An algorithm for MD5 single-block collision attack using high-performance computing cluster

Anton A. Kuznetsov

Program Systems Institute of Russian Academy of Sciences
aakuznetsoff@gmail.com
October 22, 2014

Abstract. The parallel algorithm and its implementation for performing a single-block collision attack on MD5 are described. The algorithm is implemented as MPI program based upon the source code of Dr Marc Stevens' collision search sequential program. In this paper we present a parallel single-block MD5 collision searching algorithm itself and details of its implementation. We also disclose a pair of new single-block messages colliding under MD5 that were found using our algorithm on the high-performance computing cluster.

1. Introduction

Hash functions are the one-way functions that map arbitrary input messages to a fixed-length hash values. Hashes can be considered as signatures of the original message, and can be used to check the message integrity and authenticity after it was delivered by network communication. Hash functions are designed to be fast but difficult to revert (calculate f^{-1}(hash)). MD5 is one of the most widely-used hash functions. It was designed in 1992 by R.Rivest [1].

The message pair (M,M') is called a *collision* if hashes of both messages are equal. In 2004 Wang et al. [2] have disclosed a differential method for finding MD5 collisions, and presented a collision with input messages of size 1024 bit (two-block collision). In 2010 Xie and Feng presented *single-block* colliding messages [3] but for security reasons haven't disclosed any detail about the collision searching method. They posted a challenge to cryptology community to construct a different MD5 single-block collision. In 2012 Dr Marc Stevens have answered that challenge [4] by presenting a single-block collision attack for MD5 and an example colliding message pair.

In this paper we describe a parallel algorithm for finding single-block MD5 collisions, and its MPI implementation that is based upon Dr Marc Stevens' sequential method. We also present a new single-block colliding message pair that was found using our algorithm on the high-performance computing cluster in 11 hours.

So far only one parallel collision search method exists. Citation from [5]: "it is a simple technique of parallelizing methods for solving search problems which seek collisions in pseudo-random walks. According to that method, to perform a parallel collision search, each processor proceeds as follows. Select a starting point $x_0 \in S$ and produce the trail of points $x_i = f(x_i\text{-}1)$, for i = 1, 2,... 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 for all processors and start producing a new trail from a new starting point. Depending on the application of collision search, other information must be stored with the distinguished point (e.g., one must store x_0 and d in order to quickly locate the points a and b such that f(a) = f(b)). A collision is detected when the same distinguished point appears twice in the central list."

We found no publicly available research papers about parallel methods of collision search that use Wang et al's differential method.

2. The sequential program structure

The general structure of the sequential (one-threaded) program by Dr Marc Stevens roughly is as follows (in pseudocode):

```
main():
     filltables();
     while (true) collinit();

collinit():
     // Instantiation
     // this is where random values are used
     compute Q1, Q4, Q5, Q12–Q18, m0, m5, m6, m11;
     compute Q3Q6cnt value;
     if Q3Q6cnt < 2^24: return;
     four nested 'do..while' loops:
          in the innermost 'do..while' loop:
               compute all other Qi;
               compute M and M';
               if md5compress(M) = md5compress(M'):
                    // collision was found
                    print (M,M') and Qi to stdout;
                    exit();
```

Algorithm 1: Sequential program structure

3. The MPI program structure

In MPI standard a program is executed in parallel by running the copy of the program on each core of each node of compute cluster. Each running process is assigned a rank – a decimal number from 0 to N-1, where N is the number of CPU cores in the cluster.

To develop MPI program a few obvious source modifications were made. The original algorithm was almost left intact, just a few MPI calls were inserted and some optimizations were made.

At the beginning of parallel program development for the convenience we wrote a wrapper library for MPI routines with a set of primitives:

- mpi_init() – initialize MPI computing environment;
- mpi_final() – finalize MPI computing environment;
- mpi_send() – send an array of unsigned integers to a specified rank;
- mpi_recv() – receive an array of unsigned integers (on slave ranks);
- mpi_barrier() – synchronize execution, wait until all computing processes have reached this routine;
- mpi_size() – return the total number of ranks;
- mpi_rank() – return the index of current rank;
- mpi_headrank() – check whether the current process has rank 0.

A head rank is a rank with index 0, a slave rank is a rank with non-zero index.

