Abstract

I describe the design of NooShare, a decentralised ledger similar to Bitcoin [11] with the novel feature that its proofs of work are iterations of essentially arbitrary Markov-Chain Monte-Carlo (MCMC) chains, the scheduling of which can be purchased using the currency itself. It is a novel economic basis for sharing fallow computational resources.

NooShare: A decentralized ledger of shared computational resources

Alex Coventry
coventry@gmail.com

April 25, 2012

1 Introduction

NooShare is a crypto-currency similar to Bitcoin [11]. Nooshares[1] have intrinsic utility which increases linearly with the number of people participating in the network because they can be used to purchase scheduling of essentially arbitrary Monte-Carlo algorithms in the proofs of work which secure the Nooshare blockchain. Monte Carlo algorithms have been developed for an extremely wide variety of inference [8], optimization and molecular dynamics problems[2]. The key idea is to include the output of some arbitrary computation in the string which ends up being hashed. This in itself is rather simple, but it raises some serious security and design problems. See sections 2.2, 2.3 and 3 for an overview of these problems and how I address them. There are some novel ideas there which may also be useful in other contexts.

I welcome all critical feedback on all aspects of this document and the ideas it presents. I am sure I have missed some important security concerns. The major major weakness I see at the moment is that the computational model of embarassingly parallel Monte Carlo calculations is rather exotic, and not worth the trouble for most people to adopt it.

My development of NooShare is not intended as criticism of Bitcoin. Although Bitcoin is not backed by any resource of intrinsic value, it has already proved itself as trustworthy medium of exchange and therefore a viable currency. As David Graeber demonstrates in *Debt: the First Five Thousand Years*, money is whatever we say it is and has been attached to metals of "intrinsic" value for only brief and relatively inhumane portions of its history [9]. I am absolutely fascinated by the Bitcoin ecosystem and believe it may enable previously impossible modes of human organization. My main interest in developing NooShare is that, other things being equal, there is obvious economic advantage to a blockchain ledger which secures itself with computations which can simultaneously be turned to some other purpose.

[1] A unit of the NooShare currency is called a nooshare. Abbreviated "NS."

[2] I must admit that in these three fields of Monte Carlo algorithms I am only experienced with the applications to inference, but it takes no expertise to craft a brief search which finds tens of thousands of papers related to Monte-Carlo optimization and molecular dynamics.

Also, even though NooShare provides transaction semantics roughly equivalent to Bitcoin's, it is not intended as a replacement for Bitcoin's primary role as a medium of exchange in transactions involving existing goods. NooShare's computations cannot be executed on the Graphics Processing Units (GPUs) which contribute the bulk of Bitcoin's proofs of work, so NooShare does not compete with Bitcoin for computational resources, either.

NooShare's intended role is as a new market for application of previously fallow computational resources. There have been a number of attempts at such markets in the past, such as CPUShare [1] or Parabon [2], but those I am aware of arose before the Bitcoin blockchain concept was published, and rely on a centralized transaction system. NooShare is the first fully decentralized market for computational resources.

2 Overview of protocol

2.1 Brief description of Bitcoin protocol

NooShare builds on the Bitcoin protocol. Here is a very brief overview of the main Bitcoin concepts needed to understand NooShare. Further details of Bitcoin's operation can be found in the seminal Bitcoin paper [11] and the Bitcoin wiki [4]. Relevant wiki pages are linked in the following text.

Bitcoin is essentially a ledger of *transactions* which transfer funds between *addresses* (Elliptic-Curve Digital Signature Algorithm (ECDSA) public keys) which are represented as base-58 strings. A transaction from address a to address b is essentially a commitment, signed by a, of a certain number of Bitcoins to address b. These transactions are broadcast over a peer-to-peer network, and participants (*"miners"*) gather all the transactions they've seen on the network, check whether they're already recorded in the ledger, and assemble the unrecorded transactions into *blocks*. The ledger is recorded in a series of these blocks, and each block contains a hash of the block which precedes it, so the blocks form a *chain*. Actually, the blocks theoretically form a tree rather than a linear chain, because in principle the hash of a given block B could be included in several subsequent blocks, all of which would be claiming B as their predecessor. However, the network always accepts the currently longest such chain in the tree as the canonical ledger.

