3.2] has only a logarithmic number of subsequent blocks for long messages.

Extra speed: fast batch verification. One can compute the sum of many RFSB outputs at higher speed than computing each output separately. The idea is very simple: the number of copies of $c_i[j]$ in the sum is the number of occurrences of j as m_i in the inputs; one can first count this number of occurrences, and then add $c_i[j]$ once if the number is odd. There are $2^b w$ separate counters, and computing all of them requires just one fast pass through all of the inputs. Each b bits require one counter update, which is faster than an r-bit xor for large r. All other steps become negligible as the number of inputs increases, but we nevertheless point out a speedup in those steps for large r: one can combine the $c_i[j]$ additions across i into a convolution, which in turn can be performed in subquadratic time by fast-multiplication techniques.

One can, at almost twice that speed, perform the following simple statistical check of a batch of alleged RFSB outputs: select 50% of the inputs, compute the sum of the RFSB outputs, and see whether the sum matches the sum of alleged outputs. This check cannot be fooled with probability above 50%. This is an example of what Bellare, Garay, and Rabin in [5] call the "atomic random subset test."

To achieve a much higher security level one can compute many independent sums. The cost of this computation grows sublinearly with the number of sums, and therefore sublinearly with the security level, because the sums have large overlaps that can be shared; see generally [5]. For comparison, the cost of separately computing each RFSB output grows linearly with the security level.

Extra security: avoiding variable-index table lookups. Our RFSB software performs a variable-index table lookup $c[m_i]$ for each input chunk m_i. This could be a problem for applications that hash secret data, such as HMAC: table

lookups can leak index information through cache-timing attacks, hyperthreading attacks, etc., the same way that conditional branches can leak condition information. See generally [43].

One way to hide indices is to look up *all* table entries, using arithmetic operations to combine the results into each desired $c[m_i]$. RFSB has many table lookups to perform in parallel, and for large tables one can reduce the amount of arithmetic by batching these lookups into a sorting computation, as described in the following two paragraphs.

The inputs to this sorting computation are w vectors $(m_i, 1, i)$ together with 2^b vectors $(j, 0, c[j])$. Sorting brings each j next to all m_i's that are equal to j: first $(0, 0, c[0])$ is followed by all $(m_i, 1, i)$ with $m_i = 0$, then $(1, 0, c[0])$ is followed by all $(m_i, 1, i)$ with $m_i = 1$, etc. A linear-time pass from left to right then replaces each $(m_i, 1, i)$ with $(c[m_i], 1, i)$. A second sorting computation then puts $(c[m_i], 1, i)$ back into order of i.

It is well known that essentially-linear-time sorting does not require variable array indexing, and does not require conditional branches. For example, the Batcher sorting network sorts n items using approximately $(1/2)n(\lg n)^2$ compare-exchange steps, and one can do even better for large n. See [31, Section 5.3.4] for a survey of the extensive literature on sorting networks.

Another way to avoid variable-index table lookups is to compute $c[m_i]$ directly from m_i. The Käsper–Schwabe bitsliced implementation of AES [30] takes only about 7 cycles per byte, and new Intel CPUs support AES instructions taking only about 1.4 cycles per byte in parallelizable modes, i.e., about 90 cycles to compute $c[m_i]$. This is an order of magnitude slower than our software.

We have two suggestions for improving speed in this situation. One suggestion is to replace AES with something much simpler and faster. The full security of AES is certainly not required for RFSB: all that we need is a function generating a few elements of $\mathbf{F}_2[x]/(x^r - 1)$ without any obvious linear structure. The design of such functions is outside the scope of this paper.

The other suggestion, specific to HMAC, is to eliminate the initial keying in HMAC. Normally the HMAC input is public (such as a packet sent through the network), and if no secret key is inserted then RFSB with fast table lookups can be applied to this public input. The second stage of HMAC needs a key but is applied only to a short message; this stage can simply be SHA-256. Eliminating the initial keying allows MAC forgery via offline collision attacks, but we have designed RFSB precisely to make those collision attacks fail.

4 Attacks against RFSB

This section reviews and analyzes three different strategies to find collisions in FSB-type hash functions, including some new attack improvements. All of the strategies cost more than 2^{128} to find collisions in RFSB-509. This section also reviews reducibility, an attack tool that converts many FSB-type hash functions into smaller hash functions that are easier to break, and shows that this tool is inapplicable to RFSB.

