Encrypted Product Sequence Numbers

Rick Walker, (walker@omnisterra.com)

October 15, 2012

USB devices generally have several internal variables that can be programmed into non-volatile eFuse memory at manufacture time. Some of these variables affect internal operation and are invisible to the end user. Other variables are intended to be read by either the user, or the user's operating system. The user can access these internal variables at enumeration time.

When a USB device is plugged into a host computer for the first time it, the computer will query the device for a complete list of operational characteristics which will be used to assign appropriate device drivers and application programs to the device. The assignment and handling of the serial number field is the subject of this report.

Using either an encoded-field serial number or a simple sequentially-assigned serial number can leak information to competitors. It might be possible to estimate build volumes, part changes, chip revisions and other sensitive information by simply buying several cables at different times and comparing the serial numbers. This report proposes the use of a simple, public-domain cryptographic technique which makes the ID number appear to be random, but which allows the manufacturer to easily decode the ID number into a simple sequential build number.

1 USB Device ID number

The USB protocol assigns 32 bits to the device ID field. The ID number is generally encoded in USB device at manufacturing time. When a cable end is enumerated by the host computer, the cable returns the 32 bit ID number as 8 hexidecimal ASCII numbers, each number consisting of one of the characters from the string "01234567890ABCDEF". Each character encodes 4 bits of the serial number. It is a trivial exercise to convert the hexidecimal string into an integer which ranges from 0 to 2^32-1 or about 4.29 billion different ID codes.

1.1 ID numbers with Sub fields

One way to use the 32 bit ID number is to break it into several subfields: one subfield for date of manufacture, one for wafer lot, one for revision, one for sequence number on the given date, and so on.

A disadvantage to this approach is that the number of combinations for the 32 bits gets used up rapidly and not very efficiently. The field for the quantity built on a given date will need to be big enough for the maximum possible number of units that could be built on a given day, and the unused bits for any given day will never be able to be used again.

A second disadvantage of this approach is that the encoding, once it is reverse engineered, may leak to competitors the build volume of the product manufacturing line. If one is so lucky to achieve multi-million unit annual build volume, the very existence of such a market will easily launch multiple startups to attempt to divy up such a big pie.

1.2 Sequential ID Numbers

A simple mechanism for serial numbers is to simply assign them in sequential order. Using the hexidecimal number system, the first 18 cables would then be numbered as follows:

```
00000000, 00000001, 00000002, 00000003,
00000004, 00000005, 00000006, 00000007,
00000008, 00000009, 0000000A, 0000000B,
0000000C, 0000000D, 0000000E, 0000000F,
00000010, 00000011, ...
```

From the manufacturer's viewpoint, the system is very convenient. The factory records all changes to the manufacturing process, the wafer lots received, BOM updates, etc, in a database and simply programs the chips with sequential serial numbers. If any issue arises in the field, the serial number can be used to query the database and find out all the specifics for the specific cable.

However, it is clear that this numbering scheme makes the product build volume trivially obvious to any customer that wishes to buy two cables, one year apart, and simply subtract the serial number of the older from the newer.

1.3 Scrambled Sequential Serial Numbers

To avoid the inefficient use of the 32 bit serial number, and to avoid the drawbacks of leaking build volumes to competitors, it is proposed to scramble the serial numbers.

1.3.1 Conceptual System

Conceptually, if one had a rather large deck of cards, each card numbered from 0 to 4.3 billion, it would be possible to thoroughly shuffle the deck to create a random permutation of the cards. The shuffled deck could then be sewn into a very thick book to preserve the scrambled sequence of cards.

Every time a product is programmed, the page of the book is turned and the next scrambled number is used as for the ID number. All build information could also be written on the page to record details for future failure analysis.

When it is necessary to analyze a field failure, the book can be scanned until the scrambled number corresponding to the failed cable is found. The page number of the the page will be the true sequential build number and the annotations on the page will give all the particulars for the cable assembly.

1.3.2 Practical System

It is impractical to keep an actual table for the 4.3 billion scrambled serial numbers, and a sequential search through such a list would be inefficient.

Fortunately, there exist simple mathematical techniques which allow computation of an equivalent system quickly and efficiently.

A system suitable for this application is the NSA Skipjack encryption system[1]. In particular, there is a variant of the original NSA algorithm called "skip32" [2] developed by Qualcomm in 1999 that is optimized for encrypting 32 bit blocks. When given an 80 bit password and a 32 bit sequence number, skip32 return a 32 bit encrypted output. In technical terms, the output words are guaranteed to be a permutation of the input space: each 32 bit input sequence maps to a unique 32 bit output sequence and the permutation is different for each of the possible 80 bit key values.

