One-Way Hash Functions

Using cryptographic algorithms for hashing

Bruce Schneier

Pattern matching is usually a problem more suited for elementary programming classes than *Dr. Dobb's Journal*. But what if the strings are 100 Kbytes large? What if you have to search through a million of them looking for matches? And what if you have to try and match 1000 of these strings each day? With a simple string comparison algorithm, you will have to compare one billion strings, each requiring from 1 to 100,000 individual byte comparisons. And don't forget the hellacious storage and retrieval requirements. There has to be a better way.

Probabilistic algorithms are useful for jobs like this. They work — most of the time — and they have been applied to finding large random prime numbers, solutions to particularly convoluted graph theory problems, and pattern matching. The algorithm I am going to present only fails in one out of 2^{128} (that's about 3×10^{38}) tries. There isn't a piece of commercial software on the market that fails less often. You're much more likely to get a disk error reading the program, or a power surge that fries your whole system, than you are to have this algorithm fail. By the time this algorithm fails, our species will most likely have long killed itself off, or at least mutated into some different species that has better things to do with its time than sit around comparing a million 100-Kbyte strings.

We're going to take the computer analog of fingerprints; something small that uniquely identifies something much larger. Hash functions, which are functions that take a large input string and produce a smaller output string, perform the task nicely.

Hash Functions

One of the simplest hash functions is to take the XOR of every 128-bit block in the file, as in Example 1. This works well with random data; each 128-bit hash value is equally likely. However, with more rigidly formatted data, such as ASCII, the results are less than ideal. In most normal text files, the high-order bit of each byte is always 0. Therefore, 8 bits in the hash will always be 0. This means that the effective number of possible hashes is far less than 2^{128} (or at least that each possible hash value is not equally likely), and different files are more likely to produce identical hashes.

An easy way to get around this is to roll the hash value 1 bit after each XOR. This way each bit is more likely to be throughly randomized by the time the entire file has been hashed, regardless of how the data looks beforehand. The program in Example 2 rolls each longword 1 bit to the right.

If the hash function is only going to deal with ASCII data, a further improvement might be to ignore spaces. This way, "HELLO WORLD" hashes to the same value as "HELLOWORLD." Of course, it also means that "The house holds one at seven bells" hashes to the same value as "The household son eats even bells," but nothing is perfect. Maybe the program should count the first space, but ignore any additional ones. Additional modifications that ignore punctuation and case might also be useful, or they might be detrimental. It all depends on the type of data involved.

These hash functions work great as long as there isn't a malicious user trying to gum up the works. Creating a file that hashes to a particular value is very easy with any of these functions. It's only slightly harder with more complicated functions. Usually, all that is required is to calculate the hash of any particular file, and then append a 128-bit string

Bruce has an MS in Computer Science and has worked in computer and data security for a number of public and private concerns. He can be reached at 730 Fair Oaks Ave., Oak Park, IL 60302.

at the end of it to force the new file to hash to whatever value you'd like.

In certain applications this problem is very serious, but it can be solved by using a "one-way hash function." By "one-way," I mean that given an input file, it is very easy to calculate the hash value. However, given a particular hash value, it is, for all practical purposes, impossible to produce an input file that hashes to that value. The function has a trap door: It's easy to go one way, but hard to go back the other unless you know the "secret."

The MD5 Algorithm

In theory, any cryptographic algorithm can be used as a one-way hash function. For example, the Digital Encryption Standard (DES) algorithm (see the "C Programming" column, *DDJ*, September 1990), endorsed by the National Security Agency and used by countless government agencies and business interests, works quite well. Encrypt the entire file using the DES block-chaining mode, and the two final 64-bit encrypted blocks make up the hash. (ANSI X 9.9 suggests using only the last 32 bits of the last encrypted block, but that seems inadequate.) It is random, not reversible, and every bit of the hash is dependent on every bit of the input file. But in software, DES is slow; the algorithm is not at all optimized for this task. And even worse, if the DES key is compromised, then everyone knows the secret to the trap door and the one-way property of the hash function is lost.

