# PREPRINT: Nonoutsourceable Scratch-Off Puzzles to Discourage Bitcoin Mining Coalitions

Andrew Miller, Elaine Shi, Ahmed Kosba, and Jonathan Katz University of Maryland, College Park

 ${amiller, elaine, akosba, jkatz}@cs.umd.edu$ 

This is a preprint, and may contain errors. Last updated 2014/07/15 03:23:52. A newer version may be found at https://cs.umd.edu/~amiller/nonoutsourceable.pdf.

# ABSTRACT

An implicit goal of Bitcoin's reward structure is to diffuse network influence over a diverse, decentralized population of individual participants. Indeed, Bitcoin's security claims rely on no single entity wielding a sufficiently large portion of the network's overall computational power. Unfortunately, rather than participating independently, most Bitcoin miners join coalitions called *mining pools* in which a central pool administrator largely directs the pool's activity, leading to a consolidation of power. Recently, the largest mining pool, GHash.IO, has accounted for more than half of network's total mining capacity.<sup>1</sup> Relatedly, "hosted mining" service providers offer their clients the benefit of economies-of-scale, tempting them away from independent participation. We argue that the prevalence of mining coalitions is due to a limitation of the Bitcoin proof-of-work puzzle - specifically, that it supports an effective mechanism for enforcing cooperation in a coalition. We present several definitions and constructions for "nonoutsourceable" puzzles that thwart such enforcement mechanisms, thereby deterring coalitions. We also provide an implementation and benchmark results for our schemes to show they are practical.

## 1. INTRODUCTION

Bitcoin [26] is a successful virtual currency, built around a consensus protocol for a public peer-to-peer network. The security claim in the original Bitcoin whitepaper [26] is that the network maintains an acceptable state whenever a majority of participants (weighted by their contribution of computational effort) follow the protocol correctly. This notion is not compelling on its own, however, since the *a priori* assumption of an "honest majority" is unjustified, and because it yields little insight about the role of *participation incentives* that have been key to Bitcoin's success. An implicit goal of Bitcoin's reward system is to encourage widespread

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

participation from a diverse set of independent entities, leading to diffusion (rather than consolidation) of power. Several recent works have analyzed the incentive structure of the Bitcoin protocol [15, 20]; Eyal and Sirer [15] showed in particular that the Bitcoin protocol is not incentive compatible when a single entity accounts for more than a third of the network overall. An entity that controls a majority of the network would be able to conduct harmful "history revision" or "double-spend" attacks, and therefore poses a threat to the overall stability of the network.

A single entity, in this case, need not be an individual person or computer, but instead may be a coalition of several participants. In fact, over the past several years, the vast majority of Bitcoin participants join coalitions called "mining pools" rather than participating as independents (i.e., "solo-mining"), primarily in order to reduce uncertainty in their payoff [29]. Most mining pools (with the exception of P2Pool) are administered by a trusted "pool operator" who directs how the hashpower of the pool is used. At several times over the past year, the largest mining pools have accounted for well over a third of the network's overall computing effort [7]. A key enabling factor in the growth of mining pools is a simple vet effective *enforcement mechanism*: members of a mining pool do not inherently trust one another, but instead submit cryptographic proofs (called "shares") to the other pool members (or to the pool operator), in order to demonstrate they are contributing work that can only benefit the pool (e.g., work that is tied to the pool administrator's public key).

Another form of power consolidation in Bitcoin occurs when individuals outsource their mining effort to one or a few large service providers. Hosted mining services have already emerged, such as Alydian [6], whose "launch day pricing was \$65,000 per Terahash, and mining hosting contracts are only available in 5 and 10 Th/sec blocks" [6]. Hosted mining is appealing because it can potentially reduce miners' cost due to economies of scale. If there is sufficient market demand for hosted mining, then the cloud may become a concentration of power.

We refer to both mining pools and hosted mining arrangements as "coalitions," and consider these harmful. The original Bitcoin whitepaper [26] draws an analogy between Bitcoin mining and voting in a democratic election (i.e. "onecpu-one-vote"). Viewed in this light, joining a mining coalition may be considered an abdication of responsibility, akin to selling one's vote. Therefore the goal of this paper is to have practical cryptographic mechanisms to deter coalitions. We acknowledge that even though mining pools are poten-

<sup>&</sup>lt;sup>1</sup>See http://arstechnica.com/security/2014/06/ bitcoin-security-guarantee-shattered-by-anonymousminer-with-51-network-power/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

tially hazardous, they also serve a important useful purpose of lowering the payoff variance for individual miners. In Section 8, we argue for a alternative design approach where lower variance is baked into the protocol, without relying on coalitions.

#### **1.1 Our Results and Contributions**

In this paper, we suggest that the existence of effective coalition enforcement mechanisms is a limitation of Bitcoin's underlying proof-of-work puzzle construction. The goal of our work is to construct a modified puzzle that thwarts such enforcement mechanisms, and hence prevents coalitions from forming in the first place.

**Nonoutsourceable puzzles.** We are the first to formally define a novel type of computational puzzle, called *nonoutsourceable puzzles*. In a *weakly nonoutsourceable* puzzle, if one party outsources the task of mining to a worker by *any* effective outsourcing protocol, then the worker can easily steal the puzzle solution and claim the reward for itself. We further provide a strengthened definition of *strongly* nonoutsourceable puzzles, which additionally require that if a worker defects and steals the puzzle solution, it cannot be held accountable.

**Provably secure constructions and evaluation results.** We give constructions for weakly and strongly nonoutsourceable puzzles and prove their security in the random oracle model. We show implementation and evaluation results to demonstrate the practical performance of our puzzles. Specifically, we show that our weakly nonoutsourceable puzzle construction incurs only 2% overhead for the block verification operation, in comparison with today's Bitcoin. Furthermore, for any protocol that can "effectively" outsource work to a worker, the worker can easily steal the client's reward with insignificant additional computation.

We further modify our weakly nonoutsourceable construction to achieve strong nonoutsourceability. Our strongly nonoutsourceable puzzles have two modes of operation. The normal mode of operation (encountered by independent miners) is very cheap, and incurs the same overhead as the underlying weakly nonoutsourceable puzzle. Besides this, the strongly nonoutsourceable construction additionally allows a worker to defect and "steal" a reward by using a zeroknowledge option such that it cannot be held accountable. Based on an instantiation using the succinct zero-knowledge option of Pinocchio [27], we show that it costs a cheating server roughly \$45 of Amazon EC2 computation to successfully steal a reward worth \$10,000 based on Bitcoin's current market price. We believe this provides a sufficiently strong deterrent against mining coalitions.

### 1.2 Related Work

**Computational puzzles.** Moderately hard computational puzzles, often referred to as "proofs of work," were originally proposed for the purpose of combating email spam [14] (though this application is nowadays generally considered impractical [21]). Most work on computational puzzles has focused instead on "client puzzles," which can be used to prevent denial-of-service attacks [19]. Recently, several attempts have been made to provide formal security definitions for client puzzles [10, 17, 31].

Theoretical and economic understanding of Bitcoin. Although a purely digital currency has been long sought after by researchers [8, 9, 11], Bitcoin's key insight is to frame the problem as a consensus protocol and to provide an incentive for users to participate. Although Bitcoin's security has initially been proven (informally) in the "honest majority" model [25, 26], this assumption is unsatisfying in practice, since it says nothing about what design requirements the incentive scheme should satisfy in order to ensure an honest majority. An economic analysis of Bitcoin mining was provided by Kroll et al. [20], who showed that honest participation in Bitcoin may be incentive compatible under assumptions such as a homogeneous population of miners. More recently, Eyal and Sirer considered a broader strategy space [15] and showed that when a single player (or coalition) comprises more than a third of the network's overall strength, the protocol is not incentive compatible (and in fact the threshold is typically much less than one-third, depending on other factors involving network topology). This result underscores the importance of discouraging the formation of Bitcoin mining coalitions.

Altcoins. Numerous attempts have been made to tweak the incentive structure by modifying Bitcoin's underlying puzzle. The most popular alternative, Litecoin<sup>2</sup> uses an scryptbased [28] puzzle intended to promote the use of general purpose equipment (especially CPUs or GPUs) rather than specialized equipment (e.g., Bitcoin mining ASICs). Another oft-cited goal is to make the puzzle-solving computation have an intrinsically useful side effect (this is discussed, for example, in [20]). To our knowledge, we are the first to suggest deterring mining coalitions as a design goal.

Zerocoin [23] and Zerocash [5] focus on making Bitcoin transactions anonymous by introducing a public cryptographic accumulator for mixing coins. Spending a coin involves producing a zero-knowledge proof that a coin has not yet been spent. Although our zero-knowledge proof construction may bear superficial resemblance to this approach, our work addresses a completely different problem.

Nonoutsourceable puzzles. Nonoutsourceable puzzles were first suggested by Miller et al. in the design of Permacoin [24], leaving open the intriguing open question of formally defining nonoutsourceable puzzles and constructing provably secure schemes.

Permacoin investigates how to repurpose Bitcoin's computational puzzle for long-term data archival. As part of this, Permacoin uses a proof-of-retrievability puzzle that deters consolidation of storage capacity. Our Merkle-treebased weakly outsourceable puzzle construction is directly inspired by this construction in Permacoin. Permacoin does not make any attempt to formalize the notion of nonoutsourceable puzzles, nor to consider outsourcing deterrents outside the context of archival storage. Our paper is the first formal treatment of nonoutsourceable puzzles, and the first to propose and construct *strongly* nonoutsourceable puzzles.