If the key is known, the algorithm can be run backwards to convert the encrypted 32 bit ID number back to the original sequence number.

The large key space allows 1.2e24 different unique shufflings. Assuming that the manufacturer keeps the scrambling key secret, it is not feasible for a competitor to use the cable serial number to deduce any details about the manufacture's volume or process variables.

The factory can keep all process data in an internal database keyed to a sequential build number. The skip32 algorithm will convert the sequential number into an encrypted ID for personalizing the product ASIC eFuses. When there is a need to query the database, the ID number is simply decrypted using the reverse skip32 algorithm.

The manufacturing workflow is described in Figure 1, and a typical debug workflow in Figure 2.

2 Example Program operation

A simple 173 line program written in C that implements skip32 is available from the author.

Here is a simple run of the program run for several input sequence numbers. The 80 bit secret key is specified in the source code.

[1]http://csrc.nist.gov/groups/ST/toolkit/documents/skipjack/skipjack.pdf
[2]http://www.mavi1.org/web_security/cryptography/skipjack/skip32.c

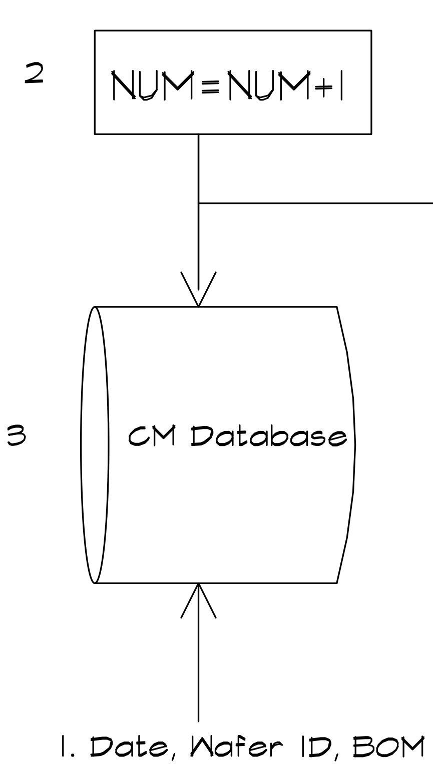
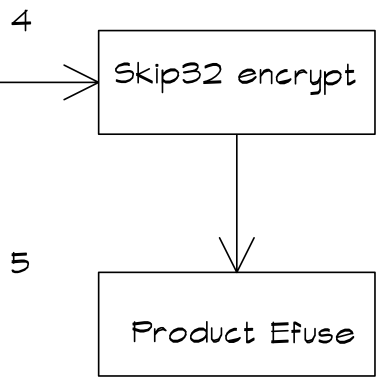

Figure 1: Manufacturing Workflow: 1) All information for a given part is collected, 2) a sequential build number is assigned for the assembly, 3) the assembly information is stored in a database keyed to the sequentially assigned build number, 4) the build number is encrypted with skip32, 5) the encrypted ID number is burned into the product Efuse.

