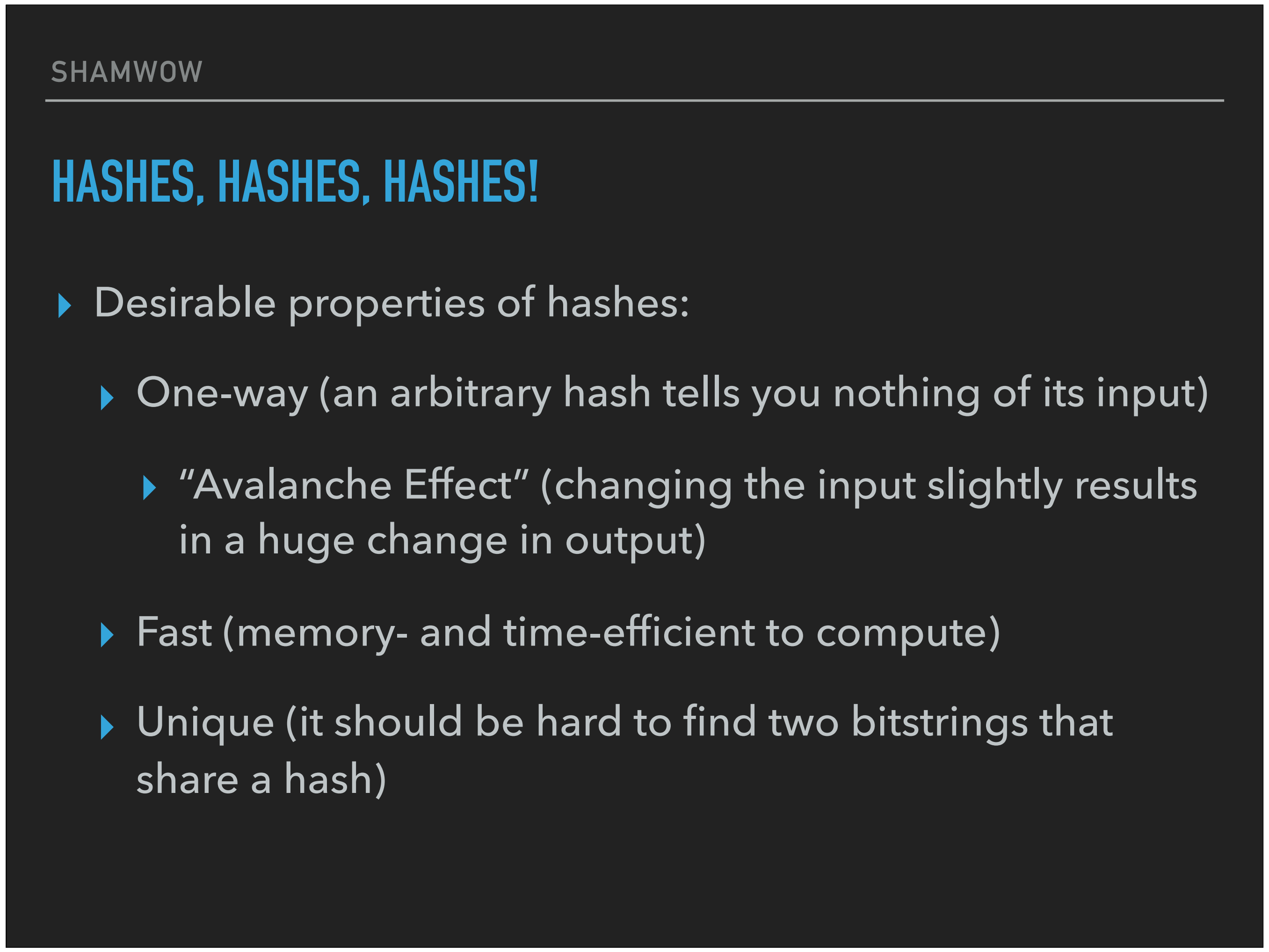
PWL NYC LIGHTNING TALK

SHAMWOW

so we use hashes a lot as programmers
even if we don't work directly with crypto
so I got curious about how it actually worked
like, how does the empty string return a hash? Why is it computationally hard to reverse a hash? What's it doing in there?



HASHES, HASHES, HASHES!

- Desirable properties of hashes:
  - One-way (an arbitrary hash tells you nothing of its input)
    - “Avalanche Effect” (changing the input slightly results in a huge change in output)
  - Fast (memory- and time-efficient to compute)
  - Unique (it should be hard to find two bitstrings that share a hash)

One way: hashes are not *encryption*, which is two-way. (I mean, ideally, encryption also leaves no trace of its input, but that’s only possible with one-time pads, I believe.)
They cannot be “reversed” except by brute-force or some flaw in the hash algorithm itself.
Part of this is the “avalanche effect”; the hash for the phrase “I am a cat” and “I am a hat” are entirely dissimilar to each other.
We want them to be fast to compute, because they’re used for checksums and the like (this is *not* a desirable property in some systems, however; see scrypt and other password-protection hashes that are slow to compute so that even a leaked list of passwords isn’t useful.)
And Unique — it should be hard to find collisions, so that you can’t substitute one bit string for another and have the same hash returned. (This is why SHAttered was a big deal — researchers found a way to generate visually distinct PDFs that nonetheless had the same hash)

SHAMWOW

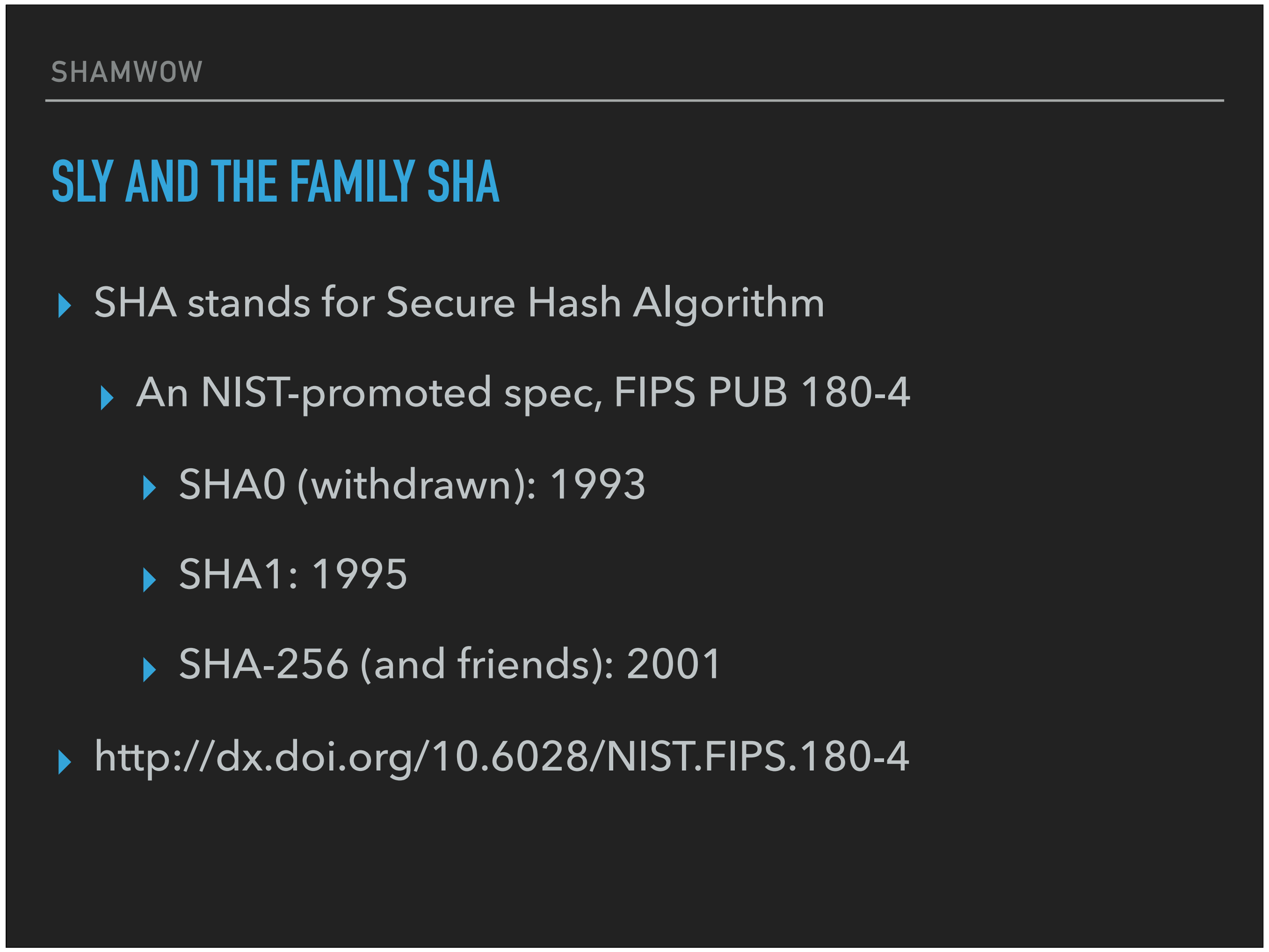
KITTY PURRY
@rabcyr_alt

I fought the SHA but the SHA1

9:22 PM - 3 Oct 2017

1,366 Retweets 4,118 Likes

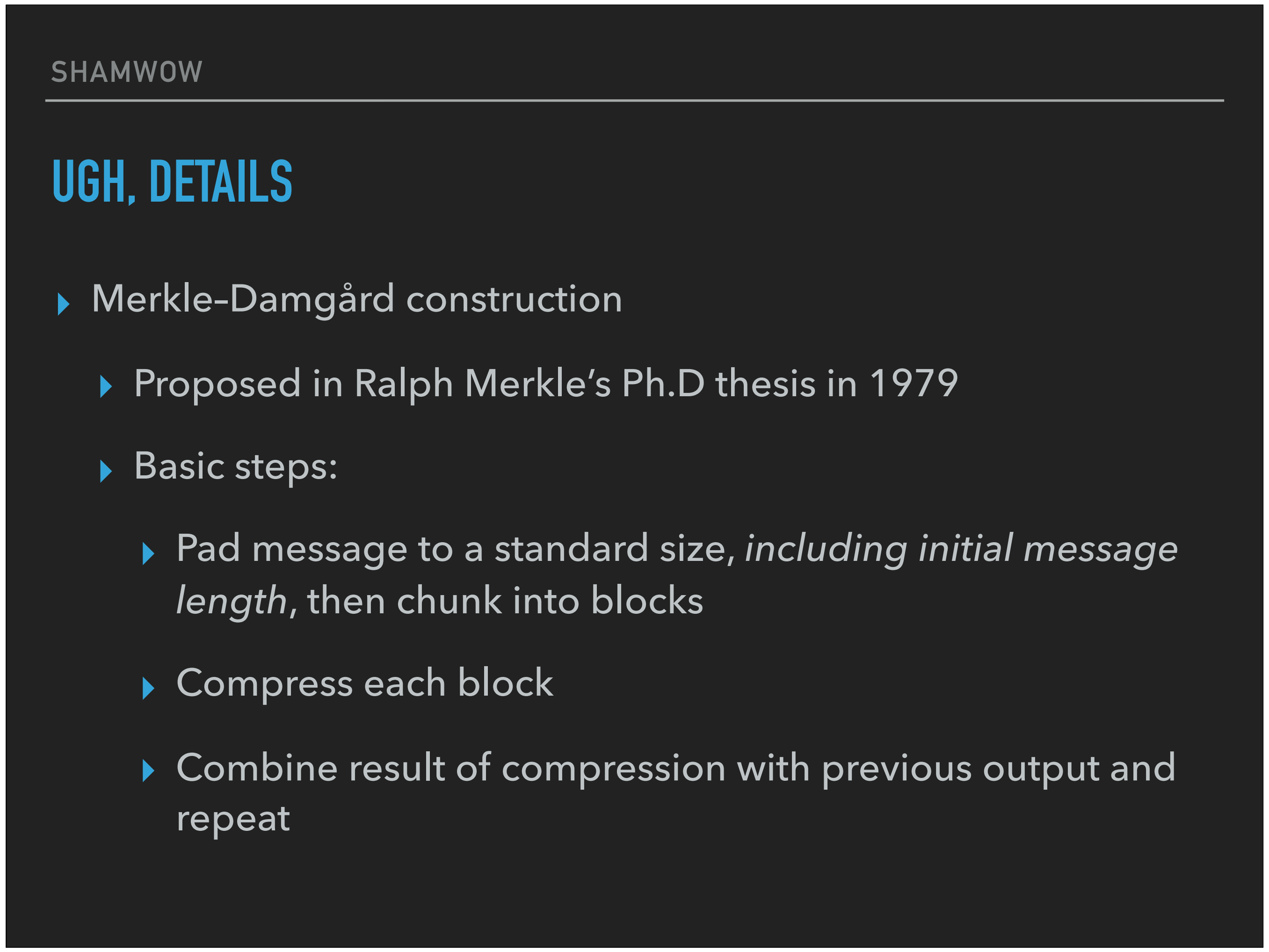
We’re not talking about SHA1 today but I couldn’t resist this tweet.



