The Pluto Scarab

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

Hash Functions

Hash functions are functions that map a bit vector to another bit vector, usually shorter than the original vector and usually of fixed length for a particular function.

There are three primary uses for hash functions:

1. Fast table lookup
2. Message digests
3. Encryption

Fast Table Lookup

Fast table lookup can be implemented using a hash function and a hash table. Elements are found in the hash table by calculating the hash of the element's key and using the hash value as the index into the table. This is clearly faster than other methods, such as examining each element of the table sequentially to find a match.

Message Digests

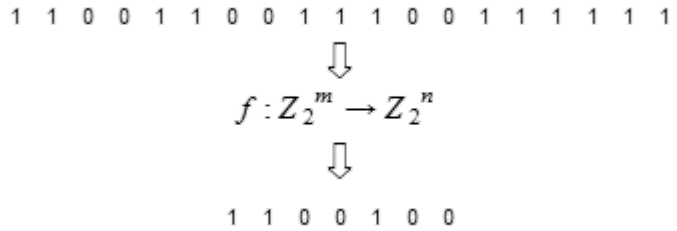
Message digests allow you to compare two large bit vectors and quickly determine if they are equal. Instead of comparing the vectors bit-by-bit, if the hash values of each bit vector are available you can compare the hash values. If the hash values are different, the original vectors must be different. If the hash values are the same then the original vectors are very likely to be the same if the hash function is good.

Message digests can use either cryptographic or non-cryptographic hash functions. If the purpose of the message digest is to determine if the original message has been tampered with, you would need to use a cryptographic hash function. If you just want to quickly tell if it's the same as another file with a different name (assuming the hash values have already been computed), you can use a non-cryptographic hash function.

Encryption

Encryption is the transformation of data into a form unreadable by anyone without a secret decryption key. Hash functions play an important role in encryption because it is their properties that cause the encrypted data to be unreadable and the original data to be unrecoverable from the encrypted data without the decryption key.

Hash functions in this context are sometimes given other names such as mixing functions.



> *A hash function maps a bit vector onto another, usually shorter, bit vector. The result is uniformly distributed, which means that for an input vector chosen at random, each out bit is equally likely to be 0 or 1 and is not correlated with the other bits (unless the size of the range is not a power of 2 in which case the high bits will show correlations).*

> *Typically, m > n and this is why hash functions are called* compression functions *in some applications. Because the function is non-invertible, it means that not all m-bit input vectors can be losslessly compressed by the same function, or even by different functions if you count the bits required to indicate which compression function is to be used.*

Properties of Hash Functions

For a function to be useful as a hash function, it must exhibit the property of *uniform distribution*. Some hash functions also have the *one-way* property. If a hash function is to be used for cryptography or for fast table lookup where the nature of the keys is unknown, the one-way property is a requirement.

Uniform Distribution

All good hash functions share the property of *collision avoidance*, which means that the desired behavior is for unique inputs to provide unique outputs. If the length of the input vector is greater than the length of the output vector it's impossible to

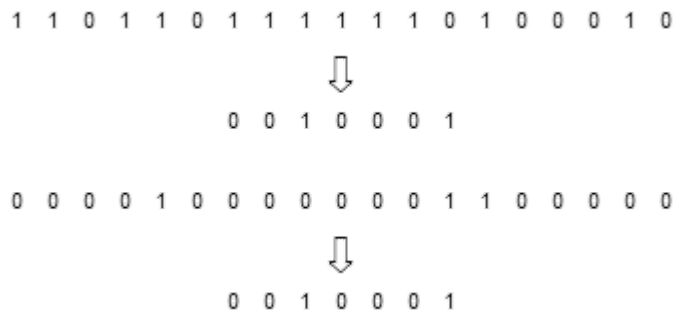
avoid all collisions because the set of input vectors will be of greater order than the set of output vectors. The hash function partitions the input set into subsets of input vectors that all produce the same output.

Since collisions cannot be avoided, the goal is to minimize their likelihood given two arbitrary input vectors. Minimization occurs when the size of the largest partition is minimized, that is, when the output vectors are evenly distributed among the corresponding input vectors. When this happens we say the output vectors are *uniformly distributed.*

One-Way Functions

Another important goal is that "similar" input vectors produce very different output vectors. For the case of table lookup, table keys are usually either human language strings, index numbers, or other data that exhibit non-randomness. For the case of message digest functions, the goal is to make it as difficult as possible to find two different messages that result in the same hash. A hash function with this property is called a *one-way* hash function.

Not all applications require hash functions to be one-way, but all cryptographic applications do. For example, a hash value for fast table lookup where the keys are telephone numbers could simply be the last four digits of the phone number. This hash function is suitable because it's likely to be uniformly distributed, but it's clearly not one-way because it is easy to devise a phone number that has a specific hash value.



> *Collision! Two different input vectors produce the same output vector. For hash tables, this means that the second table insertion will be slower because an alternate location will need to be used. For non-cryptographic message digests, it means that two messages will appear to be the same when they are not. For cryptographic message digests, it means the contents of the message and/or the identity of the sender can be undetectibly altered.*

Simple Hash Function

Figure 1.

```
uint Hash(string s)
{
    uint hash = 0;
    foreach (char c in s)
    {
        hash = hash * 33 + (uint) c;
    }
    return hash;
}
```

Let's look at a simple hash function. Figure 1 shows a hash function intended to be used for a hash table where the keys are character strings.

The hash value is initialized to zero at the start of the function, then for each character in the string the hash value is multiplied by 33 and the Unicode code point value is added to the hash value. No overflow checking is specified, so in C# the value will just wrap around within the range of System.UInt32 if it gets big enough.

The multiplication by 33 is called the *mixing step* and the function f: `uint` → `uint` defined by $f(x) = x * 33$ where * is the C# multiply-without-overflow operator is called the *mixing function*. The constant 33 is chosen arbitrarily.

The addition of the Unicode code point value of the character is called the *combining step* and the function g :`uint` × `uint` → `uint` defined by $g(x, y) = x + y$ where + is the C# add-without-overflow operator is called the *combining function*.

Note that this particular hash function has dubious uniform distribution properties. We'll evaluate that more later.

Hash Function Structure

Figure 2.

```
uint Hash(string s)
{
    // initialization
    int n = s.Length;
    uint hash = 0x79FEF6E8;
    int i = 0;
```

```
    // process each block
    while (i < n - 2)
    {
        // combining step
        hash += (uint) s[i++]
             + ((uint) s[i++]) << 16;

        // mixing step
        hash = Mix(hash);
    }

    // process partial block
    if (i < n)
    {
        // combining step
            hash += (uint) s[i++];

        // mixing step
        hash = Mix(hash);
    }

    // post-processing step
    hash = Mix(Mix(hash));
    return hash;
}

uint Mix(int hash)
{
    hash += hash << 11;
    hash ^= ~hash >> 5;
    hash -= hash << 13;
    return hash;
}
```

The preceding hash function was too simple to illustrate all of the structures commonly found in hash functions. Figure 2 shows a more complete example.

