The Pluto Scarab

Bret Mulvey's site about math, programming, electronics, physics, gaming, and sundry. Especially sundry.

Hash Functions, continued

This is a continuation of a previous post. We just finished evaluating the FNV hash using our uniform distribution and avalanche criteria. Now we look at another hash function.

Bob Jenkins' Hash

Figure 1

Bob Jenkins hash function with 96-bit internal state.

```
public class Jenkins96 : HashFunction
{
    uint a, b, c;

    void Mix()
    {
        a -= b; a -= c; a ^= (c>>13);
        b -= c; b -= a; b ^= (a<<8);
        c -= a; c -= b; c ^= (b>>13);
        a -= b; a -= c; a ^= (c>>12);
        b -= c; b -= a; b ^= (a<<16);
        c -= a; c -= b; c ^= (b>>5);
        a -= b; a -= c; a ^= (c>>3);
        b -= c; b -= a; b ^= (a<<10);
        c -= a; c -= b; c ^= (b>>15);
    }

    public override uint ComputeHash(byte[] data)
    {
        int len = data.Length;
        a = b = 0x9e3779b9;
        c = 0;
```

```
int i = 0;
while (i + 12 <= len)
{
    a += (uint)data[i++] |
        ((uint)data[i++] << 8) |
        ((uint)data[i++] << 16) |
        ((uint)data[i++] << 24);
    b += (uint)data[i++] |
        ((uint)data[i++] << 8) |
        ((uint)data[i++] << 16) |
        ((uint)data[i++] << 24);
    c += (uint)data[i++] |
        ((uint)data[i++] << 8) |
        ((uint)data[i++] << 16) |
        ((uint)data[i++] << 24);
    Mix();
}
c += (uint) len;
if (i < len)
    a += data[i++];
if (i < len)
    a += (uint)data[i++] << 8;
if (i < len)
    a += (uint)data[i++] << 16;
if (i < len)
    a += (uint)data[i++] << 24;
if (i < len)
    b += (uint)data[i++];
if (i < len)
    b += (uint)data[i++] << 8;
if (i < len)
    b += (uint)data[i++] << 16;
if (i < len)
    b += (uint)data[i++] << 24;
if (i < len)
    c += (uint)data[i++] << 8;
if (i < len)
    c += (uint)data[i++] << 16;
if (i < len)
    c += (uint)data[i++] << 24;
Mix();
return c;
```

```
        }
    }
```

This hash function is described here. It uses shifts, adds, and xors in its mixing function, and processes keys twelve octets at a time. The implementation in C++ is more elegant as C++ allow case statements to fall through, but C# does not.

Achieving avalanche seemed to be an important design goal for the hash author, and our results below confirm that this hash has excellent avalanche behavior. Another design goal was achieving processor parallelism in the mixing function. Control over low-level processor timing details is probably lost in a memory-managed and JIT-compiled language like C#, nevertheless this hash does its job well.

Figure 1 shows the listing for the Bob Jenkins hash. Since it processes the key twelve octets at a time, the post-processing step is slightly more complex than some other hash functions since it has to handle the partial final block. The post-processing step is also interesting in that it mixes the key length into the hash value. This is common practice for cryptographic hashes such as MD5 and SHA-1, but not common for hashes used for table lookup and other non-cryptographic uses.

Bob Jenkins has another shift-add-xor hash that processes keys one octet at a time and has a much simpler implementation. We'll look at that hash on a subsequent page.

Uniform Distribution Test

We examine the distribution of numbers derived from lower and upper bits of the hash output in sizes of 1 through 16 bits.

Figure 2 shows the results of this test for the Bob Jenkins hash. This test indicates that this hash produces uniformly distributed values for hash tables that are a power of two, up to at least 2^{16}, when the key octets are uniformly distributed, distributed similar to alphabetic text, or sparsely distributed.

The results show a couple of values in the 5% significance range, but these are due to chance and do not appear to indicate any weakness in the hash distribution.