SLY AND THE FAMILY SHA

- SHA stands for Secure Hash Algorithm
  - An NIST-promoted spec, FIPS PUB 180-4
    - SHA0 (withdrawn): 1993
    - SHA1: 1995
    - SHA-256 (and friends): 2001
- http://dx.doi.org/10.6028/NIST.FIPS.180-4

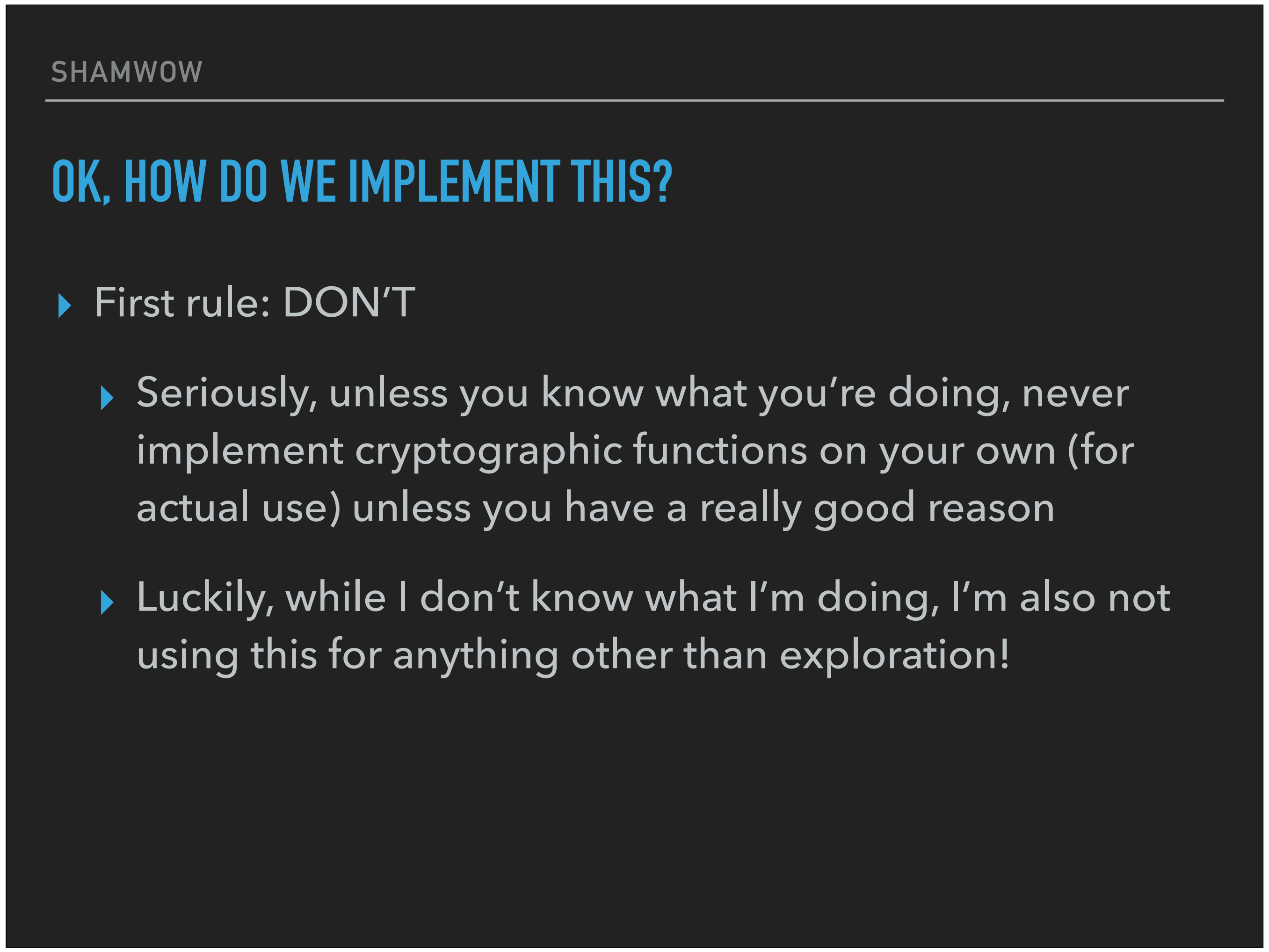
SHA was part of the “Capstone Project” from the NSA, the same project that gave the world the Clipper chip.
Initial publication of SHA was in 1993, but it was withdrawn and republished two years later with a slight variation; these are now known as SHA-0 and SHA-1
SHA2 was an update to the SHA standard, making it more secure (more bits!). It has a number of variations that differ in how many bits are in the resulting hash; the 256-bit version is the most common, and therefore people generally use “SHA2” and “SHA256” interchangeably.
SHA3 is a completely new hash, not using the same techniques as SHA1 and SHA2; it’s based on something called Keccak, and if you want to no more, I encourage you to give a lightning talk on the subject.



UGH, DETAILS

- Merkle-Damgård construction
  - Proposed in Ralph Merkle’s Ph.D thesis in 1979
  - Basic steps:
    - Pad message to a standard size, *including initial message length*, then chunk into blocks
    - Compress each block
    - Combine result of compression with previous output and repeat

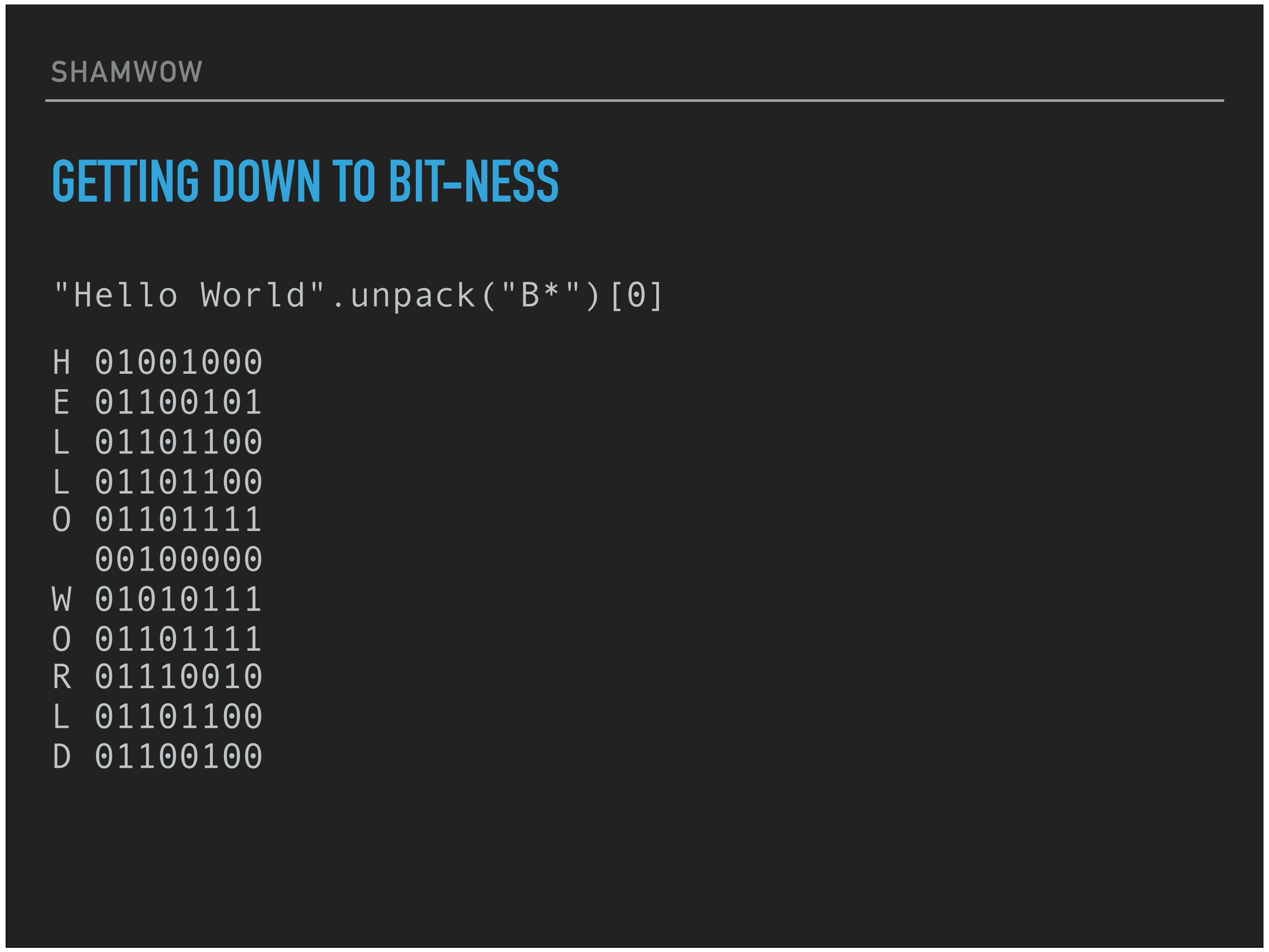
Ralph Merkle and Ivan Damgård independently proved that this is cryptographically secure as long as the compression you’re doing is secure



OK, HOW DO WE IMPLEMENT THIS?

- First rule: DON'T
  - Seriously, unless you know what you're doing, never implement cryptographic functions on your own (for actual use) unless you have a really good reason
  - Luckily, while I don't know what I'm doing, I'm also not using this for anything other than exploration!

