SHA-3 Proposal: FSB

Daniel Augot, Matthieu Finiasz, Philippe Gaborit,
Stéphane Manuel and Nicolas Sendrier

Contents

1 FSB Specifications

1.1 Overview

The FSB hash function we present uses the Merkle-Damgård domain extender [11, 18], with a large internal state and an additional final compression function (see Figure 1). It is thus based on a main compression function taking s input bits and outputting r bits. The message to hash is first split into blocks of size $s-r$ and some padding (including the message length) is added in order to obtain an integer number of blocks. The first block is compressed together with an IV of r bits to obtain the first r-bit chaining value. Then the second message block is compressed together with the first chaining value to obtain a second chaining value of r bits. This continues until the final block of message (the one containing the message length) is compressed to obtain what we will call the *pre-final hash*. This pre-final hash is then compressed using the final compression function to obtain the hash of the message.

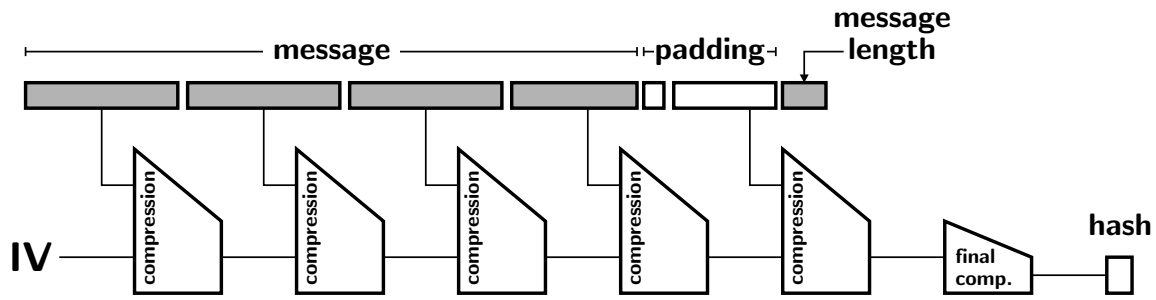

Figure 1: The Merkle-Damgård domain extender, with an additional final compression function.

The compression function of FSB uses coding theory techniques and computes the r bits syndrome of a binary word derived from the s input bits. In practice, the output of the compression function is simply the XOR of some cyclic shifts of a set of predefined vectors.

The padding is a standard Merkle-Damgård padding and for the final compression function we use a few rounds of Whirlpool.

We define 5 versions of FSB, each with a different output length and different compression function parameters. They are reported in Table 1.

	hash size	s	r
FSB_{160}	160 bits	1 120	640
FSB_{224}	224 bits	1 568	896
FSB_{256}	256 bits	1 792	1 024
FSB_{384}	384 bits	2 392	1 472
FSB_{512}	512 bits	3 224	1 984

Table 1: Parameters for the five versions of FSB.

1.2 Detailed Description

We now give a detailed description of each of the components of FSB. In this description, the first bit of a byte refers to the highest weight bit of this byte. The first byte of a table of bytes is the byte at position 0. Thus the first bit of a table of bytes is the highest weight bit of the byte at position 0.

1.2.1 Domain Extender

Padding. Suppose we want to hash a message of `databitlen` bits. The domain extender first splits this message in $\lfloor \frac{\texttt{databitlen}}{s-r} \rfloor$ full blocks of $s-r$ bits. The last block of message contains $\sigma = \texttt{databitlen} \mod (s-r)$ bits. This number of bits σ is between 0 and $s-r-1$ and a padding has to be applied. Two cases can occur:

- if $\sigma \leq s-r-65$ the padding can fit in the space remaining in this partially completed block. First a single bit "1" is appended to the message bits, then $s-r-\sigma-65$ bits "0" are appended (it is possible that no such "0" are added) leaving 64 bits in the block to append the value `databitlen` coded on 64 bits.
- if $\sigma > s-r-65$ the padding will not fit in the remaining space of the block. First a single bit "1" is appended to the message bits, then $s-r-\sigma$ bits "0" are appended (it is possible that no such "0" are added). This completes a $(s-r)$-bit block. Then a final block is appended to the message consisting of $s-r-64$ bits "0" and the value of `databitlen` coded on 64 bits.

The 64 last bits of the last block reserved for the value of `databitlen` are filled as follows: these 64 bits are seen as a table of 8 bytes, the first byte will contain the highest weight byte of the value of `databitlen`, the second block the second highest byte of `databitlen`, and so on down to the eighth byte which will contain the lowest weight byte of `databitlen`. If:

$$\texttt{databitlen} = b_0 + b_1 \times 2^8 + b_2 \times 2^{16} + ... + b_7 \times 2^{56}$$

where all the b_i are a byte of value between 0 and 255, then, the 64 for last bits of the last block will be the 8 bytes table:

$$[b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0].$$

Initial Value (IV). For all the versions of FSB, the IV is the all "0" r-bit vector.

Block Processing. After the padding, we obtain $\lceil \frac{\texttt{databitlen}+65}{s-r} \rceil$ blocks of $s-r$ bits. These blocks are processed in order starting from the first.

- The first $(s-r)$-bit block is concatenated with the r-bit IV to obtain s bits. These s bits are input to the compression function which outputs r bits. These r bits are used as a chaining value for the next message block.
- For each of the following blocks, the r-bit chaining value of the previous block is concatenated with the $s-r$ bits of message. Theses bits are input to the compression function to obtain the next round r-bit chaining value.
- When the last block (the one containing the value of `databitlen`) is compressed, the output r bits will constitute the pre-final hash of the message. This pre-final hash is input to the final compression function which outputs the hash of the message.

1.2.2 Compression Function

The compression function is defined by 4 parameters (n, w, r, p). From these parameters it is possible to compute its input size $s = w \times \log_2 \frac{n}{w}$. Table 2 below contains the values of n, w, r and p for the 5 versions of FSB.

This compression uses $b = \frac{n}{r}$ pre-defined p-bit vectors. The output is the XOR of w distinct cyclic shifts of some of these vectors, truncated to r bits.

	n	w	r	p	s
FSB_{160}	5×2^{18}	80	640	653	1 120
FSB_{224}	7×2^{18}	112	896	907	1 568
FSB_{256}	2^{21}	128	1 024	1 061	1 792
FSB_{384}	23×2^{16}	184	1 472	1 483	2 392
FSB_{512}	31×2^{16}	248	1 984	1 987	3 224

Table 2: Parameters for the five versions of FSB.

Vectors Definitions. For each FSB version, a set of $b = \frac{n}{r}$ vectors $(V_j)_{j\in[0,b-1]}$ of p bits is defined using the parity of the decimal digits of π. The first vector V_0 corresponds to the p first digits of π: if a digit is even, the vector contains a "0", if it is odd it contains a "1". For efficiency reasons, the first bit of a vector is always aligned with a digit of π at a position which is a multiple of 8. Therefore, there are some small gaps between the vectors. Figure 2 gives an example of how the vectors V_j are computed. Then these b vectors are converted to tables of bytes with the first bit of each vector going to the hight weight bit of the byte at position 0.

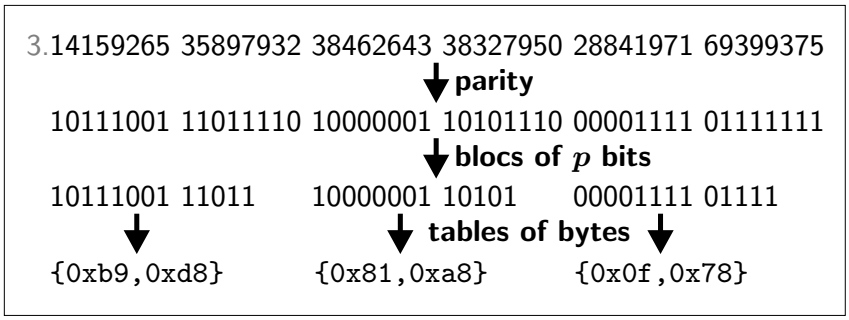

Figure 2: An example of how vectors are computed from the parity of digits of π. Here $p = 13$, so there are gaps of 3 bits between vectors.