We describe each attack for general RFSB parameters r, b, w. We illustrate the scalability of the attacks by considering the special case $b = 8$, $w \approx r/4$. In this case RFSB compresses $\approx 2r$ bits to r bits, using $r/4$ additions of r-bit vectors, i.e., 2 additions of r-bit vectors per byte of input; the compressed RFSB matrix fits into $32r$ bytes; and the cost of each attack is exponential in r.

We make some comments about preimage attacks as a stepping-stone to collision attacks, but we do not systematically analyze preimage attacks. Several modern hash-function designs, such as Keccak [12] and Quark [4], drop the traditional goal of having a preimage exponent twice as large as the best collision exponent; the question of whether RFSB reaches this goal is outside the scope of this paper. We are satisfied knowing that first preimages require breaking an output filter, and that second preimages are even more difficult to find than collisions.

Cost of computation. In this paper, cost means price/performance ratio: the size of the attack machine, multiplied by the time taken by the attack machine. For example, a brute-force k-bit key search can be carried out in time $2^k t$ by a small attack machine, or in time $2^k t/100$ by an attack machine 100 times larger, where t is the time to test a single key; these machines have the same cost.

In a classic paper thirty years ago, Brent and Kung proved that every n-bit multiplication circuit costs at least $n^{3/2}$. Here cost has a precise definition as the circuit area, multiplied by the time taken by the circuit, scaled by a particular constant reflecting the circuit speed, wire size, etc.; see [14, Theorem 3.1]. The same bound applies to other computations such as sorting; what matters is that the computations include n different shifts of one input, where the shift distance depends on the other input. The model of computation in [14] is a very broad class of two-dimensional circuits, including all of the most efficient computer technologies available today.

We use the same definition of cost in this paper. There are some future technologies, notably quantum computers, that cannot be efficiently simulated in this model, but we explicitly disregard those technologies.

We caution the reader that a naive operation count, as used in many cryptanalytic papers, is a poor predictor of cost when the allowed operations include random access to an arbitrarily large array. For example, sorting n keys uses fewer operations than performing n separate hash-function evaluations, even if the hash function is quite fast; but the *cost* of sorting n keys becomes vastly larger than the cost of n separate hash-function evaluations as n grows.

In the real world, sorting 2^{50} keys is a major engineering challenge, while 2^{50} hash-function evaluations are a rather easy computation. The current public sorting record is merely $2^{46.5}$ bytes, sorted by Yahoo's Hadoop in 10380 seconds on 3452 nodes with 13808 disks and 27616 cores. For comparison, readily available software performs 2^{47} separate evaluations of SHA-1 in 10380 seconds on just 20 PCs, each equipped with two GTX 295 graphics cards. We see overwhelming evidence that naive operation counts exaggerate the threat posed by communication-intensive cryptanalytic algorithms, and that this exaggeration

grows with the size of the problem being solved. We see no evidence of similar problems with the cost model in [14].

Linearization. The following preimage attack was introduced by Bellare and Micciancio in [6, Appendix A]. First choose $m_1, m_2, \ldots, m_w$ and compute the difference $\Delta = h \oplus c_1[m_1] \oplus c_2[m_2] \oplus \cdots \oplus c_w[m_w]$, where h is the target hash. Then, for each i, choose $m'_i \neq m_i$ and compute the difference $\delta_i = c_i[m_i] \oplus c_i[m'_i]$. Use linear algebra to find a subset of $\delta_1, \ldots, \delta_w$ with sum Δ, i.e., a linear relation $\epsilon_1\delta_1 \oplus \cdots \oplus \epsilon_w\delta_w = \Delta$, if a linear relation exists. Then $c_1[m_1 \oplus \epsilon_1(m'_1 \oplus m_1)] \oplus \ldots \oplus c_w[m_w \oplus \epsilon_w(m'_w \oplus m_w)] = h$ as desired. If no linear relation exists, try again with new choices of m_i and m'_i.

The main obstacle to this attack is that if $w < r$ then $\delta_1, \ldots, \delta_w$ generate a linear space of dimension at most w (and sometimes less), so under suitable randomness assumptions the desired linear relation exists with probability at most $2^w/2^r$. The expected number of iterations is therefore at least $2^r/2^w$; e.g., approximately $2^{0.75r}$ if $w \approx r/4$.

