Fair Client Puzzles from the Bitcoin Blockchain

Colin Boyd and Christopher Carr
Norwegian University of Science and Technology,
NTNU, Trondheim, Norway

{colin.boyd,chris.carr} @ntnu.no

Abstract

Client puzzles have been proposed as a mechanism for proving legitimate intentions by providing "proofs of work", which can be applied to discourage malicious usage of resources. A typical problem of puzzle constructions is the difference in expected solving time on different computing platforms. We call puzzles which can be solved independently of client computing resources *fair client puzzles*.

We propose a construction for client puzzles requiring widely distributed computational effort for their solution. These puzzles can be solved using the mining process of Bitcoin, or similar cryptocurrencies. Adapting existing definitions, we show that our puzzle construction satisfies formal requirements of client puzzles under reasonable assumptions. We describe a way of transforming our client puzzles for use in denial of service scenarios and demonstrate a practical construction.

Keywords: Bitcoin, Client Puzzles, Denial of Service Resistance, Distributed Computation, Proofs of Work.

1 Introduction

Client puzzles, also referred to as *proofs of work* [2, 10, 11], were originally introduced by Dwork and Naor [8] in 1993 and offer a valuable defence against denial of service (DoS). Client puzzles are designed to be used when needed, and can be turned on when a service is receiving an over abundance of requests. Motivation for client puzzles, in the general sense, is fairly intuitive. All service providers would like to ensure that those requesting service legitimately want it, but in an online setting it is difficult to discern a legitimate request for service from a malicious one. So how does one ensure the legitimate intentions of a client machine? This question led to a heuristic that says 'if a party requesting service is willing to put in some level of effort to connect, then that party is likely to be legitimate' [18]. Schemes have been proposed that are built around solving computationally expensive puzzles, such as extracting square roots modulo a prime [8], or partially inverting a hash function [3]. Legitimate clients should not find solving any single puzzle to be burdensome, yet, we do not want puzzle solutions to be found trivially either. Thus anyone wishing to exhaust the computational resources of the server by sending multiple connection requests would have to solve one instance of the puzzle per request. It thus becomes difficult for any DoS initiator to attack without access to an enormous amount of computing power.

An issue of primary importance is availability of computational resources on the client's side, which can vary considerably from device to device. This disparity is a persistent problem with client puzzle applicability. Indeed, another line of research aims to alleviate the problem with computational cost by instead relying on memory bounded cost [1, 16]. By adopting the blockchain for use as the puzzle solution algorithm, we demonstrate an alternative and simpler solution to this problem. We do this by equating monetary cost with computational cost and simultaneously introduce a notion of fairness.

This paper introduces the notion of *fair client puzzles* by exploiting modern developments of distributing computation. Specifically by employing the potential of the Bitcoin blockchain, we create an alternative form of client puzzle which is fair and avoids imposing heavy computations on clients. We go on to demonstrate the theoretical grounding of these fair client puzzles before describing a proof of concept implementation.

Bitcoin The cryptocurrency Bitcoin, in common with its many altcoin variants, requires a computational problem to be solved every time the set of transactions moves forward by formation of a new valid block in the consolidated blockchain. The computational problem is *moderately hard* which means that it is too hard for a single device to solve in a reasonable time, yet easy enough that a dedicated effort by distributed teams of solvers can obtain a solution within a predictable time. The Bitcoin rules ensure that the current difficulty of the computational problem involved is tuned to the available computational effort worldwide so that the expected time to solve a problem remains more or less constant.

The inspiration for the problems used in Bitcoin originally came from the idea of proofs of work. Proofs of work are designed to be solvable by one specific client without regard to its computational power, with solution time varying depending on the local computing power. Whilst in contrast, puzzles in Bitcoin are solvable at a predictable rate.

Based on the above observations, in this paper we propose client puzzles based on the blockchain. We then describe their use as a mechanism for mitigation of denial of service attacks. This is, in a sense, coming full circle by bringing back the *moderately hard* puzzles to a purpose they were originally designed for. At the same time, the return comes with significant benefits due to the distributed way that puzzles are solved in Bitcoin. The puzzles that we define are fairer than conventional client puzzles as they do not affect, or rely upon, the local computation of users. In fact, the fairness we allude to is *order preserving*, which is inherited from the blockchain.

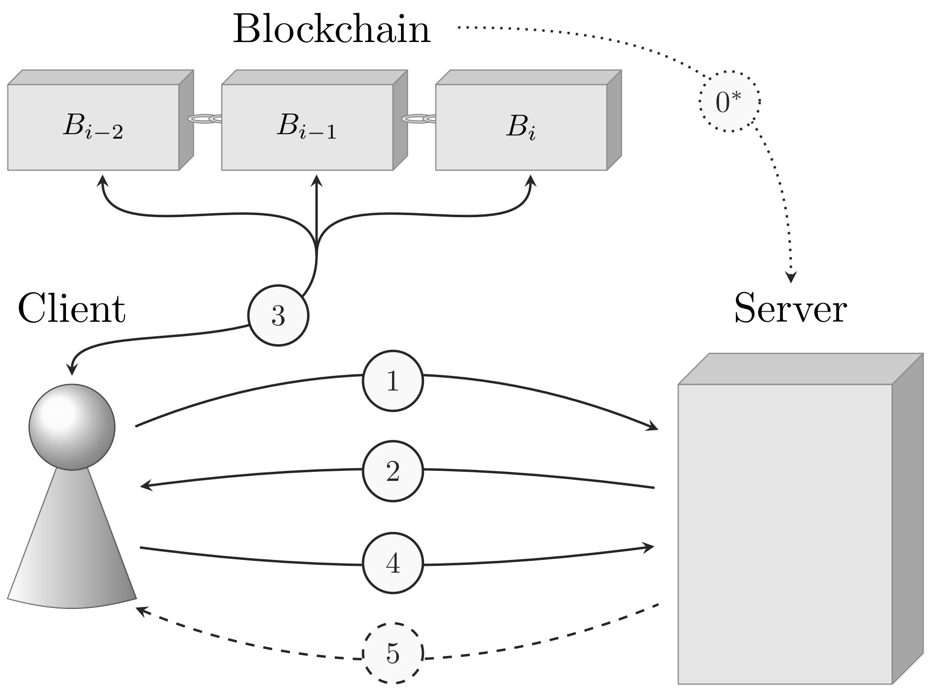

Figure 1: Client puzzles from Bitcoin

Contributions We present a novel client puzzle construction that leverages the distributed computing power of the Bitcoin network to solve them. Our central idea is to embed puzzles within the blockchain as part of the puzzle solution mechanism, and requiring clients to prove that their puzzle is contained in it. Instantiating an interactive client puzzle construction, we show that the puzzles we construct come with considerable practical advantages for both clients and service providers:

1. Any service provider can implement these puzzles, without any requirement for affiliation with the Bitcoin network. Moreover, servers can tune their puzzles to suit their needs, adjusting puzzle difficulty on the fly.

2. The puzzles provide order preserving *fairness*, a previously missing but important goal of client puzzles. In particular, service providers can implement puzzles without concern for hardware availability of their potential clients.