These vectors need to be cyclically shifted to the right and truncated to r bits before the XORing step. This operation is a standard cyclic shift (to the right) on the p bit vector followed by a truncation to the r first bits. With the conversion choice to tables of bytes, this operation also corresponds to right shifts on the bytes[1]. Figure 3 gives an example of such a cyclic shift.

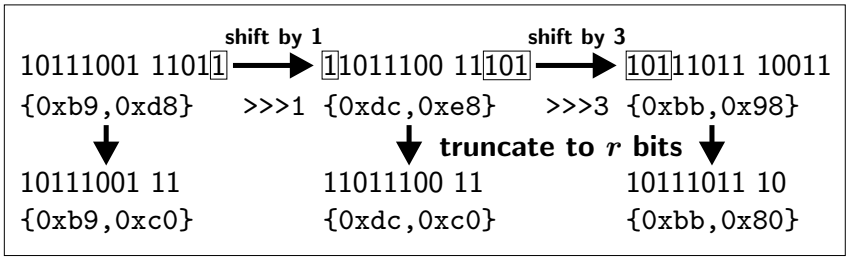

Figure 3: Cyclic shift to the right of a vector of p bits, followed by a truncation to r bits. Here $p = 13$ and $r = 10$.

Vector XORing. Given the s input bits of the function, we derive a set of w indexes $(W_i)_{i\in[0,w-1]}$ between 0 and $n-1$. The value of each W_i is computed from the inputs bits

[1]This is not a byte-wise cyclic shift, as what is shifted out of a byte comes inside the next byte

like this:

$$W_i = i \times \frac{n}{w} + \mathtt{IV}_i + \mathtt{M}_i \times 2^{\frac{r}{w}}.$$

Where $\mathtt{IV}_i$ and $\mathtt{M}_i$ are two indexes read from the r bits of IV (or chaining) and the $s-r$ bits of message computed as follows (see Figure 4):

- each $\mathtt{IV}_i$ is a value between 0 and $2^{\frac{r}{w}}-1$ derived from $\frac{r}{w}$ bits of IV (or chaining). The bits are read in order, starting from the beginning of the IV (first bytes of the table, highest weight first), the first $\frac{r}{w}$ bits define $\mathtt{IV}_0$, the next ones $\mathtt{IV}_1$, and so on. Again, for each $\mathtt{IV}_i$, the first of the $\frac{r}{w}$ bits will be the high weight bit and the last one the bit of lowest weight.
- similarly, $\mathtt{M}_i$ is a value between 0 and $2^{\frac{s-r}{w}}-1$ derived from $\frac{s-r}{w}$ bits of message. The bits are read exactly as for the IV.
- note that when computing W_i, the bits of IV are of lowest weight than those of message.

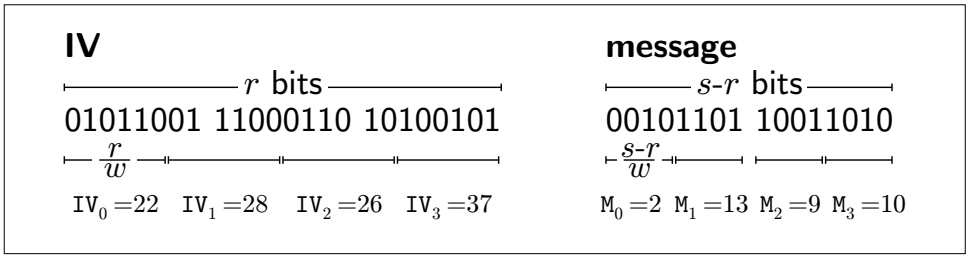

Figure 4: Computing the values $\mathtt{IV}_i$ and $\mathtt{M}_i$ from the IV and message bits. Here $w=4$, $\frac{r}{w}=6$ and $\frac{s-r}{w}=4$.

Once the W_i are computed, they are used to determine which of the $b=\frac{n}{r}$ vectors are XORed and by how much they should be shifted cyclically. Each W_i correspond to the vector number $\lfloor \frac{W_i}{r} \rfloor$ cyclically shifted to the right by $W_i \mod r$ positions. These vectors are then XORed to obtain the output h of the compression function. If we denote by V_j the j-th vector previously computed (from the digits of π), by $\mathtt{trunc}(\,\cdot\,, r)$ the truncation to r bits and by $>>>k$ the cyclic shift to the right by k positions, the output h is computed as:

$$h = \bigoplus_{i=0}^{w} \mathtt{trunc}\Big(V_{\lfloor \frac{W_i}{r} \rfloor} >>> (W_i \bmod r), r\Big).$$

Table 3 below gives the number of bits read from the IV and from the message for each index computation.

	bits per index s/w	bits from the IV r/w	bits from the message $(s-r)/w$
FSB_{160}	14	8	6
FSB_{224}	14	8	6
FSB_{256}	14	8	6
FSB_{384}	13	8	5
FSB_{512}	13	8	5

Table 3: Number of input bits per W_i index for the five versions of FSB.

1.2.3 Final Compression Function

At the last round of the previous compression function, a pre-final hash of r bits is output. This output is compressed to the desired length (`size` bits for $\mathsf{FSB}_{\mathtt{size}}$) by a final compression function. For this final compression function we use the Whirlpool hash function, designed by Baretto and Rijmen [20], which produces an output hash of 512 bits which we then truncate (we keep the first `size` bits) to the desired output length. We insist that the whole Whirlpool hash function is used (including the padding), not only the compression function of Whirlpool, as a single round of compression would not be enough for large values of r. For a detailed description of Whirlpool, please refer to the Whirlpool documentation available at http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html, or to the ISO/IEC 10118-3:2004 standard.

2 Design Rationale

The design of FSB is not new, it is an improvement of earlier published versions [1, 12, 13]. Some cryptanalysis work has also been published [10, 14, 21], bringing light on some weaknesses in previous versions. These attacks made it necessary to readjust some parameters but did not require a complete re-design of the system. The parameters we present were chosen so as to resist all currently known attacks.

2.1 Domain Extender

We use the Merkle-Damgård domain extender. It has been widely studied and, though some potential weaknesses have been pointed out, it is still believed to propagate the main security features from the compression function to the hash function. It will be enough in practice to establish our security claims.

Many other domain extenders are compatible with the FSB primitive. If needed we may use, for instance, the EMD domain extender [3] without significantly changing the algorithm.

2.2 Compression Function

The core of the compression function of FSB is the XOR of a set of binary vectors. If we compute all the possible vectors that can be XORed, that is, all the valid shifts of all the V_j vectors, we obtain n vectors that we can combine as columns of a single matrix H of size $r \times n$. The compression function of FSB then consists in generating a binary word of length n and Hamming weight w that we multiply by matrix H. From a coding theory point of view, the word of weight w can be seen as an error pattern, and the multiplication is the computation of the syndrome of this error. In Appendix A we consider the case where H is a random binary matrix and obtain tight security reductions for attacks on the compression function of FSB to well known coding theory problems.

In practice, H is not a random binary matrix but a *truncated quasi-cyclic matrix*, and only *regular words* are multiplied by H. The next two sections explain why these modifications do not weaken the security results.