Saarinen in [40, Section 5] suggested doubling the number of generators for $w \leq r/2$ by computing two differences for each i, say $\delta_i = c_i[m_i] \oplus c_i[m'_i]$ and $\delta'_i = c_i[m_i] \oplus c_i[m''_i]$. There are two obstacles to this attack: first, if $2w < r$ then a linear relation exists with probability at most $2^{2w}/2^r$; second, a relation is useful with probability only $(3/4)^w$, since a relation involving both δ_i and δ'_i is useless. The expected number of iterations is therefore at least $2^r/3^w$; e.g., approximately $2^{0.60r}$ if $w \approx r/4$.

More generally, for $k \geq 1$ and $w \leq r/k$, Saarinen suggested computing k differences $c_i[m_i] \oplus c_i[m'_i], c_i[m_i] \oplus c_i[m''_i], \ldots$ for each i. Then there are kw generators, so a linear relation exists with probability at most $2^{kw}/2^r$. A relation is useful with probability $((k+1)/2^k)^w$, so the expected number of iterations is slightly above $2^r/(k+1)^w$; e.g., approximately $2^{0.42r}$ if $w \approx r/4$ and $k = 4$.

Saarinen in [40, Section 4] suggested a different way to double the number of generators for collision attacks: compute $\delta_i = c_i[m_i] \oplus c_i[m'_i]$ and $\delta'_i = c_i[n_i] \oplus c_i[n'_i]$, and use linear algebra to find a subset of $\delta_1, \delta'_1, \ldots, \delta_w, \delta'_w$ with sum $\Delta = c_1[m_1] \oplus c_1[n_1] \oplus c_2[m_2] \oplus c_2[n_2] \oplus \cdots \oplus c_w[m_w] \oplus c_w[n_w]$. The expected number of iterations here is at least $2^r/4^w$; e.g., approximately $2^{0.50r}$ if $w \approx r/4$.

More generally, for $w \leq r/(2k)$, one can take $2k$ generators for each i, with the first k generators of the form $c_i[m_i] \oplus c_i[m'_i], c_i[m_i] \oplus c_i[m''_i], \ldots$ and the second k generators of the form $c_i[n_i] \oplus c_i[n'_i], c_i[n_i] \oplus c_i[n''_i], \ldots$. A subset of these generators has sum $\Delta = c_1[m_1] \oplus c_1[n_1] \oplus c_2[m_2] \oplus c_2[n_2] \oplus \cdots \oplus c_w[m_w] \oplus c_w[n_w]$ with probability at most $2^{2kw}/2^r$. This subset is useful, revealing a collision, with probability $((k+1)^2/4^k)^w$, so the expected number of iterations is slightly above $2^r/(k+1)^{2w}$; e.g., approximately $2^{0.21r}$ if $w \approx r/4$ and $k = 2$.

A hybrid approach is to take $2k+2$ generators for v values of i and $2k$ generators for $w-v$ values of i, assuming that $2kw+2v \leq r$ and $0 \leq v \leq w$. The expected number of iterations is then slightly above $2^r/((k+1)^{2w}((k+2)/(k+1))^{2v})$. For $k = 1$ this approach appears in [40, Section 5.2].

For RFSB-509 the optimal attack parameters are $k = 2$ and $v = 30$, and the expected number of iterations is slightly above $2^{509}/(9^{112}(16/9)^{30}) > 2^{129}$. Since

our security target is 2^{128}, we do not need to assess the cost of each iteration, but we make one comment on this cost: namely, taking more than r generators allows the cost of linear algebra to be amortized across several relations.

Note that [27, page 2] claims a simpler formula for the number of iterations for linearization, namely $(4/3)^{r-2w}$ whenever $w \leq r/2$. This claim is correct for $r/4 \leq w \leq r/2$ (take $k = 1$ in the hybrid approach above), but understates the number of iterations for $w < r/4$. The problem is that for $r/4 \leq w \leq r/2$ one can reach r generators by taking at most 4 generators for each i, but for $w < r/4$ this is no longer true. For the same reason, we disagree with the comment in [40, Section 5] that large values of k do not have "cryptanalytic advantages."

In the opposite direction, [25, Section 3.3] states that linearization is applicable only for $w \geq r/4$. Our RFSB-509 example disproves this statement. As a more extreme example, for the case $w = r/8$ used in [3], linearization finds collisions in time approximately $2^{0.41r}$. The time grows rapidly as w/r drops.

