Biuletyn WAT
Vol. LX, Nr 3, 2011

Implementation and parallel cryptanalysis of MASH hash function family

MAREK GRĄDZKI

Military University of Technology, Faculty of Cybernetics,
Institute of Mathematics and Cryptology, 00-908 Warsaw, Kaliski 2 Str.,
e-mail: mgradzki@wat.edu.pl

Abstract. In the article, two Java implementations of the MASH hash function family are presented. The first uses standard classes, the second, custom class with optimized execution time and memory consumption.
Fast and low memory implementations of MASH hash functions allowed to utilize full power of 368-core Azul Compute Appliance to perform parallel collision search using distinguished points method.
Keywords: hash functions, collision search, modular arithmetic, parallel computing

1. Introduction

Cryptographic hash functions[1] play very important role in modern cryptography. The most important applications are data integrity and message authentication.

1.1. Hash functions

Hash function h takes a message of arbitrary finite length as input and produces an n-bit output referred to as a hashcode. For most applications,

[1] Cryptographic hash functions differ from conventional hash functions commonly used in non-cryptographic computer applications in several important aspects. For more information see [1]. In this article, the term *hash function* refers to *unkeyed cryptographic hash function*.

the following properties must be satisfied:

1. *preimage resistance* – it is computationally infeasible to find any x_0 such that $h(x_0) = y$ when given any y,
2. *2nd-preimage resistance* – given x, it is computationally infeasible to find $x_0 \neq x$ such that $h(x) = h(x_0)$,
3. *collision resistance* – it is computationally infeasible to find any x, x_0 such that $h(x) = h(x_0)$.

For an *ideal* n-bit hash function, the best attack is a guessing attack, which allows to find a preimage or second preimage within 2^n hashing operations; and birthday attack that allows to find collision in about $2^{\frac{n}{2}}$ operations, and negligible memory.

Most hash functions are designed as iterative processes which hash arbitrary length input x by processing fixed-size input blocks x_i. Each block x_i then serves as an input to the *compression function* f, which computes a new intermediate hash result H_i (also called *chaining variable*) of bitlength n, as a function of the previous n-bit intermediate hash result and the next input block x_i.

2. MASH hash function family

MASH (Modular Arithmetic Secure Hash) is a hash function family based on modular arithmetic, that consists of two cryptographic hash functions MASH-1 and MASH-2, included in Part 4 of ISO/IEC 10118 standard [5].

Motivation to use modular arithmetic in hash function is based on two factors: possibility of re-using existing software or hardware (in public-key systems) for modular arithmetic, and scalability to match required security level and desired hash value size.

2.1. Hashing procedure

MASH hashing procedure involves the following operations:

1. Preparation of the data string.
2. Iterative application of the compression function f, that uses exponentiation modulo N, for input data blocks.
3. Reduction operation applied to the last chaining variable, that gives result modulo p.

N denotes an RSA modulus, that is, the product of two large primes and $m = \lceil log_2 N \rceil$ denotes its length in bits. The length n of the chaining variables in bits is then equal to the largest multiple of 16 strictly smaller than m. The length of the message blocks is equal to $\frac{n}{2}$ bits. The prime number p, such that $L_p = \lceil log_2 p \rceil \leq \frac{n}{2}$, shall not be a divisor of N and the three most significant bits of p shall be equal to 1.

Preparation of the data string

The input string is divided into t $\frac{n}{2}$-bit blocks denoted as $X_0, X_1, \ldots X_{t-1}$ (rightpadded with '0' bits if necessary). Next, a block X_t is added which contains the input string length in bits, left-padded with '0' bits. Subsequently each $\frac{n}{2}$-bit block X_i is expanded to an n-bit block $\tilde{X}_i$ by inserting four '1' bits before every 4-bit substring of X_i.

