



Abstract—This report details the design and development of a prototype distributed digital payment network and analyses various attacks against the system. The payment scheme uses an implementation of the SHA-256 algorithm written in Mathwork's MATLAB to successively hash a blockchain of public ledger entries which includes data of the token transactions and information about the total pool of participants. The ledger entries become immutable after being added to the public ledger. The robustness of the system design to the Byzantine attack and double-spending is discussed and analysed.

INTRODUCTION

Cryptocurrencies present an opportunity for global social advantage, and the democratisation of online commerce free from central authority. The widespread adoption and rampant economic speculation on online cryptocurrencies has met with much acclaim, criticism and scepticism. A decentralized, distributed digital payment system has multiple applications from small e-commerce transactions to massive disruption of the global banking system. An historic hostility to this potential threat to the financial hegemony of the international banking systems has given way to a tacit acceptance of existing early but often fervently-adopted commercial implementations such as Bitcoin, Ethereum, Ripple, Stellar, Cardano and a multiple of other digital currencies. The innovative use of a computational hashing to provide proof-of-work as implemented by Bitcoin was instrumental in the recent emergence of these systems. The energy inefficiency of such schemes has been criticised as wasteful, and there are several replacement methods attempting to ensure honest interaction of network nodes such as proof-of-stake or PoS/PoW hybrid schemes in active commercial development. However, the use of a globally accepted, international online payment system is still not a reality for the vast majority of humankind, with the existing technologies still nascent, or subject to specialised speculative investment rather than ubiquity.

1 CRYPTOCURRENCIES

Early designs and implementations in online currencies emphasised anonymity and untraceability[1][2] as core features, with a focus on distributed and peer-to-peer mechanisms of decentralised control. Influential early systems like 'Millicent' and 'Payword'[3] described workable schemes around minting tokens and online micro-transactions. There was much research into computationally light or efficient encryption schemes, such as [4] who proposed computationally efficient system of divisibility in token transactions, and 'MicroMint' suggested a trade-off of security and computational efficiency was justified in small-value, large-scale transactions. However, a persistent problem was the notion of double-spending inherent in a digital currency[5].

Any practical system must ensure that a token or coin, once spent, cannot be re-used again by the same spender for a second transaction. A related problem is the fraudulent minting of new tokens, which is difficult to control in a distributed network. Confirming the validity of new tokens without a central authority, or the fiat-issuance of a physical manifestation of a means of exchange presents a challenge. [6] suggested a system based on computational encryption that still relied on a central banking authority, as did [3] who defined different levels of credit and transactional interaction, between brokers, vendors and users. With the publication of systems of irrefutable blind signatures and anonymous communication[7], the notion of a distributed, verifiable record of all transactions became possible, in a truly decentralised system of exchange[8]. The development of asymmetric public/private key encryption such as RSA[9] provided a early practical mechanism of implementing this encryption. With a distributed network of majority consensus of the public record, mass-scale global transactions become possible.

1.1 Bitcoin

In 2008, the Bitcoin white paper was published[10], proposing a solution to the problem of double-spending, based on the computational infeasibility of replacing the agreed public transaction record (thus reversing any transaction). Trust between parties (or with larger corporate or governmental bodies that traditionally dominate national and global financial interaction between individuals) would be replaced by cryptographic proof contained in a chain of digital signatures, the blockchain. The owner of a digital token signs it over to a new recipient, using a hash of the previous transaction, and their RSA public key. The recipient of the funds can verify the transaction as it is public, and they are in possession of the private key to prove ownership of the encrypted token. The possibility of double-spending without a central authority is mitigated through the use of timestamps incorporated into the hashed transactions, the widely distributed nature of the network, and the introduction of the notion of proof-of-work, adopted from [11].

1.2 Blockchain

Transactional data is hashed with timestamps, which are themselves hashed along with proof of computationally intensive processes (repeatedly using the SHA-256 algorithm to find long sequences of zeros in the beginning of hash values. These nonces are discarded after use. In order to reverse transactions, the recalculation of all blocks in the chain must be performed. Financial incentives to

collectively participate honestly are presented in the manner of transactional fees and the stream of newly minted coins being generated from the proof-of-work mining. The efficient representation of the blockchain is achieved with Merkle tree structures to save on disk space, and reduce network traffic when publishing the blockchain.

1.3 Proof of Work

Bitcoin uses this system of computational operations to insure honest participation in the network of minting (or mining) coins, in a one-CPU/one-vote system based around computational resources essentially the conversion of electrical energy into an online currency. In order to repute the history of previous transactions, a participant must control computational resources in excess of half of the participating computers. This is considered infeasible, and more profitable to obey the existing consensus of transactions. As long as 51% of the participating computer nodes work for common profit, the system is considered secure and fair. The energy usage of Bitcoin mining has been criticized for its energy inefficient and wastage, and projections of detrimental environmental costs[12].

Alternative schemes to replace proof-of-work have been proposed, such as a hybrid scheme that has an initial proof-of-work, followed by proof-of-stake once large-scale participation has helped to ensure the honesty of participating miners, with a balance of centralised large stake shareholders and the mass of casual users[13]. It has been argued[12] that reducing the use of precious physical resources is possible while maintaining the integrity of the network from Byzantine attacks from within the network, by adjusting coin inflation vs. computational costs to favour honest participation.

2 SHA-256

Hashing algorithms are designed to cryptographically map an input message of any length to a fixed-length digest that can be used as an identifying fingerprint of the message, while being computationally infeasible to recover the original message from the digest. There are three primary standards in the NSA's FIPS, generally known as SHA-1, SHA-2 and a new hashing algorithm, SHA-3 (which is unrelated to the other two techniques). SHA-1 is considered insecure, but SHA-256 is widely used across a range of applications. The Secure Hashing Algorithm (256) is a global standard[14] for producing a 256-bit output digest using a 64 round schedule of non-linear and logical bit operations with 32-bit words. Many cryptocurrencies use SHA-256 for hashing purposes, to ensure a chain of transactional entries cannot be modified without causing a detectable difference in the hash value of the overall chain. Additionally, some currencies, notably Bitcoin, use hashing algorithms for computationally intensive operations to enforce honesty in the distributed nodal network of participants.

The input message is first padded to be a multiple of 512, and then a 64 round schedule of diffusion and confusion enciphers the message, producing a final hash value of 256 bits. The implementation of the algorithm for this project, `hash.m` is shown in Code Listing 1 in Appendix I, along with its subsidiary functions shown in Code Listings 1-12 in Appendix I.

2.1 Padding

The input message is converted to a `char` string, and then into a binary (logical) array. The message, of length l is then appended with a single '1' bit, followed by a sequence of k zeros up a total length of 448. The number of zeros, k is found by a brute force iteration through an array of [0:511] values to find the solution to equation 1.

$$l + 1 + k \equiv 448 \pmod{512} \quad (1)$$

Then the length of the message is encoded in a fixed-size 64-bit binary value, and appended to the message, producing a total length that is a multiple of 512. The message is then partitioned into (n) 512 bit blocks.

2.2 Encipherment

There are six main functions in the SHA-256 algorithm that make use of several logical bit-shifting operations, these functions are shown in equations 2-7.

$$Ch(x, y, x) = (x \wedge y) \oplus (\neg x \wedge z) \quad (2)$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \quad (3)$$

$$\Sigma_0 = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \quad (4)$$

$$\Sigma_1 = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \quad (5)$$

$$\sigma_0 = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \quad (6)$$

$$\sigma_1 = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \quad (7)$$

These operations are implemented in MATLAB functions in the solution, shown in Code Listings 3-8 in Appendix I, according to the FIPS180-4 standard[14].