An alternate algorithm, shown in Listing One (page 150), is called MD5. (Listing Two, page 151, is the header file for MD5.) Cryptographer Ron Rivest of MIT (he's the "R" in the RSA public-key algorithm) invented MD5 specifically as a one-way hash function. His personal needs for the algorithm revolve more around secure digital signatures than pattern matching, but we will come back to that later.

We're going to take the computer analog of fingerprints; something small that uniquely identifies something much larger

MD5 produces a 128-bit hash (or "Message Digest," hence the name) of an input file. The algorithm was designed for speed, simplicity, and compactness on 32-bit architectures. It is based on a simple set of primitive operations (that is, LOAD, ADD, XOR) on 32-bit operands. The algorithm also works on 8- and 16-bit microprocessors, albeit significantly slower.

The heart of the algorithm is *MD5Update*, which processes the input file in 512-bit chunks. *Transform* performs the basic MD5 algorithm. *MD5Init* kicks off the whole process. *MD5Final* is primarily concerned with the partial block at the end of the input file (only the rare input file can be divided exactly into 512-bit blocks). The final block is padded with a single *1* and a string of *0*s, with the last word reserved for a binary representation of the number of bits in the original input file. After the algorithm churns away, the resultant hash is stored in *mdContext*.

It is straightforward to implement MD5 to solve the pattern matching problem at the beginning of this article. First, create and store a hash of each of the million 100-Kbyte input strings. Each one is only 16 bytes long, so you only need 16 Mbytes of storage space instead of 100 gigabytes. Every time you have a new test string, compare its hashes with those stored. Not only are the storage requirements four orders of magnitude smaller, you only have to make one 128-bit comparison per string pair. You'll get a false match every 10^{27} years or so, so you might want to add code that explicitly checks strings that hash to the same value. I wouldn't even bother.

Putting One-Way Hash Functions to Work

One-way hash functions can be used to detect viruses, which make their living by surreptitiously modifying programs. Numerous security programs take simple hashes called Cyclic Redundancy Checks (CRCs). These programs hash clean files, and then compare the stored hash with a new hash of the file. If a virus infected the file, its hash would be different. The problem with most of these virus-detection programs is that a clever virus can make sure any virus-infected file hashes to the same value as the clean file. It would be harder to do this with CRC fingerprinters than with the simple hash programs listed earlier, but it would still be possible. MD5 makes it practically impossible. Even if the virus had complete knowledge of the MD5 algorithm, it could not modify itself so that the infected program would hash to the same value. All it could do would be to randomly append bits to the file and try to duplicate the hash by brute force. But even if we allow the virus to try a million possible hashes per second, it would take 10^{25}

```
main (int argc, char *argv[])  {
    unsigned long hash[4] = {0, 0, 0, 0}, data[4];
    FILE *fp;
    int i;
    if ((fp == fopen (argv[1], "rb")) != NULL)  {
        while ((fread (data, 4, 4, fp) != NULL)
            for (i=0; i<4; i++)
                hash[i] ^= data[i];
        fclose (fp);
        for (i=0; i<4; i++)

            printf ("%08lx", hash[i]);
        printf ("\n");
    }
}
```

Example 1: *A simple hash function that uses the XOR of every 128-bit block in the file*

```
main (int argc, char *argv[])  {
   unsigned long hash[4] = {0, 0, 0, 0}, data[4];

   FILE *fp;
   int i;
   if ((fp == fopen (argv[1], "rb")) != NULL)  {
    while ((fread (data, 4, 4, fp) != NULL)
       for (i=0; i<4; i++)  {
       hash[i] ^= data[i];
       hash[i] = hash[i]>>1 ^ hash[i]<<31;
        }
    fclose (fp);
    for (i=0; i<4; i++)
         printf ("%08lx", hash[i]);
    printf ("\n");
     }
  }
```

Example 2: *Rolling each longword 1 bit to the right*

years for it to successfully infect the file. I'm willing to live with the possibility of one successful virus infection on my system between now and when the sun goes nova.