3. The puzzles are adaptable to various Bitcoin-like alternatives (*altcoins*) and other distributed hash chaining schemes.

4. These constructions save expending any individual computational effort of client devices, in contrast with previous client puzzle schemes, by using the work already being produced within the Bitcoin network.

Summarising the DoS resistant attributes, we describe a protocol that operates between a client, or multiple clients, and a server, which may have no association with the Bitcoin network. The protocol runs by the client communication between the blockchain and the server. Referring to Figure 1, steps 1 and 2 correspond to a regular request and response round, where the client receives a puzzle in response from the server. Inserting the puzzle in the blockchain makes up step 3, and is performed by the client. Finally, step 4 is the verification step where the client provides proof that the puzzle is contained within the blockchain. Step 0* is an ongoing process where the server takes information from the blockchain, such as block difficulty. The fifth stage represents general service to the clients, provided that the server accepts at stage 4.

Structure of the paper First we introduce client puzzles, with definitions related to the literature [6, 12, 17], discussing and formally defining difficult and costly client puzzles along with their respective security games. We introduce an abstract client puzzle and prove that our construction satisfies these definitions. Next, we move from our theoretical underpinning, to practical composition by means of DoS resistance in Section 3, which describes the required properties for DoS resistance, in order to introduce a DoS resistant protocol based on previous work in the literature [17]. The penultimate section demonstrates the proof of concept for DoS mitigation with client puzzles, and discusses how it meets the properties in Section 3. The concluding section looks at possibilities for advancing this line of work.

Throughout, we assume some familiarity with the workings of the Bitcoin system. For a broad background understanding we suggest a mix of online resources [20, 21, 22, 23, 24] and academic publications [4, 5, 15, 19].

2 Client Puzzles

In this section we formally define client puzzles and their security notions, before concluding with a discussion on achieving fair client puzzles. Definition 1 closely follows the one given in the literature [17].

Definition 1 (Client Puzzle)**.** *A client puzzle* CP *is a tuple of three efficient probabilistic algorithms* `Setup`, `GenPuz`, `FindSol` *and a deterministic algorithm* `VerSol`. *Let* λ *be the setup parameter,* $\mathcal{K}$ *the key space,* $\mathcal{D}$ *the difficulty space,* $\mathcal{S}tr$ *the string space,* $\mathcal{P}$ *the puzzle space and* $\mathcal{S}ol$ *be the solution space.*

- `Setup`(1^λ) : *Select* $\mathcal{K}, \mathcal{D}, \mathcal{S}tr, \mathcal{P}, \mathcal{S}ol, k \xleftarrow{r} \mathcal{K}, params \leftarrow (\mathcal{K}, \mathcal{P}, \mathcal{S}ol, \mathcal{D}, \mathcal{S}tr)$ *Return* $(k, params)$.
- `GenPuz`$(k \in \mathcal{K}, d \in \mathcal{D}, str \in \mathcal{S}tr)$: *Return* $p \in \mathcal{P}$.
- `FindSol`$(str \in \mathcal{S}tr, p \in \mathcal{P}, t \in \mathbb{N})$: *Return* $s \in \mathcal{S}ol$ *after at most* t *clock cycles.*
- `VerSol`$(k \in \mathcal{K}, str \in \mathcal{S}tr, p \in \mathcal{P}, s \in \mathcal{S}ol)$: *Return* true *or* false.

All that is left is to define correctness. Let $(k, \mathit{params}) \leftarrow \texttt{Setup}(1^k)$ and $p \leftarrow \texttt{GenPuz}(k, d, \mathit{str})$, where $d \in \mathcal{D}$ and $\mathit{str} \in \mathcal{S}tr$, then there exists $t \in \mathbb{N}$ where

$$\Pr[\texttt{VerSol}(k, \mathit{str}, p, s) = \mathsf{true} \mid s \leftarrow \texttt{FindSol}(\mathit{str}, p, t)] = 1.$$

Since their inception, client puzzles were studied as a means to mitigate DoS, and reconsidered for use against distributed DoS. Using client puzzles for DoS prevention led to problems with initial designs [7, 13, 17] as the standard security definitions for client puzzles were not robust enough to capture DoS resistance.

2.1 Security Notions

We now formalise the security notions for client puzzles in terms of games between an adversary $\mathcal{A}$ and a server S. The most obvious way in which an adversary can undermine a client puzzle is to solve a puzzle quicker than expected. Another possibility, as discussed by Chen et al. [7], is a scenario where an adversary could create puzzles independently from the server S. For an interactive setting we desire that an adversary cannot create a puzzle that the server believes to be valid, referred to as *puzzle unforgeability.* In fact, there are a variety of properties required for DoS mitigation which may or may not be useful as properties for client puzzles. We return to this in Section 3.

To define the security games for client puzzles, we first define meaningful oracle calls to help describe them.

- $\texttt{O.GetPuz}(\mathit{str})$: Return $p \leftarrow \texttt{GenPuz}(k, d, \mathit{str})$ and record (str, p) in a list.
- $\texttt{O.GetSol}(\mathit{str}, p)$: If (str, p) was not recorded by $\texttt{O.GetPuz}$ return $\perp$. Else find s such that $\texttt{VerSol}(k, \mathit{str}, p, s) = \mathsf{true}$. Record (str, p, s) and return s.
- $\texttt{O.VerSol}(\mathit{str}, p, s)$: Return true if $\texttt{VerSol}(k, \mathit{str}, p, s) = \mathsf{true}$, (str, p) has been recorded by $\texttt{O.GetPuz}$ and the tuple (str, p, s) has not been recorded by $\texttt{O.GetSol}$. Else return false.

Let the security game $\texttt{EXEC}^{\mathsf{DIFF}}_{\mathcal{A},d,\mathsf{CP}}(\lambda)$ between an adversary $\mathcal{A}$ and server S, for a client puzzle CP with setup parameter λ and difficulty $d \in \mathcal{D}$, be defined as follows:

1. Server S performs $(k, \texttt{params}) \leftarrow \texttt{Setup}(1^\lambda)$, and gives $\texttt{params}$ to $\mathcal{A}$.
2. Adversary $\mathcal{A}$ is allowed to make queries to $\texttt{O.GetPuz}$ and $\texttt{O.GetSol}$.
3. At any point $\mathcal{A}$ can run $\texttt{O.VerSol}$, which terminates the game with the result from the algorithm.

Definition 2 (Difficult Client Puzzle)**.** *Let CP be a client puzzle for fixed setup parameter λ and $d \in \mathcal{D}$. Let $f_{\lambda,d}(t)$ be a family of monotonically increasing functions on t. Then CP is said to be* $f_{\lambda,d}(t)$-difficult *if for every $k \in \mathcal{K}, \mathit{str} \in \mathcal{S}tr, p \in \mathcal{P}$, $s \in \mathcal{S}ol$ and all adversaries $\mathcal{A}$ running in time at most t,*

$$\Pr[\texttt{EXEC}^{\mathsf{DIFF}}_{\mathcal{A},d,\mathsf{CP}}(\lambda) = \mathsf{true}] \leq f_{\lambda,d}(t).$$

2.2 Difficult Client Puzzles from the Blockchain

Now we describe a client puzzle based on hash functions, that are required to produce outputs starting with a certain number of leading zeros. This is similar to a type of puzzle that appears in both the literature and in practice [3, 9]. It is adapted here to describe a high level hash based puzzle, creatable using the blockchain.

