Algorithm-Substitution Attacks on Cryptographic Puzzles

Pratyush Ranjan Tiwari
Johns Hopkins University
pratyush@cs.jhu.edu

Matthew Green
Johns Hopkins University
mgreen@cs.jhu.edu

April 18, 2022

Abstract

In this work we study and formalize security notions for algorithm substitution attacks (ASAs) on *cryptographic puzzles*. Puzzles are difficult problems that require an investment of computation, memory or some other related resource. They are heavily used as a building block for the consensus networks used by cryptocurrencies, where they include primitives such as proof-of-work, proof-of-space, and verifiable delay functions (VDFs). Due to economies of scale, these networks increasingly rely on a small number of companies to construct opaque hardware or software (*e.g.,* GPU or FPGA images): this dependency raises concerns about cryptographic subversion. Unlike the algorithms considered by previous ASAs, cryptographic puzzles *do not rely on secret keys* and thus enable a very different set of attacks. We first explore the threat model for these systems and then propose concrete attacks that (1) selectively reduce a victim's solving capability (*e.g.,* hashrate) and (2) exfiltrate puzzle solutions to an attacker. We then propose defenses, several of which can be applied to existing cryptocurrency hardware with minimal changes. Given that these attacks are relevant to all proof of work cryptocurrencies that have a combined market capitalization around a $1 trillion USD (March, 2022), we recommend that all vulnerable mining protocols consider making the suggested adaptations today.

1 Introduction

Security for cryptographic systems depends on the existence of a trusted implementation. Unfortunately, recent experience shows that implementers cannot always be trusted. Examples of cryptographic subversion are increasingly common, ranging from backdoors in cryptocurrency wallets [1, 2] and to software packages [3] and smart contracts [4]. Highly-motivated attackers are frequently able to surreptitiously modify the implementation of critical cryptographic algorithms as a means to subvert the operation of security systems that depend on them [5, 6, 7]. In the research literature such attacks are known as *algorithm substitution attacks* (ASAs) and have been the subject of much formal study [8, 9, 10, 11, 12, 13, 14, 15].

In an algorithm substitution attack, an attacker modifies a cryptographic implementation to replace some cryptographic algorithm or protocol with an adversarial replacement. The subverted device is then adopted by an honest user who is unaware of the substitution. Previous work on ASAs [16, 9, 13, 15] focuses on algorithms that employ secret keys, such as decryption and digital signing algorithms. In this setting, the attacker's primary goal is to exfiltrate cryptographic secrets from the device: indeed, some proposed defenses have been aimed solely at increasing the cost of this exfiltration [17]. While these defenses are applicable to systems that perform decryption or digital signing, relatively little work has focused on a second class of cryptographic algorithms: those that do not employ secret keys.

Cryptographic Puzzles. *Cryptographic puzzles* are difficult problems that require an investment of computation, memory or some other related resource [18]. Puzzles were originally proposed for spam prevention or avoidance of DoS attacks [19] but have more recently been adopted by cryptocurrencies such as Bitcoin, Ethereum, Chia and Filecoin for the purpose of *mining*, i.e., achieving consensus and minting new coins [20, 21]. In each of these systems, puzzle solvers reap substantial economic rewards from solving puzzles. The dark side of this arrangement is that these financial incentives provide strong motivation for the development of sophisticated attacks [22].

In this work we consider the problem of implementing and mitigating ASAs against cryptographic puzzle solving hardware and software. The field of cryptocurrency mining is uniquely vulnerable to these attacks due to the fact that nearly all popular mining hardware (and software) is manufactured by a small number of companies, and that many of these companies also operate their own competing mining operations (see figure 2) [23, 24]. Cryptographic puzzle algorithms are also significantly different from the algorithms considered in previous work on ASAs, and therefore support both different attacks and defenses. Most notably, cryptographic puzzles typically do not make use of secret keys. Hence both the attacker goals and defenses must necessarily be quite different. Rather than exfiltrating secret keys, adversaries are instead motivated to devise attack strategies that improve their own ability to extract profits or consensus control over a cryptocurrency network, for example, by selectively reducing the solving-rate of the victim's own hardware or extracting puzzle solutions.

Attack setting: cryptocurrency mining. While ASAs can be applied against any application of puzzles, we focus primarily on the setting of cryptocurrency mining. These operations typically use large collections of puzzle-solving devices to evaluate solutions, and thus mine new blocks in a cryptocurrency network. For many proof-of-work networks like Bitcoin, the most popular mining device (and possibly the only profitable ones) are specialized mining ASICs (Application-Specific Integrated Circuits) that are typically purchased inside an enclosure. Other networks rely on FPGA or GPU devices that use specialized implementations [25]: while these can technically be reviewed by experts, the cost of such reviews is often prohibitive. In addition to purchase price of the hardware and software, the miner must also contribute substantial resources in the form of electricity (for proof-of-work) or other resources such as RAM or hard disk drives (for memory-hard proof-of-work [26] and proof-of-space constructions [27, 28, 29].)

Mining operations can be organized in various configurations. In the more traditional setting, a cryptocurrency node called the *miner* obtains puzzle instances by running the cryptocurrency network's software on a computer and then sends these instances to the possibly-subverted mining devices for solving. In this setting we make the optimistic assumption that the miner's computer operates as expected, and only the specialized mining devices are subverted. This *blockchain communication model* poses challenges for a subversion attacker, since the attacker may not have untrammeled access to send and receive messages to/from the subverted devices: all inputs and outputs must be in the form of correctly-structured puzzle inputs and solutions that must pass through the cryptocurrency network.

While the blockchain model is popular, it is not the only setting in which mining hardware is used. A second popular configuration allows the mining devices to participate in a *mining pool* with many other miners. Mining pools allow many miners to distribute the risk of mining operations, improving the predictability of mining rewards. In this setting the miner (or the device itself) receives puzzle instances from a central *pool coordinator* who, pessimistically, may collude with the subversion attacker. This latter setting greatly increases the attacker's freedom to communicate with the subverted devices, since the attacker need not filter communications via the cryptocurrency network.

Definitions, attacks and defenses. The primary challenge in an ASA is to subvert a cryptographic implementation in a manner that cannot feasibly be detected by the victim. To this end, ASA definitions

require that the subverted algorithm's behavior must remain cryptographically indistinguishable from that of the real one. At the same time, an attacker with knowledge of the subversion algorithm must be able to execute useful attacks against the device.

In this work we identify two primary forms of attack. In a *load-shedding attack*, a subversion attacker remotely triggers the puzzle solving device to reduce its solution rate, thus giving the attacker (whose own hardware is not subverted) a much larger share of the network's overall solution rate. In practice these attacks may allow the attacker to dominate the consensus network, allowing for 51% attacks [22]. In a *leeching attack* the subversion attacker employs (a portion of) the victim's hardware as their own, using it to surreptitiously exfiltrate solutions to attacker-chosen puzzles. The latter attack can allow the attacker to claim rewards from the cryptocurrency network, or even to mine on other networks that use similar puzzles. For both attacks we explore the general requirements for the attack, and then consider concrete attacks against Bitcoin proof-of-work and other puzzles.

Finally, we turn our attention to mitigations. A key observation of our work is that many of these attacks can be blocked if the subversion attacker is unable to communicate with the subverted hardware: we explore several defenses that achieve this for specific puzzles, via the use of simple software masking processes that can be applied by the (trusted) coordinator. Notably, our mitigation for Bitcoin proof-of-work requires no changes to the protocol, while our mitigation for Pietrzak's repeated-squaring VDFs [30] adds only a nominal overhead to the computation process.

1.1 Our contributions

More concretely, our contributions are as follows:

- We revisit cryptographic puzzle definitions in existing literature and simplify it. This gives us a framework to study attacks on real-world examples of cryptographic puzzles like proof-of-work, verifiable delay functions and proof of space protocols.
- We formalize security notions for algorithm-substitution attacks (ASAs) on cryptographic puzzles. The key difference from existing ASA formalizations is that the attacker goals are fulfilled in this setting even if the attacker slows down the puzzle solving device. Such a notion of attacker success is not present in previous work. Furthermore, we add another layer of complexity for mounting such attacks by considering oracles that allow timing of operations to be reported. Most previous works consider timing-based attack detection to be out of model.
- We then devise realistic threat models under which we propose two different algorithm-substitution attacks on cryptographic puzzles. First is the load-shedding attack which affects puzzle solving abilities of subverted devices on an unpredictable set of subverted inputs set by the attacker. Second is the leeching attack which exfiltrates puzzle solutions to the attacker. Both the attacks satisfy the attacker goals without the subversion being detected.
- We provide estimates for attack success probabilities and analysis keeping in mind the Bitcoin blockchain. Since, this is the highest impact application of cryptographic puzzles in the real world. We demonstrate how such an attacker can benefit from these two algorithm-substitution attacks when deployed against Bitcoin's proof-of-work mining.
- Finally, we provide countermeasures that if utilized properly, make all the proposed attacks ineffective.