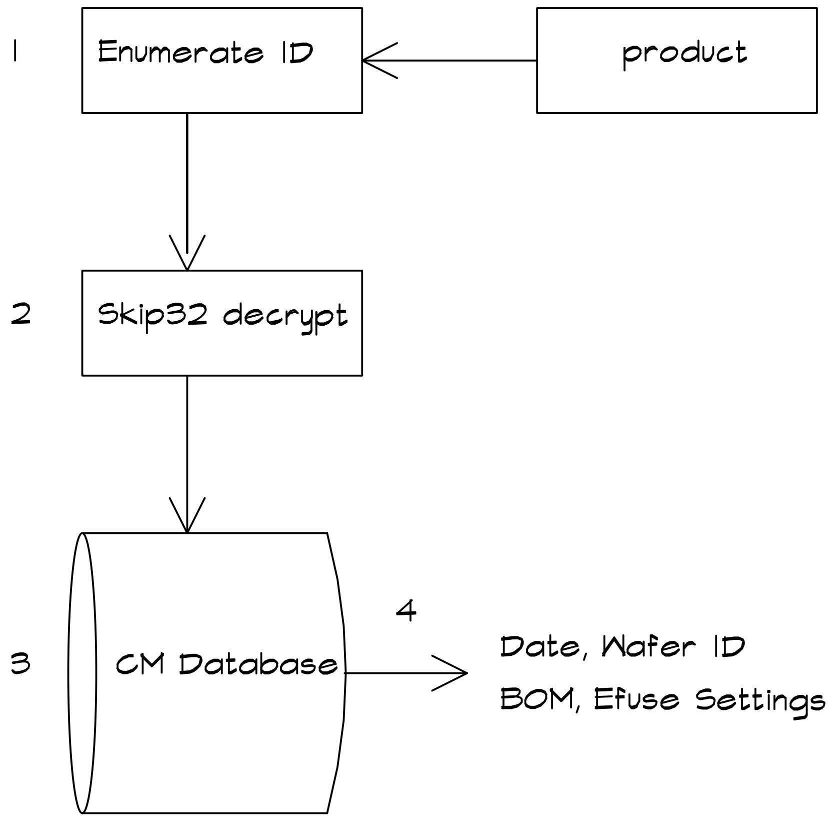

Figure 2: Debug Workflow. 1) The product is enumerated to determine its ID, 2) the ID is decrypted to generate the original build sequence number, 3+4) the database is queried for the record corresponding to the build number to give all the particulars for the cable.

```
    #!)skip32               ; program with no arguments
        usage: skip32 e/d dddddddd
    #!)skip32 e 00000001    ; encode the sequence number "00000001"
        bcc98574
    #!)skip32 e 00000002    ; encode "00000002"
        8403f34e
    #!)skip32 e 00000003    ; encode "00000003"
        ca3d5e97
    #!)skip32 d bcc98574    ; decode ID "bcc98574" back to "00000001"
        00000001
```