3 SYSTEM DESIGN

The proposed prototype payment system is constructed around two main entities, the classes `Transaction` and `LedgerEntry`, which respectively constitute the data container for individual token exchange transactions, and the chain of securely hashed blocks that these transactions are stored within. After an initial hard-coded genesis block is created as the first entry in the public ledger, the pool of participants and their available tokens are first published. Now users can transfer over tokens to other members, using their public keys to digitally sign their transactions. The transaction data is anonymous aside from these keys. Providing proof of the private key is considered sufficient proof of ownership. The subsystem of claiming ownership and creating the public/private keys are not implemented.

Each transaction can be of any discrete amount currently contained within the pool of individually available tokens tied to a cryptographic key. Users can only spend the tokens that they can prove they possess. Once a transaction is performed (shown in Figure 1) the `Transaction` object is hashed with a timestamp and the sender and recipients' public keys, and then locked, becoming immutable. This transaction data is added into a ledger entry, which then hashes the combined values of the `Transaction` object, the hash of the `Ledger`'s properties, and the previous entry's hash value. This `LedgerEntry` is then validated and added into the public ledger. The ledger would then be published in a (non-implemented) broadcast to all available nodes. Competing recieved ledgers are validated based on their properties (comparing the genesis blocks, the validity of their chain of hash values and the history of transactions). The longest valid ledger with the most entries is favoured.

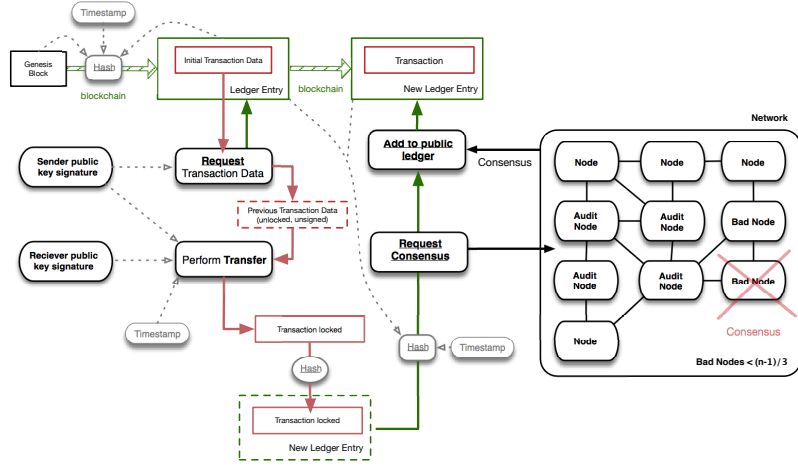


Fig. 1: Basic transaction workflow

3.1 Advantages

The system proposed is simple to understand and implement, and it is quick to verify transactions within the system and externally, using publicly available standards of encryption and hashing. The immutability of transactions provides a permanent audit trail of all token transfers, and the hashed chain of ledger entries allows verifiability. Using cryptographic keys to prove ownership ensures sufficient anonymity.

3.2 Limitations

New participants would have to purchase or earn tokens from existing members of the pool. This would have sharp inflationary effects in the case of broad adoption of the system without a system for introducing and distributing new coins into the supply, leading to over-valuation. This system is currently only suitable for small-scale micro-transactions, and requires significant modification to enable sustainable growth in a larger pool of participants.

4 ATTACKS

4.1 Byzantine Attack

To prevent deceptive or dishonest publication of false transactions from self-interested participants, a consensus-based system governed by a central group of decision making nodes is proposed but not implemented. Using the algorithm detailed in [15], a threshold of no more than $(n-1)/3$ erroneous or dishonest nodes being tolerated (by specialised auditing nodes) would establish sufficient Byzantine Fault Tolerance for system stability. A strict time-based schedule for consensus about the most recently broadcast ledger's validity would encourage rapid vote-casting for ledger acceptance. Ordinary nodes can also vote, but the threat of exclusion from the system (and loss of access to all tokens) acts as a countermeasure to deception. The criteria for inclusion in the inner network of trusted nodes limits the decentralisation of the system. The auditing nodes are also responsible for the release of new tokens, based on the value of the computational processes performed. It is hoped that performing "useful work" for scientific or research purposes would lead to beneficial outcomes for a larger collective outside the network of participants, and encourage new nodes to join the system.

4.2 Double Spending

Transactions, once accepted into the system, are irreversible without majority consensus and the entries themselves are immutable. The ledger's integrity is ensured by the use of sequential blockchain hashing, requiring a hard fork, which implies exclusion from the majority of participants. Any longer ledger becomes the new accepted record of transactions, a vulnerability that the limiting ratio of dishonest to honest nodes would mitigate.

4.3 Privacy

The use of public keys and hash-encrypted entries allows anonymous participation. The risk of financial information being publicly available must be weighed against the value of a workable, broadly decentralised transactional system. While there is no formal proof of the irreversibility of RSA and SHA, it is currently considered computationally infeasible to recover the private key with a sufficiently large keyspace, and impossible to retrieve the original message from digest created from a sufficiently large hashspace.

5 IMPLEMENTATION

MATLAB was used to write code prototyping the basic operation of the hashing function and the ledger and transactional systems, with a demonstration of non-exhaustive attempts to attack the ledger and its transactional data. This prototype system is demonstrated in the code `PaymentSystem.m`, shown in Code Listing 13

Transaction
Data
Hash
ID_index
ID_me
ID_recipient
ID_sender
Locked
Timestamp
Transaction_amount
+Transaction (data, signature)
+alterIdentity (id)
+authenticateByID (id)
+authenticateBySignature (signature)
+doesExist (id)
+getBalance (id)
+getSignature (id)
+getTotalBalance ()
+isAMember (signature)
+lock ()
+makeTransfer (amount, sender, recipient, output)
+removeByID (id)
+removeBySignature (signature)
+setBalance (value, id)
+setHash ()

Fig. 2: The Transaction Class

5.1 Transaction

The `Transaction` class, shown in Figure 2 and Code Listing 16 is a container for the pool of participants, identified by their hashed public keys. For demonstration purposes, the prototype uses fake public keys, and an id-based indexing system that would be insecure in practice. In order to make a transaction, a user retrieves a copy of the latest transaction (using the function in Code Listing 21) from the latest ledger entry, and performs a transfer. The system only permits a single transaction, with access limited to the balance of the user. Without access to a specific private key, the funds from a user cannot be altered and accepted as a valid ledger entry. The user is only able to transfer tokens below their personal balance in the latest ledger entry. Although the consensus and broadcast systems are not implemented in this prototype, the mechanism for a member to removed from the system is coded in the code shown in Code Listing 16 in Appendix I.

Once a transfer is performed, the transaction is locked and cannot be altered. The transaction is hashed and placed into a new `LedgerEntry` created with the code shown in 17, and added to the ledger using the code shown in Code Listing 18. The post-transaction authentication checks are implemented with the code shown in Code Listing 23.

5.2 LedgerEntry

LedgerEntry
Hash Index PreviousHash Timestamp TransactionData
+LedgerEntry (index, previousHash, timestamp, data, hash) +getTransactionData ()

Fig. 3: The `LedgerEntry` Class

The `LedgerEntry` class is shown in Figure 3, and in Code Listing 15. The initial `LedgerEntry` is created with `genesisEntry()`, shown in Code Listing 14 hardcoded with a specific hash and the public key of the creator, along with a hard-coded timestamp and overall hashed value of the entry properties. In the case of a forked ledger, or competing ledgers being broadcast to network nodes, the ledgers are examined within the function `replaceLedger()`, shown in Code Listing 24 and validated with the function `isEntryValid.m` which proceeds through both chains of ledger entries, starting with the genesis block, then calculating and verifying the successive hashes. If both ledgers are valid, the longest ledger wins. This makes the system vulnerable to being overtaken by nodes that are able to produce the longest valid ledger in their own interest, but the proposed system of a limited ratio of bad nodes and the threat of being removed from the system and losing all tokens is a countermeasure to limit this behaviour. The broadcasting and consensus decision-making systems of the ledger is not implemented.