2 Technical Overview

We now provide an overview of the technical contributions of this work.

Formalizing puzzles. Our first objective is to revisit and re-formalize the notion of a *cryptographic puzzle*. While previous works have offered such formalization, they have in the past been tightly coupled to specific constructions such as the proof-of-work hash puzzles used by Bitcoin. Our goal is to identify a formalization that can be applied to a broad category of puzzles. Our definitions simplify the definitions from the work of Groza and Warinschi [18]. However in our formalization, we generalize each of these systems to a puzzle defined over a specific *resource*: the minimum amount of the puzzle resource required to find a valid puzzle solution is determined by a difficulty parameter. In Section 4, we demonstrate the utility of this new primitive by showing that many important primitives such as proof-of-work, VDFs, and proof-of-space fit the definitions of a cryptographic puzzle as described in Section 4.1.

Defining ASAs on puzzles. Much previous work [9, 11, 12, 13, 14] considers ASAs on cryptographic primitives: each of these works described a simple attacker goal, namely, to compromise the security of confidential communication between two parties. This has traditionally been performed either by leaking the bits of the secret key via clever techniques (such as rejection sampling over the randomness for a randomized encryption algorithm). The puzzle setting differs from these traditional ASA targets in an important way: cryptographic puzzles are traditionally *unkeyed algorithms*, and do not operate over confidential inputs. Moreover, puzzles incorporate a notion of *resource usage* that is not present in traditional cryptographic algorithms: indeed, a key goal of many attacks is to *increase the amount of resources used by a victim puzzle solver.* This requires changes to the historical security definitions offered in previous work.

Attack strategies. Based on our threat model we propose two attack strategies, the first is the **load-shedding attack** (Sections 6.1,7.1.1) which slows down puzzle evaluation on a subverted device and a **leeching attack** (Section 6.2) which allows a subversion attacker to exfiltrate puzzle solutions from a victim's device. Both the attacks we study are *input-induced* attacks. What this means is that in both of the attacks, the malicious manufacturer first specifies in a small set of bad inputs within the puzzle solving/mining device. Prior to receiving any of these inputs, the device behaves indistinguishably from the real implementation. When the device encounters a puzzle instance from this set, it does not perform as expected.

Our load-shedding attack follows a simple strategy. When the device encounters an element from the subverted set of inputs, it either slows down the puzzle-solving process or silently rejects valid puzzle solutions at a set rate. Assuming that the subverted devices make up a significant fraction of the overall network hashrate, this can reduce the overall hashrate and consequently increase the fraction of total hashrate that is provided by unsubverted devices (see figure 4).

A major challenge in this attack occurs when the attacker interfaces with the subverted device via the blockchain (this is known as the blockchain input model). In this case, the attacker must produce trigger inputs that are also valid puzzle inputs, and satisfy the rules of the consensus protocol. For example, in Bitcoin each puzzle input is the joint hash of a previous block and some recent unmined transactions. Since the attacker has only modest influence over this data, influencing this hash can be extremely challenge. To improve the load-shedding attack strategy in lieu of this challenge, we propose that subverted devices may keep state so that the necessary "trigger" input can be fed to the device over n-consecutive blocks, some of which may be under adversarial control. This requires that we develop a scheme for delivering triggers via this relatively noisy channel.

In the leeching attack, when the device encounters an element from the subverted set of inputs, it attempts to first exfiltrate the solution it finds to the puzzle instance before returning to unsubverted/honest

behavior. An attacker can then leverage this exfiltrated solution to engage in a selfish mining [31] strategy. The challenge in this strategy is the need to exfiltrate solutions, something that is particularly challenging given that puzzle-solving miners typically only produce output when they identify a correct solution. At many reasonable network difficulties, this is a rare event and many (e.g., Bitcoin) mining devices never find a single solution. An exception to this rule occurs when devices are used in *mining pools*: in this setting, devices are asked to output low-difficulty solutions periodically as "proof" that the device is contributing to the pool. These solutions allow for the creation of a subliminal channel that can be used to exfiltrate high-difficulty solutions when they are found.

Countermeasures. Finally, we devise counter-measures (Section 8) against these attacks. First we propose a simple testing regime to detect these attacks. We then proceed to proving some general results on defense against ASAs on cryptographic puzzles. The first makes use of pre-processing algorithms that produce *unpredictable* transformations of the puzzle instance before feeding the instance to the puzzle solver. Since all of our attacks are input-induced, this type of pre-processing works by blocking these triggers. We further observe that several common puzzles deployed in cryptocurrency systems (e.g., Bitcoin) already feature pre-processing stages that can be adapted to provide this security, although in practice such adaptations may not be commonly used. This means that mining operations can defend against these attacks with only minor changes, none of which affect the consensus network or break existing mining hardware. ***We recommend that all miners consider making this adaptation today.*** We provide a realistic mitigation in which we add entropy to the coinbase transaction that is the first transaction in each Bitcoin block (as part of the proof-of-work instance). As a second result, we show that some algebraic puzzle protocols such as VDFs admit a "masking" countermeasure that does not require **any** changes to the protocol, but achieves security against input-induced attacks. The puzzle instance can be masked before being fed to the puzzle solving device and then the output of the puzzle solving device can be unmasked to get a valid solution for the original puzzle instance. This masking approach adds a miniscule overhead to puzzle solving.

Real-world concerns. Finally, we conclude by discussing real-world concerns (Section 9) which are relevant to the attacks we propose and beyond. Our basic results demonstrate that we are considering our attackers in a very strong defensive model. Namely, that the detection adversary is very powerful and can detect most input-induced attacks which affect all but a negligible fraction of the puzzle input domain. We choose this approach to be conservative: clearly this represents a lower-bound on the complexity of the attack strategy. We stress that this means that real-world attackers are likely to be even more capable than the attackers we consider in this paper. However, we also note that our countermeasures are **independent** of the strength of the detecting adversary and hence continue to work regardless of this variable.

3 Preliminaries

Hashrate/Hashpower. In a proof-of-work based consensus network, for example, the network of Bitcoin miners, the *hashrate* of the network is the total number of SHA-256 hashes that can be calculated by all the devices participating in the network. This is an important metric because the difficulty of the Bitcoin proof-of-work is determined as a function of the network's hashrate [32]. It is adjusted every 2 weeks according to changes in the network hashrate. The hashrate of an individual miner then is the total number of hashes the individual miner (device) can compute. This again is an important metric because a proof-of-work based consensus is subject to a number of attacks when a single attacker controls a large fraction of the network's hashrate.

51% attacks and Selfish Mining. A number of attacks have been proposed against proof-of-work blockchains. The most well-known is the 51% attack [22], which refers to a situation where a single party/cooperation has control of a majority of the network hashrate and can therefore "fork" and roll back transactions. A related attack called selfish mining [31] has the attacker mine a parallel competing blockchain fork, without publishing it. This attacker then continues to mine on this secret fork while the honest miners continue to mine on the previous "stale" version of the blockchain. The attacker can now judiciously make blocks from its secret branch public. This renders the computational effort of the honest miners moot, while also earning mining rewards for the attacker disproportionate to its hashrate. An attacker requires $1/3$ of the network's hashrate to successfully mount such an attack.

Bitcoin block hashing algorithm. In the real world, the Bitcoin proof-of-work solution consists of the following fields [20]:

- Version (32-bits): Block version number
- hashPrevBlock (256-bits): Hash of the previous block
- hashMerkleRoot (256-bits): Merkle tree root of all the transactions to include in the proposed block
- time (32-bits): Proposed block timestamp as seconds since $1970-01-01\mathsf{T}00:00$ UTC
- diff (32-bits): Difficulty target
- nonce (32-bits): the nonce for which $H(\mathsf{nonce}, H(\mathsf{hashMerkleRoot}, \mathsf{hashPrevBlock}))$ has the correct number of preceding $0'$s as specified by diff

4 Cryptographic Puzzles

To provide background for the attacks in this work, we require a general definition of cryptographic puzzles that captures the various cryptographic primitives that are used as puzzles. Our definitions simplify and extend a previous definition by Groza and Warinschi [18]: they study cryptographic puzzles with different difficulty requirements for puzzle solving from the perspective of puzzle generation. For a detailed discussion on puzzle difficulty bounds we refer readers to their work. Groza and Warinschi's definitions consider other properties of the puzzle from the perspective of the puzzle evaluator such as *optimality*, which tightly bounds the success probability of puzzle solving and *fairness*, which bounds the probability of an evaluator finding a puzzle solution after some number of computational steps. These puzzle properties are extremely meaningful, but not directly relevant to our work. Our goal is to generalize and simplify the cryptographic puzzle definitions to capture a variety of common puzzles, including client puzzles [33, 19], proof of work, verifiable delay functions (VDFs) and proof of space. In all applications, these puzzles require some amount of resource to solve. The nature of this resource varies between puzzles: total computation cycles in proof-of-work, sequential-time for VDFs, and storage space for proof-of-space puzzles. Our definition is designed to generalize over various types of resources as well.