Each call is a wrapper for the genuine MPI call with error-checking. For example, mpi_barrier() has the following definition:

```
void mpi_barrier() {
    int rc = MPI_Barrier(MPI_COMM_WORLD);
    if (rc != MPI_SUCCESS) {
        printf("Error in MPI_Barrier\n");
        fflush(stdout);
    }
}
```

Listing 1: mpi_barrier() wrapper subroutine

The structure of developed parallel MPI program is as follows (in pseudocode):

```
main():
     mpi_init();
     filltables();
     while (true) collinit();
     mpi_final();

collinit():
     mpi_barrier();
     if mpi_headrank(): // node 0: Instantiation
          // this is where random values are used:
          compute Q1, Q4, Q5, Q12–Q18, m0, m5, m6, m11;
     if mpi_headrank():
          mpi_send(Q); // broadcast Qi
          mpi_send(M); // broadcast M
     else:
          mpi_recv(Q); // slave ranks receive Qi
          mpi_recv(M); // slave ranks receive M
     compute Q3Q6cnt value on each rank;
     if Q3Q6cnt < 2^24: return;
     mpi_barrier();
     four nested 'do..while' loops:
          in the innermost 'do..while' loop:
               if numIter  mod  mpi_size() = mpi_rank():
               // numIter – the counter of loop iterations
                    compute all other Qi;
                    compute M and M';
                    if md5compress(M) = md5compress(M'):
                         // collision was found on some rank
                         print (M,M') and Qi to stdout;
                         mpi_final();
                         exit();
```

Algorithm 2: Parallel program structure

4. Optimizations to the source code

Following is the list of optimizations applied in the parallel (MPI) version of single-block collision search program.

- Do not declare vector and numeric variables in every iteration of the inner loop; declare these before the outermost 'do..while'. This is due to the fact that innermost loop is executed several billion times (in worst scenario), thus declaration of variables consume an amount of CPU time.

- Using a simple yet powerful free code profiler we have made a conclusion that during the program run most of the CPU clock is consumed by calls to the four routines:
  - rotate_right()
  - rotate_left()
  - md5_ff()
  - md5_gg()

  The former two were optimized using Intel compiler intrinsics:
  - rotate_right() was rewritten using _rotr()
  - rotate_left() was rewritten using _rotl()

  md5_ff() routine is optimized like this:
  rewrite from:
  - D ^ (B & (C ^ D))

  to:
  - (B & C) | (~B & D)

  md5_gg() routine is optimized like this:
  rewrite from:
  - C ^ (D & (B ^ C))

  to:
  - (D & B) | (~D & C)

- md5compress() C++ function was rewritten in Assembler. This yields about 20% speed-up.

- The following command is used to compile program source:
  ```
  mpic++ *.cpp md5compress.S –O3 -xhost –ipo –o md5sbc
  ```
  It results in a faster binary for the host Intel Xeon processor with interprocedural optimizations and aggressive loop unrolling applied.

- Source code was refactored by running a small Tcl script on it. All substrings in the source code that match the "offset+%i" mask were replaced by the actual sum of the 'offset' constant (that equals to 3) and the integer %i. This was done solely to improve code readability and examine data dependencies between program subroutines.

5. HPC cluster run

For the experiments we have used a cluster (named "Tornado") that resides in South Ural State University [6] in the city of Chelyabinsk, Russia. We had made several MPI program launches about 24 hours duration each. The final launch that took **11 hours** was successful – a collision was found (see next section). In that launch 30 nodes were used with 12 processes on each node (number of ranks – 360).

It is obvious that the more nodes used in computations the higher possibility of finding collision within reasonable timeslot (e.g. 24 hours). Program run time is different from launch to launch because random number generator is used to calculate some of the Q_i values at the

initialization stage.

The parallel algorithm is highly scalable due to the fact that in the inner loop all iterations are split equally among ranks. Total number of iterations is very high. In the worst scenario even 10-petaflop/s HPC cluster could take weeks to find collision.

We did not use any accelerator devices like Intel Xeon Phi, that are present on the cluster, but this is actually feasible for our implementation.

6. The colliding message pair

We present a new message pair colliding under MD5. It was found by running our parallel implementation of the collision finding program on the HPC cluster:

M	5D 11 69 3E 1E 33 4B 2C B3 88 EF AA F0 D0 EC F3 91 2D 73 0A 1C DD 7A AC 6E 3C E0 E4 CE 06 7B B1 8E 73 C7 **BA** A2 6A A8 19 66 C2 86 16 B3 4F 3D 07 AA B7 C8 1E 32 94 89 **64** 7C 11 73 4A 3F AF 03 EA
M'	5D 11 69 3E 1E 33 4B 2C B3 88 EF AA F0 D0 EC F3 91 2D 73 0A 1C DD 7A AC 6E 3C E0 E4 CE 06 7B B1 8E 73 C7 **BC** A2 6A A8 19 66 C2 86 16 B3 4F 3D 07 AA B7 C8 1E 32 94 89 **E4** 7C 11 73 4A 3F AF 03 EA
Common MD5 hash: 746c4e219320eae3fd23bcf3ebb7d71d	

Table 1: The single-block colliding messages

The messages are available for download at [7].