5.3 Additional code

In order to realise the system, various helper and conversion functions were written to improve or supplement the available built-in MATLAB functions. A fixed-bit-size

decimal to binary converter along with the inverse function to improve on the binary-to-decimal bit-size limitation of MATLAB's native function were implemented (shown in Code Listings 25 and 26). Helper functions to convert between logical arrays to char or string arrays were also written, to help check that the bit manipulation when reshaping binary arrays in the SHA-256 function were not mangling the message data.

5.4 Testing

Code Listing 38 demonstrates the hash function operating on a variable-sized, constantly shrinking message, with an analysis of the sequential hash collisions. The probability of successive hash nibbles being the same as the previous hash (with the input message differing by a single bit) was calculated as 0.065735. The conversion functions were all tested and verified, using the code in Code Listing 37.

6 ADDITIONAL ECONOMIC FACTORS

The laws of supply and demand require that a steady stream of new coins must be issued into a steadily growing pool of users, in order to offset inflationary or deflationary effects. One proposal for balancing a computationally complex scheme of controlling the minting of new coins with the larger wastage of physical resources is the use of weighted coin release as an incentive for performing "useful work", such as molecular analysis of proteins currently restricted to HPC clusters and distributed farms of voluntary participants in programs like Stanford's folding@home project[16]. Other examples are contributing computational power. Coin minting should be sufficiently limited to impart value to the tokens, but regular enough to limit inflation, and keep the token transaction costs low enough to become a ubiquitous mechanism for micro-purchases. Incentivising scientific or medical research-based computational resources with financial rewards in the form of spendable online tokens could be a good trade-off between energy costs and personal profitability.

CONCLUSION

A prototype digital token system was designed and implemented, without any broadcast or consensus mechanisms being practically realised. A non-canonical implementation of SHA-256 was written in MATLAB. A secure transfer OOP class-based mechanism for exchanging tokens between participants, using the principles of a blockchain hashed with successive hashes of the ledger entries' properties and timestamps was programmed. The basic mechanisms of transactions and creating, and validating the ledger was demonstrated in working code, along with testing and measurements of the hashing algorithm. Further development of the system with consensus-based broadcasting and a privileged group of auditing nodes was discussed, along with an analysis of the difficulty of defending against a Byzantine attack. A proposal for developing Byzantine Fault Tolerance based on node removal from the network to maintain a maximum bad node ratio and timestamp server broadcasting was briefly discussed. The possibility of an altruistic, research-based computational proof of work system was proposed.

REFERENCES

- [1] D. Chaum, "Security without identification: Transaction systems to make big brother obsolete," *Communications of the ACM*, vol. 28, no. 10, pp. 1030–1044, 1985.
- [2] T. Okamoto and K. Ohta, "Universal electronic cash," in *Annual International Cryptology Conference*. Springer, 1991, pp. 324–337.
- [3] R. L. Rivest and A. Shamir, "Payword and micromint: Two simple micropayment schemes," in *International workshop on security protocols*. Springer, 1996, pp. 69–87.
- [4] T. Okamoto, "An efficient divisible electronic cash scheme," in *Annual International Cryptology Conference*. Springer, 1995, pp. 438–451.
- [5] 21st Century Coin Company Inc., "HashCash," <http://hashcash.com/about.html>.
- [6] D. Chaum, A. Fiat, and M. Naor, "Untraceable electronic cash," in *Conference on the Theory and Application of Cryptography*. Springer, 1988, pp. 319–327.
- [7] D. Chaum, "Blind signatures for untraceable payments," in *Advances in cryptology*. Springer, 1983, pp. 199–203.
- [8] K. Q. Nguyen, Y. Mu, and V. Varadharajan, "Digital coins based on hash chain," in *National Information Systems Security Conference*, 1997.
- [9] R. L. Rivest, A. Shamir, and L. M. Adleman, "Cryptographic communications system and method," September 1983, US Patent 4,405,829.
- [10] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
- [11] A. Back *et al.*, "Hashcash-a denial of service counter-measure," 2002.
- [12] B. L. A. L. Make and B. Laurie, "Decentralised currencies are probably impossible."
- [13] I. Bentov, A. Gabizon, and A. Mizrahi, "Cryptocurrencies without proof of work," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 142–157.
- [14] D. E. Standard *et al.*, "Federal information processing standards publication 46," *National Bureau of Standards, US Department of Commerce*, vol. 4, 1977.
- [15] E. Zhang, "A Byzantine Fault Tolerance Algorithm for Blockchain," <http://docs.neo.org/en-us/node/whitepaper.html>.
- [16] S. University, "folding@home," <http://folding.stanford.edu>.

SHA-256 CODE

Code 1: hash.m

```

1 function [ digest ] = hash( message )
2 %hash() hashes an input message with an implementation of SHA256
3 %
4 % Input message is padded and split into 512-bit blocks.
5 % Output is a 256-bit hexadecimal digest (char)
6 % See FIPS PUB 180-4 for the implementation standard
7 % http://dx.doi.org/10.6028/NIST.FIPS.180-4
8
9 % Tyson Cross 1239448
10
11 % Initialise working register arrays
12 a = false(1,32);
13 b = false(1,32);
14 c = false(1,32);
15 d = false(1,32);
16 e = false(1,32);
17 f = false(1,32);
18 g = false(1,32);
19 h = false(1,32);
20
21 word_length = 32;
22 mod_value = 2^32;
23
24 % K is first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers
25 K1 = dec2bin(hex2dec(...
26     [ '428a2f98'; '71374491'; 'b5c0fbcf'; 'e9b5dba5'; '3956c25b'; '59f111f1'; '923f82a4'; 'ab1c5ed5'; ...
27       'd807aa98'; '12835b01'; '243185be'; '550c7dc3'; '72be5d74'; '80deb1fe'; '9bdc06a7'; 'c19bf174'; ...
28       'e49b69c1'; 'efbe4786'; '0fc19dc6'; '240calcc'; '2de92c6f'; '4a7484aa'; '5cb0a9dc'; '76f988da'; ...
29       '983e5152'; 'a831c66d'; 'b00327c8'; 'bf597fc7'; 'c6e00bf3'; 'd5a79147'; '06ca6351'; '14292967'; ...
30       '27b70a85'; '2e1b2138'; '4d2c6dfc'; '53380d13'; '650a7354'; '766a0abb'; '81c2c92e'; '92722c85'; ...
31       'a2bfe8a1'; 'a81a664b'; 'c24b8b70'; 'c76c51a3'; 'd192e819'; 'd6990624'; 'f40e3585'; '106aa070'; ...
32       '19a4c116'; '1e376c08'; '2748774c'; '34b0bcb5'; '391c0cb3'; '4ed8aa4a'; '5b9cca4f'; '682e6ff3'; ...
33       '748f82ee'; '78a5636f'; '84c87814'; '8cc70208'; '90befffa'; 'a4506ceb'; 'bef9a3f7'; 'c67178f2']));
34
35 % H is first 32 bits of the fractional parts of the square roots of the first 8 prime numbers
36 H1 = dec2bin(hex2dec(...
37     [ '6a09e667'; 'bb67ae85'; '3c6ef372'; 'a54ff53a'; '510e527f'; '9b05688c'; '1f83d9ab'; '5be0cd19' ...
38       ]));
39
40 K = false(64,32);
41 for i=1:64
42     K(i,:) = char2logical(K1(i,:));
43     assert(strcmp(logical2char(K(i,:)),K1(i,:)));
44 end
45
46 H = false(1,8,32);
47 for i=1:8
48     H(1,i,:) = char2logical(H1(i,:));
49     assert(strcmp(logical2char(H(1,i,:)),H1(i,:)));
50 end
51
52 % Message preprocessing
53 if ~islogical(message)
54     if ischar(message)
55         message_logical = str2logical(message);
56     elseif isnumeric(message)
57         message_logical = logical(message);
58     end
59 end
60
61 % Pad to make the message a multiple of 512
62 message_padded = padder(message_logical);
63
64 % Parse into n blocks of 512 bits, then reshape into 3D matrix (Nx16x32)
65 message_flat = reshape(message_padded,512,[]);
66 [N, ~] = size(message_flat);
67 [row,col] = size(message_flat);
68 M = permute(reshape(message_flat',[word_length,row/word_length,col]),[3,2,1]);
69
70 % Initialise W (word schedule)
71 [x,y,z] = size(M);
72 W = false(N*x,y,z);
73
74 clear x y z row col message_padded message_flat message_logical K1 H1 input ;
75
76 % Process message blocks (total of N message blocks)
77 % Matlab indexing starts at 1, unfortunately, so H(0) must be treated as H(1)