Definition 4.1 (Cryptographic Puzzles)**.** A cryptographic puzzle is a tuple comprising the following possibly probabilistic algorithms:

- $\mathsf{Setup}(1^\lambda, \Delta) \to \mathsf{pp}$: The puzzle generation algorithm takes as input a security parameter 1^λ and a difficulty parameter Δ. It outputs public parameters pp which fix the domain of the unprocessed

input $\mathcal{X}_{\mathsf{Pre}}$, pre-processed input domain $\mathcal{X}$, range $\mathcal{Y}$ of the puzzle and other information required to compute a puzzle or verify a puzzle solution. All the following algorithms implicitly take pp as an input.

- $\mathsf{Pre}(x', \mathsf{aux}) \rightarrow x$: The puzzle evaluation algorithm takes a puzzle input x' from the unprocessed input domain $\mathcal{X}_{\mathsf{Pre}}$ and an auxiliary input aux. It outputs a processed puzzle input $x \in \mathcal{X}$.
- $\mathsf{Eval}(x, \mathsf{aux}) \rightarrow y/\ \perp$: The puzzle evaluation algorithm takes an input x from the pre-processed input domain and the auxiliary information aux which was used for pre-processing. It outputs a puzzle solution y if there exists a valid solution for the input, auxiliary input pair (x, aux), otherwise outputs $\perp$.
- $\mathsf{Verify}(x, \mathsf{aux}, y) \rightarrow 0/1$: The puzzle verification algorithm takes a pre-processed puzzle input x, the auxiliary information aux which was used for pre-processing and an input from the range y. It outputs either 0 or 1.

Cryptographic puzzles must satisfy the correctness, soundness and resource requirement definition as defined below.

Definition 4.2 (Correctness)**.** A cryptographic puzzle is correct if $\forall \lambda, \Delta, \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta)$, and $\forall x \in \mathcal{X}$ if $y \leftarrow \mathsf{Eval}(x, \mathsf{aux})$ then $\mathsf{Verify}(x, \mathsf{aux}, y) = 1$.

Definition 4.3 (Soundness)**.** We require that an adversary can not get a verifier to accept an incorrect puzzle solution.

$$\Pr\left[\begin{array}{c|c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta) & (x, y, \mathsf{aux}) \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}) \\ y \neq \mathsf{Eval}(x, \mathsf{aux}) & \mathsf{Verify}(x, \mathsf{aux}, y) = 1 \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Furthermore, the difficulty parameter Δ defines resource requirements for finding a correct puzzle solution.

Definition 4.4 (Resource-requirement)**.** We require that no PPT adversary can solve the puzzle and consume less than $\Theta_{avg}(\Delta)$ of the puzzle resource while doing so in the average-case and $\Theta_{best}(\Delta)$ in the best-case.

1. **Average-case.** Let $\mathcal{R}^{avg}_{\mathcal{A}}(\Delta)$ represent the *average-case* resource requirement for adversary $\mathcal{A}$ to solve a puzzle with difficulty parameter Δ. For a large number n of puzzle instances, $\{x_1, x_2, \ldots, x_n\}$, let indicator $\mathbb{I} = 1$ if $\mathsf{Verify}(x_i, \mathsf{aux}_i, y_i) = 1 \forall i \in \{n\}$ and $\mathbb{I} = 0$ otherwise.

$$\Pr\left[\begin{array}{c|c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta), x_i \xleftarrow{\$} \mathcal{X} & (y_i, \mathsf{aux}_i) \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}, x_i \\ \mathcal{R}^{avg}_{\mathcal{A}}(\Delta) < \Theta_{avg}(\Delta) & \mathbb{I} = 1) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

2. **Best-case.** Let $\mathcal{R}^{best}_{\mathcal{A}}(\Delta)$ represent the *best-case* resource requirement for adversary $\mathcal{A}$ to solve a puzzle with difficulty parameter Δ. It holds true $\forall x \in \mathcal{X}$:

$$\Pr\left[\begin{array}{c|c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta), x \xleftarrow{\$} \mathcal{X} & (y, \mathsf{aux}) \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}, x) \\ \mathsf{Verify}(x, \mathsf{aux}, y) = 1 & \mathcal{R}^{best}_{\mathcal{A}}(\Delta) < \Theta_{best}(\Delta) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Note that the pre-processing phase can be optional, in which case unprocessed and pre-processed input are the same and consequently $\mathcal{X}_{\mathsf{Pre}} = \mathcal{X}$ and the auxiliary information $\mathsf{aux} = \mathsf{nil}$. A similar worst-case resource requirement can be defined similarly.

4.1 Cryptographic Puzzles in the Real World

To make the above discussion more concrete, we now consider several real-world cryptographic primitives that instantiate the cryptographic puzzle formalism.

Nakamoto Proof of Work The Nakamoto proof of work (PoW) is a costly, time-consuming computation that miners perform in order to get the opportunity to produce the next block, and consequently obtain a reward. At the time of writing, Bitcoin provides 6.25 BTC per block ($\approx$ \$ 250k.) While several currencies use proof-of-work, we focus on the Bitcoin proof-of-work (Section 3) as a specific example here. The Bitcoin proof of work can be formalized as a cryptographic puzzle as demonstrated in Appendix B.1.

Verifiable Delay Functions A verifiable delay function (VDF) [34, 30, 35] is a function where computing the output on any instance requires running a set number of sequential steps, yet the output of the function can be efficiently verified. VDF definitions and security requirements are available in Appendix B.1. Multiple applications of VDFs [21] have sparked a coalition called "VDF Alliance" funded by the Ethereum Foundation, Protocol Labs and others. VDFs can also be formalized as a cryptographic puzzle as demonstrated in Appendix B.5.

Proof of Space Proof of Space (abbreviated as PoSpace) [27, 28, 29] is a primitive which was developed to provide an alternative to Bitcoin's proof of work. It utilizes disk space as the puzzle resource instead of computation as used in Bitcoin's proof of work. PoSpace definitions and security requirements are available in Appendix B.1. PoSpace can also be formalized as a cryptographic puzzle as demonstrated in Appendix B.7.

All of the examples above can be viewed from the cryptographic-puzzle-lens as demonstrated in Appendix B.1. Bitcoin's proof of work has a dynamic average-case resource requirement for puzzle evaluation. The best case resource requirement is trivial, there could be a bitcoin proof of work instance for which just computing one hash gives a valid solution. For VDFs and PoSpace on the other hand, the aim is to have the average-case and best-case requirement be the same. This gives a more uniform domain of puzzle instances with respect to resource requirement for evaluating solutions.

5 Modeling ASAs against Cryptographic Puzzles

5.1 Threat Model

Before we can describe the attacks, it is important to set a realistic threat model. There are two main threat models which are of interest to us. In both of the models, the puzzle evaluation hardware is subverted.

Direct input model In this setting, the hardware is subverted by a malicious manufacturer. This subverted hardware is then sold to a miner. The malicious manufacturer has the ability to feed puzzle inputs to the miner's subverted mining device. The setting is motivated by mining pools where the miner contributes its computation to a pool with a malicious pool operator. The malicious pool operator is aware of the exact way in which the manufacturer subverted the puzzle evaluation hardware. The miner sends its work shares/puzzle solutions to the pool operator rather than communicating directly with the blockchain as represented in Figure 1. [1]

[1] In proof-of-work pooled mining, the miner just receives a hash output (supposed to be $H(\mathsf{hashMerkleRoot}, \mathsf{hashPrevBlock})$) and a nonce range and iterates over the nonce range to compute a double hash $H(\mathsf{nonce}, H(\mathsf{hashMerkleRoot}, \mathsf{hashPrevBlock}))$.

