# Linearization Attacks Against Syndrome Based Hashes

**(Draft Version of July 31, 2007)**

Markku-Juhani O. Saarinen

Information Security Group
Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK.
m.saarinen@rhul.ac.uk

**Abstract.** In MyCrypt 2005, Augot, Finiasz, and Sendrier proposed FSB, a family of cryptographic hash functions. The security claim of the FSB hashes is based on a coding theory problem with hard average-case complexity. In ECRYPT 2007 Hash Function Workshop, new versions with essentially the same compression function but radically different security parameters and an additional final transformation were presented. We show that hardness of average-case complexity of the underlying problem is irrelevant in collision search by presenting a linearization method that can be used to produce collisions in a matter of seconds on a desktop PC for the variant of FSB with claimed $2^{128}$ security.

## 1 Introduction

A number of hash functions have been proposed that are based on "hard problems" from various branches of computer science. Recent proposals in this genre of hash function design include VSH (factoring) [3], LASH (lattice problems) [2], and the topic of this paper, Fast Syndrome Based Hash (FSB), which is based on decoding problems in the theory of error-correcting codes [1, 5].

In comparison to dedicated hash functions designed using symmetric cryptanalysis techniques, "provably secure" hash functions tend to be relatively slow and do not always meet all of criteria traditionally expected of cryptographic hashes. An example of this is VSH, where only collision resistance is claimed, leaving the hash open to various other attacks [7].

Another feature of "provably secure" hash functions is that the proof is often a reduction to a problem with asymptotically hard worst-case or average-case complexity. Worst-case complexity measures the difficulty of solving pathological cases rather than typical cases of the underlying problem. Even a reduction to a problem with hard average complexity, as is the case with FSB, offers only limited security assurance as there still can be an algorithm that easily solves the problem for a subset of the problem space.

This common pitfall of provably secure cryptographic primitives is clearly demonstrated in this paper for FSB – it is shown that the hash function offers minimal preimage or collision resistance when the message space is chosen in a specific way.

## 2 The FSB Compression Function

The FSB compression function can be described as follows [1, 5] .

**Definition 1.** *Let $\mathcal{H}$ be an $r \times n$ binary matrix. The FSB compression function is a mapping from message vector* $\mathbf{s}$ *that contains $w$ words, each satisfying $0 \leq s_i < \frac{n}{w}$, to an $r$ bit result as follows:*

$$\mathrm{FSB}(\mathbf{s}) = \bigoplus_{i=1}^{w} \mathcal{H}_{(i-1)\frac{n}{w}+s_i+1} \ ,$$

*where $\mathcal{H}_i$ denotes column $i$ of the matrix.*

The FSB compression function is operated in Merkle–Damgård mode to process a large message. The exact details of padding and chaining of internal state across compression function iterations are not specified. [1]

With most proposed variants of FSB, the word size $\frac{n}{w}$ is chosen to be $2^8$, so that $\mathbf{s}$ can be treated as an array of as bytes for practical implementation purposes. See Appendix A for an implementation example.

For the purposes of this paper, we shall concentrate on finding collisions and pre-images in the compression function. These techniques can be easily applied for finding full collisions of the hash function. The choice of $\mathcal{H}$ is taken to be a random binary binary matrix in this paper, although quasi-cyclic matrices are considered in [5] to reduce memory usage.

The final transformation proposed in the [5] does not affect the complexity of finding collisions or second pre-images, although it makes first pre-image search difficult (equal to inverting Whirlpool [8]). Second pre-images can be easily found despite a strong final transform.

The security parameter selection of FSB is based primarily on Wagner's generalized birthday attack [9]. The security claims are summarized in Table 1.

## 3 Basic Linearization Attack

To illustrate our main attack technique, we shall first consider hashes of messages where the words in the message only have binary values: $m_i \in \{0, 1\}$ for all $i$. This is a small subset of all possible messages. Let $\mathbf{m}$ be a binary vector representing the message.