1. First is the initialization step, where the internal state of the hash function is readied.
2. Next, the message is split up into zero or more blocks of equal size, and the internal state of the hash function is updated for each block. In this example, the message is divided into 32-bit blocks. For each block, the combining function is applied to the prior state and the bits from the current block, and then the

mixing function is called. This step is repeated for each full-sized block in the message.
3. Then the final, partial block is processed, if necessary. The combining step is modified to accommodate the incomplete block.
4. Finally, some final processing is done to further randomize the internal state. In this case, the mixing step is applied two more times.

If you examine the hash function from figure 1 again, you'll see that it almost follows this structure. The initialization step is there (the hash value is set to zero), and the message is processed in blocks of 16-bits. There is no post-processing step, or rather there is an "empty" post-processing step.

The main difference is that in figure 1 the mixing step is performed before the combining step for each block. And since there is no post-processing step, it's clear that the bits from the last message block are not processed by the mixing function and therefore do not affect the upper 16 bits of the hash value. If you were to use this hash function for hash table lookup you would want to be sure to use the lower-order bits of the hash value instead of the upper-order bits.

Later we'll examine the mysterious constants in this program, in the initialization step and in the mixing function, to determine whether this is a "good" hash function.

Padding the Last Block

For non-cryptographic use, it's not extremely important how the partial final block is handled. Usually it's OK just to pad the message with 0's. But this is unacceptable for cryptographic use. If all you do is pad with 0's then it is easy to find two messages with the same hash value– just take a known message with a partial final block and append a zero to it. When the additional 0's are added for padding, the two messages look identical. If you just pad with 1's instead, then you can add a 1 bit to an existing message.

For this and other reasons, cryptographic hash functions always pad the original message with a number of bits even if the original message was an even block size, and they always incorporate the message length into the hash calculation. The SHA-1 hash function, for example, always pads the message with a 1 bit, and the last 64 bits of the last block contain the message length, in bits. The bits between the final 1 and the length bits are set to 0. This scheme prevents an attacker from devising two different messages that are identical after the padding operation.

Mixing Functions

The heart of a hash function is its mixing step. The behavior of the mixing function largely determines whether the hash function is collision-resistant. Therefore it shouldn't be surprising that good mixing functions have the same attributes as good hash functions, namely collision resistance and uniform distribution.

The most notable difference between a mixing function and a hash function is that the input and output of a mixing function are the same size. The purpose of the mixing function is to "scramble" or mix the internal state of the hash function. The input to the function is the current internal state and the output of the function becomes the new internal state.

Because the input and output are the same size, mixing functions *can* attain complete collision-resistance, and they should. If the mixing function exhibits collisions, the hash function will exhibit more collisions than are necessary. Collisions are guaranteed not to happen if the mixing function is *reversible*, that is, it's possible to determine the input of the mixing function by examining the output. If the function is not reversible, it implies that there are at least two inputs that result in the same output. By definition, that is a collision.

Reversible Operations

Figure 3

Reversible operations

```
hash ^= constant;
hash *= constant; // if constant is odd
hash += constant;
hash -= constant;
hash ^= hash >> constant;
hash ^= hash << constant;
hash += hash << constant;
hash -= hash << constant;
hash = (hash << 27) | (hash >> 5); // if 32 bits
```

Non-reversible operations

```
hash |= constant;
hash &= constant;
hash <<= constant;
hash >>= constant;
hash *= constant; // if constant is even
hash /= constant;
```

```
hash %= constant;
hash += hash >> constant;
```

Figure 3 lists some reversible and non-reversible mixing operations. It is assumed that the "hash" variable is an unsigned integer and that the arithmetic operations are performed modulo 2^n where n is the size of the hash variable in bits. It is also assumed that 0 < constant < 2^n. A mixing function is guaranteed to be reversible if it consists of steps that are each reversible.

For the purposes of this article it's not important *how* to reverse these operations, but you may be curious and it's definitely not obvious for some of them. The ^= operation is easy—just repeat it. The += and -= operations can be reversed with -= and +=.

The ^= operation combined with << or >> shift is an interesting one. To reverse `hash ^= hash >> 9` for example, you would do `hash ^= (hash >> 9) ^ (hash >> 18) ^ (hash >> 27)`.

The += and -= operations combined with << or >> shifts are equivalent to *= and can be restated that way.

The *= operation is the most difficult to reverse. Look up *modular inverse* and *Euclidean algorithm* in your favorite search engine for details. Because of this difficulty, it's tempting to use this type of mixing step for one-way hash functions. Whether or not this is a good choice largely depends on the performance of the CPU. Some CPU's perform multiplication quickly and some take much longer than addition and subtraction. If multiplication is slow and the constant has a small number of bits set to 1, then equivalent operations using += and left-shifts will work.

Avalanche!

The purpose of the mixing function is to spread the effect of each message bit throughout all the bits of the internal state. Ideally every bit in the hash state is affected by every bit in the message. And we want to do that as quickly as possible simply for the sake of program performance.

A function is said to satisfy the *strict avalanche criterion* if, whenever a single input bit is complemented (toggled between 0 and 1), each of the output bits should change with a probability of one half for an arbitrary selection of the remaining input bits.

The avalanche criterion can be tested for the hash function as a whole, or for just the mixing function. As a part of the overall evaluation of the hash function it makes

sense to examine the avalanche behavior of the mixing function first.

The avalanche behavior can be exactly determined if the size of the hash state is small. If it is larger, such as 160 bits for the SHA-1 hash, it usually has to be estimated. It is possible to construct a mixing function with large bit size for which the avalanche behavior can be exactly calculated, but these mixing functions are usually slow.

Calculating Avalanche

Figure 4

```
hash += hash << 1
```

Input					Output				
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	3	0	0	1	1
2	0	0	1	0	6	0	1	1	0
3	0	0	1	1	9	1	0	0	1
4	0	1	0	0	12	1	1	0	0
5	0	1	0	1	15	1	1	1	1
6	0	1	1	0	2	0	0	1	0
7	0	1	1	1	5	0	1	0	1
8	1	0	0	0	8	1	0	0	0
9	1	0	0	1	11	1	0	1	1
10	1	0	1	0	14	1	1	1	0
11	1	0	1	1	1	0	0	0	1
12	1	1	0	0	4	0	1	0	0
13	1	1	0	1	7	0	1	1	1
14	1	1	1	0	10	1	0	1	0
15	1	1	1	1	13	1	1	0	1

Let's look at a 4-bit mixing function to illustrate the general method for calculating the probabilities that each input but will affect each output bit. Figure 4 shows a mixing operation and lists all possible input values and their corresponding output values. By examining this table we can determine the probability that bit 1, for example, will affect bit 3.

