Breaking classical crypto with AWS quantum computing

A proof-of-concept on a “toy” hash function

Edition 1 (November 26, 2021)

Author: Christophe PARISEL

Permalink: https://github.com/labyrinthinesecurity/quantumComputing/blob/main/1_Hash.pdf

Here is a simple and complete walkthrough to break a hash function with quantum speedup using AWS Braket quantum circuits.

Hash functions are a key foundation of IT security, so in the wake of Post Quantum Cryptography I think it is of outmost importance that as many IT security professionals as possible become aware of that problem and, more importantly, understand it.

It took me a while to get a proper prototype right until I came across a paper called **Quantum Search for Scaled Hash Function Preimages** [1] published in May 2021. Since the learning curve is steep, my purpose is to show you a very simplified proof-of-concept to showcase the cracking technique and how powerful it is.

Introduction to hashes

A hash function is a many-to-one function from source={0,1}**n to destination={0,1}**m, with m<n. For each binary string from the source set, it finds a corresponding binary string in the destination set.

In cryptography, the source set is the set of all possible clear text messages one might could produce and the destination contains all corresponding hashes. The source set is usually encoded in ASCII or binary, and the destination set is generally encoded in hexadecimal.

For example, the MD5 hash of message "I am a message" encoded in ASCII is 1503e5cd3bd0f717d1096b23de18cacd.

For a hash function to be cryptographically useful, it must have extra properties. Here are the most obvious ones:

- the hashes must be evenly distributed,
- a given hash must correspond to a small number of different messages. All messages yielding the same hash are called "collisions",
- the distribution pattern must pass randomness tests.

Now breaking a hash function means the following: from a given hash, find the corresponding message in less than O(n). If you can't, then the function is considered good enough. If you can, well... You’d better be worried for your encrypted files/emails/you-name-it.

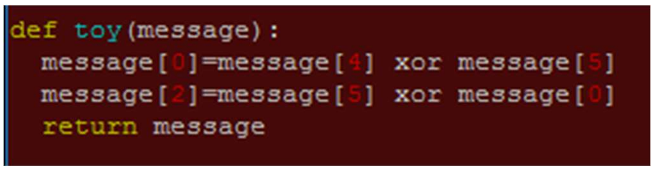
How to break hashes with Quantum Computers

The Quantum world has a technique for doing exactly that, it's called **Grover's algorithm**. This technique can find at least one message associated to a given hash in O(n/2): that's a huge improvement over using a classical computer executing in O(n).

Yet, *many examples of Grover's algorithm are frustrating*: they start by encoding the solution into an Oracle quantum circuit, then run Grover's algorithm to find the solution that we know already. Kind of pointless!

The proposal put forward in the paper I mentioned in introduction is a way to encode the hash function itself into the Oracle, not the solution. No more cheating! The algorithm doesn't know the solution: it only knows the hash and the hash function like any potential adversary.

Encoding a real cryptographic hash like SHA2 into a Quantum Oracle is well beyond reach of present-day quantum computers. It would require tens of thousands of quantum gates, and a register of at least 128 qubits (not counting an astounding number of ancillary qubits). That's why, for the sake of demonstration, we're going to resort to a **toy hash function** operating from source set {0,1}**6 to destination set {0,1}**4 which uses only two quantum XOR gates and 6 qubits:



```
def toy(message):
  message[0]=message[4] xor message[5]
  message[2]=message[5] xor message[0]
  return message
```

This toy function is very likely to have none of the good properties of a good cryptographic hash function, but this is not important: its purpose is only to help understand the technique without too much loss of generality.

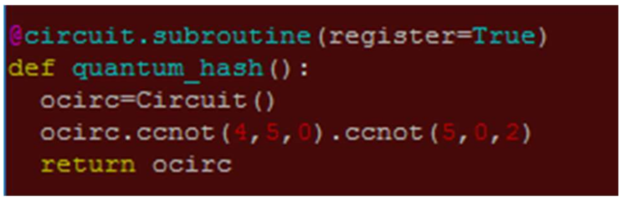
Encoding the hash function in AWS Braket

To encode the search in a quantum Oracle, we follow the paper and proceed in four steps:

1. compute toy() hash with quantum gates
2. mark the quantum states matching a given hash we want to invert
3. uncompute [2] toy() to reset the qubits register
4. amplify the marking with the diffusion operator.

#1: Compute toy function

The AWS Braket representation of our toy() function is quite straightforward:



```
@circuit.subroutine(register=True)
def quantum_hash():
  ocirc=Circuit()
  ocirc.ccnot(4,5,0).ccnot(5,0,2)
  return ocirc
```

To span the classical space of 6 bit long messages, we input 6 qubits (numbered 0 to 5) in superposition state into the quantum hash function. These qubits cover all 2**6=64 possible messages.

