Trust function

Contents
Trust function........................................................................................................................................... 1
Abstract.................................................................................................................................................. 2
Description............................................................................................................................................ 3
Summary of the “Trust”................................................................................................................................ 4
External State......................................................................................................................................... 7
Abstract

The "Trust" is a legal entity associated with a cryptographic process. This creates a trust free trust.

The "Trust" is the implementation (prototype) of a process for cryptographic key generation and safekeeping that is provided against the Bitcoin ECC private key.

Using a plurality of key agents as trustees (each having a copy of the source keys) a distributed system can be programmed with the goals of a legal trust set and unalterable other than under the terms of the trust. This would for instance stop recipients of a trust who have turned 18 from contesting the trust that offers them funds when they turn 25 in a court as the software could not be altered outside the initial provisions of the trust.

To create the keys, a copy of the code is loaded onto a secure computer system and is compared with at least one other copy of the code to validate the loaded copy of the code. Master key information and locking key information are generated by executing the code.

The master key information is then separated into a plurality of master key shares which are distributed to master key agents such that each master key agent possesses one master key share. The locking key information is separated into a plurality of locking key shares which are distributed to locking key agents such that each locking key agent possesses one locking key share. Then, the plurality of locking key shares and the plurality of master key shares are validated, and the secure system is securely shut down.

This creates a trust where the trustee cannot act other than under the terms of the trust.
Description

Recent advances in cryptography (e.g. public key cryptography) have enabled it to play a much larger role in inter-organizational transactions. To support such transactions, some organizations should function as trusted agents.

For example, an organization could function as a trust authority whose role is to issue a digital certificate to a certificate bearer. The bearer of such a certificate can present the certificate to a third party as proof of information about the bearer. Alternatively the organization could function as an escrow agent. Among the many services such an agent could perform is to certify digital documents.

For an individual or organization to act as a trusted agent, the organization should have a mechanism to generate and keep secret a private encryption key of the type typically used with the ECC encryption method deployed within Bitcoin. In addition, for such an organization to obtain the trust of others, it should be able to demonstrate to them that such a private key can, in fact, be generated and stored securely, that is, without compromise or loss. A trusted agent, then, should not only be able to both generate and store secret information. It should also be able to convince other parties that its methods are sound and the secret information (i.e. secret cryptographic key) is secure. The ultimate way to ensure that an agent can be trusted is to ensure that the system is trust free. That is, the agent does not need to be trusted.

Presently, in publicly available documents, key generation and key storage are typically addressed as independent operations. The first of these operations, key generation, is addressed by finding a source of "random numbers" from which to generate the key. Issues such as the strength of the key so generated are usually ignored. Some proposed sources for these "random numbers" have included: key-stroke timing, machine state information, disk access or network traffic data, and various other computer attached devices such as audio ports. Alternatively, electronic "black boxes" of various levels of sophistication and of sometimes secret design provide a source of randomness. The latter suffer from the defect that while they may indeed provide randomness, most users will not know how they operate and therefore will not be able to truly trust them.

One standard method for storing a secret key (or any secret information) is by using a secret-sharing scheme, such as proposed by Shamir.

With a secret-sharing scheme, the secret is broken into two or more pieces, each piece tendered to an agent (a trustee) for safekeeping. Some critical number of agents, usually fewer than the total number, can reconstruct the secret by combining their shares, while no group of agents fewer in number than the critical number, can reconstruct the secret.

To further ensure trust, the trust beneficiary can maintain a section of the key themselves. This key section can be configured to be a critical component of the Shamir scheme to have a series of agents that need to act in concert with the trust and who cannot act against the beneficiary.
Summary of the “Trust”

In accordance with the present invention, a process and system for cryptographic key generation and safekeeping is provided that substantially eliminates or reduces disadvantages and problems associated with previously developed cryptographic key generation and protection processes in the Bitcoin ECC scheme.

According to one embodiment of the present invention, a process for cryptographic key generation and safekeeping is provided. Parameters are selected with respect to a cryptographic key including a minimum number of master key agents and a minimum number of locking key agents. A plurality of key agents are selected.