Generalized birthday attacks. The k-sum problem is to find $x_1 \in L_1, \ldots, x_k \in L_k$ such that $x_1 \oplus x_2 \oplus \cdots \oplus x_k = 0$, given k lists $L_1, \ldots, L_k$ of r-bit strings drawn uniformly and independently at random.

If $k = 2^{i-1}$ and each list has $2^{r/i}$ elements then generalized birthday attacks solve this problem using $O(k \cdot 2^{r/i})$ operations. In the next three paragraphs we review Wagner's single-modulus generalized birthday attack from [44], which is slightly simpler and faster than the original multiple-modulus generalized birthday attack introduced by Camion and Patarin in [16].

Merge lists L_1 and L_2 to find all sums of elements $u \oplus v$ with $u \in L_1$ and $v \in L_2$ that are 0 on their first r/i bits. Store these sums in a new list $L_{1,2}$. The expected number of elements in $L_{1,2}$ is again $2^{r/i}$. In the same way build a list $L_{3,4}$ from lists L_3 and L_4 and so on and a list $L_{k-1,k}$ from lists L_{k-1} and L_k. This first level of operations thus generates 2^{i-2} lists of expected length $2^{r/i}$ containing r-bit strings with their first r/i bits zero.

On the next level merge lists $L_{1,2}$ and lists $L_{3,4}$ to find all sums of elements $u \oplus v$ with $u \in L_{1,2}$ and $v \in L_{3,4}$ that are 0 on their first $2r/i$ bits. Store these sums in a new list $L_{1,2,3,4}$. As r/i bits are already known to be zero, the expected size of this list is again $2^{r/i}$. Similarly build lists $L_{5,6,7,8}$ and so on to list $L_{k-3,k-2,k-1,k}$.

Continue for $i-2$ levels to build lists in the same way to obtain two lists $L_{1,\ldots,k/2}$ and $L_{k/2+1,\ldots,k}$, each containing an expected number of $2^{r/i}$ strings that are 0 on their first $(i-2)r/i$ bits. Compute all sums $u \oplus v$ with $u \in L_{1,\ldots,k/2}$ and $v \in L_{k/2+1,\ldots,k}$ to see, on average, one element with all r bits zero.

Applying this attack to an FSB-type hash function, and taking $k = w$, runs into an obstacle: there are only 2^b entries in each of the w input lists. The attack needs $2^{r/(1+\lfloor \lg k \rfloor)}$ entries. Usually b is much smaller than $r/(1 + \lfloor \lg k \rfloor)$, drastically reducing the success probability of the attack.

However, Coron and Joux in [21] used generalized birthday attacks to break many instances of the preliminary version of FSB presented in [1]. The idea is to take k smaller than w and to build the starting lists $L_1, \ldots, L_k$ by considering all possible xors of columns from one block. To build fewer but larger lists one

can also consider xors of columns from multiple blocks, two columns per block. The solution of Wagner's tree algorithm is then the xor of $2w$ columns, exactly 2 per block. For extensions see [2], [7], and [37].

In the case of RFSB-509 the number of operations is minimized for $k = 16$. There are $\binom{256}{2} \approx 2^{15}$ possible 2-column combinations $c_1[m_1] \oplus c_1[m_1']$, and therefore 2^{105} possible 14-column combinations involving $c_1, c_1, c_2, c_2, \ldots, c_7, c_7$. Generate all of these combinations, and build a list containing the combinations that have their first 4 bits equal to 0, leaving 505 bits uncontrolled; this list has approximately 2^{101} elements. Build 16 lists from $c_1, c_2, \ldots, c_{112}$ by repeating this procedure. Then apply the generalized birthday attack, zeroing $5 \cdot 101 = 505$ bits. Overall this takes 15 merging steps on lists of size 2^{101}.

The cost of a single merging step is more than 2^{150} by [14, Theorem 3.1]; see the "Cost of computation" subsection above. Reducing the list size to 2^{82} would bring the merging cost down to approximately 2^{128}; but then 5 rounds clear only 410 bits, leaving 99 bits uncontrolled. At most $105 - 82 = 23$ bits can be controlled through precomputation, so the algorithm must be repeated 2^{76} times on average, bringing the cost above 2^{200}. We have considered several further variants of Wagner's attack, including the "Pollard" variant in [9, Section 2.2], and all of them cost far more than 2^{128}.