We define a constant vector $\mathbf{c}$,

$$\mathbf{c} = \bigoplus_{i=1}^{w} \mathcal{H}_{(i-1)\frac{n}{w}+1},$$

and an auxiliary $r \times w$ binary matrix $\mathbf{A}$, whose columns $\mathbf{A}_i$, $1 \leq i \leq w$ are given by

$$\mathbf{A}_i = \mathcal{H}_{(i-1)\frac{n}{w}+1} \oplus \mathcal{H}_{(i-1)\frac{n}{w}+2}.$$

[1] Ambiguous definitions of algorithms makes experimental cryptanalytic work depend on guesswork on algorithm details. However, the attacks outlined in this paper should work, regardless of the particular details of chaining and padding.

| Security | $r$ | $w$ | $n$ | $s$ | $n/w$ |
|---|---|---|---|---|---|
| 64-bit | 512 | 512 | 131072 | 4096 | 256 |
| | 512 | 450 | 230400 | 4050 | 512 |
| | 1024 | $2^{17}$ | $2^{25}$ | $2^{20}$ | 256 |
| 80-bit | 512 | 170 | 43520 | 1360 | 256 |
| | 512 | 144 | 73728 | 1296 | 512 |
| 128-bit | **1024** | **1024** | **262144** | **8192** | **256** |
| | 1024 | 904 | 462848 | 8136 | 512 |
| | 1024 | 816 | 835584 | 8160 | 1024 |

**Table 1.** Parameterizations of FSB, as given in [5]. Line 6 (in bold) with claimed $2^{128}$ security was proposed for practical use. Pre-images and collisions can be found for this variant in a matter of seconds on a desktop PC.

By considering how the XOR operations cancel each other out, it is easy to see that for messages of this particular type the FSB compression function is entirely linear:

$$\mathrm{FSB}(\mathbf{m}) = \mathbf{A} \cdot \mathbf{s} \oplus \mathbf{c}.$$

Furthermore, let us consider the case where $r = w$, and therefore $\mathbf{A}$ is a square matrix. If $\det \mathbf{A} \neq 0$ the inverse exists and we are able to find a pre-image $\mathbf{m}$ from the Hash $\mathbf{h} = \mathrm{FSB}(\mathbf{m})$ simply as

$$\mathbf{s} = \mathbf{A}^{-1} \cdot (\mathbf{h} \oplus \mathbf{c}).$$

If $r$ is greater than $w$, the technique can still be applied to force given $w$ bits of the final hash to some predefined value. Since the order of the rows is not relevant, we can simply construct a matrix that contains only the given $w$ rows (i.e.. bits of the hash function result) of $\mathbf{A}$ that we are are interested in.

## 4 The Selection of Alphabet

We note that the selection of $\{0, 1\}$ as the set of allowable message words ("the alphabet") is arbitrary. We can simply choose any pair of word values for each $i$ so that $m_i \in \{x_i, y_i\}$ and map each $x_i$ to 0 and each $y_i$ to 1, thus creating a binary vector for the attack.

The constant is then given by

$$\mathbf{c} = \bigoplus_{i=1}^{w} \mathcal{H}_{(i-1)\frac{n}{w}+x_i},$$

and columns of the $\mathbf{A}$ matrix are given by

$$\mathbf{A}_i = \mathcal{H}_{(i-1)\frac{n}{w}+x_i+1} \oplus \mathcal{H}_{(i-1)\frac{n}{w}+y_i+1}.$$

To invert a hash $\mathbf{h}$ we first compute

$$\mathbf{b} = \mathbf{A}^{-1}(\mathbf{h} \oplus \mathbf{c})$$

and then apply the mapping $m_i = x_i + b_i(y_i - x_i)$ on the binary result $\mathbf{b}$ to obtain the correct message $\mathbf{s}$ (in the given alphabet) that produces $\mathbf{h}$.

The binary matrices are essentially random for each arbitrarily chosen alphabet. Since the success of a pre-image attack depends upon the invertibility of the binary matrix $\mathbf{A}$, we note that the probability that an $n \times n$ random binary matrix has non-zero determinant and is therefore invertible in $\mathtt{GF}(2)$ is given by

$$p = \prod_{i=1}^{n}(1 - 2^{-i}) \approx 0.28879$$

when $n$ is even moderately large.

