SCHOOL OF ELECTRICAL ENGINEERING, UNIVERSITY OF THE WITSWATERSRAND

ELEN3015 - Data and Information Management

Lecturer: Ling Cheng

Lab 4 Decentralized Digital Token Payment System

17 April 2018

Tyson Cross 1239448

***Abstract*— This report details the design and development of a prototype digital payment system and analyses various attacks against the system. The payment system uses a implementation of SHA-256 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. 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 commercial transactions with e-commerce, to global disruption of the global banking system. A historic hostility to the academic threat to the financial hegemony of the international banking system has given way to a tacit acceptance of existing early but fervently-adopted commercial implementations such as Bitcoin, Etherium, Ripple, Stellar, Cardano and a multiple of others. The innovative use of a computational hashing to provide proof-of-work as implemented by Bitcoin was instrumental in the emergence of these early currencies. The energy-inefficiency of such schemes has been widely criticised as wasteful, although 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 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 SHA-256

Hashing algorithms are designed to cryptographically hash an input message of any length, producing a fixed-length digest that can be used as an identifying fingerprint of the message, while being computationally infeasible to recover the original message. There are three primary standards in the FIPS specification, 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), designed and specified by the NSA in the United States, is a global standard[1] 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.

1.1 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.

$$l + 1 + k \equiv 448 \text{ mod } 512 \tag{1}$$

Finally, 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.

1.2 Encipherment

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

$$Ch(x, y, x) = (x \wedge y) \oplus (\neg x \wedge z) \tag{2}$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{3}$$

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

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

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

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

These are implemented in MATLAB functions in the solution, as shown in Code Listings 3-8 in Appendix I, according to the FIPS180-4 standard[1].

2 CRYPTOCURRENCIES

Early designs and implementations in online currencies emphasised anonymity and untraceability [2], [3] as core features, with a later focus on distribution and peer-to-peer mechanisms of decentralised control. Influential early systems like 'Millicent' and 'Payword'[4] described workable schemes around minting tokens and online micro-transactions. There was much research into computationally light or efficient encryption schemes, such as [5] 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[6]. Any system which is practical must ensure that a token, or coin, once spent, cannot be re-used for a separate transaction. Another problem is the fraudulent minting of new tokens, which is difficult in a distributed network. Confirming the validity of new tokens without a central authority, or fiat-issuance of a controlled physical manifestation of a means of exchange presents a challenge. [7] suggested a system based on computational encryption that still relied on a central banking authority, as did [4] who defined different levels of credit and transactional interaction, between brokers, vendors and users. The notion of a distributed, verifiable record of all transactions became the primary idea behind a truly decentralised system of exchange[8], with the publication of systems of irrefutable, blind signatures and anonymous communication[9]. The development of asymmetric public/private key encryption such as RSA [10] 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.

2.1 Bitcoin

In 2008, the Bitcoin white paper was published[11], proposing a solution to the problem of double-spending, based on computationally infeasibility to reverse any transaction. Trust between parties (or with larger corporate or governmental bodies that traditionally dominate national and global financial control with 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 problem 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 [12].

2.2 Blockchain

Transactional data is hashed with timestamps, which are themselves hashed along with proof of computationally intensive processes (using the SHA-256 algorithm, attempting 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.

2.3 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, 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[13].

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[14]. It is argued 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.

3 Design

The proposed prototype payment system is designed around two main entities, the classes `Transaction` and `LedgerEntry`, which constitute the data container for individual token exchange transactions, and the chain of securely hashed blocks that these transactions are stored within respectively. After an initial, hard-coded genesis block is created as the first entry in the public ledger, the initial distribution of available tokens are published with the pool of participants. 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.

Each transaction can be of any discrete amount currently contained within the pool of individually available tokens tied to a cryptographic key. There is no allowance for credit directly in the system, so users can only spend the tokens that they can prove they possess, unable to go below a positive balance. Once a transaction is performed, it is hashed with a timestamp and the sender and recipients' public keys, and 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 is passed, added into the Ledger and a new entry. The ledger would then be published in a non-implemented broadcast to all available nodes. competing 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 when compared.

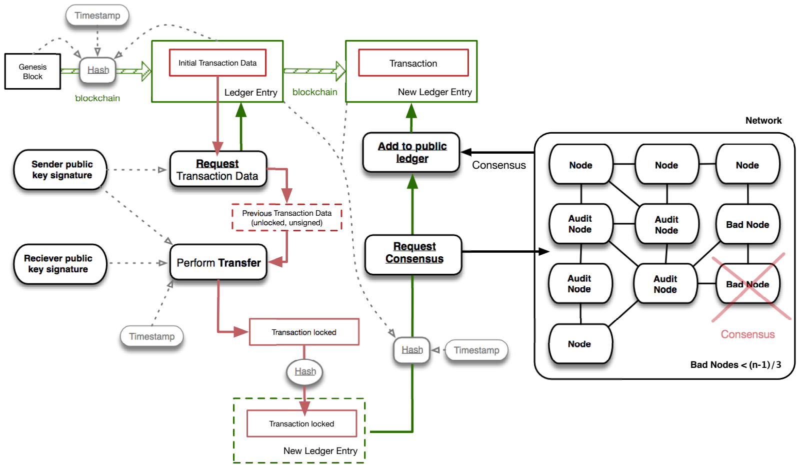

Fig. 1: Basic transaction workflow

3.1 Advantages

The system proposed is simple to understand and implement, and 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 new coins into the supply, leading to over-valuation. This system is currently only suitable for small-scale micro-transactions, and requires significant modifications to enable sustainable growth in the 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 proposed by [15], a threshold of no more than $^{(n1)}/_{3}$ 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 encourages rapid vote-casting for ledger acceptance and participation. 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 bad faith and deception. The criteria for inclusion in the inner network of trusted nodes limits the pure decentralisation of the system as a trade-off. The auditing nodes are also responsible for the mining of new tokens, based on the value of the computational processes performed. Performing "useful work" for scientific or research purposes that leads to beneficial outcomes for a larger collective outside the network of participants encourages 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 mitigates.