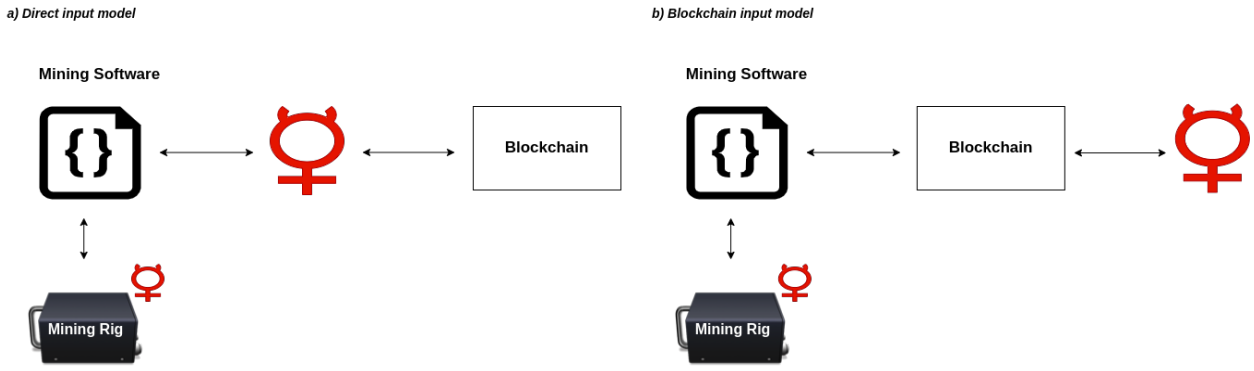

Figure 1: a) **Direct input model.** The miner/puzzle solver owns the mining/evaluation hardware which in turn communicates through the mining/evaluation software. The mining software is provided the next puzzle inputs from the pool operator to whom it submits its work. The pool operator is malicious and it reads from and writes to the blockchain. b) **Blockchain input model.** The miner/puzzle solver owns the mining/evaluation hardware which in turn communicates through the mining/evaluation software which reads from and writes to the blockchain. The miner can add bits of entropy to the mining process.

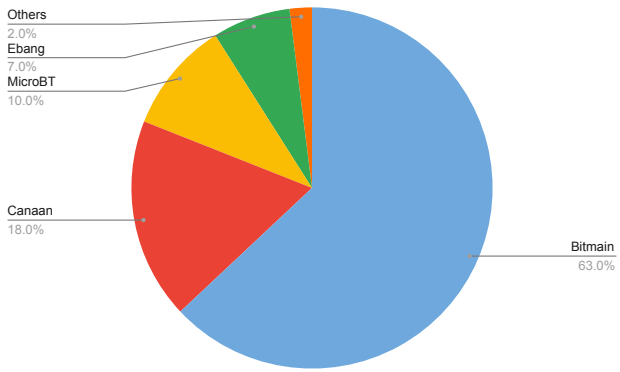

Figure 2: Market Share for Bitcoin mining ASIC manufacturers [36]

Blockchain input model This setting builds on the previous one. The malicious manufacturer and the miner both have access to the blockchain as represented in Figure 1. The goal of the malicious party is to use the blockchain to feed biased inputs to the miner's device to leverage the subversion.

Both of the above models draw inspiration from the current state of the Bitcoin blockchain. Bitmain [24] is one of the largest application-specific integrated circuit (ASIC) chip manufacturer for bitcoin mining **and** it also operates BTC.com and Antpool which have historically been the two largest mining pools for bitcoin. Reports have claimed that approximately two-thirds of the mining hardware market is controlled by Bitmain, as shown in Figure 2. While there is no evidence that Bitmain would subvert hardware, this situation provides a setting where the attacks we study can occur and go undetected. Moreover, this model applies to all proof-of-work based blockchains and some other important ones as we discuss in Section 4.1.

While we motivated the first threat model using Bitcoin's proof-of-work, at the time of writing, $\approx$ \$1

However, since a hash function has no structure, the malicious pool operator can send any input which looks like a hash. It can then use this *direct input to trigger subverted behavior on the next n puzzle instances etc.*

trillion [37] of value is locked in cryptocurrencies which utilize some form of proof-of-work. All proof-of-work based cryptocurrencies (and some others)[2] utilize the concept of pooled mining.

5.2 Security against ASAs

We now define what it means for a cryptographic puzzle scheme $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ to be under an algorithm-substitution attack (ASA). The goal of the attacker $\mathcal{A}$ is to mount an algorithm-substitution attack (ASA) such that the puzzle solving algorithm is compromised in a way beneficial to the adversary and the probability of the compromise being detected is low. Most of the work on ASAs has been in a setting where there is some secret information which $\mathcal{A}$ wants to extract by mounting the attack. In this setting, there is no secret information $\mathcal{A}$ needs to extract. Cryptographic puzzles are used in cryptocurrencies for various reasons. However, there is a reward associated with the puzzle in most applications and hence the goal of any party is to solve the puzzle as quickly as possible. Compromise in this setting then refers to the adversarial goal of $\mathcal{A}$ to slow down the puzzle solving process (or biasing it for favorable outcomes) and the goal of $\mathcal{D}$ is to detect if its running a subverted version of the algorithm Eval to solve the puzzle. Following the footsteps of other works on ASAs we define the goals of the subverting adversary/attacker $\mathcal{A}$ via the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ and the detecting adversary/detector $\mathcal{D}$ via the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$. We first describe the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ (figure 3) played between a challenger and a detecting adversary $\mathcal{D}$. Our oracles also output a counter cnt indicating the time taken to compute the response, unlike previous work on ASAs, this adds another challenge for the attacker. *Note that a subversion can be stateful, which is indicated in the functionality of the oracle which runs the subverted algorithm.*

Definition 5.1 (Detectability)**.** An algorithm-substitution attack against a cryptographic puzzle scheme $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$, where the subverted puzzle algorithms are represented by $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$, is considered detectable if there exists a detection adversary with non-negligible advantage in the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$, i.e., $\forall(\lambda, \Delta), \exists\mathcal{D}$ such that:

$$\mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^\lambda, \Delta) > \mathsf{negl}(\lambda)$$

for all negligible functions $\mathsf{negl}(\cdot)$.

We also define the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ (figure 3) played between a challenger and a subverting adversary $\mathcal{A}$.

Definition 5.2 (Subversion resistance)**.** A cryptographic puzzle scheme $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ is subversion resistant against an algorithm-substitution attack if for the subverted puzzle algorithms $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ all subverting adversaries/attacker $\mathcal{A}$ have a negligible advantage in the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$, i.e., $\forall(\lambda, \Delta, \mathcal{A})$:

$$\mathbf{Adv}^{\mathbf{Sub}}_{\mathcal{A}}(1^\lambda, \Delta) \leq \mathsf{negl}(\lambda)$$

Equipped with these definitions, we are now ready to define security against ASAs.

Definition 5.3 (Security against ASAs)**.** A cryptographic puzzle scheme $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ is considered secure against algorithm-substitution attacks if for all possible subversions $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ of P either the attack is detectable in the sense of definition 5.1 or P is subversion resistant against the attack in the sense of definition 5.2.

[2]For Chia (Proof of Space), pooled mining is very common. However, Chia encourages its miners to only work with mining pools which utilize its open-source Chia Pool operator software. In the past, there have been pools which had private code and a separate Chia client[38], which was speculated to have malicious intent.

$\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$:

- **Setup.** In the setup phase, the challenger samples a bit $b \xleftarrow{\$} \{0,1\}$. If $b = 0$, the challenger runs the unsubverted puzzle algorithms $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ and sends the output of $\mathsf{Setup}(1^\lambda, \Delta)$ to the adversary $\mathcal{D}$. Otherwise, it runs the subverted puzzle algorithms $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ and sends the output of $\overline{\mathsf{Setup}}(1^\lambda, \Delta)$ to $\mathcal{D}$.
- **Query.** In the query phase, the adversary gets access to one of the puzzle evaluation oracles $\mathsf{O}_{\mathsf{Eval}}$ or $\mathsf{O}_{\overline{\mathsf{Eval}}}$ depending on the challenger's sampled bit b being 0 or 1. These oracles are defined as follows:
  - $\mathsf{O}_{\mathsf{Eval}}(\mathsf{pp}, \cdot, \cdot)$ takes as input an element $x \in \mathcal{X}$ and the corresponding preprocessing auxiliary input aux. It computes and outputs $y \leftarrow \mathsf{Eval}(x, \mathsf{aux})$ and a counter cnt indicating the time spent in computing the output.
  - $\mathsf{O}_{\overline{\mathsf{Eval}}}(\mathsf{pp}, \mathsf{state}, \cdot, \cdot)$ takes as input an element $x \in \mathcal{X}$, the corresponding preprocessing auxiliary input aux and also maintains an internal state state. It computes and outputs $\bar{y} \leftarrow \overline{\mathsf{Eval}}(x, \mathsf{aux})$ and a counter cnt indicating the time spent in computing the output. The state update function, sets new state as $\mathsf{state}' \leftarrow \mathsf{st}_{\mathsf{upd}}(\mathsf{state}, x)$.
- **Guess.** In this phase, the adversary outputs its guess b' for b and wins the detection game if $b' = b$. We say that the game outputs 1 if the adversary wins and 0 otherwise.

