CS 297 Report

An MP3-based Currency System.

By

Timothy Chen

Project Advisor: Dr. Chris Pollett

Department of Computer Science
San José State University
One Washington Square
San José, CA 95112

Introduction

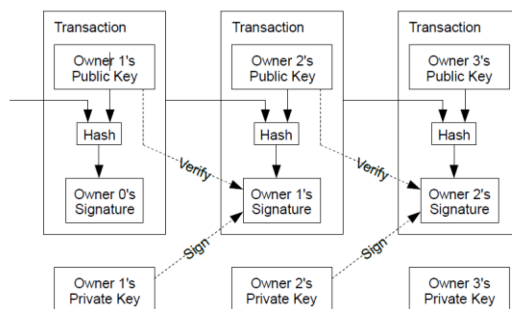
Recently crypto-currencies such as Bitcoin have become popular. This project proposes a new crypto-currency system which will allow music artists to upload their music in MP3 format to a website that I have implemented and earn the new cypto-currency I have created. The amount earned depends on how long the users of the website listen to the music. The technologies I am going to use are JavaScript for the front end of the website, PHP for the back end of the website, and a one way function to do the hash encryption for the currency coin.

My proposed MP3-based currency system will mimic the Bitcoin architecture with a modified mining process and a slightly different transaction process. Instead of using the hash cash-based proof-of-work system in mining Bitcoins, the MP3-based currency system will award artists after they upload an MP3-based song and user**s** listen **to** it. Like the Bitcoin system, the MP3-based currency system uses a public key infrastructure to verify transactions. The amount of currency transferred is based on the length of time for which an artist's song was listened to.

This semester, I worked on various experiments for the new currency system. I researched the background of the Bitcoin and how it works, and implemented hash function and music function. The work was divided into four deliverables. The first deliverable involved doing research to understand the Bitcoin, involved installing Bitcoin wallet, and involved trying to do Bitcoin mining. The second deliverable was using JavaScript to demonstrate a hash function to use for the new currency. The third deliverable was using JavaScript to implement Tiger hash or implement SHA256 hash. The fourth deliverable was using JavaScript to implement function to compute hashes as long as audio is being played.

Deliverable 1

During my research I found out that the Bitcoin virtual currency uses a peer-to-peer technology to operate without any central authority and there is no need to use banks to manage transactions. The issuing of Bitcoins is done by the network. Bitcoin is open-source; anyone can own or mine for Bitcoins. Bitcoins are spent like real currency. Each transaction is broadcast to the Bitcoin framework with details such as the amount, source, destination, timestamp of the transaction, and the public keys of the Bitcoins involved in the transaction. Each Bitcoin has a public key and a private key. The private key is used to determine who has ownership of the Bitcoin and the public key is used to sign the Bitcoin for owner verification for the transaction. For example, suppose Owner 1 needs to buy a car from Owner 2. Owner 1 needs to transfer the Bitcoins to the next person by digitally signing a hash of the previous transaction to change ownership of the Bitcoins to Owner 2 that uses the private key. The public key will be use to verify that the Bitcoin changed ownership to Owner 2 as the graph below shows:



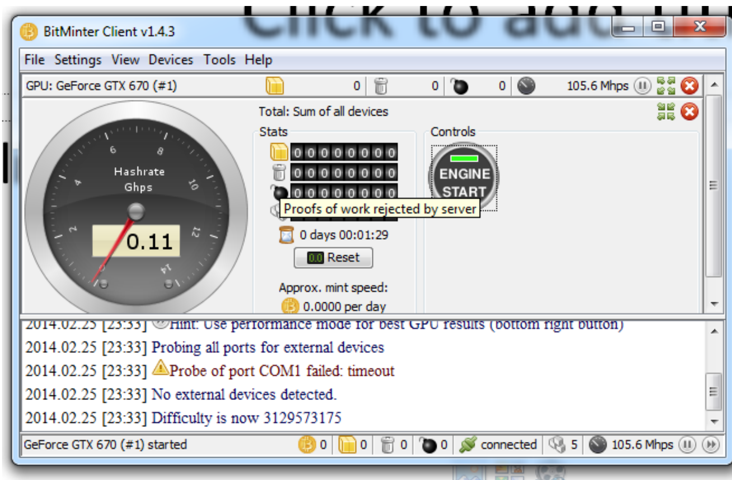
How do we prevent Owner 1 from double-spending the same Bitcoin to buy other stuff from Owner 3? The solution is to use a timeStamp. As mentioned before, the Bitcoin system is peer-to-peer, so anyone who gains ownership of Bitcoin will know the

transaction history of the Bitcoin. When someone gives someone else a Bitcoin, then everyone else knows at what time the transaction occurred. This will prevent the use of the same Bitcoin twice because once transaction occurs, the record of the transaction will go public. Everyone will know what time that Bitcoin was used, and the Bitcoin will not be able to reused by the same person twice in a row.