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

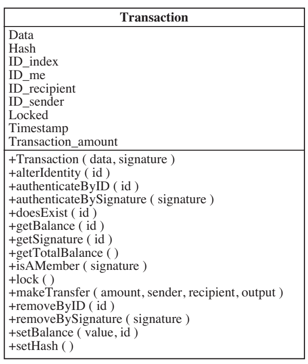

Fig. 2: The Transaction Class

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 must be a current participant, have a valid authenticated key hash, and is only able to transfer their own tokens up the the value of the 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 Ledger entry 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 Ledger entry 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. E. Standard *et al.*, "Federal information processing standards publication 46," *National Bureau of Standards, US Department of Commerce*, vol. 4, 1977.

[2] D. Chaum, "Security without identification: Transaction systems to make big brother obsolete," *Communications of the ACM*, vol. 28, no. 10, pp. 1030–1044, 1985.

[3] T. Okamoto and K. Ohta, "Universal electronic cash," in *Annual International Cryptology Conference*. Springer, 1991, pp. 324–337.

[4] R. L. Rivest and A. Shamir, "Payword and micromint: Two simple micropayment schemes," in *International workshop on security protocols*. Springer, 1996, pp. 69–87.

[5] T. Okamoto, "An efficient divisible electronic cash scheme," in *Annual International Cryptology Conference*. Springer, 1995, pp. 438–451.

[6] 21st Century Coin Company Inc., "HashCash," http://hashcash.com/about.html.

[7] D. Chaum, A. Fiat, and M. Naor, "Untraceable electronic cash," in *Conference on the Theory and Application of Cryptography*. Springer, 1988, pp. 319–327.

[8] K. Q. Nguyen, Y. Mu, and V. Varadharajan, "Digital coins based on hash chain," in *National Information Systems Security Conference*, 1997.

[9] D. Chaum, "Blind signatures for untraceable payments," in *Advances in cryptology*. Springer, 1983, pp. 199–203.

[10] R. L. Rivest, A. Shamir, and L. M. Adleman, "Cryptographic communications system and method," September 1983, US Patent 4,405,829.

[11] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[12] A. Back *et al.*, "Hashcash-a denial of service counter-measure," 2002.

[13] B. L. A. L. Make and B. Laurie, "Decentralised currencies are probably impossible."

[14] 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.