# 2. BACKGROUND ON BITCOIN AND USE OF PUZZLES

We define puzzles and nonoutsourceable puzzles as an independent concept, abstracting away the less relevant details about the Bitcoin protocol itself. Later, however, we will

<sup>&</sup>lt;sup>2</sup>https://litecoin.org/

discuss how the puzzles we introduce can be integrated into a Bitcoin-like distributed digital currency. For this reason, as well as to understand the motivation behind our formal definitions, we first present some additional background on Bitcoin and its use of computational puzzles. For a more thorough explanation of the Bitcoin protocol, we refer the readers to [4, 26].

Puzzles, rewards, and epochs. In Bitcoin, new money is printed at a predictable rate, through a distributed coinminting process. At the time of writing, roughly speaking, 25 bitcoins are minted every 10 minutes (referred to as an epoch) on average. When an epoch begins, a public puzzle instance is generated by computing an up-to-date hash of the global transaction log (called the "blockchain"). Then, Bitcoin nodes race to solve this epoch's puzzle. Whoever first finds an eligible solution to the puzzle can claim the newly minted coins corresponding to this epoch. For a miner to cryptographically claim the reward, it searches for a "solution" r such that the hash of the puzzle instance puz, a payload m, and r result in a hash value with d leading 0s. The payload *m* incorporates the miner's public key **p**k, which enables the miner to later spend the coin, along with a set of new transactions to be committed to the log. The difficulty of the puzzle, which depends on d, is adjusted according to the total amount of computational resources devoted to mining to ensure that each epoch lasts 10 minutes on average. In Section 3.1, we explain Bitcoin's puzzle construction in more detail, after formally defining a generalization called scratch-off puzzles.

Block chain and transaction verification. In Bitcoin, the coin minting process is also tied to transaction confirmation. After computing each epoch's puzzle instance, a Bitcoin miner can choose to incorporate unconfirmed transactions into the description of the puzzle. The winning puzzle solution, as well as these transactions, are then cryptographically bound to the puzzle instance. When a winning puzzle solution is found (ending the epoch), a new block is incorporated into the block chain, containing information about all newly confirmed transactions as well as the public key of the miner who claims the minted coin of this epoch. Transactions can declare a percentage of fees (similar to credit card processing fees) to be paid to miners who incorporate the transaction in their puzzle solutions; this encourages miners to incorporate and verify transactions during coin minting, since they can claim the transaction fees in addition to their base reward.

**Consensus mechanism.** Bitcoin nodes reach consensus on the history of transactions by having nodes accept the blockchain with the largest total difficulty. Roughly speaking, this defeats a history revision attack, since to revise history would involve computing a blockchain that is more difficult than the known good chain. An adversary must therefore possess a significant fraction of the total computational resources to successfully race against the rest of the network in extending the chain.

Bitcoin is novel in its use of computational puzzles as part of a consensus protocol for anonymous networks without any pre-established PKI. A related approach was earlier proposed by Aspnes et al. [3], although their network model nonetheless retained a strong assumption about preestablished point-to-point channels.

# 3. SCRATCH-OFF PUZZLES

As introduced earlier, the Bitcoin protocol is built around a moderately hard computational puzzle. Bitcoin miners compete to solve these puzzles, and whoever solves a puzzle first in each epoch receives a reward. As there is no shortcut to solving this puzzle, for an attacker to dominate the network would require the attacker to expend more computational resources than the rest of the honest participants combined. Although the Bitcoin puzzle is commonly referred to as a *proof-of-work* puzzle, the requirements of the puzzle are somewhat different than existing definitions for proof-of-work puzzles [10, 14, 17, 31].

Before proceeding with our main contribution of nonoutsourceable puzzles, we first provide a formal definition of the basic requirements of the Bitcoin puzzle, which we call a *scratch-off puzzle*.<sup>3</sup> In particular, while a traditional proofof-work puzzle need only be solvable by a single sequential computation, a scratch-off puzzle must be solvable by several concurrent non-communicating entities.

In what follows, we let  $\lambda$  denote a security parameter. A scratch-off puzzle is parameterized by parameters  $(\underline{t}, \mu, d, t_0)$  where, roughly speaking,  $\underline{t}$  denotes the amount of work needed to attempt a single puzzle solution,  $\mu$  refers to the maximum amount by which an adversary can speed up the process of finding solutions, d affects the average number of attempts to find a solution, and  $t_0$  denotes the initialization overhead of the algorithm. We typically assume that  $t_0 \ll 2^d \underline{t}$ , where  $2^d \underline{t}$  is the expected time required to solve a puzzle.

DEFINITION 1. A scratch-off puzzle is parameterized by parameters  $(\underline{t}, \mu, d, t_0)$ , and consists of the following algorithms:

- 1.  $\mathcal{G}(1^{\lambda}) \rightarrow \mathsf{puz}$ : generates a puzzle instance.
- Work(puz, m, t) → ticket: The Work algorithm takes a puzzle instance puz, some payload m, and time parameter t. It makes t unit scratch attempts, using t · <u>t</u> + t<sub>0</sub> time steps in total. Here <u>t</u> = poly(λ) is the unit scratch time, and t<sub>0</sub> can be thought of as the initialization and finalization cost of Work.
- 3. Verify(puz, m, ticket)  $\rightarrow \{0,1\}$ : checks if a ticket is valid for a specific instance puz, and payload m. If ticket passes this check, we refer to it as a winning ticket for (puz, m).

Intuitively, the honest Work algorithm makes t unit scratch attempts, and each attempt has probability  $2^{-d}$  of finding a winning ticket, where d is called the puzzle's difficulty parameter. For simplicity, we will henceforth use the notation

$$\zeta(t,d) := 1 - (1 - 2^{-d})^t$$

to refer to the probability of finding a winning ticket using t scratch attempts. For technical reasons that will become apparent later, we additionally define the shorthand  $\zeta^+(t, d) := \zeta(t+1, d)$ . For the remainder of the paper, we assume that the puzzle's difficulty parameter d is fixed, hence we omit the d and write  $\zeta(t)$  and  $\zeta^+(t)$  for simplicity. We also define the algorithm WorkTillSuccess(puz, m) = Work(puz,  $m, \infty$ );

<sup>&</sup>lt;sup>3</sup>The terms "scratch-off puzzle" and "winning ticket" are motivated by the observation that Bitcoin's coin minting process resembles a scratch-off lottery, wherein a participant expends a unit of effort to learn if he holds a winning ticket.

i.e., this algorithm runs until it finds a winning ticket for the given instance and payload.

A scratch-off puzzle must satisfy three requirements:

- 1. Correctness. For any (puz, m, t), if Work(puz, m, t) outputs ticket  $\neq \perp$ , then Verify(puz, m, ticket) = 1.
- 2. Feasibility and parallelizability. Solving a scratchoff puzzle is feasible, and can be parallelized. More formally, for any  $\ell = \text{poly}(\lambda)$ , for any  $t_1, t_2, \ldots, t_\ell = \text{poly}(\lambda)$ , let  $t := \sum_{i \in [\ell]} t_i$ .

$$\Pr\left[\begin{array}{l} \mathsf{puz} \leftarrow \mathcal{G}(1^{\lambda}), \\ m \leftarrow \{0, 1\}^{\lambda}, \\ \forall i \in [\ell] : \mathsf{ticket}_i \leftarrow \mathsf{Work}(\mathsf{puz}, m, t_i) : \\ \exists i \in [\ell] : \mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket}_i) \end{array}\right]$$

$$\geq \zeta(t) - \operatorname{negl}(\lambda).$$

Intuitively, each unit scratch attempt, taking time  $\underline{t}$ , has probability  $2^{-d}$  of finding a winning ticket. Therefore, if  $\ell$  potentially parallel processes each makes  $t_1$ ,  $t_2, \ldots, t_\ell$  attempts, the probability of finding one winning ticket overall is  $\zeta(t) \pm \operatorname{negl}(\lambda)$  where  $t = \sum_{i \in [\ell]} t_i$ .

3.  $\mu$ -Incompressibility. Roughly speaking, the work for solving a puzzle must be incompressible in the sense that even the best adversary can speed up the finding of a puzzle solution by at most a factor of  $\mu$ . More formally, a scratch-off puzzle is  $\mu$ -incompressible (where  $\mu \geq 1$ ) if for any probabilistic polynomial-time adversary  $\mathcal{A}$  taking at most  $t \cdot \underline{t}$  steps,

$$\Pr\left[\begin{array}{l} \mathsf{puz} \leftarrow \mathcal{G}(1^{\lambda}), \\ (m, \mathsf{ticket}) \leftarrow \mathcal{A}(\mathsf{puz}) : \\ \mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket}) = 1 \end{array}\right] \leq \zeta^+(\mu t) \pm \mathsf{negl}(\lambda).$$

Note that  $\zeta^+(t) = 1 - (1 - 2^{-d})^{t+1}$  is roughly the probability of outputting a winning ticket after t unit scratch attempts, though we additionally allow the adversary to make a final guess at the end (as in [31]), and hence the t + 1 in the exponent instead of just t. Ideally, we would like the compressibility factor  $\mu$  to be as close to 1 as possible. When  $\mu = 1$ , the honest Work algorithm is the optimal way to solve a puzzle.

This definition implies, in particular, that solutions to previous puzzles do not help in solving a freshly generated puzzle unseen ahead of time.