Information-set decoding. Augot, Finiasz, and Sendrier in [1, Section 4.2] presented an algorithm that uses roughly

$$\min \left\{ 2^r \Big/ \left(\binom{r/w_0}{2} + 1 \right)^{w_0} : w_0 \in \{1, 2, \ldots, w\} \right\}$$

iterations to find a collision $c_1[m_1] \oplus \cdots \oplus c_w[m_w] = c_1[m_1'] \oplus \cdots \oplus c_w[m_w']$. For $w \approx r/4$ this number of iterations is roughly $2^{0.3r}$. Each iteration uses some linear algebra, inverting an $r \times r$ matrix.

The second attack stated in Section 1 is a simplified version of the attack from [1]; the main difference is that their algorithm also allows having 0 columns in one block. For RFSB-509, with $r = 509$ and $w = 112$, the expected number of iterations is $(2^{510}/((\binom{4}{2} + \binom{4}{0})^{50}(\binom{5}{2} + \binom{5}{0})^{62}) \approx 2^{155}$.

Our new paper [10] presents a generalized version of the attack from [1]. The generalization combines ideas from various improved versions of information-set decoding, and restructures those ideas to fit the more complicated context of useful codewords having exactly two $c_1[\cdots]$, exactly two $c_2[\cdots]$, etc. In particular, the attack uses the ideas of Lee-Brickell [32], Leon [33], and Stern [42] to increase the chance of success per iteration at the expense of more effort, and more memory, per iteration. These generalized attack parameters allow the number of bit operations to be reduced below 2^{145}; but this is still far above 2^{128}, and the cost is even larger than the number of bit operations.

Reducibility. As mentioned in Section 1, the preliminary quasi-cyclic FSB proposals in [26] used powers of 2 for r, specifically $r = 512$ and $r = 1024$. Fouque and Leurent broke these proposals in [27].

To understand the Fouque–Leurent idea, consider transforming RFSB-509 into a smaller compression function f that works as follows. Take a string $(m_1, m_2, m_3, m_4, m_5)$ as input. Apply RFSB-509 to the repeated input

$$(m_1, m_2, m_3, m_4, m_5, \ldots, m_1, m_2, m_3, m_4, m_5, 0, 0).$$

Note that the output in $\mathbf{F}_2[x]/(x^{509} - 1)$ is a constant (for the $(0, 0)$) plus a multiple of $\varphi = 1 + x^{128 \cdot 5} + x^{128 \cdot 10} + \cdots + x^{128 \cdot 105}$. Subtract the constant and divide by $g = \gcd\{x^{509} - 1, \varphi\}$, obtaining an element of $\mathbf{F}_2[x]/((x^{509} - 1)/g)$. The output of f is this element.

Observe that f is another Zobrist-type hash: $f(m_1, m_2, m_3, m_4, m_5)$ has the shape $f_1[m_1] \oplus f_2[m_2] \oplus f_3[m_3] \oplus f_4[m_4] \oplus f_5[m_5]$. The difficulty of finding collisions in this hash depends on how long its output is, i.e., on the degree of $(x^{509} - 1)/g$. If this output is short then one can easily find collisions in f, and therefore collisions in RFSB-509.

This attack does not work because the output is actually very long: g turns out to be $x - 1$, so $(x^{509} - 1)/g$ has degree 508. Attacks might marginally benefit from this change in degree, but not enough to compensate for the restricted set of inputs to f.

Modifying the attack to construct multiples of other polynomials φ also cannot work. The only divisors of $x^{509} - 1$ are $x^{509} - 1$, $(x^{509} - 1)/(x - 1)$, $x - 1$, and 1, corresponding to finding multiples of 1, $x - 1$, $(x^{509} - 1)/(x - 1)$, and $x^{509} - 1$ respectively. Finding multiples of 1 or $x - 1$ is trivial but useless, as in the $(m_1, m_2, m_3, m_4, m_5)$ example. Finding multiples of $(x^{509} - 1)/(x - 1)$ or $x^{509} - 1$ is a very hard preimage problem, preventing the attack from even getting started; an attacker able to solve that preimage problem would not have any need to transform RFSB-509 into a smaller function.