2.2.1 Quasi-Cyclic Codes and Truncated Quasi-Cyclic Codes

Before showing why the use of a truncated quasi-cyclic matrix is not a threat for FSB we need to explain what are truncated quasi-cyclic matrices. We start with a few definitions.

Definition 1 *A matrix H of size $r \times r$ is* circulant *if it is obtained by cyclically right-shifting its first row (see Figure 5).*

Definition 2 *A matrix H of size $r \times n$ is* quasi-cyclic *if it is the concatenations of $\frac{n}{r}$ circulant matrices (see Figure 5).*

Definition 3 *A matrix H of size $r \times r$ is a* truncated quasi-cyclic *matrix if it is the concatenation of $\frac{n}{r}$ submatrices of size $r \times r$, and each of these submatrices is the top left submatrix of a $p \times p$ circulant matrix (see Figure 5).*

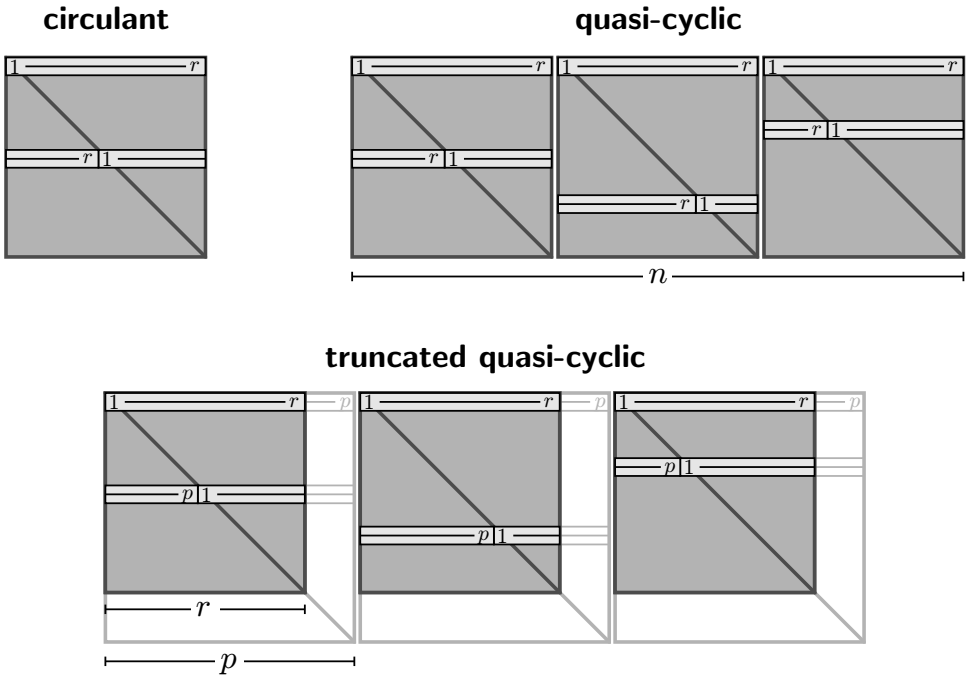

Figure 5: Structure of a circulant, a quasi-cyclic and a truncated quasi-cyclic matrix.

As one can easily check, the vector shifts used in FSB exactly correspond to choosing columns from a truncated quasi-cyclic matrix composed of $b = \frac{n}{r}$ blocks of size $p \times p$ truncated to $r \times r$. Also, these three kind of matrices have in common that they are completely defined by their first row. This is an important feature for FSB as the description of the hash function can thus be reduced by a factor r compared to using a truly random binary matrix. Also, this improves the speed of the hash function as the whole description of the matrix can now fit in the cache of a standard CPU.

In order to explain why truncated quasi-cyclic matrices are also a good choice for the security of FSB we have to look at them from a coding theory point of view.

Definition 4 *A linear code C of length $n = r \times b$ is a* quasi-cyclic code *of* order r *(and* index b*) if C has a parity-check matrix H which is a quasi-cyclic matrix composed of b circulant blocks of size $r \times r$.*

Gilbert-Varshamov Bound and Hardness of Decoding Quasi-Cyclic Codes. The Gilbert-Varshamov bound gives information about the best possible codes that exist for a given length and dimension. It is well known that codes built from a random binary $r \times n$ parity check matrix lie *almost always* on the Gilbert-Varshamov bound. This means that such codes are very

good codes (from the decoding capacity point of view) and in practice also means that they are necessarily hard to decode. This result has been generalized by Gaborit and Zémor [15] who have developed methods from which they prove that this result is also true for double-circulant codes (that is, quasi-cyclic codes of index 2). In fact they prove that double-circulant codes do even slightly better than random codes, moreover their method can also be generalized to general quasi-cyclic codes. This is in particular the case for codes of quasi-cyclic order p where p is prime and 2 is a generator of GF(p). This is how values of p have been selected for FSB.

It was proven in [5] that the syndrome decoding of random codes is NP-complete. It has recently been shown that this result is also valid for quasi-cyclic codes [4]. As a consequence, if one replaces the binary matrices of the CSD and CF problems presented in Appendix A by quasi-cyclic matrices, the two problems remain NP-hard, and the reductions to collision finding, inversion or second preimage are still valid.

Truncated Quasi-Cyclic Codes. In FSB, truncated quasi-cyclic codes are used instead of quasi-cyclic codes. Although these codes are not technically quasi-cyclic, they are closely related to them, since the number of rows and columns we truncate is small compared to the length of the matrix. Since both random codes and random quasi-cyclic codes lie on the Gilbert-Varshamov bound we assume that these codes also lie on the Gilbert-Varshamov bound. This was confirmed through experimental computation of minimum distances for such codes.

Concerning the hardness of decoding, since quasi-cyclic matrices are a special case of truncated quasi-cyclic matrices, decoding these codes is also an NP-complete problem. From a practical point of view, knowing that H is a truncated quasi-cyclic matrix does not help to improve the best decoding algorithms.

2.2.2 Regular Words

The way the indexes W_i (defining which vectors to use and how much they should be shifted) are chosen is not completely generic. If we look at the compression function as the product between a binary word of length n and Hamming weight w and an $r \times n$ matrix H, then the indexes W_i correspond to what we call *regular words*.

Definition 5 *A binary word of length n and Hamming weight w is* regular *if it is of weight exactly 1 on every interval of the form $[i \times \frac{n}{w}, (i+1) \times \frac{n}{w} - 1]$.*

This means that, contrarily to what is assumed in Appendix A, not all words of weight w and length n can be represented by the W_i. The reason for this choice is the huge gain in the speed of the conversion from s bits to a word of weight w. Using a one-to-one mapping able to code all the $\binom{n}{w}$ words of weight w would increase s thus gaining a few input bits and possibly decreasing the number of calls to the FSB compression function. However, such a one to one mapping requires to perform complex computations involving big integers whereas mapping regular words simply requires bit shifts. Tradeoffs also exist [19] but regular words are simple to describe, simple to implement and are what gives the best overall speed for FSB.

Concerning the security of regular words, restricting the set of input words of weight w cannot be a weakness. Indeed, if an adversary is able to build a collision (respectively, invert or find a second preimage) on the compression function of FSB using regular words, then this collision (respectively, inverse or second preimage) is also valid for the idealized version using all words of weight w. Therefore, if one is able to attack FSB using regular words, he will also be able to attack the idealized FSB using the exact same technique. Thus, the use of regular words cannot be a threat for the security of FSB.

In practice, attacking FSB with regular word is even harder than with a one-to-one mapping: having a smaller set of inputs decreases the number of possible solutions and thus, increases the cost of the attacks. However, taking this improvement into account could be a risk as it might be possible to fine-tune the GBA and ISD attacks to perform better than expected on regular words.