[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

```
function [ digest ] = hash( message )
%hash() hashes an input message with an implementation of SHA256
%
% Input message is padded and split into 512-bit blocks.
% Output is a 256-bit hexidecimal digest (char)
% See FIPS PUB 180-4 for the implementation standard
% http://dx.doi.org/10.6028/NIST.FIPS.180-4

% Tyson Cross 1239448

% Initialise working register arrays
a = false(1,32);
b = false(1,32);
c = false(1,32);
d = false(1,32);
e = false(1,32);
f = false(1,32);
g = false(1,32);
h = false(1,32);

word_length  = 32;
mod_value = 2^32;

% K is first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers
K1 = dec2bin(hex2dec(...
    [   '428a2f98';'71374491';'b5c0fbcf';'e9b5dba5';'3956c25b';'59f111f1';'923f82a4';'ab1c5ed5';...
        'd807aa98';'12835b01';'243185be';'550c7dc3';'72be5d74';'80deb1fe';'9bdc06a7';'c19bf174';...
        'e49b69c1';'efbe4786';'0fc19dc6';'240ca1cc';'2de92c6f';'4a7484aa';'5cb0a9dc';'76f988da';...
        '983e5152';'a831c66d';'b00327c8';'bf597fc7';'c6e00bf3';'d5a79147';'06ca6351';'14292967';...
        '27b70a85';'2e1b2138';'4d2c6dfc';'53380d13';'650a7354';'766a0abb';'81c2c92e';'92722c85';...
        'a2bfe8a1';'a81a664b';'c24b8b70';'c76c51a3';'d192e819';'d6990624';'f40e3585';'106aa070';...
        '19a4c116';'1e376c08';'2748774c';'34b0bcb5';'391c0cb3';'4ed8aa4a';'5b9cca4f';'682e6ff3';...
        '748f82ee';'78a5636f';'84c87814';'8cc70208';'90befffa';'a4506ceb';'bef9a3f7';'c67178f2']));

% H is first 32 bits of the fractional parts of the square roots of the first 8 prime numbers
H1 = dec2bin(hex2dec(...
    [ '6a09e667'; 'bb67ae85'; '3c6ef372'; 'a54ff53a'; '510e527f'; '9b05688c'; '1f83d9ab'; '5be0cd19' ...
        ]));

K = false(64,32);
for i=1:64
    K(i,:) = char2logical(K1(i,:));
    assert(strcmp(logical2char(K(i,:)),K1(i,:)));
end

H = false(1,8,32);
for i=1:8
    H(1,i,:) = char2logical(H1(i,:));
    assert(strcmp(logical2char(H(1,i,:)),H1(i,:)));
end

% Message preprocessing
if ¬islogical(message)
    if ischar(message)
        message_logical = str2logical(message);
    elseif isnumeric(message)
        message_logical = logical(message);
    end
end

% Pad to make the message a multiple of 512
message_padded = padder(message_logical);

% Parse into n blocks of 512 bits, then reshape into 3D matrix (Nx16x32)
message_flat = reshape(message_padded,512,[])';
[N, ¬] = size(message_flat);
[row,col] = size(message_flat');
M = permute(reshape(message_flat',[word_length,row/word_length,col]),[3,2,1]);

% Initialise W (word schedule)
[x,y,z] = size(M);
W = false(N*x,y,z);

clear x y z row col message_padded message_flat message_logical K1 H1 input ;

% Process message blocks (total of N message blocks)
% Matlab indexing starts at 1, unfortunately, so H(0) must be treated as H(1)
```

```
for i=1:N; j=i+1;

    % Hash computation, Stage 1
    for t=1:16
        W(i,t,:) = M(i,t,:);
    end

    for t=17:64
        alpha = sigma_1(flattenlogical(W(i,t-2,:)));
        beta = flattenlogical(W(i,t-15,:));
        Δ = sigma_0(flattenlogical(W(i,t-15,:)));
        gamma = flattenlogical(W(i,t-16,:));
        epsilon = mod_addition(alpha, beta, Δ, gamma);
        W(i,t,:) = flattenlogical(epsilon);
        clear alpha beta gamma Δ eplison;
    end

    % Hash computation, Stage 2
    a = flattenlogical(H(i,1,:));
    b = flattenlogical(H(i,2,:));
    c = flattenlogical(H(i,3,:));
    d = flattenlogical(H(i,4,:));
    e = flattenlogical(H(i,5,:));
    f = flattenlogical(H(i,6,:));
    g = flattenlogical(H(i,7,:));
    h = flattenlogical(H(i,8,:));

    % Hash computation, Stage 3
    for t=1:64
        alpha = h;
        beta = E_1(e);
        Δ = Ch(e,f,g);
        gamma = flattenlogical(K(t,:));
        epsilon = flattenlogical(W(i,t,:));
        T_1 = mod_addition(alpha, beta, Δ, gamma, epsilon);
        T_2 = mod_addition(E_0(a), Maj(a,b,c));
        h = g;
        g = f;
        f = e;
        e = mod_addition(d, T_1);
        d = c;
        c = b;
        b = a;
        a = mod_addition(T_1, T_2);
        clear alpha beta gamma Δ epsilon;
    end

    % Hash computation, Stage 2 - H(jth) intermediate hash value
    H(j,1,:) = mod_addition(a, flattenlogical(H(i,1,:)));
    H(j,2,:) = mod_addition(b, flattenlogical(H(i,2,:)));
    H(j,3,:) = mod_addition(c, flattenlogical(H(i,3,:)));
    H(j,4,:) = mod_addition(d, flattenlogical(H(i,4,:)));
    H(j,5,:) = mod_addition(e, flattenlogical(H(i,5,:)));
    H(j,6,:) = mod_addition(f, flattenlogical(H(i,6,:)));
    H(j,7,:) = mod_addition(g, flattenlogical(H(i,7,:)));
    H(j,8,:) = mod_addition(h, flattenlogical(H(i,8,:)));

    for k=1:8
        H_N{k} = dec2hex(bin2decimal(logical2char(H(j,k,:))),8);
    end
end

digest = strjoin(H_N,'');

end
```

Code 2: padder.m

```
function [ M ] = padder( input )
%padder() ensures that a message length is a multiple of 512
% The logical array is appended with a single '1' bit, then
% padded with 448 zeros, then the length of the message in binary:
%   [ logical_array 1 padded_zeros length_of_message ]

% Tyson Cross 1239448

if islogical(input)
    msg = input;
elseif ischar(input)
    msg = str2logical(input);
elseif isnumeric(input)
    msg = str2logical(num2str(input));
end
```

```
len = length(msg);
bin_len = dec2binary(len,64);                 % 64 bits

k = 0:511;                                    % Brute force find the value of the zero padding
zero_len = k(mod(len + 1 + k, 512) == 448); % to solve the equation len + 1 + k = 448 mod 512
zero_pad = false(1,zero_len);                 % Number of zeros
M = [msg true zero_pad bin_len];              % append the message length in binary (64-bits)
assert(mod(numel(M),512)==0);

end
```

Code 3: Maj.m

```
function [ word ] = Maj(x,y,z)
%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

```
function [ word ] = Ch(x,y,z)
%Ch() implements (x^y)XOR(Âňx^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

```
function [ word ] = E_0(x)
%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

```
function [ word ] = E_1(x)
%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

```
function [ word ] = sigma_0(x)
%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

```
function [ word ] = sigma_1(x)
%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

```
function [word] = ROTL(x,n)
%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

```
function [word] = ROTR(x,n)
%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

```
function [word] = SHR(x,n)
%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

```
function [output] = mod_addition(varargin)
%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(varargini);endoutput =
    dec2binary(mod(value,modvalue),32);end
```

Code 13: PaymentSystem.m[1]

```
% Structural ideas based on the broad concept of Bitcoin https://github.com/bitcoin/bitcoin
% with some implementation design from Naivechain https://github.com/lhartikk/naivechain

addpath('SHA256','utilities','Blockchain'); clc; clear all;

%% Setup
global LEDGER;