The first 4 qubits of the quantum computation yield one of the 2**4=16 possible hashes. The last 2 qubits are fully part of the calculation but are not considered part of the result, so they are just ignored in subsequent steps.

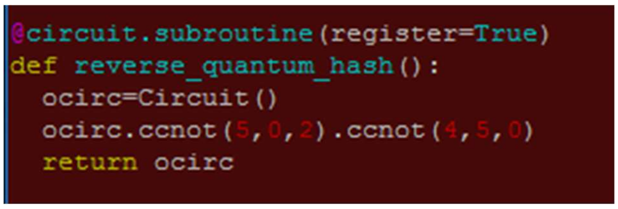
#2: Mark quantum states

Since the hash is made of 4 qubits, we resort to a CCCC-NOT gate to mark the corresponding quantum state. We use an extra ancillary qubit (numbered 6) to "evacuate" the resulting output of the gate, which we don't need, in a reversible way.

#3: Uncompute toy function

A great property of reversible computing is that to uncompute operations, all we must do is... repeat them in reverse order! Say we have an orderly sequence of unitary operations [3] labelled A,B and C. We uncompute them by re-running C, B and A.

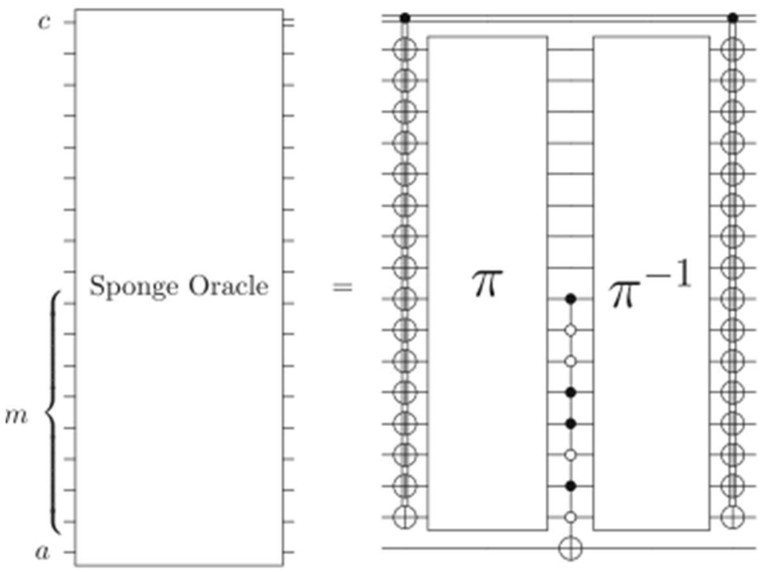
With that in mind, here is the uncomputation of our quantum hash:



```
@circuit.subroutine(register=True)
def reverse_quantum_hash():
  ocirc=Circuit()
  ocirc.ccnot(5,0,2).ccnot(4,5,0)
  return ocirc
```

Incredibly handy and simple, isn't it?

Those first three stages constitute a quantum search Oracle. They are summarized as such in the paper:



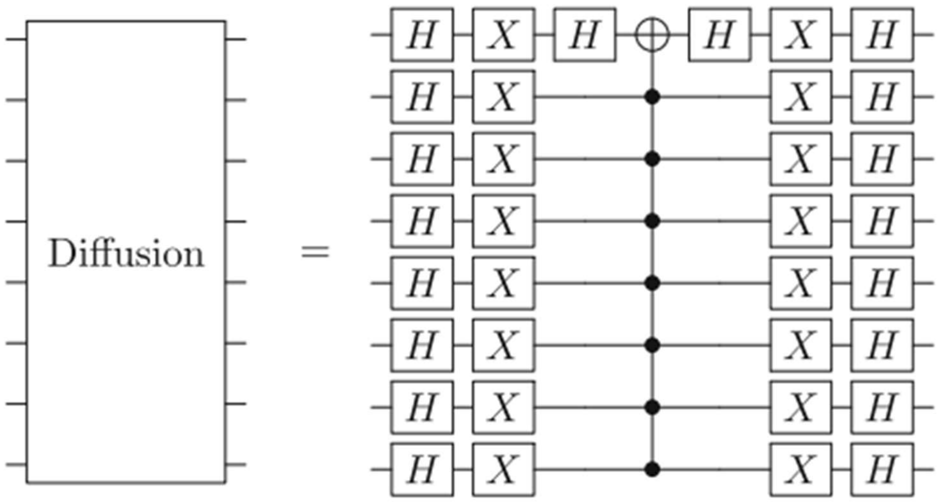
It's already much harder to read! Here is the narrative :