The key generation code is distributed to each key agent, and the copy of the code is validated.

The prototype implementation, the “Tulip Trust” is an M-of-N implementation that can work off block to distribute keys securely.

A “Brainwallet” style heuristic key generator is used to create a multisig address.

A secure computer system is set up, and the plurality of key agents are gathered along with each copy of the code. One copy of the code is loaded onto the secure computer system. The loaded copy of the source code is compared with each other copy of the code to validate the loaded copy of the code. The loaded copy of the code is then compiled.

Master key information and locking key information is generated by executing the compiled code. The master key information is then separated into a plurality of master key shares which are distributed to key agents designated to be master key agents such that each master key agent possesses one master key share. The locking key information is separated into a plurality of locking key shares. The plurality of locking key shares are distributed to key agents designated to be locking key agents such that each locking key agent possesses one locking key share. Then, the plurality of locking key shares and the plurality of master key shares are validated. The secure computer system is then shut down such that the master key information and locking key information cannot be reconstructed from the secure computer system.

In the “Tulip Trust”, a scheme of 3 of 5 keys has been created.

A “Time Release” of 2020 has been deployed. This will ned the trust and release the remainder in Dec 2020. See “External State”.

A technical advantage of the present invention is the provision of an improved process for constructing and safeguarding cryptographic keys, in particular a private key used in asymmetric encryption (e.g., ECC). The present invention comprises a process for generating keys which could be used to safeguard an organization's or individual's confidential data. More important, by using the process described herein, an organization can function as a trusted agent. Among other things, a trusted agent could provide a digital escrow service for other parties and hold for safekeeping digital documents in a provably unaltered form. A trusted agent could also issue digital identification certificates which enable commercial transactions over open communications networks.

Instead of one, we have a whole set of oracles. The majority of them need to agree on the outcome for the transaction to be finalized. We believe such a solution should be sufficient to protect any contract:

- It would be very hard and expensive to bribe more than half of the oracles.
- The judgement process is transparent due to the open source nature of the project.
- The software will become secure over time with ongoing community involvement.

Further, with a lock key, the settler or beneficiary can be setup in a manner that they cannot be excluded. That is, they may not be able to release funds themselves, but they could also stop or block any unauthorised use.

A technical advantage of the present invention is the provision of a key generation process in which values that must be kept secret depend upon provably pseudo-random values generated by a secure process whose instance and seed value can be empirically determined to be unpredictable. In one embodiment, the pseudo-random process is based upon a Blum-Blum-Shub random number generator which is known to have certain provable security properties. The present invention provides additional properties that allow the secure generation of key share information.
An additional technical advantage of the present invention is the fact that a person who possesses knowledge of the code used in the key generation process has very little chance of reproducing any system output. In addition, the source code is subject to inspection by anyone the key owner designates including members of the key owner's organization or its customers. Further, any such designee has, like the developer, very little chance of reproducing system output. This is in large part due to the fact that a large (on the order of 512 kilobyte) quantity of unpredictable input is used to initialize a pseudo-random number generator.
External State

Blockchain Scripts are, by design, pure functions. They cannot poll external servers or import any state that may change as it would allow an attacker to outrun the block chain. What's more, the scripting language is extremely limited in what it can do. Fortunately, we can make transactions connected to the world in other ways.

Consider the example of an old man who wishes to give an inheritance to his grandson, either on the grandson's 18th birthday or when the man dies, whichever comes first.

To solve this, the man first sends the amount of the inheritance to himself so there is a single output of the right amount. Then he creates a transaction with a lock time of the grandson's 18th birthday that pays the coins to another key owned by the grandson, signs it, and gives it to him - but does not broadcast it. This takes care of the 18th birthday condition. If the date passes, the grandson broadcasts the transaction and claims the coins. He could do it before then, but it doesn't let him get the coins any earlier, and some nodes may choose to drop transactions in the memory pool with lock times far in the future.

The death condition is harder. As Bitcoin nodes cannot measure arbitrary conditions, we must rely on an oracle. An oracle is a server that has a keypair, and signs transactions on request when a user-provided expression evaluates to true.