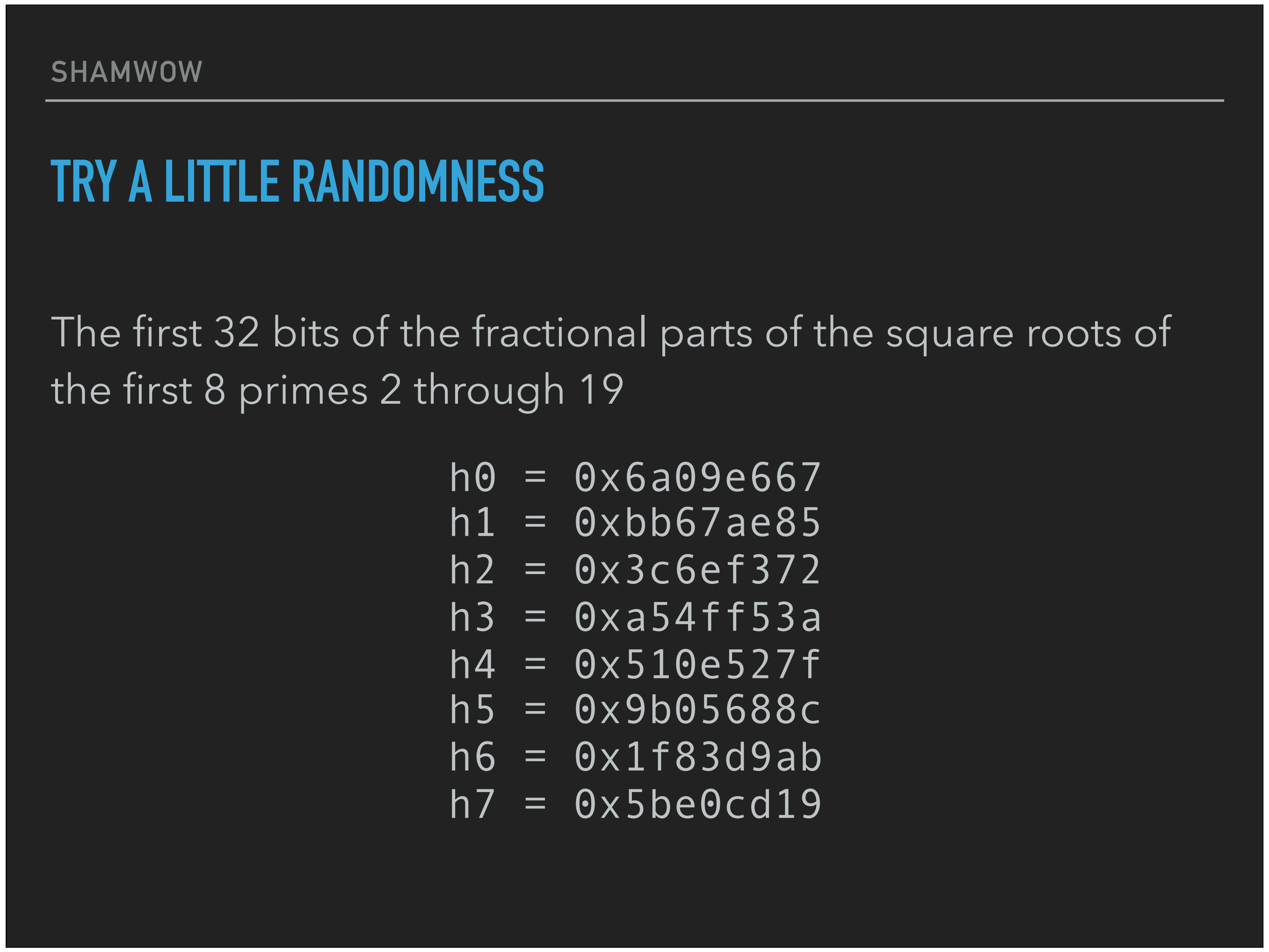
There are all sorts of things you can do wrong. Timing attacks, side-channels, padding bugs… just trust the experts when you need a hash function.



GETTING DOWN TO BIT-NESS

```
"Hello World".unpack("B*")[0]

H 01001000
E 01100101
L 01101100
L 01101100
O 01101111
  00100000
W 01010111
O 01101111
R 01110010
L 01101100
D 01100100
```

"First, split your input into 32-bit numbers"
OK, I'll just use Ruby's, uh...
Ruby just has integers. No width is specified.
Well, OK, whatever, some googling tells me the magic I want to turn our string into a big pile of bits is `unpack("B*")` so let's start there and work out this "32-bit integer" problem later

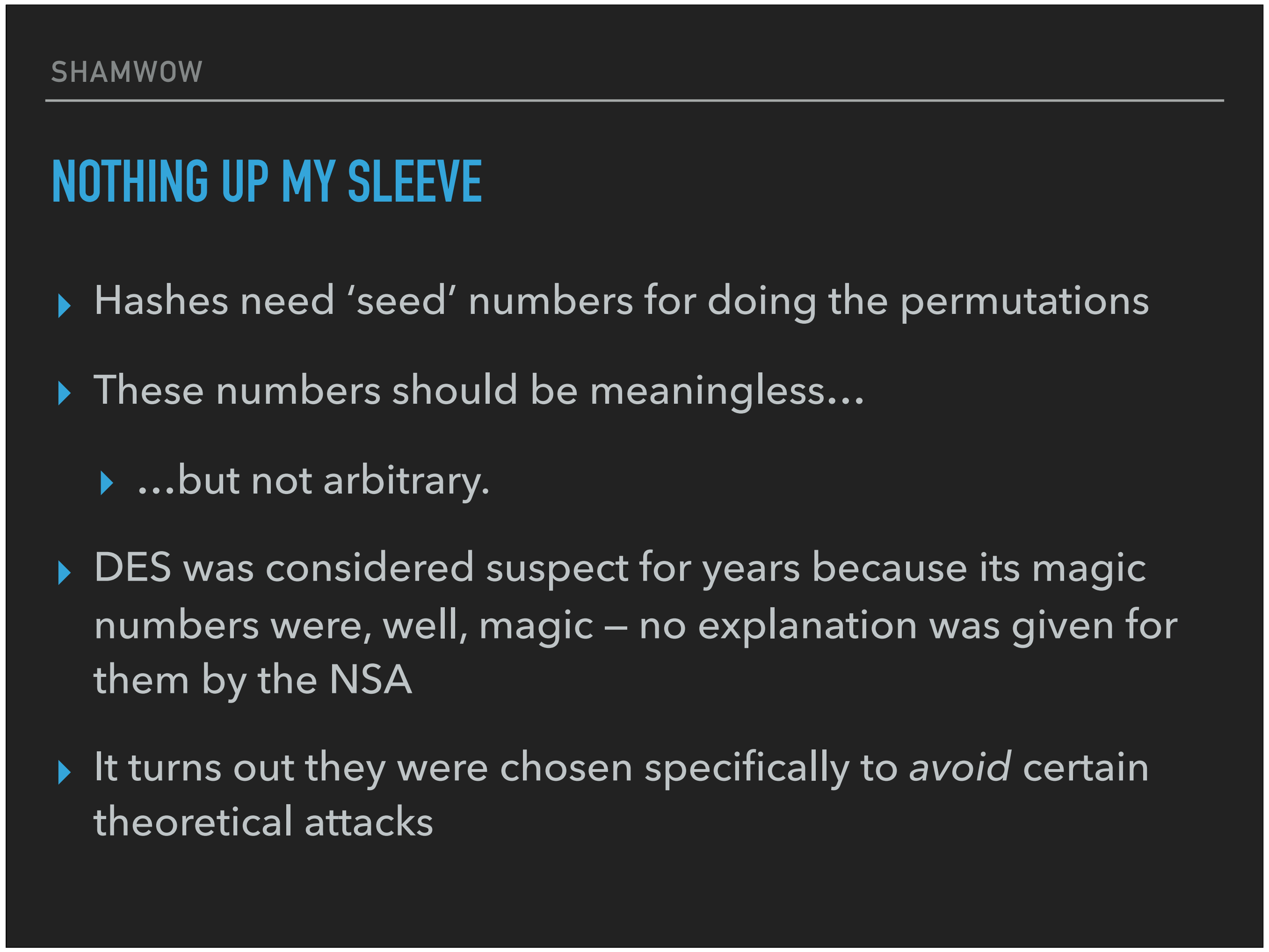


TRY A LITTLE RANDOMNESS

The first 32 bits of the fractional parts of the square roots of the first 8 primes 2 through 19

```
h0 = 0x6a09e667
h1 = 0xbb67ae85
h2 = 0x3c6ef372
h3 = 0xa54ff53a
h4 = 0x510e527f
h5 = 0x9b05688c
h6 = 0x1f83d9ab
h7 = 0x5be0cd19
```

I also need all these magic hex numbers. These will be the seeds for the start of the compression function inside the algorithm.



NOTHING UP MY SLEEVE

- Hashes need 'seed' numbers for doing the permutations
- These numbers should be meaningless…
  - …but not arbitrary.
- DES was considered suspect for years because its magic numbers were, well, magic – no explanation was given for them by the NSA
- It turns out they were chosen specifically to *avoid* certain theoretical attacks