#### 3.1 Bitcoin's Scratch-Off Puzzle

An abstraction of Bitcoin's scratch-off puzzle is shown in Figure 1. This scratch-off puzzle abstraction allows us to study it as a stand-alone concept. We elaborate more below on how this scratch-off puzzle would be integrated in the Bitcoin application.

In the actual Bitcoin network, a new puzzle is generated for each epoch (on average, every 10 minutes) by taking a hash of the puzzle solution from the previous epoch. This serves two purposes:

1. There is, in general, a single globally-known **puz** instance each epoch. Further, this **puz** instance can be generated without a trusted third-party – everyone can independently compute the **puz** instance by downloading the up-to-date blockchain. Let  $\mathcal{H}:\{0,1\}^*\to \{0,1\}^\lambda$  be a hash function modeled as a random oracle.

- $\mathcal{G}(1^{\lambda})$ : Draw a puzzle randomly,  $\mathsf{puz} \xleftarrow{\$} \{0,1\}^{\lambda}$ .
- Work(puz, m, t): For i ∈ [t]: //repeat up to t unit scratch attempts Draw a random nonce, r ← {0,1}<sup>λ</sup>. If H(puz||m||r) < 2<sup>λ-d</sup> then return ticket := r. Return ⊥.
  Verify(puz, m, ticket). Check that H(puz||m||ticket) < 2<sup>λ-d</sup>.

# Figure 1: An abstraction of the Bitcoin scratch-off puzzle.

2. It prevents precomputation of puzzle solutions (since puz is unpredictable).

The payload message m includes the following:

- The miner's public key pk. This allows the miner who owns the corresponding secret key sk to later claim the reward (i.e., the newly minted coins) if it succeeds in finding a winning ticket.
- A batch of transactions to be committed. Miners are incentivized to commit transactions since they can claim the transaction fees associated with them.

We assume that each random-oracle call takes time  $t_{\rm RO}$ , and all other work in each iteration of Work takes  $t_{\rm other}$  time. We then have the following:

THEOREM 1. The construction in Figure 1 is a  $(d, \underline{t}, t_0, \mu)$ scratch-off puzzle, where  $\underline{t} = O(\lambda) \cdot t_{RO}$ ,  $t_0 = 0$ , and  $\mu = t_{RO}/(t_{RO} + t_{other})$ .

PROOF. The correctness proof is trivial, as is the proof of feasibility and parallelizability. For  $\mu$ -incompressibility, observe that for any adversary that makes t random oracle calls, its probability of successfully finding a winning ticket is at most  $\zeta^+(t)$ . Since the honest Work algorithm takes  $(t_{\rm RO} + t_{\rm other}) \cdot t$  time, this scratch-off puzzle is  $t_{\rm RO}/(t_{\rm RO} + t_{\rm other})$ -incompressible.  $\Box$ 

#### **3.2** Non-Transferability

For a practical scheme we could integrate into Bitcoin, we should require that the payload of a ticket is non-transferable, in the following sense: if an honest party publishes a ticket attributed to a payload m (e.g., containing a public key belonging to the party to whom the reward must be paid), the adversary should not gain any advantage in obtaining a puzzle solution attributed to some different payload  $m^*$ for the same puz. This is because in Bitcoin, each epoch is defined by a globally known, unique puzzle instance puz; at most one winning ticket for puz and a payload message is accepted into the blockchain; and a user who solves a puzzle only receives the reward if their message is the one that is attributed. If an adversary can easily modify a victim's winning ticket to be attributed to a different payload of its choice, then the adversary can listen for when the victim's ticket is first announced in the network, and then immediately start propagating the modified ticket (e.g., containing its own public key for the reward payment) and attempt to outrace the victim. It is possible that the network will

now deem the adversary as the winner of this epoch—this is especially true if the adversary has better network connectivity than the victim (as described in [15]). For simplicity in developing our constructions and nonoutsourceable definition, we define this non-transferability requirement separately below. Intuitively, non-transferability means that seeing a puzzle solution output by an honest party does not help noticeably in producing a solution attributed to a different payload  $m^*$ .

DEFINITION 2. Let  $\delta$  be a non-negative function of  $\ell$ . A scratch-off puzzle is  $\delta$ -non-transferable if it additionally satisfies the following property:

• for any  $\ell = \text{poly}(\lambda)$ , and for any adversary  $\mathcal{A}$  taking  $t \cdot \underline{t}$  steps,

$$\Pr \begin{bmatrix} \mathsf{puz} \leftarrow \mathcal{G}(1^{\lambda}) \\ m_1, m_2, \dots, m_{\ell} \leftarrow \mathcal{A}(1^{\lambda}) \\ \forall i \in [\ell] : \mathsf{ticket}_i \leftarrow \mathsf{WorkTillSuccess}(\mathsf{puz}, m_i), \\ (\mathsf{puz}, m^*, \mathsf{ticket}^*) \leftarrow \mathcal{A}(\mathsf{puz}, \{m_i, \mathsf{ticket}_i\}_{i=1}^{\ell}) : \\ \mathsf{Verify}(\mathsf{puz}, \mathsf{ticket}^*, m^*) \land (\forall i \in [\ell] : m^* \neq m_i) \end{bmatrix} \\ \leq \zeta^+((\mu + \delta)t) + \mathsf{negl}(\lambda)$$

# 4. OUTSOURCED MINING AND WEAKLY NONOUTSOURCEABLE PUZZLES

The Bitcoin scratch-off puzzle described in the previous section is amenable to secure outsourcing, in the sense that it is possible for one party (the *server*) to perform mining work for the benefit of another (the *client*) and to *prove* to the client that the work done can only benefit the client.

To give a specific example, let m be the public key of the client; if the server performs  $2^{d'}$  scratch attempts, on average it will have found at least one value r such that  $\mathcal{H}(\mathsf{puz}||m||r) < 2^{\lambda-d'}$ . The value r can be presented to the client as a "share" (since it represents a portion of the expected work needed to find a solution); intuitively, any such work associated with m cannot be reused for any other  $m^* \neq m$ . This scheme is an essential component of nearly every Bitcoin mining pool to date [29]; the mining pool operator chooses the payload m, and mining participants are required to present shares associated with m in order to receive participation credit. The rise of large, centralized mining pools is due in large part to the effectiveness of this mechanism.

We now formalize a generalization of this outsourcing protocol, and then proceed to construct puzzles that are *not* amenable to outsourcing (i.e., for which no effective outsourcing protocol exists).

#### 4.1 Notation and Terminology

**Parties.** We use the terminology *client* and *server* referring respectively to the party outsourcing the computation and the party performing the computation. For example, in mining pool operations, individual miners joining the pool are the "servers," and the pool operator is the "client." In cloud hosted mining, the cloud server is the "server," and the individuals who outsource their mining infrastructure are the clients.

**Protocol executions.** A protocol is defined by two algorithms S and C, where S denotes the (honest) server, and C the (honest) client. We use the notation  $(o_S; o_C) \leftarrow (S, C)$ 

to mean that a pair of interactive Turing Machines S and C are executed, with  $o_S$  the output of S, and  $o_C$  the output of C.

In this paper we assume the client is honest but the server may be malicious. We use the notation  $(\mathcal{A}, \mathcal{C})$  to denote an execution between a malicious server  $\mathcal{A}$  and an honest client  $\mathcal{C}$ . Note that protocol definition always uses the honest algorithms, i.e.,  $(\mathcal{S}, \mathcal{C})$  denotes a protocol or an honest execution; whereas  $(\mathcal{A}, \mathcal{C})$  represents an execution.

#### 4.2 **Definitions**

**Outsourcing protocol.** We now define a generalization of outsourced mining protocols, encompassing both mining pools and hosted mining services. Essentially, a server helps the client attempt to compute a winning ticket.

DEFINITION 3. An  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_{e})$ -outsourcing protocol for a scratch-off puzzle ( $\mathcal{G}$ , Work, Verify), where  $t_{e} < t_{S} + t_{C}$  and  $t_{c} < t_{e}$ , is a two-party protocol, ( $\mathcal{S}, \mathcal{C}$ ), such that

- The client's input is puz, and the server's input is  $\perp$ .
- The client C runs in at most t<sub>C</sub> · <u>t</u> time, and the server S in at most t<sub>S</sub> · <u>t</u> time.
- C outputs a tuple (ticket, m) at the end, where ticket is either a winning ticket for payload m or ticket = ⊥. Further, when interacting with an honest S, C outputs a ticket ≠ ⊥ with probability at least ζ(t<sub>e</sub>) - negl(λ).

Formally,

$$\Pr\left[\begin{array}{c} \mathsf{puz} \leftarrow \mathcal{G}(1^{\lambda}) \\ (\cdot;\mathsf{ticket},m) \leftarrow (\mathcal{S},\mathcal{C}(\mathsf{puz})): \\ \mathsf{Verify}(\mathsf{puz},m,\mathsf{ticket}) \end{array}\right] \geq \zeta(t_e) - \mathsf{negl}(\lambda).$$

The parameter  $t_e$  is referred to as the *effective billable* work, because the protocol (S, C) has the success probability of performing  $t_e$  unit scratch attempts. Note that it must be the case It must be the case that  $t_e < \mu(t_S + t_C)$ . Intuitively, an outsourcing protocol allows effective outsourcing of work by the client if  $t_e \gg t_C$ .

Note that this definition does not specify how the payload m is chosen. In typical Bitcoin mining pools, the client (i.e., the pool operator) chooses m so that it contains the client's public key. However, our definition also includes schemes where m is jointly computed during interaction between S and C, for example.