% Generate a basic system of participants and pre-existing tokens
people = loadPeople(9);

% My identity
my_Signature = people{1:1};
my_ID = 1;

% create a simple blockchain
% ideally a tree or linked list structure, for now a simple vector array
LEDGER = [genesisEntry()];
disp('Created a new ledger')

disp(' ')
disp('Initialise the transaction data, put the available people and tokens into the ledger...')
tx2 = Transaction(people, my_Signature);
lg2 = createLedgerEntry(tx2,LEDGER);
LEDGER = addEntry(lg2, LEDGER);

%% Output
disp(' ')
disp(' ID                              Signature                                                    ...
    Tokens')
for i=1:numel(people)/2
    disp([' ', num2str(i), '    ', char(people{i,1}), '      ', num2str(people{i,2})])
end
original_tokens = LEDGER(end).getTransactionData.getTotalBalance;
disp(['Total: ', num2str(original_tokens) ' Tokens'])
clear people tx2;

%% Transactions
disp(' ')
disp('Perform a transaction as another person (for demonstration)...')
tx3 = createNewTransactionFromLastRecord(LEDGER, my_ID);
[ amount, sender, reciever  ] = randomTransactionParameters(tx3);
tx3.alterIdentity(sender);
tx3.makeTransfer(amount, sender, reciever, true);
lg3 = createLedgerEntry(tx3,LEDGER);
LEDGER = addEntry(lg3, LEDGER);
clear tx3;

%% Attacks
disp(' ')
disp('Attempting attacks on the ledger')
disp('-------')
disp(' ')
disp('Attempt to repeat a previous transaction...')
% will fail because of an invalid index check
LEDGER = addEntry(lg3, LEDGER);
clear lg3;

disp(' ')
disp('Attempt to add an invalid entry, favouring Person1@Computer1...')
% transfer 1000 tokens from another person to myself
% will fail because of an authentication check
tx_invalid = createNewTransactionFromLastRecord(LEDGER, my_ID);
tx_invalid.alterIdentity(my_ID);
tx_invalid.makeTransfer(1000, 2, 1, true);
clear tx_invalid;

disp(' ')
disp('Attempt to transfer more tokens than current balance...')
tx_overspend = createNewTransactionFromLastRecord(LEDGER, my_ID);
tx_overspend.alterIdentity(my_ID);
amount = tx_overspend.getBalance(my_ID) + 1000;
tx_overspend.makeTransfer(amount, my_ID, 2, true);
clear tx_overspend;

disp(' ')
disp('Attempt to manipulate the token balance...')
tx_cheat = createNewTransactionFromLastRecord(LEDGER, my_ID);
tx_cheat.alterIdentity(my_ID);
amount = tx_cheat.getBalance(my_ID) + 10000;
```

```
try tx_cheat.Data(my_ID,2) = amount;
catch
    disp('The Data property of Transaction is not accessible')
end
clear tx_cheat;

disp(' ')
disp('Attempt to insert an forged transaction into the ledger...')
lg_insert = copy(LEDGER(end));
tx_insert = copy(lg_insert.getTransactionData);
tx_insert.alterIdentity(2);
tx_insert.makeTransfer(10, 2, 1, false);
lg_insert_fake = createLedgerEntry(tx_insert,LEDGER);
try lg_insert.Index =lg_insert_fake.Index;
catch
    disp('Unable to alter the index, relevant properties of the class are read-only')
end
try LEDGER = addEntry(lg_insert, LEDGER);
catch
    disp('Unable to add ')
end
clear tx_insert lg_insert lg_insert_fake;

disp(' ')
disp('Simple attempt to corrupt the ledger...')
tx_corrupt = createNewTransactionFromLastRecord(LEDGER, randi(9));
lg_corrupt = createLedgerEntry(tx_corrupt,LEDGER);
try LEDGER = addEntry(tx_corrupt, LEDGER);
catch
    if isValidLedger(LEDGER)
        disp('Causes an error, but the ledger is unaffected.')
    else
        disp('The ledger has been corrupted')
    end
end
clear tx_corrupt lg_corrupt;

% Make a fork of the ledger...
ledger_competing = copy(LEDGER);

disp(' ')
disp('Perform some transactions to grow the ledger... (ouput disabled for brevity)')
for i=4:6
    tx_new = createNewTransactionFromLastRecord(LEDGER, my_ID);
    [ amount, sender, reciever  ] = randomTransactionParameters(tx_new);
    tx_new.alterIdentity(sender);
    tx_new.makeTransfer(amount, sender, reciever, false);
    lg_new = createLedgerEntry(tx_new,LEDGER);
    LEDGER = addEntry(lg_new, LEDGER);
    fprintf('.')
end
disp(' ')
clear lg_new tx_new;

disp(' ')
disp('Attempt to replace the ledger with a shorter, competing ledger...')
LEDGER = replaceLedger(ledger_competing, LEDGER);

disp(' ')
disp('Perform more transactions on the competing ledger... (ouput disabled for brevity)')
for i=4:8
    tx_competing = createNewTransactionFromLastRecord(ledger_competing, my_ID);
    [ amount, sender, reciever  ] = randomTransactionParameters(tx_competing);
    tx_competing.alterIdentity(sender);
    tx_competing.makeTransfer(amount, sender, reciever, false);
    lg_competing = createLedgerEntry(tx_competing,ledger_competing);
    ledger_competing = addEntry(lg_competing, ledger_competing);
    fprintf('.')
end
disp(' ')
clear lg_competing tx_competing;

disp(' ')
disp('Attempt to replace the ledger with the longer competing ledger...')
LEDGER = replaceLedger(ledger_competing, LEDGER);

disp(' ')
disp('End of attacks')

%% Output
disp(' ')
disp(' ')
disp(['Total Tokens at original initialisation:', num2str(original_tokens) ' Tokens'])
disp(['Total Tokens in pool at end of testing: ', ...
    num2str(LEDGER(end).getTransactionData.getTotalBalance) ' Tokens'])
```

Code 14: genesisEntry.m

```
function [genesis_entry] = genesisEntry()
%genesisEntry() creates the first entry in the ledger.