```



```

77 for i=1:N; j=i+1;
78
79     % Hash computation, Stage 1
80     for t=1:16
81         W(i,t,:) = M(i,t,:);
82     end
83
84     for t=17:64
85         alpha = sigma_1(flattenlogical(W(i,t-2,:)));
86         beta = flattenlogical(W(i,t-15,:));
87         Δ = sigma_0(flattenlogical(W(i,t-15,:)));
88         gamma = flattenlogical(W(i,t-16,:));
89         epsilon = mod_addition(alpha, beta, Δ, gamma);
90         W(i,t,:) = flattenlogical(epsilon);
91         clear alpha beta gamma Δ epsilon;
92     end
93
94     % Hash computation, Stage 2
95     a = flattenlogical(H(i,1,:));
96     b = flattenlogical(H(i,2,:));
97     c = flattenlogical(H(i,3,:));
98     d = flattenlogical(H(i,4,:));
99     e = flattenlogical(H(i,5,:));
100    f = flattenlogical(H(i,6,:));
101    g = flattenlogical(H(i,7,:));
102    h = flattenlogical(H(i,8,:));
103
104    % Hash computation, Stage 3
105    for t=1:64
106        alpha = h;
107        beta = E_1(e);
108        Δ = Ch(e,f,g);
109        gamma = flattenlogical(K(t,:));
110        epsilon = flattenlogical(W(i,t,:));
111        T_1 = mod_addition(alpha, beta, Δ, gamma, epsilon);
112        T_2 = mod_addition(E_0(a), Maj(a,b,c));
113        h = g;
114        g = f;
115        f = e;
116        e = mod_addition(d, T_1);
117        d = c;
118        c = b;
119        b = a;
120        a = mod_addition(T_1, T_2);
121        clear alpha beta gamma Δ epsilon;
122    end
123
124    % Hash computation, Stage 2 - H(jth) intermediate hash value
125    H(j,1,:) = mod_addition(a, flattenlogical(H(i,1,:)));
126    H(j,2,:) = mod_addition(b, flattenlogical(H(i,2,:)));
127    H(j,3,:) = mod_addition(c, flattenlogical(H(i,3,:)));
128    H(j,4,:) = mod_addition(d, flattenlogical(H(i,4,:)));
129    H(j,5,:) = mod_addition(e, flattenlogical(H(i,5,:)));
130    H(j,6,:) = mod_addition(f, flattenlogical(H(i,6,:)));
131    H(j,7,:) = mod_addition(g, flattenlogical(H(i,7,:)));
132    H(j,8,:) = mod_addition(h, flattenlogical(H(i,8,:)));
133
134    for k=1:8
135        H_N{k} = dec2hex(bin2decimal(logical2char(H(j,k,:))),8);
136    end
137 end
138
139 digest = strjoin(H_N,'');
140
141 end

```

Code 2: padder.m

```

1 function [ M ] = padder( input )
2 %padder() ensures that a message length is a multiple of 512
3 % The logical array is appended with a single '1' bit, then
4 % padded with 448 zeros, then the length of the message in binary:
5 % [ logical_array 1 padded_zeros length_of_message ]
6
7 % Tyson Cross 1239448
8
9 if islogical(input)
10     msg = input;
11 elseif ischar(input)
12     msg = str2logical(input);
13 elseif isnumeric(input)
14     msg = str2logical(num2str(input));
15 end

```

```

16
17 len = length(msg);
18 bin_len = dec2binary(len,64); % 64 bits
19
20 k = 0:511; % Brute force find the value of the zero padding
21 zero_len = k(mod(len + 1 + k, 512) == 448); % to solve the equation len + 1 + k = 448 mod 512
22 zero_pad = false(1,zero_len); % Number of zeros
23 M = [msg true zero_pad bin_len]; % append the message length in binary (64-bits)
24 assert(mod(numel(M),512)==0);
25
26 end

```

Code 3: Maj.m

```

1 function [ word ] = Maj(x,y,z)
2 %Maj() implements (x^y)XOR(x^z)XOR(y^z) 4.1.2
   (4.3)checkLength(x,32);checkLength(y,32);checkLength(z,32);checkLogical([x y z]);word =
   bitxor(bitxor(x,y),z);end

```

Code 4: Ch.m

```

1 function [ word ] = Ch(x,y,z)
2 %Ch() implements (x^y)XOR(¬x^z) 4.1.2
   (4.2)checkLength(x,32);checkLength(y,32);checkLength(z,32);checkLogical([x y z]);a = bitand(x,y);b =
   bitand(¬x,z);word = bitxor(a,b);end

```

Code 5: E0.m

```

1 function [ word ] = E_0(x)
2 %E_0() implements (ROTR^2(x))XOR(ROTR^13(x))XOR(ROTR^22(x)) 4.1.2
   (4.4)checkLogical(x);checkLength(x,32);word = bitxor(bitxor(ROTR(x,2),ROTR(x,13)),ROTR(x,22));end

```

Code 6: E1.m

```

1 function [ word ] = E_1(x)
2 %E_1() implements (ROTR^6(x))XOR(ROTR^11(x))XOR(ROTR^25(x)) 4.1.2
   (4.5)checkLogical(x);checkLength(x,32);word = bitxor(bitxor(ROTR(x,6),ROTR(x,11)),ROTR(x,25));end

```

Code 7: sigma0.m

```

1 function [ word ] = sigma_0(x)
2 %sigma_0() implements (ROTR^7(x))XOR(ROTR^18(x))XOR(SHR^3(x)) 4.1.2
   (4.6)checkLogical(x);checkLength(x,32);word = bitxor(bitxor(ROTR(x,7),ROTR(x,18)),SHR(x,3));end

```

Code 8: sigma1.m

```

1 function [ word ] = sigma_1(x)
2 %sigma_1() implements (ROTR^17(x))XOR(ROTR^19(x))XOR(SHR^10(x)) 4.1.2
   (4.7)checkLogical(x);checkLength(x,32);word = bitxor(bitxor(ROTR(x,17),ROTR(x,19)),SHR(x,10));end

```

Code 9: ROTL.m

```

1 function [word] = ROTL(x,n)
2 %ROTL() implements the rotate-left (circular left shift) ...
   2.2.2checkLogical(x);checkLength(x,32);checkRange(n,0,32);word = circshift(x,[0
   -n]);end

```

Code 10: ROTR.m

```

1 function [word] = ROTR(x,n)
2 %ROTR() implements the rotate-right (circular right shift) ...
   2.2.2checkLogical(x);checkLength(x,32);checkRange(n,0,32);word = circshift(x,[0
   n]);end

```

Code 11: SHR.m

```

1 function [word] = SHR(x,n)
2 %SHR() implements the shift-right operation 2.2.2checkLength(x,32);checkLogical(x);a = x(1:end-n); z =
   false(1,n); word = [z a]; assert(isequal(length(word),length(x)));end

