



4 Nov 2016 | 9 min. (1767 words)

# Designing a good non-cryptographic hash function

So, I've been needing a hash function for various purposes, lately. None of the existing hash functions I could find were sufficient for my needs, so I went and designed my own. These are my notes on the design of hash functions.

## What is a hash function *really*?

Hash functions are functions which maps a infinite domain to a finite codomain. In the domain,  $a, b$  are said to collide if  $h(a) = h(b)$ .

The ideal hash functions has the property that the distribution of image of a subset of the domain is statistically independent of the probability of said subset occurring. That is, collisions are not likely to occur even within non-uniform distributed sets.

Consider you have an english dictionary. Clearly, `hello` is more likely to be a word than `ctyhbnkmaasrt`, but the hash function must not be affected by this statistical redundancy.

In a sense, you can think of the ideal hash function as being a function where the output is uniformly distributed (e.g., chosen by a sequence of coinflips) over the codomain no matter what the distribution of the input is.

With a good hash function, it should be hard to distinguish between a truly random sequence and the hashes of some permutation of the domain.

Hash function ought to be as chaotic as possible. A small change in the input should appear in the output as if it was a big change. This is called the hash function butterfly effect.

# Non-cryptographic and cryptographic

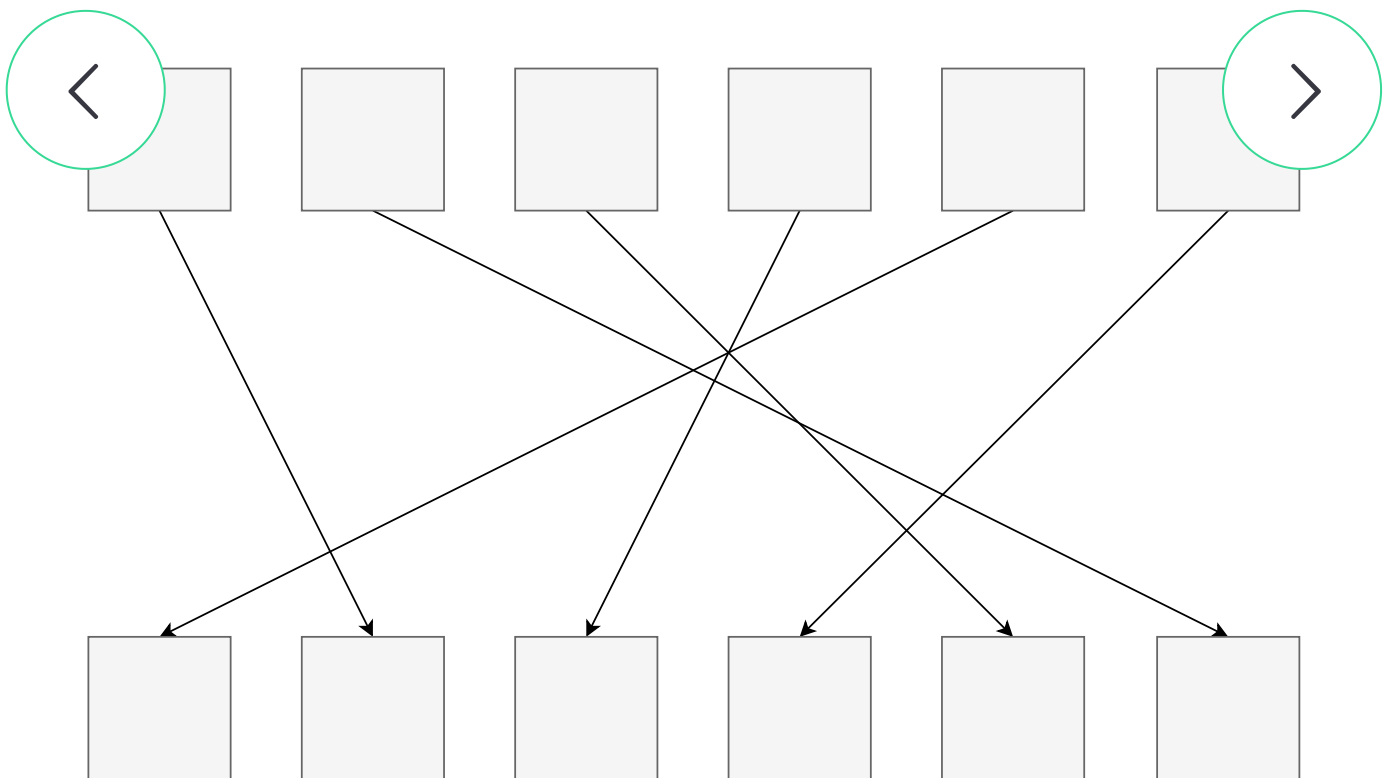
One must make the distinction between cryptographic and non-cryptographic hash functions. In a cryptographic hash function, it must be infeasible to:

1. Generate the input from its hash output.
2. Generate two inputs with the same output.

Non-cryptographic hash functions can be thought of as approximations of these invariants. The reason for the use of non-cryptographic hash function is that they're significantly faster than cryptographic hash functions.

## Diffusions and bijection

The basic building block of good hash functions are diffusions. Diffusions can be thought of as bijective (i.e. every input has one and only one output, and vice versa) hash functions, namely that input and output are uncorrelated:



This diffusion function has a relatively small domain, for illustrational purpose.

## Building a good diffusion

Diffusions are often build by smaller, bijective components, which we will call "subdiffusions".

## Types of subdiffusions

One must distinguish between the different kinds of subdiffusions.

The first class to consider is the **bitwise subdiffusions**. These are quite weak when they stand alone, and thus must be combined with other types of subdiffusions. Bitwise subdiffusions might flip certain bits and/or reorganize them:

$$d(x) = \sigma(x) \oplus m$$

(we use  $\sigma$  to denote permutation of bits)

The second class is **dependent bitwise subdiffusions**. These are diffusions which permutes the bits and XOR them with the original value:



$$d(x) = \sigma(x) \oplus x$$



(exercise to reader: prove that the above subdivision is revertible)

Another similar often used subdiffusion in the same class is the XOR-shift:

$$d(x) = (x \ll m) \oplus x$$

(note that  $m$  can be negative, in which case the bitshift becomes a right bitshift)

The next subdiffusion are of massive importance. It's the class of **linear subdiffusions** similar to the LCG random number generator:

$$d(x) \equiv ax + c \pmod{m}, \quad \gcd(x, m) = 1$$

(gcd means "greatest common divisor", this constraint is necessary in order to have  $a$  have an inverse in the ring)

The next are particularly interesting, it's the **arithmetic subdiffusions**:

$$d(x) = x \oplus (x + c)$$

## Combining subdiffusions

Subdiffusions themselves are quite poor quality. Combining them is what creates a good diffusion function.

Indeed if you combine enough different subdiffusions, you get a good diffusion function, but there is a catch: The more subdiffusions you combine the slower it is to compute.

As such, it is important to find a small, diverse set of subdiffusions which has a good quality.

## Zero-sensitivity

If your diffusion isn't zero-sensitive (i.e.,  $f(0) = \{0, 1\}$ ), you should ~~panic~~ come up with something better. In particular, make sure your diffusion contains at least one sensitive subdiffusion as component.

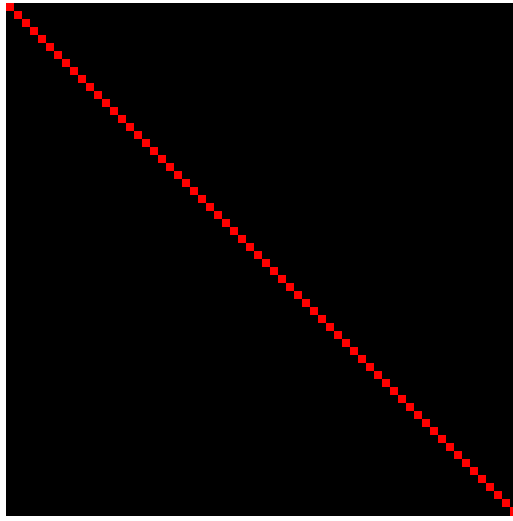


## Avalanche diagrams

Avalanche diagrams are the best and quickest way to find out if your diffusion function has a good quality.