How do we get Bitcoins? There are three ways to get Bitcoins: first, they can be bought from a Bitcoin exchange website such as like https://www.mtgox.com/ (now defunct) and https://coinbase.com/. The price for Bitcoins fluctuates on a daily basis from about $100 to $1,000, similar to stocks. Second, one can get Bitcoins by Bitcoin mining. The mining system uses SHA256 hash generates a hash give 32 hex digit. If the generate begins with a zero bit in a single hash, then you got a Bitcoin. Over time, more people are doing Bitcoin mining, so it will get harder and harder to mine Bitcoins. Previously one could mine Bitcoin using a string such as 'abc' and by generating a 32-bit number that starts with two zeroes like 00123456789012345678901234567891, but now you have to generate a hash string like 00000056789012345678901234567891 with six zeroes at the beginning in order to mine a Bitcoin. The third way to get Bitcoins is to do verification. Whenever there is any transaction moving Bitcoins, people need to verify whether this coin was previously used or not by utilizing their computer's processing power. Whoever gets notified and verifies the transaction first will be paid small reward, which is typically a small part of a Bitcoin such as .00001 Bitcoin or even less depending on the amount of Bitcoin in the transaction. Bitcoin is a framework of coins made from digital signatures, which provides strong control of ownership and prevents double-spending. The only way to attack this framework may be to use fake nodes to send a lot of verify requests. The

verifying will be very slow and during that time, the attacker can double-send while verifications are not finished. Bitcoin exchange websites need to defend against this attack by coding carefully to prevent attackers from selling anything while the Bitcoin is not finished verifying prior transactions.

The next part of this deliverable was to install a Bitcoin wallet and mine Bitcoins. I went to the https://bitminter.com/ website and created an account, so that I have a wallet at the website. All the Bitcoin I mine are saved to this wallet. Then I installed the "BitMiner Client" application, which generates different hashes in order for me to mine Bitcoins and also do verification. All the Bitcoins I obtained from mining are saved to my account in bitminter.com. A screenshot of the "BitMiner Client" application is shown below:



The screenshot below shows how many Bitcoins I got from mining:

Account Details for timchen623

Unconfirmed income is added to your balance if and when blocks are confirmed. To improve your income per block, improve your score in the shifts eligible for payments by increasing your hash power.

Personal Assets	Balance	Unconfirmed	Future	Expected per block
Bitcoins [send]	0.00001243	-	0.00001243	0.00000423
Namecoins [send]	0.00002894	-	0.00002894	0.00000846

Email settings

To ensure messages are not caught in your spamfilter, please add noreply@bitminter.com and operator@bitminter.com to your address book and/or whitelist them with your anti-spam solution.

Your email address:

The application examines my GPU processing power to determine how many hashes I can generate every second to send to the Bitcoin framework and see if we can get any strings of hashes with starting zero the Bitcoin machine want it. The mining tool can use: ASIC card, our CPU, and graphics card. The speed at which Bitcoins are mined depends on how powerful our computer graphics card or ASIC machine is that we use to generate the hash. The chart below shows how different ASIC machines, Nvidia graphic card, and AMD(ATI) graph card can mine Bitcoins rate, and their current price I got below information from https://en.bitcoin.it/wiki/Mining_hardware_comparison/, http://www.videocardbenchmark.net/video_lookup.php?gpu=GeForce+GTX+670/, http://www.tomshardware.com/reviews/geforce-gtx-660-ti-benchmark-review,3279-12.html/, and http://www.Amazon.com/ :