To remain on the safe side, we chose to use regular words for the speed gain they bring, but yet, to evaluate the security against GBA and ISD as if all $\binom{n}{w}$ words of weight w were available for collision search, inversion or second preimage.

2.3 Final Compression Function

FSB uses a large internal state of r bits and has to use a final compression function to obtain the desired output size. The main reason why this final compression function is necessary is that our compression function cannot achieve a security of $2^{\frac{r}{2}}$ against collision search or 2^r against inversion: if our compression function is to compress (that is, $s > r$), then the GBA attack will always have a complexity lower than $2^{\frac{r}{4}}$ for collision search.

Knowing that a final compression function was necessary we tried to precisely identify the requirements for this function:

- it has to be non-linear: a linear function would decrease the security of FSB as applying it to each columns of H would give a new matrix H' of size $r' \times n$ (this time with r' equal to the output size of the hash function) on which GBA could still be used for collision search or inversion. A linear function would thus decrease the security of FSB.

- being collision resistant is not necessary: if a collision is found on the final compression function, two pre-final hashes with the same hash are obtained. Finding a collision on the whole hash function still requires to find inverses for these pre-final hashes. Similarly, being hard to invert is not necessary.

This leaves us with a lot of liberty for the choice of the final compression function. We chose to use Whirlpool because it is highly non-linear, it behaves well when its output is truncated to a smaller size and people are confident that it has a good security. However, we must insist that if collisions are found on Whirlpool, this will not make collision finding on FSB any easier. Whirlpool must thus be seen more as a safe choice than as a practical need: much simpler non-linear functions would be suitable, but would probably also be considered as a possible weakness.

3 Practical Security

As seen in Section A.3, the best attacks against the compression function of FSB are well identified and it is possible to precisely evaluate their complexity. Table 4 gives the complexities of the different attacks we considered when selecting parameters for FSB.

As explained in the previous section, the Merkle-Damgård domain extender and the final compression function Whirlpool were chosen so as to preserve as much as possible the security of the FSB compression function. More precisely, noting CF the compression function and HF the whole hash function, we have:

- for collision resistance, Merkle and Damgård proved that b bits of security on CF yield b bits of security on HF.

- for inversion, b bits of security on CF also yield b bits of security on HF.

	collision search		inversion		second preimage	
	GBA$(n, 2w, r)$	ISD$(n, 2w, r)$	GBA(n, w, r)	ISD(n, w, r)	GBA$(n-w, w, r)$	ISD$(n-w, w, r)$
FSB_{160}	$2^{119.2}$	$2^{100.3}$	$2^{163.6}$	$2^{211.1}$	$2^{163.6}$	$2^{211.1}$
FSB_{224}	$2^{166.9}$	$2^{135.3}$	$2^{229.0}$	$2^{292.0}$	$2^{229.0}$	$2^{292.0}$
FSB_{256}	$2^{190.7}$	$2^{153.0}$	$2^{261.6}$	$2^{330.8}$	$2^{261.6}$	$2^{330.8}$
FSB_{384}	$2^{281.0}$	$2^{215.5}$	$2^{391.5}$	$2^{476.7}$	$2^{391.5}$	$2^{476.7}$
FSB_{512}	$2^{378.7}$	$2^{285.6}$	$2^{527.4}$	$2^{687.8}$	$2^{527.4}$	$2^{687.8}$

Table 4: Complexity of the best attacks known against the compression function of FSB.

- for second preimage, no such result exists. The best known attack is the herding attack by Kelsey and Kohno [17] which for a message of length 2^k reduces b bits of security on CF to $b-k$ bits of security on HF.

The complexities of the attacks on the FSB compression function reported in Table 4 can thus be transposed directly to the whole hash function and are all above the complexities of generic attacks on the whole FSB, which gives us the following security claim.

Security Claim for FSB

For $size \in \{160, 224, 256, 384, 512\}$ and a message of length at most 2^k, the best attacks on FSB_{size} are generic attacks that apply to any hash function with the following complexities:

collision:	$2^{\frac{size}{2}}$
inversion:	2^{size}
second preimage:	2^{size-k}

Toy Version of FSB. In addition to the 5 standard hash sizes we propose for FSB, our reference implementation also implements FSB_{48}, a toy version of FSB designed for practical cryptanalysis. This version uses the parameters $n = 3 \times 2^{17}$, $w = 24$, $r = 192$ and $p = 197$. These parameters were selected exactly as for the other sets of parameters: so as to obtain the highest throughput on our reference platform while maintaining the required security level. This toy version offers the following security levels:

	collision search		inversion		second preimage	
	GBA$(n, 2w, r)$	ISD$(n, 2w, r)$	GBA(n, w, r)	ISD(n, w, r)	GBA$(n-w, w, r)$	ISD$(n-w, w, r)$
FSB_{48}	$2^{35.8}$	$2^{37.4}$	$2^{49.2}$	$2^{73.9}$	$2^{49.2}$	$2^{73.9}$

Table 5: Complexity of the best attacks against the compression function of FSB_{48}.

Used within the Merkle-Damgård domain extender and once the final compression function applied, FSB_{48} will have security levels against collision, inversion and second preimage of respectively 24 bits, 48 bits and $48-k$ bits.

4 Computational Efficiency

4.1 General Considerations

The compression function of FSB is very simple: read the input bits by small blocks to compute some indexes and use these indexes to determine how much vectors have to be shifted before being XORed. The index computations can be done using only shifts and masking so that only

bitwise logical operations are used in the compression function. These operations are available on any kind of processor making the implementation of FSB very easy, whatever the hardware.

However, for efficiency, FSB has to use quite a lot of memory: all versions have to load about 2^{18} bytes, precomputed from the digits of π. This was chosen so as to fit well in the cache of modern CPUs, but is probably not the best choice for memory constrained environments. If this is a concern, other parameter choices could decrease the memory requirements without any loss of security, this would however change the KATs. Also note that in our optimized implementation the tables have been hard-coded in the program, avoiding to have to recompute them at startup. Due to this the program size is much larger than for the reference implementation which recomputes these tables at initialization. However, while the reference implementations takes several million cycles at startup, the optimized implementation behaves a lot better.

4.2 Efficiency on a Core2 Processor

All the tests we conducted were done using `gcc 4.1.2` (with the `-O3` flag) under 32-bit Linux on a 3GHz Core2 Duo processor with 6MB of cache and 4GB of memory. This platform should behave similarly to the NIST reference platform.

Table 6 below contains the results we obtained from the various tests we performed. The column "100 000 inits" gives the time taken to successively hash a 512-bit message block 100 000 times, similarly to what is done in the Monte Carlo test performed by `genKAT`. The timings obtain are very similar for the optimized and the reference implementations, except for the initialization time where the optimized version is much faster. Note that if initialization is much faster for FSB_{384} and FSB_{512} than for the other hash sizes it is because these two versions use $b = \frac{n}{r} = 1\,024$ instead of 2 048.

	100 000 inits	file hash time		init	hash speed
	time	1 MB	100 MB	cycles	cycles/byte
FSB_{160}	15.9s	0.09s	8.57s	$\sim 475\,000$	~ 257
FSB_{224}	15.9s	0.10s	9.89s	$\sim 475\,000$	~ 297
FSB_{256}	16.1s	0.11s	10.8s	$\sim 480\,000$	~ 324
FSB_{384}	9.6s	0.14s	14.1s	$\sim 290\,000$	~ 423
FSB_{512}	10.7s	0.17s	16.9s	$\sim 320\,000$	~ 507

Table 6: Performances of the 5 versions of FSB on a 3GHz Core2 Duo processor.