To determine the effect of bit 1, we look at all eight possible combinations of the other three input bits and group the sixteen input values into eight pairs where each element of each pair differs only in bit 1. Input values 4 and 6 form such a pair if we are considering bit 1. These two input values match in bit positions 0, 2, and 3. The other pairs for bit 1 are 0 and 2, 1 and 3, 5 and 7, 8 and 10, etc.

Next we examine the corresponding pairs of output values to see which bits change when bit 1 changes. For input values 4 and 6 we see that the corresponding output values change in bits 1, 2, and 3 but output bit 0 does not change when bit 1 of the input changes. Therefore we say that bit 1 of the input affects bits 1, 2, and 3 of the output but not bit 0. If we look at all eight input pairs for bit 1, we find that bit 0 changes for none of the pairs, bit 1 changes for all of the pairs, bit 2 changes for 50% of the pairs, and bit 3 changes for 75% of the pairs. Therefore this mixing step *by itself* does not come close to satisfying the strict avalanche criterion. We want to see about 50% for each output bit position for each input bit.

Propagation Criterion

The strict avalanche condition is a special case of the *propagation criterion*. It's called the propagation criterion of degree 1, which means that if you toggle any single bit of the input, each output bit will change 50% of the time if you consider all the possible combinations of the n - 1 remaining bits.

A function satisfies the propagation criterion of degree 2 if you toggle any *two* bits in the input, each output bit changes 50% of the time if you consider all the possible combinations of the n - 2 remaining bits.

This method of calculating the avalanche behavior is only feasible because there are only four bits to deal with. For a 32-bit mixing function, we would have to examine sixty-four-thousand million pairs. This can be done with a PC given sufficient time, but it would be out of the question for a 64-bit, 96-bit, or larger mixing function.

For larger mixing functions we can estimate the avalanche behavior by choosing input values at random and toggling each bit on and off to see which output bits are affected. If we keep a total of the results for each random choice, then if we choose a lot of random input values we can get a good estimate of the probabilities.

Actual C# Code

Figure 5

Base class for mixing functions. It provides a data member for holding the internal hash state, a Size property to query the number of bits that the mixing function handles, and a method for actually mixing the state.

```
public abstract class MixingFunction
{
    protected uint state = 0;
```

```
    public abstract int Size { get; }

    public abstract void Mix();
}
```

Bob Jenkins' 32-bit hash for integer hash table keys. Note that the mixing function uses only reversible operations.

```
public class Jenkins32 : MixingFunction
{
    public override int Size
    { get { return 32; } }

    public override void Mix()
    {
        state += (state << 12);
        state ^= (state >> 22);
        state += (state << 4);
        state ^= (state >> 9);
        state += (state << 10);
        state ^= (state >> 2);
        state += (state << 7);
        state ^= (state >> 12);
    }
}
```

Before we write a program to calculate bit-to-bit avalanche behavior, we'll need a "harness" into which we can strap different mixing functions. Figure 5 shows an abstract base class for a mixing function that ensures that each mixing function will have the same interface. I used an abstract class instead of an `interface` because we need to have a member variable for the hash state. This version has a 32-bit member variable called `state`.

Figure 5 also shows a 32-bit mixing function designed by Bob Jenkins. It is intended to be used for a hash table with integer keys. It's tempting to think you can tell which bits affect which other bits just by looking at the code. For example, after the first step `state += (state << 12)`, it's pretty clear that bits 0 through 19 will affect bits 12 through 31 in some way. But when you combine this step with subsequent steps it's possible that some of these effects will cancel out earlier effects. The last step shifts things 12 bits to the right, for example. And if you look at all the steps

except the last and add up all the shifts, counting left shifts as positive and right shifts as negative, you'll see that the total amount of shifting before the last step is zero. It's complicated. We'll just have to measure it to see how it does.

We'll create a function for calculating the *avalanche matrix*. Each element `i,j` of this 32x32 matrix of floating-point values will tell us the probability that bit `i` of the input will affect bit `j` of the output. We want to see a value of 0.5 for every combination.

The best place to put this function is in the `MixingFunction` base class, which means the function automatically becomes available as a member function for any mixing function we create. It essentially becomes a feature of mixing functions in general, instead of having to create this method in some other class and then passing the mixing function as a parameter to the function.

Figure 6

```
public double[,] AvalancheMatrix(int trials,
    int repetitions)
{
    int size = this.Size;
    if (size != 32)
        throw new InvalidOperationException(
            "Hash must be 32 bits.");

    if (trials <= 0 || repetitions <= 0)
        throw new ArgumentOutOfRangeException();

    uint save, inb, outb;
    double dTrials = trials;
    int nBytes = 4;
    byte[] bytes = new byte[nBytes];
    RandomNumberGenerator rng =
        new RNGCryptoServiceProvider();
    int[,] t = new int[size, size];

    while (trials-- > 0)
    {
        rng.GetBytes(bytes);
        save = state = (uint) (bytes[0]
            + 256U * bytes[1]
            + 65536U * bytes[2]
            + 16777216U * bytes[3]);
        for (int r=0; r<repetitions; r++)
```

```
            Mix();
        inb = state;
        for (int i=0; i<size; i++)
        {
            state = save ^ (1U << i);
            for (int r=0; r<repetitions; r++)
                Mix();
            outb = state ^ inb;
            for (int j=0; j<size; j++)
            {
                if ((outb & 1) != 0)
                    t[i, j]++;
                outb >>= 1;
            }
        }
    }

    double[,] result = new double[size, size];
        for (int i=0; i<size; i++)
            for (int j=0; j<size; j++)
                result[i, j] = t[i, j] / dTrials;
    return result;
}
```

Figure 6 shows the code for the `AvalancheMatrix` method of the `MixingFunction` class. It takes two parameters. The first is `trials` which is the number of random input vectors we're going to generate. Evaluating all 2^{31}different combinations of input vectors for each bit combination will take too long, so we'll just pick about a million of them at random. That's the `trials` parameter.

The second parameter is `repetitions`. We'll see in a little while that even if a mixing function doesn't achieve avalanche in one call, it will sometimes do better if the mixing is performed two or more times in a row. The`repetitions` parameter lets us easily test the behavior of multiple applications of the mixing function.

The first thing the function does is to ensure the mixing function is a 32-bit mixing function, since this version of the algorithm doesn't support other sizes. We also check that the function parameters are valid.

