Collisions in the Original Version of a Chaotic Hash Function

Scott Contini

Department of Computing, Macquarie University, NSW 2109, Australia

Abstract. At the Crypto 2005 rump session, a new hash function based upon a chaotic map was presented. We display several messages that collide to the same output for this hash function.

1 Introduction

At the Crypto 2005 rump session [1], a new hash function was designed based upon chaos theory. Although the function is extremely efficient, it has a major shortcoming: collisions can easily be found. Contrary to the author's design goals, these collisions can be found without inverting the chaotic map. The author's argument for its security is a heuristic rather than a provable one [2].

This note shows several different inputs that hash to the same output. These collisions were given to the author, at which time he replied that he had earlier designed two variants that he believed were stronger. We were able to find a simple collision in the first variant using techniques that we used to attack the original. We did not look at the second variant. This note only shows collisions in the original version.

2 The Chaotic Hash Function

A C implementation of the chaos hash function is given in Figure 1 (derived from the specifications [2]). Here, integers are 32-bit words. The hash takes as input `len` 128-bit blocks and outputs a 128-bit hash value. The specifications do not tell how to hash inputs which are not a multiple of this length, so this in itself is a problem. However, we can cryptanalyze it even with this restriction. The macros ROTL and ROTR are circular rotation left and circular rotation right of the specified word by the specified number of bits.

3 Collisions

Notice that during any processing of a 128-bit block, an attacker can choose the inputs to make `(X,Y,Z,U)` whatever he wants, since he will know what `(x,y,z,u)` are at any given time. Thus, he has complete control of the newly computed `(x,y,z,u)`. Also, if the same `(x,y,z,u)` are used twice, then the

```c
void
hash( unsigned int *input, int len, unsigned int output[4] )
{
    unsigned int x, y, z, u, X, Y, Z, U, A, B, C, D, RV1, RV2, RV3, RV4;
    unsigned int M = 0xffff;
    int i, offset;

    x = 0x0124fdce; y = 0x89ab57ea; z = 0xba89370a; u = 0xfedc45ef;
    A = 0x401ab257; B = 0xb7cd34e1; C = 0x76b3a27c; D = 0xf13c3adf;
    RV1 = 0xe12f23cd; RV2 = 0xc5ab6789; RV3 = 0xf1234567; RV4 = 0x9a8bc7ef;

    for (i=0; i < len; ++i) {
        offset = 4*i;

        X = input[offset + 0] ^ x; Y = input[offset + 1] ^ y;
        Z = input[offset + 2] ^ z; U = input[offset + 3] ^ u;

        /* compute chaos */
        x = (X & 0xffff)*(M-(Y>>16)) ^ ROTL(Z,1) ^ ROTR(U,1) ^ A;
        y = (Y & 0xffff)*(M-(Z>>16)) ^ ROTL(U,2) ^ ROTR(X,2) ^ B;
        z = (Z & 0xffff)*(M-(U>>16)) ^ ROTL(X,3) ^ ROTR(Y,3) ^ C;
        u = (U & 0xffff)*(M-(X>>16)) ^ ROTL(Y,4) ^ ROTR(Z,4) ^ D;

        RV1 ^= x; RV2 ^= y; RV3 ^= z; RV4 ^= u;
    }

    /* now run 4 more times */
    for (i=0; i < 4; ++i) {
        X = x; Y = y; Z = z; U = u;

        /* compute chaos */
        x = (X & 0xffff)*(M-(Y>>16)) ^ ROTL(Z,1) ^ ROTR(U,1) ^ A;
        y = (Y & 0xffff)*(M-(Z>>16)) ^ ROTL(U,2) ^ ROTR(X,2) ^ B;
        z = (Z & 0xffff)*(M-(U>>16)) ^ ROTL(X,3) ^ ROTR(Y,3) ^ C;
        u = (U & 0xffff)*(M-(X>>16)) ^ ROTL(Y,4) ^ ROTR(Z,4) ^ D;

        RV1 ^= x; RV2 ^= y; RV3 ^= z; RV4 ^= u;
    }

    output[0] = RV1; output[1] = RV2; output[2] = RV3; output[3] = RV4;
}
```

Fig. 1. C implementation of chaotic hash function.

results cancel out when exclusive-or'ed to the "rotation vectors" `RV1, ..., RV4`. Finally, observe that making `(X,Y,Z,U)` all `0xffffffff` or all `0` results in the same output of the chaos function.