4.3 Efficiency on an 8-bit processor

We did not have any 8-bit processor at hand to test FSB in practice. Thus, we only present estimates of how FSB should behave on such processor. With our implementation, the most cycle-consuming operation in the compression function is the XORing of the w shifted vectors. As the shifts are precomputed we only have to perform the XORs which are responsible for approximately 80% of the cycles (the rest corresponding to the bit manipulations to obtain the indexes W_i). This XORing part will simply be 4 times longer on an 8-bit processor than on a 32-bit processor. However, 8-bit processors usually do not have a large cache and the shift will have to be recomputed : each XOR should thus be replaced by two shifts and two XORs, which should multiply the number of cycles by 4. Concerning the index computation, this part only manipulates small numbers of bits at a time, and it will probably not be more costly on 8-bit processors than on 32-bit processors. Summing up we get that 80% of the cycles will cost 16 times more and that 20% will cost the same. From N cycles per byte on a 32-bit architecture,

we increase to $13 \times N$ cycles per byte on an 8-bit architecture. However, this is only a very rough estimate and optimizations are certainly possible.

Concerning the initialization time, if no shift precomputations are performed then it is possible to directly read in the table of digits of π when hashing (here the little gaps between the vectors used to keep an 8-bit alignment are very useful) and thus, initialization time will probably be reduced to a few hundred cycles.

5 Advantages and Limitations

FSB has many advantages over traditional hash functions, in particular concerning security reductions, however this comes at a cost, and FSB also has a few limitations.

5.1 Limitations

- First of all, FSB is slower than the traditional MD5 or SHA-1. It uses much more cycles per byte to hash and also has an important initialization time (this will probably be improved through implementation optimizations).

- FSB has a large description: about 2 millions bits generated from digits of π are used. The parameters of FSB could be changed to improve this, but this would also change the hash values: this is a one time change. Another solution would be to introduce additional hash sizes with tweaked parameters so as to use less memory: for example, we could introduce hashes of 264 bits, with a much slower hash speed than for 256 bits, but with a smaller memory usage. FSB description will however always be much larger than for traditional hash functions.

- FSB uses a large internal state (about 4 times the final hash size) which makes it necessary to add a final compression function. Reducing the size of the internal state will reduce the security of the compression function, so this final compression is mandatory. Here, we use Whirlpool, but any other function could have been used: we need non-linearity but not necessarily collision resistance. If collisions are found on Whirlpool this will not affect the security of FSB.

- FSB still uses the Merkle-Damgård domain extender. This is probably not the best choice, but it is the simplest one. Any other domain extender preserving the collision resistance and inversion resistance of the compression function could be used.

5.2 Advantages

- The most decisive advantage of FSB is that it comes with a proof of reduction to hard algorithmic problems. An algorithm able to find collisions on FSB or to invert FSB is also able to solve hard problems from coding theory. These problems have been well studied for many years and it is unlikely that new attacks could be much more efficient than the existing ones. The parameters of FSB were chosen accordingly.

- The design of FSB is very simple: wether you see the compression function as the XOR of shifted vectors or the XOR of columns of a quasi-cyclic matrix, in both cases, describing FSB is simple. Having an easy to understand design helps when it comes to implementation.

- Also FSB uses mostly very simple operations: some shifts and some XORs. It is thus easy to obtain an optimized implementation spending a majority of time on operations that cannot be compressed.
- FSB is very flexible in terms of parameters: these have to be selected with caution in order to ensure all the security requirements, but proposing sets of parameters for all lengths from 160 to 512 bits would be possible. This could avoid having to use truncated hashes.
- The reference implementation of FSB we provide is relatively efficient (apart from the long initialization time) and is suitable for any set of parameters. A single implementation can be used for all versions of FSB.

A Security of the FSB Compression Function

A.1 Difficult computational problems from coding theory

The core of the FSB compression function is the syndrome primitive ($r < n$)

$$\begin{array}{rccc} S_H: & \{0,1\}^n & \longrightarrow & \{0,1\}^r \\ & y & \longmapsto & yH^T \end{array}$$

where H is a binary $r \times n$ matrix. We will call yH^T the H-syndrome of y. This mapping is linear and, for any element s in $\{0,1\}^r$, finding y in $\{0,1\}^n$ such that $yH^T = s$ is easy. If we restrict the problem to words y of Hamming weight (we denote $w_H(y)$) lower than some bound w the inversion becomes difficult for well chosen parameters n, r and w. In fact the following problem is NP-hard [5]

Problem 1 (Computational Syndrome Decoding) *Given a binary $r \times n$ matrix H, a word $s \in \{0,1\}^r$ and an integer $w > 0$, find a word e in $\{0,1\}^n$ of Hamming weight $\leq w$ such that $eH^T = s$.*

Moreover, decades of research in coding theory strongly suggest that the problem is hard in the average case [2]. Another related problem is the codeword finding problem

Problem 2 (Codeword Finding) *Given a binary $r \times n$ matrix H and an integer $w > 0$, find a non-zero word e in $\{0,1\}^n$ of Hamming weight $\leq w$ with an all zero H-syndrome.*

Obviously CF is a particular case of CSD. In practice the two problems can be tightly reduced to one another [5, 19].

The idealized one-way primitive used as compression function for FSB is the restriction of S_H to the set $W_{n,w}$ of binary words of length n and Hamming weight w:

$$\begin{array}{rccc} \mathcal{F}_{H,w}: & W_{n,w} & \longrightarrow & \{0,1\}^r \\ & e & \longmapsto & eH^T \end{array}$$

Hence the parameters are r, n and w and H a random $r \times n$ binary matrix. We will denote CSD(n, r, w) and CF(n, r, w) to refer to the above problems with specific sizes. All known algorithms for solving those problems have essentially the same complexity for given parameters. For a fixed matrix size $r \times n$, the typical cost for finding a word of weight w with a given H-

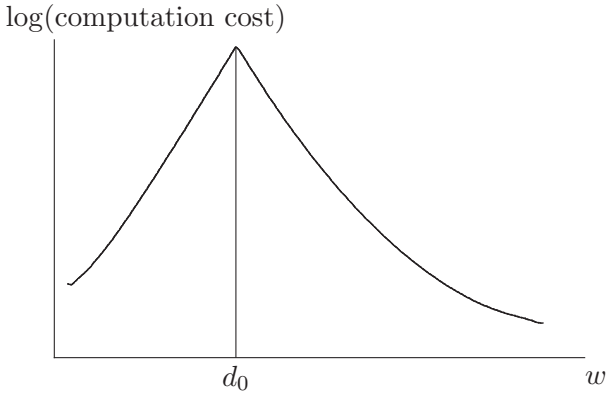

Figure 6: Cost (log scale) for finding words of weight w with a given H-syndrome.

syndrome (provided such a word exists) will vary as described in Figure 6. The value d_0 for which the cost is maximal is called the Gilbert-Varshamov distance and is defined as the largest integer such that

$$\sum_{i=0}^{d_0-1} \binom{n}{i} \leq 2^r.$$

We must have $w > d_0$ else the compression function provides no compression. The greater is w, the greater is the compression ratio (and the speed) and the lower is the security.

A.2 Security reduction

Let (n, r, w) be a set of parameters for the compression function.

