COMSM0010 Cloud Computing
Cloud Nonce Discovery Programming Project Report

Finn Hobson - `fh16413`
Candidate Number - 35645

December 2019

The source code and deployment instructions for this project can be found using the link:
`https://github.com/finnhobson/Cloud-Nonce-Discovery-System`

1 Introduction

This report describes my Cloud Nonce Discovery (CND) system, explains the development process and justifies some of the implementation decisions that have been made. Section 4 of this report discusses the performance analysis of my system, showing how run-times vary as the user-specified parameters are altered.

As described in the coursework brief, the purpose of this CND system is to perform a brute force search for a golden nonce and report this nonce value back to the user. A golden nonce is a value, that when appended to some data block, produces a SHA-256[2] hash result that has at least a specific number, known as the difficulty level, of leading zeros. Finding these values is an important step in the Proof of Work system used by Blockchain. This brute force search is an embarrassingly parallel task, meaning it can be horizontally scaled across virtual machines running in the cloud to complete in a faster time.

I have chosen to write the code for my CND system in Python 3 and use Amazon Web Services (AWS) for cloud provision. The Boto 3 framework has been used as a Python interface to AWS.

2 Nonce Discovery Program

To begin building my CND system, I created a Python script that could perform a brute force search for a golden nonce value entirely on a local machine. This nonce discovery program requires the user to specify some parameters such as a difficulty level. As this code would eventually be executed remotely on virtual machines running in the cloud, it would not have been suitable for the program to take live user input using the `input()` Python function. Therefore, I decided that using command line arguments would the best way to specify parameters to the program.

Once the program has processed the input parameters, steps are taken to find a golden nonce value. These steps can be seen in the `proof_of_work()` function within the associated `cnd_worker.py` script:

1. The "target" value required for a golden nonce to be found is determined based on the user-specified difficulty level.

2. The previous "block" of data (the string "COMSM0010cloud" in this case) is converted into a binary string via ASCII encoding.

3. A 32-bit integer nonce value is determined by a `for` loop. This nonce value is converted into a binary string and appended to the data block binary string.

4. This block-and-nonce binary string is hashed using the SHA-256 cryptographic algorithm. I have used the `hashlib` Python library as this includes a trusted implementation of the SHA-256 algorithm. A 256-bit hash result is produced.
5. As Blockchain calls SHA-256^2 on the input block for Proof of Work, this hash result is hashed once again using SHA-256. This produces a 256-bit result.
6. This SHA-256^2 result is interpreted as a 256-bit integer and compared to the target value. If the result is less than the target value then a golden nonce has been found. This golden nonce value is printed along with the time taken for the search to complete and some other useful information.
7. If the hash result is not less than the target value, then steps 3 to 6 are repeated until a golden nonce is found.

This program successfully finds golden nonce values when it is run on a local machine. Analysis of how the run-time varies as the difficulty level increases can be seen in Section 4 of this report.

3 Cloud Nonce Discovery System

Once golden nonce values could be found on a local machine, I began to investigate how this program could instead be run on a virtual machine in the cloud and then how the workload could be divided amongst multiple virtual machines running in the cloud to improve discovery times.

After researching similar horizontal scaling services running in the cloud such as Mitch Garnaat's Monster Muck Mashup[1], I found that AWS would be a very suitable cloud provider to use to build my CND system. I could create AWS Elastic Compute Cloud (EC2) instances to run the nonce discovery Python script, store output information in Simple Storage Service (S3) buckets and potentially use Simple Queue Service (SQS) for communication between the local machine and the cloud instances. I chose to set up a full AWS account rather than an AWS Educate Starter Account because a full account offers more flexibility, and resources would not be lost if my account runs out of credit.

3.1 Creating the EC2 Instances

In order to write a Python script that could create and manage EC2 instances, I installed AWS Command Line Interface (CLI) and the Boto 3 Python library. For CLI commands and Boto 3 functions to be able to access AWS services, an AWS user profile had to be created with the necessary permissions policies. On the Identity and Access Management (IAM) console in AWS, I created a new user with programmatic access and added the `AmazonEC2FullAccess` permissions policy since I knew that my CND setup script would need to be able to create and manage EC2 instances. Access credentials were generated for this new user, which I entered using the `aws configure` command in a terminal.