Product Name	Generate Mhash/s	Price

ASIC Avalon Asic #1	107	$1299
ASIC Avalon Asic #2	117	$1499
ASIC Avalon Asic #3	117	$1499
AMD(ATI) Radeon HD 7970	603.8	$350
AMD(ATI) Radeon HD 7950	517	$229.99
AMD(ATI) Radeon HD 6970	389.9	$169.99
Nvidia GTX 770	123	$370
Nvidia GTX 670	112	$289
Nvidia GTX 660 Ti	96	$189

We found out ATI graphics cards are good for Bitcoin mining. For the same price, they are much faster than Nvidia. For example, the Nvidia GTX770 and ATI Radeon HD 7970 both cost around $350, but the GTX770 can only generate 123 hash strings while the Radeon HD 7970 can generate 603 hash strings. The Radeon HD7970 is four to five times faster at generating hashes than the GTX 770 for the same price.

Deliverable 2

The second deliverable is using JavaScript to demonstrate a hash function for use in a new currency. After the research I did on Deliverable 1, I found out how to generate hashes by having the user enter a string and how many zeroes they want at the start of the hash string, and then the hash function will return a hash string with the requested number of zeroes at the start. Below is a screenshot of my test application:

Demonstration

Text to hash	abc
Enter Zero	3
Calculate	MD5
Result	900150983cd24fb0d6963f7d28e17f72
ADDSTRING	
Result1	0006055c086a2fbc961b4c07263b949b

The “text to hash” input box allows a user to enter the text they want to hash and the “enter zero” input box will allows the user to choose how many zeros at the beginning are required in order to mine for a coin. The “calculate” section uses the MD5 hash algorithm to generate the 32 hex digit to be displayed in the "result" text box from the string that the user entered in the "Text to hash" text box. The code was obtained from http://pajhome.org.uk/crypt/md5/.

```
<input type="button" onclick="document.getElementById('hash').value =
hex_md5(document.getElementById('userString').value)" value="MD5">
```

Clicking “ADDSTRING” button, generates the hash based on the user inputs in the “text to hash” and “enter zero” input boxes using a function called “hex_add.” For example, if a user entered ‘abc’ this function will add characters after it. I use a for loop to determine the ASCII code from 0 to 255 to put into the MD5 function to see if the number of zeroes in the start of the hash string generated will match the value of the “Enter Zero” input box. If adding one character cannot generate a hash string as specified, then the function will keep changing and adding additional characters until the generated hash string satisfies the “Enter Zero” input. For example, given input ‘abc’ and ‘2’, character would result in string ‘bac1’ which will be changed to ‘abc2’ if abc1’ does not generate a hash string as specified. If just adding one character does not generate a hash string starting with two zeroes, we will add another character at end, and it will be abc11, then abc12, and so on. Basically, it tries all 256 ASCII codes for each added character until we generate the desired hash string. The screenshot below shows how this code works.

```
for (var i=0; i<len; i++)

{
        var currentone = (str.charAt(i)).charCodeAt(0);
        if (currentone <= 255)
        {
                        currentone = currentone+1;
                        var res = String.fromCharCode(currentone);
                        str = str.replaceAt(i,String.fromCharCode(currentone));
                        var hash = hex_md5(userstring+str);
                                if (hash.charAt(0) == "0")
                                {
                                        found = true;
                                }

                                for (var i=1; i<zerolength; i++)
                                {
                                        if (hash.charAt(i) != "0")
```

```
                        {
                                found = false;
                        }
                }
                if (found)
                {
                        break;
                }
        }
        else
        {
                currentone = 0;
                var init = String.fromCharCode(currentone);
                str = str.replaceAt(i,String.fromCharCode(currentone));
                str += String.fromCharCode(0);
        }
}
```