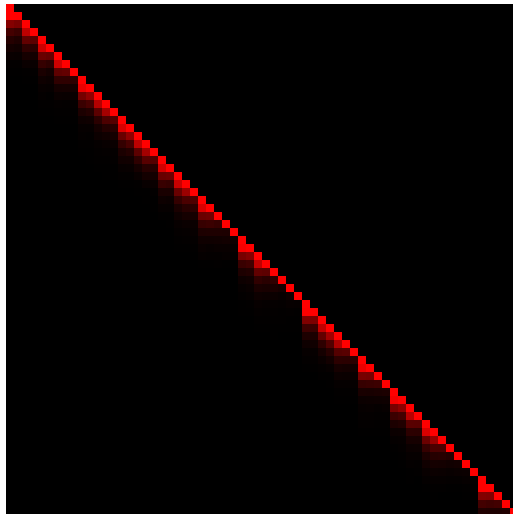
Essentially, you draw a grid such that the  $(x, y)$  cell's color represents the probability that flipping  $x$ 'th bit of the input will result of  $y$ 'th bit being flipped in the output. If  $(x, y)$  is very red, the probability that  $d(a')$ , where  $a'$  is  $a$  with the  $x$ 'th bit flipped, has the  $y$ 'th bit flipped is very high.

Here's an example of the identity function,  $f(x) = x$ :

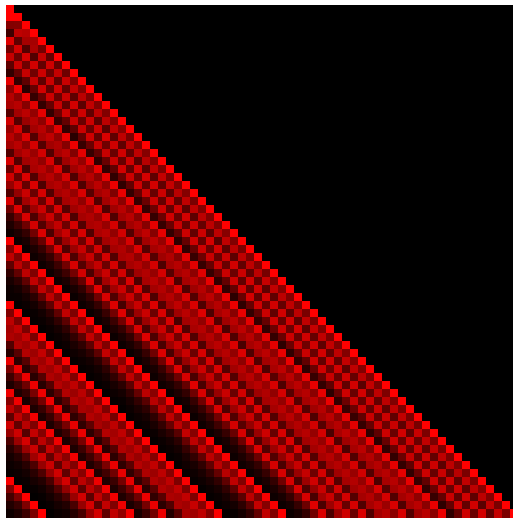


So why is it a straight line?

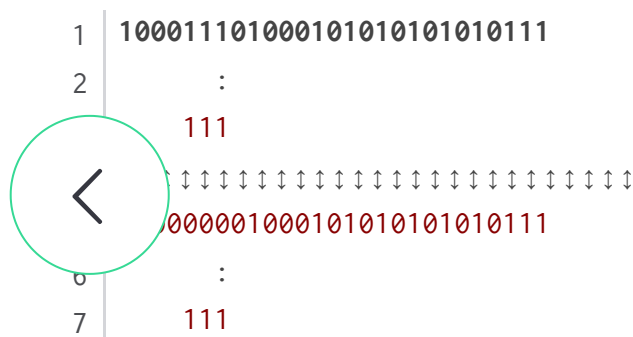
Well, if you flip the  $n$ 'th bit in the input, the only bit flipped in the output is the  $n$ 'th bit. That's kind of boring, let's try adding a number:



Meh, this is kind of obvious. Let's try multiplying by a prime:



Now, this is quite interesting actually. We call all the black area "blind spots", and you can see here that anything with  $x > y$  is a blind spot. Why is that? Well, if I flip a high bit, it won't affect the lower bits because you can see multiplication as a form of overlay:



Flipping a single bit will only change the integer forward, never backwards, hence it forms this blind spot. So how can we fix this (we don't want this bias)?