Two trials with two distinct alphabets are on the average enogh to find an invertible matrix (total probability for 2 trials is $1 - (1 - p)^2 \approx 0.49418$).

## 5 Finding collisions when $r = 2w$

We shall expand our approach for producing collisions in $2w$ bits of the hash function result by controlling $w$ message words. This is twice the number compared to pre-image attack. The complexity of the attack remains essentially the same.

Assume that (by selection of alphabets) there are two distinct linear presentations for FSB, one containing the matrix $\mathbf{A}$ and constant $\mathbf{c}$ and the other one $\mathbf{A}'$ and $\mathbf{c}'$ correspondingly. To find a pair of messages $\mathbf{s}$, $\mathbf{s}'$ that produces a collision we must find a solution to the equation

$$\mathbf{A} \cdot \mathbf{s} \oplus \mathbf{c} = \mathbf{A}' \cdot \mathbf{s}' \oplus \mathbf{c}'.$$

It is easy to see that this can be manipulated to form

$$\left(\mathbf{A} \mid \mathbf{A}'\right) \cdot \left(\frac{\mathbf{s}}{\mathbf{s}'}\right) = \left(\frac{\mathbf{c}}{\mathbf{c}'}\right).$$

The solution of the inverse $(\mathbf{A} \mid \mathbf{A}')^{-1}$ will allow us to compute the message pair $(\mathbf{s} \mid \mathbf{s}')^T$ that yields the same hash in $2w$ different message bits (since $r = 2w$ yields a square matrix in this case).

Collisions for all "128-bit security" variants in Table 1 can be easily produced in this way, despite the details of chaining and final transformation.

## 6 Conclusions

We have shown that Fast Syndrome Based Hashes (FSB) are not secure against pre-image or collision attacks under the proposed security parameters. Collisions for a variant with claimed 128-bit security can be found in less than a second on a low-end PC.

We feel that the claim of "provable security" is hollow in the case of FSB, where the security proof is based on a problem with hard average-case complexity, but which is almost trivially solvable for special classes of messages.

## References

1. D. Augot, M. Finiasz, and N. Sendrier. A family of fast syndrome based cryptographic hash functions. In E. Dawson and S. Vaudenay (Eds.), *Mycrypt 2005*, Springer-Verlag LNCS 3615, pp. 64–83, 2005.
2. K. Bentahar, D. Page, M.-J.O. Saarinen, J.H. Silverman, and N. Smart. LASH. *2nd NIST Cryptographic Hash Workshop*. 2006.
3. S. Contini, A.K. Lenstra, and R. Steinfeld. VSH, an efficient and provably collision-resistant hash function. IN S. vaudenay (Ed.), *Eurocrypt 2006*, LNCS 4004, pp. 165–182, Springer, 2006.
4. I.B. Damgård. A design principle for hash functions. In G. Brassard (Ed.), *Advances in Cryptology – CRYPTO '89*, LNCS 435, pp. 416–427, Springer-Verlag, 1990.
5. M. Finiasz, P. Gaborit, and N. Sendrier. Improved fast syndrome based cryptographic hash functions. *ECRYPT Hash Function Workshop 2007*, 2007.
6. R.C. Merkle. A fast software one-way hash function. *Journal of Cryptology*, **3**, 43–58, 1990.
7. M.-J.O. Saarinen. Security of VSH in the real world. In R. Barua and T. Lange (Eds.): *INDOCRYPT 2006*, LNCS 4329, pp. 95–103, Springer, 2006.
8. V. Rijmen and P. Barreto. Whirlpool. Seventh hash function of ISO/IEC 10118-3:2004, 2004.
9. D. Wagner. A generalized birthday problem. In M. Yung (Ed.), *Advances in Cryptology – Crypto 2002*, LNCS 2442, pp. 288 – 304, Springer, 2002.