Weak nonoutsourceability. If we want a puzzle to discourage outsourcing, we would like to say that for *any* outsourcing protocol that can "effectively" outsource work to a server (i.e., if the client does less work than the server, and the server does not incur too much overhead relative to honest Work) then the server can easily "steal" the work for its own benefit. More specifically, the server gains an advantage in generating a winning ticket associated with a payload of its own choice, over which the client has no influence. Based on this intuition, we now formally define the notion of a *weakly nonoutsourceable* scratch-off puzzle.

DEFINITION 4. A scratch-off-puzzle is  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_{e}, \alpha, p_{s})$ weakly-nonoutsourceable if for every  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_{e})$ -outsourcing protocol  $(\mathcal{S}, \mathcal{C})$ , there exists an adversary  $\mathcal{A}$  that runs in time at most  $t_{\mathcal{S}} \cdot \underline{t} + \alpha$ , such that: Let m<sup>\*</sup> 
 <sup>\$</sup> {0,1}<sup>λ</sup>. Then, at the end of an execution (A(puz, m<sup>\*</sup>), C(puz)), the probability that A outputs a winning ticket for payload m<sup>\*</sup> is at least p<sub>s</sub>ζ(t<sub>e</sub>). Formally,

$$\Pr\left[\begin{array}{c} \mathsf{puz} \leftarrow \mathcal{G}(1^{\lambda}); m^* \overset{\$}{\leftarrow} \{0, 1\}^{\lambda} \\ (\mathsf{ticket}^*; \mathsf{ticket}, m) \leftarrow (\mathcal{A}(\mathsf{puz}, m^*), \mathcal{C}(\mathsf{puz})) \colon \\ \mathsf{Verify}(\mathsf{puz}, \mathsf{ticket}^*, m^*) \end{array}\right] \geq p_s \zeta(t_e).$$

• Let view<sub>h</sub> denote the client's view in an execution with the honest server (S, C(puz)), and let view<sup>\*</sup> denote the client's view in an execution with the adversary (A(puz, m<sup>\*</sup>), C(puz)). Then,

$$view^* \stackrel{c}{\equiv} view_h$$

When C interacts with A, the view of the client view<sup>\*</sup> is computationally indistinguishable from when interacting with an honest S.

Later, when proving that puzzles are weakly-nonoutsourceable, we typically construct an adversary  $\mathcal{A}$  that runs the honest protocol  $\mathcal{S}$  until it finds a ticket for m, and then transforms the ticket into one for  $m^*$  with probability  $p_s$ . For this reason, we refer to the adversary  $\mathcal{A}$  in the above definition as a stealing adversary for protocol  $(\mathcal{S}, \mathcal{C})$ . In practice, we would like  $\alpha$  to be small, and  $p_s \leq 1$  to be large, i.e.,  $\mathcal{A}$ 's run-time is not much different from that of the honest server, but  $\mathcal{A}$ can steal a ticket with high probability.

If the client outputs a valid ticket for m and the server outputs a valid ticket for  $m^*$ , then there is a race to determine which ticket is accepted by the Bitcoin network and earns a reward. Since the  $\mu$ -incompressibility of the scratch-off puzzle guarantees the probability of generating a winning ticket associated with either m or  $m^*$  is bounded by  $\zeta^+(\mu(t_S + t_C))$ , the probability of the client outputting a ticket – but not the server – is bounded by  $\zeta^+(\mu(t_S + t_C)) - p_s$ .

Note that weak nonoutsourceability does not imply that the puzzle is transferable. In other words, a puzzle can be simultaneously non-transferable and weakly nonoutsourceable. This is so because the stealing adversary  $\mathcal{A}$  may rely on its view of the entire outsourcing protocol when stealing the ticket for its own payload  $m^*$ , whereas the adversary for the non-transferability game is only given winning tickets as input (but no protocol views).

As we mentioned in the beginning of this section, the prevalence of Bitcoin mining pools can be attributed in part to the effective outsourcing protocol used to coordinate untrusted pool members - in other words, the Bitcoin puzzle is not nonoutsourceable. We state and prove a theorem to this effect in the full online version of this paper.

# 5. A WEAKLY NONOUTSOURCEABLE CONSTRUCTION

In this section, we describe a weakly nonoutsourceable construction based on a Merkle-hash tree construction. We prove that our construction satisfies weak nonoutsourceability (for a reasonable choice of parameters) in the random oracle model. In particular, we show that for *any* outsourcing protocol that can effectively outsource a fixed constant fraction of the effective work, an adversarial server will be able to steal the puzzle with at least constant probability. Our construction is inspired by the Floating Preimage Signature (FPS) scheme used in Permacoin [24], which is a puzzle integrated with a proof-of-retrievability. However, Permacoin [24] only described the issue of nonoutsourceability informally, and made no attempt to formalize the definition nor to discuss nonoutsourceability beyond the context of archival storage.

#### 5.1 Construction

Our Merkle tree-based weakly nonoutsourceable puzzle construction is described formally in Figure 2.

**Intuition.** To solve a puzzle, a node first builds a Merkle tree with random values at the leaves; denote the root by digest. Then the node repeatedly samples a random value r, computes  $h = \mathcal{H}(\mathsf{puz}||r||\mathsf{digest})$ , and uses h to select q leaves of the Merkle tree and their corresponding branches (i.e., the corresponding Merkle proofs). It then hashes those branches (along with  $\mathsf{puz}$  and r) and checks to see if the result is less than  $2^{\lambda-d}$ .

Once successful, the node has a value r what was "difficult" to find, but is not yet bound to the payload message m. To effect such binding, a "signing step" is performed in which  $\mathcal{H}(\mathsf{puz}||m||\mathsf{digest})$  is used to select a set of 4q' leaf nodes. Any q' of these leaves, along with their corresponding branches, constitute a signature for m and complete a winning ticket.

Intuitively, this puzzle is weakly nonoutsourceable because in order for the server to perform scratch attempts, it must either

- know a large fraction of the leaves and branches of the Merkle tree, in which case it will be able to sign an arbitrary payload m\* with high probability – by revealing q' out of the 4q' leaves (and their corresponding branches) selected by m\*,
- or incur a large amount of overhead, due to aborting scratch attempts for which it does not know the necessary leaves and branches,
- or interact with the client frequently, in which case the client performs a significant fraction of the total number of random oracle queries.

To formally prove this construction is weakly nonoutsourceable, we assume that the cost of the Work algorithm is dominated by calls made to random oracles. Thus, for simplicity, in the following theorems we equate the running time with the number of calls to the random oracle. However, the theorem can be easily generalized (i.e., relaxing by a constant factor) as long as the cost of the rest of the computation is only a constant fraction of the random-oracle calls.

THEOREM 2. The construction in Figure 2 is a scratchoff puzzle.

PROOF. Correctness, feasibility/parallelizability proofs are trivial. We now prove incompressibility.

For any adversary  $\mathcal{A}$  to obtain a winning ticket with  $\zeta^+(t_e)$ probability, the adversary must have made at least  $t_e$  good scratch attempts. A good scratch attempt consists of at least two random oracle queries,  $h := \mathcal{H}(\mathsf{puz}||r||\mathsf{digest})$  and  $\mathcal{H}(\mathsf{puz}||r||\sigma_h)$  such that the branches  $\sigma_h$  are consistent with h and digest. For each good scratch attempt the adversary must know at least a constant fraction (for any constant < 1) of the branches, and have made random oracle calls to generate the branches such that they are consistent with the digest. Otherwise, when the adversary calls the random oracle for selecting the leaves (notice that the digest is an input Let  $\mathcal{H}, \mathcal{H}_2: \{0,1\}^* \to \{0,1\}^{\lambda}$  denote random oracles.

•  $\mathcal{G}(1^{\lambda})$ : generate puz  $\xleftarrow{\$} \{0,1\}^{\lambda}$ 

```
• Work(puz, m, t):
```

**Construct Merkle tree.** Sample L random strings  $\mathsf{leaf}_1, \ldots, \mathsf{leaf}_L \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$ , and then construct a Merkle tree over the leaf<sub>i</sub>'s using  $\mathcal{H}_2$  as the hash function. Let digest denote the root digest of this tree.

Scratch. Repeat the following scratch attempt t times:

- \* Draw a random nonce,  $r \stackrel{\$}{\leftarrow} \{0,1\}^{\lambda}$ .
- Compute  $h := \mathcal{H}(\mathsf{puz} || r || \mathsf{digest})$ , and use the value h to select q distinct leaves from the tree.
- \* Let  $B_1, B_2, \ldots, B_q$  denote the branches corresponding to the selected leaves. In particular, for a given leaf node, its branch is defined as the Merkle proof for this leaf node, including leaf node itself, and the sibling for every node on the path from the leaf to the root.

\* Compute  $\sigma_h := \{B_i\}_{i \in [q]}$  in sorted order of i. \* If  $\mathcal{H}(\mathsf{puz}\|r\|\sigma_h) < 2^{\lambda-d}$  then record the solution pair  $(r, \sigma_h)$  and goto "Sign payload".