  The advantage of an adversary $\mathcal{D}$ is defined as $\mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^\lambda, \Delta) = \Pr[b' = b] - 1/2$.

$\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$:

- **Setup.** In the setup phase, the challenger samples a bit $b \xleftarrow{\$} \{0,1\}$. If $b = 0$, the challenger runs the unsubverted puzzle algorithms $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ and sets $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta)$ to the adversary $\mathcal{A}$. Otherwise, it runs the subverted puzzle algorithms $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ and sets $\mathsf{pp} \leftarrow \overline{\mathsf{Setup}}(1^\lambda, \Delta)$. It sends $(\mathsf{pp}, \overline{\mathsf{P}})$ to the adversary $\mathcal{A}$.
- **Query.** In the query phase, the adversary $\mathcal{A}$ gets access to one of the puzzle evaluation oracles $\mathsf{O}_{\mathsf{Eval}}$ or $\mathsf{O}_{\overline{\mathsf{Eval}}}$ depending on the challenger's sampled bit b being 0 or 1. These oracles are exactly the same in the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$.
- **Guess.** In this phase, the adversary outputs its guess b' for b and wins the detection game if $b' = b$. We say that the game outputs 1 if the adversary wins and 0 otherwise.

  The advantage of an adversary $\mathcal{D}$ is defined as $\mathbf{Adv}^{\mathbf{Sub}}_{\mathcal{A}}(1^\lambda, \Delta)$ $= \Pr[b' = b] - 1/2$.

Figure 3: The detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ and the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$

5.2.1 The role of state

There might be attacks which leverage the use of state in the subverted puzzle algorithms. Since our model (Section 5.1 assumes a malicious manufacturer for the puzzle solving hardware/software, the only algorithm where state can be utilized for attacks is the subverted evaluation algorithm $\overline{\mathsf{Eval}}$. Therefore, we term an attack ***stateful*** if it requires $\overline{\mathsf{Eval}}$ to maintain state and ***stateless*** otherwise. If the the attack is stateless then the state update function $\mathsf{st}_{\mathsf{upd}}$ for the oracle running the subverted algorithm outputs the same state it takes as input.

The adversarial model of a PPT detector is inspired from previous work [9, 11, 12, 13, 14] on ASAs. Considering our focus on cryptographic puzzles and that most such puzzles are utilized in a blockchain setting, we further discuss the consequences of the detector and attacker strengths in Section 4.1.

6 Attacks in the Direct input Model

We now look at concrete adversarial strategies to mount algorithm substitution attacks on cryptographic puzzles in the direct input model as described in Section 5.1. All our attacks are input-induced attacks in the black-box setting. Such a setting is reflected in scenarios where a puzzle solver buys some puzzle-solving hardware from an untrusted party. A simple example of an input-induced attack is to have a set of hard-coded inputs in the puzzle evaluation hardware/software on which the evaluation fails/does not proceed as expected. However, in this case, the set of such "bad" inputs will need to be negligibly small compared to the puzzle input domain in order to avoid detection.

6.1 Load-Shedding Attack

The goal of a load-shedding attack is to slow down the puzzle solving algorithm. If this happens for a decent number of devices, the networks puzzle-solving ability reduces and the un-subverted devices have a higher percentage of the network's capacity. This creates a opportunity for the adversary to benefit massively as its percentage network capacity increases. Furthermore, load-shedding can be combined with the selfish mining attack strategy and suddenly the attacker can mount a selfish mining attack by owning only a small fraction of the hashrate (see Figure 4) and subverting some fraction of mining devices. If this happens every once in a while, and is undetected, the adversary keeps on benefiting continually. Clearly, such a simple strategy will be easily detected if the fraction of "bad" inputs on which the puzzle-solving algorithm malfunctions is non-negligible. To improve such an attack, an interesting adversarial strategy would be to keep this set of "bad" inputs small but increase the likelihood with which these inputs are selected. In the direct input model, the adversary can pick the bad inputs while mounting an attack.

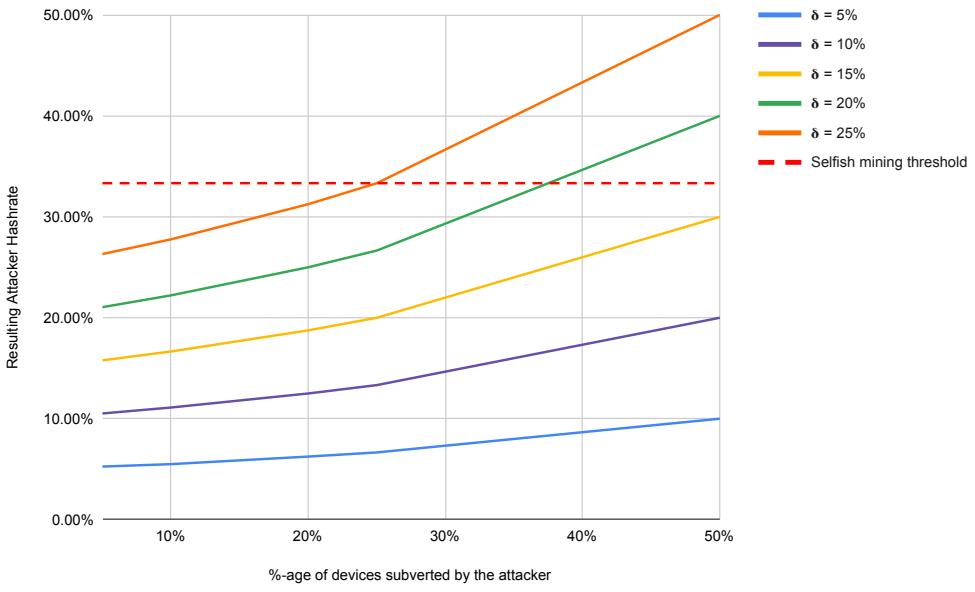

Figure 4: Percentage (effective) increase in attacker hashrate plot as a function of the attacker's original hashrate (δ) and the percentage of subverted devices. This assumes that the subverted devices, when fed a trigger, are unable to find a puzzle solution when under a load-shedding attack *in the direct input model.*

The untrusted party can program in a set of inputs $\overline{\mathcal{X}}$, where $\overline{\mathcal{X}} \subset \mathcal{X}$ such that if $x \in \overline{\mathcal{X}}$ then the evaluation algorithm misbehaves to prevent or delay the evaluation of the puzzle solution. Therefore, in this setting for a cryptographic puzzle $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ the subverted puzzle algorithms are of the form $\overline{\mathsf{P}} = (\mathsf{Setup}, \mathsf{Pre}, \overline{\mathsf{Eval}}, \mathsf{Verify})$. Due to the black-box nature of the attack setting, any detector can only detect the subversion through oracle access.

$\overline{\mathsf{P}}.\overline{\mathsf{Eval}}(\mathsf{pp}, x, \overline{\mathcal{X}})$: *The subverted puzzle evaluation algorithm takes as input the public parameters* pp*, an input from the domain x, and a set $\overline{\mathcal{X}} \subset \mathcal{X}$. If $x \in \mathcal{X} \setminus \overline{\mathcal{X}}$ it outputs a puzzle solution y. Otherwise, the evaluation algorithm fails to produce an output.*

The values $\overline{\mathcal{X}}$ are hard-coded into the evaluation hardware. We first prove some **general results** on load-shedding attacks where the size of the adversarially hard-coded bad inputs is negligible in the security parameter. This models the setting where the detector $\mathcal{D}$ is testing its hardware to ensure proper performance. Since, the set of bad inputs is negligibly small, the probability of detection is negligibly low. However, when the attacker plays the subversion game, it can easily detect whether the challenger is operating a subverted evaluation algorithm. This is because the attacker knows the set of hard-coded bad inputs. [3]

Theorem 6.1. *For all load-shedding attacks where $\frac{|\overline{\mathcal{X}}|}{|\mathcal{X}|} < \mathsf{negl}(\lambda)$, there exists no PPT detector $\mathcal{D}$ which wins the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ with non-negligible advantage.*

