Using Cloud Services to Distribute an Embarrassingly Parallelisable Proof-of-Work Computation

Charlie Harding
University of Bristol
charlie.harding@bris.ac.uk

Contents

1 Introduction **1**

2 Searching for a golden nonce **1**

3 Performing the search remotely **2**

4 Parallelising the remote computation **3**

5 Estimating runtime **4**

6 Conclusion **6**

Appendices **7**

A Setting up AWS **7**
A.1 Setting up the Amazon Machine Image 7
A.2 Setting up the queue 7
A.3 Setting up the local script's permissions 8
A.4 Setting up the remote VMs' permissions 8

1 Introduction

The code for this project can be found at `https://github.com/xsanda/cloud`.

Cryptocurrencies such as Bitcoin rely on a system of proof-of-work to ensure that the *blockchain* (sequence of transactions structured in a Merkle Tree [6]) is not altered maliciously [5]. The idea is that it is computationally expensive to calculate a single entry in the chain, and so it becomes prohibitively expensive to alter history by recomputing a block and all subsequent nodes.

The computationally heavy part of the node calculation is finding the golden nonce: the number n included such that the result of SHA^2-hashing the block with the number n appended has a certain number D of leading zeros in its bit-pattern. Because the bits of a SHA hash are approximately uniformly randomly distributed (though not perfectly [9]), the only way of discovering a golden nonce is by testing vast numbers of possibilities (*nonces*).

This search is *embarrassingly parallelisable*: the different branches of computation can be run in parallel without any need to share any computation data, they only need to be notified when to stop (when another branch has succeeded). This means it can easily be shared by discrete virtual machines, using commercially available cloud services, to share the workload to arrive at a result faster.

In this report, I will look into how to search for a nonce, and then distribute the workload using Amazon's [4] cloud services.

2 Searching for a golden nonce

The simplest way of finding a golden nonce is to iterate through every possible nonce (the integers $0 \leq n < 2^{32}$ [7]), calculate the hash, and stop searching when sufficient leading zeros are found.

The basic code to do this can be seen in Figure 1, implemented in Python. This can be run locally, and for low enough difficulty, this is the fastest way to perform the search, as there is no overhead of starting up a VM: everything is ready to run locally. The time required is approximately

exponential in the difficulty (the number of leading zeroes), as can be seen in Figure 2.

However, the exact time required is not predictable, as it depends on the block given. The variation in runtime is small for a given block and difficulty (Figure 2a), but with more blocks it becomes far larger, though with a more consistent exponential mean (Figure 2b).

This is implemented in the Python program using the command-line arguments `./nonce.py local <block> <difficulty>`, for example `./nonce.py local COMSM0010cloud 24` to find a 24-bit golden nonce for the block `COMSM0010cloud`.

3 Performing the search remotely

To save computation locally, and to prepare for scaling up, we can use a virtual machine in the cloud to perform the computationally heavy cloud operations. For this, we need to create an AWS account, and then perform some setup (see appendix A).

Once the AMI, queue service and authorisation have been set up, it is time to actually run the golden nonce search remotely. The way this is done is by starting up a VM, with the initialisation script set to save the nonce-finding Python script onto the VM and then run it. When creating the VM, we specify the IAM user set up in subsection A.4, which means that we do not need to worry about transferring credentials to the VM ourself.

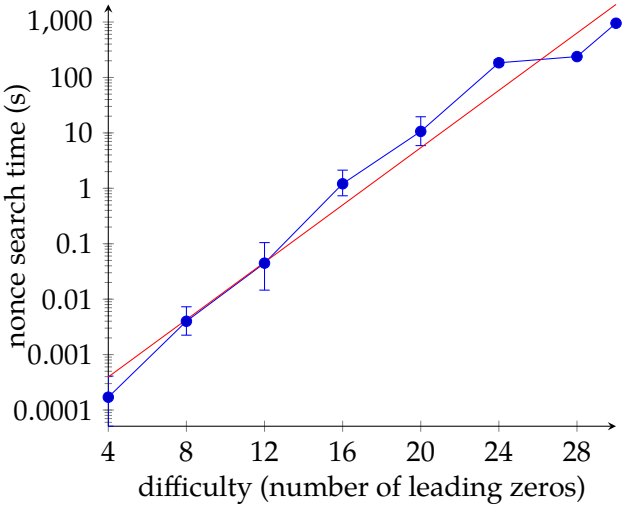
This way, the script can be tweaked locally and the latest version will be copied across remotely automatically. As this script is needed anyway to start the process going, using it to transfer the script to be executed also reduces the amount of network connections needed at the start, so the VM can get under way as quickly as possible.