Here is an example. The man creates a transaction spending his output, and sets the output to:

<hash> OP_DROP 2 <sons pubkey> <oracle pubkey> CHECKMULTISIG

This is the oracle script. It has an unusual form - it pushes data to the stack then immediately deletes it again. The pubkey is published on the oracle's website and is well-known. The hash is set to be the hash of the user-provided expression stating that he has died, written in a form the oracle knows how to evaluate. For example, it could be the hash of the string:

if (has_died('john smith', born_on=1950/01/02)) return (10.0, 1IxrRHEHi86zYzHN2U4KMyRCg4LvwNUrp);

This little language is hypothetical, it'd be defined by the oracle and could be anything. The return value is an output: an amount of value and an address owned by the grandson.

Once more, the man creates this transaction but gives it directly to his grandson instead of broadcasting it. He also provides the expression that is hashed into the transaction and the name of the oracle that can unlock it.

It is used in the following algorithm:

The oracle accepts a measurement request. The request contains the user-provided expression, a copy of the output script, and a partially complete transaction provided by the user. Everything in this transaction is finished except for the scriptSig, which contains just one signature (the grandson's) - not enough to unlock the output.

The oracle checks the user-provided expression hashes to the value in the provided output script. If it doesn't, it returns an error.

The oracle evaluates the expression. If the result is not the destination address of the output, it returns an error.
Otherwise the oracle signs the transaction and returns the signature to the user. Note that when signing a Bitcoin transaction, the input script is set to the connected output script. The reason is that when OP_CHECKSIG runs, the script containing the opcode is put in the input being evaluated, _not_ the script containing the signature itself. The oracle has never seen the full output it is being asked to sign, but it doesn't have to. It knows the output script, its own public key, and the hash of the user-provided expression, which is everything it needs to check the output script and finish the transaction.

The user accepts the new signature, inserts it into the scriptSig and broadcasts the transaction. If, and only if, the oracle agrees that the man is dead, the grandson can broadcast the two transactions (the contract and the claim) and take the coins.

Oracles can potentially evaluate anything, yet the output script form in the block chain can always be the same. Consider the following possibilities:

```plaintext
today() == 2011/09/25 && exchange_rate(mtgoxUSD) >= 12.5 && exchange_rate(mtgoxUSD) <= 13.5
```

Require exchange rate to be between two values on a given date

```plaintext
google_results_count(site:www.google.com/hostednews 'Mike Hearn' olympic gold medal) > 0
```
A bet on me doing something that I will never actually do

```plaintext
// Choose between one of two winners of a bet on the outcome of the Eurovision song contest.
if (eurovision_winner() == 'Azerbaijan')
    return 1Lj9udBVDwptFffGJ2SohCfudQgSTPD;
else
    return 1JxgRXEHi86zYzHN2U4KMyRCg4LvwNUrp;
```

The conditions that control whether the oracle signs can be arbitrarily complex, but the block chain never needs to contain more than a single hash.
How the Trust protocol works (simplified example)

Let’s say that Alice promises to send Bob 200BTC as soon as it starts raining in Death Valley, California. They both agree that the condition will be met once the site http://deathvalley-weather.info/ contains the phrase “it’s raining today”.

Furthermore, Alice and Bob choose 7 nodes from the default node list at The Oracle List. In theory, they could choose any set of running nodes, but the default list is curated and contains the most reliable/trustworthy nodes. Choosing from the default list also saves time and effort that would otherwise be spent on list negotiation.

Alice now transfers her money to a temporary multisig address / “safe”. The money will be stored in the safe until 4 of the 7 chosen oracles decide that either the condition is met and the money should go to Bob, or that the money should get sent back to Alice.

What we don’t want though is to create an address that requires simply 4 of 7 oracle signatures, because that would allow for the majority of the oracles to send the money to an arbitrary address. We would ideally want both the **fund receiver’s signatures, and 4 of 7 oracle signatures**.