- **Sign payload.** If no solution was found, return  $\perp$ . Otherwise, sign the payload m as follows:
- \* Compute  $h' := \mathcal{H}(puz||m||digest)$ , and use the value h' to select a set of 4q' distinct leaves from the tree such that these leaves are not contained in  $\sigma_h$ . From these, choose an arbitrary subset of q' distinct leaves. Collect the corresponding branches for these q' leaves, denoted  $B_1, B_2, \ldots, B_{q'}$ .
- $\begin{array}{l} * \ \operatorname{Let} \ \sigma'_h := \{B_i\}_{i \in [q']} \ \text{in sorted order of} \ i. \\ * \ \operatorname{Return} \ \operatorname{ticket} := (\operatorname{digest}, r, \sigma_h, \sigma'_h). \end{array}$

• Verify(puz, m, ticket):

Parse ticket :=  $(\text{digest}, r, \sigma_h, \sigma'_h)$ Compute  $h := \mathcal{H}(\mathsf{puz}||r||r||\mathsf{digest})$ . Compute  $h' := \mathcal{H}(\mathsf{puz}||m||\mathsf{digest})$ . Verify that  $\sigma_h$  and  $\sigma'_h$  contain leaves selected by h and h' respectively. Verify that  $\sigma_h$  and  $\sigma'_h$  contain valid Merkle paths with respect to digest. Verify that  $\mathcal{H}(\mathsf{puz}||r||\sigma_h) < 2^{\lambda-d}$ .

#### Figure 2: A weakly nonoutsourceable and non-transferable scratch-off puzzle.

to this call for selecting leaves), except with negligible probability, no polynomial-time adversary can output the selected branches (which are then fed into another random oracle to determine the difficulty). Also, for each good scratch attempt (that contributes to the  $\zeta^+(t_e)$  success probability), the adversary must also make the random oracle call to select the branches, and the call to determine the difficulty. Therefore, the adversary must make an arbitrarily large constant fraction (for any constant < 1) of the random oracle calls for each good scratch attempt. Therefore, the scratchoff puzzle is  $\mu$ -incompressible for an appropriate choice of  $\mu$ , assuming that the rest of the computation is only at most a constant fraction of the random oracle calls.

THEOREM 3. Let  $q, q' = O(\lambda)$ . Let the number of leaves  $L \ge q + 8q'$ . Suppose d > 10 and  $t_e \cdot 2^{-d} < 1/2$ . Under the aforementioned cost model, the above construction is a  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha, p_s)$  weakly nonoutsourceable puzzle, for any  $0 < \infty$  $\gamma < 1$  s.t.  $t_{\mathcal{C}} < \gamma t_e, p_s > \frac{1}{2}(1-\gamma) - \operatorname{negl}(\lambda), and \alpha = O(\lambda^2);$ and is 0-non-transferable.

In other words, if the client's work  $t_{\mathcal{C}}$  is a not a significant fraction of  $t_e$ , i.e., work is effectively outsourced, then an adversarial server will be able to steal the client's ticket with a reasonably big probability, and without too much additional work than the honest server.

The proof that this puzzle is weakly nonoutsourceable can be found in the Appendix, but we sketch the main idea here. Very informally, to "effectively" outsource work to the server, the server must know more than a constant fraction (say, 1/3) of the leaves before calling the random oracle to determine whether an attempt is successful. However, if the server knows more than 1/3 fraction of the leaves, due to a simple Chernoff bound, it will be able to easily steal the solution should one be found. To make this argument formally is more intricate (see Appendix).

For lack of space, we defer (to the full online version of our paper) the proof that this puzzle is non-transferable.

# 6. STRONGLY NONOUTSOURCEABLE PUZZLES

In the previous section, we formally defined and constructed a scheme for weakly nonoutsourceable puzzles, which ensure that for any "effective" outsourcing protocol, there exists an adversarial server that can steal the client's winning ticket with significant probability, should a winning ticket be found. This can help deter outsourcing when individuals are expected to behave selfishly.

One potential drawback of the weakly nonoutsourceable scheme is that a stealing adversary may be detected when he spends his stolen reward, and thus might be held accountable through some external means, such as legal prosecution or a tainted public reputation. Ideally, we should enable the stealing adversary to evade detection and leave no trail of evidence. Therefore, in this section, we define a "strongly nonoutsourceable" puzzle, which has the additional requirement that a stolen ticket cannot be distinguished from a ticket produced through independent effort.

Definition 5. Apuzzleis $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_{e}, \alpha, p_{s})$ strongly-nonoutsourceable if it is  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha, p_s)$ -weaklynonoutsourceable, and additionally the following holds:

For any  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e)$ -outsourcing protocol  $(\mathcal{S}, \mathcal{C})$ , there exists an adversary A for the protocol such that the stolen ticket output by A for payload  $m^*$  is computationally indistinguishable from a honestly computed ticket for  $m^*$ , even given the client's view in the execution  $(\mathcal{A}, \mathcal{C})$ . Formally, let puz  $\leftarrow \mathcal{G}(1^{\lambda}), \ let \ m^* \stackrel{\$}{\leftarrow} \{0,1\}^{\lambda}.$  Consider a protocol execution  $(\mathcal{A}(\mathsf{puz}, m^*), \mathcal{C}(\mathsf{puz}))$ : let view<sup>\*</sup> denote the client  $\mathcal{C}$ 's view and ticket<sup>\*</sup> the stolen ticket output by A in the exeLet NIZK be a non-interactive zero-knowledge proof system. Also assume that  $\mathcal{E}=(\mathsf{Key},\mathsf{Enc},\mathsf{Dec})$  is a CPA-secure public-key encryption scheme.

Let  $(\mathcal{G}', \mathsf{Work}', \mathsf{Verify}')$  be a weakly nonoutsourceable scratch-off puzzle scheme. We now construct a strongly nonoutsourceable puzzle scheme as below.

•  $\mathcal{G}(1^{\lambda})$ : Run the puzzle generation of the underlying scheme  $puz' \leftarrow \mathcal{G}'(1^{\lambda})$ . Let  $crs \leftarrow \mathsf{NIZK.Setup}(1^{\lambda})$ ; and let  $(\mathsf{sk}_{\mathcal{E}},\mathsf{pk}_{\mathcal{E}}) \leftarrow \mathcal{E}.\mathsf{Key}(1^{\lambda})$ . Output

$$puz \leftarrow (puz', crs, pk_{\mathcal{E}})$$

• Work(puz, m, t):

 $\begin{array}{l} \text{Parse } \mathsf{puz} := (\mathsf{puz}', \mathsf{crs}, \mathsf{pk}_{\mathcal{E}}).\\ \mathsf{ticket}' \leftarrow \mathsf{Work}'(\mathsf{puz}', m, t),\\ \text{Encrypt } c \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathcal{E}}; \mathsf{ticket}'; \mathsf{r}).\\ \text{Compute } \pi \leftarrow \mathsf{NIZK}.\mathsf{Prove}(\mathsf{crs}, (c, m, \mathsf{pk}_{\mathcal{E}}, \mathsf{puz}'), (\mathsf{ticket}', r))\\ \mathsf{for the following NP statement:}\\ \mathsf{Verify}'(\mathsf{puz}', m, \mathsf{ticket}') \wedge c = \mathsf{Enc}(\mathsf{pk}_{\mathcal{E}}; \mathsf{ticket}'; \mathsf{r})\\ \text{Return ticket} := (c, \pi).\\ \bullet \mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket}); \end{array}$ 

Parse  $puz := (puz', crs, pk_{\mathcal{E}})$ , and parse ticket as  $(c, \pi)$ . Check that Verify $(crs, (c, m, pk_{\mathcal{E}}, puz'), \pi) = 1$ .

Figure 3: A generic transformation from any *weakly* nonoutsourceable scratch-off puzzle to a strongly non-outsourceable puzzle.

cution. Let ticket<sub>h</sub> denote an honestly generated ticket for  $m^*$ , i.e., ticket<sub>h</sub> := WorkTillSuccess(puz,  $m^*$ ), and let view<sub>h</sub> denote the client's view in the execution (S, C(puz)). Then,

 $(view^*, ticket^*) \stackrel{c}{\equiv} (view_h, ticket_h)$ 

In Figure 3, we present a generic transformation that turns any weakly nonoutsourceable puzzle into a strongly nonoutsourceable puzzle. The strengthened puzzle is essentially a *zero-knowledge* extension of the original – a ticket for the strong puzzle is effectively a proof of the statement "I know a solution to the underlying puzzle."

THEOREM 4. If (GenKey', Work', Verify') is a  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha, p_s)$ weakly nonoutsourceable puzzle, then the puzzle described in Figure 3 is a  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha + t_{enc} + t_{NIZK}, p_s - \operatorname{negl}(\lambda))$  strongly nonoutsourceable puzzle, where  $t_{enc} + t_{NIZK}$  is the maximum time required to compute the encryption and NIZK in the honest Work algorithm.

PROOF. We first prove that this derived puzzle preserves weak nonoutsourceability of the underlying puzzle. Suppose that  $(\mathcal{S}, \mathcal{C})$  is a  $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e)$  outsourcing protocol. We will construct a suitable stealing adversary  $\mathcal{A}$ . To do so, we begin by deriving an outsourcing protocol  $(\mathcal{S}', \mathcal{C}')$  for the underlying puzzle; the stealable property of the underlying puzzle allows to introduce an adversary  $\mathcal{A}'$ , from which we will derive  $\mathcal{A}$ . Let the outsourcing protocol  $(\mathcal{S}', \mathcal{C}')$  for the underlying puzzle be constructed as follows: Suppose that  $\mathcal{C}'$  is given the input puz'.