Proof. The set of subverted inputs $\overline{\mathcal{X}}$ is a random subset of the set of inputs $\mathcal{X}$. Given that $\frac{|\overline{\mathcal{X}}|}{|\mathcal{X}|} < \mathsf{negl}(\lambda)$, for a PPT detector $\mathcal{D}$, the advantage of winning the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ is same as the probability of querying the evaluation oracle on at least one input $x \in \overline{\mathcal{X}}$. Let event E be the query instance when $\mathcal{D}$ queries the evaluation oracle on an input $x \in \overline{\mathcal{X}}$. Let $\Pr[\overline{E}] = 1 - \Pr[E]$ be the probability of event $\overline{E}$ (complement of event E), i.e. , for query x of $\mathcal{D}$, $x \notin \overline{\mathcal{X}}$. Let q be the number of queries that $\mathcal{D}$ asks. Then,

$$\mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^\lambda, \Delta) = 1 - \left(\Pr[\overline{E}]\right)^q$$

$$= 1 - \left(\Pr[x \in \mathcal{X} \setminus \overline{\mathcal{X}}]\right)^q = 1 - (1 - \mathsf{negl}(\lambda))^q$$

$$\approx 1 - (1 - q.\mathsf{negl}(\lambda)) = q.\mathsf{negl}(\lambda)$$

Since, q is the number of queries $\mathcal{D}$ asks, $q = \mathsf{poly}(\lambda)$. Therefore,

$$\mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^\lambda, \Delta) = q.\mathsf{negl}(\lambda) = \mathsf{negl}'(\lambda)$$

Theorem 6.2. *For all load-shedding attacks, there exists a PPT attacker $\mathcal{A}$ which wins the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ with non-negligible advantage.*

Proof. The attacker $\mathcal{A}$ leverages its knowledge of the set of subverted inputs $\overline{\mathcal{X}}$. Upon receiving the public parameters pp and the subverted algorithms $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$, in the query phase, the attacker first queries on inputs $x \in \mathcal{X} \setminus \overline{\mathcal{X}}$. After this, the attacker picks $x' \in \overline{\mathcal{X}}$ as a query. Using the expected response time from querying on the unsubverted set of inputs, the attacker decides whether its querying the subverted oracle or not based on the (timed) query response received for the subverted input x'. This is because the unsubverted oracle would respond with a correct solution in the expected time.

[3] We assume for theorems 6.1-6.3 that the puzzle evaluator does not utilize the puzzle pre-processing algorithm and feeds puzzle inputs without pre-processing to the evaluation hardware/software.

Theorem 6.3. *All cryptographic puzzle schemes* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Verify})$ *are susceptible to subversion via any load-shedding attacks where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$.

Proof. This follows from Theorem 6.1 and Theorem 6.2 under definition 5.3. □

6.1.1 More realistic adversarial constraints.

The above theorems are focused on load-shedding attacks which load-shed with negligible probability. These might seem like ineffective attacks but they are important to study because they are of relevance to both our threat models. In the direct input model, the adversary decides and picks the puzzle inputs. And in the blockchain input model, the puzzle inputs can be adversarially biased and even adversarially decided. Therefore, it is important to consider the potential of the adversary biasing the blockchain's state in order to increase the likelihood of the hard-coded attack instances being fed to the puzzle solving device. Furthermore, a stateful variant of the attack needs to be considered as well. If a stateful attack induces an attack trigger over the course of multiple blocks, even an adversary with limited capabilities can attempt to bias the blockchain's state by a few bits every block. The goal of the adversary then becomes to bias the blockchain's entropy which has been studied in some detail. We discuss this further in Section 7.

6.2 Leeching Attack

The goal of the adversary with the leeching attack is to compromise the puzzle evaluation hardware in such that it exfiltrates puzzle solutions to the attacker. While we use Bitcoin mining as a high-impact example, this attack is relevant to almost all proof of work based cryptocurrencies. This is also an input induced attack in the black-box setting. The untrusted party can program in a set of inputs $\overline{\mathcal{X}}$, where $\overline{\mathcal{X}} \subset \mathcal{X}$ such that if $x \in \overline{\mathcal{X}}$ then the evaluation algorithm tries to exfiltrate the first solution it finds to the puzzle without outputting it, and continues to find the second solution which it then outputs. Therefore, in this setting for a cryptographic puzzle $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ the subverted puzzle algorithms are also of the form $\overline{\mathsf{P}} = (\mathsf{Setup}, \mathsf{Pre}, \overline{\mathsf{Eval}}, \mathsf{Verify})$. We describe the exfiltration process and the attack in detail below. Such exfiltration is not unrealistic because the puzzle evaluation hardware is an online device, ex. bitcoin mining rigs [39, Page 5]. Hence, a subverted device can use the connection channel to exfiltrate its puzzle solution. Clearly, in the example of selfish mining such an attack is very beneficial. *Now, the attacker does not need to own a third of the hashrate but instead just needs to subvert enough devices to reach the attack threshold.*

Such an exfiltration channel is motivated by real-world scenarios. For example, in Bitcoin mining many miners pool their efforts with different mining pools. Each mining pool has its own software, and the pools have a *pool operator* who uses an algorithm to split work between into *shares* that are farmed out to workers. Let the block reward minus the fee charged by the pool operator be B. Each worker then is paid reward $R = B \cdot \frac{n}{N}$, where n is fraction of work their shares represent and N is the total amount of work represented by all shares. To prove that workers are contributing, they must output solutions found at much lower difficulty levels (this assumes a puzzle that can find low-difficulty solutions while also searching for high-difficulty solutions, a property that is not common to all puzzles.) These low-difficulty solutions are used as a subliminal channel for exfiltration. The exact exfiltration process and the exfiltration game are presented in figure 5.

Definition 6.1 (Security of Exfiltration Channels)**.** An exfiltration channel E is secure if no detector $\mathcal{D}$ can win the exfiltration game $\mathbf{Exf}_{\mathcal{D}}$ with non-negligible advantage.

Assume that $\mathcal{T}_{x,\mathsf{aux}}(x,\mathsf{aux})$ represents the communication transcript between the pool operator and the puzzle solving/mining device, given a puzzle input x and auxiliary information aux.

- $\mathsf{Exf}(sk, y, \mathcal{T}_{x,\mathsf{aux}}) \rightarrow \overline{\mathcal{T}}_{x,\mathsf{aux}}$: The exfiltration algorithm takes as input a secret key sk for symmetric encryption, a puzzle solution $y \in \mathcal{Y}$ and the communication transcript for the corresponding puzzle input, auxiliary information pair (x, aux). It outputs a $\overline{\mathcal{T}}_{x,\mathsf{aux}}$ to replace $\mathcal{T}_{x,\mathsf{aux}}$.
- $\mathsf{Ret}(sk, \overline{\mathcal{T}}_{x,\mathsf{aux}}) \rightarrow y/\perp$: The retrieve algorithm takes as input a secret key sk for symmetric encryption and the communication transcript $\overline{\mathcal{T}}_{x,\mathsf{aux}}$ for the corresponding puzzle input, auxiliary information pair (x, aux). It outputs a puzzle solution $y \in \mathcal{Y}$ or $\perp$.
- $\overline{\mathsf{P}}.\overline{\mathsf{Eval}}(sk, x, \mathsf{aux}, \overline{\mathcal{X}})$: The subverted puzzle evaluation algorithm takes a secret key sk for symmetric encryption, an input from the domain x, auxiliary information aux and a set $\overline{\mathcal{X}} \subset \mathcal{X}$.
  - If $x \in \mathcal{X} \setminus \overline{\mathcal{X}}$:
    - Compute a puzzle solution y_1.
    - Run the exfiltration algorithm $\mathsf{Exf}(sk, y_1, \mathcal{T}_{x,\mathsf{aux}})$.
    - Compute and output a puzzle solution y_2 and auxiliary information aux. Such that $y_2 \neq y_1$.
  - Else output a puzzle solution y and a proof π.