All RFSB parameters, and all parameters in the FSB SHA-3 submission [3], are protected in the same way against the Fouque–Leurent attack: r is chosen so that $(x^r - 1)/(x - 1)$ is irreducible. We are not aware of attacks against primes r with reducible $(x^r - 1)/(x - 1)$, but insisting on irreducibility does not severely restrict the choice of r.

References

[1] Daniel Augot, Matthieu Finiasz, Nicolas Sendrier, *A fast provably secure cryptographic hash function* (2003). URL: http://eprint.iacr.org/2003/230. Citations in this document: §1, §2, §4, §4, §4, §4.

[2] Daniel Augot, Matthieu Finiasz, Nicolas Sendrier, *A family of fast syndrome based cryptographic hash functions*, in Mycrypt 2005 [24] (2005), 64–83. URL: http://lasecwww.epfl.ch/pub/lasec/doc/AFS05.pdf. Citations in this document: §1, §1, §2, §4.

[3] Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stéphane Manuel, Nicolas Sendrier, *SHA-3 proposal: FSB* (2008). URL: http://www-rocq.inria.fr/secret/CBCrypto/fsbdoc.pdf. Citations in this document: §1, §2, §2, §4, §4.

[4] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Maria Naya-Pasencia, QUARK*: a lightweight hash*, in CHES 2010 [35] (2010), 1–15. URL: http://131002.net/quark/quark_full.pdf. Citations in this document: §4.

[5] Mihir Bellare, Juan A. Garay, Tal Rabin, *Fast batch verification for modular exponentiation and digital signatures*, in Eurocrypt '98 [38] (1998), 236–250. URL: http://cseweb.ucsd.edu/~mihir/papers/batch.html. Citations in this document: §3, §3.

[6] Mihir Bellare, Daniele Micciancio, *A new paradigm for collision-free hashing: incrementality at reduced cost*, in Eurocrypt '97 [28] (1997), 163–192. URL: http://www-cse.ucsd.edu/~mihir/papers/incremental.html. Citations in this document: §2, §2, §3, §3, §4.

[7] Daniel J. Bernstein, *Better price-performance ratios for generalized birthday attacks*, in Workshop Record of SHARCS'07: Special-purpose Hardware for Attacking Cryptographic Systems (2007). URL: http://cr.yp.to/papers.html#genbday. Citations in this document: §4.

[8] Daniel J. Bernstein, Tanja Lange (editors), *eBASH: ECRYPT Benchmarking of All Submitted Hashes (accessed 21 April 2011)*, 2011. URL: http://bench.cr.yp.to. Citations in this document: §1, §3.

[9] Daniel J. Bernstein, Tanja Lange, Ruben Niederhagen, Christiane Peters, Peter Schwabe, *FSBday: implementing Wagner's generalized birthday attack against the SHA–3 round–1 candidate FSB*, in Indocrypt 2009 [39] (2009), 18–38. URL: http://eprint.iacr.org/2009/292. Citations in this document: §1, §4.

[10] Daniel J. Bernstein, Tanja Lange, Christiane Peters, Peter Schwabe, *Faster 2-regular information-set decoding*, in IWCC 2011 [17] (2011), 81–98. URL: http://eprint.iacr.org/2011/120. Citations in this document: §4.

[11] Daniel J. Bernstein, Peter Schwabe, *New AES software speed records*, in Indocrypt 2008 [18] (2008), 322–336. URL: http://cr.yp.to/papers.html#aesspeed. Citations in this document: §2.

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, *Note on Keccak parameters and usage* (2010). URL: http://keccak.noekeon.org/NoteOnKeccakParametersAndUsage.pdf. Citations in this document: §4.

[13] Gilles Brassard (editor), *Advances in cryptology — CRYPTO '89, 9th annual international cryptology conference, Santa Barbara, California, USA, August 20–24, 1989, proceedings*, Lecture Notes in Computer Science, 435, Springer, Berlin, 1990. ISBN 0-387-97317-6. See [22].

[14] Richard P. Brent, H. T. Kung, *The area-time complexity of binary multiplication*, Journal of the ACM **28** (1981), 521–534. URL: http://wwwmaths.anu.edu.au/~brent/pub/pub055.html. Citations in this document: §4, §4, §4, §4.

[15] Johannes Buchmann, Jintai Ding (editors), *Post-quantum cryptography, second international workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17–19, 2008, proceedings*, Lecture Notes in Computer Science, 5299, Springer, 2008. See [25].