As I mentioned, one-way hash functions have uses in digital signatures. A digital signature scheme is where one party can "sign" a document in such a way that no one except the signer could sign the document, and anyone can verify that the signer actually did sign the document. Most digital signature schemes involve public-key cryptography; they are computational nightmares and often painfully slow. Speed increases drastically if the digital signature algorithm operates on a hash of the document rather than the document itself. Because the chances of two documents having the same hash are only 1 in 2^{128}, anyone can safely equate a signature of the hash with a signature of the document. If a conventional hash function were used, it would be a trivial matter to invent multiple documents that hashed to the same value, so that anyone signing a particular document would be duped into signing a multitude of documents. The protocol just could not work without one-way hash functions.

One-way hash functions can also be used by an archival system to verify the existence of documents without storing their contents. The central database could just store the hashes of files. It doesn't have to see the files at all; users submit their hashes to the database, and the database time stamps the submissions and stores them. If there is any disagreement in the future about who created a document and when, the database could resolve it by finding the hash in its files. Enhancements to this scheme, such as having the database use a digital signature protocol and sign a concatenation of the hash and a date/time code, would make this protocol even more useful. This has vast implications with regard to privacy: Someone could copyright a document but still keep it secret. Only if he wished to prove his copyright would he have to make it public.

Someone could copyright a document but still keep it secret

The source code has been successfully compiled and run under MS-DOS, Unix, and Macintosh OS. Because of space constraints, I've included with this article only the C source that implements the MD5 algorithm; the include files, makefiles, and sample test routines are available electronically as described on page 3. RSA Data Security Inc. has put the MD5 algorithm into the public domain so that anyone can use it. Their only requirement is that any copies of their source code or documentation (but not different implementations of the mathematical algorithm) retain the copyright notice at the beginning of Listing One. Rivest cautions that while the algorithm improves over MD4, it is still new and has not been completely cryptoanalyzed. It is always possible that some clever person will find a way to break it, but over the years as more clever people try and fail, confidence in the algorithm will grow.