When miners generate blocks, they include a previously unbroadcast *coinbase transaction*, which grants a *reward* to the public key of an address they control. This reward is halved every four years. A summary of the block ("*header*") is generated along with a nonce, and the combination is passed through two iterations of the `SHA256` hash algorithm. If this results in a hash image less than some threshold, the block, hash and nonce are reported to the network, and as long as all the information in the block is self-consistent and consistent with the ledger up to that point, the block is prospectively accepted as the next in the chain. The work of searching for these hash images is motivated by the reward in the coinbase transaction. This sort of distributed exhaustive search for the solution to

a cryptographic problem is sometimes called a "Chinese Lottery" [13].

As long as a majority of the computational resources devoted to the lottery are controlled by honest participants who only assemble blocks of transactions consistent with the ledger to date, this hash-image requirement forces consensus about the ledger contents. For a party to change the ledger, they would have to generate a chain of blocks with altered transactions forking off from an earlier block, and find nonces for their altered blocks which give appropriate hash images. An attack of this sort is called *"forking the blockchain,"*, or sometimes the *51% attack*, because it requires more than half the network's computational resources. Because the Bitcoin network always takes the longest chain as canonical, a dishonest party would have to be able to find these nonces faster than the rest of the network combined.

The threshold imposed on the hash image is controlled by the difficulty parameter, and is periodically adjusted according to an estimate of the current computational resources devoted to the problem, so that the problem is solved approximately once every 10 minutes.

2.1.1 GPU-hostile crypto-currencies derived from Bitcoin

The `SHA256` hash function used in the Bitcoin proof of work is highly amenable to optimization on massively parallel Graphics Processing Units (GPUs) and most Bitcoin mining is now done on specialized GPU hardware as a result. The crypto-currencies Tenebrix [6] and Litecoin [5] are derived from the Bitcoin codebase, but use members of the `scrypt` family of hash functions [12] in place of `SHA256`. These hash functions can be tuned to require rapid access a very large memory space, making them particularly hard to optimize to specialized massively parallel hardware.

2.2 Changing the hashing function to do a useful computation

Section 2.3 describes NooShare transactions which can schedule computation of arbitrary Monte Carlo functions in the proof-of-work lottery. In this section I focus on the aspects of the protocol which would be needed even if the Monte Carlo function used in the proof of work never changes. Denote this function by $A(s, d)$, where s is a seed for a pseudo-random generator (PRNG) and d is some data string, perhaps from an earlier iteration of A. Let R be some metric which takes a result generated by A and returns a number representing its usefulness to the problem A is being used to solve. For instance, if A is an MCMC procedure for sampling from a Bayesian posterior distribution, it might return choices of model parameters, and $R(A(s, d))$ might be the log likelihood for those parameters given some fixed set of observations. Or, if A is a Monte-Carlo molecular dynamics simulation, its return value might represent a molecular configuration and R might be the negative of the configuration's potential energy.

The function A can be included in a hash function H_A by the following simple procedure: Let $H(\cdot)$ be a conventional cryptographic hash function such as `SHA256` or `scrypt`. Let $D(H(\cdot))$ be some function which generates

a data string of the form A requires for its second argument ("d" in the previous paragraph.)[3] Given a string S, define $\hat{A}(S) = A(H(S), D(H(S)))$. A hash incorporating the computation in A is given by

$$H_A^1(S) = H(H(S) + \hat{A}(S)),$$

where + denotes concatenation as strings. That is, we use $H(S)$ to choose the PRNG seed and initial data string we feed to A, then concatenate the data string from A with $H(S)$ and hash again. The hash function H_A^1 could be plugged straight into Bitcoin's proof of work lottery, but this would not be secure and the results reported in the blockchain would probably not be very interesting. In this section I will describe how to extend this simple idea to make it more secure and useful, building the expression for H_A up in stages. To help you keep track of the extra components of H_A at each step the current version is indicated by a superscript ("$H_A^{\mathbf{1}}$", "$H_A^{\mathbf{2}}$", "$H_A^{\mathbf{3}}$", ...) The two problems I will address in this section are