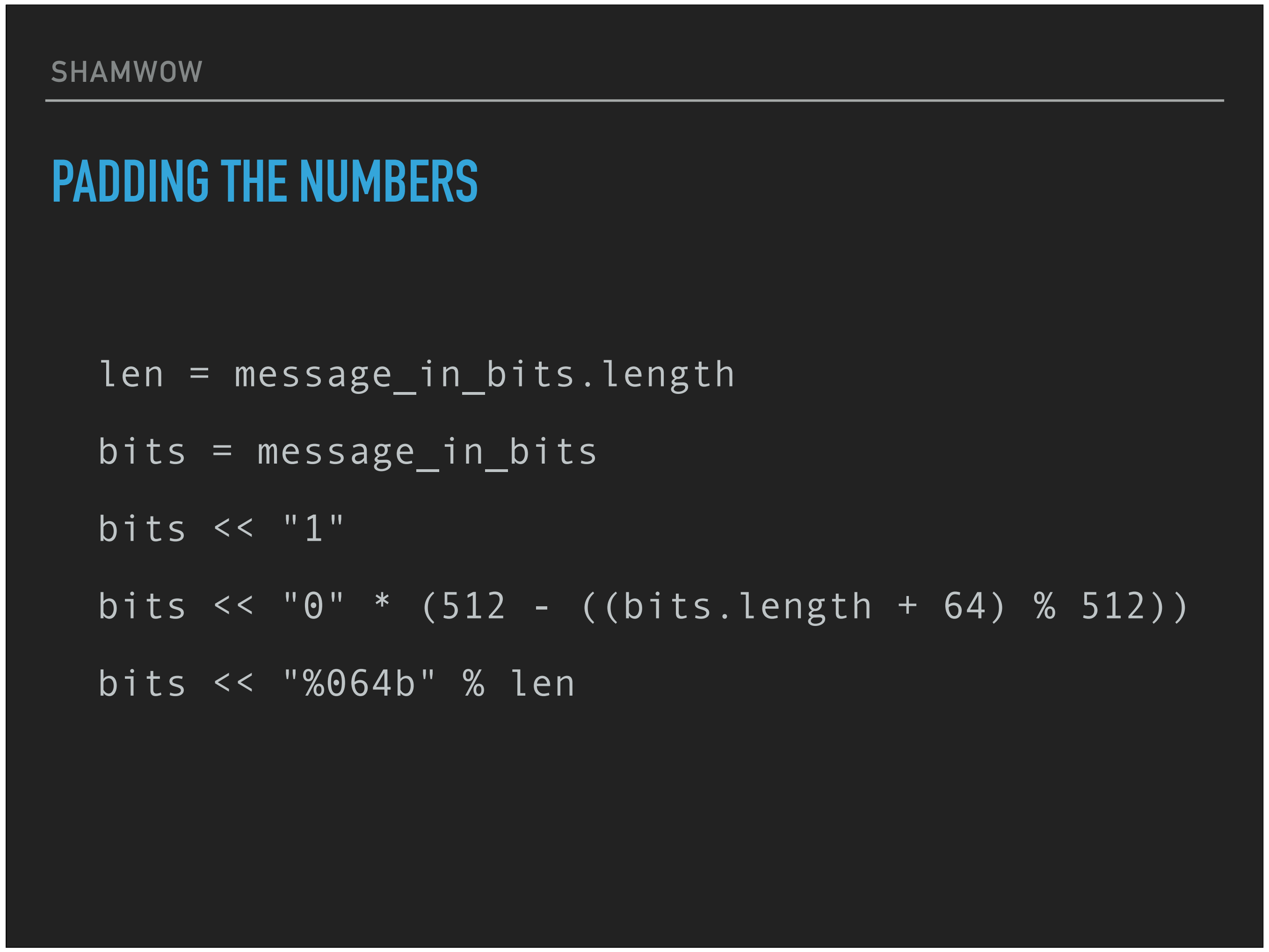
Nothing-up-my-sleeve numbers are used when you need random seed numbers, but in order to prove to everyone else that you haven't actually designed a fiendish backdoor into your system, you have to be able to explain why they aren't *arbitrary* numbers. For example, it'd be perfectly acceptable to say "the first 1000 digits of pi", but it would be very suspicious to say "the 2,509,374th through 2,510,899th digits of pi" -- people would assume you were picking values that somehow advantaged you for breaking the hash later.

OK, A LOT OF RANDOMNESS

The first 32 bits
of the fractional parts
of the cube roots
of the first 64 primes
2 through 311

THIS WON'T BE ON THE FINAL

```
k = [
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01,
0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,
0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da, 0x983e5152,
0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc,
0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,
0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116, 0x1e376c08,
0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f,
0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]
```



PADDING THE NUMBERS

```
len = message_in_bits.length
bits = message_in_bits
bits << "1"
bits << "0" * (512 - ((bits.length + 64) % 512))
bits << "%064b" % len
```

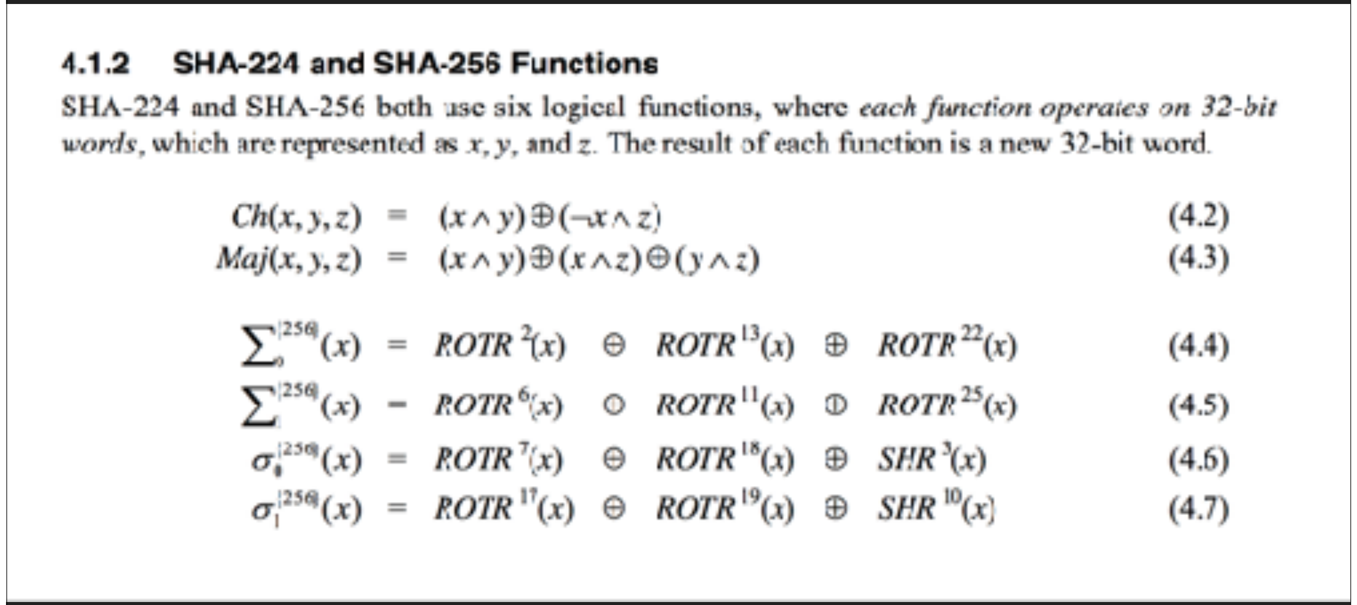
Anyway, back to the algorithm. Take your message as bits, append a `1` bit, append enough 0s to get it up to the next multiple of 512, then set the last 64 bits to the length of your message. Note this is one of the reasons that the hashes for "0" and "00" are different -- although we're padding our message with zeros, the extra 1 bit and the length serve as differentiators. This is one of Merkle–Damgård’s insights that makes the padding of the message secure.

BITS, CHUNKS, AND WORDS

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

Once we've done all that, We’re going to split this into chunks of 512 bits, and then loop over the chunks and, for each chunk, squirt the bits into an empty 64-element array, turning them into 32 bit “words”, and start massaging them.
Those of you doing math in your heads are noticing that it only fills up the first quarter of the array. To generate the rest of the array, we start bit-shifting!

I THOUGHT THIS WAS CRYPTOGRAPHY, NOT LATIN CLASS



4.1.2 SHA-224 and SHA-256 Functions

SHA-224 and SHA-256 both use six logical functions, where *each function operates on 32-bit words*, which are represented as x, y, and z. The result of each function is a new 32-bit word.

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \quad (4.2)$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \quad (4.3)$$

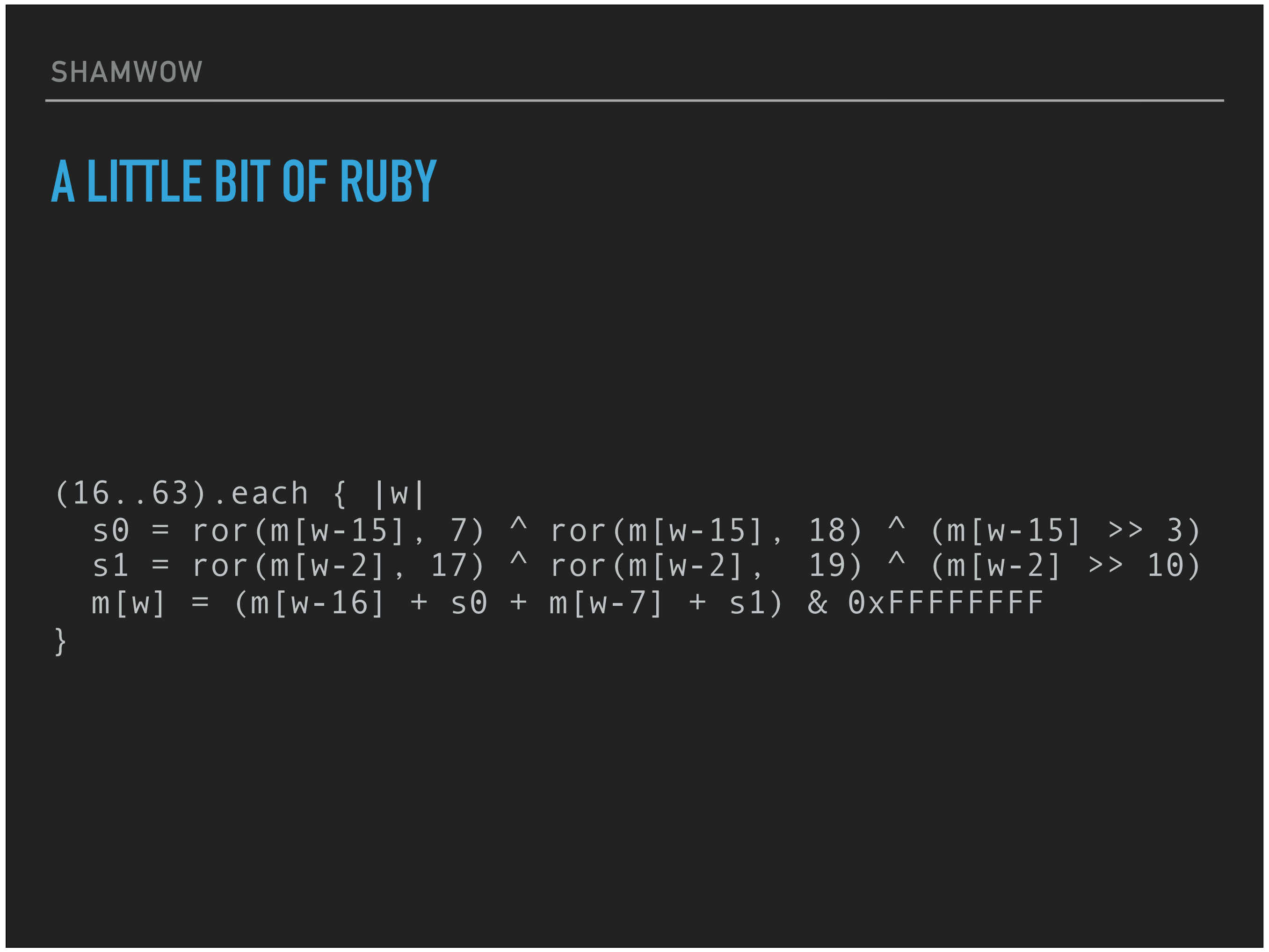
$$\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \quad (4.4)$$

$$\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \quad (4.5)$$

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \quad (4.6)$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \quad (4.7)$$

Here’s the functions that power the inner loop of the compression function.
^ is AND (1 if both are 1, 0 otherwise)
Plus-circle is XOR (which returns 1 if the two input bits differ but 0 if they’re the same)
The angle-bracket in Ch before the x is NOT (flip from 0 to 1 and vice versa)
SHR is just bit-wise right-shift
And ROTR. We’ll come to those in a second.