Let $\mathcal{H}_i, 1 \leq i \leq 4$ be publicly available hash functions running in polynomial time, taking inputs in $\{0,1\}^*$ and producing outputs in $\{0,1\}^{l_i}$. Then define publicly available algorithms as follows:

- `BC.VerTx` on input x, p, m_1 runs $\tilde{x} \leftarrow (m_1, \mathcal{H}_1(p), \mathcal{H}_2(m_1, \mathcal{H}_1(p)))$ and returns true if $x = \tilde{x}$. Else returns false.

This process verifies that a transaction x contains a puzzle p along with some auxiliary data m_1, which captures the extra information required to form a transaction.

- `BC.Merk` takes up to n inputs $\{t_{x_1}, \ldots, t_{x_n}\}$, for some fixed integer n, and outputs a Merkle root r_t, by forming a hash tree with hash function $\mathcal{H}_3$ and returning a single hash output.

The algorithm `BC.Merk` takes n ordered inputs, under the assumption that the maximum number of inputs to `BC.Merk` is some polynomial on the input parameter, constructing a Merkle tree [14]. These inputs, t_{xi} can be thought of as the transactions within Bitcoin that are contained within each block. The first input t_{x_1} is hashed with the second input t_{x_2}, using the hash algorithm $\mathcal{H}_3$, producing an output in $\{0,1\}^{l_3}$. This output is fed back into $\mathcal{H}_3$ again with the next original input t_{x_3}, and so on, until all n inputs have been included:

$$r_t = \mathcal{H}_3(\ldots \mathcal{H}_3(\mathcal{H}_3(t_{x_1}, t_{x_2}), t_{x_3}) \ldots).$$

- `BC.VerMerk` takes input r_t, a Merkle root, and up to n inputs $t_{x_1}, \ldots, t_{x_n}$, then runs $\tilde{r_t} \leftarrow$ `BC.Merk`$\{t_{x_1}, \ldots, t_{x_n}\}$, and returns true if $r_t = \tilde{r_t}$. Else returns false.

This verification algorithm checks that the submitted Merkle leaves form into the correct Merkle root.

- `BC.VerBlk` takes inputs s', r_t and m_2, returning true if $\mathcal{H}_4(s', r_t, m_2)$ has d leading zeros. Else returns false.

This process mimics the Bitcoin block verification algorithm, where s' is the solution and r_t is the Merkle tree containing the transactions and m_2 is any auxiliary or extra data to be included.

Abstract Blockchain Based Client Puzzle. Using these algorithms, we can now define an *abstract blockchain-based client puzzle* by instantiating the algorithms in Definition 1 as follows:

- $\texttt{Setup}(1^\lambda)$: Selects $\mathcal{K} = \emptyset$, $\mathcal{D} \subseteq \{0, 1, \ldots, l_4\}$, $\mathcal{S}tr \subseteq \{0,1\}^*$, $\mathcal{P} \subseteq \{0,1\}^*$, $\mathcal{S}ol \subseteq \{0,1\}^*$ and returns $\texttt{params} = (\mathcal{D}, \mathcal{S}tr, \mathcal{P}, \mathcal{S}ol)$.
- $\texttt{GenPuz}(d, str)$: Returns puzzle $p = str$.
- $\texttt{FindSol}(str, p)$: Returns s of the form $s = (s', r_t, x, t_{x_2}, \ldots, t_{x_q}, m_1, m_2)$.
- $\texttt{VerSol}(p, s)$ where $s = (s', r_t, x, t_{x_2}, \ldots, t_{x_q}, m_1, m_2)$: Returns true if $\texttt{BC.VerTx}(x, p, m_1)$ returns true, $\texttt{BC.VerMerk}(r_t, x, t_{x_2}, \ldots, t_{x_q})$ returns true and $\texttt{BC.VerBlk}(s', r_t, m_2)$ returns true. Else returns false.

Assuming $\mathcal{H}_4$ is at least surjective, it is easy to find a t such that correctness holds. Note that we purposely omit describing the inputs to the client puzzle algorithms where they are not used; in this instance the client puzzle is not keyed, so we do not list $k \in \mathcal{K}$ as an input to either $\texttt{GenPuz}$ or $\texttt{VerSol}$.

Theorem 1. *Let CP be an abstract blockchain based client puzzle, for security parameter λ, $d \in \mathcal{D}$, and let $\mathcal{H}_i$, $1 \leq i \leq 4$ be random oracles, with output lengths $l_1, l_3, l_4 \geq d$. Let $f_{\lambda,d}(t+1) = \frac{t}{2^d}$. Then CP is an $f_{\lambda,d}(t)-$difficult client puzzle.*

Proof. For Adversary $\mathcal{A}$ to win the game, it equates to finding an s' such that $\mathcal{H}_4(s', r_t, m)$ has at least d leading zeros. $\mathcal{A}$ first queries $\texttt{GenPuz}$ on some string str, returning the puzzle $p = str$, thus (str, str) is now recorded. This step must take place, otherwise $\texttt{O.VerSol}$ could only output false. Next, $\mathcal{A}$ generates a valid transaction x, and Merkle root r_t, using the public algorithms $\mathcal{H}_1, \mathcal{H}_2$ and $\mathcal{H}_3$. This first process takes at least one step. All that is left is to find a valid s' such that $\mathcal{H}_4(s', r_t, m)$ has d leading zeros. W.l.o.g. fix m, then $\mathcal{A}$ proceeds to run the function $\mathcal{H}_4(s', r_t, m)$ for different values of s'. For any one run, the chance of finding a valid s' satisfying this condition is $1/2^d$. If a solution has not been found after the penultimate attempt, $\mathcal{A}$ may attempt a guess for s', along with the call to $\texttt{O.VerSol}$. Hence, the probability of $\mathcal{A}$ succeeding, and $\texttt{O.VerSol}$ returning true, is bounded by $t/2^d$. $\square$

This client puzzle construction can be realised using the blockchain. Finding a solution to a puzzle is simply the process of encapsulating it within a transaction in a subsequent block. Thus, the difficulty must be chosen with respect to the difficulty specified by the blockchain protocol. For the Bitcoin system, taking difficulty d from the blockchain makes it improbable for any individual to solve a puzzle in a reasonable amount of time. Rather than expect a user to mine a block, we rely on the Bitcoin system to solve the puzzle. Consequently, a client needs only to form a transaction on the blockchain including the puzzle, then wait for the miners to find a solution to the block – hence the puzzle. This client puzzle effectively replaces computational cost by a time delay. Constructing a client puzzle is demonstrated in Section 4, with a proof-of-concept implementation.