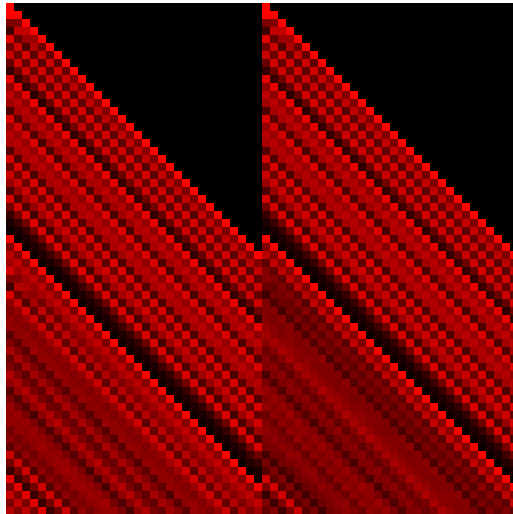
## Designing a diffusion function -- by example

If we throw in (after prime multiplication) a dependent bitwise-shift subdiffusions, we have

$$\begin{aligned}
 x &\leftarrow x + 1 \\
 x &\leftarrow x \oplus (x \ggg z) \\
 x &\leftarrow px \\
 x &\leftarrow x \oplus (x \lll z)
 \end{aligned}$$

(note that we have the  $+1$  in order to make it zero-sensitive)

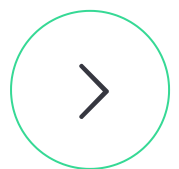
This generates following avalanche diagram



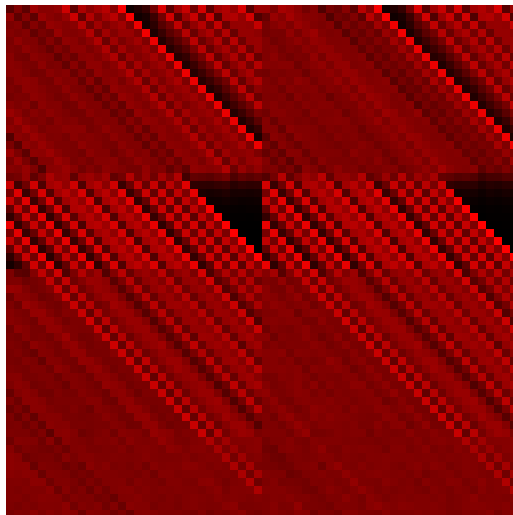
What can cause these? Clearly there is some form of bias. Turns out that this bias mostly originates in the lack of hybrid arithmetic/bitwise sub. Without such hybrid, the behavior tends to be relatively local and not interfering well with each other.



$$x \leftarrow x + \text{ROL}_k(x)$$



int, it looks something like

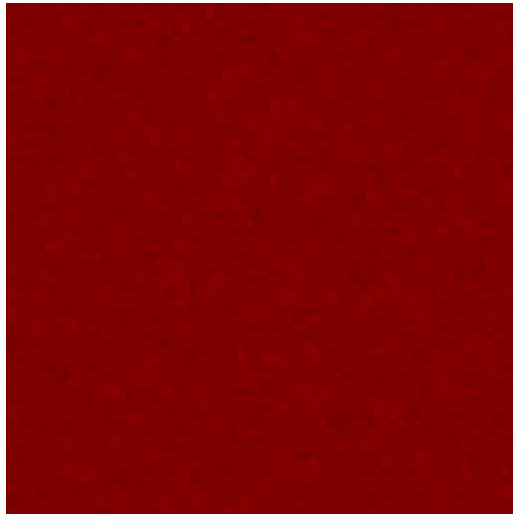


That's good, but we're not quite there yet...