Application of the compression function

The MASH-1 compression function, which maps $2n$ bits to n bits, is defined as follows:

$$f\left(\tilde{X}_{i-1}, H_{i-1}\right) = \left(\left(\left(\left(H_{i-1} \oplus \tilde{X}_{i-1}\right) \vee E\right)^e \pmod N\right) \sim n\right) \oplus H_{i-1}. \quad (1)$$

Where $E = 0\text{xF}00\ldots00$ and $\sim n$ denotes that the rightmost n bits of the m-bit result are kept. Value of exponent e depends on the version of MASH function: $e = 2$ for MASH-1, $e = 257$ for MASH-2. The iteration is performed as follows:

$$H_0 = 0, \quad (2)$$

$$H_i = f\left(\tilde{X}_{i-1}, H_{i-1}\right), i \ = 1, 2, \ldots, t. \quad (3)$$

Reduction modulo p

At the end, a complex output transformation involving eight additional iterations of the compression function, is applied[2] to H_t. Hashcode is computed as $X_{t+8} \bmod p$.

[2] For more detailed description of MASH functions see [6] and [5].

2.2. Security

MASH-1 and MASH-2 have been designed to have all properties described in subsection 1.1. When the factorization of the modulus N is not known, the best known preimage, 2nd preimage and collision attacks on MASH-1 require 2^{L_p} and $2^{\frac{L_p}{2}}$ operations, where L_p denotes length in bits of prime p. For very high security requirements it is advisable to select MASH-2 instead of MASH-1 (where some undesirable statistical properties may be present).

3. Implementation

Arbitrary-precision modular arithmetic in Java is possible using standard BigInteger class. Object of BigInteger class is immutable – once created cannot change its state, each operation on BigInteger object produces new object instance. Immutable objects have many advantages in standard applications (simplicity, reusability, thread safety, etc. see [7]). However, we show later, that immutable objects usage affect hash function performance.

```
1.  private final BigInteger compress(BigInteger X,
      BigInteger H) {
2.    BigInteger XX = BigInteger.valueOf(0);
3.    BigInteger rem;
4.    for (int j = k - 4; j >= 0; j -= 4) {
5.      XX = XX.shiftLeft(4).or(FIFTEEN);
6.      rem = block.shiftRight(j).mod(SIXTEEN);
7.      XX = XX.shiftLeft(4).or(rem);
8.    }
9.    return H.xor(XX).or(E).pow(2).mod(N).mod(
          TWO_POW_N).xor(H);
10. }
```

Listing 1. MASH-1 compression function implementation using BigInteger class

The idea of the first implementation is based on MASH implementation published in [4]. Published version does not implement reduction procedure and has some minor errors, therefore we made some modifications. The result was tested against test vectors published in [5]. Implementation of MASH-1 compression function using BigInteger class is presented on listing 1.

X denotes $\frac{n}{2}$-bit input block, H is a n-bit chaining variable. XX is an n-bit expanded X block (denoted earlier as $\tilde{X}_i$). Lines 2-8 describe expansion procedure. Line 9 describes round function as presented earlier in equation 1. Difference in line 9 implementation of the MASH-2 compression function is presented on listing 2.

```
9.      return H.xor(XX).or(E).modPow(EXP, N).mod(
            TWO_POW_N).xor(H);
```

Listing 2. MASH-2 compression function implementation using BigInteger class

The line from listing 2, creates at least 5 BigInteger object during one execution, which affects performance. Efficiency decrease can be observed when hashing long messages or performing parallel cryptanalysis. We describe later in the article, that expensive memory management makes it impossible to achieve full computation speed-up in multithreaded computation.

High memory consumption of immutable BigInteger class motivated us to provide second implementation of MASH hash functions using BigNum class we created. The class operates directly on vectors of primitive int type, which are created once and then reused. We implemented multiprecision arithmetic algorithms (addition, subtraction, division, modular exponentiation using squaring and Montgomery's reduction) as described in [1], squaring algorithm from Java JDK 6 was used[3].

Implementation of MASH-1 compression function using BigNum class[4] is presented on listing 3.

```
1.  private final void compress(int[] X, int[] H) {
2.      BigNum.expand(XX, X);
3.      BigNum.xor(XX, H);
4.      BigNum.or(XX, E);
5.      BigNum.square(XX, XX_SQUARE);
6.      BigNum.mod(XX_SQUARE, N, XX);
7.      BigNum.xor(H, XX);
8.  }
```

Listing 3. MASH-1 compression function implementation using BigNum class

[3] JDK sources are available at [9].