```


Code 12: modAddition.m

```
1 function [output] = mod_addition(varargin)
2 %mod_addition() implements addition modulo 2^32 2.2.2value = uint64(0);modvalue = uint64(pow(2,32));for
   i=1:length(varargin)value = value + bin2decimal(varargin{i});endoutput =
   dec2binary(mod(value,modvalue),32);end
```

PAYMENT SYSTEM CODE

Code 13: PaymentSystem.m[1][2]

```

1  % Structural ideas based on the broad concept of Bitcoin https://github.com/bitcoin/bitcoin
2  % with some implementation design from Naivechain https://github.com/lhartikk/naivechain
3
4  addpath('SHA256','utilities','Blockchain'); clc; clear all;
5
6  %% Setup
7  global LEDGER;
8
9  % Generate a basic system of participants and pre-existing tokens
10 people = loadPeople(9);
11
12 % My identity
13 my_Signature = people{1:1};
14 my_ID = 1;
15
16 % create a simple blockchain
17 % ideally a tree or linked list structure, for now a simple vector array
18 LEDGER = [genesisEntry()];
19 disp('Created a new ledger')
20
21 disp(' ')
22 disp('Initialise the transaction data, put the available people and tokens into the ledger...')
23 tx2 = Transaction(people, my_Signature);
24 lg2 = createLedgerEntry(tx2,LEDGER);
25 LEDGER = addEntry(lg2, LEDGER);
26
27 %% Output
28 disp(' ')
29 disp(' ID                               Signature                               ...
30       Tokens')
31 for i=1:numel(people)/2
32     disp([' ', num2str(i), ' ', char(people{i,1}), ' ', num2str(people{i,2})])
33 end
34 original_tokens = LEDGER(end).getTransactionData.getTotalBalance;
35 disp(['Total: ', num2str(original_tokens) ' Tokens'])
36 clear people tx2;
37
38 %% Transactions
39 disp(' ')
40 disp('Perform a transaction as another person (for demonstration)...')
41 tx3 = createNewTransactionFromLastRecord(LEDGER, my_ID);
42 [ amount, sender, reciever ] = randomTransactionParameters(tx3);
43 tx3.alterIdentity(sender);
44 tx3.makeTransfer(amount, sender, reciever, true);
45 lg3 = createLedgerEntry(tx3,LEDGER);
46 LEDGER = addEntry(lg3, LEDGER);
47 clear tx3;
48
49 %% Attacks
50 disp(' ')
51 disp('Attempting attacks on the ledger')
52 disp('-----')
53 disp(' ')
54 disp('Attempt to repeat a previous transaction...')
55 % will fail because of an invalid index check
56 LEDGER = addEntry(lg3, LEDGER);
57 clear lg3;
58
59 disp(' ')
60 disp('Attempt to add an invalid entry, favouring Person1@Computer1...')
61 % transfer 1000 tokens from another person to myself
62 % will fail because of an authentication check
63 tx_invalid = createNewTransactionFromLastRecord(LEDGER, my_ID);
64 tx_invalid.alterIdentity(my_ID);
65 tx_invalid.makeTransfer(1000, 2, 1, true);
66 clear tx_invalid;
67
68 disp(' ')
69 disp('Attempt to transfer more tokens than current balance...')
70 tx_overspend = createNewTransactionFromLastRecord(LEDGER, my_ID);
71 tx_overspend.alterIdentity(my_ID);
72 amount = tx_overspend.getBalance(my_ID) + 1000;
73 tx_overspend.makeTransfer(amount, my_ID, 2, true);
74 clear tx_overspend;
75
76 disp(' ')
77 disp('Attempt to manipulate the token balance...')
78 tx_cheat = createNewTransactionFromLastRecord(LEDGER, my_ID);
79 tx_cheat.alterIdentity(my_ID);
80 amount = tx_cheat.getBalance(my_ID) + 10000;

```

```

80 try tx_cheat.Data(my_ID,2) = amount;
81 catch
82     disp('The Data property of Transaction is not accessible')
83 end
84 clear tx_cheat;
85
86 disp(' ')
87 disp('Attempt to insert an forged transaction into the ledger...')
88 lg_insert = copy(LEDGER(end));
89 tx_insert = copy(lg_insert.getTransactionData);
90 tx_insert.alterIdentity(2);
91 tx_insert.makeTransfer(10, 2, 1, false);
92 lg_insert_fake = createLedgerEntry(tx_insert, LEDGER);
93 try lg_insert.Index =lg_insert_fake.Index;
94 catch
95     disp('Unable to alter the index, relevant properties of the class are read-only')
96 end
97 try LEDGER = addEntry(lg_insert, LEDGER);
98 catch
99     disp('Unable to add ')
100 end
101 clear tx_insert lg_insert lg_insert_fake;
102
103 disp(' ')
104 disp('Simple attempt to corrupt the ledger...')
105 tx_corrupt = createNewTransactionFromLastRecord(LEDGER, randi(9));
106 lg_corrupt = createLedgerEntry(tx_corrupt, LEDGER);
107 try LEDGER = addEntry(tx_corrupt, LEDGER);
108 catch
109     if isValidLedger(LEDGER)
110         disp('Causes an error, but the ledger is unaffected.')
111     else
112         disp('The ledger has been corrupted')
113     end
114 end
115 clear tx_corrupt lg_corrupt;
116
117 % Make a fork of the ledger...
118 ledger_competing = copy(LEDGER);
119
120 disp(' ')
121 disp('Perform some transactions to grow the ledger... (ouput disabled for brevity)')
122 for i=4:6
123     tx_new = createNewTransactionFromLastRecord(LEDGER, my_ID);
124     [ amount, sender, reciever ] = randomTransactionParameters(tx_new);
125     tx_new.alterIdentity(sender);
126     tx_new.makeTransfer(amount, sender, reciever, false);
127     lg_new = createLedgerEntry(tx_new, LEDGER);
128     LEDGER = addEntry(lg_new, LEDGER);
129     fprintf('.')
130 end
131 disp(' ')
132 clear lg_new tx_new;
133
134 disp(' ')
135 disp('Attempt to replace the ledger with a shorter, competing ledger...')
136 LEDGER = replaceLedger(ledger_competing, LEDGER);
137
138 disp(' ')
139 disp('Perform more transactions on the competing ledger... (ouput disabled for brevity)')
140 for i=4:8
141     tx_competing = createNewTransactionFromLastRecord(ledger_competing, my_ID);
142     [ amount, sender, reciever ] = randomTransactionParameters(tx_competing);
143     tx_competing.alterIdentity(sender);
144     tx_competing.makeTransfer(amount, sender, reciever, false);
145     lg_competing = createLedgerEntry(tx_competing, ledger_competing);
146     ledger_competing = addEntry(lg_competing, ledger_competing);
147     fprintf('.')
148 end
149 disp(' ')
150 clear lg_competing tx_competing;
151
152 disp(' ')
153 disp('Attempt to replace the ledger with the longer competing ledger...')
154 LEDGER = replaceLedger(ledger_competing, LEDGER);
155
156 disp(' ')
157 disp('End of attacks')
158
159 %% Output
160 disp(' ')
161 disp(' ')
162 disp(['Total Tokens at original initialisation:', num2str(original_tokens) ' Tokens'])
163 disp(['Total Tokens in pool at end of testing: ', ...
    num2str(LEDGER(end).getTransactionData.getTotalBalance) ' Tokens'])

```

Code 14: genesisEntry.m

```

1 function [genesis_entry] = genesisEntry()
2 %genesisEntry() creates the first entry in the ledger.
3
4 % Tyson Cross 1239448
5
6     tx0 = Transaction( ...
7         {'39DF49683542CA728C04AC46C523D602541B25C1A8962E59DB639841CAB6B86A+Person1@Computer1' [0]}, ...
8         '39DF49683542CA728C04AC46C523D602541B25C1A8962E59DB639841CAB6B86A+Person1@Computer1');
9     tx0.lock();
10    genesis_entry = LedgerEntry(0, '0', datetime(737151,'ConvertFrom','datenum'), tx0,...
11        '816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7');
12 end