- 1.  $\mathcal{S}'$  executes  $\mathcal{S}$  unchanged.
- 2. C' first generates a keypair according to the encryption scheme,  $(\mathsf{pk}_{\mathcal{E}}, \mathsf{sk}_{\mathcal{E}}) \leftarrow \mathcal{E}.\mathsf{Key}(1^{\lambda})$ , runs the NIZK setup

$$\label{eq:crs} \begin{split} \mathsf{crs} &\leftarrow \mathsf{NIZK}.\mathsf{Setup}(1^\lambda), \ \mathrm{and} \ \mathrm{then} \ \mathrm{runs} \ \mathcal{C} \ \mathrm{using} \ \mathrm{puzzle} \\ (\mathsf{puz}',\mathsf{crs},\mathsf{pk}_{\mathcal{E}}). \end{split}$$

3. If C outputs a ticket  $(c, \pi)$ , then C' decrypts c and outputs ticket'  $\leftarrow \mathsf{Dec}(\mathsf{sk}_{\mathcal{E}}, \mathsf{c})$ .

When run interacting with S, the original client C outputs a valid ticket with probability at least  $\zeta(t_e) - \operatorname{negl}(\lambda)$ ; therefore, the derived client C' decrypts a valid ticket with probability  $\zeta(t_e) - \operatorname{negl}(\lambda)$ . Since the underlying puzzle scheme is assumed to be weakly nonoutsourceable, there exists a stealing adversary  $\mathcal{A}'$  running in time  $t'_a = t_S + \alpha$ . We can thus construct an  $\mathcal{A}$  that runs  $\mathcal{A}'$  until it outputs ticket'\_{\mathcal{A}} for the underlying puzzle, and then generate an encryption and a zero-knowledge proof.

We also need to prove it satisfies the additional indistinguishability property required from a strongly nonoutsourceable puzzle. This follows in a straightforward manner from the CPA-security of the encryption scheme, and the fact that the proof system is zero-knowledge.  $\Box$ 

We next state a theorem that this generic transformation essentially preserves the non-transferability of the underlying puzzle.

THEOREM 5. If the underlying puzzle ( $\mathcal{G}'$ , Work', Verify') is  $\delta'$ -non-transferable, then the derived puzzle through the generic transformation is  $\delta$  non-transferable for

$$\mu + \delta' \le \frac{(\mu + \delta)t}{t \cdot \underline{t} + (t_{\text{enc}} + t_{\text{nizk}})\ell}$$

where  $t_{enc}$  and  $t_{nizk}$  are the time for performing each encryption and NIZK proof respectively.

Again, due to space restrictions, we defer the proof of this theorem to the full online version of our paper.

# 7. IMPLEMENTATION AND MICROBENCHMARKS

In order to demonstrate the practicality of our schemes, we implemented both our weakly-nonoutsourceable and stronglynonoutsourceable puzzle schemes and provide benchmark results below.

**Metrics.** We are concerned with two main performance criteria. First, the size of a ticket and cost of verifying a ticket should be minimal, since each participant on the network is expected to verify every ticket independently. Second, in order for our scheme to be an effective deterrent, the cost and *latency* required to "steal" a ticket should be low enough that it is at least *plausible* for an outsourcing server to compute a stolen ticket *and* propagate it throughout the network before any other solution is found.

When comparing the verification cost of our schemes to that of the current Bitcoin protocol, we include both the cost due to the puzzle itself, as well as the total cost of validating a block including transactions. At present, there are over 400 transactions per block on average; <sup>4</sup> we assume each transaction carries at least 1 ECDSA signature that must be verified. In general, the computational cost of validating blocks in Bitcoin is largely dominated by verifying the ECDSA signatures in transactions rather than verifying puzzle solutions. We measured that the time to verify an

<sup>&</sup>lt;sup>4</sup>Average number of transactions per block: https:// blockchain.info/charts/n-transactions-per-block

ECDSA signature on a 2.4GHZ Intel CPU is  $1.7\mathrm{ms.}^5~$  On average, a block contains about 0.15 megabytes of data.  $^6$ 

#### 7.1 Our Weakly Nonoutsourceable Puzzle

The weakly-nonoutsourceable puzzle is straightforward to implement, and its overhead relative to the Bitcoin puzzle consists only of  $\lambda \log \lambda$  additional hashes; we implemented this in unoptimized Python and discuss its performance later on. The strongly-nonoutsourceable puzzle, however, requires much more care in implementation due to the NIZK proof. For this purpose we used Pinocchio [27], an implementation of the generic GGPR [16] NIZK scheme that takes high-level C code as input.

We used the SHA-1 hash function throughout our implementation, since this has a relatively efficient implementation as an arithmetic circuit [27]. We restricted our focus to the following puzzle parameters: the signature tree consists of  $2^{h} = 2^{10}$  leaves, and the number of leaves revealed during a scratch attempt and a claim is q = q' = 10. This provides roughly 50 bits of security for the non-transferability property.

**Performance results.** In Table 2, we show that if we replace Bitcoin's puzzle with our weakly nonoutsourceable puzzle, the slowdown for the block verification operation will be only 2%. More specifically, while our puzzle verification itself is over a thousand times more expensive than the Bitcoin puzzle, puzzle verification only accounts for a very small percentage of the overall verifier time. Therefore, the overall performance slowdown is insignificant for practical purposes. Likewise, while the size of the ticket in our scheme is almost a hundred times larger than that of Bitcoin, the ticket is a small fraction of the total size of a block when transactions are included.

An adversarial server can steal a ticket in a marginal amount of time (only one additional hash in expectation, for example, assuming the server knows at least a third of the Merkle tree branches used during scratch attempts). This cost is insignificant compared to the expected time for solving a puzzle.

#### 7.2 Our Strongly Nonoutsourceable Puzzle

We next describe more details of our instantiation and implementation of our strongly nonoutsourceable puzzle, followed by evaluation.

**Diffie-Hellman** + **SHACAL Encryption** Pinocchio efficiently supports group operations in  $\mathbb{Z}_p^*$ , where p is a 254-bit prime corresponding to the BN family of "pairing-friendly" elliptic curves. Unfortunately, the order of this group, p-1, is not prime, nor was p constructed so that p-1 has a large prime factor. Therefore, taking inspiration from "Pinocchio-Coin" [12], we do our public key operations in a prime-order subgroup of the Galois extension field  $\mathbb{F}_{p^{\mu}}$ , where we have chosen  $\mu = 4$  such that  $p^{\mu}$  is over 1000 bits, and  $\mathbb{F}_{p^{\mu}}$  has a subgroup of prime-order  $q \approx 501$  bits.

Our encryption scheme is a variant of DHIES [2], except we replace the symmetric-key encryption component AES with SHACAL [18] (i.e. SHA-1 used as a block cipher) since SHA-1 has a relatively efficient implementation in Pinocchio. Note that the known attacks on SHACAL-1 [13] require an adaptive distinguisher to ask for multiple encryptions under a single key, whereas in our setting an encryption key is used one time only, and therefore is not vulnerable to this attack.

Statement-level parallelism using HMAC commitments. The GGPR scheme underlying Pinocchio is in fact stronger than a NIZK; it is also *succinct* (i.e., a SNARK), meaning that the cost of verifying a proof is independent of the size of the circuit. However, the prover overhead grows superlinearly in the size of the circuit. Additionally, although the GGPR scheme is largely parallelizable, <sup>7</sup> the current Pinocchio implementation does not take advantage of this. Therefore, we take the approach of decomposing the overall NP statement into several smaller sub-statements, for which all the proofs may be computed separately. In particular, we produce two types of statements. The first type of statement (Type I statements) processes a portion (C layers at a time) of one of the q+q' Merkle tree branches, and checks that they are encrypted properly;  $(q+q') \left\lceil \frac{h}{C} \right\rceil$  of these statements must be proven. The Type II statement (of which we require only one instance) performs the public-key operation, checks the winning condition of the puzzle solution, and checks that the q' chosen tree indices are a subset of the 4q' selected based on the attribution message. For every variable shared between substatements, we add to the substatements a SHA-based HMAC commitment. In total, the ticket thus consists of  $(q + q') \left\lceil \frac{h}{C} \right\rceil + 2$  160-bit HMAC tags, (q+q')(h+1) 160-bit SHACAL ciphertexts, and one 1024-bit Diffie-Hellman group element.

**Performance results.** The prover and verifier costs for our strongly-nonoutsourceable implementation are presented in Table 1. Each of the top rows corresponds to a different setting of the parameter C, the number of 160-bit blocks (of the underlying ticket) checked by each substatement. The total number of substatements required (#) is reported along with computing time *per circuit* for the prover and verifier. We also report the total verification time over all the statements, as well as the total proof size. The bottom row is for the second type of statement, which does not depend on C. Due to the longer time required to compute these proofs, the quantities reported are averaged over only three trials.

Keeping in mind our goal is to prove it is *plausible* for a server to produce stolen ticket proof with low latency, we believe it is reasonable to assume that such a server has access to parallel computing resources. Even absent parallelism in the underlying NIZK implementation, our statement-parallelism approach leads to proof times in under 4 minutes at the minimal setting. Since the average time between puzzle solutions in the Bitcoin network is 10 minutes, this may already be a plausible deterrent. On the other hand, with this setting, verification of an entire proof takes under 3 seconds. Since approximately 144 Bitcoin puzzle solutions are produced each day, it would take almost 6 minutes for a single-threaded verifier to validate a day's worth of puzzle solutions. Thus, replacing the underlying SNARK implementation with a fully-parallel one would increase both the plausibility of our scheme as a deterrent, and reduce the verification cost to a constant by requiring only a single circuit.