[4] Some technical details were removed to improve code readability.

Inspection of JDK sources showed, that the same algorithms are used in BigNum and BigInteger implementations. Our implementations were designed to be used in MASH hash functions. This assumption allowed for some optimizations:

- working on vectors of the same, fixed length,
- working on unsigned numbers,
- modular exponentiation only for exponents 2 and 257,
- modular reduction only for odd modulus.

We tested both implementations hashing random files of 100 MB[5]. Average results are presented in Table 1. Implementation based on our library is almost 4 times faster in case of MASH-2 and over 11 times faster in case of MASH-1.

TABLE 1
Hash functions implementation speed

function	class	KB/s
MASH-1	BigInteger	484.1
MASH-1	BigNum	**5548.23**
MASH-2	BigInteger	335.3
MASH-2	BigNum	**1250.18**

4. Parallel collision search

To perform a parallel collision search of $F : S \longrightarrow S$, each processor proceeds as follows. Select a starting point $x_0 \in S$ and produce the trail of points $x_i = F(x_{i-1})$, for $i = 1, 2, \ldots$ until a distinguished point x_d is reached based on some easily testable distinguishing property such as a fixed number of leading zero bits. Add the distinguished point to a single common list (together with x_0 and d) for all processors and start producing a new trail from a new starting point. A collision is detected when the same distinguished point appears twice in the central list (see: Figure 1).

To locate the collision begin by stepping the longer trail forward until its remaining length is the same as the colliding trail's length. Then, step the trails forward (can be done in parallel), until they both hit the same

[5] Tests were performed on 2.5 GHz Core 2 Duo laptop using 128-bit p and 1028-bit N

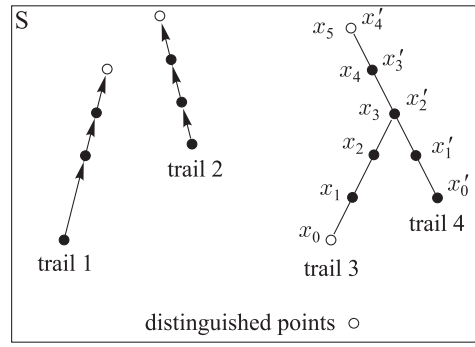


Fig. 1. Parallel collision search using distinguished points method

point. The run-time to detect and locate useful collision is:

$$T = \frac{\sqrt{\frac{\pi n}{2q}}}{m} + \frac{2.5}{\theta}, \tag{4}$$

where: $n = |S|$ – number of all points, m – number of processors, q – probability of useful collision, $\theta = \frac{D}{S}$ – proportion of points that satisfy distinguished property[6].

Attack scenario

Suppose we have a message m that we would like the victim to digitally sign, but he is not willing to do so and some other message m' that the victim is willing to sign. Find several ways to modify each of m and m' that do not alter their respective semantic meaning. Then, hash the different versions of m and m' until we find collision. The victim will sign the version of m' and we can copy the signature to m.

Let $f : M \longrightarrow R$ – compression function, $m \in M$, and let $g_m : R \longrightarrow M$ be an injective function which takes a hash result as input and produces a perturbation of the message m with the same semantic meaning to the signer. Partition the set R into two equal size subsets S_1 and S_2 based on some easily testable property of a hash result. Then, define a function $F : R \longrightarrow R$ as follows:

$$F(r) = \begin{cases} f(g_m(r)) & \text{if } r \in S_1 \\ f(g_{m'}(r)) & \text{if } r \in S_2 \end{cases}. \tag{5}$$

[6] The choice of θ influences time to locate collision, memory needed for storing distinguished points and frequency of sending them to the common list. For more details how to choose it see [2] and [3].