1. From the perspective of whatever Monte Carlo problem motivates A, choosing results by the hash they generate rather than their R value is basically useless. For the network's repeated computation of A to be useful to the Monte Carlo problem, there has to be an incentive for miners to report the best results they have seen.[4]
2. It may be possible to maliciously craft the block chain so that the second argument to A, $D(H(S))$, results in A running very slowly, or never even producing any output. This could be used to fork the block chain with minimal computational resources, so there has to be a way to detect such malice, and an alternative protocol for the network when faced with it.
3. Someone could find a way to compute A much more rapidly than everyone else, which could also be used to facilitate a blockchain fork.

2.2.1 Encouraging miners to report the best results

For the security of the NooShare ledger the hash function $H_A(S)$ must be a good approximation to a random oracle and therefore have no correlation at all with $R(\hat{A}(S))$, the measure of value for the Monte Carlo problem.

[3] For some applications, like Bayesian sampling from a posterior distribution, complete analysis of the results requires a full Markov Chain (d_i) with $d_{i+1} = A(\cdot, d_i)$. It may therefore be useful for D to take extra parameters, such as relevant parts of the blockchain to date or earlier results from local execution of A. Actually reporting the full chain to the network would require far too much bandwidth, so the best results the network reports must be seen as approximate modes in the inference problem's parameter space, the neighborhoods of which can be investigated more thoroughly in locally computed Markov Chains which start with these modes.

[4] If NooShare were actually using a fixed algorithm A in its proof of work, there would be a simpler scheme than the one described here. Adjusting the difficulty threshold required for $H_A(h)$ according to the rank of $R(\hat{A}(h))$ relative to results reported in earlier blockchain lotteries would generally result in better results making it into the blockchain. That scheme won't work for the full NooShare protocol because as described in section 2.3, A can be specified by potential attackers.

This means that out of all the iterations of A which go into generating a NooShare block, the one which wins the proof of work lottery is essentially arbitrary. Many interesting results from other iterations of A will thus simply be thrown away unless they are reported in some other fashion. For this reason each block generation in NooShare is followed by a second competition in which miners report the best results they generated during the lottery, and the reporter of the very best receives a reward. This section describes how the competition is structured so no one can cheat it.

Once someone has generated a block using H_A and broadcast it to the network, each miner broadcasts their best result $(\hat{A}(S), R(\hat{A}(S)))$, in the sense that the value metric $R(\hat{A}(S))$ is largest. To prevent a flood of these reports, the default client delays the report by a random interval, and does not bother to send it if it has already seen a better report. All miners watch the network for the best such report, and verify it by recomputing $R(\hat{A}(S))$. This verification is potentially moderately computationally intensive, so to secure the network against a malicious flood of bogus reports with high R values, the reports must be signed by ECDSA keys containing miniature proofs of work, namely their base-58 representations[5] are required to start with some fixed prefix, perhaps the characters `NShr`. On average, a key with this specific prefix can be found by generating about 200,000 random keys, an expensive operation which must be completed before mining. Bulk generation of Bitcoin addresses can be dramatically accelerated by using GPUs, so the NooShare address-generation algorithm replaces all `SHA256` hashes in the Bitcoin algorithm with `scrypt`, a GPU-hostile hashing algorithm.[6] I will call such proof-of-work keys *work addresses* and denote them by the symbol a. Any work addresses which have been used to sign bogus reports are permanently blacklisted by the network, and the blacklist is tracked both internally by the default NooShare client and in the blockchain by the inclusion of one bogus report per blacklisted work address.

By a mechanism which I will describe shortly, the best report ends up in the NooShare blockchain ledger, and the work address which signs it is awarded a prize. It has to be possible to verify that the work address a which signed the report belongs to the miner which generated the result. Otherwise, miners could rip the result out of the best report they've seen, sign it themselves, and include the stolen report in the blocks they subsequently generate, so that if they win the block-generation lottery they also win the best-result competition from the prior block. To prevent this both the result reports and the headers which are hashed in the proof of work lottery itself must include some extra information. The headers include an entry for a, and an entry for the signature of the block by a. The signature entries of the headers are excluded from the signed texts for obvious reasons. Let $s_a(h)$ denote this signature. Then we define

$$H_A^2(h) = H_A^1(h + s_a(h))$$

[5] To distinguish them from Bitcoin addresses, NooShare addresses will start with `N` instead of `1`.