Even using the parameter choice with the most overhead,

<sup>&</sup>lt;sup>5</sup>Unless otherwise noted, we conducted our measurements over at least 1000 trials, and omit the error statistic if the standard deviation is within  $\pm 1\%$ .

<sup>&</sup>lt;sup>6</sup> Average block size: https://blockchain.info/charts/ avg-block-size

<sup>&</sup>lt;sup>7</sup>For instance, Zaatar [30] reports a near-linear parallel speedup for a similar scheme.

 Table 1: Strongly Nonoutsourceable Benchmarks

Type I Statements								
C	#	Gates	Prove	Verify	Total	Size		
1	220	213k	268.2s	11ms	2.42s	$29.4 \mathrm{KB}$		
2	120	280k	578.4s	11ms	1.32s	23.2 KB		
3	80	392k	1002.9s	11ms	0.80s	$17.0 \mathrm{KB}$		
4	60	467k	1242.1s	11ms	0.66s	$10.8 \mathrm{KB}$		
Type II Statement								
	#	Gates	Prove	Verify	Total	Size		
	1	282K	508.5s	10ms	0.01s	<1KB		

C = 1, and assuming computational power can be rented at \$2.50 per hour (based on Amazon EC2 prices for the most powerful machine), it would cost \$45 in total to produce a stolen ticket proof. This is much less than reward for a puzzle solution, which at the current time is over \$10,000.

# 8. DISCUSSION

Integration in Bitcoin. We have presented our schemes and definitions in an abstract way that hides many of the details of how puzzles are integrated in the Bitcoin protocol. We now clarify two important integration details: First, we have assumed that the puzzle instance is generated independently of all previous puzzles. In practice, each puzzle instance puz should be generated by some deterministic function of the previous puzzle solution; the parameters crs and  $pk_{\varepsilon}$  may be generated once at the outset and reused between puzzles. Second, the attribution message m contains (a commitment to) a payload of transactions to be added to the log. While in Bitcoin a puzzle solution is immutably bound to a single payload, in our nonoutsourceable schemes the payload may be easily modified after the fact (at least by the original solver); in particular, a dishonest puzzle solver who computes a ticket may easily create numerous alternate tickets as well. However, a puzzle solution in our scheme remains immutably bound (via the puzzle instance) to a single previous puzzle solution. Thus an attacker only gains a oneblock advantage in any history revision attack; transactions should be considered "confirmed" after seven blocks rather than six (as is the currently accepted standard in Bitcoin).

**Cheap plaintext option.** It is fairly expensive and timeconsuming to produce the zero-knowledge proof. Although we have shown it is plausible for a stealing server (with parallel resources) to compute such a proof, this would place an undue burden on honest independent miners. However, it is possible to modify our generic transformation so that there are *two* ways to claim a ticket: the first is with a zeroknowledge proof as described, while the second is simply by revealing a plaintext winning ticket for the underlying puzzle. We note that the non-transferable property still applies to tickets claimed with this plaintext option.

Mining pools. Mining pools result in consolidation of influence, which we would like to deter. In a mining pool, individuals are typically anonymous and may join and leave frequently. In this scenario, even our weakly nonoutsourceable puzzle may be a sufficient deterrent, since an individual can claim a winning ticket for himself, and even if the pool operator detects, the cheater cannot easily be punished.

On the other hand, mining pools serve a desirable purpose of reducing the payoff variance for individual participants in the coin minting process. With our nonoutsourceable puzzles, we must seek a separate mechanism to achieve lower variance in mining. This can potentially be achieved by designing and incorporating a more flexible lottery mechanism into Bitcoin. Specifically, the lottery mechanism should allow players to choose different puzzle difficulties, and the higher the difficulty, the higher the promised reward. How to design such a lottery mechanism and understanding its economic implications is part of our future work.

Hosted mining. While mining pools are nearly ubiquitous in Bitcoin, hosted mining has yet to become widespread; the need to discourage is therefore speculative (though plausible, given announcements such as [6]). Furthermore, regardless of cryptographic enforcement, it may be that some users are willing to trust a hosted mining service on the basis of reputation or some enforcement mechanism not captured by our model, such as institutional auditing. Nonetheless, we believe our strong nonoutsourceable scheme provides an effective deterrent by creating the plausibility that a hosted mining server could both cheat and get away with it.

Limitations. One limitation of our proof of Theorem 3 (and shared by other moderately-hard puzzle proofs [10, 17, 31) is the use of random oracles. Since random oracles do not exist in practice, we must instantiate them with cryptographic hash functions. Our proof in particular relies on an "indivisibility" property of a random oracle - every query is either made by the client or made by the server. On the other hand, we know of strong results, for example, in homomorphic encryption [22] which apply to every efficient computation, including the hash functions with which we approximate the random oracle. In practice, the best known generic techniques for fully homomorphic encryption require a very large overhead. Although no lower-bound is known for the cost of techniques like fully homomorphic encryption, it is reasonable to expect that such techniques will at least introduce some relative large cost factor. It is our future work to construct non-outsourceable puzzles in the standard model. Conceivably, any provably secure construction in the standard model must eliminate FHE-based outsourcing schemes by saying that FHE-based outsourcing schemes (or alike) require a large  $t_{\mathcal{S}}$  in comparison with  $t_e$ - this seems to require a more fundamental understanding of the separation (in terms of cost) of computing over encrypted data as opposed to computing in cleartext.

Another limitation is that the zero-knowledge proof system we used in our implementation, GGPR [16], relies on a common reference string (CRS), which must be generated by a trusted party during a setup phase; this setup requirement maybe unacceptable for a Bitcoin-like application . Our constructions, however, would apply equally using any other NIZK system; in particular we rely on neither the succinctness nor the extractability properties of the GGPR scheme. Similarly, if the public key for the encryption scheme can be obliviously sampled using public randomness (i.e., without producing a corresponding private key), then a trusted setup phase may be avoided.

## 9. CONCLUSION

The prevalence of Bitcoin mining coalitions (including both mining pools and hosted mining services), which lead to consolidation of power and increased systemic risk to the network, are a result of a built-in design limitation of the Bitcoin puzzle itself – specifically, that it admits an effective

Table 2:	Estimated	puzzle and	l block	verification	$\mathbf{costs}$	for	various schemes.	

		Puzzle only				Transactions Included			
S	Scheme	Verif.	Ratio	Size	Ratio	Verif.	Ratio	Size	$\mathbf{Ratio}$
I	Bitcoin	$11.7 \mu s$	1	80B	1	0.68s	1	150KB	1
	Weak	$15.1 \mathrm{ms}$	1.3e3	6.6KB	83	0.70s	1.02	157KB	1.05
	C = 4	$0.67 \mathrm{s}$	5.7e4	10.8KB	135	1.35s	1.99	161KB	1.07
	C = 3	0.81s	6.9e4	17.0KB	213	1.49s	2.19	167KB	1.11
	C = 2	1.33s	1.1e5	23.2KB	290	2.01s	2.96	173KB	1.15
	C = 1	2.43s	2.0e5	$29.4 \mathrm{KB}$	368	3.11s	4.57	179 KB	1.19

coalition enforcement mechanism. To address this, we have proposed formal definitions of *nonoutsourceable* puzzles for which no such enforcement mechanism exists. We have contributed two constructions: a weak nonoutsourceable puzzle provable in the random oracle model, and a generic transformation from any weak nonoutsourceable puzzle to a strong one. The former may already be a sufficient deterrent against mining pools, while the latter thwarts both hosted mining and mining pools. We have implemented both of our techniques and provide performance evaluation results showing these add only a tolerable overhead to the cost of Bitcoin blockchain validation.

Acknowledgments We thank readers and reviewers of earlier drafts of this paper who contributed invaluable feedback, including Gregory Maxwell, Ian Miers, and Sergio Lerner. This work was supported in part by NSF awards #0964541 and #1223623.

#### References

- [1] Anonymous online full version. http:// anonymouspaper.com/bitcoin.html.
- [2] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. Dhaes: An encryption scheme based on the diffiehellman problem. Available at citeseer. ist. psu. edu/abdalla99dhaes. html, 1999.
- [3] James Aspnes, Collin Jackson, and Arvind Krishnamurthy. Exposing computationally-challenged byzantine impostors. Department of Computer Science, Yale University, New Haven, CT, Tech. Rep, 2005.
- [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*, pages 399–414. Springer, 2012.
- [5] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP)*, 2014 IEEE Symposium on. IEEE. IEEE, 2014.
- [6] Danny Bradbury. Alydian targets big ticket miners with terahash hosting. http://www.coindesk. com/alydian-targets-big-ticket-miners-withterahash-hosting/, August 2013.
- [7] Vitalik Buterin. Bitcoin network shaken by blockchain fork. http://bitcoinmagazine.com/3668/bitcoinnetwork-shaken-by-blockchain-fork/, 2013.
- [8] David Chaum. Blind signatures for untraceable payments. In Crypto, volume 82, pages 199–203, 1982.