$\mathbf{Exf}_{\mathcal{D}}$:

- **Setup.** In the setup phase, the challenger samples a bit $b \xleftarrow{\$} \{0,1\}$. If $b = 0$, the challenger runs the unsubverted puzzle algorithms $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ and sets $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta)$ to the adversary $\mathcal{B}$. Otherwise, it runs the subverted puzzle algorithms $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ and sets $\mathsf{pp} \leftarrow \overline{\mathsf{Setup}}(1^\lambda, \Delta)$. It sends $(\mathsf{pp}, \overline{\mathsf{P}})$ to the adversary $\mathcal{B}$.
- **Query.** In the query phase, the adversary $\mathcal{B}$ gets access to one of the puzzle evaluation oracles $\mathsf{O}_{\mathcal{T}}$ or $\mathsf{O}_{\overline{\mathcal{T}}}$ depending on the challenger's sampled bit b being 0 or 1. These oracles are defined as follows:
  - $\mathsf{O}_{\mathcal{T}}(\mathsf{pp}, \cdot, \cdot)$ takes as input an element $x \in \mathcal{X}$ and the corresponding pre-processing auxiliary information aux. It computes $y \leftarrow \mathsf{Eval}(x, \mathsf{aux})$ and it outputs the transcript for $\mathcal{T}_{x,\mathsf{aux}}$ for the input, auxiliary information pair.
  - $\mathsf{O}_{\overline{\mathcal{T}}}(\mathsf{pp}, \mathsf{state}, sk, \cdot, \cdot)$ takes as input an element $x \in \mathcal{X}$ and the corresponding pre-processing auxiliary information aux. It computes $y \leftarrow \overline{\mathsf{Eval}}(sk, x, \mathsf{aux})$ and it outputs the transcript $\overline{\mathcal{T}}_{x,\mathsf{aux}} \leftarrow \mathsf{Exf}(sk, y, \mathcal{T}_{x,\mathsf{aux}})$.
- **Guess.** In this phase, the adversary outputs its guess b' for b and wins the detection game if $b' = b$. We say that the game outputs 1 if the adversary wins and 0 otherwise.

  The advantage of an adversary $\mathcal{D}$ is defined as $\mathbf{Adv}^{\mathbf{Exf}}_{\mathcal{D}}(1^\lambda, \Delta) = \Pr[b' = b] - 1/2$.

Figure 5: The exfiltration process and the security for an exfiltration channel using the following exfiltration game $\mathbf{Exf}_{\mathcal{D}}$ played between a detector $\mathcal{D}$ and a challenger.

We again proceed by first proving some **general results** on leeching attacks where the size of the adversarially hard-coded bad inputs is negligible in the security parameter. Just as discussed in the previous sub-section, this models the setting where the detector $\mathcal{D}$ is testing its hardware to ensure proper performance. Since the set of bad inputs is negligibly small, the probability of detection is negligibly low. However, when the attacker plays the subversion game, it can easily detect whether the challenger is operating a subverted evaluation algorithm. This is because the attacker knows the set of hard-coded bad inputs.

Theorem 6.4. *For all leeching attacks where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$*, there exists no PPT detector* $\mathcal{D}$ *which wins the detection game* $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ *with non-negligible advantage.*

Proof sketch. Proof arguments are exactly the same as the proof for Theorem 6.1.

Theorem 6.5. *For all leeching attacks, there exists a PPT attacker* $\mathcal{A}$ *which wins the subversion game* $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ *with non-negligible advantage.*

Proof sketch. Proof arguments are exactly the same as the proof for Theorem 6.2.

Theorem 6.6. *If there exists a secure exfiltration channel* E*, all cryptographic puzzle schemes* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ *are susceptible to subversion via any leeching attacks where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$.

Proof. If there exists a secure exfiltration channel then the advantage of a detector $\mathcal{D}$ in detecting the leeching attack comes from its advantage in detecting via testing on a subverted input offline or detecting exfiltration. Hence, total advantage $\mathbf{Adv}^{\mathbf{total}}_{\mathcal{D}}$ of a detector in detecting a leeching attack:

$$\mathbf{Adv}^{\mathbf{total}}_{\mathcal{D}} = \mathbf{Adv}^{\mathbf{Exf}}_{\mathcal{D}}(1^\lambda, \Delta) + \mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^\lambda, \Delta)$$

The first term in the sum is due to the exfiltration channel being secure and the second term follows from Theorem 6.4. The above along with the fact that any subverting adversary can win the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ (Theorem 6.5), viewed under definition 5.3, completes our proof. □

Leeched Selfish-Mining Attack against Bitcoin. The following attack strategy can be employed by a Bitcoin mining pool operator who also manufactures mining hardware or colludes with the manufacturer. Let the set of devices manufactured by this manufacturer be D, a subset $\overline{\mathsf{D}}$ of D are subverted.

1. Embed the subverted set of inputs $\overline{\mathcal{X}}$ in the hardware during manufacturing.
2. Use the work shares (low-difficulty solutions) as a secure exfiltration channel E
3. On receiving an exfiltrated puzzle solution, follow the selfish-mining attack strategy and secretly mine on this fork of the blockchain on devices in the set $\mathsf{D} \setminus \overline{\mathsf{D}}$.

The following is a concrete method to utilize the malicious pool operator and the subverted hardware for exfiltration:

1. Given m work shares, utilize them communicate an n-bit string
2. Submit work shares such that the low-order bit of the i'th work share is also the i'th bit of the (encoded, see figure 5) puzzle solution

All the bits of the solution can be leaked even if ≈ 300 work shares are used. As a point of reference, consider that all Bitcoin mining pools currently set the work share difficulty for each device such that one work share is generated every 2-3 seconds [40]. Given the Bitcoin block time of 10 minutes, each miner is sending up to 300 work shares per block to its pool operator. *This amount of communication is enough to exfiltrate all the bits of a proof of work, encrypted or unencrypted.*

7 Attacks in the Blockchain Input Model

In the blockchain input model, the attacker is leveraging the fact that the mining device reads from the blockchain. Therefore, the adversary's influence on the blockchain's state determines the strength of any such attack. We operate with the simple assumption that any such adversary will own some fraction of the puzzle solving resource in the blockchain's network.

The adversary's attempt to bias the blockchain is then captured by the following game $\delta^{\mathcal{A}}_{\mathsf{bias}}$ played between a set of honest miners/puzzle solvers and an adversary $\mathcal{A}$. The goal of the adversary is to bias the probability distribution of the extract of a *decisive block*, which is a future block of interest to the adversary. The adversary holds influence over a δ fraction of the blockchain network's puzzle solving resource. The adversary wins if the extract x of the decisive block B, $\mathsf{Ext}(B) = x$ falls in a subset favorable to the

adversary. We denote this via the indicator function, $\mathbb{I}$ where $\mathbb{I}(x) = 1$ if $x \in \mathcal{F}$ and $\mathbb{I}(x) = 0$ otherwise, for $\mathcal{F}$ being the favorable set for the adversary. Clearly, the size of the set of favorable set $\mathcal{F}$ affects the adversary's success. This model is inspired from the work of Pierrot and Wesolowski [41] which models an adversary who biases a blockchain's entropy. The assumption that each new block has some min-entropy follows from the work of Bonneau, Clark and Goldfeder [42]. We describe the models in these related works in Appendix A.

Definition 7.1 (δ-bias Ability)**.** An adversary $\mathcal{A}$ has δ-bias ability if for $s_{\mathcal{A}}$ being the number of puzzle solutions per second the adversary can calculate and s being the number of puzzle solutions per second all miners combined (including $\mathcal{A}$) can calculate,

$$\delta = \frac{s_{\mathcal{A}}}{s}$$

This is a lower bound of bias considering the hashrate controlled by the attacker. The attacker might collude with or bribe other parties to increase their bias ability.

7.1 Load-Shedding Attack

In this setting, if an attacker aims to mount a load-shedding attack, they leverage their influence over the blockchain. Therefore, we first focus on the probability with which such an adversary can bias the blockchain's inputs based on their bias ability. Assuming the puzzle solvers start solving for a solution as soon as the a next potential block is broadcast, the adversary's goal is to broadcast a block favorable to it. In the process, the adversary might throw away some a valid puzzle solutions it finds. The variable μ represents the favorable set $\mathcal{F}$ as a fraction of the puzzle domain size. From the work of Pierrot and Wesolowski [41, Section 3.3], we borrow the following result which gives us the probability with which as adversary $\mathcal{A}$ with δ-bias ability wins the $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game:

$$\Pr\left[\delta^{\mathcal{A}}_{\mathsf{bias}} = 1\right] = \sum_{a>0}(\delta(1-\mu))^a\mu = \frac{\mu}{1-\delta(1-\mu)}$$

Similar to the cited work, we note that when $\delta \geq 0$ and $\mu \leq 1$, the above probability is better than μ.

As is clear from the calculation above, when μ is negligibly small compared to the security parameter, this attack does not work very well. However, we now discuss a variant of the attack which improves this success probability by making the attack stateful.