The VM will call exactly the same function to do the calculation as before (`find_nonce` in Figure 1), but then instead of printing the result it will use AWS's Simple Queue Service to transmit it back to the user. Included in this transfer will also be the timestamp of the start of the search. This is so that any previous items still on the queue (from failed previous runs, or duplicates from SQS' "At-Least-Once Delivery" system [3]) can be ignored.

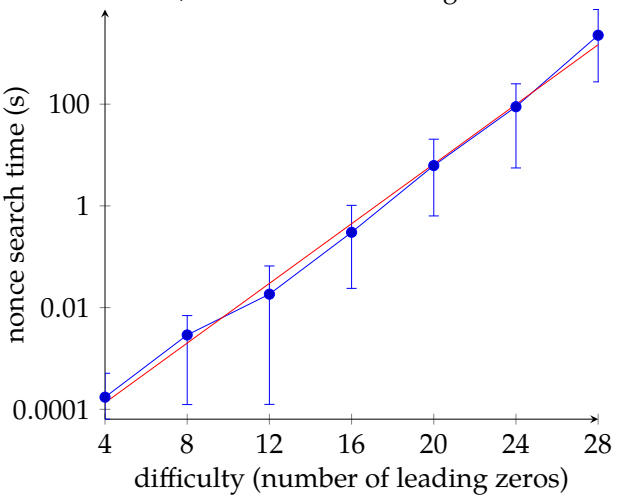
The program running locally, having told the EC2 instance to start, polls the SQS queue for a

```python
from hashlib import sha256
from itertools import count

def find_nonce(block: bytes,
               difficulty: int,
               ) -> (int, int):
    for nonce in range(start, 2**32):
        bytestring = build_nonce(block, nonce)
        hash = sha2(bytestring)
        zeros = leading_zeros(hash)
        if zeros >= difficulty:
            return nonce, zeros
    return (-1, 0)


def build_nonce(
        block: bytes,
        number: int) -> bytes:
    return (block
        + number.to_bytes(4, byteorder='big'))

def sha2(block: bytes) -> bytes:
    return sha(sha(block))

def sha(block: bytes) -> bytes:
    SHA256 = sha256()
    SHA256.update(block)
    return SHA256.digest()

def leading_zeros(hash: bytes) -> int:
    i = 0
    for c in hash.hex():
        if c == '0':
            i += 4
            continue
        elif int(c, 16) < 2: i += 3
        elif int(c, 16) < 4: i += 2
        elif int(c, 16) < 8: i += 1
        break
    return i
```

Figure 1: A loop to find a golden nonce.



(a) Finding a golden nonce for the given block `COMSM0010cloud`, with a line of linear regression in red.



(b) Finding a golden nonce, averaged for 12 different blocks, with a line of linear regression in red.

Figure 2: The time to find a golden nonce locally.

response. When it receives a response (either a successful golden nonce or a report that none was found in the whole search space), it tells the VM to terminate, and reports the result back to the user.

As shown in Figure 3, the runtime of this algorithm is similar for large enough difficulty (at least $D = 24$). This does of course depend on the ratio of speed of the local processor, and it will also be affected by other background tasks that are running on my local processor.

However, the far more significant difference is the startup time of at least 30 seconds before the VM is ready to start computation, shown in green on these graphs. As the local processor is able to find a golden nonce at difficulty $D = 20$ in less than 30 seconds, it is not worth starting a VM at all for such computations.

This is implemented in the Python program using the command-line arguments `./nonce.py master <block> <difficulty> 1`, for example `./nonce.py master COMSM0010cloud 30 1` to find a 30-bit golden nonce. The mode `master` is used because it then calls `slave` mode on each VM it starts up.

4 Parallelising the remote computation

Now that we have the computation running on a remote virtual machine, we can spread the load between multiple virtual machines to speed up the search. This means dividing the search space up, such that each VM scans a roughly equal range of distinct nonces. The simplest way of dividing the space with n VMs numbered $0 \leq i < n$ is for VM i to start at nonce i, and then increment by n each time, until the maximum 32-bit integer ($2^{32} - 1 = 4\,294\,967\,295$) is exceeded. Now every remote node needs to be passed its starting point i and the number of VMs i. Otherwise the operation per VM is unchanged from section 3.