DDJ

Listing One

```
/**********************************************************************
 ** md5.c -- the source code for MD5 routines                        **
 ** RSA Data Security, Inc. MD5 Message-Digest Algorithm             **
 ** Created: 2/17/90 RLR                                            **
 ** Revised: 1/91 SRD,AJ,BSK,JT Reference C Version                  **
 ** Revised (for MD5): RLR 4/27/91                                   **
 **********************************************************************
 ** Copyright (C) 1990, RSA Data Security, Inc. All rights reserved. **
 ** License to copy and use this software is granted provided that   **
 ** it is identified as the "RSA Data Security, Inc. MD5 Message-    **
 ** Digest Algorithm" in all material mentioning or referencing this **
 ** software or this function.                                       **
 ** License is also granted to make and use derivative works         **
 ** provided that such works are identified as "derived from the RSA **
 ** Data Security, Inc. MD5 Message-Digest Algorithm" in all         **
 ** material mentioning or referencing the derived work.             **
 ** RSA Data Security, Inc. makes no representations concerning      **
 ** either the merchantability of this software or the suitability   **
 ** of this software for any particular purpose.  It is provided "as **
 ** is" without express or implied warranty of any kind.             **
 ** These notices must be retained in any copies of any part of this **
 ** documentation and/or software.                                   **
 **********************************************************************/

#include "md5.h"

/* forward declaration */
static void Transform ();

static unsigned char PADDING[64] = {
  0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

/* F, G, H and I are basic MD5 functions */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

/* ROTATE_LEFT rotates x left n bits */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4 */
/* Rotation is separate from addition to prevent recomputation */
#define FF(a, b, c, d, x, s, ac) \
  {(a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
   (a) = ROTATE_LEFT ((a), (s)); \
   (a) += (b); \
  }
#define GG(a, b, c, d, x, s, ac) \
  {(a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
   (a) = ROTATE_LEFT ((a), (s)); \
   (a) += (b); \
  }
#define HH(a, b, c, d, x, s, ac) \
  {(a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
   (a) = ROTATE_LEFT ((a), (s)); \
   (a) += (b); \
  }
#define II(a, b, c, d, x, s, ac) \
  {(a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
   (a) = ROTATE_LEFT ((a), (s)); \
   (a) += (b); \
  }

/* The routine MD5Init initializes the message-digest context
   mdContext. All fields are set to zero. */
void MD5Init (mdContext)
MD5_CTX *mdContext;
{
  mdContext->i[0] = mdContext->i[1] = (UINT4)0;

  /* Load magic initialization constants.
   */
  mdContext->buf[0] = (UINT4)0x67452301;
  mdContext->buf[1] = (UINT4)0xefcdab89;
  mdContext->buf[2] = (UINT4)0x98badcfe;
  mdContext->buf[3] = (UINT4)0x10325476;
}

/* The routine MD5Update updates the message-digest context to
   account for the presence of each of the characters inBuf[0..inLen-1]
   in the message whose digest is being computed. */
void MD5Update (mdContext, inBuf, inLen)
MD5_CTX *mdContext;
unsigned char *inBuf;
unsigned int inLen;
{
  UINT4 in[16];
  int mdi;
  unsigned int i, ii;

  /* compute number of bytes mod 64 */
  mdi = (int)((mdContext->i[0] >> 3) & 0x3F);

  /* update number of bits */
  if ((mdContext->i[0] + ((UINT4)inLen << 3)) < mdContext->i[0])
    mdContext->i[1]++;
  mdContext->i[0] += ((UINT4)inLen << 3);
```

```
  mdContext->i[1] += ((UINT4)inLen >> 29);
  while (inLen--) {
    /* add new character to buffer, increment mdi */
    mdContext->in[mdi++] = *inBuf++;

    /* transform if necessary */
    if (mdi == 0x40) {
      for (i = 0, ii = 0; i < 16; i++, ii += 4)
        in[i] = (((UINT4)mdContext->in[ii+3]) << 24) |
                (((UINT4)mdContext->in[ii+2]) << 16) |
                (((UINT4)mdContext->in[ii+1]) << 8) |
                ((UINT4)mdContext->in[ii]);
      Transform (mdContext->buf, in);
      mdi = 0;
    }
  }
}

/* The routine MD5Final terminates the message-digest computation and
   ends with the desired message digest in mdContext->digest[0...15]. */
void MD5Final (mdContext)
MD5_CTX *mdContext;
{
  UINT4 in[16];
  int mdi;
  unsigned int i, ii;
  unsigned int padLen;

  /* save number of bits */
  in[14] = mdContext->i[0];
  in[15] = mdContext->i[1];

  /* compute number of bytes mod 64 */
  mdi = (int)((mdContext->i[0] >> 3) & 0x3F);

  /* pad out to 56 mod 64 */
  padLen = (mdi < 56) ? (56 - mdi) : (120 - mdi);
  MD5Update (mdContext, PADDING, padLen);

  /* append length in bits and transform */