On one hand, we define two adversaries related to the difficult computational problems of the previous section. Let us define a $\mathrm{CSD}(n, r, w)$ adversary as a mapping

$$\mathcal{A}_{\mathrm{CSD}} : \{0,1\}^{r \times n} \times \{0,1\}^r \to \{0,1\}^n.$$

It is successful on input (H, s) if $\mathcal{A}_{\mathrm{CSD}}(H, s) = e$ with $w_H(e) \leq w$ and $eH^T = s$. Similarly, a $\mathrm{CF}(n, r, w)$ adversary is a mapping

$$\mathcal{A}_{\mathrm{CF}} : \{0,1\}^{r \times n} \to \{0,1\}^n,$$

which is successful on input H if $\mathcal{A}_{\mathrm{CF}}(H) = e$ with $w_H(e) \leq w$ and $eH^T = 0$.

On the other hand we define three adversaries related to attacks on the compression function. We define

- A collision adversary is a mapping

  $$\mathcal{A}_c : \{0,1\}^{r \times n} \to W_{n,w} \times W_{n,w}.$$

  It is successful on input H if $\mathcal{A}_c(H) = (e, e')$ with $e \neq e'$ and $\mathcal{F}_{H,w}(e) = \mathcal{F}_{H,w}(e')$.

- An inversion adversary is a mapping

  $$\mathcal{A}_i : \{0,1\}^{r \times n} \times \{0,1\}^r \to W_{n,w}.$$

  It is successful on input (H, s) if $\mathcal{A}_i(H, s) = e$ with $\mathcal{F}_{H,w}(e) = s$.

- A second preimage adversary is a mapping

  $$\mathcal{A}_{2p} : \{0,1\}^{r \times n} \times W_{n,w} \to W_{n,w}.$$

  It is successful on input (H, e) if $\mathcal{A}_{2p}(H, e) = e'$ with $e \neq e'$ and $\mathcal{F}_{H,w}(e) = \mathcal{F}_{H,w}(e')$.

For any adversary $\mathcal{A}$, we will denote $|\mathcal{A}|$ its maximal running time. The purpose of the statements coming next in this section is to formally relate the existence of efficient adversary for inverting, finding collision or finding second preimage with the existence of efficient CF or CSD adversaries.

Proposition 1 *Let $\mathcal{A}_c$ be a collision adversary. For all $H \in \{0,1\}^{r \times n}$, we define a $\mathrm{CF}(n, r, 2w)$ adversary $\mathcal{A}$ as $\mathcal{A}(H) = e + e'$ where $\mathcal{A}_c(H) = (e, e')$. We have*

$$\begin{cases} |\mathcal{A}| = |\mathcal{A}_c| + O(n) \\ \forall H, \mathcal{A}_c(H) \text{ successful} \Rightarrow \mathcal{A}(H) \text{ successful.} \end{cases}$$

input: $H \in \{0,1\}^{(r-w)\times(n-w)}$, $w > 0$
$R \xleftarrow{r} \{0,1\}^{w\times n}$
$U \xleftarrow{r} \{0,1\}^{r\times r}$ non singular
$P \xleftarrow{r} n \times n$ permutation matrices
$H' \leftarrow U \left(\begin{array}{c|c} 0 & H \\ \hline R & \end{array} \right) P$
$e \leftarrow (1^w, 0^{n-w})P$
$e' \leftarrow \mathcal{A}_{2p}(H', e)$
$e'' \leftarrow$ the last $n-w$ bits of $(e+e')P^T$
return e''

Where $\mathcal{A}_{2p}$ is a second preimage adversary for the parameters (n, r, w) and $\xleftarrow{r}$ means "pick uniformly in".

Figure 7: CF$(n-w, r-w, w)$ adversary from second preimage adversary.

Proposition 2 *Let $\mathcal{A}_i$ be an inversion adversary. For all $(H, s) \in \{0,1\}^{r\times n} \times \{0,1\}^r$, we define the* CSD$(n, r, w)$ *adversary $\mathcal{A}$ as $\mathcal{A}(H, s) = \mathcal{A}_i(H, s)$. We have*

$$\begin{cases} |\mathcal{A}| = |\mathcal{A}_i| \\ \forall (H,s), \mathcal{A}_i(H,s) \text{ successful} \Leftrightarrow \mathcal{A}(H,s) \text{ successful.} \end{cases}$$

Proposition 3 *Let $\mathcal{A}$ be the* CF$(n-w, r-w, w)$ *adversary derived from the second preimage adversary $\mathcal{A}_{2p}$ as defined in Figure 7. We have $|\mathcal{A}| = |\mathcal{A}_{2p}| + O(r^2 n)$ and*

$$\begin{aligned} &\text{Prob}\left(\mathcal{A}(H) \text{ successful} \mid H \xleftarrow{r} \{0,1\}^{(r-w)\times(n-w)}\right) \\ &\quad \geq \text{Prob}\left(\mathcal{A}_{2p}(H,e) \text{ successful} \mid H \xleftarrow{r} \{0,1\}^{r\times n}, e \xleftarrow{r} W_{n,w}\right). \end{aligned}$$

Proof: We use the notations of Figure 7.

$$U^{-1}H'P^T = \begin{array}{|c|c|} \hline 0 & H \\ \hline \multicolumn{2}{|c|}{R} \\ \hline \end{array}$$

$$eP^T = \begin{array}{|ccc|ccc|} \hline 1 & \cdots & 1 & 0 & \cdots & 0 \\ \hline \end{array}$$

When H, R, U, P are independently and uniformly chosen then H' and e are independently and uniformly distributed.

If the call $\mathcal{A}_{2p}(H', e)$ is successful, then $e + e'$ is a codeword of the code $\mathcal{C}'$ of parity check matrix H'. Thus we have $(e+e')H'^T = 0$ and $(e+e')P^T(U^{-1}H'P^T)^T = 0$ and, if e'' is defined as in the algorithm, we have $e''H^T = 0$. Moreover the Hamming weight of e'' is less or equal to w because it cannot exceed the weight of e'. The adversary $\mathcal{A}$ is thus successful. This proves the inequality of the probabilities.

Finally, if we exclude the call to $\mathcal{A}_{2p}$, the computations in $\mathcal{A}$ are limited to $O(r^2 n)$ (cost of the matrix multiplication by U). ◇

A.3 Best known attacks

In practice, solving CSD(n, r, w) or CF(n, r, w) requires a comparable computation effort. When the weight w varies for fixed (n, r), the best decoding (or codeword finding) technique will be one of three algorithms: information set decoding, generalized birthday attack [23], or linearization attack [21].

Information set decoding is always the most efficient method when $w \leq d_0$ (see Figure 6). The linearization attack is devastating for $w \geq r$ and remains possible with increasing efficiency in the range $r/2 < w < r$. Below this value, the information set decoding or the generalized birthday attack is more efficient.

Following the previous section, the FSB compression function must resist decoding and/or codeword finding attacks for the weights w and $2w$. The parameters we propose are such that $2w = r/4$, which is out of reach of the linearization attack. This means that we will only consider information set decoding and generalized birthday attack. In the considered range, we are not aware of any attack on the FSB primitive that would be more efficient.

We will first give a lower bound on the cost of those two attacks. Then we will consider how they can be used to threaten our compression function.

A.3.1 Decoding attacks

Information set decoding. Information set decoding belongs to the folklore of coding theory. The version we consider here is due to Stern [22]. The algorithm has been first improved and implemented in [8] and again more recently in [6]. A quick survey can be found in [19]. The algorithm iterates the two following steps

1. Permute the columns of H and perform a Gaussian elimination to get H'.
2. Find a set of w columns of H' which add to a prescribed value (possibly zero).