In the local master program, the process is also much the same. One key distinction is the termination condition: if one success is reported then all instances are terminated, but only when all n send failure are they terminated from failure. This is because one VM may exhaust its search space before another has found the golden nonce in its own search space, due to variation in run speed, and to ensure the golden nonce is found

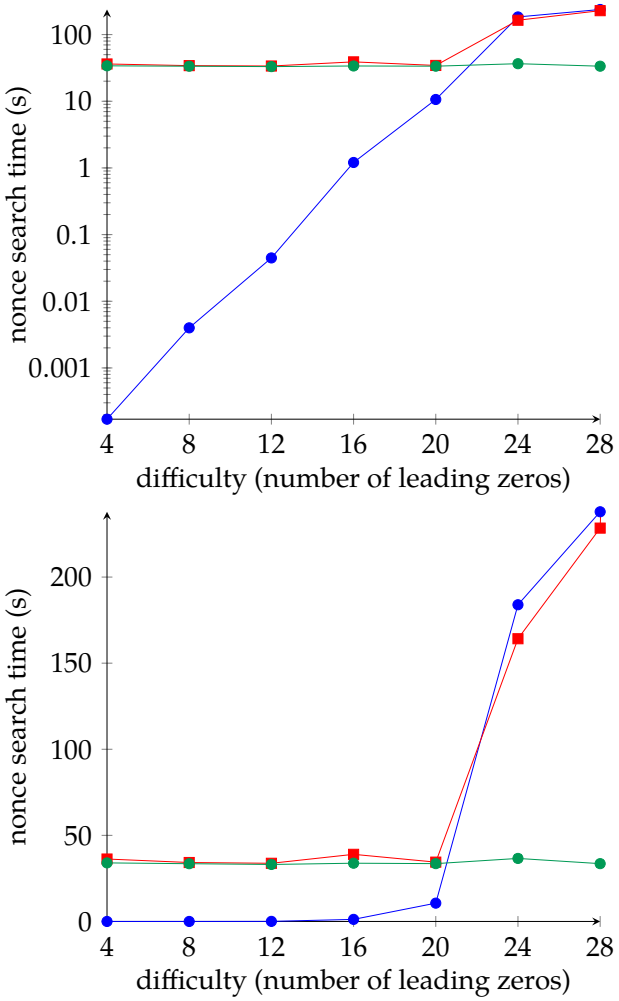


Figure 3: The runtime of nonce calculation on the server (in red) compared to locally (in blue), of which the green part of the server runtime is used for startup, in both logarithmic (top) and linear (bottom) vertical scales.

we must wait for all VMs to report failure before failing.

The fixed startup overhead is similar to that with one VM, but as Figure 4 on page 5 shows, once the VMs are started the search is significantly faster.

For small values of D, the runtime for the parallelised golden nonce search increases linearly with n. This is because the computation is such a small part of the overall runtime, and the iteration to create the VMs (typically around 2 seconds per VM) increases directly proportionally to the number of VMs. This can be seen in Figure 5 with $D = 8$.

However, once D becomes sufficiently large, for example 24, the trend reverses for lower numbers of VMs, with the runtime increasing with fewer VMs. This is because the saved computation time becomes greater than the additional cost of startup. This can be seen in Figure 6, and is even more pronounced with larger values of D still.

This is implemented in the Python program using the command-line arguments `./nonce.py master <block> <difficulty> <VMs>`, for example `./nonce.py master COMSM0010cloud 30 8` to find a 30-bit golden nonce using 8 VMs.

5 Estimating runtime

In order to estimate how long it will take to find the golden nonce, we need to estimate how many nonces need to be searched. Because the bits of a SHA hash are approximately uniformly randomly distributed (though not perfectly [9]), the probability of any given nonce producing a hash with a given binary digit being zero is 0.5.

Because the digits in a hash can be assumed to be independent, the probability of D given digits, for example the first D digits, all being zero is simply the Dth power of one half, 0.5^D. Each nonce we test produces an independently distributed hash, and we would like to work out the number of trials needed to find a success, which occurs with probability 0.5^D.