% Tyson Cross 1239448

    tx0 = Transaction( ...
        {'39DF49683542CA728C04AC46C523D602541B25C1A8962E59DB639841CAB6B86A+Person1@Computer1' [0]}, ...
        '39DF49683542CA728C04AC46C523D602541B25C1A8962E59DB639841CAB6B86A+Person1@Computer1');
    tx0.lock();
    genesis_entry =     LedgerEntry(0, '0', datetime(737151,'ConvertFrom','datenum'), tx0,...
    '816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7');
end
```

Code 15: LedgerEntry.m

```
classdef LedgerEntry < matlab.mixin.Copyable %handle
    %LedgerEntry is a class for ledger entry objects, an individual entry in a simple
    %   blockchain of hash-signed transactional records.

    % Tyson Cross 1239448

    properties (SetAccess = private, GetAccess = private)
        TransactionData
    end
    properties (SetAccess = private, GetAccess = public)
        Index
        PreviousHash
        Timestamp
        Hash
    end
    methods
        function obj = LedgerEntry(index, previousHash, timestamp, data, hash)
        %LedgerEntry() contructor, creates the ledger entry
            obj.Index = index;
            obj.PreviousHash = previousHash;
            obj.Timestamp = timestamp;
            obj.TransactionData = data;
            obj.Hash = hash;
        end
        function r = getTransactionData(obj)
            r = obj.TransactionData;
        end
    end
end
```

Code 16: Transaction.m

```
classdef Transaction < matlab.mixin.Copyable
    %Transaction is a container class for transactional ledger data

    % Tyson Cross 1239448

    properties (SetAccess = private, GetAccess = private)
        Data
    end
    properties (SetAccess = private, GetAccess = public)
        ID_index
    end
    properties (NonCopyable)
        ID_me
        ID_recipient
        ID_sender
        Timestamp
        Transaction_amount
        Hash
        Locked = false;
    end
    methods
        function obj = Transaction(data, signature)
            %Transaction() contructor, sets the initial financial data
            obj.Data = data;
            obj.ID_index = [1:numel(obj.Data)/2];
            if isAMember(obj, signature)
                obj.ID_me = signature;
                setHash(obj);
                obj.Locked = false;
            end
        end

        function makeTransfer(obj, amount, sender, recipient, output)
```

```
            %makeTransfer() makes a transaction between participants
            if obj.Locked
                error('This transaction is already completed')
            elseif ¬doesExist(obj,sender) || ¬doesExist(obj,recipient)
                disp('Invalid ID')
            elseif (getBalance(obj,sender)-amount≤0)
                disp('Insufficient tokens')
            elseif ¬isAMember(obj, getSignature(obj,sender))
                disp('Not a participant, sorry');
            elseif ¬authenticateByID(obj, sender)
                disp('Authentication failure')
            elseif strcmp(getSignature(obj,recipient),getSignature(obj,sender))
                disp('Invalid transfer, sender and recipient must be seperate')
            else
                tic;
                obj.ID_me = getSignature(obj,sender);
                obj.ID_sender = getSignature(obj,sender);
                obj.ID_recipient = getSignature(obj,recipient);
                obj.Timestamp = datetime(clock);
                obj.Transaction_amount = amount;
                original_balance = getBalance(obj,sender);
                setBalance(obj,-amount,sender);
                setBalance(obj,amount,recipient);
                new_balance = getBalance(obj,sender);
                setHash(obj);
                t = toc;
                if output
                    disp(' ')
                    disp(['Sender:    ', getSignature(obj,sender)])
                    disp(['Recipient: ', getSignature(obj,recipient)])
                    disp(['Transferring ' , num2str(amount), ' tokens']);
                    disp(['Balance before transfer: ',num2str(original_balance), ' tokens']);
                    disp(['New balance: ', num2str(new_balance), ' tokens']);
                    disp(['Transaction completed at ', [char(obj.Timestamp), ' in ' num2str(t), ' ...
                        seconds']])
                    disp(['Transaction ID: ', obj.Hash]);

                end
                obj.lock();
            end
        end

        function r = getBalance(obj,id)
            r = obj.Data{id,2};
        end

        function setBalance(obj,value, id)
            if obj.Locked
                error('This transaction is already completed')
            else
                obj.Data{id,2} = obj.Data{id,2} + value;
                setHash(obj);
            end
        end

        function r = getSignature(obj,id)                  % retrieve hash of private key
            r = obj.Data{id,1};
        end

        function r = getTotalBalance(obj)                  % total token in system
            val = 0;
            for i=1:length(obj.Data)
                val = val + obj.Data{i,2};
            end
            r = val;
        end

        function r = isAMember(obj, signature)
            if ismember(signature,{obj.Data{:,1}})>0
                r = true;
            else
                r = false;
            end
        end

        function r = doesExist(obj,id)
            if ismember(id,obj.ID_index)>0
                r = true;
            else
                r = false;
            end
        end

        function setHash(obj)             % sets hash of transaction (for ledger hashing)
            if obj.Locked
```