Next we declare our variables, initialize some of them (C# initializes everything else to default values automatically), allocate some memory, and initialize the random number generator. I used RNGCryptoServiceProvider from the

System.Security.Cryptography namespace because I know the System.Random class has linear correlations between values x_n, x_{n-34}, and x_{n-55} since it is a two-tap linear feedback PRNG. I'm not 100% sure what RNGCryptoServiceProvider is doing, but the scanty documentation I've seen suggest it uses entropy sources for seed generation and updating, and a cryptographic hash for output whitening. I like that better, although it's really slow.

Inside the main loop we start by getting 32 random bits and setting the internal state. We also save that state for later. Then we call the mixing function the appropriate number of times and save the new state.

Then we toggle each of the 32 input bits one at a time with the statement `state = save ^ (1U << i)`, then we mix that new state and compare it to the original mixed state by doing `outb = state ^ inb` which gives us 0's where the two states match and 1's where they differ. Then we count the 1's and keep a tally.

Finally, after all the trials are finished, we divide each count by the total number of trials to normalize them into the [0, 1] range. Remember that 0.5 is our goal.

The Verdict

Figure 7

Here's how to call the AvalancheMatrix calculator:

```
MixingFunction mf = new Jenkins32();
double[,] am = mf.AvalancheMatrix(1000000, 1);
```

Now we're ready to find out how well Bob Jenkins did with this mixing function with regards to achieving avalanche. Figure 7 shows the two lines of code we need to create an instance of the `Jenkins32` mixing function and call the `AvalanceMatrix` method.

Here are the results, shown as percentages from 0 to 100, rounded to 1%:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
1	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
2	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
3	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
4	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
5	51	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50

```
 6  50 51 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
 7  49 50 51 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50

 8  50 49 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
 9  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
10  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
11  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
12  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
13  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
14  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
15  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50

16  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
17  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
18  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
19  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
20  50 55 50 54 50 50 48 50  51 50 50 50 50 50 50 50  50 50 50 50 51 50
21  53 50 54 50 52 50 50 48  50 50 50 50 50 50 50 50  50 50 50 50 50 50
22  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
23  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50

24  50 50 50 50 50 49 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
25  50 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
26  49 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
27  49 50 50 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
28  50 48 49 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
29  50 50 48 50 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
30  48 50 50 48 50 50 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
31  50 48 50 50 50 51 50 50  50 50 50 50 50 50 50 50  50 50 50 50 50 50
```

Each row represents an input bit and each column represents an output bit. So to determine the probability that bit 6 affects bit 1, look at the row numbered “6” (the 7th row) and the column numbered “1” and you’ll see that the result is 51%.

The technical term for these results is *sweet*. We can be pretty confident that this mixing function will do a good job at collision-avoidance with 32-bit integers for the purposes of hash table lookup.

But if we examine the data *very* closely, we do find that this function doesn’t behave quite the same as a function that satisfies the strict avalanche criterion (SAC) would. This does *not* suggest that this is a bad mixing function. It’s very good. I’m just using it as an example to show how to analyze the avalanche behavior.

The numbers in red show the places where the probability differs by at least 4% from the expected value. For example, the 0th bit affects the 31st bit 54% of the time. To tell whether this is significant or whether it's just a random variation due to the fact that we didn't actually check *every* combination of input bits and instead just did a sample, we need to compare this to a mixing function that satisfies SAC exactly.

Such a function would have exactly a 50% probability in every position, but because we're sampling there would be variations. Since we did one million samples it would behave like flipping a coin one million times, and we would see a binomial distribution. This would look essentially the same as a normal distribution with a mean of 50 and a standard deviation of 0.05. But some positions would be worse than others. With 1024 elements (32 × 32), we can expect that the *worst* positive deviation is going to be about 3.5 standard deviations above the mean, or about 0.175. Because this is well under 0.5, we shouldn't see any non-50's in the table if we round to the nearest 1%.

Figure 8

```
public class Artificial32 : MixingFunction
{
    RandomNumberGenerator rng =
        new RNGCryptoServiceProvider();
    byte[] b = new byte[4];

    public override int Size
    { get { return 32; } }

    public override void Mix()
    {
        rng.GetBytes(b);
        state = BitConverter.ToUInt32(b, 0);
    }
}
```

Since we don't (yet) know of a mixing function that obeys SAC exactly, we can instead create one that actually does a 50% coin flip (simulated). This isn't a real mixing function because the outputs don't depend on the inputs, but it will fool the avalanche calculator into thinking it's real. Figure 8 shows this mixing function.

I won't show the results of this function here because it would be a complete waste of bandwidth, but you can run it yourself and see that it shows 50's in every position, and the raw data shows that the worst positions probably deviate from this by about

0.175 percentage points (it varies each time you run it, of course, so your results may differ).

How Bad Can It Get?

Figure 9

Knuth Multiplicative mixing function

```
public class KnuthMultiplicative : MixingFunction
{
    public override int Size
    { get { return 32; } }

    public override void Mix()
    {
        state *= 2654435761U;
    }
}
```

A sub-section of the avalanche matrix:

	0	1	2	3	4	5	6	7
0	100	0	0	0	100	50	75	63
1	0	100	0	0	0	100	50	75
2	0	0	100	0	0	0	100	50
3	0	0	0	100	0	0	0	100
4	0	0	0	0	100	50	25	13
5	0	0	0	0	0	100	50	25
6	0	0	0	0	0	0	100	50
7	0	0	0	0	0	0	0	100

If you compare the Jenkins32 function to a more primitive multiplicative mixer that people have used in real hash functions, you'll see that we could do much, much worse. Figure 9 shows such a function and a sub-section of the results matrix. It clearly shows that this hash function is bad at propagating high-order bits into low-order bit positions. Whether that's acceptible or not depends on the application. And it's not really as bad as it seems, because the intended use of this function requires taking a prime modulus of the final value, which does scramble the lower bits somewhat.

Can We Do Better?

Is it possible to do better than Jenkins32? The answer is yes. To illustrate this, let's just run Jenkins32 *twice*. The first step already does a great job at mixing, so if we do it twice it should be even better, right? We can easily do this by calling the `AvalancheMatrix` method with a `repetitions` parameter of 2. And we find that, in fact, the results are near-perfect. They are practically indistinguishable from a SAC-perfect mixing function.

Note that this *isn't* true of every mixing function. With `KnuthMultiplicative`, those 0's and 100's never go away because of where they're positioned.

You might wonder if a mixing function that exactly satisfies SAC even exists. Once again the answer is yes. Figure 10 shows a 4-bit mixing function with this property. The truth table of this function is shown so you can confirm that SAC is actually satisfied.

