Updated 2018.04.04: see bottom

This is a writeup of an idea that I introduced at the meetup in Bangkok herehttps://youtu.be/OOJVpL9Nsx8?t=3h24m51s

Proof of work was proposed in the 1990s originally as a form of email spam prevention by Dwork and Naohere. The idea is that a user's email client would only show emails that come with a proof of work nonce n

attached, such that n

is the solution to some math problem based on the contents of the email; in Adam Back's hashcash it could be simplified to hash(email_message + n) < \frac{2^{256}}{D}

, where D

is a difficulty parameter. This requires all email senders to do a small amount of computational work to send an email that the recipient will read, with the hope that legitimate users (who have a high valuation for the ability to send an email) will be willing to do this cost but spammers (who have a low valuation) will not.

The problem is that (i) there's an imbalance in favor of the spammers, as specialized implementations can do proof of work better than regular users (not to mention smartphones), and (ii) it turns out the happy medium that's low enough for users and too high for spammers does not really exist.

But we can improve this approach using proof of stake! The idea here is that we set up a smart contract mechanism where along with an email the recipient gets a secret key (the preimage of a hash) that allows them to delete some specified amount (eg. $0.5) of the sender's money, but only if he wants to; we expect the recipient to not do this for legitimate messages, and so for legitimate senders the cost of the scheme is close to zero (basically, transaction fees plus occasionally losing $0.5 to malicious receivers).

The contract code might look like this:

deposits: public({withdrawn_at: timestamp, size: wei_value, creator: address}[bytes32])

@public @payable def make_deposit(hash: bytes32): assert self.deposits[hash].size == 0 self.deposits[hash] = {withdrawn_at: 0, size: msg.value, creator: msg.sender}

@public def delete_deposit(secret: bytes32): self.deposits[sha3(secret)] = None

@public def start_withdrawal(hash: bytes32): assert self.deposits[hash].creator == msg.sender self.deposits[hash].withdrawn_at = block.timestamp

@public def withdraw(hash: bytes32): assert self.deposits[hash].withdrawn_at <= block.timestamp - 3600 send(self.deposits[hash].creator, self.deposits[hash].size) self.deposits[hash] = None

An email client would check that a deposit is active and a withdrawal attempt has not yet started. Individual recipients could also demand that the secret key they receive has some properties (eg. containing their own email address at the beginning), thereby preventing reuse of the same secret key among multiple users.

Optimizations:

1. If a sender needs to send multiple emails at a time (eg. a group email, or a legitimate newsletter), they could publish a Merkle root of the secret keys, and give each recipient a Merkle branch of their secret.

2. If you don't want the sender to know if any individual recipient punished them, then you could have a game where all senders put in $0.5, by default have a 50% chance of getting $1 back, but if the recipient wishes the chance the sender will get $1 back drops to 0%. This can be done by requiring the recipient to pre-commit to a list of N values, then for each email, after some time passes and a block hash becomes available as random data, the recipient would normally see if the value they committed to for that email has the same parity (even vs odd) as the block hash, and if it is they would publish it and unlock the sender's double-deposit gain. If a recipient has a grudge against the sender, they could simply never publish the value regardless of parity. It would be expected as a default that the committed values are deleted as soon as they are used, so that recipients could not be forced to prove how they acted.

Update 2018.04.14

:

This could also be used for paywalled content services: a user needs to pay to access the content, but at the end the user can choose whether the funds go to the author or to charity. This removes perverse incentives in existing paywalls (eg. yours.org) where it's in writers' interest to put their best content before the paywalled portion of a post begins in order to advertise it; with this scheme, unhappy writers could deny authors their revenues after being disappointed with the initially hidden portion of a piece.

Suggestion from @danrobinson: another option is that if a user dislikes a piece of content, the funds paid by the user go into

a pool which refunds all previous users (ideally only previous users, to avoid making the content cheaper and thereby perversely more popular). This increases the scheme's "efficiency" (users and writers don't find it in their mutual benefit to switch to a platform that helps both keep their money more) as well as its "fairness" (if everyone finds that some piece of content sucks, then everyone gets most of their money back).