```
                error('This transaction is already completed')
            else
                a = strcat(...
                    [char(obj.Data{:,1})],...
                    [char(obj.Data{:,2})],...
                    [char(obj.ID_me)],...
                    [char(obj.ID_recipient)],...
                    [char(obj.ID_sender)],...
                    [char(obj.Timestamp)],...
                    [num2str(obj.Transaction_amount) ]);
                a = a(¬isspace(a(:)));
                obj.Hash = hash(a);
            end
        end

        function r = authenticateByID(obj, id)                  % simulates comparing private key ...
            hash (by ID)
            if ¬strcmp(getSignature(obj,id),obj.ID_me)
                r = false;
            else r = true;
            end
        end

        function r = authenticateBySignature(obj, signature)    % simulates comparing private key hash
            if ¬strcmp(signature,obj.ID_me)
                r = false;
            else r = true;
            end
        end

        function lock(obj)
            obj.Locked = true;
        end
%      end

%      methods (Access = protected)
        function alterIdentity(obj, id)                     % Demo function to change ID
            obj.ID_me = getSignature(obj,id);
            setHash(obj);
        end

        function removeByID(obj,id)                             % for bad nodes
            if doesExist(id) && authenticateByID(obj,signature)
                disp(['Removing ',obj.Data(id,:)])
                obj.ID_index(id) = [];
                obj.Data(id,:) = [];
                setHash(obj);
            else
                disp('ID not found')

            end
        end

        function removeBySignature(obj,signature)               % for bad nodes
            [¬, pos] = ismember(signature,obj.Data(:,1));
            if pos>0 && authenticateBySignature(obj,signature)
                    disp(['Removing ', obj.Data(pos,:)])
                    obj.ID_index(pos) = [];
                    obj.Data(pos,:) = [];
                    setHash(obj);
            else
                disp('ID not found')
            end
        end

    end

end
```

Code 17: createLedgerEntry.m

```
function [ ledger_entry ] = createLedgerEntry( transaction, ledger )
%createLedgerEntry() creates a new ledger entry, retrieving the previous ledger entry's data

% Tyson Cross 1239448

previous_ledger_entry = getLatestLedgerEntry(ledger);
next_index = previous_ledger_entry.Index + 1;
next_timestamp = datetime(clock);
next_hash = calculateHash(  next_index,...
                            previous_ledger_entry.Hash,...
                            next_timestamp,...
                            transaction.Hash);
```

```
ledger_entry = LedgerEntry(next_index, previous_ledger_entry.Hash, next_timestamp, transaction, ...
    next_hash);

end
```

Code 18: addEntry.m

```
function [ ledger ] = addEntry( new_entry, ledger )
%addEntry() adds a new entry to the ledger

% Tyson Cross 1239448

if isEntryValid(new_entry, getLatestLedgerEntry(ledger))
    ledger = [ledger new_entry];
    ledger(end).getTransactionData().lock();
end

end
```

Code 19: calculateEntryHash.m

```
function [ hash_value ] = calculateEntryHash( entry )
%calculateHash() is a wrapper for the SHA-256 hash function to calculate ledger entries' hashes

% Tyson Cross 1239448

hash_value = calculateHash( entry.Index, entry.PreviousHash, datenum(entry.Timestamp), ...
    entry.getTransactionData().Hash );

end
```

Code 20: calculateHash.m

```
function [ hash_value ] = calculateHash( index, previous_hash, timestamp, datahash )
%calculateHash calculates the hash of a new leger entry using an implementation of SHA-256

a = [char(num2str(index)), ...
    [char(previous_hash)], ...
    [char(timestamp)], ...
    [char(datahash)] ];

a = char(a(¬isspace(a(:))));

hash_value = hash(a);

end
```

Code 21: createNewTransactionFromLastRecord.m

```
function [ new_transaction ] = createNewTransactionFromLastRecord( ledger, id)
%createTransaction retrieves the record of the last transaction

temp = getLatestLedgerEntry( ledger );
new_transaction = copy(temp.getTransactionData);
new_transaction.alterIdentity(id);

end
```

Code 22: getLatestLedgerEntry.m

```
function [ ledger_entry ] = getLatestLedgerEntry( ledger )
%getLatestLedgerEntry() returns the handle to the last ledger class object
%   in the ledger entry.

% Tyson Cross 1239448

ledger_index = numel(ledger);
ledger_entry = ledger(ledger_index);

end
```

Code 23: isEntryValid.m

```
function [ value ] = isEntryValid( new_entry, last_entry )
%checkEntryValid() checks the latest ledger entry for validity

% Tyson Cross 1239448

value = false;
if new_entry.Index ≠ last_entry.Index()+1
    disp('Invalid chain, index is not consistent')
elseif new_entry.PreviousHash ≠ last_entry.Hash
    disp('Invalid chain, hash values do not match')
elseif calculateEntryHash(new_entry) ≠ new_entry.Hash
    disp('Invalid hash!')
else
    value = true;
end

end
```

Code 24: replaceLedger.m

```
function [ LEDGER ] = replaceLedger( new_ledger, current_ledger)
%replaceLedger() Replaces the old ledger with a new ledger, if the length is longer

if isValidLedger(new_ledger) && (length(new_ledger) > length(current_ledger))
    LEDGER = new_ledger;
    disp('New ledger is valid. Replacing old ledger with newly received ledger');
else
    LEDGER = current_ledger;
    disp('Received ledger invalid. Retaining original ledger.');
end

end
```

Code 25: bin2decimal.m

```
function [ decimal ] = bin2decimal( bin )
%bin2decimal() converts a binary value to an unsigned decimal value
% Accurate conversion from 0 to intmax('uint64') is supported, binary
% values from 8 to 64 bits.