```

Code 15: LedgerEntry.m

```

1 classdef LedgerEntry < matlab.mixin.Copyable %handle
2     %LedgerEntry is a class for ledger entry objects, an individual entry in a simple
3     % blockchain of hash-signed transactional records.
4
5     % Tyson Cross 1239448
6
7     properties (SetAccess = private, GetAccess = private)
8         TransactionData
9     end
10    properties (SetAccess = private, GetAccess = public)
11        Index
12        PreviousHash
13        Timestamp
14        Hash
15    end
16    methods
17        function obj = LedgerEntry(index, previousHash, timestamp, data, hash)
18            %LedgerEntry() constructor, creates the ledger entry
19            obj.Index = index;
20            obj.PreviousHash = previousHash;
21            obj.Timestamp = timestamp;
22            obj.TransactionData = data;
23            obj.Hash = hash;
24        end
25        function r = getTransactionData(obj)
26            r = obj.TransactionData;
27        end
28    end
29 end

```

Code 16: Transaction.m

```

1 classdef Transaction < matlab.mixin.Copyable
2     %Transaction is a container class for transactional ledger data
3
4     % Tyson Cross 1239448
5
6     properties (SetAccess = private, GetAccess = private)
7         Data
8     end
9     properties (SetAccess = private, GetAccess = public)
10        ID_index
11    end
12    properties (NonCopyable)
13        ID_me
14        ID_recipient
15        ID_sender
16        Timestamp
17        Transaction_amount
18        Hash
19        Locked = false;
20    end
21    methods
22        function obj = Transaction(data, signature)
23            %Transaction() constructor, sets the initial financial data
24            obj.Data = data;
25            obj.ID_index = [1:numel(obj.Data)/2];
26            if isAMember(obj, signature)
27                obj.ID_me = signature;
28                setHash(obj);
29                obj.Locked = false;
30            end
31        end
32
33        function makeTransfer(obj, amount, sender, recipient, output)

```

```

34 %makeTransfer() makes a transaction between participants
35 if obj.Locked
36     error('This transaction is already completed')
37 elseif ~doesExist(obj, sender) || ~doesExist(obj, recipient)
38     disp('Invalid ID')
39 elseif (getBalance(obj, sender)-amount<=0)
40     disp('Insufficient tokens')
41 elseif ~isAMember(obj, getSignature(obj, sender))
42     disp('Not a participant, sorry');
43 elseif ~authenticateByID(obj, sender)
44     disp('Authentication failure')
45 elseif strcmp(getSignature(obj, recipient), getSignature(obj, sender))
46     disp('Invalid transfer, sender and recipient must be seperate')
47 else
48     tic;
49     obj.ID_me = getSignature(obj, sender);
50     obj.ID_sender = getSignature(obj, sender);
51     obj.ID_recipient = getSignature(obj, recipient);
52     obj.Timestamp = datetime(clock);
53     obj.Transaction_amount = amount;
54     original_balance = getBalance(obj, sender);
55     setBalance(obj, -amount, sender);
56     setBalance(obj, amount, recipient);
57     new_balance = getBalance(obj, sender);
58     setHash(obj);
59     t = toc;
60     if output
61         disp(' ')
62         disp(['Sender: ', getSignature(obj, sender)])
63         disp(['Recipient: ', getSignature(obj, recipient)])
64         disp(['Transferring ', num2str(amount), ' tokens']);
65         disp(['Balance before transfer: ', num2str(original_balance), ' tokens']);
66         disp(['New balance: ', num2str(new_balance), ' tokens']);
67         disp(['Transaction completed at ', [char(obj.Timestamp), ' in ' num2str(t), ' ...
68             seconds']])
69         disp(['Transaction ID: ', obj.Hash]);
70     end
71     obj.lock();
72 end
73 end
74
75 function r = getBalance(obj, id)
76     r = obj.Data{id, 2};
77 end
78
79 function setBalance(obj, value, id)
80     if obj.Locked
81         error('This transaction is already completed')
82     else
83         obj.Data{id, 2} = obj.Data{id, 2} + value;
84         setHash(obj);
85     end
86 end
87
88 function r = getSignature(obj, id) % retrieve hash of private key
89     r = obj.Data{id, 1};
90 end
91
92 function r = getTotalBalance(obj) % total token in system
93     val = 0;
94     for i=1:length(obj.Data)
95         val = val + obj.Data{i, 2};
96     end
97     r = val;
98 end
99
100 function r = isAMember(obj, signature)
101     if ismember(signature, {obj.Data{:, 1}})>0
102         r = true;
103     else
104         r = false;
105     end
106 end
107
108 function r = doesExist(obj, id)
109     if ismember(id, obj.ID_index)>0
110         r = true;
111     else
112         r = false;
113     end
114 end
115
116 function setHash(obj) % sets hash of transaction (for ledger hashing)
117     if obj.Locked

```

```

118         error('This transaction is already completed')
119     else
120         a = strcat(...
121             [char(obj.Data{:,1})],...
122             [char(obj.Data{:,2})],...
123             [char(obj.ID_me)],...
124             [char(obj.ID_recipient)],...
125             [char(obj.ID_sender)],...
126             [char(obj.Timestamp)],...
127             [num2str(obj.Transaction_amount) ]);
128         a = a(~isspace(a(:)));
129         obj.Hash = hash(a);
130     end
131 end
132
133 function r = authenticateByID(obj, id) % simulates comparing private key ...
134     hash (by ID)
135     if ~strcmp(getSignature(obj,id),obj.ID_me)
136         r = false;
137     else r = true;
138     end
139 end
140
141 function r = authenticateBySignature(obj, signature) % simulates comparing private key hash
142     if ~strcmp(signature,obj.ID_me)
143         r = false;
144     else r = true;
145     end
146 end
147
148 function lock(obj)
149     obj.Locked = true;
150 end
151
152 % methods (Access = protected)
153 function alterIdentity(obj, id) % Demo function to change ID
154     obj.ID_me = getSignature(obj,id);
155     setHash(obj);
156 end
157
158 function removeByID(obj,id) % for bad nodes
159     if doesExist(id) && authenticateByID(obj,signature)
160         disp(['Removing ',obj.Data(id,:)])
161         obj.ID_index(id) = [];
162         obj.Data(id,:) = [];
163         setHash(obj);
164     else
165         disp('ID not found')
166     end
167 end
168
169 function removeBySignature(obj,signature) % for bad nodes
170     [~, pos] = ismember(signature,obj.Data(:,1));
171     if pos>0 && authenticateBySignature(obj,signature)
172         disp(['Removing ', obj.Data(pos,:)])
173         obj.ID_index(pos) = [];
174         obj.Data(pos,:) = [];
175         setHash(obj);
176     else
177         disp('ID not found')
178     end
179 end
180
181 end
182
183 end
184

```

Code 17: createLedgerEntry.m

```

1 function [ ledger_entry ] = createLedgerEntry( transaction, ledger )
2 %createLedgerEntry() creates a new ledger entry, retrieving the previous ledger entry's data
3
4 % Tyson Cross 1239448
5
6 previous_ledger_entry = getLatestLedgerEntry(ledger);
7 next_index = previous_ledger_entry.Index + 1;
8 next_timestamp = datetime(clock);
9 next_hash = calculateHash( next_index,...
10                             previous_ledger_entry.Hash,...
11                             next_timestamp,...
12                             transaction.Hash);

```

```

13 ledger_entry = LedgerEntry(next_index, previous_ledger_entry.Hash, next_timestamp, transaction, ...
    next_hash);
14
15 end