The second step above is rather involved and depends on two positive integers p and ℓ which have to be optimized for every set of parameters (n, r, w). For a fixed triple (n, r, w) let us define the following quantities depending on p and ℓ

$$\begin{aligned}
L(p) &= \binom{(n-r)/2}{p}, \\
N(p,\ell) &= \frac{2^r}{L(p)^2 \binom{r-l}{w-2p}}, \\
L'(p,\ell) &= L(p)\sqrt{N(p,\ell)}, \\
C(p,\ell,X) &= 2X\left(\log_2(X) + \ell\right) + \frac{X^2}{2^\ell} 4p(w-2p), \\
K(p,\ell) &= \begin{cases} N(p,\ell)C(p,\ell,L(p)) & \text{if } N(p,\ell) > 1 \\ C(p,\ell,L'(p,\ell)) & \text{else.} \end{cases}
\end{aligned}$$

For w greater than the Gilbert-Varshamov distance (in fact $\binom{n}{w} \geq 2^r$ which is almost the same thing), the binary workfactor of any known variant of the information set decoder is lower bounded by

$$WF(n, r, w) = \min_{p,\ell} K(p, \ell).$$

If w is smaller than the Gilbert-Varshamov distance (which is of no interest here), we would have to replace 2^r by $\binom{n}{w}$ in the formula giving $N(p, \ell)$. The value $N(p, \ell)$ is the expected number of iterations of the algorithm. The cost of one iteration is $C(p, \ell, L(p))$. When $N(p, \ell) \leq 1$ (we

expect to succeed at the first iteration) the cost of the unique iteration is smaller (we replace $L(p)$ by $L'(p,\ell)$ which is smaller).

Generalized birthday attack. The generalized birthday algorithm (GBA) was presented by Wagner [23] and is a generalization of the well known birthday technique. It was first applied to decoding problems by Coron and Joux [10] and had already been introduced in a few other articles before [7, 9, 16]. Suppose you have a set of binary vectors of length r and you want to find a subset of w vectors XORing to 0. With the standard birthday technique, one would build two lists of L elements resulting from the XOR of $\frac{w}{2}$ vectors, sort the first of these lists and lookup the elements of the second list in the first one. Each time a match is found, a valid subset of w vectors is also found. If the size L of the lists is such that $L^2 \geq 2^r$, then matches will probably be found. This gives a solution to the problem with complexity $O(r \times 2^{\frac{r}{2}})$.

The idea behind GBA is to use 2^a lists instead of only 2, where the value of the parameter a will depend on the size of the set of vectors. GBA works in a steps, and at each step lists are merged pairwise so as to divide the number of lists by 2. One starts with 2^a lists of L elements resulting from the XOR of $\frac{w}{2^a}$ vectors. Then at first step, these lists are grouped by pairs and each pair is merged into a single list by XORing one element of the first list with one of the second. However, only the XORs starting with $\log_2 L$ zeros are kept: this will discard some possible valid solutions but this will, in average, maintain the size of the lists constant, keeping the complexity of the algorithm low. At the end of the first step one has 2^{a-1} lists of L elements resulting of the XOR of $\frac{w}{2^{a-1}}$ vectors, all starting with $\log_2 L$ zeros. The second step does exactly the same so as to obtain 2^{a-2} lists of vectors starting with $2\log_2 L$ zeros, and so on, until only 2 lists are left. At this point a simple application of the standard birthday technique tells us that if $L^2 \geq 2^{r-(a-1)\log_2 L}$, then a match will probably be found. This gives us the minimal value of L for the algorithm to work: $L \geq 2^{\frac{r}{a+1}}$. For $a = 1$ we have the standard birthday algorithm result, for a larger, the complexity can be greatly improved, as long as enough vectors are available to build sufficiently large lists. If it is possible to build lists of size $2^{\frac{r}{a+1}}$, then the algorithm has a total complexity of $O(r \times 2^{\frac{r}{a+1}})$.

This result gives us an algorithm to solve both CF and CSD with a good complexity when a lot of solutions exist: for CF this is straight-forward, for CSD one needs first to XOR the target syndrome with all the elements of one of the starting lists and then the algorithm can be applied. For collision search, one will thus have to find null combinations of $2w$ vectors of r bits starting from a set of n vectors, for inversion one has to find null combinations of w vectors.

Practical Complexity of GBA. When applying GBA, one starts with lists of size L containing XORs of $\frac{w}{2^a}$ columns of H. The size of the lists must verify the bound $\binom{L}{2^a} \leq \binom{n}{w}$ (any combination of w columns of H can be seen as a combination of 2^a combinations of $\frac{w}{2^a}$ columns of H). Also, for GBA to apply, the lists must contain enough elements, that is: $L \geq 2^{\frac{r}{a+1}}$. We obtain the following bound on the value of parameter a:

$$\binom{2^{\frac{r}{a+1}}}{2^a} \leq \binom{n}{w} \quad \overset{a \text{ small}}{\Longleftrightarrow} \quad \frac{r}{a+1} \leq \frac{\log_2 \binom{n}{w}}{2^a} \quad \Longleftrightarrow \quad \frac{2^a}{a+1} \leq \frac{\log_2 \binom{n}{w}}{r}. \tag{1}$$

The complexity of GBA is thus $O(r \times 2^{\frac{r}{a+1}})$ for the maximum value of a verifying equation (1).

Using Non-Integer Values of Parameter a. Normally, GBA only applies to integer values of a, leading to discontinuities in its complexity when w varies. In practice, it is possible to remove these discontinuities by modifying the algorithm a little: if one can build larger lists than required, then it is possible to start with lists of vectors all starting by a few zeros, thus slightly decreasing the value of r and the complexity of the attack. This improvement almost correspond to being able to use non-integer values of a. When computing the complexity of

GBA we solve equation (1) for the maximum value of a and consider the complexity to be $2^{\frac{r}{a+1}}$. Obtaining this complexity with the generalized birthday algorithm might not be possible, but this is a safe bound.

We will denote respectively $\mathrm{ISD}(n, r, w)$ and $\mathrm{GBA}(n, r, w)$ to refer to an information set decoder or a generalized birthday algorithm for a given set of parameters (n, r, w).

A.3.2 Best attacks on the FSB compression function

Collision resistance. The Proposition 1 states that finding a collision for the compression function $\mathcal{F}_{H,w}$ is at least as difficult as finding a codeword of weight $\leq 2w$. Conversely, we may build a collision adversary from any $\mathrm{CF}(n, r, 2w)$ adversary.

Proposition 4 *Let $\mathcal{A}_{\mathrm{CF}}$ be a $\mathrm{CF}(n, r, 2w)$ adversary returning words of weight $2w$ exactly. For all $H \in \{0,1\}^{r \times n}$, we define the collision adversary $\mathcal{A}$ as $\mathcal{A}(H) = (e, e')$ where e and e' are two words of weight w such that $e + e' = \mathcal{A}_{\mathrm{CF}}(H)$. We have*

$$\begin{cases} |\mathcal{A}| = |\mathcal{A}_{\mathrm{CF}}| + O(n) \\ \forall H, \mathcal{A}_{\mathrm{CF}}(H) \text{ successful} \Rightarrow \mathcal{A}(H) \text{ successful} \end{cases}$$

The CF adversary in the above statement is slightly stronger as we require it to return errors of weight $= 2w$ instead of just $\leq 2w$. In practice, it won't be a problem because all the decoding technique we consider have this feature. For all the proposed parameters, the best algorithm for finding codeword of weight $2w$ is always information set decoding.