3 Appendix: Listing of skip32.c

```
/*
   SKIP32 -- 32 bit block cipher based on SKIPJACK.
   Written by Greg Rose, QUALCOMM Australia, 1999/04/27.
   Modified by Rick Walker, walker@omnisterra.com 10/15/2012

   http://www.mavi1.org/web_security/cryptography/skipjack/skip32.c

   In common: F-table, G-permutation, key schedule.
   Different: 24 round feistel structure.
   Based on:  Unoptimized test implementation of SKIPJACK algorithm
              Panu Rissanen <bande@lut.fi>

  SKIPJACK and KEA Algorithm Specifications Version 2.0 29 May 1998

  Note from the perl implementation at
  http://search.cpan.org/~esh/Crypt-Skip32-0.17/lib/Crypt/Skip32.pm

  "This cipher can be handy for scrambling small (32-bit) values
  when you would like to obscure them while keeping the encrypted
  output size small (also only 32 bits).  One example where
  Crypt::Skip32 has been useful: You have numeric database record
  ids which increment sequentially.  You would like to use them in
  URLs, but you don't want to make it obvious how many X's you have
  in the database by putting the ids directly in the URLs.  You can
  use Crypt::Skip32 to scramble ids and put the resulting 32-bit
  value in URLs (perhaps as 8 hex digits or some other shorter
  encoding).  When a user requests a URL, you can unscramble the
  id to retrieve the object from the database."

  Not copyright, no rights reserved.
*/
```

```
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char   BYTE; /* 8 bits */
typedef unsigned short  WORD; /* 16 bits */

const BYTE ftable[256] = {
0xa3,0xd7,0x09,0x83,0xf8,0x48,0xf6,0xf4,
0xb3,0x21,0x15,0x78,0x99,0xb1,0xaf,0xf9,
0xe7,0x2d,0x4d,0x8a,0xce,0x4c,0xca,0x2e,
0x52,0x95,0xd9,0x1e,0x4e,0x38,0x44,0x28,
0x0a,0xdf,0x02,0xa0,0x17,0xf1,0x60,0x68,
0x12,0xb7,0x7a,0xc3,0xe9,0xfa,0x3d,0x53,
0x96,0x84,0x6b,0xba,0xf2,0x63,0x9a,0x19,
0x7c,0xae,0xe5,0xf5,0xf7,0x16,0x6a,0xa2,
0x39,0xb6,0x7b,0x0f,0xc1,0x93,0x81,0x1b,
0xee,0xb4,0x1a,0xea,0xd0,0x91,0x2f,0xb8,
0x55,0xb9,0xda,0x85,0x3f,0x41,0xbf,0xe0,
0x5a,0x58,0x80,0x5f,0x66,0x0b,0xd8,0x90,
0x35,0xd5,0xc0,0xa7,0x33,0x06,0x65,0x69,
0x45,0x00,0x94,0x56,0x6d,0x98,0x9b,0x76,
0x97,0xfc,0xb2,0xc2,0xb0,0xfe,0xdb,0x20,
0xe1,0xeb,0xd6,0xe4,0xdd,0x47,0x4a,0x1d,
0x42,0xed,0x9e,0x6e,0x49,0x3c,0xcd,0x43,
0x27,0xd2,0x07,0xd4,0xde,0xc7,0x67,0x18,
0x89,0xcb,0x30,0x1f,0x8d,0xc6,0x8f,0xaa,
0xc8,0x74,0xdc,0xc9,0x5d,0x5c,0x31,0xa4,
0x70,0x88,0x61,0x2c,0x9f,0x0d,0x2b,0x87,
0x50,0x82,0x54,0x64,0x26,0x7d,0x03,0x40,
0x34,0x4b,0x1c,0x73,0xd1,0xc4,0xfd,0x3b,
0xcc,0xfb,0x7f,0xab,0xe6,0x3e,0x5b,0xa5,
0xad,0x04,0x23,0x9c,0x14,0x51,0x22,0xf0,
0x29,0x79,0x71,0x7e,0xff,0x8c,0x0e,0xe2,
0x0c,0xef,0xbc,0x72,0x75,0x6f,0x37,0xa1,
0xec,0xd3,0x8e,0x62,0x8b,0x86,0x10,0xe8,
0x08,0x77,0x11,0xbe,0x92,0x4f,0x24,0xc5,
0x32,0x36,0x9d,0xcf,0xf3,0xa6,0xbb,0xac,
0x5e,0x6c,0xa9,0x13,0x57,0x25,0xb5,0xe3,
0xbd,0xa8,0x3a,0x01,0x05,0x59,0x2a,0x46
};


WORD g(BYTE *key, int k, WORD w)
{
    BYTE g1, g2, g3, g4, g5, g6;
    g1 = (w>>8)&0xff;
    g2 = w&0xff;

    g3 = ftable[g2 ^ key[(4*k)%10]] ^ g1;
    g4 = ftable[g3 ^ key[(4*k+1)%10]] ^ g2;
    g5 = ftable[g4 ^ key[(4*k+2)%10]] ^ g3;
    g6 = ftable[g5 ^ key[(4*k+3)%10]] ^ g4;

    return ((g5<<8) + g6);
}

void skip32(BYTE key[10], BYTE buf[4], int encrypt)
{
    int         k; /* round number */
    int         i; /* round counter */
    int         kstep;
    WORD        wl, wr;

    /* sort out direction */
    if (encrypt)
        kstep = 1, k = 0;
    else
        kstep = -1, k = 23;

    /* pack into words */
    wl = (buf[0] << 8) + buf[1];
    wr = (buf[2] << 8) + buf[3];

    /* 24 feistel rounds, doubled up */
    for (i = 0; i < 24/2; ++i) {
        wr ^= g(key, k, wl) ^ k;
        k += kstep;
        wl ^= g(key, k, wr) ^ k;
        k += kstep;
    }

    /* implicitly swap halves while unpacking */
    buf[0] = wr >> 8;   buf[1] = wr & 0xFF;
    buf[2] = wl >> 8;   buf[3] = wl & 0xFF;
}

int sanity_test() {

    // these are the test keys to verify the  implementation:
    BYTE        in[4] = { 0x33,0x22,0x11,0x00 };
    BYTE        key[10] = { 0x00,0x99,0x88,0x77,0x66,0x55,
                        0x44,0x33,0x22,0x11 };
    int          i, encrypt;
    int          bt;

    skip32(key, in, 1);
    if (in[0]!=0x81 || in[1]!=0x9d || in[2]!=0x5f || in[3]!=0x1f) {
        printf("got: %02x%02x%02x%02x\n", in[0], in[1], in[2], in[3]);
        printf("819d5f1f is the answer! Didn't encrypt correctly!\n");
        return 0;
    }
    skip32(key, in, 0);
    if (in[0]!=0x33 || in[1]!=0x22 || in[2]!=0x11 || in[3]!=0x00) {
        printf("%02x%02x%02x%02x\n", in[0], in[1], in[2], in[3]);
        printf("33221100 is the answer! Didn't decrypt correctly!\n");
        return 0;
    }

    return(1);
}

int main(int argc, char *argv[])
{
    // this is the secret key:
    BYTE  key[10] = { 0x52,0x69,0x63,0x6b,0x57,
                        0x61,0x6c,0x6b,0x65,0x72 };

    BYTE  in[4];
    int i, encrypt, bt;

    if (!sanity_test()) {
       exit(1);
    }

    // if we get this far, we passed the test suite and are good to go...

    if (argc != 3) {
        fprintf(stderr, "usage: %s e/d dddddddd\n", argv[0]);
        return 1;
    } else {
        encrypt = (argv[1][0] == 'e');
        for (i = 0; i < 4; ++i) {
            sscanf(&argv[2][i*2], "%02x", &bt);
            in[i] = bt;
            }
        skip32(key, in, encrypt);
        printf("%02x%02x%02x%02x\n", in[0], in[1], in[2], in[3]);
    }
    return 0;
}
```