Using the parallel collision search, find pairs of hash results $a \neq b$, such that $F(a) = F(b)$. A collision is useful if a and b are in different subsets S_1, S_2 of R, which will occur with probability $\frac{1}{2}$.

Application to MASH hash functions

We applied distinguished points method to MASH hash functions without reduction procedure. To reduce the amount of computation, we will search for compression function collisions. For g_m and $g_{m'}$ to be injective, we must be able to code the bits of intermediate hash values into the message blocks. For most of popular hash functions it is easy, because message blocks are longer than chaining variables. In case of MASH functions quite the contrary: length of message block is a half of the chaining variable length.

The message m consists of the blocks $m_1, \ldots m_j, m_{j+1}, \ldots m_l$. Message m' consists of the blocks $m'_1, \ldots m'_j, m'_{j+1}, \ldots m'_l$. Collision bits are coded into blocks m_j, m_{j+1} and m'_j, m'_{j+1}. If the intermediate hash results are the same for m and m' after block number $j+1$, then the final hash results will be the same as well. Let r_{j-1} and r'_{j-1} be the intermediate hash values for m and m' after $j-1$ blocks; $high(r)$ – first and $low(r)$ – last $\frac{n}{2}$ bits of n-bit r. Use the following function F for collision search:

$$F(r) = \begin{cases} f(low(r),\ f(high(r),\ r_{j-1})) & \text{if } r \text{ is even} \\ f(low(r),\ f(high(r),\ r'_{j-1})) & \text{if } r \text{ is odd} \end{cases} . \tag{6}$$

4.1. Attack implementation

We implemented attack[7] using Java language and tested on Core 2 Duo 2.5 GHz laptop and Azul Compute Appliance. Azul Compute Appliance was designed to provide massive amounts of computing capacity for Java applications. The latest version based on Vega 3 54-core processor, can contain up to 864 processor cores and 768 GB of memory[8]. We tested our application on Azul 3840B Compute Appliance – one of the first versions offered with 384 cores (364 available for programmer) and 128 GB of memory based on 24-core Vega 1 processor.

[7] We describe results only for MASH-1. Results for MASH-2 differ only in computation time.

[8] For more detailed description of Azul Compute Appliances, see [8].

Benchmark

To compare both testing platforms and choose attainable attack parameters, we prepared a benchmark that invocates hash compression function in infinite loop and notifies counting thread every arbitrary chosen number of iterations. We present average results[9] in Table 2.

TABLE 2

MASH-1 benchmark results

Azul 3840B				
	MASH-1 using BigInteger		MASH-1 using BigNum	
threads	iterations/s	speed-up	iterations/s	speed-up
1	2967	1	39805	1
128	357523	120.50	4932843	123.92
256	669532	225.66	9824129	246.80
368	729632	245.92	14620768	367.3
Pentium Core 2 Duo 2.5 GHz				
1	53719	1	745145	1
2	72358	1.35	1379601	1.85

In both tables, difference in resource utilization can be seen. MASH function implementations using BigInteger class are not only slower, but also create many objects, that makes memory management difficult and make it impossible to utilize all cores of Azul 3840B (cores are involved in garbage collection instead of computation). Processors and memory usage for MASH-1 implementations using BigInteger and BigNum classes are shown in Figure 2.

Collision search results

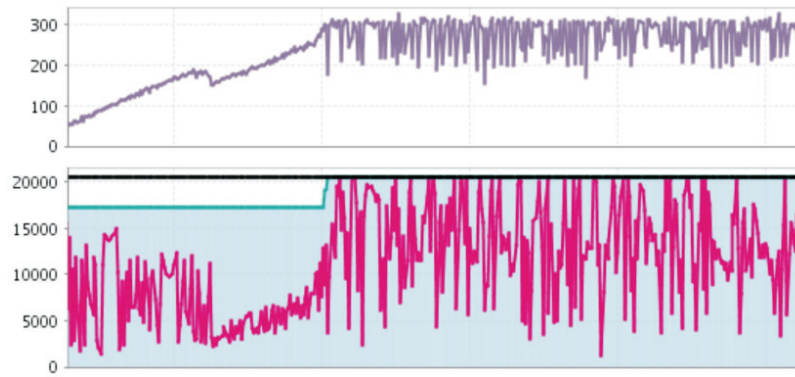
Below, in Table 3, we present expected number of iterations of function given by equation (6) for some arbitrary chosen parameters.