## A Appendix: A Collision and Pre-Image Example

For parameter selection $r = 1024$, $w = 1024$, $n = 262144$, $s = 8192$, $n/w = 256$, the FSB compression function can be implemented in C as follows.

```
typedef unsigned char u8;          // u8 = single byte
typedef unsigned long long u64;    // u64 = 64-bit word

void fsb(u64 h[0x40000][0x10],     // "random" matrix
         u8 s[0x400],              // 1k message block
         u64 r[0x10])              // result
{
  int i, j, idx;

  for (i = 0; i < 0x10; i++)       // zeroise result
    r[i] = 0;
  for (i = 0; i < 0x400; i++)      // process a block
    {
      idx = (i << 8) + s[i];       // index in H
      for (j = 0; j < 0x10; j++)
        r[j] ^= h[idx][j];         // xor over result
    }
}
```

Since the FSB specification does not offer any standard way of defining the "random" matrix $\mathcal{H}$ (or `h[][]` above), we will do so here using the Data Encryption Standard. Each 64-bit word `h[i][j]` is created by encrypting the 64-bit input value `(i << 4) ^ j` under an all-zero 56-bit key (`00 00 00 00 00 00 00 00`). The input and output values are handled in big-endian fashion. Some of the values are: [2]

```
Input to DES         Table Index         Value
0x0000000000000000   h[0x00000][0x0]  =  0x8CA64DE9C1B123A7
0x0000000000000001   h[0x00000][0x1]  =  0x166B40B44ABA4BD6
0x0000000000000002   h[0x00000][0x2]  =  0x06E7EA22CE92708F
                              ....
0x0000000000000010   h[0x00001][0x0]  =  0x5B711BC4CEEBF2EE
0x0000000000000011   h[0x00001][0x1]  =  0x799A09FB40DF6019
0x0000000000000012   h[0x00001][0x2]  =  0xAFFA05C77CBE3C45
                              ....
0x00000000003FFFFD   h[0x3FFFF][0xD]  =  0x313C4BDBE2F7156A
0x00000000003FFFFE   h[0x3FFFF][0xE]  =  0x19F32D6B2D9B57F5
0x00000000003FFFFF   h[0x3FFFF][0xF]  =  0x804DB568319F4F8B
```

We shall define two 1024-byte message blocks that produce the same 1024-bit chosen output value in the FSB compression function, hence demonstrating the ease of pre-image and collision search on a variant with claimed $2^{128}$ security. They were found in less than a second on an iBook G4 laptop.

[2] Please note that x86 platforms are little-endian. Bi-endian `gcc` source code for producing pre-images can be downloaded from: `http://www.m-js.com/misc/fsb_test.tar.gz`

The first message block uses the ASCII alphabet $\{\texttt{A},\texttt{C}\}$ or $\{\texttt{0x41},\texttt{0x42}\}$:

```
CAACACACCACAACACACACCACAACCCCCCACCAACACCAAACAAACACCAACACCACACCAA
ACACACCCCCAACCCAAAAAACCCACCACCCACCAAACACACCCCCCAACCACACCCAACACCA
AACCCACCCCCAACCCAAACAAAAACCCACAAAACACACCACCACCCCCACAACCCCACACAAA
AACCCCACCCCAACAACAAAAACAAAACCACACACACACCCCCAAACCCCCAAAAACCCACAAC
CAAACAACCCAAACACCAACCCCACACCCCAAAACCCAAAAAACACAAACCCCAACAAAACCAA
ACACCCCCCCCCAACAAAAACACCCACCCAACAAAAAAACACACCCCCCCAACCCACCCCAACA
AAAACCAACAACACCACCCCACCCCCACCACAAACACCCACCACCCAACCCCACCCAACAAAAC
ACCACCCCAACCCACAACCACCCAACACCAACACCAAAACACACCAAAACACCCAACACACCCC
CAAACACACACCACCACCACCCAAAAAAACCACACACCCCAAAAAAACCCAAACCACCACCCCA
CACAAACCCCAACCCAACCCAACCAACCACCAAAACCCAACCCCCAAAAAACAACCAAACCCCA
AACACCCACAAACACCACCACAACAAAAACCAAACCCAAAAACCCACCACACCCACACACAAAA
CCACCCCAACCCCCAACAACCCCACACAACACAAACCACCCAACCCCAACCACAAAAACCCACC
ACAACCCAAACACACCCCAACAAACCAAACCCCACACCCAAAACCCCACACCACACACAAACAC
CACCCAAAAAACAACAACCACACACAACAAACCAAACAAAAAAAAAACCAAAAAACCCCCAACC
CACCCACCCACAAACAAAACCAAAAAAAACCCAAAAAAACCCAAAACCACAACCACCCCAACCA
CCCACCAAACAACAACCACACAAAAACACCCCACACCCCCCCACCAACACAAAACCAAAAACCA
```