Figure 2

χ^2 test results for Bob Jenkins' hash.

```
      Uniform keys     Text keys      Sparse keys
```

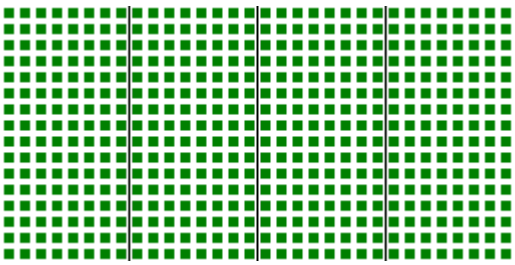
Bits	Lower	Upper	Lower	Upper	Lower	Upper
1	0.203	0.396	0.258	0.777	0.777	0.090
2	0.468	0.321	0.433	0.585	0.286	0.650
3	0.400	0.435	0.421	0.692	0.563	0.233
4	0.341	0.548	0.041	0.400	0.679	0.513
5	0.767	0.924	0.539	0.274	0.043	0.443
6	0.343	0.281	0.330	0.769	0.289	0.834
7	0.585	0.945	0.675	0.093	0.802	0.517
8	0.854	0.861	0.816	0.808	0.048	0.792
9	0.874	0.816	0.925	0.411	0.590	0.979
10	0.735	0.501	0.810	0.605	0.325	0.495
11	0.399	0.126	0.064	0.945	0.315	0.527
12	0.355	0.125	0.439	0.803	0.198	0.819
13	0.620	0.407	0.166	0.183	0.093	0.169
14	0.933	0.381	0.044	0.540	0.532	0.375
15	0.619	0.634	0.828	0.220	0.804	0.767
16	0.636	0.870	0.525	0.232	0.360	0.102

Avalanche Test

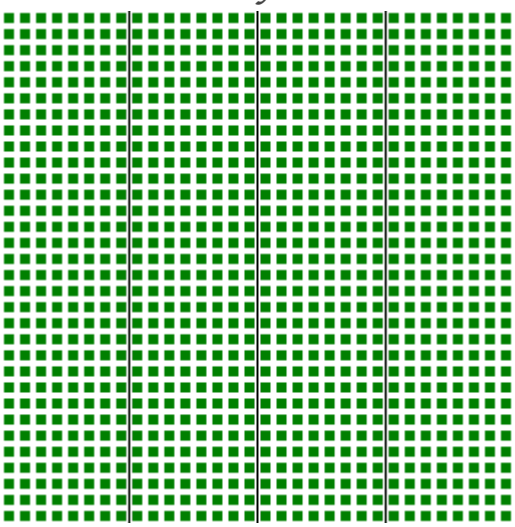
Figure 3

Avalanche behavior of the Bob Jenkins hash.

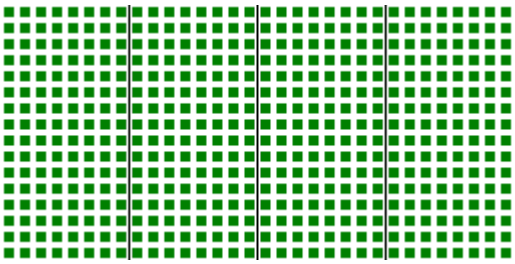
For two-octet keys:



For four-octet keys:



For 256-octet keys:



We examine the diffusion of input bits into different bit positions in the output.

Figure 3 shows the results of this test for the Bob Jenkins hash. This test indicates that this hash has excellent avalanche behavior. It is unlikely that there are any specific classes of keys that will cause problems with this function, as key bits from every position appear to be dispersed well into all hash bit positions, even for very short hash keys.

One fact not shown here is the avalanche behavior for the full 96 bits of the mixing function. In the mixing function, the MSB of variable *b* is not distributed at all into the lower 12 bits of either the *a* or *b* variables until the next mixing step. There is also incomplete avalanche of other bits into the bits of *a* and *b*. That shouldn't be a concern, however, because these variables are only used as intermediate values in the hash calculation and only variable *c* is used as the final result. All bits of *a* and *b* do mix into all bit positions in *c*.

Conclusion

This hash function by Bob Jenkins should be suitable for general purpose use, either for hash table lookup, basic file fingerprinting, or other non-cryptographic uses. Both the upper and lower bits are uniformly distributed. The hash function achieves avalanche for every bit combination tested.