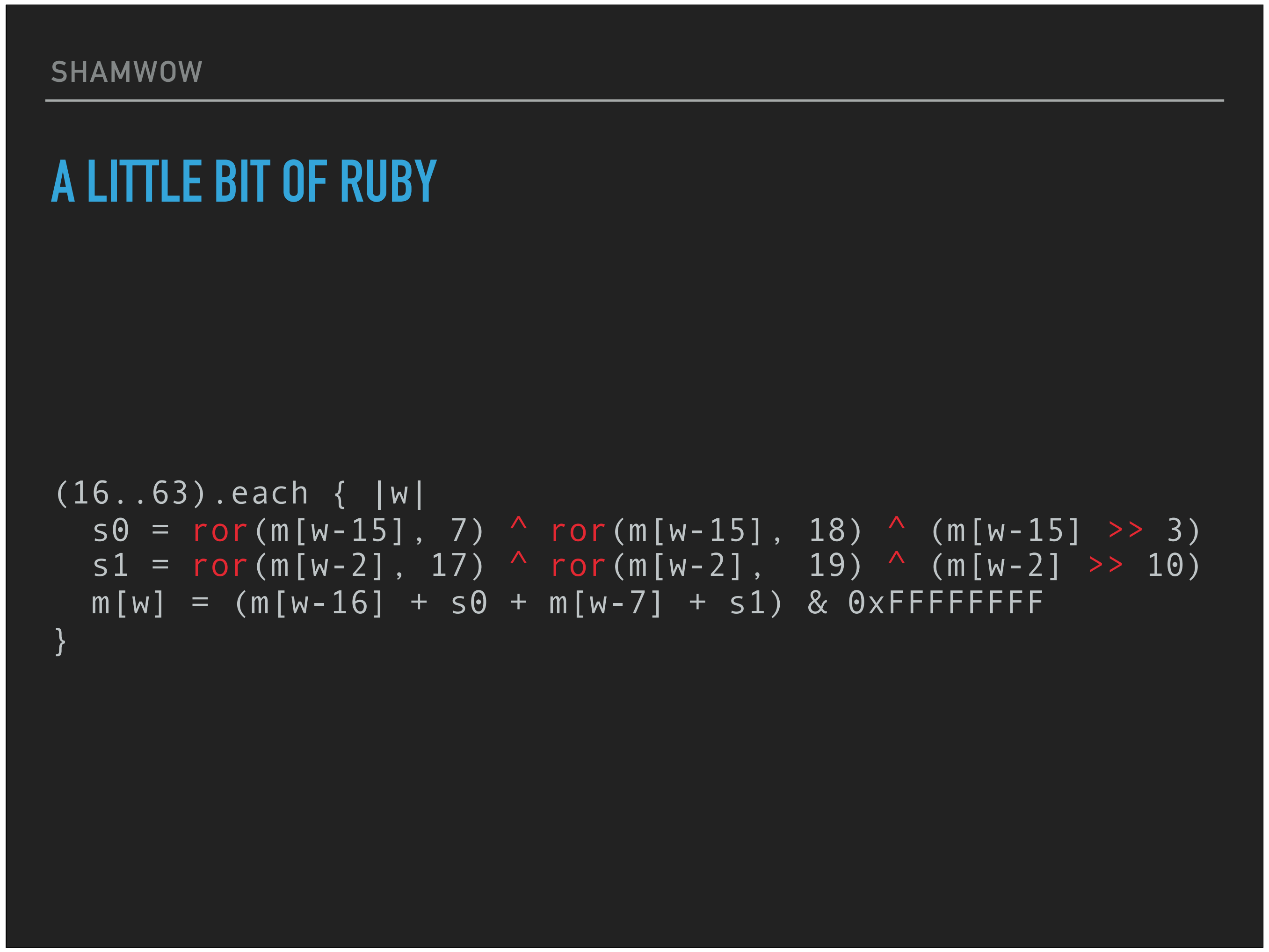


A LITTLE BIT OF RUBY

```ruby
(16..63).each { |w|
  s0 = ror(m[w-15], 7) ^ ror(m[w-15], 18) ^ (m[w-15] >> 3)
  s1 = ror(m[w-2], 17) ^ ror(m[w-2],  19) ^ (m[w-2] >> 10)
  m[w] = (m[w-16] + s0 + m[w-7] + s1) & 0xFFFFFFFF
}
```

What does that look like in code?

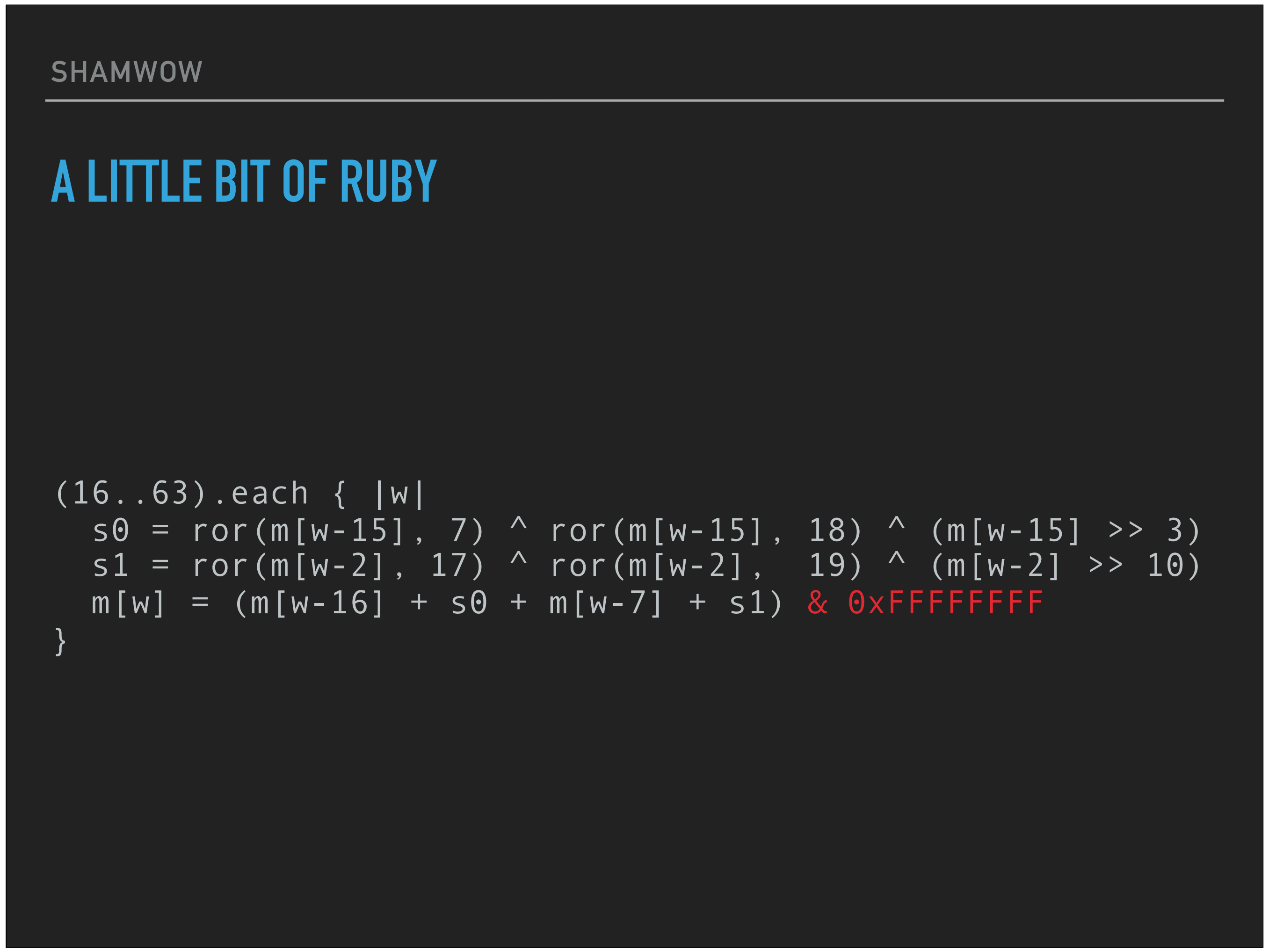
Here's the first and second "aha" moments while I was doing this.



A LITTLE BIT OF RUBY

```
(16..63).each { |w|
  s0 = ror(m[w-15], 7) ^ ror(m[w-15], 18) ^ (m[w-15] >> 3)
  s1 = ror(m[w-2], 17) ^ ror(m[w-2],  19) ^ (m[w-2] >> 10)
  m[w] = (m[w-16] + s0 + m[w-7] + s1) & 0xFFFFFFFF
}
```

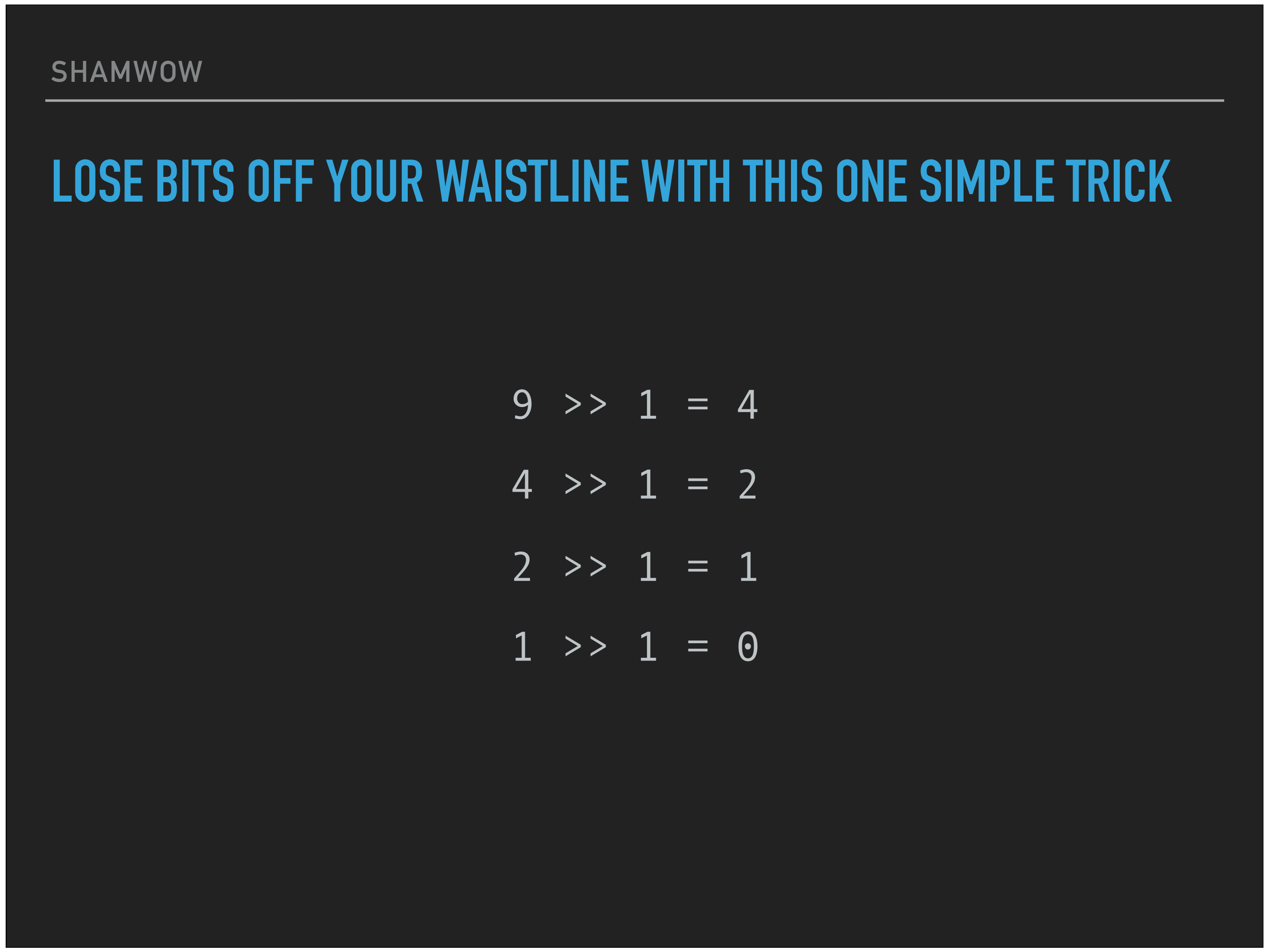
When you're doing bitwise operations on numbers, they'll never "jump the tracks", as it were. For example: an XOR or an AND never push bits between columns; they only operate within a bit! So you can XOR and ROR to your hearts content without worrying about popping outside of a 32-bit interger.