This means we can use the geometric distribution [8] to model the number of nonces to search, $X \sim Geo\left(0.5^D\right)$. The expected number to search $E(X)$ is $1/0.5^D = 2^D$ nonces. If we would like to find a golden nonce with a probability of p, then we

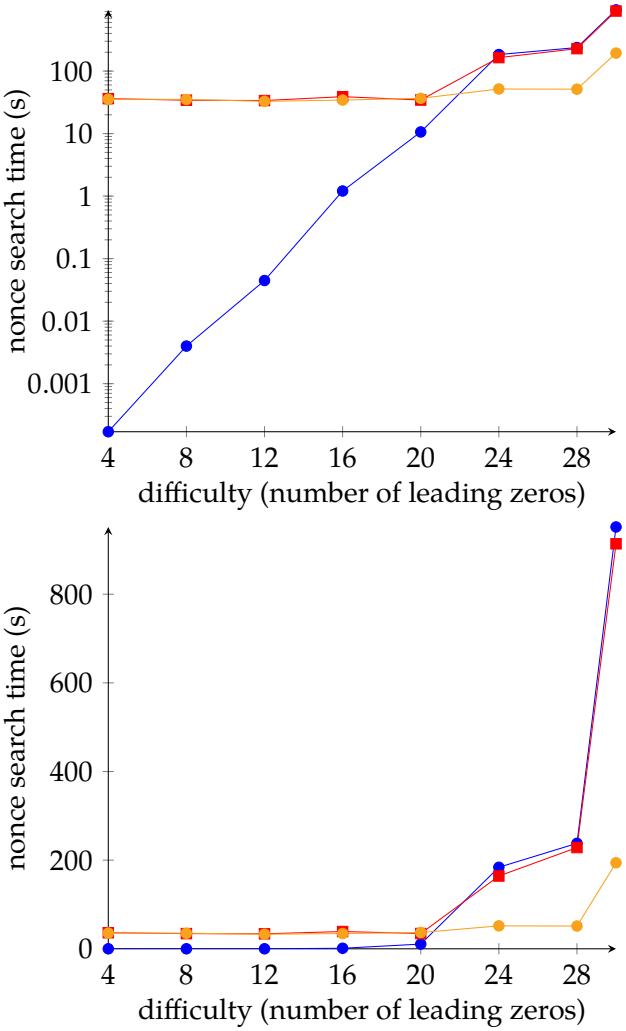


Figure 4: The runtime of nonce calculation with 10 VMs (in orange) compared to one VM (red) and locally (in blue), in both logarithmic (top) and linear (bottom) vertical scales.

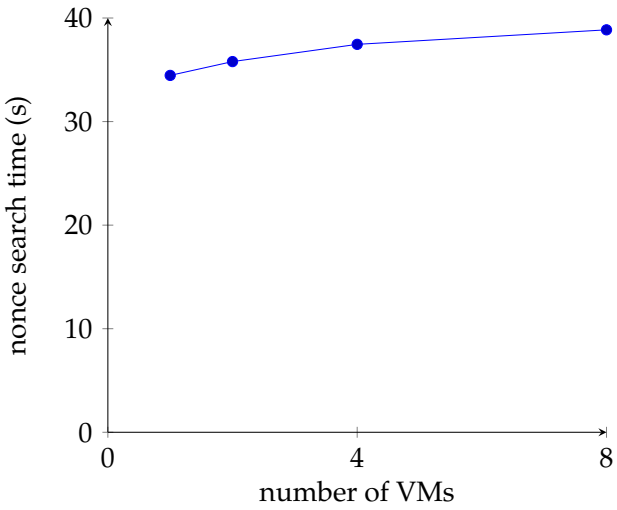


Figure 5: The runtime of nonce calculation with a difficulty of 8.

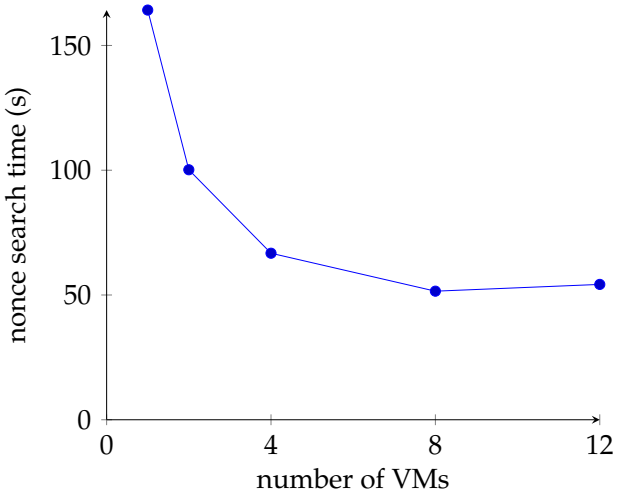


Figure 6: The runtime of nonce calculation with a difficulty of 24.