  for (i = 0, ii = 0; i < 14; i++, ii += 4)
    in[i] = (((UINT4)mdContext->in[ii+3]) << 24) |
            (((UINT4)mdContext->in[ii+2]) << 16) |
            (((UINT4)mdContext->in[ii+1]) << 8) |
            ((UINT4)mdContext->in[ii]);
  Transform (mdContext->buf, in);

  /* store buffer in digest */
  for (i = 0, ii = 0; i < 4; i++, ii += 4) {
    mdContext->digest[ii] = (unsigned char)(mdContext->buf[i] & 0xFF);
    mdContext->digest[ii+1] =
      (unsigned char)((mdContext->buf[i] >> 8) & 0xFF);
    mdContext->digest[ii+2] =
      (unsigned char)((mdContext->buf[i] >> 16) & 0xFF);
    mdContext->digest[ii+3] =
      (unsigned char)((mdContext->buf[i] >> 24) & 0xFF);
  }
}

/* Basic MD5 step. Transforms buf based on in. */
static void Transform (buf, in)
UINT4 *buf;
UINT4 *in;
{
  UINT4 a = buf[0], b = buf[1], c = buf[2], d = buf[3];

  /* Round 1 */
#define S11 7
#define S12 12
#define S13 17
#define S14 22
  FF ( a, b, c, d, in[ 0], S11, 0xd76aa478); /* 1 */
  FF ( d, a, b, c, in[ 1], S12, 0xe8c7b756); /* 2 */
  FF ( c, d, a, b, in[ 2], S13, 0x242070db); /* 3 */
  FF ( b, c, d, a, in[ 3], S14, 0xc1bdceee); /* 4 */
  FF ( a, b, c, d, in[ 4], S11, 0xf57c0faf); /* 5 */
  FF ( d, a, b, c, in[ 5], S12, 0x4787c62a); /* 6 */
  FF ( c, d, a, b, in[ 6], S13, 0xa8304613); /* 7 */
  FF ( b, c, d, a, in[ 7], S14, 0xfd469501); /* 8 */
  FF ( a, b, c, d, in[ 8], S11, 0x698098d8); /* 9 */
  FF ( d, a, b, c, in[ 9], S12, 0x8b44f7af); /* 10 */
  FF ( c, d, a, b, in[10], S13, 0xffff5bb1); /* 11 */
  FF ( b, c, d, a, in[11], S14, 0x895cd7be); /* 12 */
  FF ( a, b, c, d, in[12], S11, 0x6b901122); /* 13 */
  FF ( d, a, b, c, in[13], S12, 0xfd987193); /* 14 */
  FF ( c, d, a, b, in[14], S13, 0xa679438e); /* 15 */
  FF ( b, c, d, a, in[15], S14, 0x49b40821); /* 16 */

  /* Round 2 */
#define S21 5
#define S22 9
#define S23 14
#define S24 20
  GG ( a, b, c, d, in[ 1], S21, 0xf61e2562); /* 17 */
  GG ( d, a, b, c, in[ 6], S22, 0xc040b340); /* 18 */
  GG ( c, d, a, b, in[11], S23, 0x265e5a51); /* 19 */
  GG ( b, c, d, a, in[ 0], S24, 0xe9b6c7aa); /* 20 */
  GG ( a, b, c, d, in[ 5], S21, 0xd62f105d); /* 21 */
  GG ( d, a, b, c, in[10], S22,  0x2441453); /* 22 */
  GG ( c, d, a, b, in[15], S23, 0xd8a1e681); /* 23 */
  GG ( b, c, d, a, in[ 4], S24, 0xe7d3fbc8); /* 24 */
  GG ( a, b, c, d, in[ 9], S21, 0x21e1cde6); /* 25 */
  GG ( d, a, b, c, in[14], S22, 0xc33707d6); /* 26 */
  GG ( c, d, a, b, in[ 3], S23, 0xf4d50d87); /* 27 */
  GG ( b, c, d, a, in[ 8], S24, 0x455a14ed); /* 28 */
  GG ( a, b, c, d, in[13], S21, 0xa9e3e905); /* 29 */
  GG ( d, a, b, c, in[ 2], S22, 0xfcefa3f8); /* 30 */
  GG ( c, d, a, b, in[ 7], S23, 0x676f02d9); /* 31 */
  GG ( b, c, d, a, in[12], S24, 0x8d2a4c8a); /* 32 */
  /* Round 3 */
#define S31 4
#define S32 11
#define S33 16
#define S34 23
  HH ( a, b, c, d, in[ 5], S31, 0xfffa3942); /* 33 */
  HH ( d, a, b, c, in[ 8], S32, 0x8771f681); /* 34 */
  HH ( c, d, a, b, in[11], S33, 0x6d9d6122); /* 35 */
  HH ( b, c, d, a, in[14], S34, 0xfde5380c); /* 36 */
  HH ( a, b, c, d, in[ 1], S31, 0xa4beea44); /* 37 */
  HH ( d, a, b, c, in[ 4], S32, 0x4bdecfa9); /* 38 */
  HH ( c, d, a, b, in[ 7], S33, 0xf6bb4b60); /* 39 */
  HH ( b, c, d, a, in[10], S34, 0xbebfbc70); /* 40 */
  HH ( a, b, c, d, in[13], S31, 0x289b7ec6); /* 41 */
  HH ( d, a, b, c, in[ 0], S32, 0xeaa127fa); /* 42 */
  HH ( c, d, a, b, in[ 3], S33, 0xd4ef3085); /* 43 */
  HH ( b, c, d, a, in[ 6], S34,  0x4881d05); /* 44 */
  HH ( a, b, c, d, in[ 9], S31, 0xd9d4d039); /* 45 */
  HH ( d, a, b, c, in[12], S32, 0xe6db99e5); /* 46 */
  HH ( c, d, a, b, in[15], S33, 0x1fa27cf8); /* 47 */
  HH ( b, c, d, a, in[ 2], S34, 0xc4ac5665); /* 48 */

  /* Round 4 */
#define S41 6
#define S42 10
#define S43 15
#define S44 21
  II ( a, b, c, d, in[ 0], S41, 0xf4292244); /* 49 */
  II ( d, a, b, c, in[ 7], S42, 0x432aff97); /* 50 */
  II ( c, d, a, b, in[14], S43, 0xab9423a7); /* 51 */
  II ( b, c, d, a, in[ 5], S44, 0xfc93a039); /* 52 */
  II ( a, b, c, d, in[12], S41, 0x655b59c3); /* 53 */
  II ( d, a, b, c, in[ 3], S42, 0x8f0ccc92); /* 54 */
  II ( c, d, a, b, in[10], S43, 0xffeff47d); /* 55 */
  II ( b, c, d, a, in[ 1], S44, 0x85845dd1); /* 56 */
  II ( a, b, c, d, in[ 8], S41, 0x6fa87e4f); /* 57 */
  II ( d, a, b, c, in[15], S42, 0xfe2ce6e0); /* 58 */
  II ( c, d, a, b, in[ 6], S43, 0xa3014314); /* 59 */
  II ( b, c, d, a, in[13], S44, 0x4e0811a1); /* 60 */
  II ( a, b, c, d, in[ 4], S41, 0xf7537e82); /* 61 */
  II ( d, a, b, c, in[11], S42, 0xbd3af235); /* 62 */
  II ( c, d, a, b, in[ 2], S43, 0x2ad7d2bb); /* 63 */
  II ( b, c, d, a, in[ 9], S44, 0xeb86d391); /* 64 */

  buf[0] += a;
  buf[1] += b;
  buf[2] += c;
  buf[3] += d;
}

/*****************************************************************
 ** End of md5.c                                                **
 *****************************************************************/
```