The Use of Substitution Boxes in Hash Functions

Figure 4

A simple, fast, and effective hash function. Because of the one-kilobyte S-box you probably won't be able to remember this one to write down on a napkin later.

```
public class SBoxHash : HashFunction
{
    uint[] sbox = new uint[]
    {
    0xF53E1837, 0x5F14C86B, 0x9EE3964C, 0xFA796D53,
```

```
0x32223FC3, 0x4D82BC98, 0xA0C7FA62, 0x63E2C982,
0x24994A5B, 0x1ECE7BEE, 0x292B38EF, 0xD5CD4E56,
0x514F4303, 0x7BE12B83, 0x7192F195, 0x82DC7300,
0x084380B4, 0x480B55D3, 0x5F430471, 0x13F75991,
0x3F9CF22C, 0x2FE0907A, 0xFD8E1E69, 0x7B1D5DE8,
0xD575A85C, 0xAD01C50A, 0x7EE00737, 0x3CE981E8,
0x0E447EFA, 0x23089DD6, 0xB59F149F, 0x13600EC7,
0xE802C8E6, 0x670921E4, 0x7207EFF0, 0xE74761B0,
0x69035234, 0xBFA40F19, 0xF63651A0, 0x29E64C26,
0x1F98CCA7, 0xD957007E, 0xE71DDC75, 0x3E729595,
0x7580B7CC, 0xD7FAF60B, 0x92484323, 0xA44113EB,
0xE4CBDE08, 0x346827C9, 0x3CF32AFA, 0x0B29BCF1,
0x6E29F7DF, 0xB01E71CB, 0x3BFBC0D1, 0x62EDC5B8,
0xB7DE789A, 0xA4748EC9, 0xE17A4C4F, 0x67E5BD03,
0xF3B33D1A, 0x97D8D3E9, 0x09121BC0, 0x347B2D2C,
0x79A1913C, 0x504172DE, 0x7F1F8483, 0x13AC3CF6,
0x7A2094DB, 0xC778FA12, 0xADF7469F, 0x21786B7B,
0x71A445D0, 0xA8896C1B, 0x656F62FB, 0x83A059B3,
0x972DFE6E, 0x4122000C, 0x97D9DA19, 0x17D5947B,
0xB1AFFD0C, 0x6EF83B97, 0xAF7F780B, 0x4613138A,
0x7C3E73A6, 0xCF15E03D, 0x41576322, 0x672DF292,
0xB658588D, 0x33EBEFA9, 0x938CBF06, 0x06B67381,
0x07F192C6, 0x2BDA5855, 0x348EE0E8, 0x19DBB6E3,
0x3222184B, 0xB69D5DBA, 0x7E760B88, 0xAF4D8154,
0x007A51AD, 0x35112500, 0xC9CD2D7D, 0x4F4FB761,
0x694772E3, 0x694C8351, 0x4A7E3AF5, 0x67D65CE1,
0x9287DE92, 0x2518DB3C, 0x8CB4EC06, 0xD154D38F,
0xE19A26BB, 0x295EE439, 0xC50A1104, 0x2153C6A7,
0x82366656, 0x0713BC2F, 0x6462215A, 0x21D9BFCE,
0xBA8EACE6, 0xAE2DF4C1, 0x2A8D5E80, 0x3F7E52D1,
0x29359399, 0xFEA1D19C, 0x18879313, 0x455AFA81,
0xFADFE838, 0x62609838, 0xD1028839, 0x0736E92F,
0x3BCA22A3, 0x1485B08A, 0x2DA7900B, 0x852C156D,
0xE8F24803, 0x00078472, 0x13F0D332, 0x2ACFD0CF,
0x5F747F5C, 0x87BB1E2F, 0xA7EFCB63, 0x23F432F0,
0xE6CE7C5C, 0x1F954EF6, 0xB609C91B, 0x3B4571BF,
0xEED17DC0, 0xE556CDA0, 0xA7846A8D, 0xFF105F94,
0x52B7CCDE, 0x0E33E801, 0x664455EA, 0xF2C70414,
0x73E7B486, 0x8F830661, 0x8B59E826, 0xBB8AEDCA,
0xF3D70AB9, 0xD739F2B9, 0x4A04C34A, 0x88D0F089,
0xE02191A2, 0xD89D9C78, 0x192C2749, 0xFC43A78F,
0x0AAC88CB, 0x9438D42D, 0x9E280F7A, 0x36063802,
0x38E8D018, 0x1C42A9CB, 0x92AAFF6C, 0xA24820C5,
```

```
        0x007F077F, 0xCE5BC543, 0x69668D58, 0x10D6FF74,
        0xBE00F621, 0x21300BBE, 0x2E9E8F46, 0x5ACEA629,
        0xFA1F86C7, 0x52F206B8, 0x3EDF1A75, 0x6DA8D843,
        0xCF719928, 0x73E3891F, 0xB4B95DD6, 0xB2A42D27,
        0xEDA20BBF, 0x1A58DBDF, 0xA449AD03, 0x6DDEF22B,
        0x900531E6, 0x3D3BFF35, 0x5B24ABA2, 0x472B3E4C,
        0x387F2D75, 0x4D8DBA36, 0x71CB5641, 0xE3473F3F,
        0xF6CD4B7F, 0xBF7D1428, 0x344B64D0, 0xC5CDFCB6,
        0xFE2E0182, 0x2C37A673, 0xDE4EB7A3, 0x63FDC933,
        0x01DC4063, 0x611F3571, 0xD167BFAF, 0x4496596F,
        0x3DEE0689, 0xD8704910, 0x7052A114, 0x068C9EC5,
        0x75D0E766, 0x4D54CC20, 0xB44ECDE2, 0x4ABC653E,
        0x2C550A21, 0x1A52C0DB, 0xCFED03D0, 0x119BAFE2,
        0x876A6133, 0xBC232088, 0x435BA1B2, 0xAE99BBFA,
        0xBB4F08E4, 0xA62B5F49, 0x1DA4B695, 0x336B84DE,
        0xDC813D31, 0x00C134FB, 0x397A98E6, 0x151F0E64,
        0xD9EB3E69, 0xD3C7DF60, 0xD2F2C336, 0x2DDD067B,
        0xBD122835, 0xB0B3BD3A, 0xB0D54E46, 0x8641F1E4,
        0xA0B38F96, 0x51D39199, 0x37A6AD75, 0xDF84EE41,
        0x3C034CBA, 0xACDA62FC, 0x11923B8B, 0x45EF170A,
    };

    public override uint ComputeHash(byte[] data)
    {
        uint hash = 0;
        foreach (byte b in data)
        {
            hash ^= sbox[b];
            hash *= 3;
        }
        return hash;
    }
}
```