Now that command line programs could programmatically access AWS services, I began writing a Python script that would create an EC2 instance and run the nonce discovery program. This could be done using the Boto 3 function `resource('ec2').create_instances()`. This function requires a list of parameters that specify the attributes of the created instances such as the Amazon Machine Image (AMI), type, size, amount, tags, IAM roles and more.

Initially, I created instances using the standard, unmodified Amazon Linux AMI and specified the `t2.micro` type. My nonce discovery program does not require multiple CPUs on the same instance or a high amount of memory, so this free tier instance type is suitable for my CND system and does not cost anything to use.

So that I could SSH into the created EC2 instances during the development process, I needed to create an EC2 keypair. To do this, I wrote a Python script that generated a keypair using the Boto 3 function `resource('ec2').create_key_pair()` and then stored the private key in a local file.

At this point, my CND setup script could create a number of EC2 instances and I could see the instances appear correctly on the AWS console. I could then SSH into these instances to install Python 3, download the nonce discovery Python script and execute the script. However, these commands needed to be executed automatically when an instance is created. Conveniently, the `UserData` parameter of the `create_instances()` function allows an initialisation script to be passed to the instances, which executes when the instance are launched. This allowed the instances to automatically install Python 3, download the nonce discovery script and execute the script. Since the script was being executed remotely and the terminal of the EC2 instance was not visible while the script was being executed, I redirected the terminal output to a file. This meant that if I used SSH to access an instance after the nonce discovery script had executed, an output file would be present containing the golden nonce output information.

Installing Python 3 and downloading the nonce discovery script when an instance launched was adding to the total time taken for the program to execute. To reduce this time, I decided to create a customised AMI that had Python 3 already installed and the nonce discovery script already downloaded. To do this, I created an EC2 instance on the AWS console, accessed the instance using SSH, performed all of the necessary setup commands and then used the AWS console to create an image from this instance. I swapped the `ImageID` parameter in the `create_instances()` function from the ID of the unmodified Amazon Linux Image to the ID of this customised image. This meant that the created instances would no longer need to run commands to setup the execution environment on launch, leaving only the command to execute the nonce discovery script.

3.2 Outputting the Golden Nonce Result

Using SSH to access the created EC2 instances to check for an output file is a very inefficient method of finding if a golden nonce result has been found. Therefore, I needed to find a method to automatically send the result back to the local machine once it has been found so that it can be printed on the terminal where the CND setup script was executed. I researched two potential methods that could be used to send the golden nonce result from an EC2 instance to the local machine - using SQS and using S3.

The SQS method would involve using Boto 3 to create a queue that could be accessed by the EC2 instances and the local machine. When a golden nonce result is found by an EC2 instance, a message containing the output data could be placed on the queue. The CND setup script running on the local machine could repeatedly poll the queue until a message appears, and then print this result to the terminal. However, one problem with this implementation is if the terminal running the CND setup script is closed or times out, then it may not be possible to retrieve the golden nonce result. Also, configuring SQS is relatively complicated and there are restrictions such as the inability to give a queue the same name as a previously deleted queue for 60 seconds after deletion.

The S3 method would involve using an AWS CLI command to copy the generated output file into an S3 bucket once the nonce discovery script has found a result and finished executing on an EC2 instance. After the EC2 instances have been created, the CND setup script running on the local machine could repeatedly search the S3 bucket until the expected output file appears, and then print this result to the terminal. In addition to this, the output file could be accessed at any time directly from the S3 bucket, even if the CND setup script is stopped or the terminal times out. This also allows the CND setup script to be run multiple times with differently named output files, so a batch of golden nonce results could be produced simultaneously and stored in the S3 bucket.

After considering the advantages and disadvantages of these two methods, I decided to use the S3 method for my CND system. I created an S3 bucket using the S3 management console, and made sure that public access was blocked so only my AWS account could access the bucket and its contents. Although security is not a top priority for this CND system, some serious data leaks have occurred because S3 buckets were given the incorrect permissions or were left publicly accessible. To allow the created EC2 instances to upload files to the S3 bucket, I used the IAM console to create a new role with the `AmazonS3FullAccess` permissions policy and I passed this role as a parameter to the `create_instances()` function.