[6] I haven't yet determined how difficult this proof of work needs to be. It should be a simple matter to test it by changing `vanitygen` to use `scrypt` instead of `SHA256`.

When miners report the best results they have generated, they send

$$(a, h, s_a(h), \hat{A}(h + s_a(h)), R(\hat{A}(h + s_a(h)))).$$

It is then possible to verify reports by checking $s_a(h)$, recomputing $\hat{A}(h + s_a(h))$ and $R(\hat{A}(h + s_a(h)))$, and checking equality.

Ensuring that the best result reports are included in the blockchain For this competition to work, it is also important that miners have an incentive to include in the blocks they generate the best results they have seen reported. This is similar to the problem recently addressed by the "red balloons" payment strategy [7], but admits of a much simpler solution because we only have to encourage the inclusion of a single report per generated block.

The best-result competition for a given block is allowed to run until ten subsequent blocks have been generated, and the NooShare ledger then assigns the prize to the best report embedded in these ten blocks. The coinbase address of the first block to include the best report is also awarded a prize. Any miner who tries to bias the competition by excluding the best report from a block they generate simply loses this prize to a subsequent miner who is prepared to include it.

2.2.2 Adapting to prohibitively long runtimes for A

It may happen that $A(S, d)$ does not halt in a reasonable time for some values of S and d, and NooShare miners need a way to respond to this. Although this is already a potential problem for fixed A (perhaps a somewhat remote one), it becomes critical in section 2.3, where untrusted parties are able to specify A.

For this reason, the NooShare network always accepts blocks hashed using $H_{\mathcal{A}}$, where $\mathcal{A}(S) = S$ is the identity algorithm, but the difficulty threshold for $H_{\mathcal{A}}$ set to ten times the difficulty for H_A. The default NooShare mining application always keeps track of the CPU time used by the current hashing function, and periodically runs $H_{\mathcal{A}}$ to measure its CPU usage. If any iteration of H_A takes 20 times more CPU time than the mean for the last fifty runs of $H_{\mathcal{A}}$, that iteration is killed. If the mean CPU time for H_A exceeds ten times that of $H_{\mathcal{A}}$, the miner switches over to using $H_{\mathcal{A}}$ until the next block is generated.

To keep the runtimes for H_A and $H_{\mathcal{A}}$ roughly comparable, at least the outer hash function needs to be GPU-hostile, *i.e.*

$$H_A^3(S) = H_{\texttt{scrypt}}(H(S + s_a(S)) + \hat{A}(H(S + s_a(S))))$$

If it's substantially more efficient to do so, the inner hash function H can be something faster like `SHA256`.

2.2.3 Protecting against massive speedups in computation of A

Someone who figured out a much faster way to compute the Monte Carlo function A could place a proportionately greater fraction of entries in

the Chinese lottery, making the blockchain ledger vulnerable to a "51% attack" with relatively modest computational resources. For this reason, every second block uses the fixed benchmark algorithm $\mathcal{A}$ with the hash-image difficulty set so that on average a block is generated every 30 minutes.[7]

This is a decentralized modification of the "trusted nodes" concept[8] used by the SolidCoin cryptocurrency [3].[9] It means that even if someone figures out a way to compute H_A instantly, they still need to either do the same for $H_{\mathcal{A}}$ (which essentially means cracking the `scrypt` hash), or control more than a quarter of the network's hashing power. Effectively, the 51% attack would become a 26% attack, but as the network grows this should be a sufficient defense.

2.3 Allowing control of A in return for NooShares

The computation A can be specified by a special transaction T_A which destroys the NooShares it contains. This is known as a "*scheduling transaction.*" After T_A appears in the blockchain (in, say, block n), the corresponding H_A is scheduled for use in the proof of work. The number of the block in which H_A is used is $n_A = F_{n+24+2e}$, where F_m is the earliest block available for scheduling with number at least m and e is sampled from the geometric distribution with mean 12, using a PRNG seeded from the hash of block $n + 1$. If a block contains multiple scheduling transactions, they are scheduled in the order the block lists them. NooShare uses this random scheduling of the hashing algorithm to complicate any attempts to facilitate forking the blockchain by scheduling maliciously crafted hashing algorithms. Any such attempt would require control over the hashing algorithm in an unbroken range of blocks.