Preimage resistance. The Proposition 2 states that inverting the compression function $\mathcal{F}_{H,w}$ for some $s \in \{0,1\}^r$ is at least as difficult as solving the syndrome decoding problem with input (H, s, w). Conversely we have

Proposition 5 *Let $\mathcal{A}_{\mathrm{CSD}}$ be a $\mathrm{CSD}(n, r, w)$ adversary returning words of weight w exactly. For all $(H, s) \in \{0,1\}^{r \times n} \times \{0,1\}^r$, we define the inversion adversary $\mathcal{A}$ as $\mathcal{A}(H, s) = \mathcal{A}_{\mathrm{CSD}}(H, s)$. We have*

$$\begin{cases} |\mathcal{A}| = |\mathcal{A}_i| \\ \forall (H, s), \mathcal{A}_i(H, s) \text{ successful} \Leftrightarrow \mathcal{A}(H, s) \text{ successful} \end{cases}$$

For all the proposed parameters, the best algorithm for decoding w errors is always the generalized birthday attack.

Second preimage resistance. The Proposition 3 states that finding a second preimage for the compression function $\mathcal{F}_{H,w}$ and some $e \in W_{n,w}$ is at least as difficult as finding a codeword of weight $\leq w$ in a code whose parity check matrix is a $(r - w) \times (n - w)$ submatrix of H. Conversely, we can also build a second preimage adversary from a CF adversary.

Proposition 6 *Let $\mathcal{A}_{\mathrm{CSD}}$ be a $\mathrm{CSD}(n-w, r, w)$ adversary returning words of weight w exactly. For all $(H, e) \in \{0,1\}^{r \times n} \times W_{n,w}$, we define the second preimage adversary $\mathcal{A}$ as $\mathcal{A}(H, e) = \mathcal{A}_{\mathrm{CSD}}(H', eH^T)$ where H' is the $r \times (n - w)$ submatrix of H obtained by removing the columns whose index are in the support of e. We have*

$$\begin{cases} |\mathcal{A}| = |\mathcal{A}_{\mathrm{CSD}}| + O(rw) \\ \forall (H, e), \mathcal{A}_{\mathrm{CSD}}(H', eH^T) \text{ successful} \Rightarrow \mathcal{A}(H', e) \text{ successful} \end{cases}$$

As for preimage, with all the proposed parameters, the most efficient attack is the generalized birthday attack.

A.3.3 Practical security of the FSB primitive

All the security features are summarized in the Table 7. It is interesting to see that the security reduction for both collision and inversion is very tight. In both cases, the best attack solves precisely the problem to which the security is reduced.

Collision resistance :	Best attack	$\mathrm{ISD}(n, r, 2w)$
	Security reduction	$\mathrm{CF}(n, r, 2w)$
Preimage resistance :	Best attack	$\mathrm{GBA}(n, r, w)$
	Security reduction	$\mathrm{CSD}(n, r, w)$
2^{nd} preimage resistance :	Best attack	$\mathrm{GBA}(n-w, r, w)$
	Security reduction	$\mathrm{CF}(n-w, r-w, w)$

Table 7: Security of the FSB compression function

For second preimage resistance, the situation is slightly different. There is a small gap between the best second preimage attack and the problem to which the security is reduced.

In practice, we will use $\mathrm{GBA}(n-w, r, w)$ to measure the security against second preimage attacks. Because $w \ll n$ and the parameter n has a relatively small impact on the decoding complexity, the workfactor for $\mathrm{GBA}(n-w, r, w)$ is almost the same a the workfactor for $\mathrm{GBA}(n, r, w)$ which is used for preimage resistance.

Finally, note that in the unlikely event that a 2^{nd} preimage attack based on $\mathrm{GBA}(n-w, r-w, w)$ can be found, the security will not decrease a lot. We use $w = r/8$ and the number of bits of security is roughly proportional to r. Instead of b bits of security, we would "only" have $\frac{7}{8}b$ bits of security.

References

[1] D. Augot, M. Finiasz, and N. Sendrier. A family of fast syndrome based cryptographic hash functions. In E. Dawson and S. Vaudenay, editors, *Mycrypt 2005*, volume 3715 of *Lecture Notes in Computer Science*, pages 64–83. Springer, 2005.

[2] A. Barg. Complexity issues in coding theory. In V. S. Pless and W. C. Huffman, editors, *Handbook of Coding theory*, volume I, chapter 7, pages 649–754. North-Holland, 1998.

[3] M. Bellare and T. Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. In X. Lai and K. Chen, editors, *Advances in Cryptology - Asiacrypt 2006*, number 4284 in Lecture Notes in Computer Science, pages 299–314. Springer, 2006.

[4] T. Berger, P-L. Cayrel, P. Gaborit, and A. Otmani. Reducing key length of the McEliece cryptosystem. preprint, available at http://www.unilim.fr/pages_perso/philippe.gaborit/reducing.pdf, 2008.

[5] E. R. Berlekamp, R. J. McEliece, and H. C. van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3), May 1978.

[6] D.J. Bernstein, T. Lange, and C. Peters. Attacking and defending the McEliece cryptosystem. In J. Buchmann and J. Ding, editors, *PQCrypto 2008*, volume 5299 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2008.

[7] P. Camion and J. Patarin. The knapsack hash function proposed at crypto'89 can be broken. In D.W. Davies, editor, *Advances in Cryptology - Eurocrypt 91*, volume 547 of *Lecture Notes in Computer Science*. Springer, 1991.

[8] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: Application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, January 1998.

[9] P. Chose, A. Joux, and M. Mitton. Fast correlation attacks: An algorithmic point of view. In L.R. Knudsen, editor, *Advances in Cryptology - Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 209–221. Springer, 2002.

[10] J.-S. Coron and A. Joux. Cryptanalysis of a provably secure cryptographic hash function. IACR eprint archive `http://eprint.iacr.org/2004/013`, 2004.

[11] I.B. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology - Crypto 89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–426. Springer, 1989.

[12] M. Finiasz. Syndrome based collision resistant hashing. In J. Buchmann and J. Ding, editors, *PQCrypto 2008*, volume 5299 of *Lecture Notes in Computer Science*, pages 137–147. Springer, 2008.

[13] M. Finiasz, P. Gaborit, and N. Sendrier. Improved fast syndrome based cryptographic hash functions. In V. Rijmen, editor, *ECRYPT Workshop on Hash Functions*, 2007.

[14] P.-A. Fouque and G. Leurent. Cryptanalysis of a hash function based on quasi-cyclic codes. In T. Malkin, editor, *CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2008.

[15] P. Gaborit and G. Zémor. Asymptotic improvement of the Gilbert-Varshamov bound for linear codes. *IEEE Transactions on Information Theory*, 54(9):3865–3872, 2008.

[16] A. Joux and R. Lercier. "Chinese & Match", an alternative to Atkin's "Match and Sort" method used in the SEA algorithm. *Math. Comput.*, 70(234):827–836, 2001.

[17] J. Kelsey and T. Kohno. Herding hash functions and the Nostradamus attack. In S. Vaudenay, editor, *Advances in Cryptology - Eurocrypt 2006*, number 4004 in Lecture Notes in Computer Science, pages 183–200. Springer, 2006.

[18] R. C. Merkle. One way hash functions and DES. In G. Brassard, editor, *Advances in Cryptology - Crypto 89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[19] R. Overbeck and N. Sendrier. Codes-based cryptography. In D. Bernstein, J. Buchmann, and J. Ding, editors, *Post-Quantum Cryptography*, pages 95–145. Springer, 2008.

[20] V. Rijmen and P. Barreto. Whirlpool. Seventh hash-function of ISO/IEC 10118-3:2004, 2004.

[21] M-J.O. Saarinen. Linearization attacks against syndrome based hashes. In M. Yung, K. Srinathan, and C. Pandu Rangan, editors, *Indocrypt 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2007.

[22] J. Stern. A method for finding codewords of small weight. In G. Cohen and J. Wolfmann, editors, *Coding theory and applications*, volume 388 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 1989.

[23] D. Wagner. A generalized birthday problem. In M. Yung, editor, *Crypto 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–304. Springer, 2002.