The command used to copy the generated golden nonce output file into the S3 bucket can be seen in the instance initialisation script (`init_script`) that is created within the `cnd_setup.py` script. Once this file is uploaded to the S3 bucket, the CND setup script finds the file, prints its contents and displays the total time taken for the golden nonce value to be discovered.

3.3 Dividing and Assigning the Workload

Now that my CND system could run the nonce discovery script on a single EC2 instance and output the result, I could try to improve the discovery times by horizontally scaling the system across multiple EC2 instances. A golden nonce can be found faster if the nonce values are tried at a faster rate, so each created EC2 instance should try a different portion of the potential nonce values simultaneously. If a golden nonce can be found in C seconds on a single instance, the aim is for the search to complete in C/N seconds when divided between N instances. The initial nonce discovery script tried all of the nonce values from 0 to 2^{32}, so this script needed to be modified to allow each instance to try a distinct set of nonce values.

Initially, the CND setup script allowed users to input the number of instances, N, that they wished to split the workload between. For each EC2 instance to know what portion of the potential nonce values to try, the nonce discovery script would need to know the unique `worker_id` of the instance and how many instances the task is being divided between. These details could have been sent from the local machine to the EC2 instances using SQS, but instead I chose to create a unique initialisation script for each instance within the CND setup script, and include these variables as command line arguments to the nonce discovery script.

Using the `worker_id` and `number_of_workers` parameters, I tried two different methods of partitioning the potential nonce values. The first method involved dividing the entire range of 32-bit positive integers between the number of instances, N, and giving each instance a different portion. For example, if there are two workers, the first worker would try the nonce values from 0 to 2147483648 and the second worker would try the nonce values from 2147483649 to 4294967296. However, this method was unsuccessful in reducing the local search time to the desired C/N level because if the first worker finds a golden nonce value before the other workers, then the search will take the same amount of time to complete as a search running on a single instance.

The other method involves counting up in multiples of N, each worker adding their `worker_id` to the multiple and trying this value as a nonce. For example, if there are two workers, the first worker would try the nonce values 0, 2, 4 and so on, and the second worker would try the nonce values 1, 3, 5 and so on. This method is successful in reducing the local search time to approximately the desired C/N level so it is used by my CND system to divide the workload between the EC2 instances. The successful impact of this workload division method can be seen in Section 4.3 of this report.

3.4 Terminating the EC2 Instances

At this point, my CND system could be horizontally scaled across any number of EC2 instances to successfully improve discovery times. However, the instances would continue to run after a golden nonce result has been found and overwrite the output file in the S3 output bucket. To prevent this from happening, the EC2 instances need to be terminated as soon as a golden nonce result is found.

AWS Lambda is a serverless computing platform that allows code to automatically execute in response to specific trigger events. I decided to set up a Lambda function that could automatically terminate the created EC2 instances when an output file is produced.

This Lambda function uses Boto 3 to retrieve the EC2 instances running on my AWS account and terminate them. So that only the instances running the nonce discovery program are terminated, the CND setup script adds a "CNDAutoOff" tag to the instances that are created. The Lambda function filters the instances using this tag and terminates the instances in the filtered list.

S3 events such as object creation and object deletion can be used to trigger Lambda functions. My CND system terminates the running EC2 instances by triggering the Lambda function when an output file is placed into the S3 output bucket (a PUT event).

Lambda functions can also be invoked at any time using Boto 3. This allowed me to add a scram functionality to my CND system. When the user initiates a scram, the termination Lambda function is invoked and the EC2 instances are terminated.

3.5 System Architecture Diagram

All of the elements discussed so far in this report constitute my horizontal scaling CND system. The AWS architecture diagram of my CND system can be seen in Figure 1. This diagram shows the following steps, which are taken to find and display a golden nonce value:

1. The user runs the CND setup script on a local machine and enters input parameters such as a difficulty level and a number of EC2 instances to split the task between.
2. The CND setup script launches the desired number of EC2 instances each with a unique initialisation script.
3. When each EC2 instance launches, the initialisation script is run, which executes the nonce discovery script with unique parameters. Each instance performs a brute force search for a golden nonce value, trying a distinct portion of the potential nonce values.
4. When a golden nonce is found, the nonce discovery script prints the golden nonce information into an output file. This output file is copied into an S3 bucket.
5. A Lambda function is triggered when a file is put into the S3 output bucket. This function terminates all of the created EC2 instances.
6. The CND setup script repeatedly searches the S3 output bucket. When the expected output file appears, the contents are displayed to the user along with the total time taken for the result to be returned.