In Figure 3, sources of two MASH-1 colliding PostScript documents are presented. Collision bits are hidden in PostScript comment (starts with '%'). To find a collision, about 2^{41} iterations of function given by equation (6) was needed. Computations took about 35 hours on Azul 3840B.

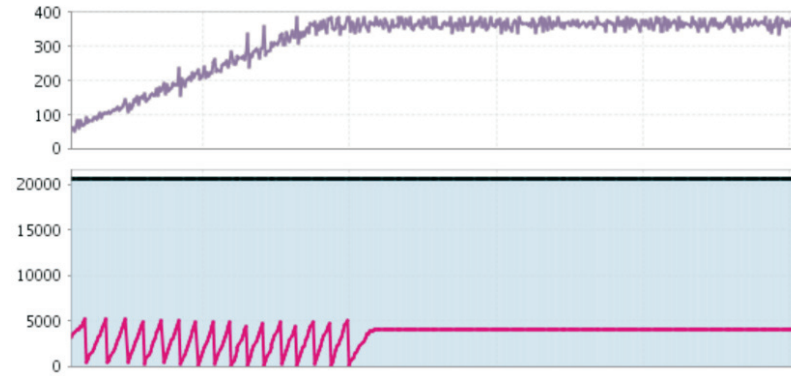
Below hex-encoded documents with bolded differences:

Hash value of both documents is equal to `ccfa968e545a2b46c20a` for MASH-1 modulus $N =$ `08a9d5fd787968c43932f7`.

[9] Benchmarking was performed using 256-bit N and 128-bit p.



(a) MASH using BigInteger



(b) MASH using BigNum

Fig. 2. Processor and memory (MB) usage during benchmarking with 368 threads

TABLE 3

Expected number of F function iterations for distinguished points collision search method. Computing speed is average measured on Azul 3840B for different modulus lengths

n	θ	iterations	10^6 iterations/s	computation time
2^{96}	2^{-16}	$2^{48.83}$	16.7	346 days
2^{80}	2^{-16}	$2^{40.83}$	18.3	30 hours
2^{64}	2^{-16}	$2^{32.83}$	20.9	6 minutes
2^{48}	2^{-16}	$2^{24.83}$	27.2	1 second

```
252150532d41646f62652d312e300d 0a2525426f756e64696e67426f783a
2030203020363132203739320a2f54 696d65732d4974616c69632066696e
252150532d41646f62652d312e300d 0a2525426f756e64696e67426f783a
2030203020363132203739320a2f54 696d65732d4974616c69632066696e
0a2f54696d65732d526f6d616e2066 696e64666f6e74203230207363616c
65666f6e7420736574666f6e740d0a 323520363030206d6f7665746f2028
4d6f6e6579207472616e7366657229 2073686f770d0a323520353230206d
6f7665746f2028506c656173652073 656e6420312c303030206575726f73
20746f20426f622773206163636f75 6e742e292073686f770a2f54696d65
732d4974616c69632066696e64666f 6e74203230207363616c65666f6e74
20736574666f6e740d0a3235203438 30206d6f7665746f2028416c696365
292073686f770d0a73686f77706167 652020202020202020202530a651069 7
5adb8612cd
```

```
252150532d41646f62652d312e300d 0a2525426f756e64696e67426f783a
2030203020363132203739320a2f54 696d65732d4974616c69632066696e
64666f6e7420323020736574666f6e 6f6e7420736574666f6e740a353030
20373030206d6f7665746f20283031 2e30352e323030392920736f770d
0a2f54696d65732d526f6d616e2066 696e64666f6e74203230207363616c
65666f6e7420736574666f6e740d0a 323520363030206d6f7665746f2028
4d6f6e6579207472616e7366657229 2073686f770d0a323520353230206d
6f7665746f2028506c656173652073 656e6420312c303030 2c3030302065
75726f7320746f2045766527732061 63636f756e742e292073686f770a2f
54696d65732d4974616c6963206669 6e64666f6e74203230207363616c65
666f6e7420736574666f6e740d0a32 3520343830206d6f7665746f202841
6c696365292073686f770d0a73686f 777061676520202020025157fbff61f
8acbd70d22
```