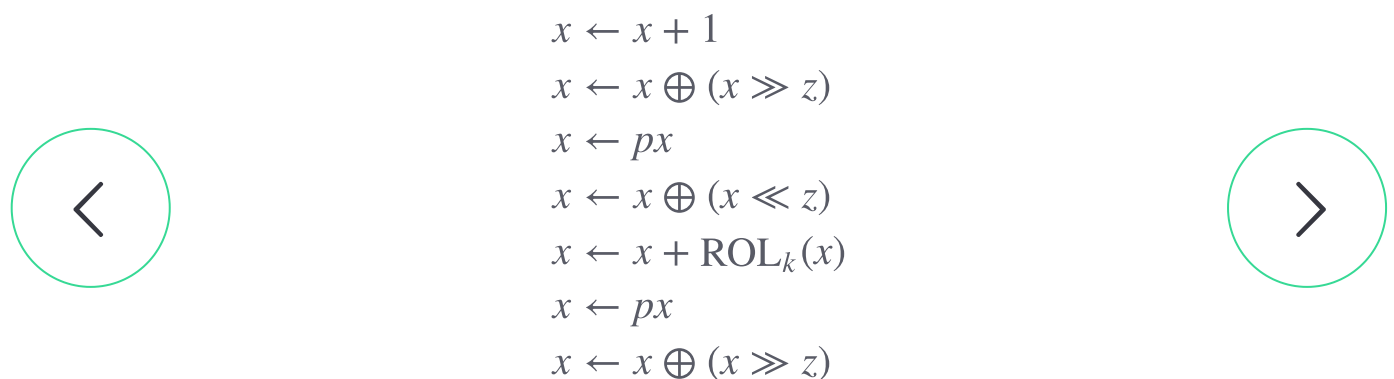
Let's throw in the following bijection:

$$x \leftarrow px \oplus (px \gg z)$$

And voilà, we now have a perfect bit independence:

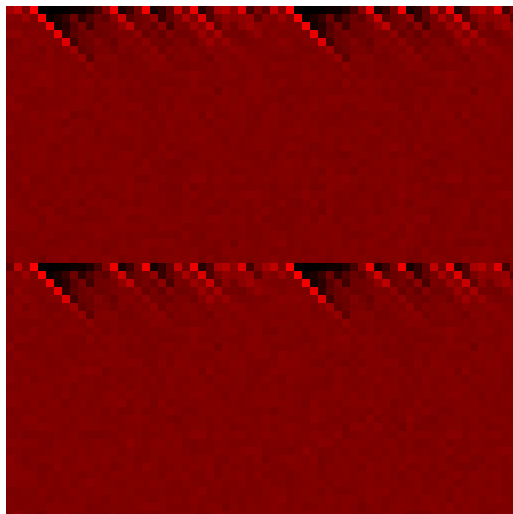


So our finalized version of an example diffusion is



That seems like a pretty lengthy chunk of operations. We will try to boil it down to few operations while preserving the quality of this diffusion.

The most obvious think to remove is the rotation line. But it hurts quality:



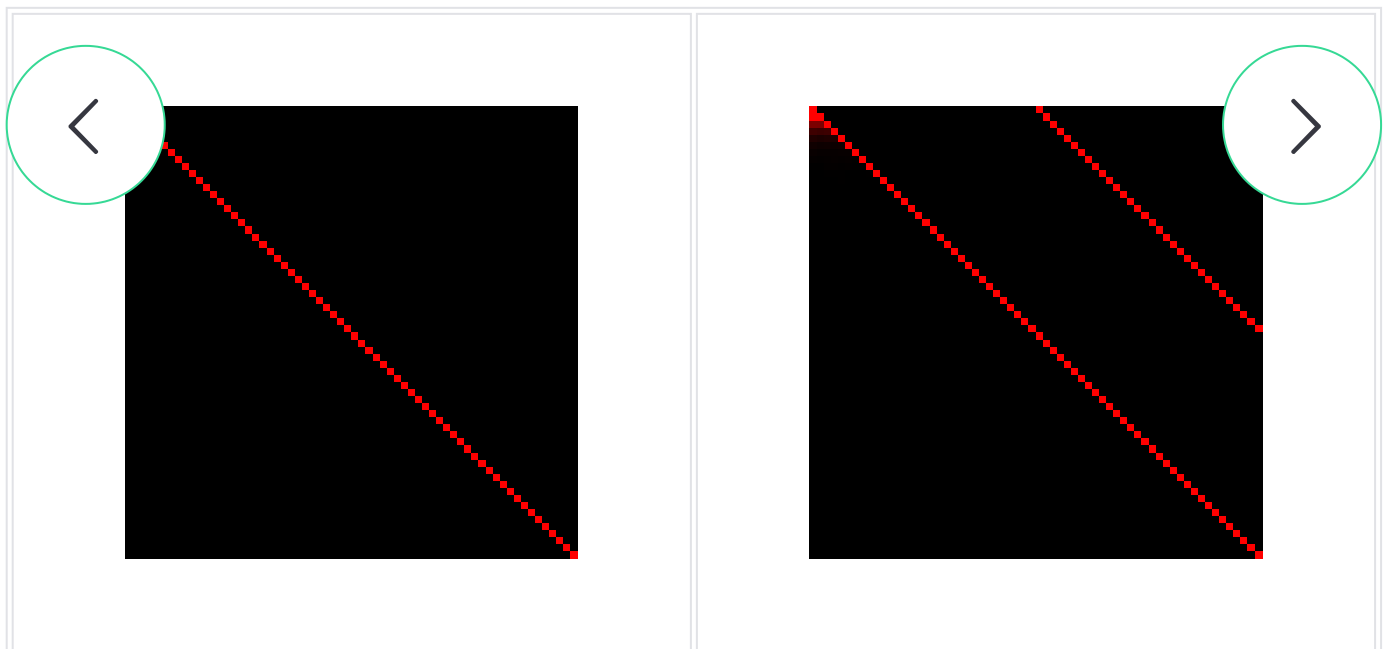


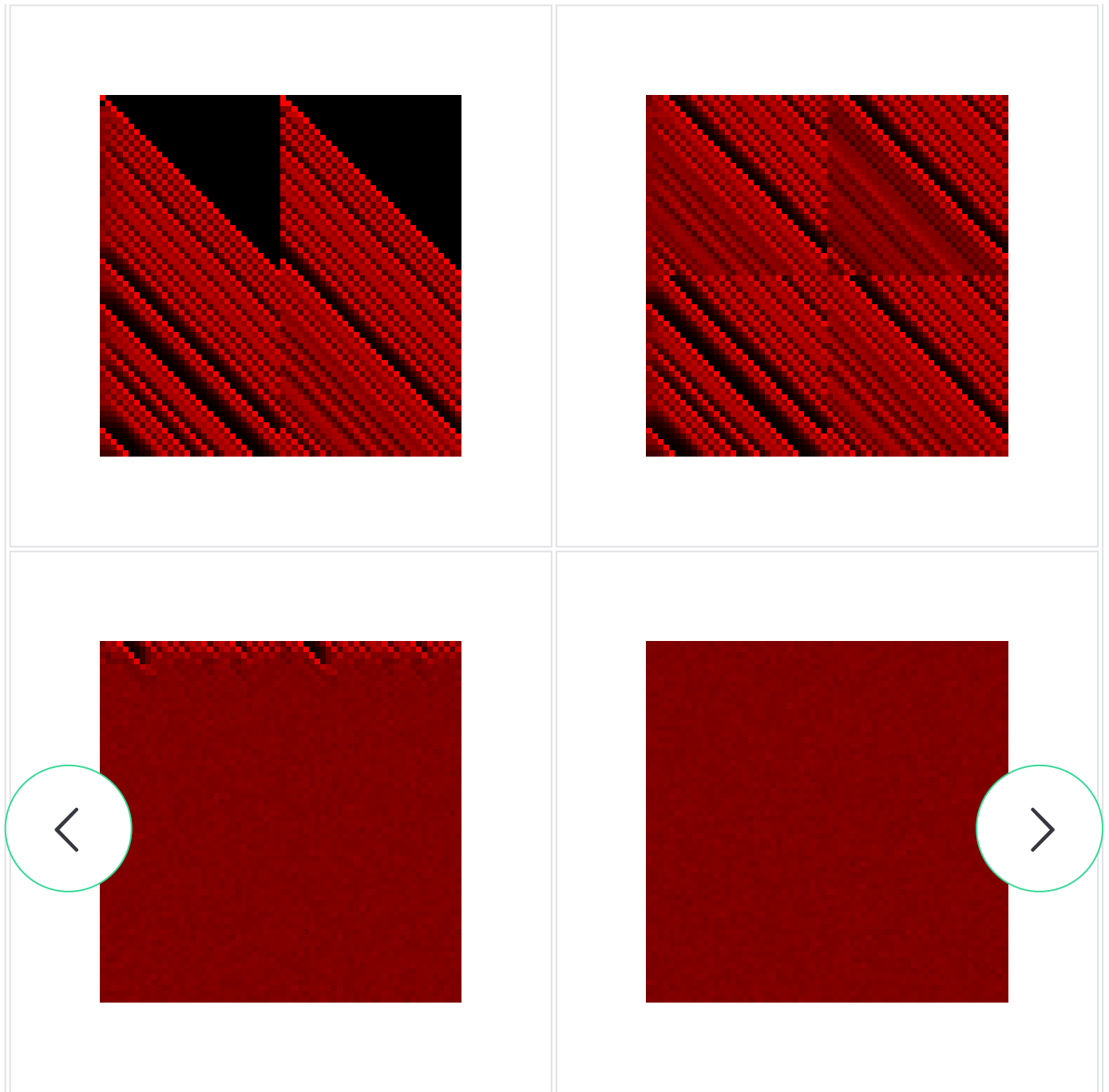
Where do these blind spot comes from? The answer is pretty simple: shifting left moves the entropy upwards, hence the multiplication will never really flip the lower bits. For example, if we flip the sixth bit, and trace it down the operations, you will how it never flips in the other end.