- [9] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In Advances in Cryptology— CRYPTO'88, pages 319–327. Springer, 1990.
- [10] Liqun Chen, Paul Morrissey, Nigel P Smart, and Bogdan Warinschi. Security notions and generic constructions for client puzzles. In Advances in Cryptology– ASIACRYPT 2009, pages 505–523. Springer, 2009.
- [11] Wei Dai. b-money, 1998.
- [12] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies, pages 27–30. ACM, 2013.
- [13] Orr Dunkelman, Nathan Keller, and Jongsung Kim. Related-key rectangle attack on the full shacal-1. In *Selected Areas in Cryptography*, pages 28–44. Springer, 2007.
- [14] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In CRYPTO, 1993.
- [15] I. Eyal and E. Gun Sirer. Majority is not enough: Bitcoin mining is vulnerable. arXiv (arXiv:1311.0243v4), 8 Nov. 2013 (v4).
- [16] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Advances in Cryptology– EUROCRYPT 2013, pages 626–645. Springer, 2013.
- [17] Bogdan Groza and Bogdan Warinschi. Cryptographic puzzles and dos resilience, revisited. *Designs, Codes* and Cryptography, pages 1–31, 2013.
- [18] Helena Handschuh, H Helena, and David Naccache. Shacal (-submission to nessie-). 2000.
- [19] Ari Juels and John G Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In NDSS, volume 99, pages 151–165, 1999.
- [20] Joshua A Kroll, Ian C Davey, and Edward W Felten. The economics of bitcoin mining or, bitcoin in the presence of adversaries. WEIS, 2013.
- [21] Ben Laurie and Richard Clayton. Proof-of-workâÂÎ proves not to work; version 0.2. In Workshop on Economics and Information, Security, 2004.
- [22] Tancrède Lepoint, Jean-Sébastien Coron, Mehdi Tibouchi, et al. Scale-invariant fully-homomorphic encryption over the integers. In 17th International Conference on Practice and Theory in Public-Key Cryptography (PKC'14), 2014.

- [23] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed ecash from bitcoin. In *Security and Privacy (SP)*, 2013 *IEEE Symposium on*, pages 397–411. IEEE, 2013.
- [24] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for long-term data preservation. In *IEEE Symposium* on Security and Privacy, 2014.
- [25] Andrew Miller and Joseph J. LaViola, Jr. Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin. UCF Tech Report. CS-TR-14-01.
- [26] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [27] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.
- [28] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. 2012.
- [29] Meni Rosenfeld. Analysis of bitcoin pooled mining reward systems. arXiv preprint arXiv:1112.4980, 2011.
- [30] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 71– 84. ACM, 2013.
- [31] Douglas Stebila, Lakshmi Kuppusamy, Jothi Rangasamy, Colin Boyd, and Juan Gonzalez Nieto. Stronger difficulty notions for client puzzles and denialof-service-resistant protocols. In *Topics in Cryptology– CT-RSA 2011*, pages 284–301. Springer, 2011.

# APPENDIX

# A. PRELIMINARIES

A Non-Interactive Zero-knowledge Proof system (NIZK) is a collection of three algorithms NIZK = (Setup, Prove, Verify):

- crs ← Setup(1<sup>λ</sup>) : Takes in the security parameter λ, and generates a common reference string crs.
- $\pi \leftarrow \text{Prove}(\text{crs}, \text{stmt}, w)$  : Takes in crs, a statement stmt, and a witness w such that  $(\text{stmt}, w) \in L$ , outputs a proof  $\pi$ .
- b ← Verify(crs, stmt, π): Takes in the crs, a statement stmt, and a proof π, and outputs 0 or 1, denoting rejection or acceptance. stmt, and a witness w,

**Completeness.** A NIZK system is said to be complete, if an honest prover with a valid witness can always convince an honest verifier. More formally, for any  $(\mathsf{stmt}, w) \in R$ , we have

$$\Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^{\lambda}), \\ \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w) : \\ \mathsf{Verify}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1 \end{array}\right] = 1 - \mathsf{negl}(\lambda)$$

**Soundness.** A NIZK system is said to be sound if it is infeasible for any polynomial-time adversary  $\mathcal{A}$  to prove a false statement. More formally,

$$\Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^{\lambda}), \ (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}(\mathsf{crs}):\\ (\mathsf{stmt} \notin L) \ \land \ (\mathsf{Verify}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1) \end{array}\right] = \mathsf{negl}(\lambda)$$

**Zero-knowledge.** Informally, a NIZK system is computationally zero-knowledge, if the proof does not reveal any information about the witness to any polynomial-time adversary. More formally, a NIZK system is said to be computationally zero-knowledge, if there exists a simulator S =(SimSetup, SimProve), such that for all non-uniform polynomialtime adversary  $\mathcal{A}$ , for any stmt, w such that (stmt, w)  $\in \mathbb{R}$ , it holds that

$$\Pr \begin{bmatrix} \operatorname{crs} \leftarrow \operatorname{Setup}(1^{\lambda}), \\ \pi \leftarrow \operatorname{Prove}(\operatorname{crs}, \operatorname{stmt}, w) : \\ \mathcal{A}(\operatorname{crs}, \operatorname{stmt}, \pi) = 1 \end{bmatrix}^{-} \\ \Pr \begin{bmatrix} (\widetilde{\operatorname{crs}}, \tau) \leftarrow \operatorname{SimSetup}(1^{\lambda}, \operatorname{stmt}), \\ \widetilde{\pi} \leftarrow \operatorname{SimProve}(\operatorname{crs}, \operatorname{stmt}, \tau) : \\ \mathcal{A}(\widetilde{\operatorname{crs}}, \operatorname{stmt}, \widetilde{\pi}) = 1 \end{bmatrix} = \operatorname{negl}(\lambda)$$

# B. PROOF OF WEAK NONOUTSOURCE-ABILITY

We now prove Theorem 3. For simplicity of presentation, we prove it for the case when  $\gamma = 1/2$ . It is trivial to extend the proof for general  $0 < \gamma < 1$ . To summarize, we would like to prove the following: for a protocol  $(\mathcal{S}, \mathcal{C})$ , if no adversary  $\mathcal{A}$  (running in time not significantly more than the honest server) is able to steal the winning ticket with more than  $\frac{1}{2}\zeta(t_e)$  probability, then  $t_{\mathcal{C}}$  must be a significant fraction of  $t_e$ , i.e., the client must be doing a significant fraction of the effective work. This would deter outsourcing schemes by making them less effective.

If the ticket output at the end of the protocol execution contains a  $\sigma_h$  such that 1) the selected leaves corresponding to  $\sigma_h$  were not decided by a random oracle call during the execution; or 2)  $\sigma_h$  itself has not been supplied as an input to the random oracle during the execution, then this ticket is valid with probability at most  $2^{-d}$ . For  $(\mathcal{S}, \mathcal{C})$  to be an outsourcing protocol with  $t_e$  effective billable work, the honest protocol must perform a number of "good scratch attempts" corresponding to  $t_e$ . Every good scratch attempt queries the random oracle twice, one time to select the leaves, and another time to hash the collected branches. We now define the notion of a "good scratch attempt".

DEFINITION 6. During the protocol (S, C), if either the client or server makes the two random oracle calls  $h := \mathcal{H}(\mathsf{puz}||r||\mathsf{digest})$  and  $\mathcal{H}(\mathsf{puz}||r||\sigma_h)$  for a set of collected branches  $\sigma_h$  that is consistent with h and digest, this is referred to as a good scratch attempt. Each good scratch attempt requires calling the random oracle twice – referred to as scratch oracle calls.

Without loss of generality, we assume that in the honest protocol, if a good scratch attempt finds a winning ticket, the client will accept the ticket. This makes our proof simpler because all good scratch attempts contribute equally to the client's winning probability. If this is not the case, the proof can be easily extended – basically, we just need to make a weighted version of the same argument. For each good scratch attempt, there are two types of random oracle calls. Type 1 calls select the leaves. Type 2 calls hash the collected branches. Assume in the extreme case that the server makes all the Type 1 calls (which accounts for 1/2of all work associated with good scratch attempts). Now consider Type 2 work which constitutes the other half: for each good scratch attempt, if the server knows < 1/3 fraction of leaves of the corresponding tree before the Type 1 random oracle call for selecting the leaves, then the client must have done at least one unit of work earlier when creating the Merkle tree digest. This is because if the server knows < 1/3 fraction of the leaves before leaves are selected, then the probability that the selected leaves all fall into the fraction known by the server is negligible. Since this is a good scratch attempt, for the selected leaves that the server does not know, the client must then know the leaves and the corresponding Merkle branches. This means that the client earlier called the random oracle on those leaves to construct the Merkle digest.

If we want the client's total work to be within 1/2 of the total effective work, then for at least 1/2 of good scratch attempts: server must know at least 1/3 of leaf nodes before the Type 1 oracle is called to select the leaves.

Suppose that d > 10 is reasonably large and that  $t_e \cdot 2^{-d} < 1/2$ . Basically, in this case, the probability that two or more tickets are found within  $t_e$  good attempts are a constant fraction smaller than the probability of one winning ticket being found. If for at least 1/2 of the good scratch attempts, the server knows at least 1/3 fraction of leaves before leaves are selected, then an adversarial server  $\mathcal{A}$  would be able to steal the ticket with constant probability given that a winning ticket is found. To see this, first observe that the probability that a single winning ticket is found is a constant fraction of  $\zeta^+(t_e)$ . Conditioned on the fact that a single winning ticket is found, the probability that this belongs to an attempt that the server knows > 1/3 leaves before leaves are selected is constant. Therefore, it suffices to observe the following fact.

FACT 1. For a good scratch attempt, if a server knows > 1/3 fraction of leaves before leaves are selected, then conditioned on the fact that this good scratch attempt finds a winning ticket, the server can steal the ticket except with probability proportional to  $\exp(-cq')$  for an appropriate positive constant c.

PROOF. By a simple Chernoff bound. The argument is standard and deferred to the online full version. [1]