Figure 10

A four-bit mixing function that exactly satisfies the strict avalanche criterion.

```
public void Mix()
{
    uint[] tab =
        {8, 7, 0, 10, 1, 3, 5, 12,
         11, 13, 15, 14, 2, 6, 9, 4};

    state = tab[state];
}
```

The truth table for the above function:

Input		Output	
0	0 0 0 0	8	1 0 0 0
1	0 0 0 1	7	0 1 1 1
2	0 0 1 0	0	0 0 0 0
3	0 0 1 1	10	1 0 1 0
4	0 1 0 0	1	0 0 0 1
5	0 1 0 1	3	0 0 1 1
6	0 1 1 0	5	0 1 0 1
7	0 1 1 1	12	1 1 0 0
8	1 0 0 0	11	1 0 1 0
9	1 0 0 1	13	1 1 0 1
10	1 0 1 0	15	1 1 1 1
11	1 0 1 1	14	1 1 1 0

```
12   1 1 0 0    2    0 0 1 0
13   1 1 0 1    6    0 1 1 0
14   1 1 1 0    9    1 0 0 1
15   1 1 1 1    4    0 1 0 0
```

We know that we can get arbitrarily close just by concatenating a good-enough function two or more times. The challenge is to get closer without adding additional computation time.

Mixing Function Search

Warning! I can think of no practical reason why you would need "perfect" avalanche instead of the excellent avalanche achieved by the `Jenkins32` integer hash. The rest of this page is motivated purely by intellectual curiosity.

To create the mixing function of Figure 10 I did a random search of different permutation tables until I came across one that satisfied the SAC. That works for a 4-bit function, but a lookup table isn't going to be feasible for a 32-bit or larger function. So we can choose to limit our search to functions that are a combination of the reversible operations from the previous page. That's where those "magic numbers" come in. We choose them at random! Some values will be good, some will be bad, but we have a way to measure to see which are the best.

But a completely random search will be too slow. Even if we restrict our search to functions of the same form as`Jenkins32`, that's still 31^8 combinations, or almost a million million. And most of those will be complete rubbish.

Figure 11

Results of searching for coefficient vectors adjacent to known-good vectors. The starting vector (12, 22, 4, 9, 10, 2, 7, 12) corresponds to the eight constants in the Jenkins32 mixing function.

Error	Vector
0.0257	12 22 4 9 10 2 7 12
0.0185	12 22 4 9 10 2 7 **16**
0.0116	12 **18** 4 9 10 2 7 16
0.0112	12 18 4 9 10 2 7 **15**
0.0105	12 18 4 9 10 2 **8** 15
0.0088	12 18 **5** 9 10 2 8 15
0.0085	12 18 5 9 10 **3** 8 15
0.0083	**16** 18 5 9 10 3 8 15

```
0.0080  16 17 5 9 10 3 8 15
0.0052  16 17 5 9 10 3 6 15
0.0052  16 13 5 9 10 3 6 15
0.0048  16 13 5 9 10 3 6 17
0.0047  16 13 5 7 10 3 6 17
0.0044  16 13 5 7 10 2 6 17
0.0039  16 13 5 7 10 2 6 16
0.0039  16 13 4 7 10 2 6 16
0.0038  16 13 4 7 10 2 8 16
0.0024  16 13 4 7 10 5 8 16
```

Instead what we'll do is start with a known-good function and try modifying each of the constants one at a time to see if we can improve on the previous avalanche matrix. In order to compare two functions to determine which is better, we'll need a single number that summarizes the avalance matrix results. I chose to use the "sum of squared errors", which just means I take the difference between each matrix entry and the expected value of 0.5, square it, and then sum all 1024 values together.

I used 100 thousand trials for the matrix calculation. If you use too few trials there is too much "noise" in the results and you can't reliably compare values. The minimum result we can expect to get this way is 0.00256, which is what you would get if the each matrix value were distributed binomially, which is what you'd expect from a function that satisfies SAC. `Jenkins32` gets about 0.0257, about 10 times the expected squared error.

Figure 11 shows the trajectory of the best "search path" that I've been able to find using `Jenkins32` as the starting point. It shows that we're able to reduce the squared error essentially equal to the theoretical minimum, minus some sampling noise.

Evaluating Hash Functions

Figure 12

Abstract base class for hash functions.

```
public delegate byte[] GetRandomKey();

public abstract class HashFunction
{
    // PRNG for generating random keys
    static RandomNumberGenerator rng =
        new RNGCryptoServiceProvider();
```

```
static byte[] b = new byte[4];

// main hashing operating
public abstract
    uint ComputeHash(byte[] data);

// generate a random key length
private static int GetRandomLength()
{
    rng.GetBytes(b);
    uint s = (uint) (b[0] + 256U*b[1]
        + 65536U*b[2] + 16777216U*b[3]);
    double x = 1-s/(uint.MaxValue+1.0);
    return (int) Math.Floor(
        Math.Sqrt(-800.0 * Math.Log(x)));
}

// generate a key with random octets
public static byte[] GetUniformKey()
{
    int length = GetRandomLength() + 2;
    byte[] key = new byte[length];
    rng.GetBytes(key);
    return key;
}

// generate a key with "text" octets
public static byte[] GetTextKey()
{
    int length = GetRandomLength() + 4;
    byte[] key = new byte[length];
    rng.GetBytes(key);
    for (int i=0; i<length; i++)
        key[i] = (byte) (65 +
            (key[i]*key[i]*26)/65026);
    return key;
}

// generate a key with sparse octets
public static byte[] GetSparseKey()
{
    int length = GetRandomLength() + 6;
    byte[] key = new byte[length];
```

```
            rng.GetBytes(key);
            for (int i=0; i<length; i++)
                key[i] = (byte)(1<<(key[i]&7));
            return key;
        }
    }
```

Note that the ComputeHash function prototype isn't sufficient for all real-world hashing scenarios. It requires that all the data is contained in a single octet array, whereas in reality you may need to provide data in chunks or in a format other than octets. But this definition is sufficient for our evaluation scenario where we control the keys.

The base class includes functionality for generating random keys for various tests. This function is independent of the behavior of any particular hash function.

In the preceding section we looked at the avalanche behavior of mixing functions. Now we're ready to look at the behavior of full hash functions including initialization, combining, mixing, and post-processing steps. We'll evaluate the avalanche, distribution, and correlation performance of several existing string hash functions, including a hash with 96-bit internal state by Bob Jenkins, the Fowler/Noll/Vo (FNV) 32-bit hash, a simplistic hash for a baseline comparison, and a cryptographic hash.

First we'll describe the tests and provide results for our baseline hash function (defined below), and in subsequent sections we'll evaluate each hash function individually. Finally, we'll summarize the results for all of our test candidates.