[16] Paul Camion, Jacques Patarin, *The knapsack hash function proposed at Crypto'89 can be broken*, in Eurocrypt '91 [23] (1991), 39–53. URL: http://hal.inria.fr/inria-00075097/en/. Citations in this document: §2, §4.

[17] Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, Chaoping Xing (editors), *Coding and cryptology: third international workshop, IWCC 2011, Qingdao, China, May 30–June 3, 2011, proceedings*, Lecture Notes in Computer Science, 6639, Springer, 2011. ISBN 978-3-642-20900-0. See [10].

[18] Dipanwita Roy Chowdhury, Vincent Rijmen, Abhijit Das (editors), *Progress in cryptology — INDOCRYPT 2008, 9th international conference on cryptology in India, Kharagpur, India, December 14–17, 2008, proceedings*, Lecture Notes in Computer Science, 5365, Springer, 2008. ISBN 978-3-540-89753-8. See [11].

[19] Christophe Clavier, Kris Gaj (editors), *Cryptographic hardware and embedded systems — CHES 2009, 11th international workshop, Lausanne, Switzerland, September 6–9, 2009, proceedings*, Lecture Notes in Computer Science, 5747, Springer, 2009. ISBN 978-3-642-04137-2. See [30].

[20] Gérard D. Cohen, Jacques Wolfmann (editors), *Coding theory and applications, 3rd international colloquium, Toulon, France, November 2–4, 1988, proceedings*, Lecture Notes in Computer Science, 388, Springer, 1989. ISBN 0-387-51643-3. See [42].

[21] Jean-Sébastien Coron, Antoine Joux, *Cryptanalysis of a provably secure cryptographic hash function* (2004). URL: http://eprint.iacr.org/2004/013. Citations in this document: §4.

[22] Ivan B. Damgård, *A design principle for hash functions*, in Crypto '89 [13] (1990), 416–427. Citations in this document: §1, §2, §3.

[23] Donald W. Davies (editor), *Advances in cryptology — EUROCRYPT '91, workshop on the theory and application of cryptographic techniques, Brighton, UK, April 8–11, 1991, proceedings*, Lecture Notes in Computer Science, 547, Springer, 1991. ISBN 3-540-54620-0. See [16].

[24] Ed Dawson, Serge Vaudenay (editors), *Progress in cryptology — Mycrypt 2005, first international conference on cryptology in Malaysia, Kuala Lumpur, Malaysia, September 28–30, 2005, proceedings*, Lecture Notes in Computer Science, 3715, Springer, 2005. ISBN 3-540-28938-0. See [2].

[25] Matthieu Finiasz, *Syndrome based collision resistant hashing*, in PQCrypto 2008 [15] (2008), 137–147. URL: http://www-rocq.inria.fr/secret/Matthieu.Finiasz/research/2008/finiasz-pqcrypto08.pdf. Citations in this document: §1, §4.

[26] Matthieu Finiasz, Philippe Gaborit, Nicolas Sendrier, *Improved fast syndrome based cryptographic hash functions*, in Proceedings of ECRYPT Hash Workshop 2007 (2007). URL: http://www-roc.inria.fr/secret/Matthieu.Finiasz/research/2007/finiasz-gaborit-sendrier-ecrypt-hash-workshop07.pdf. Citations in this document: §1, §1, §1, §1, §1, §1, §1, §2, §4.

[27] Pierre-Alain Fouque, Gaëtan Leurent, *Cryptanalysis of a hash function based on quasi-cyclic codes*, in CT-RSA 2008 [34], 19–35. Citations in this document: §1, §1, §4, §4.

[28] Walter Fumy (editor), *Advances in cryptology — EUROCRYPT '97, international conference on the theory and application of cryptographic techniques, Konstanz, Germany, May 11–15, 1997, proceedings*, Lecture Notes in Computer Science, 1233, Springer, Berlin, 1997. ISBN 3-540-62975-0. See [6].

[29] Christoph G. Günther, *Advances in cryptology — EUROCRYPT '88, proceedings of the workshop on the theory and application of cryptographic techniques held in Davos, May 25–27, 1988*, Lecture Notes in Computer Science, 330, Springer, Berlin, 1988. ISBN 3-540-50251-3. See [32].