So what do we do? Instead of shifting left, we need to shift right, since multiplication only affects upwards:

```
x ← x + 1  
x ← x ⊕ (x ≫ z)  
x ← px  
x ← x ⊕ (x ≫ z)  
x ← px  
x ← x ⊕ (x ≫ z)
```

And we're back again. This time with two less instructions.





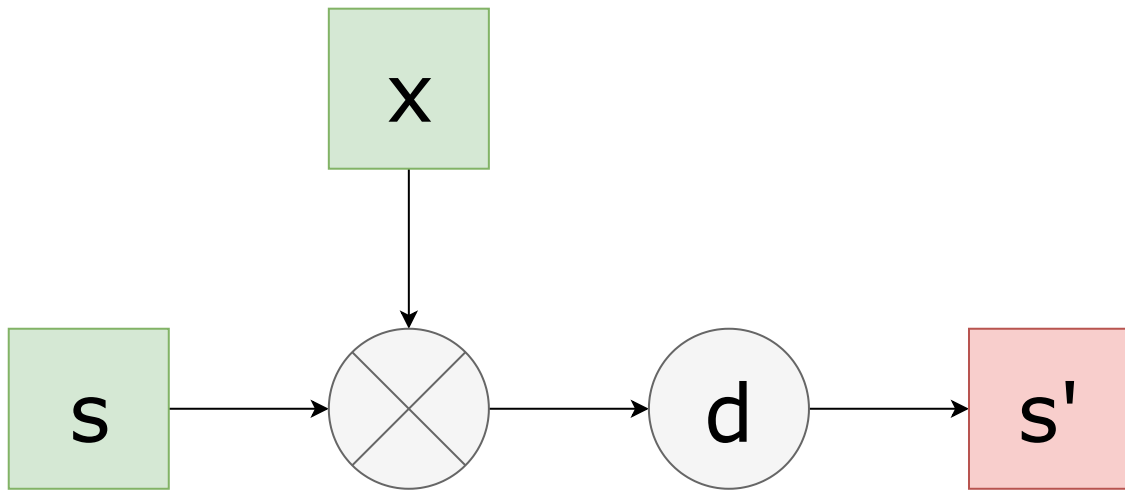
## Combining diffusions

Diffusions maps a finite state space to a finite state space, as such they're not alone sufficient as arbitrary-length hash function, so we need a way to combine diffusions.