The computational resources demanded by a scheduled computation A are controlled by the fact that miners will switch over to hashing with $H_{\mathcal{A}}$ if the mean runtime for A is ten times greater than for $\mathcal{A}$.

2.4 Prices and rewards

The coinbase reward for generating a block is 50 NooShares. Unlike the Bitcoin block reward, this reward is never reduced, because NooShares have intrinsic utility (the computations they can buy) which increases with the size of the computational resources devoted to the network. There is thus no need to encourage mining with the prospect of deflationary scarcity.

[7] The $H_{\mathcal{A}}$ difficulty for the other blocks is the same, and the difficulty is reset at each block by taking the median time to solution of the last 336 $H_{\mathcal{A}}$ blocks.

[8] The recent SolidCoin "trusted node" exploit is not a risk here, because there is no node to attack. The only parallel is that every second block is verified by a different method, meaning that any attack which overruns the blockchain needs to undermine at least two distinct proofs of work.

[9] This citation acknowledges the source of a technical idea and in no way reflects any investment, emotional or otherwise, in the SolidCoin drama. Please leave NooShare and I out of it.

The prize for the result with the best R value is five NooShares, and the prize for the coinbase address of the block which first reports it is also five NooShares.

The scheduling transactions T_A must commit at least 65 NooShares, with anything in excess of that going to the best result reported.

It is important that the reward for block generation is substantially higher than the reward for the best result, since the blockchain's need for bit-identical calculations (see section 3.2.1) forces H_A to run a good deal slower than A could run standalone, mainly because hardware implementations of floating-point arithmetic are inconsistent. This disparity means that, for instance, if the prize for reporting the best result was 100 times the reward for generating a block, miners might get a higher expected return from generating results from A independently of the blockchain lottery, and only generating H_A for the result with the highest R value that they choose to report to the network.

It is also important that the total of the prizes connected to a given block (block reward + best-result reward + best-result-in blockchain award) be less than the cost of scheduling a computation A for that block. Otherwise it would be cost-effective for an attacker to repeatedly specify computations which they can solve much faster than everybody else, and collect all the associated prizes.

3 Implementation

The NooShare client builds on the Satoshi Bitcoin client, the most complete and thoroughly tested Bitcoin client currently available.

3.1 Security

On the one hand, the best outcome for NooShare is that it becomes a marketplace for the allocation of substantial computational resources, on the other it executes code provided by anonymous strangers, and the history of applications which do this is fraught with privilege-escalation and denial-of-service attacks. Fortunately, sandboxing technologies are now extremely reliable, and by combining a few orthogonal sandboxes it is possible to ensure that a succesful attack on NooShare by this route would require multiple zero-day exploits in high-profile security technologies.

3.1.1 Lua

The Monte Carlo algorithms used for NooShare's proofs of work are expressed as Lua scripts. Although Lua does not count as a "high-profile security technology," it has is written in extremely simple C and provides simple namespace-based sandboxes which have been used in multiple untrusted-execution contexts with moderate security demands, such as Zippy Log, GINGA-NCL [15] and World of Warcraft. While the Lua sandbox has not been used in particularly hostile or high-stakes environments, any attack on the NooShare hashing function would need to exploit a vulnerability in it.

Lua makes it easy to specify a custom memory allocator. The Lua process will be pre-assigned a block of memory to allocate from, and unable to allocate memory beyond that.

3.1.2 Google Native Client

The Lua library is built and run in Google's Native Client (NaCl) framework [16]. The Lua process communicates with a trusted process via a custom protocol over NaCl's "simple RPC" API. This drastically reduces the attack surface relative NaCl's standard usage in Google Chrome, as the trusted codebase is only responsible for brokering data transfer and the extensive system resources brokered by the Pepper API in Google Chrome are simply not available to the sandboxed computation.