A LITTLE BIT OF RUBY

```ruby
(16..63).each { |w|
  s0 = ror(m[w-15], 7) ^ ror(m[w-15], 18) ^ (m[w-15] >> 3)
  s1 = ror(m[w-2], 17) ^ ror(m[w-2],  19) ^ (m[w-2] >> 10)
  m[w] = (m[w-16] + s0 + m[w-7] + s1) & 0xFFFFFFFF
}
```

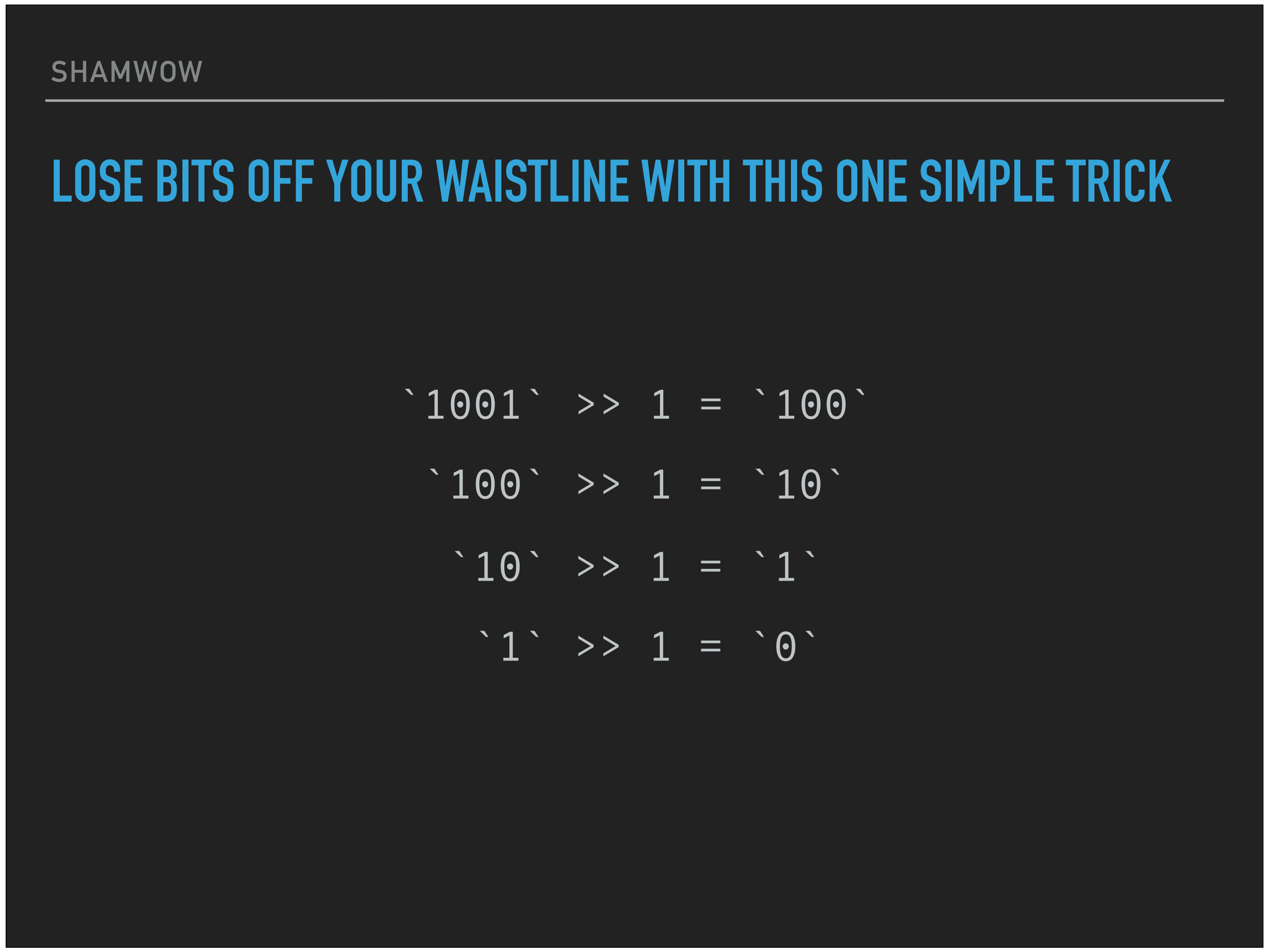
The second realization was that in the absence of proper 32-bit unsigned integers I can just fake them with a bitmask, so that when I do this addition, I can just lop off the high bits by ANDing the number with a 32-bit all-1 number (0xFFFFFFFF, which is easier to type than "1" 32 times).



LOSE BITS OFF YOUR WAISTLINE WITH THIS ONE SIMPLE TRICK

```
9 >> 1 = 4
4 >> 1 = 2
2 >> 1 = 1
1 >> 1 = 0
```

Right-shift “moves” the bits in a number to the right, and drops the right-most (that is, least-significant) digit. Here’s right-shift being run repeatedly on a number…

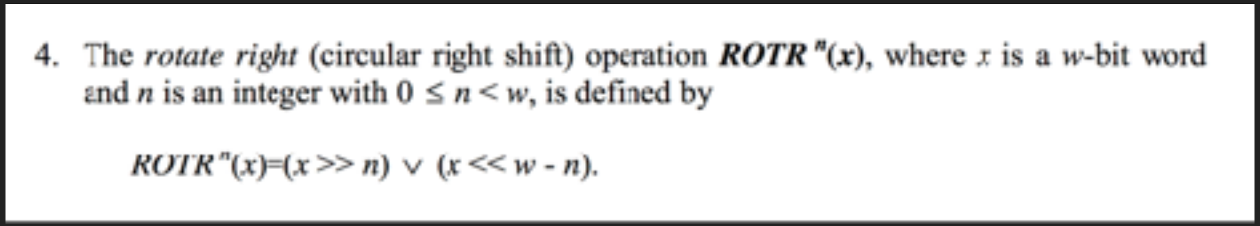


LOSE BITS OFF YOUR WAISTLINE WITH THIS ONE SIMPLE TRICK

```
`1001` >> 1 = `100`
 `100` >> 1 = `10`
  `10` >> 1 = `1`
   `1` >> 1 = `0`
```

…and here’s a somewhat easier to grok version actually showing the binary. This is pretty common in programming; it’s a fast way to divide by two with no remainder, for example.

SPIN ME RIGHT ROUND



4. The *rotate right* (circular right shift) operation ***ROTR*** ^{n}(***x***), where x is a w-bit word and n is an integer with $0 \le n < w$, is defined by

   $$ROTR^n(x) = (x >> n) \lor (x << w - n).$$

THERE IS A SEASON, TURN, TURN, TURN

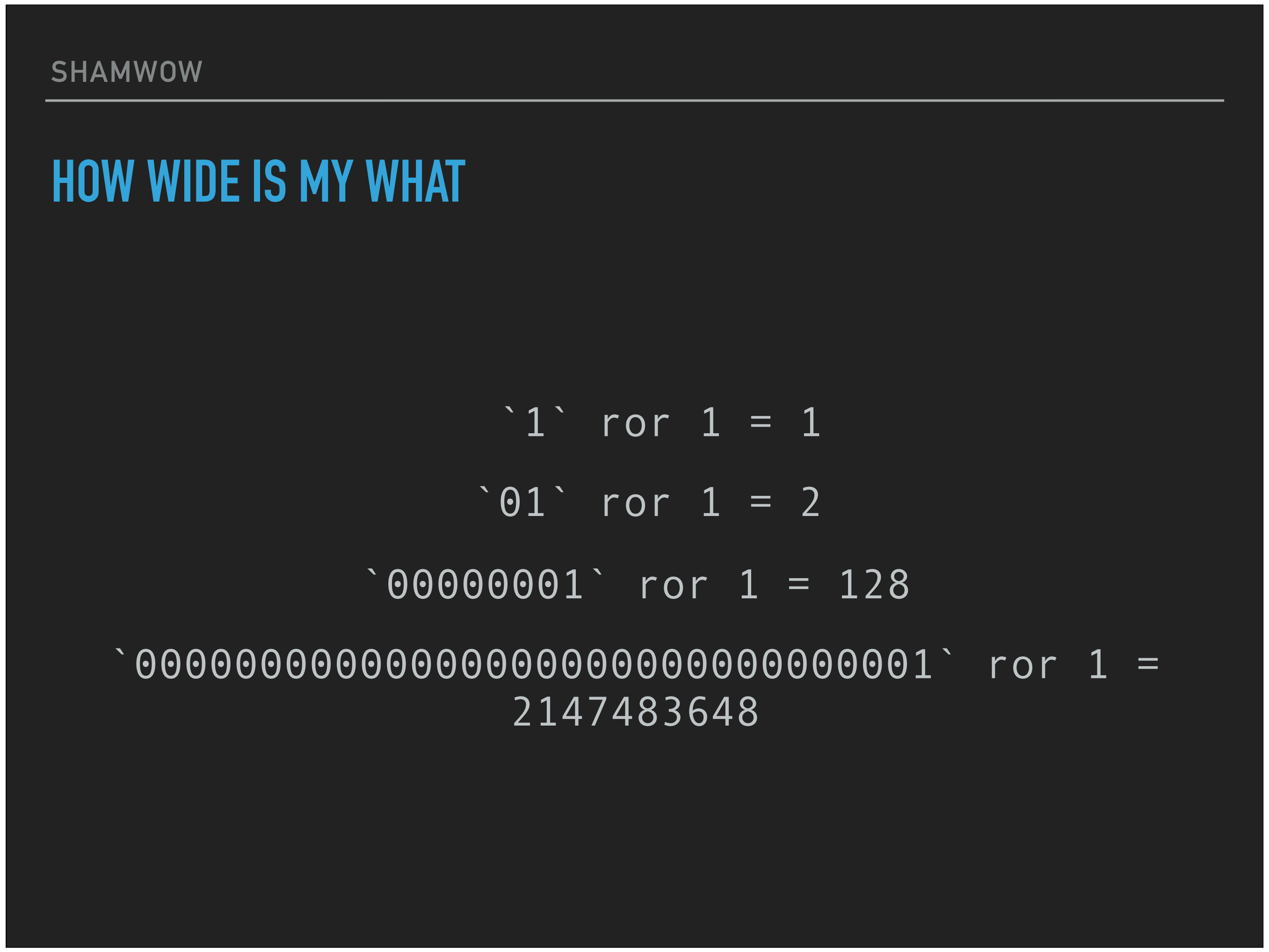
```
9  >>> 1 = 12
12 >>> 1 = 6
6  >>> 1 = 3
3  >>> 1 = 9
```

THERE IS A SEASON, TURN, TURN, TURN

```
`1001` >>> 1 = `1100`
`1100` >>> 1 = `0110`
`0110` >>> 1 = `0011`
`0011` >>> 1 = `1001`
```

THERE IS A SEASON, TURN, TURN, TURN

```
`1001` >>> 1 = `1100`
`1100` >>> 1 = `0110`
`0110` >>> 1 = `0011`
`0011` >>> 1 = `1001`
`1001` >>> 1 = `1100`
`1100` >>> 1 = `0110`
`0110` >>> 1 = `0011`
`0011` >>> 1 = `1001`
```



HOW WIDE IS MY WHAT

```
`1` ror 1 = 1

`01` ror 1 = 2

`00000001` ror 1 = 128

`00000000000000000000000000000001` ror 1 =
2147483648
```

Now, the other difference between right-shift and right-rotate is that right-rotate has to be "width-aware". After all, if you're rotating the number 1, is that `1`? `01`? `00000001`? `00000000000000000000000000000001`? The resulting value is wildly different depending on the answer!