In particular, we can eat  $N$  bytes of the input at once and modify the state based on that:

$$s' = d(f(s', x))$$

Or in graphic form,



$f(s', x)$  is what we call our combinator function. It serves for combining the old state and the new input block ( $x$ ).  $d(a)$  is just our diffusion function.

It doesn't matter if the combinator function is commutative or not, but it is crucial that it is not biased, i.e. if  $a, b$  are uniformly distributed variables,  $f(a, b)$  is too. Ideally, there should exist a bijection,  $g(f(a, b), b) = a$ , which implies that it is not biased.

An example of such combination function is simple addition.



$$f(a, b) = a + b$$



Another is

$$f(a, b) = a \oplus b$$

I'm partial towards saying that these are the only sane choices for combinator functions, and you must pick between them based on the characteristics of your diffusion function:

1. If your diffusion function is primarily based on arithmetics, you should use the XOR combinator function.
2. If your diffusion function is primarily based on bitwise operations, you should use the additive combinator function.

The reason for this is that you want to have the operations to be as diverse as possible, to create complex, seemingly random behavior.

# SIMD, SIMD, SIMD

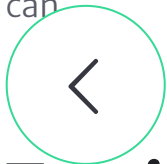
If you want good performance, you shouldn't read only one byte at a time. By reading multiple bytes at a time, your algorithm becomes several times faster.

This however introduces the need for some finalization, if the total number of written bytes doesn't divide the number of bytes read in a round. One possibility is to pad it with zeros and write the total length in the end, however this turns out to be somewhat slow for small inputs.

A better option is to write in the number of padding bytes into the last byte.

## Instruction level parallelism

Fetching multiple blocks and sequentially (without dependency until last) running a round is something I've found to work well. This has to do with the so-called instruction pipeline in which modern processors run instructions in parallel when they can



## Testing the hash function

Multiple test suits for testing the quality and performance of your hash function. [Smhasher](#) is one of these.

## Conclusion

Many relatively simple components can be combined into a strong and robust non-cryptographic hash function for use in hash tables and in checksumming. Deriving such a function is really just coming up with the components to construct this hash function.

Breaking the problem down into small subproblems significantly simplifies analysis and guarantees.

The key to a good hash function is to try-and-miss. Testing and throwing out candidates is the only way you can really find out if you hash function works in practice.

Have fun hacking!

Follow me on [Twitter](#) or [Github](#).

hash   algorithms   notes

ALSO ON TICKI.GITHUB.IO

Skip Lists: Done Right · Ticki's blog

4 years ago • 13 comments

Skip lists are a wonderful data structure, but it is hard to get it right. This blog ...

SeaHash: Explained · Ticki's blog

4 years ago • 2 comments


I explain how SeaHash works.


You Are (Probably) Doing Login Wrong · Ticki's blog


3 years ago • 1 comment


This blog post discusses various ideas for secure login systems.


ticki.github.io




 Sandeep ▾

 Recommend


 Tweet

 Share

Sort by Best ▾



Join the discussion...



Volker Diels-Grabsch · 3 years ago · edited

Very good explanation of how to evaluate hash functions!

Here are my 2 cents:

It would have been nice to get the exact formula by which the diffusion map is computed. Or, to point to another article or Wikipedia entry where this is explained.

There are two small typos in section "Combining diffusions", using  $s'$  (the new state) where  $s$  (the old state) is meant.

The article says:

$$s' = d(f(s',x))$$

It should be:

$$s' = d(f(s,x))$$

Then, the article says:

$f(s',x)$  is what we call our combinator function.

This should be:

$f(s,x)$  is what we call our combinator function.

(Comment cross-posted from HN: <https://news.ycombinator.co...> )

^ | v · Reply · Share ›