2.3 Stronger Security from Cost-Based Puzzles

Definition 2 only captures the difficulty of solving one puzzle in t computational steps. This does not fully express the intuition behind client puzzles [9, 17], as it says nothing about the possibility of attempting to solve many puzzles at once. We want client puzzles where solving n puzzles is approximately n times as difficult as solving just one. Client puzzles with this property are referred to as *strongly difficult*. We formalise this notion slightly differently, by employing abstract blockchain based client puzzles, and replacing difficulty in terms of computational steps by *cost* in terms of monetary expense.

Despite a time delay feature, the abstract scheme as described is free of any significant cost for the clients and whilst this is the case clients are able to solve as many puzzles as they like provided they do not mind waiting. What is more, any client can solve multiple puzzles in the same time it takes to solve one. Later we will see that both of these problems can be addressed using Bitcoin by imposing a monetary cost to embedding puzzles within a block, and thus finding a solution. Formalising this idea is achieved in a similar manner to the work on *interactive strong puzzle difficulty*, originally given by Stebila et al. [17]. For now, we need to define the security experiment, and make adversarial assumptions.

Define the security game for $\mathtt{EXEC}^{\mathsf{CCP}}_{\mathsf{CP},\mathcal{A},d,n,c_t}(\lambda)$ between a p.p.t adversary $\mathcal{A}$ and server S, for a client puzzle CP with setup parameter λ, $n \geq 1$ and a predetermined cost $c_t \geq 0$ as follows.

1. Server S performs $(k, \mathtt{params}) \leftarrow \mathtt{Setup}(1^\lambda)$, and gives $\mathtt{params}$ to $\mathcal{A}$.
2. Server S runs $p_i \leftarrow \mathtt{O.GetPuz}(str_i)$ for distinct, random $str_i, i = 1, \ldots, n$. All (str_i, p_i) are recorded and given to $\mathcal{A}$.
3. Adversary $\mathcal{A}$ has oracle access to $\mathtt{O.GetSol}$ and a solution verification oracle taking inputs (str, p, s) and returning $\mathtt{VerSol}(k, str, p, s)$. Additionally, $\mathcal{A}$ may solve any puzzle p_i at a fixed cost c_t.
4. Adversary $\mathcal{A}$ generates $L = \{(str_i, p_i, s_i) : i \in \{1, \ldots, n\}\}$.
5. At any point $\mathcal{A}$ can end the game by submitting L as a solution. The algorithm returns true if for all i from 1 to n, $\mathtt{O.VerSol}(str_i, p_i, s_i) = \mathsf{true}$.

With this new version of the puzzle security game we can now define a *costly client puzzle* as an analogue of the strongly difficult client puzzles [17].

Definition 3 (Costly Client Puzzle)**.** *Let CP be a client puzzle, let $f_{\lambda,d,n}(t)$ be a family of functions monotonically increasing in t and let negl be a negligible function where*

$$|f_{\lambda,d,n}(t) - f_{\lambda,d,1}(t/n)| \leq negl(\lambda, d),$$

for all t, n such that $f_{\lambda,d,n}(t) \leq 1$. For a fixed setup parameter λ and difficulty parameter $d \in \mathcal{D}$, for $n \geq 1$, then CP is an $f_{\lambda,d,n}(t)-$costly client puzzle if for all *p.p.t algorithms $\mathcal{A}$ running in time at most t, where each puzzle has associated cost $c_t > 0$,*

$$\Pr[\mathtt{EXEC}^{\mathsf{CCP}}_{\mathsf{CP},\mathcal{A},d,n,c_t}(\lambda) = \mathsf{true}] \leq f_{\lambda,d,n}(t).$$

Remark 1. An adversary attacking the strong security of a blockchain based client puzzle may attempt to embed multiple puzzles in one block. This may be possible, and in fact is a design goal of blockchain based puzzles. However, as the number of embedded puzzles per block increases, the average computational cost expended per puzzle decreases. Therefore the design of the puzzle should ensure that the number of puzzles embedded in a block is limited to some threshold n. Furthermore, the cost to the adversary of mining a block should remain (much) higher than the cost of solving one puzzle in the intended manner, namely embedding it in a valid Bitcoin transaction.

To achieve strong security of a blockchain-based client puzzle we assume that the adversary is unable to reduce the average cost of finding solutions to puzzles below a certain threshold cost c_t which we call the *design cost* of the puzzle. The design cost can be assigned in different ways, such as applying transaction fees or "proof-of-burn"[22] (to list just two methods). Note that c_t is a monetary cost which we freely compare with computational cost. This is important, as it means clients can outsource their puzzles for a cost, which is itself equatable to computation, meaning the outsource is not achieved for free.

Returning to our abstract puzzle definition, the adversary can attempt to solve multiple puzzles at once within a single Merkle root r_t, and finding one s' and auxiliary m such that $\mathcal{H}_4(s', r_t, m)$ has the requisite number of leading zeros. We need this to cost more than the cost of solving the maximum number of puzzles which can be embedded in one block.

Assumption 1. *Let CP be an abstract blockchain based client puzzle, with security parameter λ. Let c_t be the design cost of solving one puzzle, and fix a parameter of difficulty d. Then there exists $q \in \mathbb{Z}_{>0}$ where, for all r_t and m, the average computational cost of finding an s' such that $\mathcal{H}_4(s', r_t, m)$ has d leading zeros, is at least $q \cdot c_t$.*

In our Bitcoin embodiment, Assumption 1 expresses the idea that generating a block is so stupendously expensive that it is cheaper to bear the cost of embedding a significant number of puzzles. In practice there is a limit on the number of puzzles that can be embedded in one Bitcoin block due to a fixed limit on the size of blocks. Each block has a capped limit of 1Mb [22] and a puzzle included within a transaction with one input and one output is approximately 214 bytes, so currently just under $4,900$ transactions are embeddable per block. Thus for Bitcoin this can be a realistic choice for q.

Using Assumption 1 we can prove that the abstract blockchain based puzzle is a costly client puzzle. Thus we can use this puzzle to ensure that an adversary who wishes to solve puzzles must incur a cost that is linear in the number of puzzles they intend to solve, which is a significant disincentive to carrying out DoS attacks by attempting to solve multiple puzzles. This can be an attractive mechanism to use in practice on the assumption that legitimate users are willing to accept a single fee c_t in order to maintain access to a service during times of attack. We explore the use of such client puzzles in more detail in Section 3.

Theorem 2. *Let CP be an abstract blockchain based client puzzle, for security parameter λ, $d \in \mathcal{D}$, cost per puzzle c_t, and let $\mathcal{H}_i$, $1 \leq i \leq 4$ be random oracles, with output lengths $l_1, l_3, l_4 \geq d$. Let $f_{\lambda,d,n}(t) = (t - c_t(n-1))(n-1)/2^d$. Then CP under Assumption 1 with $q > n$ is an $f_{\lambda,d,n}(t)-$costly client puzzle.*

Proof. Following from the assumption, if $\mathcal{A}$ attempts to attach the received puzzles within a Merkle root r_t, and then tries to compute an s', such that $\mathcal{H}_4(s', r_t, m)$ has d leading zeros, it becomes far too costly, i.e. the total cost would be $q \cdot c_t$, which is much more expensive than if $\mathcal{A}$ simply won the game by paying for each solution.