need to search x nonces, such that $P(X \leq x) \geq p$.

$$\begin{aligned} P(X \leq x) &\geq p \\ 1 - (1 - 0.5^D)^x &\geq p \\ (1 - 0.5^D)^x &\leq 1 - p \\ x \log(1 - 0.5^D) &\leq \log(1 - p) \\ x &\geq {}^{\log(1-p)}/_{\log(1-0.5^D)} \end{aligned}$$

However, if we want to estimate runtime, then we do not want to simply find the number of hashes we need to test to find a golden nonce: there does not always exist any golden nonce within the 32-bit integer space from which we are taking our nonces. This means that to get a result, either success or failure, with p probability, it is enough to simply search $x = \min\left({}^{\log(1-p)}/_{\log(1-0.5^D)}, 2^{32}\right)$ nonces.

For example, if we wish to find a 30-bit golden nonce with 90% probability, it is enough to search ${}^{\log(1-p)}/_{\log(1-0.5^D)} = {}^{\log 0.1}/_{\log(1-0.5^{30})} \approx 2\,472\,381\,917$ possibilities.

However, if we want to find a nonce 32-bit golden nonce with 90% probability, we would need to search ${}^{\log 0.1}/_{\log(1-0.5^{32})} \approx 9\,889\,527\,670$ possibilities, more than 2^{32}, so we can get a result (possibly failure) after searching just 2^{32} nonces.

From our experiments, we found that 165 000 nonces can be hashed and checked per second. This means that the entire space can be searched in ${}^{2^{32}}/_{165\,000} = 26\,000$ seconds, or 7 hours. It then takes 2 seconds per VM to request that EC2 creates the instances, plus 30 seconds of loading time for the VMs in parallel. Therefore, in order to determine whether there is a D-bit golden nonce with confidence p in time t, we need to test

$$x = \min\left(\frac{\log(1-p)}{\log(1-0.5^D)}, 2^{32}\right)$$

nonces, taking

$$\frac{x}{165\,000}$$

seconds of computation. This means the total runtime in seconds is

$$\frac{x}{165\,000n} + 30 + 2n$$

when running across n VMs.

$$\begin{aligned} \frac{x}{165\,000n} + 30 + 2n &\leq t \\ \frac{x}{165\,000n} &\leq t - 2n - 30 \\ \frac{x}{165\,000} &\leq n(t - 2n - 30) \\ 2n^2 - n(t-30) + \frac{x}{165\,000} &\leq 0 \end{aligned}$$

This yields a lower bound for the number of machines as

$$n \geq \frac{t - 30 - \sqrt{(t-30)^2 - 8\frac{x}{165\,000}}}{4}$$

and an upper bound of

$$n \leq \frac{t - 30 + \sqrt{(t-30)^2 - 8\frac{x}{165\,000}}}{4}$$

using the quadratic formula, where $x = \min\left({}^{\log(1-p)}/_{\log(1-0.5^D)}, 2^{32}\right)$. When these overlap, and so there are no possible (integer) values for n, the computation is not possible in the time given.

Note that, as p tends to 100%, ${}^{\log(1-p)}/_{\log(1-0.5^D)}$ tends to positive infinity, so we set x is set to 2^{32} when p is provided as 100%.

This is implemented in the Python program using the command-line arguments `./nonce.py auto <block> <difficulty> <time-seconds> <percentage-confidence>`, for example `./nonce.py auto COMSM0010cloud 30 600 90` to find a 30-bit golden nonce in 600 seconds (10 minutes) with 90% confidence.

6 Conclusion

Cloud computing provides a way to easily spread computation out over many different compute nodes without a great deal of initial investment. The result of this is that it can be used to perform calculations significantly faster than would be possible locally, but more importantly it is possible to build applications that can scale as needed by using more cloud resources when demand increases. In my own experiments, I found that the speed per computation on a VM is comparable to that of a local computer, plus some start-up overhead, but the performance gains really come when scaling.