3.1.3 VMWare

The hashing function is executed in a headless VMWare virtual machine instance running linux using hardware virtualization. Requiring hardware virtualization restricts the use of NooShare to relatively modern machines, but older machines are unlikely to find much use on the network anyway.

An appealing feature of the Google Native Client code verifier is that it forbids all VM-related x86 opcodes.[10]. Thus, even if a hostile Lua script manages to induce execution of arbitrary Native Client code, there is nothing in the binary which will allow it direct communication with the VMWare Virtual machine monitor (VMM.) It would probably also be reasonably straightforward to modify the VMM to take a "sandbox" signal from the guest OS, after which all but a restrictive whitelist of VM interactions would fail and shut down the guest, but that isn't necessary at this stage.

3.1.4 Linux sandboxing

The linux utrace framework can be used for sandboxing untrusted code [10]. The Lua process will run in a sandbox which is restricted to the syscalls required by the Native Client framework.[11] It will also run under an AppArmor profile [14, p. 265] which denies it file system and remote network access.

3.1.5 Update mechanism

So that security patches to the sandbox component systems can be applied reliably, NooShare needs a mechanism for pushing updates to clients. For the time being, mandatory NooShare client updates are enforced using network-wide alert system which causes affected clients to cease operating. This is based on the Bitcoin alert system.

[10] See function `NaClAddNaClIllegalIfApplicable` in the Native Client source code, subversion revision 6679, in particular the `NACLi_SVM` and `NACLi_VMX` cases.

[11] It's simple enough to do, but I haven't yet determined the exact list of syscalls it needs.

3.2 Bit-identical calculations

Execution of a cryptographic hash algorithm needs to be bit-identical across machines, and Lua needs to be adjusted to ensure that this will be the case for the NooShare hashes.

3.2.1 Software floating point

It is notoriously difficult to ensure bit-identical calculations using hardware floating point. For this reason, the NooShare Lua is modified to use a software floating point library.

3.2.2 The Lua table data structure

Lua tables refer to certain types of objects by their memory address, leading to variability in the order of iteration over a table's entries. To prevent this, the Lua table algorithm is modified to keep a counter that is updated whenever a new Lua object is added to a table. The current counter value is stored in the object, and the modified Lua tables refer to this value rather than the memory address when looking up an object.

References

[1] `http://packages.ubuntu.com/maverick/cpushare`.

[2] `http://www.parabon.com/capacity-market/`.

[3] `http://wiki.solidcoin.info/wiki/Trusted_Nodes`.

[4] Bitcoin Wiki. `http://bitcoin.it`.

[5] Litecoin web page. `http://litecoin.org`.

[6] Tenebrix web page. `http://tenebrix.org`.

[7] Moshe Babaioff, Shahar Dobzinski, Sigal Oren, and Aviv Zohar. On bitcoin and red balloons. `http://research.microsoft.com/apps/pubs/?id=156072`, 2011.

[8] S.P. Brooks, Gelman, G.L. A.E. Jones, and X.L. Meng, editors. *Handbook of Markov chain Monte Carlo*. Springer-Verlag, 2010.

[9] David Graeber. *Debt: the First Five Thousand Years*. Melville House, 2011.

[10] Roland McGrath. seccomp via utrace. `http://www.redhat.com/archives/utrace-devel/2009-March/msg00159.html`, 2009.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2008.

[12] Colin Percival. Stronger key derivation via sequential memory-hard functions. presented at BSDCan '09, May 2009. See also `http://www.tarsnap.com/scrypt/scrypt.pdf`.

[13] Jean-Jacques Quisquater and Yvo G. Desmedt. Chinese Lotto as an Exhaustive Code-Breaking Machine. *Computer*, 24(11), 1991.

[14] Kyle Rankin and Benjamin Hill. *The Official Ubuntu Server Book*. Prentice Hall, 2009.

[15] L.F.G. Soares, M.F. Moreno, and C. De Salles Soares Neto. Ginga-NCL: Declarative middleware for multimedia IPTV services. *Communications Magazine, IEEE*, 48(6):74–81, 2010.

[16] B. Yee, D. Sehr, G. Dardyk, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, 2009.