Deliverable 3

The third deliverable was to implement the Tiger hash function or SHA256 hash function. I decided to implement the SHA256 using the pseudocode from http://en.wikipedia.org/wiki/SHA-2. SHA256 is a one way hashing method, meaning it can encrypt text but cannot be used to generate the original text. The implementation works like this: when a user enters a string, the input string is divided into 512 bit message blocks. Each message block and its prior intermediate hash value is processed by a message schedule and a compression function to produce a 256 bit intermediate hash value. The initial hash value is the square root of the first eight primes 2…19. Each message block is further broken down into 16 32-bit words. The 16 words are extended to a 64 entry message schedule array. The message schedule function is to improve the compression function's quality since the compression function is operated on a longer message schedule array. The compression function consists of bitwise operations such as XOR, AND, OR, SHIFT operations and so on. The pseudo code of my implementation of the SHA256 algorithm from the Wikipedia article is described in more detail below:

```
Initialize hash values:
(first 32 bits of the fractional parts of the square roots of the first
8 primes 2..19):
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

Initialize array of round constants:
(first 32 bits of the fractional parts of the cube roots of the first
64 primes 2..311):
k[0..63] :=
   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

Pre-processing:
append the bit '1' to the message
append k bits '0', where k is the minimum number >= 0 such that the
resulting message
    length (modulo 512 in bits) is 448.
append length of message (without the '1' bit or padding), in bits, as
64-bit big-endian integer
    (this will make the entire post-processed length a multiple of 512
bits)

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    create a 64-entry message schedule array w[0..63] of 32-bit words
    (The initial values in w[0..63] don't matter, so many
implementations zero them here)
    copy chunk into first 16 words w[0..15] of the message schedule
array

    Extend the first 16 words into the remaining 48 words w[16..63] of
the message schedule array:
    for i from 16 to 63
        s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor
(w[i-15] rightshift 3)
        s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor
(w[i-2] rightshift 10)
        w[i] := w[i-16] + s0 + w[i-7] + s1

    Initialize working variables to current hash value:
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4
    f := h5
    g := h6
    h := h7

    Compression function main loop:
    for i from 0 to 63
        S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e
rightrotate 25)
```

```
        ch := (e and f) xor ((not e) and g)
        temp1 := h + S1 + ch + k[i] + w[i]
        S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a
rightrotate 22)
        maj := (a and b) xor (a and c) xor (b and c)
        temp2 := S0 + maj

        h := g
        g := f
        f := e
        e := d + temp1
        d := c
        c := b
        b := a
        a := temp1 + temp2

    Add the compressed chunk to the current hash value:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h

Produce the final hash value (big-endian):
digest := hash := h0 append h1 append h2 append h3 append h4 append h5
append h6 append h7
```

My implementation produced an identical hash result with website http://www.movable-type.co.uk/scripts/sha256.html, when I enter string ‘abc’ as my input text and also enter the same input string to that website, both give the same result of ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad. That shows that my implementation is correct.

A **cryptographic hash** (sometimes called 'digest') is a kind of 'signature' for a text or a data file. SHA-256 generate
most-unique 256-bit (32-byte) signature for a text. See **below** for the source code.

Enter any message to check its SHA-256 hash

Message abc

Generate Hash

Hash ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

Note SHA-256 hash of 'abc' should be: ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

Demonstration

Text to hash abc

Enter Zero

Calculate MD5 MY-SHA256

Result ba7816bf8f01cfea414140de5dae2223b003(

ADDSTRING

Result1

Deliverable 4

The fourth deliverable was using JavaScript to implement a function to compute hashes while playing audio. I used the function I implemented in Deliverable 2 to add a music bar into the GUI, which will be while the music player. When the user presses play audio, it will ignore the “Enter Zero” input field. While it plays, my implementation will run a while loop and add additional zeros out the beginning till the music is finished playing or the pause audio button has been pressed.

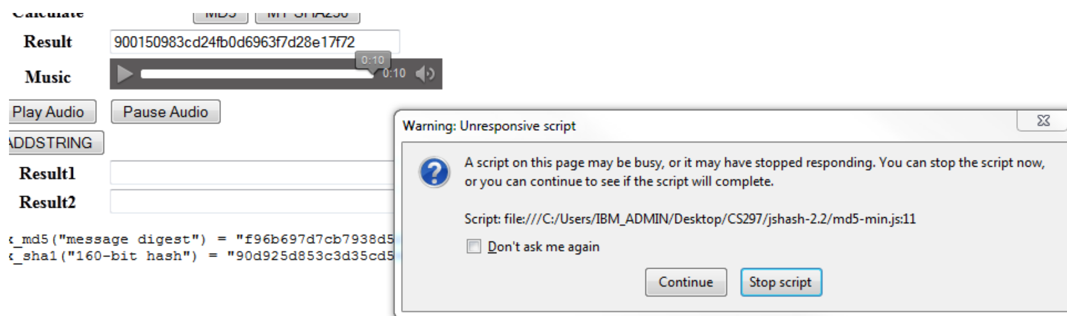
Demonstration

Text to hash abc
Enter Zero
Calculate MD5 MY-SHA256
Result 900150983cd24fb0d6963f7d28e17f72
Music 0:10 0:10
Play Audio Pause Audio
ADDSTRING
Result1
Result2 000004e22e53459b46ccd22b76606855