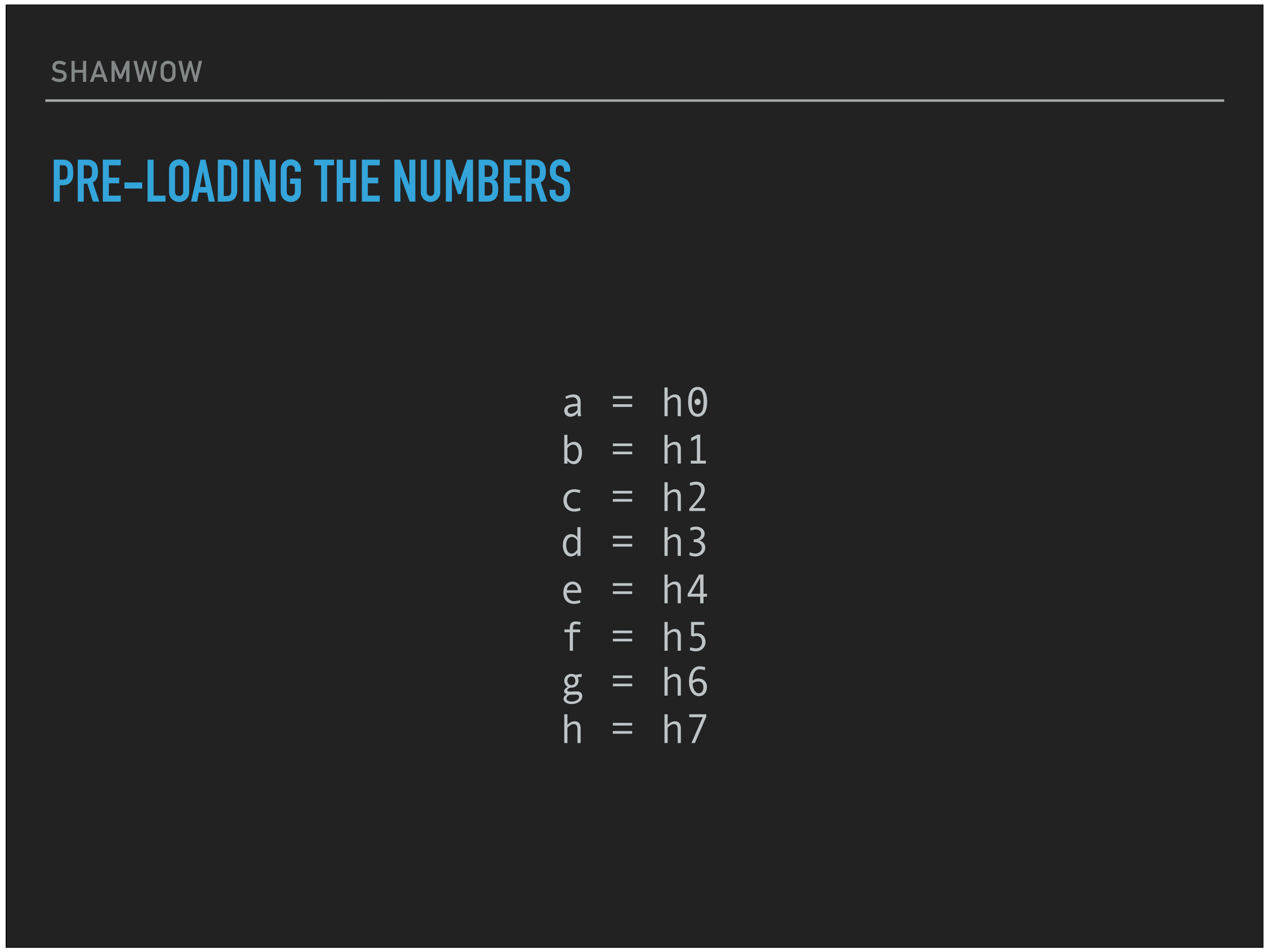
Luckily, in our case, we don't have to guess, since all of the numbers inside the algorithm are defined as 32-bit. Now, the first time I did this, I had this awful string-based for-loop manipulation built out. It was awful. But then I went back and actually read the spec again, and it defines it in terms of left and right-shifts!

BITMASKS FOR DUMMIES

```
(((num >> shift) | (num << (32-shift))) & ((2 ** 32) - 1))

(((1 >> 1) | (1 << (32-1))) & ((2 ** 32) - 1))

00000000000000000000000000000000 |
10000000000000000000000000000000 &
11111111111111111111111111111111

10000000000000000000000000000000

2147483648
```

BITMASKS FOR DUMMIES

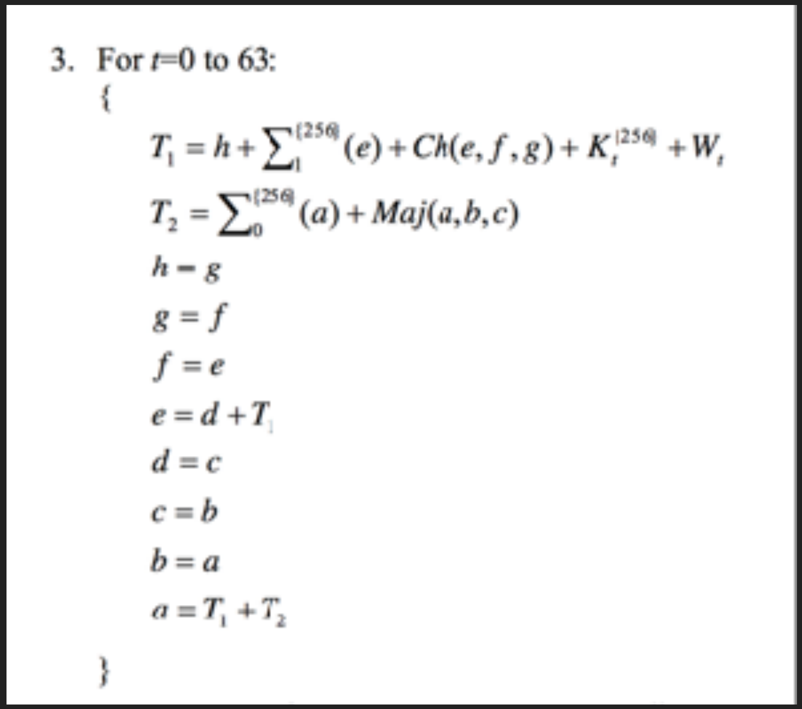
```
201 = `11001001`

(((num >> shift) | (num << (32-shift))) & ((2 ** 32) - 1))

(((201 >> 2) | (201 << (32-2))) & ((2 ** 32) - 1))

      00000000000000000000000000110010 |
11001001000000000000000000000000000000 &
      11111111111111111111111111111111

    01000000000000000000000000110010

    4294967496
```



PRE-LOADING THE NUMBERS

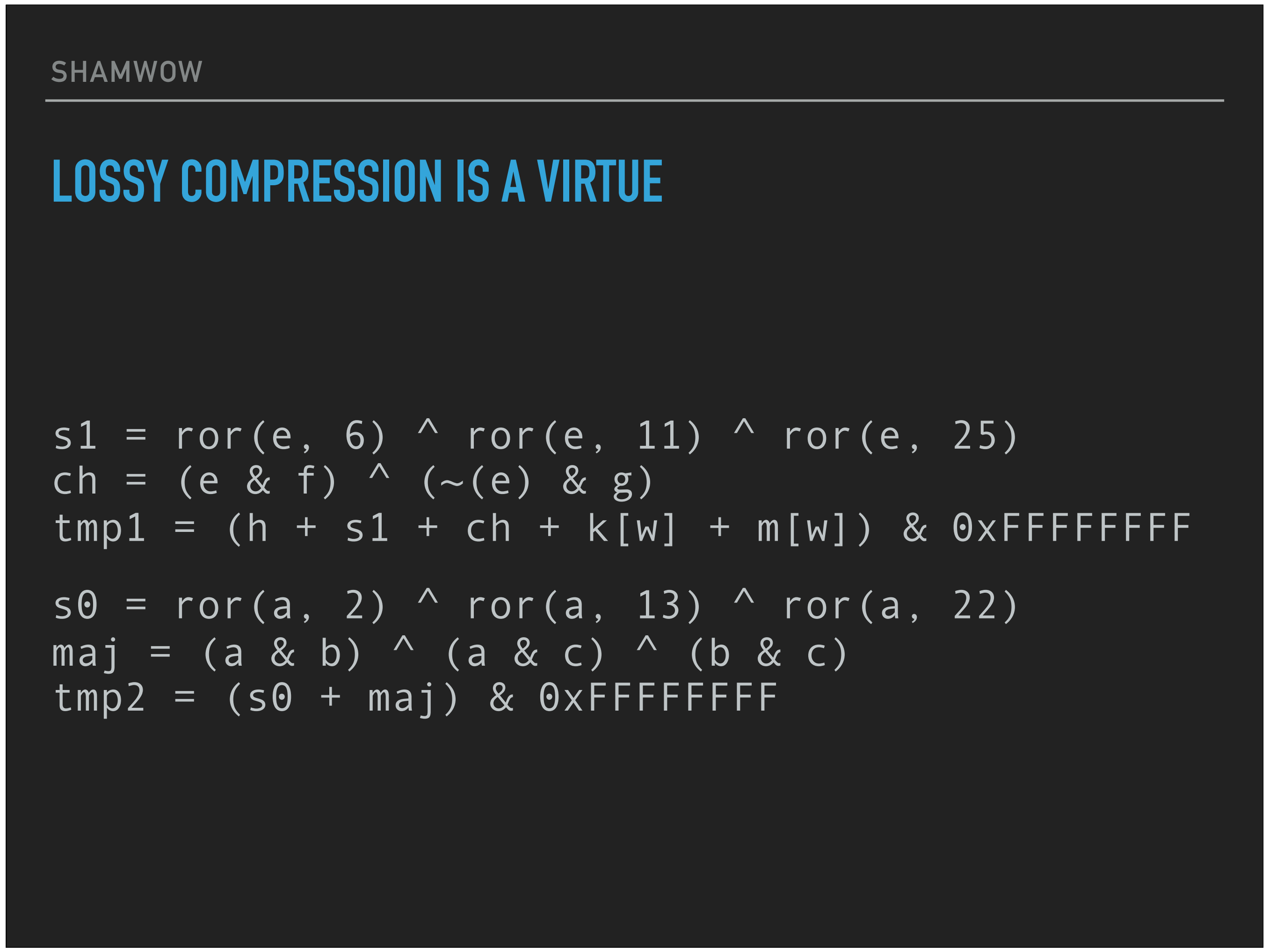
```
a = h0
b = h1
c = h2
d = h3
e = h4
f = h5
g = h6
h = h7
```