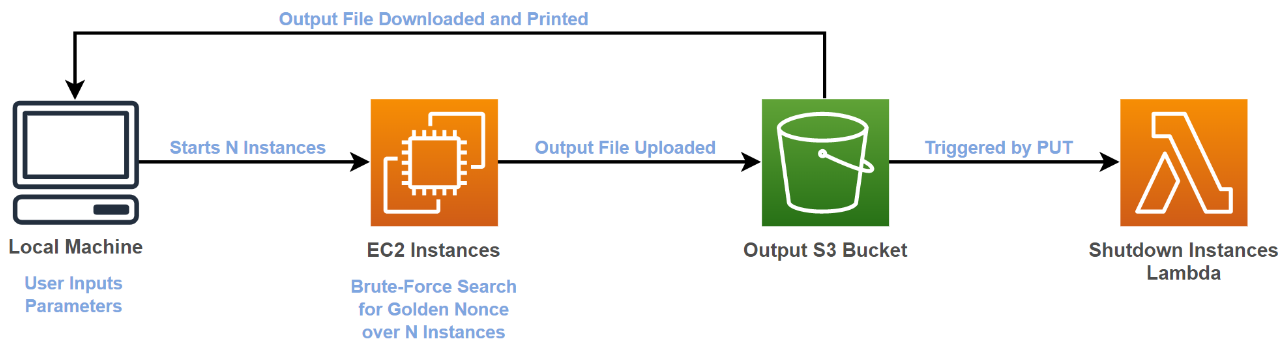


Figure 1: AWS Architecture Diagram of my Cloud Nonce Discovery System

4 Performance Analysis

As I developed my CND system, I ran performance tests to investigate how the time taken for a golden nonce to be found is affected by the user-specified parameters. This section presents and analyses the results of these performance tests.

4.1 Local Machine Performance

Figure 2 shows the change in time taken for a golden nonce to be found on a local machine as the difficulty level increases. These times were produced by executing the nonce discovery script a number of times for each difficulty level on my local machine and averaging the resulting times for each difficulty level. The graph shows that the discovery time is very small and increases slowly up to a difficulty level of about 18. Increasing the difficulty level further than this causes the discovery time to rapidly increase and become very large. This is similar to the behaviour of an exponential function.

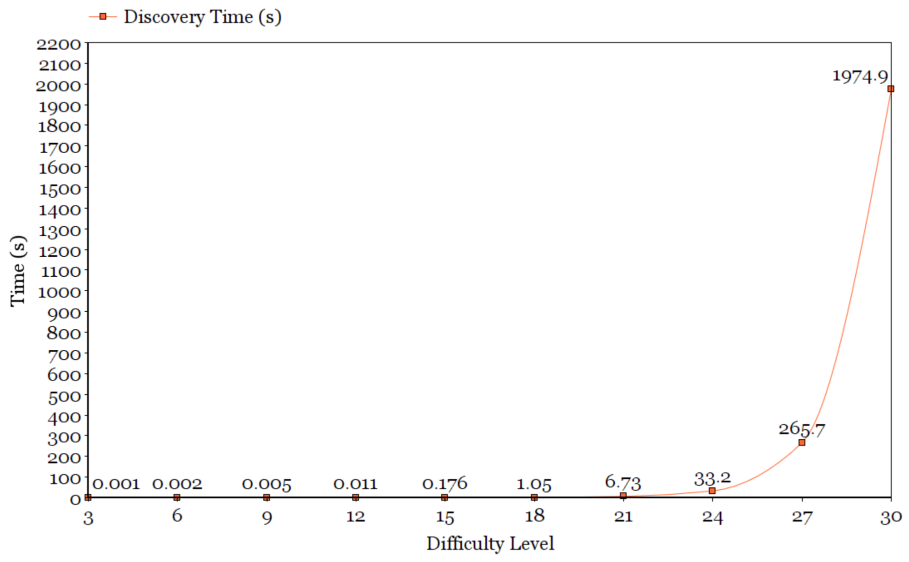


Figure 2: Line graph showing how discovery time changes as the difficulty level increases when running on a local machine.