```

Code 18: addEntry.m

```

1 function [ ledger ] = addEntry( new_entry, ledger )
2 %addEntry() adds a new entry to the ledger
3
4 % Tyson Cross 1239448
5
6 if isEntryValid(new_entry, getLatestLedgerEntry(ledger))
7     ledger = [ledger new_entry];
8     ledger(end).getTransactionData().lock();
9 end
10
11 end

```

Code 19: calculateEntryHash.m

```

1 function [ hash_value ] = calculateEntryHash( entry )
2 %calculateHash() is a wrapper for the SHA-256 hash function to calculate ledger entries' hashes
3
4 % Tyson Cross 1239448
5
6 hash_value = calculateHash( entry.Index, entry.PreviousHash, datenum(entry.Timestamp), ...
    entry.getTransactionData().Hash );
7
8 end

```

Code 20: calculateHash.m

```

1 function [ hash_value ] = calculateHash( index, previous_hash, timestamp, datahash )
2 %calculateHash calculates the hash of a new ledger entry using an implementation of SHA-256
3
4 a = [char(num2str(index)), ...
5     [char(previous_hash)], ...
6     [char(timestamp)], ...
7     [char(datahash)] ];
8
9 a = char(a(~isspace(a(:)))));
10
11 hash_value = hash(a);
12
13 end

```

Code 21: createNewTransactionFromLastRecord.m

```

1 function [ new_transaction ] = createNewTransactionFromLastRecord( ledger, id)
2 %createTransaction retrieves the record of the last transaction
3
4 temp = getLatestLedgerEntry( ledger );
5 new_transaction = copy(temp.getTransactionData());
6 new_transaction.alterIdentity(id);
7
8 end

```

Code 22: getLatestLedgerEntry.m

```

1 function [ ledger_entry ] = getLatestLedgerEntry( ledger )
2 %getLatestLedgerEntry() returns the handle to the last ledger class object
3 % in the ledger entry.
4
5 % Tyson Cross 1239448
6
7 ledger_index = numel(ledger);
8 ledger_entry = ledger(ledger_index);
9
10 end

```

Code 23: isEntryValid.m


```

1 function [ value ] = isEntryValid( new_entry, last_entry )
2 %checkEntryValid() checks the latest ledger entry for validity
3
4 % Tyson Cross 1239448
5
6 value = false;
7 if new_entry.Index ≠ last_entry.Index()+1
8     disp('Invalid chain, index is not consistent')
9 elseif new_entry.PreviousHash ≠ last_entry.Hash
10     disp('Invalid chain, hash values do not match')
11 elseif calculateEntryHash(new_entry) ≠ new_entry.Hash
12     disp('Invalid hash!')
13 else
14     value = true;
15 end
16
17 end

```

Code 24: replaceLedger.m

```

1 function [ LEDGER ] = replaceLedger( new_ledger, current_ledger)
2 %replaceLedger() Replaces the old ledger with a new ledger, if the length is longer
3
4 if isValidLedger(new_ledger) && (length(new_ledger) > length(current_ledger))
5     LEDGER = new_ledger;
6     disp('New ledger is valid. Replacing old ledger with newly received ledger');
7 else
8     LEDGER = current_ledger;
9     disp('Received ledger invalid. Retaining original ledger. ');
10 end
11
12 end

```

UTILITY FUNCTION CODE

Code 25: bin2decimal.m

```

1 function [ decimal ] = bin2decimal( bin )
2 %bin2decimal() converts a binary value to an unsigned decimal value
3 % Accurate conversion from 0 to intmax('uint64') is supported, binary
4 % values from 8 to 64 bits.
5
6 % Tyson Cross 1239448
7
8 if numel(bin)>64
9     error('Input cannot be greater than 64-bits')
10 end
11
12 if ischar(bin)
13     bin = logical(bin(:)('-'0'));
14 end
15 value = uint64(length(bin)-1:-1:0);           % Array of exponents from binary entries
16 base = uint64(2).^value;                     % Decimal values for each bit
17 decimal = sum(base.*uint64(bin), 'native');   % Sum the entries
18
19 end

```

Code 26: dec2binary.m

```

1 function [ bin ] = dec2binary( decimal, num_of_bits )
2 %dec2binary() converts an unsigned decimal value to a fixed length binary value
3 % if the bitdepth is not specified, then an appropriate fixed length is
4 % chosen, or 64,32,16 or 8 bits. Accurate conversion from 8 to
5 % 64-bits is supported, for decimal values from 0 to intmax('uint64').
6
7 % Tyson Cross 1239448
8
9 if decimal > intmax('uint64')
10     error('Input cannot be greater than intmax(''utint64'')')
11 end
12
13 if nargin<2
14     if decimal > intmax('uint32')
15         num_of_bits = 64;
16     elseif decimal > intmax('uint16')
17         num_of_bits = 32;
18     elseif decimal > intmax('uint8')
19         num_of_bits = 16;
20     else
21         num_of_bits = 8;
22     end
23 end
24
25 decimal = uint64(decimal);
26 value = uint64(num_of_bits-1:-1:0);           % Array of exponents for binary entries
27 base = uint64(2).^value;                     % Decimal values for each bit
28
29 if decimal > sum(uint64(base), 'native')
30     error('Not enough bits specified to represent decimal value')
31 end
32
33 bin = false(1,num_of_bits);                  % Initialise logical array
34 for i=1:num_of_bits
35     if decimal >= base(i)                     % For each applicable column of 2^i
36         decimal = decimal - base(i);         % Reduce the value of decimal
37         bin(i) = true;                       % Set the binary bit
38     end
39 end
40
41 end

```

Code 27: char2logical.m

```

1 function [ logical_array ] = char2logical( char_array )
2 %char2logical() converts a char array (assumed to be of '0' and '1' chars) to a logical array
3
4 % Tyson Cross 1239448
5
6 if ~ischar(char_array)
7     error('Input must be a char array')
8 end
9
10 logical_array = logical(char_array(:)('-'0'));

```

```
11
12 end
```

Code 28: logical2char.m

```
1 function [ char_array ] = logical2char( logical_array )
2 %logicaltochar() converts a logical array into a char array
3
4 % Tyson Cross 1239448
5
6 if ~islogical(logical_array)
7     error('Input must be a char array')
8 end
9
10 char_array = repmat('0',1,numel(logical_array));
11 for i=1:numel(logical_array)
12     if logical_array(i)
13         char_array(i)='1';
14     else
15         char_array(i)='0';
16     end
17 end
18 % char_array = fprintf('%d', logical_array);
19
20 end
```

Code 29: logical2str.m

```
1 function [ str ] = logical2str( logical_array )
2 %str2logical() Converts a logical array to a string based on the char ascii values
3 % The array is assumed to be a padded logical array, with the following structure:
4 % [ logical_array 1 padded_zeros length_of_message ]
5
6 % Tyson Cross 1239448
7
8 if ~islogical(logical_array)
9     error('Input must be a logical array')
10 end
11
12 bin_array = reshape(logical_array',1,[]); % flatten into single row
13 len = bin_array(end-63:end); % extract message length
14 len_dec = bin2decimal(len);
15 bin_array = bin_array(1:len_dec); % trim padding
16 % bin_array = bin_array(1:floor(numel(bin_array)/7)*7); % <- not be required if message ...
17 raw_decoded_msg = reshape(bin_array, 7, []); % parse for decoding individual ...
18 str = char(bin2dec(num2str(raw_decoded_msg))); % decode into ASCII
19
20 end
```