What to Look For in a Hash Function

Before we do some testing, I want to point out that the true test involves testing the hash function *in situ*. Different applications have different hash requirements, and what works well in one situation may not work well in another. What we're going to look at is the generic behavior of hash functions, assuming we know nothing about the keys that are going to be hashed. This should tell us something about the worst-case performance of the functions.

Since we assume no knowledge about the hash keys, we'll use completely random keys. Keys will be arrays of octets. We'll use keys that consist of uniformly distributed octets, keys that simulate text, and keys that are very bit-sparse (a single "1" bit per octet). Why not just use random keys with uniformly distributed octets? Because a

hash function isn't truly needed in that scenario. If you know ahead of time that your keys are composed of completely random bits, you can just use a subsection of the key as your hash value and be guaranteed perfect distribution properties. Adding "text" keys and sparse keys will illustrate a broader range of real-world requirements.

Good *general purpose* hash functions should have these properties:

1. The output should be uniformly distributed. Each pattern of output bits should be equally likely. We'll use a one-tailed χ^2 bucket test for this. See this site by Amar Patel for a very clear explanation of the χ^2 test, or this page for a more formal explanation.
2. Every input bit should affect every output bit about 50% of the time if you consider all possible combinations of the other input bits. This is the avalanche condition. Conversely, every output bit should be affected by every input bit. This is similar to the "no-funnels" condition described by Bob Jenkins. We'll need to modify our avalanche evaluator to handle variable-sized input bit vectors.
3. There should be no correlations between pairs of output bits. If output bits are correlated, you're not getting a full *n* bits of output. We won't test for this because it is easy for all but the most unsuitable hash functions to avoid these correlations.
4. It should be computationally infeasible to identify two keys that produce the same hash value or to identify a key that produces a given hash value. This is a requirement for cryptographic hashes, which are much longer than 32-bits. Since it's easy to find collisions for even an ideal 32-bit hash, this isn't a requirement that we'll use.

Figure 13

A rudimentary hash function.

```
public class SimpleHash : HashFunction
{
    public override
        uint ComputeHash(byte[] data)
    {
        uint hash = 0;
        foreach (byte b in data)
            hash = (hash + b) * 0x50003;
        return hash;
```

```
    }
}
```

Test Harness

We first need to create a test harness for hash functions just like we did for mixing functions. Figure 12 shows the `HashFunction` abstract base class which we'll use for all the hashes we'll investigate. We're going to assume a 32-bit hash output for all the hash functions.

Different types of keys require different lengths. In order for the χ^2 tests with 2^{16} buckets to be valid, all of the keys need to contain at least 16 bits of information. The sparse keys only contain three bits of information in each octet, so we need at least six octets to ensure 16 bits of information. The "text" keys contain 4.34 bits per octet on average, so we need at least four octets for that one. We only need two octets for the uniformly distributed octets.

Figure 13 lists a rudimentary hash function which we'll use to work out our testing methodology. The constant `0x50003` was selected for the ability for the "3" to mix key bits into the lower half of the hash value and the "5" to mix key bits into the upper half. We avoid using the same number for both halves to minimize correlations between each set of bits.

Figure 14

χ^2 test results for the SimpleHash function. Tests were done by bucketizing values obtained from the upper and lower ends of the 32-bit hash values, for bucket counts 2^1 through 2^{16}.

	Uniform keys		Text keys		Sparse keys	
Bits	Lower	Upper	Lower	Upper	Lower	Upper
1	0.480	1.000	0.396	0.203	0.322	0.396
2	0.585	0.678	0.835	0.932	0.873	0.864
3	0.766	0.694	0.377	0.862	0.139	0.496
4	0.982	0.527	0.607	0.463	0.225	0.886
5	0.847	0.469	0.683	0.921	0.391	0.371
6	0.943	0.624	0.773	0.679	0.117	0.860
7	0.415	0.854	0.161	0.184	0.318	0.883
8	0.682	0.687	0.886	0.335	0.248	0.919
9	0.647	0.906	0.302	0.640	0.475	0.904
10	0.565	0.740	0.779	0.362	0.421	0.481

11	0.693	0.875	0.268	0.755	0.814	0.111
12	0.499	0.476	0.443	0.846	0.903	0.794
13	0.140	0.852	0.234	0.856	0.640	0.665
14	0.390	0.436	~~0.000~~	0.624	0.129	0.756
15	~~0.000~~	~~0.003~~	~~0.000~~	0.079	0.113	0.276
16	~~0.000~~	~~0.000~~	~~0.000~~	*0.031*	~~0.000~~	0.108

~~Struck~~ *values show failures at the 1% significance level, and* oblique *values show failures at the 5% level. This hash has severe weaknesses for hash tables with* 2^{14} *or more buckets, although this function did acceptably well in the upper bits for sparse keys all the way up to* 2^{16} *buckets. Therefore this is a fast, simple, and easy-to-remember function suitable for small hash tables up to 8192 buckets for a wide variety of keys.*

Testing Uniform Distribution

Hash functions used with hash tables of size 2^n would typically use only the lower-order bits of a 32-bit hash function output. But to be thorough in our tests we'll look at the distribution properties of both the low-order and high-order bits, up to 16 bits long.

For each test we'll call the hash function repeatedly with random keys, and look to see which bucket we're assigned. We'll count the hits to each bucket, and compare the result to the expected result for a truly uniform distribution. We'll use the one-tailed χ^2 test for this. (We don't care if the distribution is "too uniform" since we're throwing random keys at each function. If we were testing a random number generator we might want a two-tailed test.) The χ^2degrees of freedom for each measurement will be 2^m-1, where m is the bit-size of the sub-range that we're testing.

The result of each χ^2 test will be a p-value, which is a number from 0.0 to 1.0, and we would expect these to be uniformly distributed in that range if the hash output is uniform. The p-value indicates the probability that a truly uniform distribution would be worse that the observed distribution by chance alone. For example a p-value of 0.01 indicates that a truly uniform distribution would only be worse than the observed distribution only 1% of the time, which is bad.

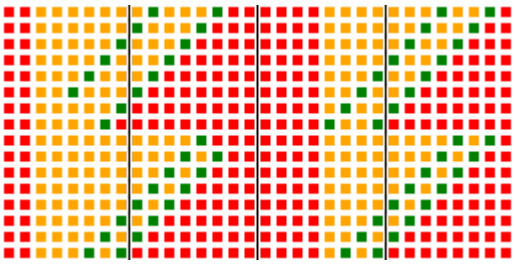
Even if we define p=0.01 as the cut-off for a "bad" distribution, a truly uniform distribution would have a p-value worse than this 1% of the time, by definition. So we need to look at each failure in context to see if it's likely just random chance or whether it forms part of a pattern. In general I will re-run a test several times to determine which results are significant and which are not, and I only present the

most representative results. I wouldn't do this in a scientific context, but this isn't that.