A substitution box (or S-box) is simply a lookup-table. Use of an S-box provides a simple means for confusing the relationship between the input keys and the hash result.

S-boxes are used extensively in cryptographic hashes, and in those applications the values used in the S-box are often chosen very carefully to avoid specific classes of cryptographic attack. The values chosen for the S-box in this simple example were generated randomly.

Figure 4 shows a hash function based on an S-box. Although this function takes more memory than the other functions we've looked at, the implementation of the function is very simple. The S-box contents can fit in the on-chip CPU cache for most modern CPUs. The mixing function can be implemented in just a few CPU instructions.

The 4-bit hash function from the previous post was an example of an S-box hash. That example used a carefully selected S-box to ensure that the strict avalanche criteria was satisfied exactly. Theoretically you could construct a mixing function that used an S-box with 2^{32} values to produce a SAC-perfect mixing function but that would use an enormous amount of memory (128 GB). So for this function I took a shortcut and just used 256 values, and then I did some simple shifting and adding to make the hash dependent on the positions of the key bytes.

Test Result

Even though this hash function is very simple, the random nature of the S-box values causes the input bits to be scrambled very effectively. This hash function achieves avalanche for all input bits, and passes the χ^2 test for all upper and lower bit combinations for all three types of keys.

S-Boxes in Cryptography

Like most of the hash functions presented in this site, this hash function is intended for hash-table use, not cryptographic use. But randomized S-boxes can play an important role in cryptography.

Often, S-box values are carefully chosen to avoid specific types of cryptographic attack. For example linear cryptanalysis can be used to attacked cryptographic functions that use linear boolean functions (e.g. you would not want the S-box values to be a linear function of the input values). Differential cryptanalysis can be used to attack S-boxes that produce similar values for similar inputs.

S-boxes generated randomly are clearly not optimized to protect against either of these types of attacks, nor would you expect them to provide optimal protection against any specific type of attack. But on the other hand, randomly-generated S-boxes should be relatively immune to a broad spectrum of attacks. It has already been proven that random S-boxes of sufficient size are almost guaranteed to be resistant to both linear and differential cryptanalysis. If the S-box values are random, there is no systematic relationship between values and therefore no systematic weakness.

Source Code

- C# source code for the hash functions and test code used in this article

References

A somewhat random collection of links to pages I found useful while writing this article.

- Hash function, Wikipedia, 2006
- Avalanche effect, Wikipedia, 2004
- Hash Functions for Hash Table Lookup, Robert J. Jenkins Jr., 1997
- A Resursive Construction Method of S-boxes Satisfying Strict Avalanche Criterion, Kwangjo Kim, Tsutomu Matsumoto, Hideki Imai, 1990
- General Purpose Hash Function Algorithms, Arash Partow
- S-Box Design: A Literature Survey, Terry Ritter, 1997
- Design of Substitution Blocks Satisfying Strict Avalanche Criterion, Martin Stanek and Daniel Olejár

4 years ago

#hash functions *#programming*