If I were to build my own system using cloud infrastructure, I would probably look into keeping the VMs running between requests, only shutting them down after the request volume has substantially dropped, and only starting up when it starts to climb again, to save the time of the VMs constantly starting up and terminating.

References

[1] Amazon Machine Images (AMI). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html.

[2] Amazon Simple Queue Service. https://aws.amazon.com/sqs/.

[3] Amazon SQS standard queues. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/standard-queues.html.

[4] Amazon Web Services. https://aws.amazon.com/.

[5] Blockchain guide - Bitcoin. https://bitcoin.org/en/blockchain-guide#proof-of-work.

[6] What is a Merkle tree? Beginner's guide to this blockchain component. https://blockonomi.com/merkle-tree/, Jul 2018.

[7] Cliff, D. Horizontal scaling for an embarrassingly parallel task: Blockchain proof-of-workin the cloud, Oct 2019.

[8] Leemis, L. Geometric distribution. http://www.math.wm.edu/~leemis/chart/UDR/PDFs/Geometric.pdf.

[9] Leurent, G., and Nguyen, P. Q. How risky is the random-oracle model? Cryptology ePrint Archive, Report 2008/441, 2008. https://eprint.iacr.org/2008/441.

Appendices

A Setting up AWS

Once an AWS account has been created, we need to perform some setup.

A.1 Setting up the Amazon Machine Image

Firstly, we can either use the public Amazon Machine Image (AMI [1]) that I created, or create our own. A custom AMI is used so that the correct version of Python and its package `boto3` are preinstalled, saving time every time a VM starts up.

The one already created exists in the region `us-east-2`, and has the ID `ami-0ad788d4ae566815b`, but can easily be replicated as follows:

1. Launch a new EC2 instance using Amazon Linux, for example *Amazon Linux AMI 2018.03.0 (HVM), SSD Volume Type*.
2. Choose *t2.micro* as the instance type.
3. Create a key pair as necessary to allow for shell access into the VM.
4. Connect by SSH to the VM, and run the following commands:

```
sudo -s
yum install -y python36
python3 -m pip install --upgrade pip
python3 -m pip install boto3 botocore
```

5. Create an image from the running instance, using the AWS console website.
6. Once the image shows as 'available' on the AMI page of the AWS console, make a note of the AMI ID.
7. Terminate the running instance.
8. Replace the value of the constant `AMI` in `nonce.py` (line 13) with the newly created AMI instance.

A.2 Setting up the queue

In order for the VMs to return their status (success or search space exhaustion), a queue is required. For this, we can use Amazon's Simple Queue Service (SQS [2]).

1. Open the Simple Queue Service in the AWS Console.

2. Click to create a new queue in the same region as your AMI (`us-east-2` for the provided AMI).
3. Set the name for the queue, for example "`NonceOutput`".
4. Set the queue type to Standard.
5. Create the queue without further configuration.
6. Replace the value of the constant `QUEUE_NAME` in `nonce.py` (line 15) with the name chosen for this queue, if a different name is chosen to "`NonceOutput`".

A.3 Setting up the local script's permissions

In order for the local script to start up EC2 instances and to read from the queue, a new IAM (Identity and Access Management) user is required.

1. Open the Users tab of the IAM service in the AWS Console.
2. Click to create a new user.
3. Assign them a username, for example "`nonce`".
4. Attach the following permissions directly to them:
   - `AmazonEC2FullAccess`
   - `AmazonSQSFullAccess`
   - `IAMFullAccess`
5. No tags are needed for this user.
6. Create the user.
7. Store the access key in your local computer user's AWS config file (`~/.aws/config`), by using the following template and replacing the `AAA` and `aaa` with the *Access key ID* and *Secret access key* presented in the AWS Console.

```
[default]
aws_access_key_id=AAA
aws_secret_access_key=aaa
```

A.4 Setting up the remote VMs' permissions

In order for the remote script to push to the queue, a new IAM (Identity and Access Management) role is required.

1. Open the Roles tab of the IAM service in the AWS Console.
2. Click to create a new role.
3. Select EC2 as the service that will use this role.
4. Attach the following permissions directly to them:
   - `AmazonSQSFullAccess`
5. No tags are needed for this role.
6. Assign the role a name, for example "`NonceQueueRole`".
7. Create the role.
8. In the role's summary page, copy the *Instance Profile ARN*, and store it in the constant `IAM` in `nonce.py` (line 17).