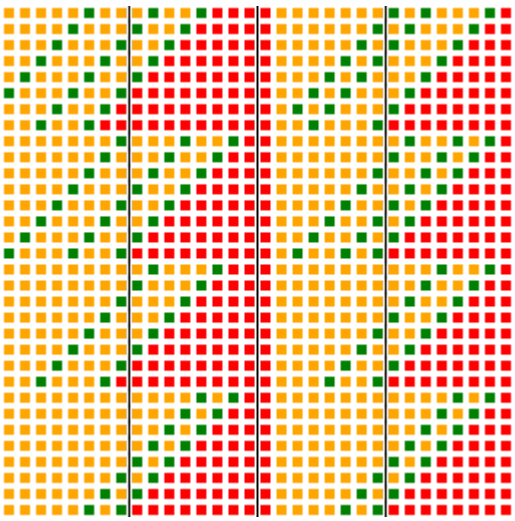
Figure 15

Avalanche behavior for SimpleHash function. Each row represents one input bit and the 32 squares in each row indicate the influence on each hash bit by that input bit. Green indicates acceptible avalanche behavior, orange is unacceptible, and red means that there was no mixing at all (either 0% or 100% influence). The top row corresponds to the LSB of the 0th-index key octet and the bottom row corresponds to the MSB of the last key octet.

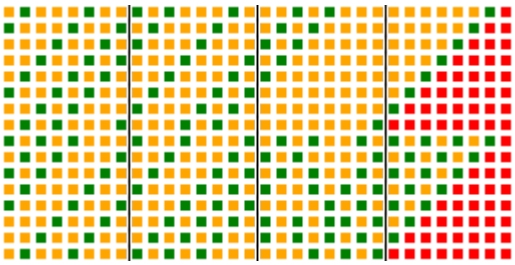
For two-octet keys:



For four-octet keys:



For 256-octet keys:



This hash has absolutely horrible avalanche behavior. Some output bits are not mixed at all. The fact that the LSB doesn't get mixed at all indicates that this hash

would not provide a uniform distribution if the LSBs of the keys were not already uniform.

This overall behavior is typical of multiplicative hashes unless extra steps are taken during post-processing to improve the mixing behavior.

For each test, we'll generate enough random keys to fill each bucket with an average of 100 keys. The key length will be a random variable `k+floor(sqrt(-800*ln(x)))` where x is a uniformly distributed random variable on (0, 1] and k is the minimum key length required to produce 16 bits of data in the key, which varies by the type of key. The function `floor` indicates rounding down, `sqrt`indicates square-root, and `ln` is the natural logarithm. We'll need a minimum key length of at least k because we'll be doing χ^2 tests with up to 2^{16} buckets, and the presence of some too-short keys would force non-uniform output in those cases and cause those tests to fail regardless of the hash function used.

Figure 14 shows the results for the `SimpleHash` function. It shows weakness in both the upper and lower bits when use 14 or more of the bits. The upper bits are a little stronger–this half of the hash gets some "overflow" from multiplications that carry out of the lower half.

Avalanche Testing

If a hash function passes the uniform distribution test, why do any of the other tests matter? While it's true that the primary use for most hashes is for hash table lookup, that's not the only use. Small hashes can be used for non-cryptographic document fingerprinting, for example. Also, other tests can identify classes of keys for which the uniform distribution might fail even if the tests with random keys succeed.

Unlike the previous section which pursued the Strict Avalanche Condition to paranoid levels, we'll consider "good enough" avalanche to be when each input bit affects each output bit between 1/3rd and 2/3rds of the time. The mixing function will be applied repeatedly in most cases, which will only improve the avalanche behavior if it's good enough to start with. If single-round avalanche is borderline, we might consider calling the mixing function one or two more times as part of the post-processing step. This test will tell is which functions would benefit from that.

The definition of avalanche requires that we only evaluate keys with uniformly distributed input octets, so this test won't use GetTextKeys or GetSparseKeys. Also, it's challenging to test every input bit when keys get very long, so we're going to restrict keys to specific lengths. We'll examine two-octet keys which will allow us to

exactly calculate the avalanche matrix, as well as four-octet keys for which we'll need to do sampling just as before. Finally we'll use 256-octet keys to see the avalanche behavior when multiple rounds of the mixing function are used, and we'll look at input bits in the first and last octets.

The 16x32 and 32x32 avalanche matrices are cumbersome to present on a web page, so we'll use a graphical summary of the results instead. We'll use green squares for bit combinations that achieved avalanche, orange squares that did not, and red squares where the input bits had either no affect or a direct affect on the output (0% or 100% affect).

Figure 15 shows the results of the avalanche tests for the `SimpleHash` function. Next we'll look at some other well-known hash functions.

Fowler/Noll/Vo Hash

Figure fnv1

Fowler/Noll/Vo (FNV) 32-bit hash function.

```
public class FNV : HashFunction
{
    int shift;
    uint mask;

    // hash without xor-folding
    public FNV()
    {
        shift = 0;
        mask = 0xFFFFFFFF;
    }

    // hash with xor-folding
    public FNV(int bits)
    {
        shift = 32 - bits;
        mask = (1U << shift) - 1U;
    }

    public override uint ComputeHash(byte[] data)
    {
        uint hash = 2166136261;
        foreach (byte b in data)
```

```
            hash = (hash * 16777619) ^ b;
        if (shift == 0)
            return hash;
        return (hash ^ (hash >> shift)) & mask;
    }
}
```

This hash function is described here. It is a simple multiplicative hash with the addition of a post-processing step called *xor-folding* to remove some linearity in the lower bits. The FNV authors recommend using xor-folding if the desired hash size is not a power of 2, but they actually mean to use xor-folding when the hash size is not a power of 2 or is less than 32.

There are different initialization and multiplication constants for use with 32-bit, 64-bit, 128-bit hashes, etc., but we'll only examine the 32-bit version of FNV here. In our uniform distribution test we'll use xor-folding for the lower bits since that is the intended use of this hash, but we'll examine the upper bits as-is. Since our avalanche test is a full 32-bit test, we can't use xor-folding there. This will allow us to identify classes of keys for which using xor-folding is a necessity.

The primary appeals of this hash are its simplicity and its speed, but its speed is dependent on whether or not the CPU architecture supports fast integer multiplication. The Jenkins shift-add-xor hashes are faster on CPU architectures without fast multiplication.

Figure fnv1 shows the listing for the FNV 32-bit hash.