5. Summary

Hash functions from MASH family are slower than custom hash functions like SHA, however they can be easily changed to match required security level. Slow speed and some construction details (e.g., longer chaining variable than message block), make generic collision attacks impossible when longer hash values are used, even using specialized hardware. Faster software implementations are possible without using general purpose libraries, especially in languages that have automatic memory management (e.g., Java). The same optimization methods we used for MASH should also apply to other cryptographic primitives implementations e.g. RSA.

```
%!PS-Adobe-1.0
%%BoundingBox: 0 0 612 792
/Times-Italic findfont 20 scalefont setfont
500 700 moveto (01.05.2009) show
/Times-Roman findfont 20 scalefont setfont
25 600 moveto (Money transfer) show
25 520 moveto (Please send 1,000 euros to Bob's account.) show
/Times-Italic findfont 20 scalefont setfont
25 480 moveto (Alice) show
showpage        %0ŚQ□□□□ZŮ□□□□Í
```

(a) Document Alice is willing to sign

```
%!PS-Adobe-1.0
%%BoundingBox: 0 0 612 792
/Times-Italic findfont 20 scalefont setfont
500 700 moveto (01.05.2009) show
/Times-Roman findfont 20 scalefont setfont
25 600 moveto (Money transfer) show
25 520 moveto (Please send 1,000,000 euros to Eve's account.) show
/Times-Italic findfont 20 scalefont setfont
25 480 moveto (Alice) show
showpage    %□□□□żö□□□□Ëx
"
```

(b) Document Alice is not willing to sign

Fig. 3. Two colliding PostScript document sources

Work financially supported from Ministry of Science and Higher Education founds reserved for science as development project O R00 0043 07.

Received February 07 2011 Revised April 2011.

REFERENCES

[1] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press LLC, 1997.

[2] P.C. van Oorschot and M.J. Wiener, *Parallel Collision Search with Application to Hash Functions and Discrete Logarithms*, 2nd ACM Conference on Computer and Communications Security, Fairfax, Virginia, November 1994, pp. 210–218.

[3] P. van Oorschot and M. Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology, 12(1):1–28, 1999.

[4] D. Bishop, *Introduction to Cryptography with Java Applets*, 2003.

[5] ISO/IEC 10118. *Information technology–security. Part 4: Hash-functions using modular arithmetic*, 1998.

[6] H.C.A. VAN TILBORG, *Encyclopedia of Cryptography and Security*, Springer-Verlag New York, Inc., Secaucus, NJ, 2005.

[7] J. BLOCH, *Effective Java: Programming Language Guide*, Addison Wesley, 2001.

[8] Azul Systems , Documentation and specifications available at `http://www.azulsystems.com`.

[9] Sun Developer Network, Documentation, specifications and source codes available at `http://java.sun.com/`.

M. GRĄDZKI

Implementacja i równoległa kryptoanaliza funkcji skrótu z rodziny MASH

Streszczenie. W artykule przedstawiono dwie implementacje funkcji skrótu z rodziny MASH wykonane w języku Java. Pierwsza z nich wykorzystuje standardowe klasy, druga klasę zoptymalizowaną pod względem szybkości działania i zużycia pamięci.
Szybkie i oszczędne pamięciowo implementacje funkcji skrótu z rodziny MASH pozwoliły wykorzystać pełne możliwości 368-rdzeniowego Urządzenia Przetwarzającego Azul do równoległego znajdowania kolizji metodą punktów rozróżnialnych.
Słowa kluczowe: funkcje skrótu, poszukiwanie kolizji, arytmetyka modularna, obliczenia równoległe