Once the music stops, it will give the number of zeros it has calculated during the music time length. For example, the above screenshot shows that during the ten second interval, it has calculated five zeroes. I used an HTML5 audio tag to get music from a local file and for the play audio button, I implemented a function called clickplay. When I press the play audio button, it will use the value from the ‘text to hash’ inputbox, which is used by my clickplay function. While the audio is playing, it will keep incrementing the zeroes to

pass to the function hex add starting with only one zero. If it returns a value and the music is still playing, it will add another zero to hex_add function. If the music stops or is paused then it will return the last value generated to the 'Result2' output text box.

This function seems to be working perfectly, but it has a problem: When it runs, I cannot press the pause button because the JavaScript cannot multi-task. So when I try to press the pause button, it will not work because it is still running in the while loop as well as playing audio and it creates a problem. The screenshot below shows the results:



Since JavaScript is single-threaded, I cannot use sleep() function in this method; I have to use setTimeout function instead. During the setTimeout, the code runs smoothly, but once the timer expires, the problem shows up again. The only solution for this problem is to use WebCL; it supports parallel computing in HTML5 web browsers and will allow me to run two threads at same time, which are play audio function and clickplay function. It uses the GPU and CPU, unlike my example website that only uses the CPU, so it will be much faster.

According to WebCL https://www.khronos.org/webcl/:

“WebCL 1.0 defines a JavaScript binding to the Khronos OpenCL standard for heterogeneous parallel computing. WebCL enables web applications to harness GPU and multi-core CPU parallel processing from within a Web browser, enabling significant acceleration of applications such as image and video processing and advanced physics for WebGL games. WebCL has been developed in close cooperation with the Web community and provides the potential to extend the capabilities of HTML5 browsers to accelerate computationally intensive and rich visual computing application.”

The problem with this approach is anyone who wants to run it must have both the OpenCL driver and the WebCL browser extension. I will use WebCL for my thesis implementation later.

Conclusion

The MP3-based crypto-currency system consists of the hash function and “play music” function. The ability to upload songs, play songs and WebCL are not included in this project. As Deliverable 1 we found out a wallet containing the history of each Bitcoin objects that keeps track of the public key, private key, and the balance of currency of a particular address. Its method includes:

- Generation of public/private key pairs
- Hashing and encoding of a public key into an address
- Store and retrieval of currency

For Deliverable 2 I implemented a simple hash function, similar to the one used by the Bitcoin framework. For Deliverable 3, I implemented SHA256, which works as expected. For Deliverable 4 I implemented the music function, which can play music and generate zeroes at same time, but it cannot pause because it was implemented using JavaScript.

For next semester I will implement the upload music function, use webCL to be able pause and play smoothly, and after webCL is implemented, I will solve a problem that if the user pauses, it resets the counter; that mean that the artist will be paid less if a user keeps pausing.

Reference:

- Satoshi Nakamoto (2009), Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved from https://bitcoin.org/bitcoin.pdf
- Florian Mendel and Vincent Rijmen, Cryptanalysis of the Tiger Hash Function. Retrieved from https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=31042
- SHA-256 Cryptographic Hash Algorithm. Retrieved from http://www.movable-type.co.uk/scripts/sha256.html
- WebCL Heterogeneous parallel computing in HTML5. web browsers Retrieved from https://www.khronos.org/webcl/
- WebCL. Retrieved from http://webcl.nokiaresearch.com/tutorials/tutorials.html
- Mining hardware comparison. Retrieved from https://en.bitcoin.it/wiki/Mining_hardware_comparison
- GeForce GTX 660 Ti Review: Nvidia's Trickle-Down Keplernomics. Retrieved from http://www.tomshardware.com/reviews/geforce-gtx-660-ti-benchmark-review,3279-12.html
- SHA-2 Retrieved from **http://en.wikipedia.org/wiki/SHA-2**
- Videocard Benchmarks Retrived from http://www.videocardbenchmark.net/video_lookup.php?gpu=GeForce+GTX+670
- Bitminer Retrieved from https://bitminter.com/login?F308838864701HX5WKM=_