Another strategy for $\mathcal{A}$ is to ask for the solution to all but one puzzle, each costing c_t computational steps, leaving $\mathcal{A}$ with $t - c_t(n-1)$ steps remaining. Now, $\mathcal{A}$ can generate a legitimate transaction $x \leftarrow (m_1, \mathcal{H}_1(p), \mathcal{H}_2(m_1, \mathcal{H}_1(p)))$. Note that for each of the paid for $n-1$ puzzle and solution pairs, (p_j, s_j), the solution contains a Merkle root r_{t_j}. Thus $\mathcal{A}$ can attempt to form a Merkle root r_t by finding some value m', such that $\mathcal{H}_3(x, m')$ gives $r_t = r_{t_j}$ for any j. This approach is essentially attempting to find a collision within one of the $n-1$ Merkle trees. The output space of $\mathcal{H}_3$ is 2^{l_3}, so the probability of finding a solution in t steps is $(t - c_t(n-1))(n-1)/2^{l_3}$. As $l_3 > d$ we have $(t - c_t(n-1))(n-1)/2^{l_3} \leq (t - c_t(n-1))(n-1)/2^d$ as required. □

2.4 Achieving Fair Client Puzzles

So far we have explored difficult and costly client puzzles, but have not considered fairness. Why do we need fairness in client puzzles? Consider a case that behaves very similarly to a DoS attack, where multiple legitimate clients request service all at once, referred to as a *flash flood* scenario. Here, the natural approach is to serve clients on a first come, first served basis. However, a dated smart phone would be able to perform significantly less operations per second than a high end desktop computer, and so would be at a disadvantage if it were forced to solve client puzzles to gain access. These are fairness considerations.

Essentially, the fairness required is order preserving. We state this informally as: for any two clients C and C', if C attempts to solve a puzzle before C', it should find a solution no later than C'. For our abstract blockchain-based client puzzle, instantiated with the Bitcoin blockchain, this becomes achievable. There is an order preserving algorithm, namely the process of adding a transaction to the blockchain. It is also reasonable to assume that any adversary we are concerned with would not generate a block before the worldwide collection of Bitcoin miners. The Bitcoin protocol cannot be fair in an ideal way due to factors such as transaction fees and availability of nearby nodes, which affect the time until a transaction is included in the blockchain. However, in general complete availability is not achievable, and we must assume, along with the built in assumptions on Bitcoin, that provided the correct transaction fees are included nodes do not arbitrarily reject certain transactions.

3 Application to Denial of Service Resistance

Denial of service (DoS) attacks, including distributed denial of service (DDoS) attacks, are popular and widely employed techniques of attacking web based resources and services, causing considerable concern for online service providers[1]. DoS is an easily exploitable attack vector, challenging to defend against and relatively simple to invoke. Despite the awareness and wide acknowledgement of the problem, DoS is still a prevalent threat.

A conventional DoS attack, and the one we focus on, aims to deplete the computational resources of the server by requesting and re-requesting connection [17]. If the server demands secure authentication this will require expending computational effort, which the attacker exploits by not completing the authenticated handshake. This type of DoS attack makes the effort expended asymmetric. The attacking clients are free to ignore all replies, thus expending no effort other than a request for service, whilst the server expends considerable effort for each request. Ameliorating this imbalance is the aim of this section, using the abstract client puzzle construction as described so far. The intention is to incorporate these client puzzles within a pre established DoS resistant framework.

3.1 Client Puzzles for Denial of Service Resistance

The criteria provided below are extracted from recent literature on DoS resistance using client puzzles [6, 7, 9, 18], with the exception of Criterion 4, which expresses the notion of fairness developed in Section 2.4. The term, *expensive operations* is left ambiguous, since it depends on capabilities of the server.

1. A server S should not carry out any *expensive operations* unless it believes that the client C legitimately wants service.
2. Server S should not perform any *expensive operations* unless it believes that client C intends to communicate with S, and not with another server $\tilde{S}$.
3. All cost incurred by the client C in order to solve a client puzzle for server S, can only be used to prove C's intentions to communicate with S.
4. Any client C, that attempts to solve a challenge p from server S, before any other client $\tilde{C}$ attempts to solve challenge $\tilde{p}$, should find a solution to p no later than $\tilde{C}$ finds a solution to $\tilde{p}$.

[1]Throughout we use DoS to refer to both DoS and DDoS, when not explicitly stated otherwise.

5. The cost of solving n puzzles is approximately n times the cost of solving 1 puzzle.

6. No party other than S should be able to generate valid puzzles for S, with non-negligible probability.

7. Any client that receives puzzles from S cannot solve and store the puzzles, then later respond with an overwhelming number of legitimate puzzle solutions at one time.

8. It is not possible to replay previously accepted puzzle solutions.

9. The cost required for solving a puzzle is adjustable.

Criterion 2 is designed to prevent a malicious server redirecting requests for service to a legitimate server S. This exploit is discussed in detail by Mao and Paterson [13], and is crucial to DoS resilience. The concern here is that, with almost no effort expended by the malicious server $\tilde{S}$, it is possible to forward all requests on to a legitimate server S, where the client believes it is communicating with $\tilde{S}$, and the server S believes it is communicating with C. Another property, Criterion 6, refers to the unforgeability of client puzzles. For interactive client puzzles, this is necessary, however some client puzzles have been designed specifically to be non-interactive, so that a client can generate a valid puzzle. For non-interactive puzzles, unforgeability is meaningless.

3.2 Creating a DoS Resistant Protocol

Stebila et al. [17] demonstrate a way of transforming any *interactive strongly difficult* client puzzle into a DoS resistant protocol. This transformation is also achievable for any interactive costly client puzzles as described here. First observe that the only difference between costly client puzzles and strongly difficult ones is the way that cost is measured. We can create an upper bound in terms of cost, which is equatable with computational steps, and thus use the proof from Steblia et al. [17, Eq 7, p.28].

Building on this construction, we demonstrate the use of our abstract blockchain-based client puzzle in a DoS resistant scenario, by means of an example – see Figure 2. This protocol is constructed as detailed by Stebila et al. [17, § 5]. We assume that server and client have public identities id_s and id_c. The aim is for the server to accept an identity from the client and store the identity in a list of accepted identities only if a valid run of the protocol for that identity has taken place. The intention here is that the protocol is run prior to any further, perhaps more expensive, demands on the server, such as key agreement. However, the more general construction [17] allows for any protocol to be built into the first 3 stages, provided that the server does not perform any expensive computation until after the response has been accepted – so after line 15 in Figure 2.