Based upon these observations, we create collisions as follows:

appending: For the first input we have two 128-bit blocks. We choose the first 128-bit block to be the same as the default `(x,y,z,u)` in order to make `(X,Y,Z,U)` as all 0's, and then the newly computed `(x,y,z,u)` = `(A,B,C,D)`. For the second 128-bit block we send in the same values as `(A,B,C,D)`, thus making `(X,Y,Z,U)` and `(x,y,z,u)` the same as the previous iteration, and bringing `(RV1, RV2, RV3, RV4)` back to their default values.
For the second input, we append to the first input the 128-bit block `(A,B,C,D)` two more times. In the processing of these blocks, `(X,Y,Z,U)` and `(x,y,z,u)` remain the same, and after the fourth block is processed, `(RV1, RV2, RV3, RV4)` returns to its default value again.

complementing: In the above collision, we always have `(X,Y,Z,U)` as all `0`'s. We similarly can get collisions by making `(X,Y,Z,U)` all `0xffffffff`'s, which corresponds to complementing any of the 128-bit input blocks.

choosing arbitrary first block: We can make the first 128-bit block whatever we want, such as `(0x496e7365, 0x63757265, 0x20686173, 0x682e2e2e)`. This determines `(x,y,z,u)`, so for the second block, we choose a block value that will result in the same `(X,Y,Z,U)` as the previous iteration, thus bringing `(RV1, RV2, RV3, RV4)` back to its default value. In this example, the next input would be `(0xfd2dcee4, 0xda1dcc00, 0x702140cb, 0x1ed13af2)`. For the third block, we then choose the inputs in such a way as to make `(X,Y,Z,U)` all `0`'s (or all `0xffffffff`'s) and for the fourth, we choose inputs again to make `(X,Y,Z,U)` all `0`'s (or all `0xffffffff`'s).

For completeness, we explicitly state a few of these collisions, which all result in the hash value of `0xdaf69a60 0x76f0a062 0x8312e25d 0x50c8b3fe`.

– The first input is:

0x0124fdce	0x89ab57ea	0xba89370a	0xfedc45ef
0x401ab257	0xb7cd34e1	0x76b3a27c	0xf13c3adf

– Appending to the first input:

0x0124fdce	0x89ab57ea	0xba89370a	0xfedc45ef
0x401ab257	0xb7cd34e1	0x76b3a27c	0xf13c3adf
0x401ab257	0xb7cd34e1	0x76b3a27c	0xf13c3adf
0x401ab257	0xb7cd34e1	0x76b3a27c	0xf13c3adf

– Complementing entire first input:

0xfedb0231	0x7654a815	0x4576c8f5	0x0123ba10
0xbfe54da8	0x4832cb1e	0x894c5d83	0x0ec3c520

– Complementing second block of first input:

0x0124fdce	0x89ab57ea	0xba89370a	0xfedc45ef
0xbfe54da8	0x4832cb1e	0x894c5d83	0x0ec3c520

– Choosing arbitrary first block:

0x496e7365	0x63757265	0x20686173	0x682e2e2e
0xfd2dcee4	0xda1dcc00	0x702140cb	0x1ed13af2
0xb567404f	0x30c3e98f	0xeac016b2	0x88235133
0x401ab257	0xb7cd34e1	0x76b3a27c	0xf13c3adf

References

1. Chil-Min Kim. New hash function with chaos. Crypto 2005 Rump Session. Aug 16 2005, Santa Barbara, California.
2. Chil-Min Kim. New hash function with chaos. Unpublished manuscript, 2005.