We also present here the list of Q_i values that was found by the parallel program and was used to generate message M:

Q_{-3}=0x67452301 Q_{-2}=0x10325476 Q_{-1}=0x98BADCFE Q_0 =0xEFCDAB89 Q_1 =0xD9A89593 Q_2 =0xDA361481 Q_3 =0x0660DFEA Q_4 =0x04812801 Q_5 =0xEB78D1DC Q_6 =0x77D76EFF Q_7 =0xBE675C82	Q_8 =0x29F20526 Q_9 =0x3E1893ED Q_{10}=0x00000040 Q_{11}=0xFFFFFDFE Q_{12}=0xB62EA109 Q_{13}=0x062DA1C8 Q_{14}=0x1661D7EA Q_{15}=0x00050621 Q_{16}=0x14810A21 Q_{17}=0xA8009748 Q_{18}=0xADABC8E8	Q_{19}=0x410F3F70 Q_{20}=0x71936434 Q_{21}=0xF7D2E265 Q_{22}=0x09D6ECD5 Q_{23}=0xF8B84FB6 Q_{24}=0xBCCE16A3 Q_{25}=0x463268A8 Q_{26}=0x34EFF95F Q_{27}=0x5E7E0F7D Q_{28}=0xE8514E70 Q_{29}=0xC677D867

Table 2: Q values list

Note: all the rest Q values (Q_{30}—Q_{64}) are equal to 0.

Message M = (m_0 ... m_{15}) is derived from Q values as follows:

$$m_t = RR(Q_{t+1} - Q_t, RC_t) - AC_t - f_t(Q_t, Q_{t-1}, Q_{t-2}) - Q_{t-3}$$

where:

RR(X, n) is a cyclic right rotation of X by n bit positions;

RC_t is a rotation constant: $(RC_t, RC_{t+1}, RC_{t+2}, RC_{t+3}) = (7, 12, 17, 22)$ for $t = (0, 4, 8, 12)$;

AC_t is an additive constant: $AC_t = \lfloor 2^{32} |\sin(t+1)| \rfloor$;

$f_t(X, Y, Z) = F(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z)$.

Message M' is derived from M as follows:

$$M' = M + (0,0,0,0,0,0,0,0,2^{25},0,0,0,0,2^{31},0,0)$$

Q_i values presented here generally satisfy bitconditions (see Table 3 in [4]) but not all. There are four values that do not satisfy bitconditions: Q_3, Q_4, Q_9 and Q_{14}. It is an open question why this divergence occured.

7. Conclusion

We presented the collision searching parallel algorithm that was derived from Dr Marc Stevens' original method. It is implemented as MPI program and successfully used to find a pair of messages colliding under MD5.

Dr Marc Stevens' algorithm has a runtime cost of 2^{50} md5compress() calls. We believe that a single-block collision searching algorithm can be substantially improved, so that it requires much less computational power. This is the subject for further research.

The collision search program can be adapted to run on other massively parallel devices: multi-core CPUs, Nvidia CUDA devices, Intel Xeon Phi accelerators. This can greatly speed up collision search on the workstation and/or computational cluster.

Acknowledgements

This work is supported by the Russian Academy of Sciences through the project No.01201354596.

We express gratitude to Dr Marc Stevens for permission to modify the source code of his single-block collision search program [8].

References

1. Ronald L. Rivest, The MD5 Message-Digest Algorithm, Internet Request for Comments, April 1992, RFC 1321

2. Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu, Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD, Cryptology ePrint Archive, Report 2004/199, 2004

3. Tao Xie and Dengguo Feng, Construct MD5 Collisions Using Just A Single Block Of Message, Cryptology ePrint Archive, Report 2010/643, 2010

4. Marc Stevens, Single-block collision attack on MD5, Cryptology ePrint Archive, Report 2012/040, 2012

5. Paul C. Van Oorschot, Michael J. Wiener, Parallel collision search with cryptanalytic applications, Journal of Cryptology, 1999, vol.12, pp. 1-28

6. http://supercomputer.susu.ac.ru/computers/tornado/

7. http://www.botik.ru/~botik/rnd/message1ak , http://www.botik.ru/~botik/rnd/message2ak

8. http://marc-stevens.nl/research/md5-1block-collision/