Code 30: str2logical.m

```
1 function [ str ] = logical2str( logical_array )
2 %str2logical() Converts a logical array to a string based on the char ascii values
3 % The array is assumed to be a padded logical array, with the following structure:
4 % [ logical_array 1 padded_zeros length_of_message ]
5
6 % Tyson Cross 1239448
7
8 if ~islogical(logical_array)
9     error('Input must be a logical array')
10 end
11
12 bin_array = reshape(logical_array',1,[]); % flatten into single row
13 len = bin_array(end-63:end); % extract message length
14 len_dec = bin2decimal(len);
15 bin_array = bin_array(1:len_dec); % trim padding
16 % bin_array = bin_array(1:floor(numel(bin_array)/7)*7); % <- not be required if message ...
17 raw_decoded_msg = reshape(bin_array, 7, []); % parse for decoding individual ...
18 str = char(bin2dec(num2str(raw_decoded_msg))); % decode into ASCII
19
20 end
```

Code 31: flatten.m

```

1 function C = flatten(A)
2 C = {};
3 for i=1:numel(A)
4     if (~iscell(A{i}))
5         C = [C,A{i}];
6     else
7         Ctemp = flatten(A{i});
8         C = [C,Ctemp{:}];
9     end
10
11 end
12 C = squeeze(C(:));
13 C = C(~isspace(C(:)));
14 end

```

Code 32: flattenlogical.m

```

1 function [ output ] = flattenlogical( input )
2 %flattenlogical() Reduces a 1x1xW logical array to a 1xW logical array
3
4 % Tyson Cross 1239448
5
6 output = char2logical(logical2char(input));
7
8 end

```

Code 33: loadPeople.m

```

1 function [ participants ] = loadPeople( num_of_participants)
2 %loadPeople() Creates simple signatures for each participant
3
4 if nargin <1
5     num_of_participants = 9;
6 end
7
8 for i=1:num_of_participants
9     id_val = strcat({'id_'},num2str(i),'.pub');
10    if i==1
11        participants{i,1} = ...
12            ['39DF49683542CA728C04AC46C523D602541B25C1A8962E59DB639841CAB6B86A+Person1@Computer1'];
13    else
14        participants{i,1} = [hash(char(randi(1000))) '+Person', num2str(i), '@Computer', num2str(i)];
15    end
16    participants{i,2} = randi([1000 10000]);
17 end
18 end

```

Code 34: randomTransactionParameters.m

```

1 function [ amount, sender, reciever ] = randomTransactionParameters( transaction )
2 %randomTransactionParameters() sets up a random transfer of tokens
3
4 N = numel(transaction.ID_index);
5 sender = randi(N);
6 reciever = randi(N);
7 while sender==reciever
8     sender = randi(N);
9 end
10 while reciever==sender
11     reciever = randi(N);
12 end
13 amount = randi(100);
14
15 end

```

Code 35: checkLength.m

```

1 function checkLength(input, len)
2 %checkLength() checks the input length
3
4 % Tyson Cross 1239448
5
6 if ~length(input)==len
7     error(['Length is not ',num2str(len)])
8 end
9 end

```

```
1 function checkLogical(input)
2 %checkLogical() checks the input is of type logical
3
4 % Tyson Cross 1239448
5
6     if ~islogical(input)
7         error('Input must be logical array')
8     end
9 end
```

TEST CODE

Code 37: testConversionUtilities.m

```
1 % Tyson Cross 1239448
2 clc; clear all;
3
4 % conversion utilities tests
5
6 value_64 = intmax('uint64')/2;
7 bin_64 = dec2binary(value_64);
8 decimal_64 = bin2decimal(bin_64);
9 assert(isequal(decimal_64,value_64));
10
11 value_32 = intmax('uint32')/2;
12 bin_32 = dec2binary(value_32);
13 decimal_32 = bin2decimal(bin_32);
14 assert(isequal(decimal_32,value_32));
15
16 value_16 = intmax('uint16')/2;
17 bin_16 = dec2binary(value_16);
18 decimal_16 = bin2decimal(bin_16);
19 assert(isequal(decimal_16,value_16));
20
21 value_8 = intmax('uint8')/2;
22 bin_8 = dec2binary(value_8);
23 decimal_8 = bin2decimal(bin_8);
24 assert(isequal(decimal_8,value_8));
25 disp('dec2binary() and bin2decimal() passed basic functionality tests')
26
27
28 try bin_test = dec2binary(value_test,32);
29 catch ME
30     disp([ 'dec2binary() correctly threw an error (due to insufficient bitsize)' ])
31 end
32
33 try bin_test = dec2binary(18446744073709551616);
34 catch ME
35     disp([ 'dec2binary() correctly threw an error (due to a decimal value being too large)' ])
36 end
37
38 try decimal_test = bin2decimal(true(1,65));
39 catch ME
40     disp([ 'bin2decimal() correctly threw an error (due to a binary value being too large)' ])
41 end
42
43 % prompt = 'Enter a decimal value to convert to binary:\n';
44 % value = input(prompt);
45 value = randi(intmax('uint32'));
46 bin = dec2binary(value);
47 decimal = bin2decimal(bin);
48 assert(isequal(decimal,value));
49 % fprintf('%d', bin)
50
51 disp('-----')
52 disp('All tests passed for dec2binary() and bin2decimal()')
53 disp('-----')
54 disp(' ')
55
56 bin_array = dec2binary(intmax('uint32')/2);
57 char_array = logical2char(bin_array);
58 bin_array2 = char2logical(char_array);
59 char_array2 = logical2char(bin_array);
60 assert(isequal(char_array,char_array2));
61 assert(isequal(bin_array,bin_array2));
62 disp('logical2char() and char2logical() passed basic functionality tests')
63
64
65 try char_array = logical2char(char_array2);
66 catch ME
67     disp([ 'logical2char() correctly threw an error (due to invalid input type)' ])
68 end
69
70 try bin_array = char2logical(bin_array2);
71 catch ME
72     disp([ 'char2logical() correctly threw an error (due to invalid input type)' ])
73 end
74 disp('-----')
75 disp('All tests passed for logical2char() and char2logical()')
76 disp('-----')
77 disp(' ')
```

```

1  % Tyson Cross 1239448
2
3  clc; clear all;
4
5  message = [ 'And above all, watch with glittering eyes the whole world around you '...
6             'because the greatest secrets are always hidden in the most unlikely '...
7             'places. Those who don't believe in magic will never find it. '];
8
9  total_time = 0;
10 len = numel(message);
11 fprintf('Hashing... \n')
12
13 for i=1:len;
14     tic;
15     message = message(1:end-1);
16     fprintf('%s\n',message)
17     hash_value(i,:) = hash(message);
18     t = toc;
19     total_time = total_time + t;
20     assert(length(hash_value(i,:))==64)
21     if i > 1
22         dif_hash(i,:) = (hash_value(i,:) == hash_value(i-1,:));
23     end
24 end
25
26 clc;
27 fprintf('Hash values: \n')
28 disp(hash_value)
29 fprintf('Collision Map: \n')
30 for i=1:length(dif_hash)
31     disp(logical2char(dif_hash(i,:)))
32 end
33
34 % I = mat2gray(dif_hash)
35 % rez = get(groot,'ScreenSize');
36 % figure('Position',[rez(4)/2 rez(3)/2 rez(4)/4 rez(4)/2])
37 % imshow(I,'InitialMagnification','fit');
38
39 disp(['Average time to hash: ', num2str(total_time/len)]);
40 disp(['Average bit-wise sequential collision probability: ', ...
41       num2str(sum(sum(dif_hash))/numel(dif_hash))]);
42
43 clear message;

```

REFERENCES

- [1] L. Hartikka, "NaiveChain: A Blockchain in 200 lines of code," <https://github.com/lhartikk/naivechain>.
- [2] Bitcoin.org, "Bitcoin github repository," Bitcoin <https://github.com/bitcoin/bitcoin>.

DECLARATION

TABLE I: Division of Work

Name	Student	Code	Design
Tyson Cross	1239448	100%	100%

TABLE II: Project Milestones

Milestone	Achieved
Research	✓
Initial system design	✓
Utilities	✓
SHA-256	✓
Transactions	✓
Network/Node Broadcast	X
Coding	✓
Testing	✓
Report	✓

Signed 