- "m" are the potential messages (in qubits)
- "a" is the ancillary qubit required by C*NOT gates
- PI is the quantum hash function (our step #1)
- The C*NOT gate squeezed between PI, and PI-minus-one is set up to look for hash 10011010 (our step #2)
- PI-minus-one is the uncomputed quantum hash function (our step #3)

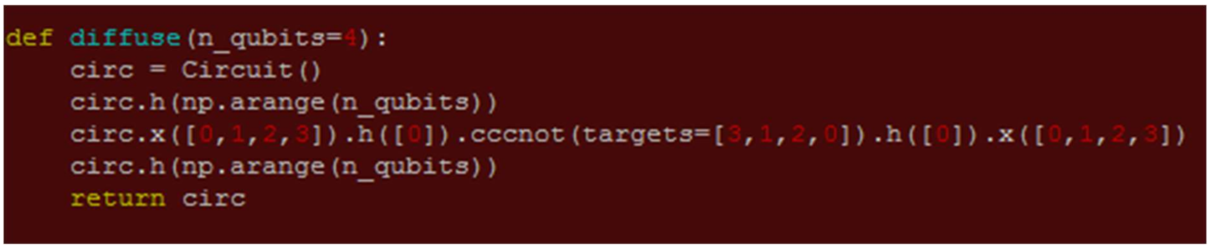
Once the register is back to its original state, the signal is ready to be amplified.

#4: Amplification

We take the same diffusion operator as in the paper:



The only difference is that we need only 4 qubits, not 8. (Because our hashes are 4 bits long).



```
def diffuse(n_qubits=4):
    circ = Circuit()
    circ.h(np.arange(n_qubits))
    circ.x([0,1,2,3]).h([0]).cccnot(targets=[3,1,2,0]).h([0]).x([0,1,2,3])
    circ.h(np.arange(n_qubits))
    return circ
```

The 4 steps can be repeated more than once, but if you repeat too much, it starts to be destructive.

Testing time! So... Is quantum speedup a fraud or a boon?

Since our toy functions operates on 2**6=64 messages, on average a classical computer will reverse the hash after 64/2=32 attempts. The promise of Quantum Computing is to reverse a hash with M collisions in only sqrt(64/M). That's only 8 attempts if M=1, and 4 attempts if M=4.

Let's suppose we are interested to find a message corresponding to hash 1111 (implemented as ccccnot([0,1,2,3,6]) in AWS Braket).

To verify that our quantum search inverts hashes correctly, we need to know which messages match this 1111 hash. The good thing about our toy function is that it is straightforward to enumerate all messages and deduce the hashes (or to infer the colliding messages from a given hash by pure reasoning). It turns out that four collisions yield hash 1111: these are messages 010111,110101,111110 and 111100.

Now let's run our AWS Braket program twice with 2 rounds each time (that's 2*2 = 4 attempts).

We repeat this 1000 to get an average. Here is the JSON output, the best matches are **highlighted in bold**:

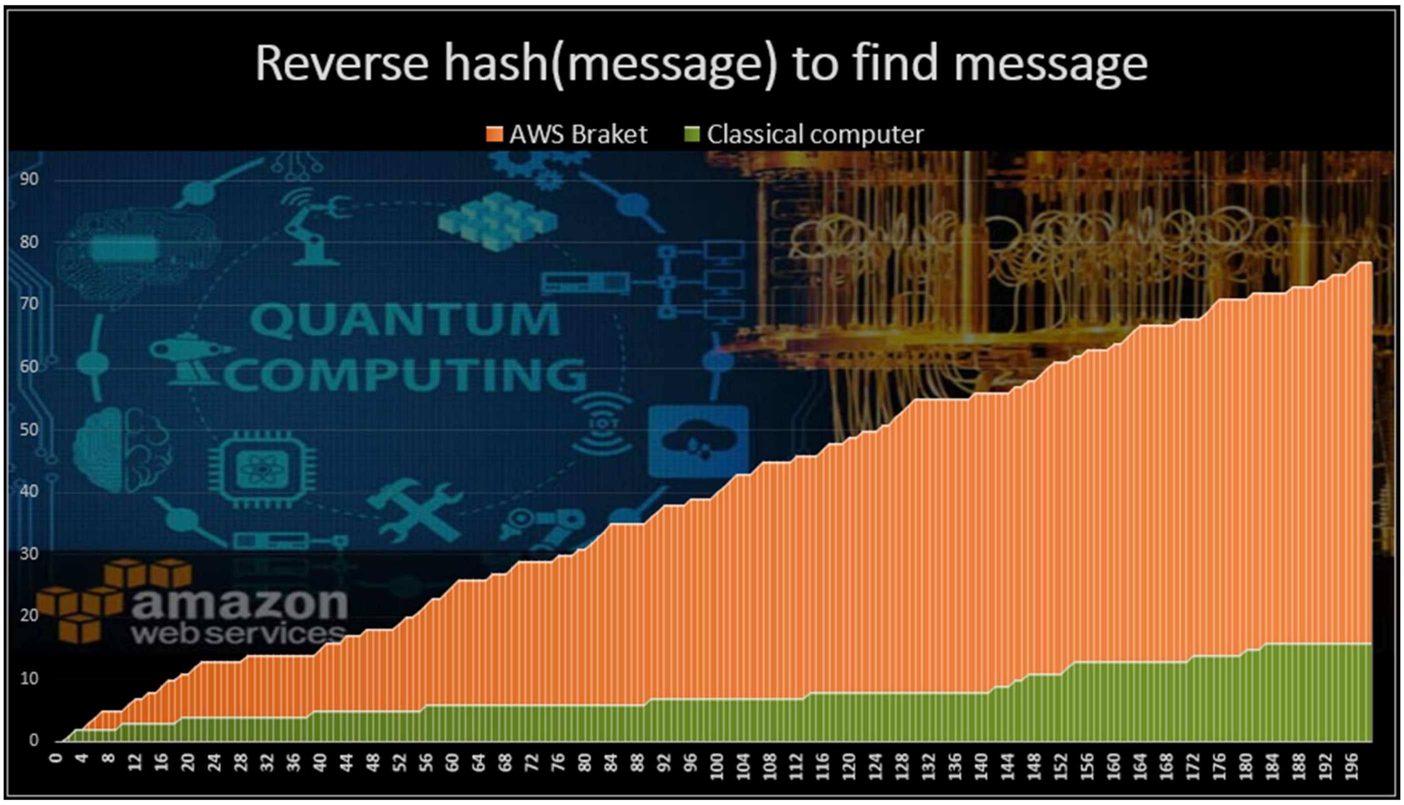
{**'111110': 482**, **'110101': 416**, '110001': 4, '100000': 2, **'010111': 460**, '011111': 6, **'111100': 435**, '101111': 4, '111010': 7, '011101': 6, '111000': 4, '101101': 5, '101110': 4, '000000': 8, '110100': 3, '000100': 1, '100010': 5, '000110': 6, '010100': 4, '100111': 5, '010001': 3, '001100': 4, '111001': 5, '110110': 2, '101011': 3, '010000': 7, '001011': 2, '100011': 5, '011100': 2, '110000': 3, '010010': 1, '011011': 6, '111101': 5, '001110': 4, '101000': 4, '111111': 3, '011000': 3, '110011': 4, '011001': 4, '111011': 3, '110010': 7, '100110': 1, '011110': 4, '100100': 2, '110111': 3, '001000': 2, '001010': 4, '100101': 5, '000101': 5, '100001': 3, '101100': 1, '011010': 3, '001111': 3, '000001': 2, '101001': 2, '000010': 4, '010101': 2, '010011': 3, '000111': 1, '101010': 1, '001101': 2}

What does it all mean?

After 4 attempts, the probability for our quantum circuit to find message 010111 is 46.0%, for message 111110 it is 48.2%, for message 111100 it is 43.5% and for the last one 110101 the probability is 41.6%. All in all, the probability to find at least one of them is about 90%.

For a classical computer, after 4 attempts, the chance to find a given message chosen between the 4 collisions is only about 5%. The probability to find at least one of them is about 23%.

The figure below shows how many times a quantum circuit (in orange) and a classical computer (in green) manage to find message '010111' from hash '1111' in 4 attempts or less. The trials are repeated 200 times, and the results are cumulated from left to right (spanning 0 to 200).



Conclusion

We have an experimental confirmation that Quantum Computing accelerates the inversion of hash functions dramatically so even if quantum computers are still way too small to catch big fishes like SHA-2, we have reasons to worry about the strength of current security primitives based on combinatronics including not only hashes but symmetric encryption algorithms like AES256.

The source code of this prototype is available on my GitHub account [4]. Time for some tinkering?

References

[1] https://arxiv.org/abs/2009.00621

[2] https://en.wikipedia.org/wiki/Uncomputation

[3] https://myentangled.com/en/2019/03/quantum-operations-unitary-and-reversible-matrices/

[4] https://github.com/labyrinthinesecurity/quantumComputing/blob/main/awsHashBreakingProofOfConcept.py