Uniform Distribution Test

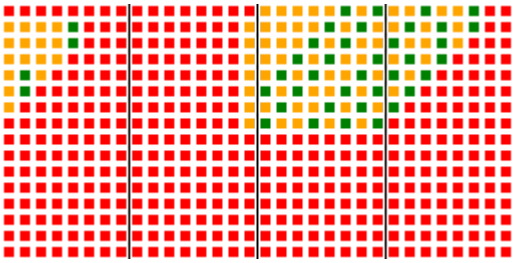
We examine the distribution of numbers derived from lower and upper bits of the hash output in sizes of 1 through 16 bits.

Figure fnv3 shows the results of this test for the FNV hash. This test indicates that the FNV 32-bit hash with xor-folding produces uniformly distributed values for hash tables that are a power of two, up to at least 2^{14}, when the key octets are uniformly distributed, distributed similar to alphabetic text, or sparsely distributed.

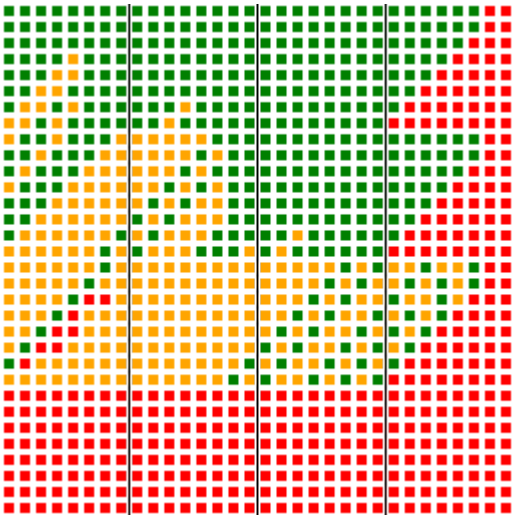
Figure fnv2

Avalanche behavior of the FNV 32-bit hash.

For two-octet keys:



For four-octet keys:



For 256-octet keys:

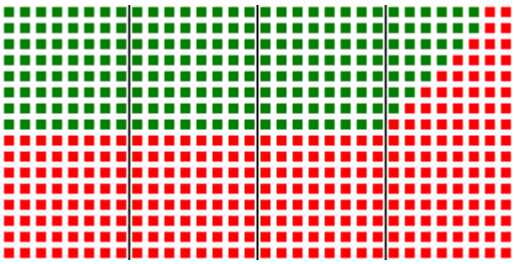


Figure fnv3

χ^2 test results for FNV 32-bit hash, with xor-folding for the lower-bit tests.

Uniform keys		Text keys		Sparse keys	
Bits	Lower	Upper	Lower	Upper	Lower	Upper
1	0.777	0.888	0.888	0.888	0.480	0.480
2	0.967	0.326	0.407	0.197	0.513	0.720
3	0.109	0.390	0.498	0.103	0.573	0.016
4	0.548	0.416	0.649	0.210	0.143	0.469
5	0.360	0.606	0.931	0.992	0.665	0.201
6	0.328	0.753	0.584	0.416	0.882	0.361
7	0.436	0.560	0.995	0.302	0.981	0.124
8	0.297	0.729	0.856	0.730	0.472	0.113
9	0.222	0.349	0.629	0.951	0.701	0.769
10	0.731	0.208	0.066	0.646	0.875	0.551

11	0.813	0.356	0.678	0.820	0.519	0.556
12	0.076	0.229	0.521	0.068	0.091	0.474
13	0.780	0.565	0.719	0.090	0.117	0.132
14	0.225	0.813	0.269	0.251	0.855	0.568
15	0.583	0.005	0.699	0.370	0.571	0.218
16	0.004	0.000	0.117	0.012	0.024	0.211

The upper bits are not uniformly distributed if you use more than 14 or 15 bits. Because of this, I don't recommend using this hash with hash tables larger than 2^{14} buckets. The results are acceptible up to 2^{16} bits for text keys, so you may be able to use it for that purpose if you carefully test the performance in your particular application.

Avalanche Test

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

Figure fnv2 shows the results of this test for the FNV hash. This test indicates that the FNV hash has poor avalanche behavior, as do all simple multiplicative hashes. This means that there will be some classes of keys for which the hash function does not produce uniform output, even for small bucket counts where the χ^2 tests succeeded above.

Of particular concern are the two low-order bits. These are always just a simple linear function of the two low-order bits of the keys octets. For example, in the full 32-bit hash value, the low-order bit will always just be a simple XOR of the LSBs of the key octets and the LSB from the intialization constant. Also of concern is that fact that the upper bits of the key octets do not have any influence on the low-order bits of the hash output (without xor-folding). The MSB of each key octet does not diffuse at all into the entire lower 8 bits of the hash value. This indicates that you *must* follow the xor-folding recommendations for all classes of keys.

Also, the mixing step of this hash is never applied to the last octet of the key. This shows clearly in the avalanche results. The authors offer an alternative form of the hash where the combining step is done *before* the mixing step, and I recommend that you adopt this alternative if you use FNV. There is no reason to do otherwise, and I'm surprised that the authors do not recommend this by default.

Conclusion

Figure fnv4

Modified FNV with good avalanche behavior and uniform distribution with larger hash sizes.

```
public class ModifiedFNV : HashFunction
{
    public override uint ComputeHash(byte[] data)
    {
        const uint p = 16777619;
        uint hash = 2166136261;
        foreach (byte b in data)
            hash = (hash ^ b) * p;
        hash += hash << 13;
        hash ^= hash >> 7;
        hash += hash << 3;
        hash ^= hash >> 17;
        hash += hash << 5;
        return hash;
    }
}
```

I don't recommend using 32-bit FNV as a general-purpose hash as-is. It can produce uniform output, but its suitability needs to be tested for each class of keys for which you intend to use it. It is likely to produce non-uniform output if you have more than 2^{14} or 2^{15} hash buckets. If the CPU architecture does not have fast integer multiplication, use a shift-add-xor hash instead.

If you want to use an FNV-style hash function, I recommend using the modified version listed in Figure fnv4. This version passes all the uniform distribution tests above and it achieves avalanche for every tested combination of input and output bits (green squares everywhere). No xor-folding step is required.

The only difference between this version and the original version is that the mixing steps occurs *after* the combining step in each round, and it adds a post-processing step to mix the bits even further. These two changes completely correct the avalanche behavior of the function. As a result, this version of FNV passes all of the χ^2 tests above, all the way up to 2^{16} buckets. I haven't tested larger sizes but I suspect it would be OK there as well.

Continue to part 2

4 years ago

#hash functions *#programming*

Home

Ask me anything

Ashley theme by Jxnblk