4.2 Single EC2 Instance Performance

Figure 3 shows the change in time taken for a golden nonce to be found on a single EC2 instance as the difficulty level increases. This graph shows that there is approximately a 30 to 40 second start-up time needed for an EC2 instance to be launched. When the time taken for the EC2 instance to find a golden nonce (local discovery time) is almost zero, the total discovery time is still over 30 seconds. This suggests that it is not worth running the nonce discovery script in the cloud if the start-up time is more significant than the local discovery time. The local discovery time begins to outweigh the start-up time at a difficulty level of approximately 24, so it can be worth running the nonce discovery script in the cloud at this difficulty level and above.

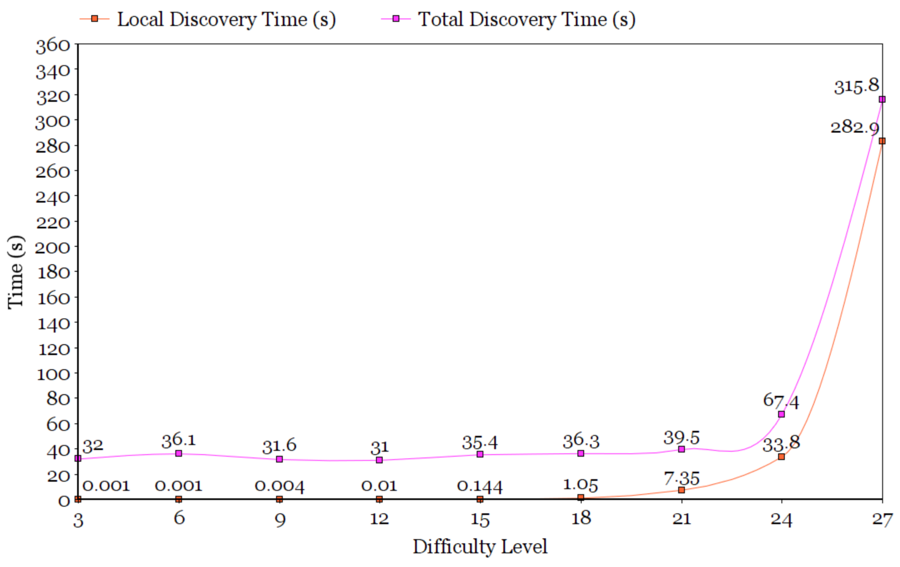
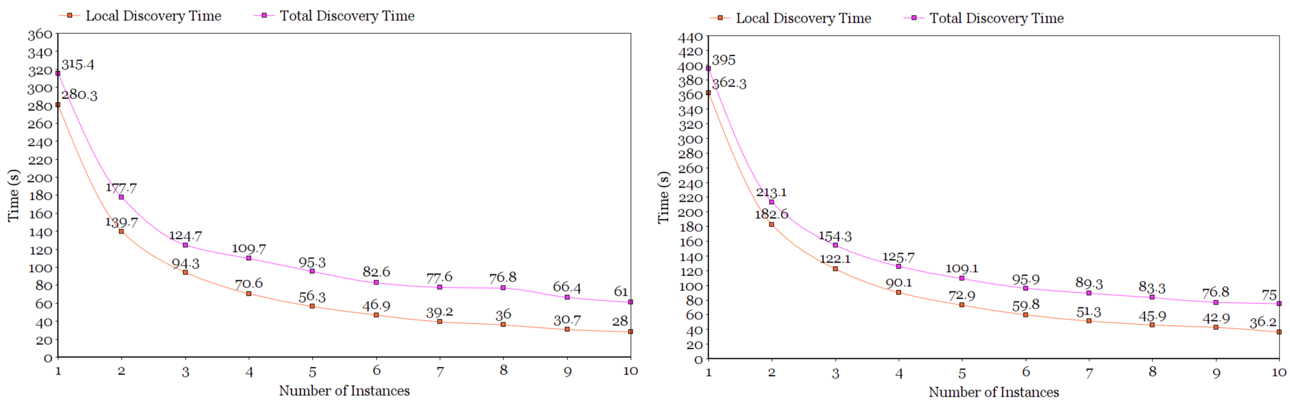


Figure 3: Line graph showing how discovery times change as the difficulty level increases when running on a single EC2 instance.