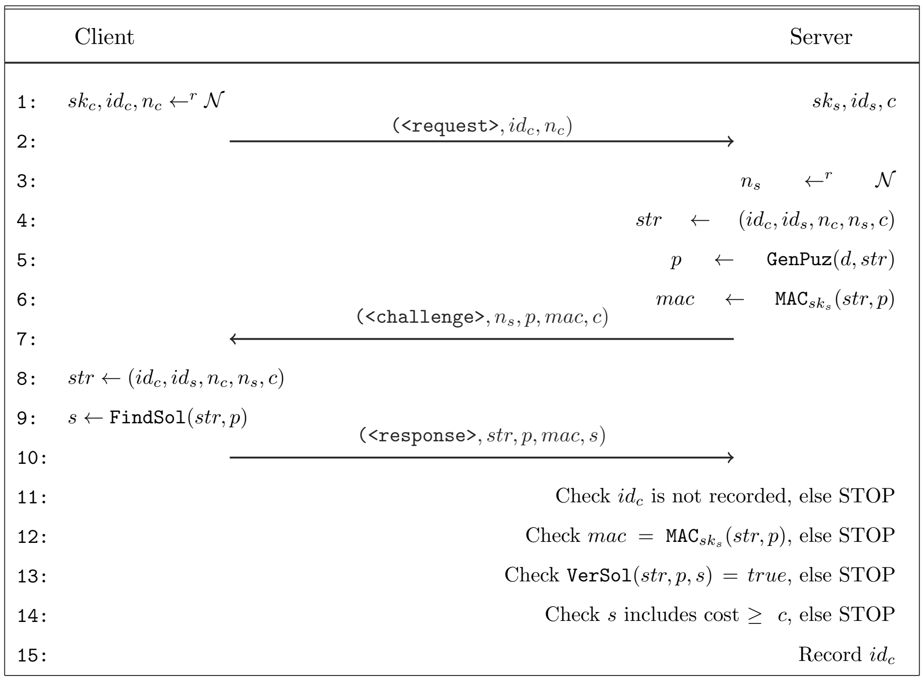
Client
Server
1: $sk_c, id_c, n_c \xleftarrow{r} \mathcal{N}$
sk_s, id_s, c
2: $(\texttt{<request>}, id_c, n_c)$
3: $n_s \xleftarrow{r} \mathcal{N}$
4: $str \leftarrow (id_c, id_s, n_c, n_s, c)$
5: $p \leftarrow \texttt{GenPuz}(d, str)$
6: $mac \leftarrow \texttt{MAC}_{sk_s}(str, p)$
7: $(\texttt{<challenge>}, n_s, p, mac, c)$
8: $str \leftarrow (id_c, id_s, n_c, n_s, c)$
9: $s \leftarrow \texttt{FindSol}(str, p)$
10: $(\texttt{<response>}, str, p, mac, s)$
11: Check id_c is not recorded, else STOP
12: Check $mac = \texttt{MAC}_{sk_s}(str, p)$, else STOP
13: Check $\texttt{VerSol}(str, p, s) = true$, else STOP
14: Check s includes cost $\geq c$, else STOP
15: Record id_c

Figure 2: Abstract Blockchain DoS Resistant Protocol

3.3 Meeting the Denial of Service Resistant Criteria

Relating to the DoS resistance criteria specified in Section 3.1, notice that for most of the desired properties it is straightforward to see that they are achieved. Here, we expand further on the less obvious criteria.

Point 4 holds with the assumption that once a transaction is created and broadcast, it will be included in the next block, which is a fundamental assumption discussed previously discussed in Section 2.4. Thus, under this assumption, we achieve an order preserving fairness where puzzle solutions cannot be found any sooner than solutions to earlier puzzles.

Observe that point 7 is not satisfied in this instance. Originally noted by Groza and Warinschi [9], the proposed method for adapting client puzzles for DoS prevention [17] provides no defence against a next–day attack, as there is no limit on the time allowed to return a solved puzzle. A simple solution involves the server changing keys at regular intervals. To accomplish this, whilst maintaining fairness for clients, the server generates and stores two keys: an old key sk_{s_1} and a new key sk_{s_2}. The server then periodically updates the keys, redeclaring the current newest as the oldest $sk_{s_1} = sk_{s_2}$, then generating and storing a new key sk_{s_2}. The new key, sk_{s_2}, is then used to generate the MAC on line 6 of Figure 2, but the verification on line 12 will have to test both keys and continue if the MAC returned by the client is valid under one of the keys.

We have excluded this rekeying process from the protocol for two reasons. Firstly, it further convolutes an otherwise straightforward protocol, and is very easy to solve. Secondly, perhaps most crucially, we expect that a server would only employ this DoS mitigation protocol when receiving high volumes of traffic, and thus generate a fresh key at each instantiation. Unless there is some sustained attack, there may be no need to implement a rekeying procedure. Thus, we leave this process as an optional mechanism.

Point 3 is slightly trickier, as there is nothing stopping a client solving a puzzle on the behalf of someone else. However, if we consider an established beneficiary of C to be a part of C itself, it becomes clear that this property is also achieved, by virtue of the string generation on the client side (Step 8 Fig 2), and the subsequent MAC verification on the server side (Step 12, Fig 2).

4 Proof of Concept

We provide a practical demonstration of the protocol described in Figure 2, by creating an example puzzle and storing it within a transaction using the Bitcoin Testnet.

Suppose a client C wishes to register identity id_c with a server S, in order to later receive some service. First C generates a nonce n_c, and sends it along with id_c to the server S. In response, the server generates and returns a puzzle p, a server nonce n_s, a cost c and a message authentication code mac.

To create the response, the server first forms the string str, which is made up of the server address id_s and the client address id_c, along with their nonces and the server specified cost c. For our example, we take

$$
\begin{aligned}
id_c &= \texttt{mjMnKihRdbr5VVdDV67QGd13EVnUBm6F7k}\\
id_s &= \texttt{mmsU7xHjJLdiqgL3udyJ7oooNXTo94M9nE}\\
n_c &= \texttt{27e7e82f79c5ab86e99fcf7024fd4003b87fc8a7eb99fd00e77d9ba55a02e197}\\
n_s &= \texttt{d053c6e0c1756705b0abfb3ff9374e85a5c1e85d9ed7481f1b61926dcce17f9f}\\
c &= \texttt{1}\\
str &= (id_c||id_s||n_c||n_s||c)
\end{aligned}
$$

Addresses id_c and id_s are encoded in the Bitcoin specified base-58, whilst the nonces used are hexadecimal. Cost c is represented in decimal with cost 1 equating to 10^{-7} bitcoins. In this instance the generate puzzle algorithm, `GenPuz`, simply returns the string $p = str$, as the difficulty on the blockchain does not need to be known to create puzzles.

Now the client can solve the puzzle p by encapsulating it within the next block. One way of inputting this puzzle into the blockchain is to form the puzzle into an address, which requires running the puzzle p through the Bitcoin public key to address generation algorithm. The client does this by running $p' \leftarrow \texttt{RIPEMD}^{160}(\texttt{SHA-256}(p))$, then prepends some auxiliary message data m_1

to p', then takes m_2, the first four bytes of SHA-256(SHA-256($m_1||p'$)), and appends them to $m_1||p'$ to form the full output $m_1||p'||m_2$. This process is designed to agree with the address generation method of Bitcoin.