% Tyson Cross 1239448

if numel(bin)>64
    error('Input cannot be greater than 64-bits')
end

if ischar(bin)
    bin = logical(bin(:)'-'0');
end
value = uint64(length(bin)-1:-1:0);                        % Array of exponents from binary entries
base = uint64(2).^value;                                   % Decimal values for each bit
decimal = sum(base.*uint64(bin), 'native');                % Sum the entries

end
```

Code 26: dec2binary.m

```
function [ bin ] = dec2binary( decimal, num_of_bits )
%dec2binary() converts an unsigned decimal value to a fixed length binary value
% if the bitdepth is not specified, then an appropriate fixed length is
% chosen, or 64,32,16 or 8 bits. Accurate conversion from 8 to
% 64-bits is supported, for decimal values from 0 to intmax('uint64').

% Tyson Cross 1239448

if decimal > intmax('uint64')
   error('Input cannot be greater than intmax(''utint64'')')
end

if nargin<2
    if decimal > intmax('uint32')
        num_of_bits = 64;
    elseif decimal > intmax('uint16')
        num_of_bits = 32;
    elseif decimal > intmax('uint8')
        num_of_bits = 16;
    else
        num_of_bits = 8;
    end
end

decimal = uint64(decimal);
value = uint64(num_of_bits-1:-1:0);                           % Array of exponents for binary entries
base = uint64(2).^value;                                  % Decimal values for each bit

if decimal > sum(uint64(base), 'native')
   error('Not enough bits specified to represent decimal value')
end

bin = false(1,num_of_bits);                               % Initialise logical array
for i=1:num_of_bits
    if decimal ≥ base(i)                                 % For each applicable column of 2^i
        decimal = decimal - base(i);                      % Reduce the value of decimal
        bin(i) = true;                                    % Set the binary bit
    end
end

end
```

Code 27: char2logical.m

```
function [ logical_array ] = char2logical( char_array )
%char2logical() converts a char array (assumed to be of '0' and '1' chars) to a logical array

% Tyson Cross 1239448

if ¬ischar(char_array)
    error('Input must be a char array')
end

logical_array = logical(char_array(:)'-'0');
```

```

end
```

Code 28: logical2char.m

```
function [ char_array ] = logical2char( logical_array )
%logicaltochar() converts a logical array into a char array

% Tyson Cross 1239448

if ¬islogical(logical_array)
        error('Input must be a char array')
end

char_array = repmat('0',1,numel(logical_array));
for i=1:numel(logical_array)
    if logical_array(i)
        char_array(i)='1';
    else
        char_array(i)='0';
    end
end
% char_array = fprintf('%d', logical_array);

end
```

Code 29: logical2str.m

```
function [ str ] = logical2str( logical_array )
%str2logical() Converts a logical array to a string based on the char ascii values
% The array is assumed to be a padded logical array, with the following structure:
%   [ logical_array 1 padded_zeros length_of_message ]

% Tyson Cross 1239448

if ¬islogical(logical_array)
    error('Input must be a logical array')
end

bin_array = reshape(logical_array',1,[]);                    % flatten into single row
len = bin_array(end-63:end);                                 % extract message length
len_dec = bin2decimal(len);
bin_array = bin_array(1:len_dec);                            % trim padding
% bin_array = bin_array(1:floor(numel(bin_array)/7)*7);       % <- not be required if message ...
    was char string
raw_decoded_msg = reshape(bin_array, 7, [])';                % parse for decoding individual ...
    characters
str = char(bin2dec(num2str(raw_decoded_msg)))';              % decode into ASCII

end
```

Code 30: str2logical.m

```
function [ str ] = logical2str( logical_array )
%str2logical() Converts a logical array to a string based on the char ascii values
% The array is assumed to be a padded logical array, with the following structure:
%   [ logical_array 1 padded_zeros length_of_message ]

% Tyson Cross 1239448

if ¬islogical(logical_array)
    error('Input must be a logical array')
end

bin_array = reshape(logical_array',1,[]);                    % flatten into single row
len = bin_array(end-63:end);                                 % extract message length
len_dec = bin2decimal(len);
bin_array = bin_array(1:len_dec);                            % trim padding
% bin_array = bin_array(1:floor(numel(bin_array)/7)*7);       % <- not be required if message ...
    was char string
raw_decoded_msg = reshape(bin_array, 7, [])';                % parse for decoding individual ...
    characters
str = char(bin2dec(num2str(raw_decoded_msg)))';              % decode into ASCII

end
```

Code 31: flatten.m

```
function C = flatten(A)
C = {};
for i=1:numel(A)
    if(¬iscell(A{i}))
        C = [C,A{i}];
    else
       Ctemp = flatten(A{i});
       C = [C,Ctemp{:}];
    end

end
    C = squeeze(C(:));
    C = C(¬isspace(C(:)));
end
```

Code 32: flattenlogical.m

```
function [ output ] = flattenlogical( input )
%flattenlogical() Reduces a 1x1xW logical array to a 1xW logical array

