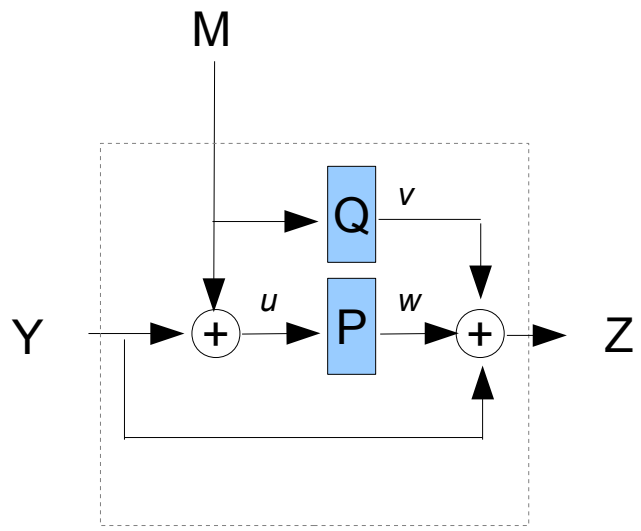
Some notes on Grøstl
John Kelsey, NIST, April 2009

These are some quick notes on some properties and observations of Grøstl. Nothing in this note threatens the hash function; instead, I'm pointing out some properties that are a bit surprising, and some broad approaches someone might take to get attacks to work. I've discussed these with the Grøstl team, and gotten quite a bit of useful feedback. I'm pretty sure they agree that my observations are correct, but obviously, any errors here are mine alone.

1 Grøstl Compression Function

The Grøstl compression function is built from two different fixed permutations, P and Q, on 2n bits, where n is the final hash output size. The design is quite interesting; among other things, this (like LANE and Luffa, among others) takes away the key schedule as a way of affecting things inside P and Q.

Here is a picture of the Grøstl compression function:



In this diagram, I've labeled the input hash chaining value Y, the output Z, and the message M. Similarly, I have introduced variables for the internal values inside the compression function--u, v, and w. I'll use this notation a lot in the next few pages to simplify discussion. To write this in terms of equations:

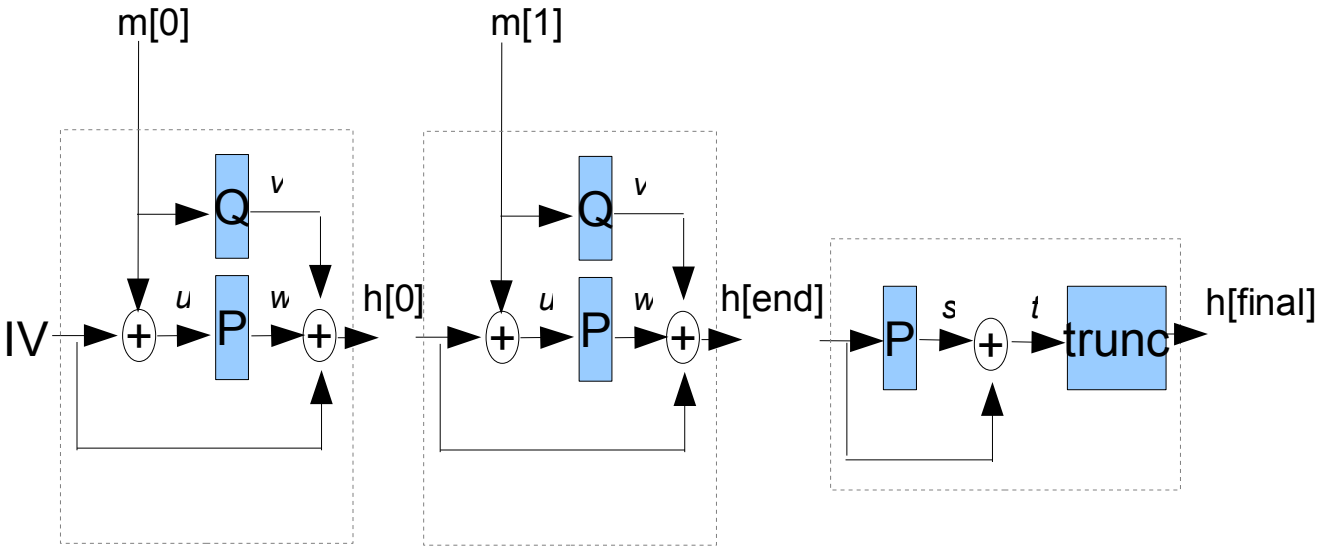
u = Y xor M
v = Q(M)
w = P(u)
Z = v xor w xor Y

or, in a shorter form:

Z = Q(M) xor P(Y xor M) xor Y

It's important to remember here that all variables are 2n bits, so for a 256-bit hash, they're 512 bits each.