The second message block uses ASCII alphabet $\{\texttt{A},\texttt{H}\}$ or $\{\texttt{0x41},\texttt{0x48}\}$:

```
AHHHHAAAAAHAAAHAHAAAHAHHAHHAAHAAHHAHHHAAAAAHHAAHHHAHAHAAHAAAHHAA
AAAAHAHHAAAHAHHAHAAAHAAHAHAAAAHHHHHHHAAHAHAAAAAHAHHHHHAAHHHHAHAH
AAHAAAHAHAHHHHHAHHAHAHAAAHAHAAHAHHAAAAHAAHAAAHAAHHHHHAHAAHHAAHAH
HHAHAAHHHAAHAAAHHHHAHHHHAAHAAHAAAAAHAAHHAAAHAAHHHAAHAHAHHHAHAAHA
AHHAAAHHAAAAAHHAHAAAAAHAHAHHAHHAHAAHHAHAHAAHHHHAAHAHHHAAHHAHAAHH
AAHAHAAAHAHAAAHHAAAHAHHAHAHHAAAAAHHHHAAHAHAHHAHHHHHAAHHAAHHHHAHH
HHHAAAAAAAHHHAHAAAAHAAAHAAAAAAAHAAHHAHHAHHAHHAHHHAAAAAAAHAHAAAHH
HAHHHHHHAHAAAHHAHAAHHHHAAHHAHHAHHAAHHHAHHAHHHAAHHAAAHHAHAAHAHHHA
AAHAHAAAHAAHAAAAHHHHAHHHHHAAHHHAAHHHAHHAAAHHHAHHAHAHHHHAAHAHHAHH
AAHAHAAHHAHHAAAAHHAHAHHHHHAAHHHAAHAAAHAAAHAAHHAHHAHHHAHHHHHAHHHA
AHAHAAAAHHAAAAHHAAHHHHHAAHAAHAAHHAAAHAHHAAAAAHHAAHAHHAHHHAAHHHAA
HHHAHHAAHAAHAAHAAHHHHHAAHAHHAHHAAHAAAAHHAHHHHHAHAHHHHHAHHHHHAAAA
HHHHHAAAAHHHAHHHHAHAAAHHAHAAAHHAAAHAHAHAAAHHHHHHHAHAAHAAHAAAAHAA
HAAAHAHAHHHAHHAHHAHAAHAHHAAAAHAAAAHHAAHHHHAHHAAHHHAHAAAHAAAHHHAA
HAAHAAHAAAHAHHHAAHAHAAHAAAHAHHAHAAHHHAAHAAAAAHHAAAAHHHAHAHAAAAAH
AAAHAHAHHAAAAHHHAAHHAHAAHHHHAHAAHHAHHHAAHAHHAHHHAAAAHHHAAHAAAAHH
```

The 1024-bit / 128-byte result of compressing either one of these blocks blocks is:

```
Index     Hex                                 ASCII
00000000  5468697320697320  6120636f6c6c6973  |This is a collis|
00000010  696f6e20616e6420  7072652d696d6167  |ion and pre-imag|
00000020  6520666f72204661  73742053796e6472  |e for Fast Syndr|
00000030  6f6d652042617365  6420486173682e20  |ome Based Hash. |
00000040  4172626974726172  79207072652d696d  |Arbitrary pre-im|
00000050  616765732063616e  20626520666f756e  |ages can be foun|
00000060  6420696e20612066  72616374696f6e20  |d in a fraction |
00000070  6f66206120736563  6f6e642120202020  |of a second!    |
```