% Tyson Cross 1239448

output = char2logical(logical2char(input));

end
```

Code 33: loadPeople.m

```
function [ participants ] = loadPeople( num_of_participants)
%loadPeople() Creates simple signatures for each participant

if nargin <1
    num_of_participants = 9;
end

for i=1:num_of_participants
    id_val = strcat({'id_'},num2str(i),'.pub');
    if i==1
        participants{i,1} = ...
            ['39DF49683542CA728C04AC46C523D602541B25C1A8962E59DB639841CAB6B86A+Person1@Computer1'];
    else
        participants{i,1} = [hash(char(randi(1000))) '+Person', num2str(i),'@Computer',num2str(i)];
    end
    participants{i,2} = randi([1000 10000]);
end

end
```

Code 34: randomTransactionParameters.m

```
function [ amount, sender, reciever  ] = randomTransactionParameters( transaction )
%randomTransactionParameters() sets up a random transfer of tokens

N = numel(transaction.ID_index);
sender = randi(N);
reciever = randi(N);
while sender==reciever
    sender = randi(N);
end
while reciever==sender
    reciever = randi(N);
end
amount = randi(100);

end
```

Code 35: checkLength.m

```
function checkLength(input, len)
%checkLength() checks the input length

% Tyson Cross 1239448

    if ¬length(input)==len
        error(['Length is not ',num2str(len)])
    end
end
```

Code 36: checkLogical.m

```
function checkLogical(input)
%checkLogical() checks the input is of type logical

% Tyson Cross 1239448

    if ¬islogical(input)
        error('Input must be logical array')
    end
end
```

Code 37: testConversionUtilities.m

```
% Tyson Cross 1239448
clc; clear all;

% conversion utilities tests

value_64 = intmax('uint64')/2;
bin_64 = dec2binary(value_64);
decimal_64 = bin2decimal(bin_64);
assert(isequal(decimal_64,value_64));

value_32 = intmax('uint32')/2;
bin_32 = dec2binary(value_32);
decimal_32 = bin2decimal(bin_32);
assert(isequal(decimal_32,value_32));

value_16 = intmax('uint16')/2;
bin_16 = dec2binary(value_16);
decimal_16 = bin2decimal(bin_16);
assert(isequal(decimal_16,value_16));

value_8 = intmax('uint8')/2;
bin_8 = dec2binary(value_8);
decimal_8 = bin2decimal(bin_8);
assert(isequal(decimal_8,value_8));
disp('dec2binary() and bin2decimal() passed basic functionality tests')


try bin_test = dec2binary(value_test,32);
catch ME
    disp([  'dec2binary() correctly threw an error (due to insufficient bitsize)' ])
end

try bin_test = dec2binary(18446744073709551616);
catch ME
    disp([  'dec2binary() correctly threw an error (due to a decimal value being too large)' ])
end

try decimal_test = bin2decimal(true(1,65));
catch ME
    disp([  'bin2decimal() correctly threw an error (due to a binary value being too large)' ])
end

% prompt = 'Enter a decimal value to convert to binary:\n';
% value = input(prompt);
value = randi(intmax('uint32'));
bin = dec2binary(value);
decimal = bin2decimal(bin);
assert(isequal(decimal,value));
% fprintf('%d', bin)

disp('---------------------------------------------------')
disp('All tests passed for dec2binary() and bin2decimal()')
disp('---------------------------------------------------')
disp(' ')

bin_array = dec2binary(intmax('uint32')/2);
char_array = logical2char(bin_array);
bin_array2 = char2logical(char_array);
char_array2 = logical2char(bin_array);
assert(isequal(char_array,char_array2));
assert(isequal(bin_array,bin_array2));
disp('logical2char() and char2logical() passed basic functionality tests')


try char_array = logical2char(char_array2);
catch ME
    disp([  'logical2char() correctly threw an error (due to invalid input type)' ])
end

try bin_array = char2logical(bin_array2);
catch ME
    disp([  'char2logical() correctly threw an error (due to invalid input type)' ])
end
disp('------------------------------------------------------')
disp('All tests passed for logical2char() and char2logical()')
disp('------------------------------------------------------')
disp(' ')
```

Code 38: testHash.m

```
% Tyson Cross 1239448

clc; clear all;

message = [ 'And above all, watch with glittering eyes the whole world around you '...
            'because the greatest secrets are always hidden in the most unlikely '...
            'places. Those who don''t believe in magic will never find it. '];

total_time = 0;
len = numel(message);
fprintf('Hashing... \n')

for i=1:len;
    tic;
    message = message(1:end-1);
    fprintf('%s\n',message)
    hash_value(i,:) = hash(message);
    t = toc;
    total_time = total_time + t;
    assert(length(hash_value(i,:))==64)
    if i > 1
        dif_hash(i,:) = (hash_value(i,:) == hash_value(i-1,:));
    end
end

clc;
fprintf('Hash values: \n')
disp(hash_value)
fprintf('Collision Map: \n')
for i=1:length(dif_hash)
    disp(logical2char(dif_hash(i,:)))
end

% I = mat2gray(dif_hash)
% rez = get(groot,'ScreenSize');
% figure('Position',[rez(4)/2 rez(3)/2 rez(3)/4 rez(4)/2])
% imshow(I,'InitialMagnification','fit');

disp(['Average time to hash: ', num2str(total_time/len)]);
disp(['Average bit-wise sequential collision probability: ', ...
    num2str(sum(sum(dif_hash))/numel(dif_hash))]);

clear message;
```

References

[1] L. Hartikka, "NaiveChain: A Blockchain in 200 lines of code," https://github.com/lhartikk/naivechain.