Running this algorithm on the string above we get `mwwCiu...p7NS` encoded in base-58. This is now a legitimate Testnet address. The client creates a transaction including this address and cost c, then sends it to the network.

```
1: "vout" : [
2:      {
3:          "value" : 0.00000010,
4:          "n" : 0,
5:          "scriptPubKey" : {
6:              "asm" : "OP_DUP OP_HASH160 b4180a2
7:              fdaef5c1afdc3b0e73fe699094e634ff7
8:              OP_EQUALVERIFY OP_CHECKSIG",
9:              "hex" : "76a914b4180a2fdaef5c1afdc
10:             3b0e73fe699094e634ff788ac",
11:             "reqSigs" : 1,
12:             "type" : "pubkeyhash",
13:             "addresses" : [
14:                 "mwwCiu1hfeJVppaWtfAHCWwKZ2j57Fp7NS"
15:             ]
16:         }
17:     },
```

Above is a portion of a Bitcoin Testnet transaction, created to include the puzzle in the block. Line 14 specifies the receiving address, which we recognise as $m_1||p'||m_2$, a function of the puzzle p. On line 3, the value specifies the number of bitcoins assigned to the address. This is the specified cost. The full transaction is described in Appendix A.

It is now possible to verify that the transaction is recorded in the block, within transaction $x = \texttt{c55}\ldots\texttt{2d5}$, which is a double SHA-256 of the complete transaction. This transaction is then encoded within the block $\texttt{000}\ldots\texttt{aa4}$, which can be verified using the block's details.[2] This also allows us to verify the cost associated with the transaction, and note that the address of the puzzle is assigned 0.00000010 bitcoins, as specified by the puzzle – we are using a cost of 1 to be equivalent to 10^{-7} bitcoins. The cost is a protocol level specification, verified at line 14 in the figure. Merkle root creation and block generation are performed by the mining nodes, encompassing the transaction within the blockchain. Once this process is complete, the client can return the valid string str, p, s, m.

Upon receiving the response, the verification algorithms `BC.VerMerk` and `BC.VerBlk` are run by the server, which also take into account the difficulty of Bitcoin. In Bitcoin, the blockchain difficulty remains static for 2016 blocks at

[2] `https://www.blocktrail.com/tBTC/block/000000000000015b97c1c04e9ec0ce7266d837/9dceac7e3b0a38a32872687aa4`

a time. For this example, the difficulty d was set at 227267.00000000. This verification process is incredibly cheap in terms of computation, and is similar to the process used when participating as a node in the Bitcoin network. In simulations, it was possible to generate one hundred thousand puzzles in 1.225 seconds, with the MAC verification step taking on average 1.061 seconds per hundred thousand puzzle responses.

Provided the straightforward verification stages 11, 12 and 14 from Figure 2 return true, the only other verification needed is to check that x, returned by the client as part of the solution s, exits as a transaction within the block, which includes an encoding of the puzzle p and supplementary data m.

5 Conclusion

There are various ways to extend and adapt this work. One possibility is to develop a rigorous notion of fairness in terms of client puzzles, and relating that to what is achievable with puzzles constructed from the blockchain, along with analysis that focuses on achieving a fairness property in a practical setting. Extending this work by looking at altcoins could lead to more efficient, and potentially simpler client puzzles constructions, running quicker than on the Bitcoin network. This approach may increase the scope of potential applications, though we expect there to be some trade off between security and application. From a broader perspective, research on the willingness of clients to spend money over allowing their machine to perform computations is a nice avenue for investigation. In all, there still remain challenges to developing and deploying practical client puzzles based on Bitcoin.

5.1 Real-time delays

Bitcoin blocks are generated approximately every ten minutes which may seem impractical for client puzzle applications such as in DoS prevention. We emphasise that, like many DoS resistant measures, client puzzles should *only be used during times of stress* for the server, when there is an abnormally high volume of traffic. For typical situations the puzzles are not used and there is zero overhead. During exceptional periods, waiting around 10 minutes for service is not unreasonable depending on the type of service and the circumstance. A good example of this is online ticketing, where frequently online queues are introduced that redirect to backup systems in order to cope with the demand. Here, a waiting time of far more than ten minutes is not uncommon.

Various alternatives to the standard Bitcoin protocol are available, with many altcoins offering significantly lower block generation time. Litecoin, Dogecoin and Quarkcoin are examples of Bitcoin-like schemes offering block generation times of 2.5 minutes, 1 minute and 30 seconds respectively. It must be pointed out that the use of an alternative currency must be weighed up based on an assessment of its own merit, and only provided that Assumption 1 in Section 2.3 is considered reasonable in that context. For example, whilst the

Bitcoin system is producing approximately 747,930,000 giga hashes per second, the quark system is producing much less, at 264.13 mega hashes per second.[3] On top of this, Litecoin, Dogecoin and Quarkcoin are based on different hash functions than Bitcoin, which have not experienced as much scrutiny from the community. Although altcoins have good potential, any assumptions on Bitcoin cannot be trivially carried over to its Bitcoin-like alternatives.

5.2 Implementation challenges

The protocol described in Figure 2 does not limit a third party from including the puzzle in the blockchain on the behalf of a client, allowing the solution finding process to be outsourced. In this way a large service provider, or many smaller providers, could take a prepaid fee in order to allow a client to use their service in the future. This approach also allows the service providers to group multiple puzzle solutions into one transaction, enabling the cost per puzzle solution to be reduced.

A potential concern is that with the space of 1Mb per block, using Bitcoin as a true client puzzle scheme would soon overwhelm the network with transactions. Whilst this is a concern for Bitcoin currently, there may be larger block sizes in the near future[4]. Furthermore, by collecting multiple puzzles into one transaction, via a service provider or otherwise, this will substantially reduce the size of each puzzle. The ability to embed more puzzles in a single block must also be scrutinised in the context of Assumption 1, which depends on factors such as the difficulty of the blockchain, and the design cost per puzzle.

An alternative way of including cost in a puzzle is to include a cost in with the transaction fee of the puzzle. However, the approach we have taken makes it possible to solve multiple puzzles for the cost of one single transaction fee, and therefore we prefer to make the assignment of the value to an unspendable address – the puzzle. A neater alternative is to use special transactions designed specifically for embedding messages within the blockchain, namely OP–RETURN transactions [22]. For completeness, we have also embedded the same puzzle in an OP–RETURN transaction on the Testnet, with a minor change in cost parameters.[5]

This method also allows the cost can be sent to an address the server owns, along with the identifying message. As a result, the server could actually gain monetarily from any attempted DoS attack. It also allows the server to reimburse the cost of solving the puzzle back to the user, albeit without returning the cost of a transaction fee. This approach also comes with the advantage of not increasing the size of the set of unspent transaction outputs (UXTO), which would otherwise have ramifications for those running Bitcoin nodes.

Overall, we have presented a novel approach to client puzzles, providing a fair alternative to previously proposed proof of space and proof of work schemes,

[3]bitinfocharts.com – 19 Jan 16

[4]`https://bitcoinxt.software/index.html` Jan – 2016

[5]`https://www.blocktrail.com/tBTC/tx/97c68c417f4c254b9cefc7c5495aad47bc006b4e8/0dbec0497e35a67f63acb71`

which lack this property. Using Bitcoin's proof of work system for client puzzles is in some ways coming full circle, but we have gained considerably along the way. We can now construct client puzzles which are solvable in a fixed time, and independent of client device. Focus can now be directed towards implementation issues, such as speed, reliability and practical application.