4.3 Horizontal Scaling Performance

Figure 4 shows the change in time taken for a golden nonce to be found for two different difficulty levels when the number of EC2 instances being used is increased.



(a) Difficulty level = 26 (b) Difficulty level = 28

Figure 4: Line graphs showing how discovery times change as the number of instances increases for two different difficulty levels.

For both of these difficulty levels, the discovery times are significantly reduced as the number of instances increases. For example, the local discovery time when 10 instances are used is almost exactly 10 times faster than the local discovery time when only one instance is used. These results show that there is a near perfect inverse relationship between the local discovery time and the number of instances:

$$\text{Local Discovery Time for } N \text{ instances} \approx \frac{\text{Local Discovery Time for 1 instance}}{N}$$

However, because the start-up time is not reduced when the number of instances increases, the same relationship does not exist for the total discovery time. For the difficulty levels shown in Figure 4, the total discovery time when 10 instances are used is approximately 5 times faster than the total discovery time when only one instance is used.

5 Setting N Implicitly

Using the results and observations made while performance testing my CND system, I was able to modify the system to set the number of instances N, implicitly. Rather than specifying N directly, the user can specify a desired run-time T and a desired confidence level L, and the system will automatically choose a value of N that yields a golden nonce within T seconds with $L\%$ confidence. This section explains how I derived an equation that takes in a number of parameters to produce a value for N.

I first defined equations for the run-time of my CND system:

$$P(t \leq T) = L \tag{1}$$

where:

t = total run-time (seconds)
T = desired run-time (seconds)
L = confidence level

$$t = \frac{X}{NY} + S \tag{2}$$

where:

X = attempts required to find golden nonce
N = number of EC2 instances
Y = attempts made by each EC2 instance per second
S = start-up time

Substituting equation (2) into equation (1) gives:

$$\begin{gathered} P(\frac{X}{NY} + S \leq T) = L \\ P(X \leq (T - S)NY) = L \end{gathered} \tag{3}$$

The number of attempts required to find a golden nonce (X) can be modelled as an exponential distribution. If the difficulty level is 1 then there is a 50% likelihood that any nonce will produce a hash with a single leading zero and therefore be a golden nonce. If the difficulty level is 2 then there is a 25% likelihood that any nonce will produce a hash with two leading zeros and therefore be a golden nonce. This shows:

$$P(\text{Some nonce is a golden nonce}) = (\frac{1}{2})^D \tag{4}$$

where D = difficulty level

Based on observations made while performance testing my CND system, X appears to increase exponentially as D increases. Therefore, X can be modelled with the following exponential distribution:

$$X \sim Exp((\frac{1}{2})^D) \tag{5}$$

Using the exponential cumulative distribution function along with equation (3) produces the following:

$$P(X \leq (T - S)NY) = 1 - e^{-(\frac{1}{2})^D (T-S)NY} \tag{6}$$

Therefore:

$$L = 1 - e^{-(\frac{1}{2})^D (T-S)NY} \tag{7}$$

$$1 - L = e^{-(\frac{1}{2})^D (T-S)NY} \tag{8}$$

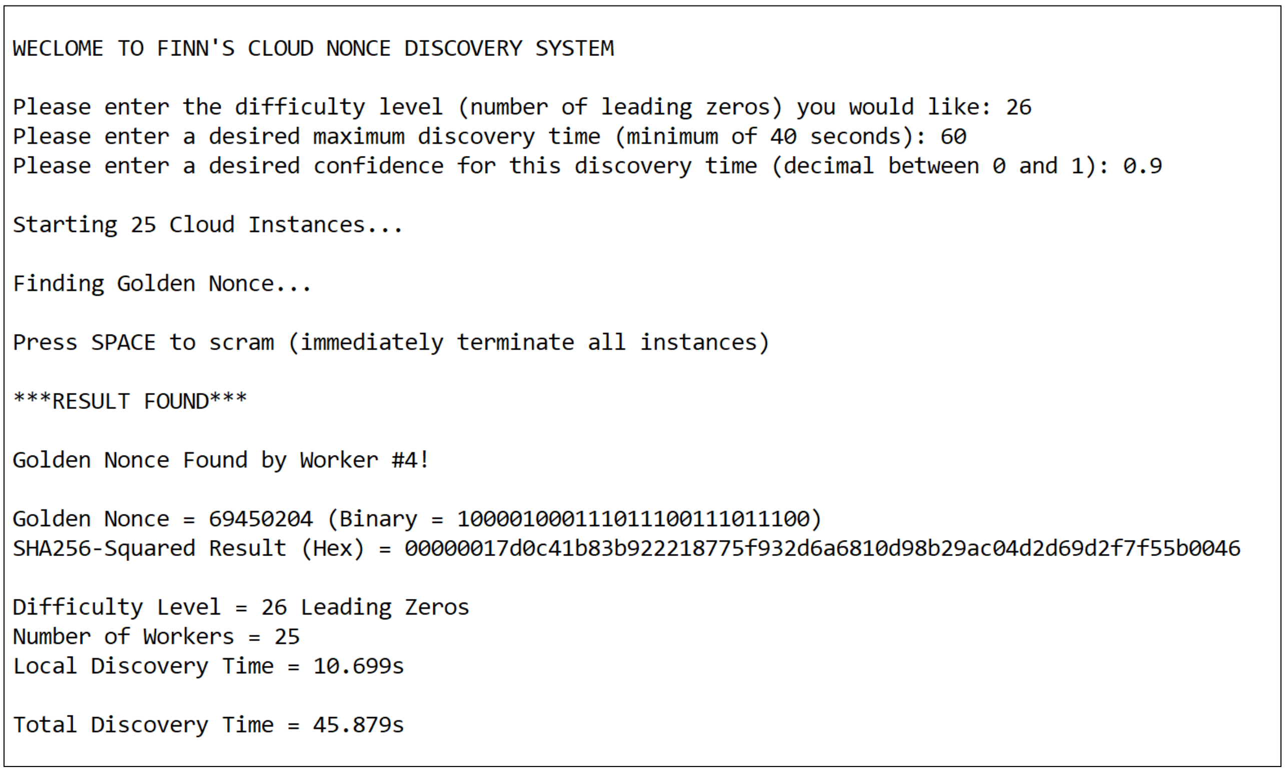
$$\ln(1 - L) = -(\frac{1}{2})^D (T - S)NY \tag{9}$$

$$N = -\frac{\ln(1 - L)}{(\frac{1}{2})^D (T - S)Y} \tag{10}$$

This equation is used to calculate a value of N based on the user-specified parameters D, T and L. Although, the start-up time of my CND system (S) and the nonce attempts made per second by each instance (Y) are not constant and can vary by a relatively significant amount, I have given them a constant value to simplify this equation. Based on the data gathered while performance testing the system, I have set S equal to 35 seconds and Y equal to 250,000 attempts per second. These approximations may make the equation slightly unreliable in some cases, but further testing has shown that this equation for N does produce accurate and reasonable discovery times based on the user specified values of T and L.

Appendices

Program Output Example



```
WECLOME TO FINN'S CLOUD NONCE DISCOVERY SYSTEM

Please enter the difficulty level (number of leading zeros) you would like: 26
Please enter a desired maximum discovery time (minimum of 40 seconds): 60
Please enter a desired confidence for this discovery time (decimal between 0 and 1): 0.9

Starting 25 Cloud Instances...

Finding Golden Nonce...

Press SPACE to scram (immediately terminate all instances)

***RESULT FOUND***

Golden Nonce Found by Worker #4!

Golden Nonce = 69450204 (Binary = 100001000111011100111011100)
SHA256-Squared Result (Hex) = 00000017d0c41b83b922218775f932d6a6810d98b29ac04d2d69d2f7f55b0046

Difficulty Level = 26 Leading Zeros
Number of Workers = 25
Local Discovery Time = 10.699s

Total Discovery Time = 45.879s
```

Figure 5: Example of the output printed by my Cloud Nonce Discovery System.

References

[1] Garnaat, M 2007, *Monster Muck Mashup - Mass Video Conversion Using AWS*, Amazon Web Services, viewed 3 December 2019, https://aws.amazon.com/articles/monster-muck-mashup-mass-video-conversion-using-aws/