End Listing One

Listing Two

```
/*****************************************************************
 ** md5.h -- header file for implementation of MD5              **
 ** RSA Data Security, Inc. MD5 Message-Digest Algorithm        **
 ** Created: 2/17/90 RLR                                        **
 ** Revised: 12/27/90 SRD,AJ,BSK,JT Reference C version         **
 ** Revised (for MD5): RLR 4/27/91                              **
 *****************************************************************/
/* typedef a 32-bit type */
typedef unsigned long int UINT4;

/* Data structure for MD5 (Message-Digest) computation */
typedef struct {
  UINT4 i[2];                   /* number of _bits_ handled mod 2^64 */
  UINT4 buf[4];                                    /* scratch buffer */
  unsigned char in[64];                              /* input buffer */
  unsigned char digest[16];     /* actual digest after MD5Final call */
} MD5_CTX;

void MD5Init ();
void MD5Update ();
void MD5Final ();

/*****************************************************************
 ** End of md5.h                                                **
 *****************************************************************/
```

End Listings

C Language Q&A

#27

Q: What is the "ANSI C Standard"?

A: In 1983, the American National Standards Institute commissioned a committee, X3J11, to standardize the C language. After a long, arduous process, the committee's work was finally ratified as an American National Standard, X3.159-1989, on December 14, 1989, and published in the spring of 1990. The Standard has also been adopted as ISO/IEC 9899:1990.