Assign eight variables; the starting values on the first loop are the fractional parts of the square roots I mentioned before.



```
3. For t=0 to 63:
   {
       $T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$
       $T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$
       $h = g$
       $g = f$
       $f = e$
       $e = d + T_1$
       $d = c$
       $c = b$
       $b = a$
       $a = T_1 + T_2$
   }
```

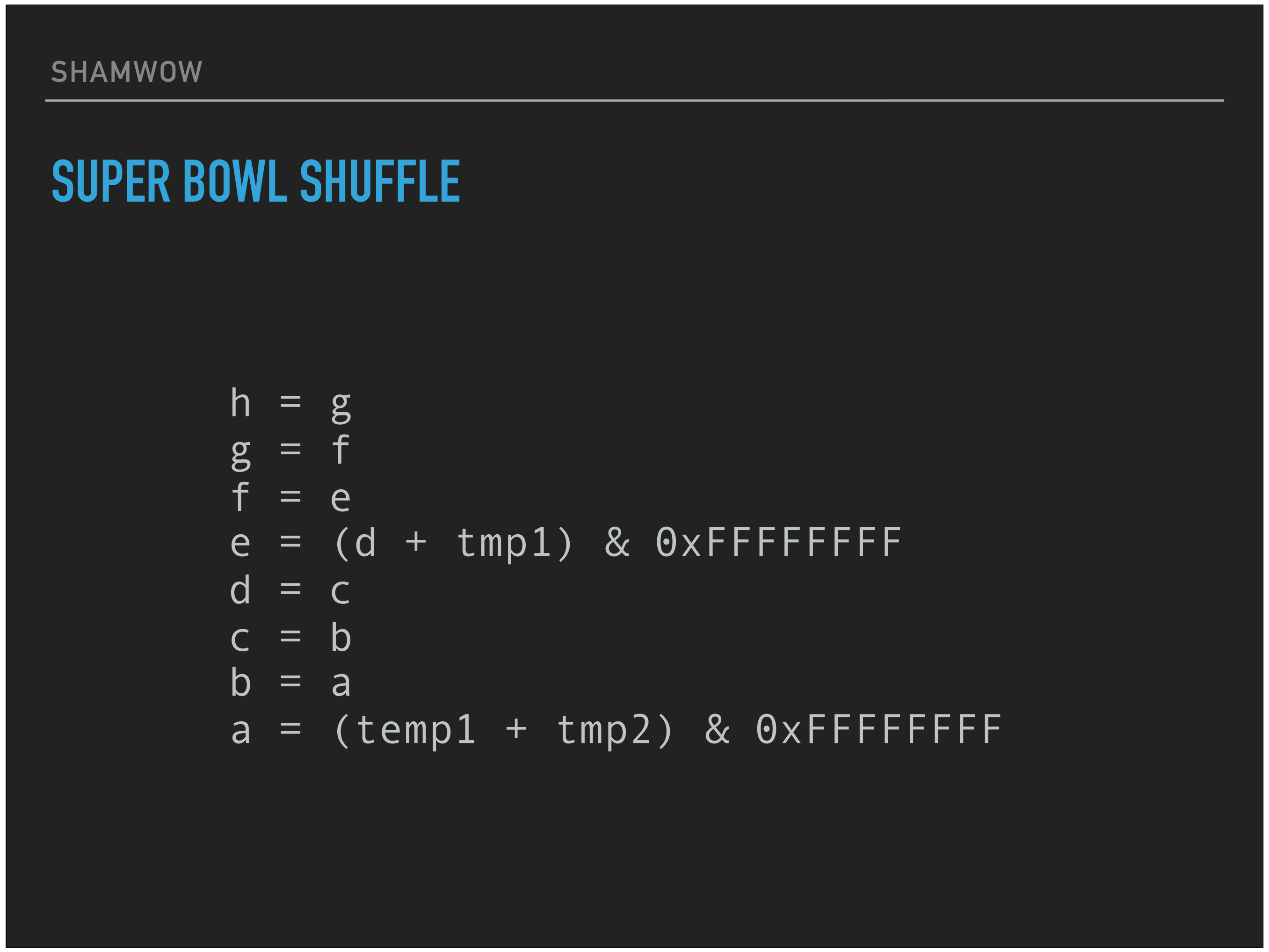
For each “word” in the array, perform the following violence. Remember that K is the BIG list of prime numbers.



LOSSY COMPRESSION IS A VIRTUE

```
s1 = ror(e, 6) ^ ror(e, 11) ^ ror(e, 25)
ch = (e & f) ^ (~(e) & g)
tmp1 = (h + s1 + ch + k[w] + m[w]) & 0xFFFFFFFF

s0 = ror(a, 2) ^ ror(a, 13) ^ ror(a, 22)
maj = (a & b) ^ (a & c) ^ (b & c)
tmp2 = (s0 + maj) & 0xFFFFFFFF
```

In case you've forgotten those what those functions are, here's the ruby for it.



SUPER BOWL SHUFFLE

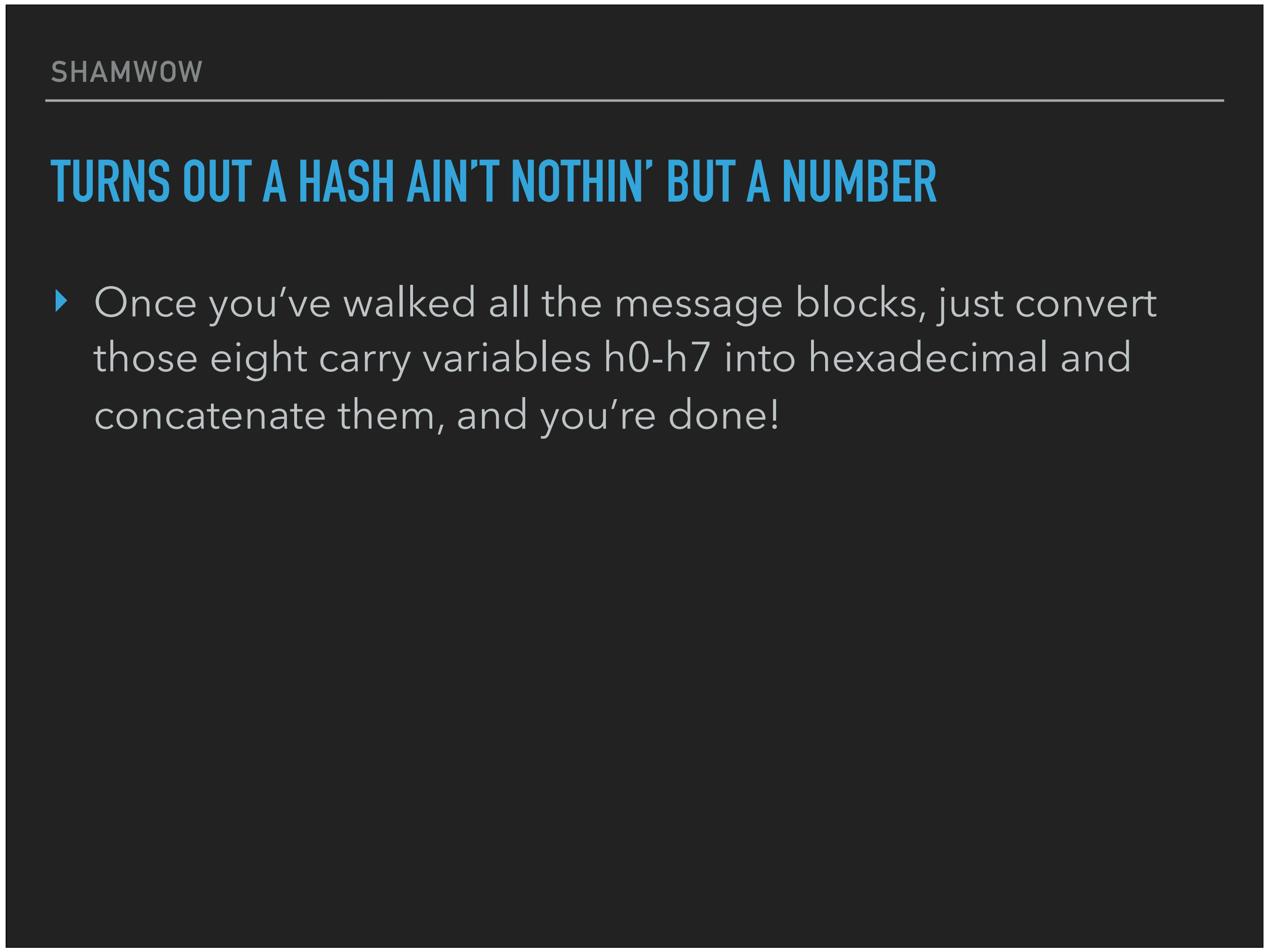
```
h = g
g = f
f = e
e = (d + tmp1) & 0xFFFFFFFF
d = c
c = b
b = a
a = (temp1 + tmp2) & 0xFFFFFFFF
```

This is also in the word loop. We're ratcheting down the list of temporary variables *per word*. Remember, ANDing by FFFFFFFF just truncates to the lower 32 bits after addition.

...AND MERGE IT BACK IN

```
h0 = (h0 + a) & 0xFFFFFFFF
h1 = (h1 + b) & 0xFFFFFFFF
h2 = (h2 + c) & 0xFFFFFFFF
h3 = (h3 + d) & 0xFFFFFFFF
h4 = (h4 + e) & 0xFFFFFFFF
h5 = (h5 + f) & 0xFFFFFFFF
h6 = (h6 + g) & 0xFFFFFFFF
h7 = (h7 + h) & 0xFFFFFFFF
```

Sum and repeat until you run out of message.



TURNS OUT A HASH AIN'T NOTHIN' BUT A NUMBER

- Once you've walked all the message blocks, just convert those eight carry variables h0-h7 into hexadecimal and concatenate them, and you're done!

Then, once we've run out of message, the hash is just those eight variables, concatenated together, all 256 bits of it (voilá, the reason for the name SHA-265), generally represented in hexadecimal for sanity's sake.