Now, the hash function also includes a final transformation, and it's useful to look at the hash function as a whole. So this is the full hash function as it would be used to hash a two block message:



In this diagram, I've marked the first message block as m[0], the second as m[1] (which includes the padding), and have also included the output transformation. The box marked "trunc" simply chops half the bits from the value before releasing it; this is how the 512 bit intermediate states are used to ultimately produce a 256 bit hash.

Now, a pretty trivial observation here is that any single-block collision is going to have to put a nonzero difference into both permutations. This is inevitably going to complicate the attack, as:

- The same XOR difference must work for P and Q.
- If we assume the hash chaining value coming in is fixed (for example, the IV), then the attacker is quite constrained--by controlling the input to Q, the attacker gives up control of the input to P. Alternatively, the attacker can get more choice of what goes into P by losing some control over Q.

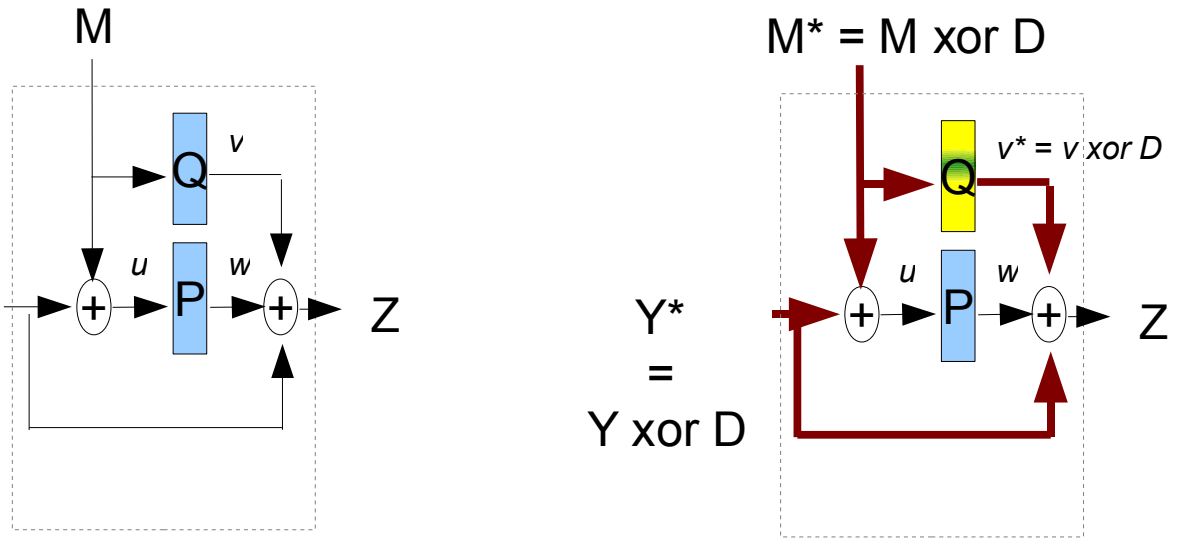
To me, this suggests strongly that the right way to attack Grøstl will be with multiblock collisions. The first block of the message with a difference in it will have both P and Q active, but later message blocks can restrict their control to only P or only Q. I'll talk a bit about how to do that below.

2 XOR Differences in Hash Values and Chosen Inputs in the Permutations

There is an interesting property brought about by the compression function constuction used in Grøstl: Once I know any pair of inputs M,M* with the property that:

M xor M* = Q(M) xor Q(M*) = D

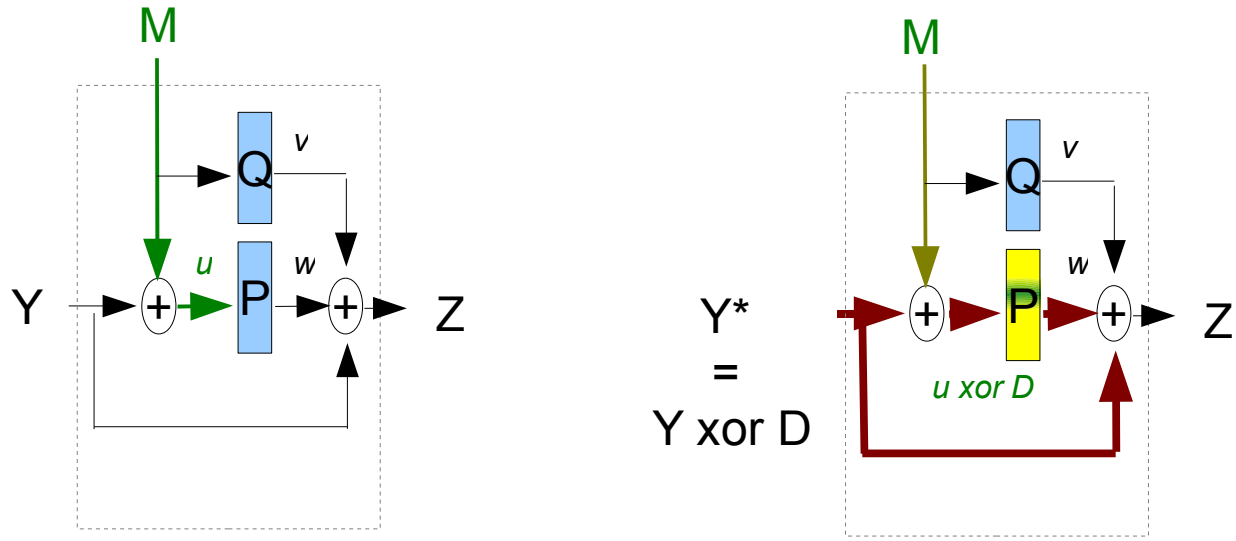
I can *always* get a collision in the hash function, once I get a pair of hash chaining values whose XOR difference is D. This diagram shows what's going on:



In the diagram, we know a pair M, M* such that M* = M xor D, and Q(M) xor Q(M*) = D. Note that here, we're talking about actual values input into Q which have a known XOR difference in and out. Now, given knowledge of such a pair, if we ever get a pair of hash inputs Y,Y* such that Y* = Y xor D, we can force a collision.

The interesting thing about this is that it is independent of the *values* of Y and Y*. We care about the specific values of M and M* for this to work, but we have no need to care about the specific values of Y and Y*; instead, we care only about the XOR difference between them. This is quite different from the behavior of a hash function like SHA256 or Whirlpool, where we would be quite surprised to have the same message pair lead to collisions for absolutely any pair of input chaining values with the right XOR difference.

We can do a variant of the same trick using permutation P:

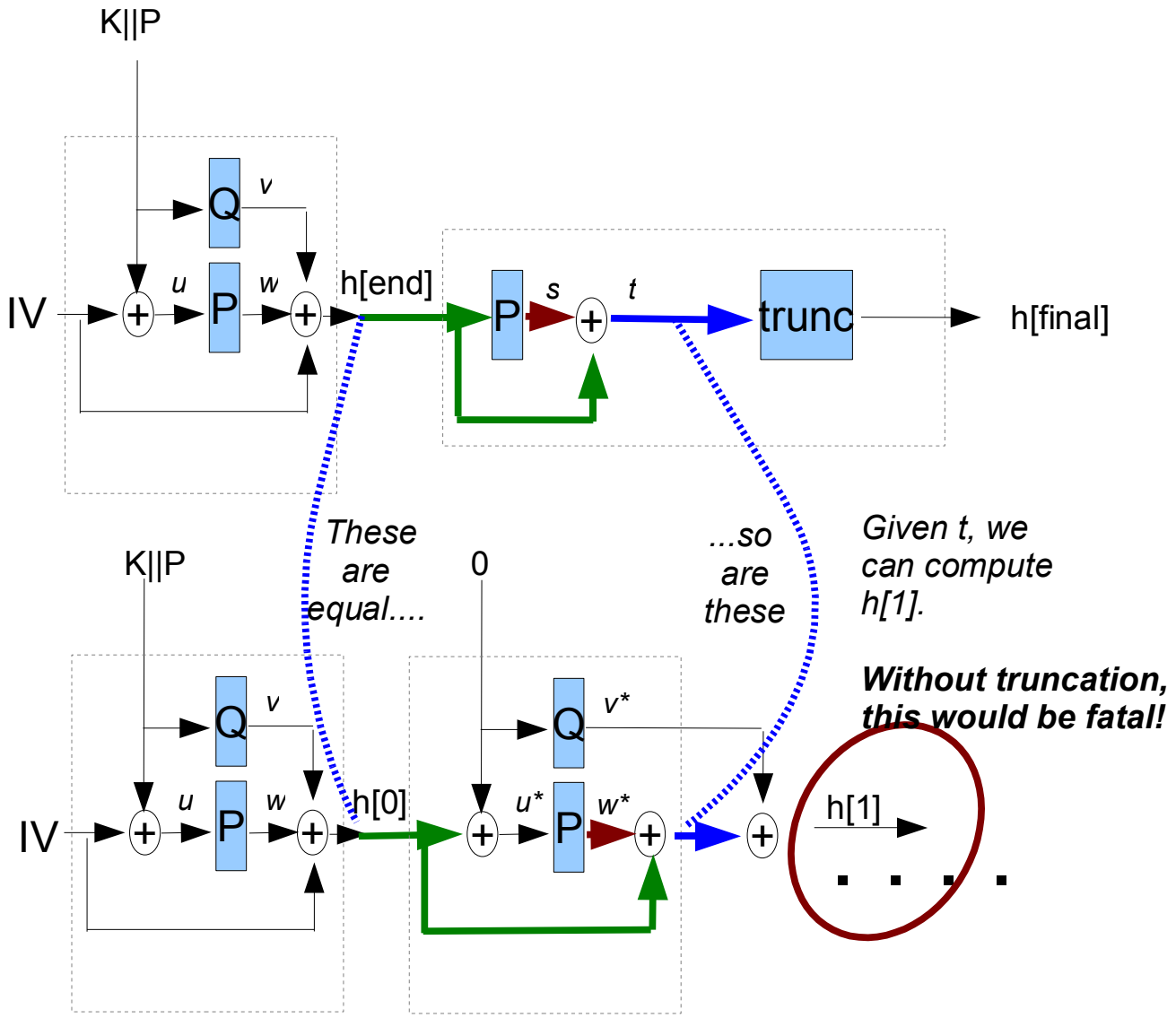


In this diagram, we start by knowing a pair of inputs u,u* such that u xor u* = P(u) xor P(u*) = D. Now, given any pair Y,Y* such that Y xor Y* = D, we can get the collision to work: We use M (shown in green) to force the input to P on the left to go to u. This also forces the input on the right to go to

u* = u xor D.
Again, we would not expect this for a compression function, in general; it's a specific and interesting property of the Grøstl compression function. I have no idea how to use this to get any further on Grøstl, but it does seem like the kind of observation that might come in handy somehow.

Søren Thompson, one of the Grøstl designers, has pointed out in an email that this property is implicit in the discussion of Wagner's generalized birthday attack in the Grøstl submission document. However, since it wasn't obvious to me until I spent a bit of time looking, I think it's worthwhile to point it out here.

3 Most of the Output Transformation Doesn't Help



Grøstl includes an output transformation. Since it is a wide pipe design, some kind of output transformation is needed to map the 512 bit intermediate state to a 256 bit final hash value. In Grøstl, this is done by applying P once more to the result of the compression function on the final message block, doing a feedforward XOR, and then truncating the result. That is, if h[end] is the result of the

compression function on the last padded message block, then

h[final] = Truncate(P(h[end]) XOR h[end]).

The output transformation is intended, among other things, to block the length extension attack. However, the use of P and the feedforward here makes the resulting hash no stronger against length extensions than it would be, if the output transformation included nothing but truncation.
To see this, look at the above picture, and imagine that we eliminated the truncation from the output transformation. In that case, the final hash of the first message, K (shown with padding P) would be the value of the variable

t = P(h[end]) XOR h[end].

We're showing the first two compression function computations of the process of hashing a second message, consisting of the two blocks K||P and 0. From the diagram, it's easy to see that the output of the second compression function is

P(h[0]) xor h[0] xor Q(0) = P(h[end]) xor h[end] xor Q(0)

In this, I'm using the fact that h[0] in hashing second message equals h[end] in hashing the first message; this is easy to see in the diagram.

Now, note that we can compute Q(0) without knowing anything about K||P. This means that if we were given the non-truncated value t from the hashing of the first message, we could use it to compute h[1] from the second message. That would allow us to use knowledge of hash(K) to compute hash(K||P||0|| x) for any x. In other words, without the truncation in the output transformation, the hash would be vulnerable to a length-extension attack. (This attack works just as well if K is a whole block and the padding P appears in the second block--that gives the attacker a more powerful attack, but it also requires a more complicated diagram to explain.)

Grøstl's output transformation includes truncation, however, so we don't have a length extension attack here. However, it's interesting to note that the output transformation seems to be providing little security benefit except for that truncation. I think the output transformation would accomplish more, if it used Q instead of P as its permutation. Given the Grøstl compression function, the intermediate hash value never goes through Q in the normal operation of the hash, and so the result of computing Q(h[end]) xor h[end] would not give any information about the intermediate state of a different hash computation.

Praveen Gauravarum and Nasour Bagheri have used the above observation to show that without truncation, Grøstl is differentiable from a random oracle.

4 Wrapup

I've shown a couple of interesting and surprising properties of the Grøstl hash function, but nothing that suggests any particular weakness. The output transformation accomplishes less than it should, but still seems to resist the relevant attacks. An attacker can in general concern himself with specific values of inputs to the permutations P and Q, but need only worry about XOR differences in the chaining values, but since the chaining values are twice the length of the hash output, this doesn't seem to lead to any obvious weaknesses. I hope these observations are of use to other people trying to analyze Grøstl.