Transaction of 1+(m of n) signatures would be non-standard, but we can hack around this limitation. Alice creates an 8 of 11 multisig “safe”, where 7 of the signatures belong to the oracles, and 4 of the signatures belong to the receiver. With such a setup, the transaction needs both receiver’s, and at least 4 out of the 7 oracle’s approval to forward the funds to the destination address. In other words, we turn 1+(m of n) into (n+1 of n+m).

Alice creates an “unlock” transaction which forwards the funds from the safe, and pays fees to the oracles and to the Trust.

Alice sends the transaction to the oracles via a broadcasted Bitmessage, along with the redemption rules. We use Bitmessage as the protocol of communication with the oracles as an additional way of protecting their IP addresses and providing extra security.

The oracles check the validity of the transaction and rules. If they are valid, they add the transaction to their schedulers and reply with acknowledgement via a broadcasted Bitmessage.

Both Bob and Alice make sure that all the oracles acknowledged the validity of the transaction. If so, they then send funds to the safe. The contract is now active.

After many months, a drop of rain falls in Death Valley and the site deathvalley-weather.info displays a flashing sign: “it’s raining today”.

The oracles, one by one, notice the message on the website and add their signature to the transaction. The signed transaction is then broadcasted via Bitmessage.

Once there are enough signatures, Bob grabs the transaction off Bitmessage, and broadcasts it over the Bitcoin network. The funds are released.
Why Bitmessage is used as the communication channel

There are a few reasons to use the Bitmessage protocol instead of direct IP communication:

- We want to protect the anonymity of nodes so it’s harder to attack them.
- The proof of work required to sign Bitmessages prevents spam.
- Bitmessage provides nice and easy broadcasting capabilities - we want the communication to be as transparent as possible so node monitoring is easier.
FIG. 1

START

SELECT PARAMETERS AND KEY AGENTS

DISTRIBUTE AND VALIDATE SOURCE CODE

SET UP SECURE COMPUTER

GATHER AGENTS WITH SOURCE CODE

LOAD, COMPARE, AND COMPILE SOURCE CODE

GENERATE AND VALIDATE KEY INFORMATION

REBUILD MASTER KEY AND REISSUE SHARES

SHUT DOWN PROCEDURE

DONE
FIG. 2

START

FAIL

SAMPLE ENVIRONMENTAL NOISE FOR RANDOM BITS

TEST THE COLLECTION OF BITS FOR RANDOMNESS

NO

PASS?

YES

INITIALIZE THE SECURE RANDOM NUMBER GENERATOR

BUILD THE MASTER KEY

ISSUE SELF SIGNED CERTIFICATE FOR THE MASTER KEY

GENERATE A LOCKING KEY

GENERATE MASTER KEY SHARES AND ENCRYPT THEM WITH THE LOCKING KEY

GENERATE LOCKING KEY SHARES

WRITE MASTER KEY AND LOCKING KEY INFORMATION TO DISKS

READ MASTER KEY AND LOCKING KEY INFORMATION FROM THE DISKS

VALIDATE THE MASTER KEY AND LOCKING KEY INFORMATION

DONE
FIG. 3

RECONSTRUCT MASTER KEY FROM SECRET SHARES

FAIL

SAMPLE ENVIRONMENTAL NOISE FOR RANDOM BITS

TEST COLLECTION OF BITS FOR RANDOMNESS

PASS

INITIALIZE THE SECURE RANDOM NUMBER GENERATOR

GENERATE A LOCKING KEY

GENERATE MASTER KEY SHARES AND ENCRYPT THEM WITH THE LOCKING KEY

GENERATE LOCKING KEY SHARES

WRITE MASTER KEY AND LOCKING KEY INFORMATION TO DISKS

READ MASTER KEY AND LOCKING KEY INFORMATION FROM THE DISKS

VALIDATE THE MASTER KEY AND LOCKING KEY INFORMATION

DESTROY OLD SHARE DISKS

DONE
I'm glad we can work together. We know it's difficult to get things going, but sometimes it takes a lot of effort to make things happen. I hope we can make progress soon.

The idea is to get things moving with respect to the email and other matters. I can't see how it can be done without the support we have, and I hope we can make progress.

Let's keep in touch and move forward.