[30] Emilia Käsper, Peter Schwabe, *Faster and timing-attack resistant AES-GCM*, in CHES 2009 [19] (2009), 1–17. URL: http://eprint.iacr.org/2009/129. Citations in this document: §3.

[31] Donald E. Knuth, *The art of computer programming, volume 3: sorting and searching*, 2nd edition, Addison-Wesley, Reading, 1998. ISBN 0-201-89685-0. Citations in this document: §3.

[32] Pil Joong Lee, Ernest F. Brickell, *An observation on the security of McEliece's public-key cryptosystem*, in Eurocrypt '88 [29] (1988), 275–280. Citations in this document: §4.

[33] Jeffrey S. Leon, *A probabilistic algorithm for computing minimum weights of large error-correcting codes*, IEEE Transactions on Information Theory **34** (1988), 1354–1359. Citations in this document: §4.

[34] Tal Malkin (editor), *Topics in cryptology — CT-RSA 2008, the cryptographers' track at the RSA conference 2008, San Francisco, CA, USA, April 8–11, 2008, proceedings*, Lecture Notes in Computer Science, 4964, Springer, 2008. ISBN 978-3-540-79262-8. See [27].

[35] Stefan Mangard, François-Xavier Standaert (editors), *Cryptographic hardware and embedded systems, CHES 2010, 12th international workshop, Santa Barbara, CA, USA, August 17–20, 2010, proceedings*, Lecture Notes in Computer Science, 6225, Springer, 2010. ISBN 978-3-642-15030-2. See [4].

[36] Claire Mathieu (editor), *Proceedings of the twentieth annual ACM-SIAM symposium on discrete algorithms, SODA 2009, New York, NY, USA, January 4–6, 2009*, SIAM, 2009. See [37].

[37] Lorenz Minder, Alistair Sinclair, *The extended k-tree algorithm*, in SODA 2009 [36] (2009), 586–595. URL: http://www.cs.berkeley.edu/~sinclair/ktree.pdf. Citations in this document: §4.

[38] Kaisa Nyberg (editor), *Advances in cryptology — EUROCRYPT '98, international conference on the theory and application of cryptographic techniques, Espoo, Finland, May 31–June 4, 1998, proceedings*, Lecture Notes in Computer Science, 1403, Springer, 1998. ISBN 3-540-64518-7. See [5].

[39] Bimal Roy, Nicolas Sendrier (editors), *Progress in cryptology — INDOCRYPT 2009, 10th international conference on cryptology in India, New Delhi, India, December 13–16, 2009, proceedings*, Lecture Notes in Computer Science, 5922, Springer, 2009. ISBN 978-3-642-10627-9. See [9].

[40] Markku-Juhani Olavi Saarinen, *Linearization attacks against syndrome based hashes*, in Indocrypt 2007 [41] (2007), 1–9. Citations in this document: §1, §4, §4, §4, §4.

[41] Kannan Srinathan, C. Pandu Rangan, Moti Yung (editors), *Progress in cryptology — INDOCRYPT 2007, 8th international conference on cryptology in India, Chennai, India, December 9–13, 2007, proceedings*, Lecture Notes in Computer Science, 4859, Springer, 2007. ISBN 978-3-540-77025-1. See [40].

[42] Jacques Stern, *A method for finding codewords of small weight*, in [20] (1989), 106–113. Citations in this document: §4.

[43] Eran Tromer, Dag Arne Osvik, Adi Shamir, *Efficient cache attacks on AES, and countermeasures*, Journal of Cryptology **23** (2010), 37–71. URL: http://people.csail.mit.edu/tromer/papers/cache-joc-official.pdf. Citations in this document: §3.

[44] David Wagner, *A generalized birthday problem*, in CRYPTO 2002 [45] (2002), 288–304. URL: http://www.cs.berkeley.edu/~daw/papers/genbday.html. Citations in this document: §4.

[45] Moti Yung (editor), *Advances in cryptology — CRYPTO 2002, 22nd annual international cryptology conference, Santa Barbara, California, USA, August 18–22, 2002, proceedings*, Lecture Notes in Computer Science, 2442, Springer, Berlin, 2002. ISBN 3-540-44050-X. See [44].

[46] Albert L. Zobrist, *A new hashing method with application for game playing*, Technical Report 88, Computer Sciences Department, University of Wisconsin (1970). URL: https://www.cs.wisc.edu/techreports/1970/TR88.pdf. Citations in this document: §2, §3.