References

[1] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. In *Security and Cryptography for Networks, SCN 2014*, volume 8642 of *LNCS*, pages 538–557. Springer, 2014.

[2] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In Bruce Christianson et al., editors, *Security Protocols, 8th International Workshop*, volume 2133 of *LNCS*, pages 170–177. Springer, 2001.

[3] Adam Back. Hashcash-a denial of service counter-measure. `http://www.hashcash.org/papers/hashcash.pdf`, 2002.

[4] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to better - how to make Bitcoin a better currency. In *Financial Cryptography and Data Security, FC 2012*, volume 7397 of *LNCS*, pages 399–414. Springer, 2012.

[5] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *IEEE Symposium on Security and Privacy, SP 2015*, pages 104–121. IEEE Computer Society, 2015.

[6] Colin Boyd, Juan Gonzalez-Nieto, Lakshmi Kuppusamy, Harikrishna Narasimhan, C Pandu Rangan, Jothi Rangasamy, Jason Smith, Douglas Stebila, and V Varadarajan. Cryptographic approaches to denial-of-service resistance. In *An Investigation into the Detection and Mitigation of Denial of Service (DoS) Attacks*, pages 183–238. Springer, 2011.

[7] Liqun Chen, Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. Security notions and generic constructions for client puzzles. In *Advances in Cryptology, ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 505–523. Springer, 2009.

[8] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology, CRYPTO 1992*, volume 740 of *LNCS*, pages 139–147. Springer, 1992.

[9] Bogdan Groza and Bogdan Warinschi. Revisiting difficulty notions for client puzzles and DoS resilience. In *Information Security, ISC 2012*, volume 7483 of *LNCS*, pages 39–54. Springer, 2012.

[10] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks: Communications and Multimedia Security*, volume 152 of *IFIP Conference Proceedings*, pages 258–272. Kluwer, 1999.

[11] Ari Juels and John G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 1999*. The Internet Society, 1999.

[12] Ghassan Karame and Srdjan Capkun. Low-cost client puzzles based on modular exponentiation. In *European Symposium on Research in Computer Security, ESORICS 2010*, volume 6345 of *LNCS*, pages 679–697. Springer, 2010.

[13] Wenbo Mao and Kenneth G Paterson. On the plausible deniability feature of Internet protocols. `www.isg.rhul.ac.uk/~kp/IKE.ps`, 2002.

[14] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology, CRYPTO 1987*, volume 293 of *LNCS*, pages 369–378. Springer, 1988.

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

[16] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009. `http://bitcoin-class.org/0/classes/class16/scrypt.pdf`.

[17] Douglas Stebila, Lakshmi Kuppusamy, Jothi Rangasamy, Colin Boyd, and Juan Manuel González Nieto. Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols. In *RSA Conference 2011*, volume 6558 of *LNCS*, pages 284–301. Springer, 2011.

[18] Douglas Stebila and Berkant Ustaoglu. Towards denial-of-service-resilient key agreement protocols. In *Australasian Conference on Information Security and Privacy, ACISP 2009*, volume 5594 of *LNCS*, pages 389–406. Springer, 2009.

[19] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IACR ePrint Archive*, 2015:464, 2015.

[20] Web. Ken shirriff www.righto.com. `http://www.righto.com/2014/02/bitcoins-hard-way-using-raw-bitcoin.html`, 2014. [Acc: Oct 15].

[21] Web. Bitcoin Block Explorer. `http://blockexplorer.com/`, 2015. [Acc: Nov 15].

[22] Web. Bitcoin Wiki. `https://en.bitcoin.it/wiki/Main_Page`, 2015. [Acc: Nov 15].

[23] Web. CoinDesk. `http://www.coindesk.com/`, 2015. [Acc: Nov 15].

[24] Web. michaelnielsen.org. `http://www.michaelnielsen.org/ddi/how-the-bitcoin-protocol-actually-works/`, 2015. [Acc: Nov 15].

Appendix

A Constructed Client Puzzle

Below is presented the complete transaction including the puzzle on the Bitcoin Testnet, comprising of 3 main parts. Lines 6 to 29 specify the inputs to the transaction. Lines 31 to 46 assigns 0.00000010 bitcoins to the puzzle (address) we created in Section 4. Lines 47 to 62, specify the value being assigned to an address that is owned by us (the client), which returns the change. Included is a transaction fee of 0.00000010, which is the difference between the total value of the input to the transaction and the total value assigned to the two receiving addresses.

```
1: {
2:   "txid" : "c55d38e3aa21dcfe3f2179b47eb881b5f832
3:   9f3231a21a837c52a1e43a3dd2d5",
4:   "version" : 1,
5:   "locktime" : 0,
6:   "vin" : [
7:       {
8:           "txid" : "d863c7d202d1aa31cc0e802c781b
9:           f1e223f0cc8850886def66e43e8a960348e3",
10:          "vout" : 1,
11:          "scriptSig" : {
12:              "asm" : "304402203e673472cbb2062a1
13:              6d7fb941725ee39db9bdb5af6e7d2c7623
14:              54979fcf1818a0220238d678989bdcf45c
15:              3aa86bf8612a915cd9b39caaa81546be14
16:              f0af68e10c30f0103d3e79264b08929cd4
17:              69398326b2c923430e9fdb68bf841fe134
18:              8146685ee22c9",
19:              "hex" : "47304402203e673472cbb2062
20:              a16d7fb941725ee39db9bdb5af6e7d2c76
21:              2354979fcf1818a0220238d678989bdcf4
22:              5c3aa86bf8612a915cd9b39caaa81546be
23:              14f0af68e10c30f012103d3e79264b0892
24:              9cd469398326b2c923430e9fdb68bf841f
25:              e1348146685ee22c9"
26:          },
27:          "sequence" : 4294967295
```

```
    }
 ],
 "vout" : [
    {
        "value" : 0.00000010,
        "n" : 0,
        "scriptPubKey" : {
            "asm" : "OP_DUP OP_HASH160 b4180a2
            fdaef5c1afdc3b0e73fe699094e634ff7
            OP_EQUALVERIFY OP_CHECKSIG",
            "hex" : "76a914b4180a2fdaef5c1afdc
            3b0e73fe699094e634ff788ac",
            "reqSigs" : 1,
            "type" : "pubkeyhash",
            "addresses" : [
                "mwwCiu1hfeJVppaWtfAHCWwKZ2j57Fp7NS"
            ]
        }
    },
    {
        "value" : 0.13999980,
        "n" : 1,
        "scriptPubKey" : {
            "asm" : "OP_DUP OP_HASH160 fd2e085
            3c80c493fc4f1bd1514e56c885f9c6f29
            OP_EQUALVERIFY OP_CHECKSIG",
            "hex" : "76a914fd2e0853c80c493fc4f
            1bd1514e56c885f9c6f2988ac",
            "reqSigs" : 1,
            "type" : "pubkeyhash",
            "addresses" : [
                "n4bePyywwEu13TKefhFfyXH7qTXppmiHTn"
            ]
        }
    }
 ]
}
```