7.1.1 Stateful Load-shedding

In the stateful variant of the attack, the attacker makes the subversion stateful. This is an extremely crucial attack variant to consider due to the fact that an adversary mounting a stateful attack does not need a lot of influence over the blockchain's state. For example, a simple strategy can be to pick the low-order bit in the block header, subvert the puzzle solving device so that if it sees n consecutive blocks such that the low-order bits form a particular n-bit string (some $x \in \overline{\mathcal{X}}$) then it fails to compute the puzzle solution completely or becomes much slower. However, some constraints remain the same. The subverted set of inputs $\overline{\mathcal{X}}$ still has to be negligibly small in order to avoid getting detected. There could be many different ways to trigger failure based on states. The main issue is that now the attacker can feed in triggers slowly, block by block, to make the device fail eventually. Similar to previous attacks, the more influence the attacker holds over the blockchain's state, the stronger the attack. We show some general results on stateful load-shedding attacks in the blockchain input model.

More formally, the adversary's aim is to bias the blockchain's state over n blocks, such that for some encoding Enc, $\mathsf{Enc}(B_n) = 1$, where B_n represents the state of the blockchain over the last n blocks. Assuming an error-correcting code such that given a message length of k-bits and some failure rate p, it encodes the message in n-bits such that even if $p.k$ bits of the encoding get flipped, the message can still be decoded successfully. Therefore, for the attacker to successfully trigger a stateful load-shedding attack, it needs to win the following $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game (K)-out-of-n times (where $K = n - p.k$). The goal of the adversary is to bias the blockchain such that the extract of the decisive block $\mathsf{Ext}(B) = 1$, here $\mathsf{Ext}(B) = 1$ if the low-order bit of the block header is 1. This happens with probability $1/2$, therefore the favorable set of the adversary is $1/2$ of the possible outcomes. This gives us that $\mu = 1/2$. Let $P_{\mathcal{A}}$ be the win probability in the $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game as defined above,

$$P_{\mathcal{A}} = \frac{\mu}{1 - \delta(1-\mu)} = \frac{\frac{1}{2}}{1 - \frac{\delta}{2}} = \frac{1}{2-\delta} \text{ as } \mu = \frac{1}{2}$$

Therefore, given that the stateful load-shedding attack succeeds if the adversary wins the above game at least (K)-out-of-n times (where $K = n - p.k$). Let the string $s \in \{0,1\}$ represent the results of n consecutive $\delta^{\mathcal{A}}_{\mathsf{bias}}$ games played by the adversary. The i-th bit of s represents the result of the i-th $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game. Let s_1 represent the number of 1s in s, and s_0 represent the number of 0s in s. The attack success probability:

$$\Pr[s_1 \geq K] = \Pr[s_0 < K]$$

$$= 1 - \left(\sum_{i=1}^{K-1} \binom{n}{i} \left(\frac{1}{2-\delta}\right)^i \left(\frac{1-\delta}{2-\delta}\right)^{n-i} \right) = 1 - \frac{1}{(2-\delta)^n} \left(\sum_{i=1}^{K-1} \binom{n}{i} (1-\delta)^{n-i} \right)$$

As a result of this strategy, probability of subversion increases drastically while probability of detection does not change at all. We plot the values of K, n, the encoding parameters, the probability of success, next to different values of the adversarial resource fraction δ in Table 1. Note that such an attack can not be detected by any detection adversary (Theorem 6.1) and any attacker can figure out if the puzzle-solving device is running a subverted variant of the puzzle evaluation algorithm by simply checking if the encoding of the blockchain's state in the last n blocks, $\mathsf{Enc}(B_n)$ is a part of the subverted/trigger set $\overline{\mathcal{X}}$.

Attacker hashrate (δ)	RS Encoding Parameters RS(n, K)	Stateful Success Probability $1 - \frac{1}{(2-\delta)^n} \left(\sum_{i=1}^{K-1} \binom{n}{i} (1-\delta)^{n-i} \right)$	Stateless Success Probability
0.05	RS(520, 256)	0.83	$\mathsf{negl}(\lambda)$
0.10	RS(526, 256)	0.96	$\mathsf{negl}(\lambda)$
0.15	RS(533, 256)	0.997	$\mathsf{negl}(\lambda)$
0.20	RS(541, 256)	0.9997	$\mathsf{negl}(\lambda)$
0.25	RS(549, 256)	≈ 1	$\mathsf{negl}(\lambda)$

Table 1: Attack success probabilities and Reed-Solomon Encoding Parameters (with 1-bit symbols) for the stateful Load-shedding attack (in the blockchain input model), depending on attacker hashrate for a 256-bits trigger. Note that the attacker can have multiple triggers, but the number of triggers $\overline{\mathcal{X}}$ has to be negligible in terms of the security parameter to avoid detection. This also assumes that attacker hashrate never goes below δ for n consecutive blocks, changes will decrease probability of success.

8 Securing Puzzles against ASAs

8.1 Via Unpredictable Puzzle Pre-processing

The unpredictability of the pre-processing that the puzzle construction supports captures how much control and flexibility the evaluator has, once a puzzle instance is received. If the evaluator has a lot of flexibility then the auxiliary information aux used by the pre-processing algorithm is a crucial part of the algorithm. So much so that without knowledge of the randomness utilized in pre-processing, it could be hard to predict the pre-processed input. Such a mechanism is a great countermeasure against all input-induced attacks. Given that the set of subverted inputs $\overline{\mathcal{X}}$ is small enough, if the pre-processing algorithm has the unpredictability property as defined below, then the adversary is not able to trigger the subverted device with non-negligible probability. Unpredictable pre-processing prevents any adversary from knowing exactly what bits are input to the subverted device, which is all that is needed to prevent an adversary benefiting from an input-induced attack.

Definition 8.1 (Unpredictable Pre-processing)**.** The pre-processing algorithm Pre has the unpredictability property if no PPT adversary $\mathcal{A}$ can predict the output of the algorithm, given only the unprocessed puzzle input and no knowledge of the randomness used to generate aux.

$$\Pr\left[\begin{array}{c|c} x \leftarrow \mathsf{Pre}(x', \mathsf{aux}) & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta) \\ x_{\mathcal{A}} = x & x_{\mathcal{A}} \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}, x') \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Now, while the identity function is a validate candidate for the pre-processing algorithm Pre, it clearly does not have the unpredictability property. The Bitcoin proof-of-work (Section 3) puzzle is a good example of puzzles with unpredictable pre-processing.

Theorem 8.1. *In the direct input model, if the evaluator uses a puzzle pre-processing algorithm not satisfying the unpredictability property, all cryptographic puzzle schemes* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Verify})$ *are susceptible to subversion via any stateless load-shedding attacks where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$.

Proof. This is a corollary of Theorem 6.3.

Theorem 8.2. *In the direct input model, if the evaluator uses a puzzle pre-processing algorithm satisfying the unpredictability property, then there exists no PPT attacker* $\mathcal{A}$ *which wins the subversion game* $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ *with non-negligible advantage for any input-induced attack where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$.

Proof. If the puzzle pre-processing algorithm satisfies the unpredictability property then the probability with which the adversary can feed the device a subverted input is

$$= \sum_{x_{\mathcal{A}} \in \overline{\mathcal{X}}} \Pr\left[\begin{array}{c|c} x \leftarrow \mathsf{Pre}(x', \mathsf{aux}) & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta) \\ x_{\mathcal{A}} = x & x_{\mathcal{A}} \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}, x') \end{array}\right]$$

$$= |\overline{\mathcal{X}}|.\Pr\left[\begin{array}{c|c} x \leftarrow \mathsf{Pre}(x', \mathsf{aux}) & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta) \\ x_{\mathcal{A}} = x & x_{\mathcal{A}} \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}, x') \end{array}\right]$$

$$= \mathsf{negl}(\lambda).\mathsf{negl}'(\lambda) \text{ (invoking definition 8.1)}$$

Theorem 8.3. *In the direct input model, all cryptographic puzzle schemes* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ *utilizing a puzzle pre-processing algorithm satisfying the unpredictability property are secure against subversion via all input-induced attacks.*

Proof sketch. We split the proof into two parts:

1. if the size of the set of subverted inputs $\left|\overline{\mathcal{X}}\right|$ is a non-negligible function of λ then the detector $\mathcal{D}$ detects the subverted inputs with non-negligible probability just by polynomial testing
2. if $\left|\overline{\mathcal{X}}\right| = \mathsf{negl}(\lambda)$ then the attacker $\mathcal{A}$ can not win the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ with non-negligible probability by Theorem 8.2

In either case, by definition 5.3, all cryptographic puzzle schemes $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ utilizing a puzzle pre-processing algorithm satisfying the unpredictability property are secure against subversion via all input-induced attacks. □

8.2 Protecting proof-of-work against ASAs

To ensure that the pooled mining protocols with pool operators allow unpredictable pre-processing of the proof-of-work puzzle input we suggest a new standardized protocol for pooled mining under pool operators (and not peer-to-peer mining). Such standardization also demands that a public implementation of such a protocol is utilized. While the protocol described above is specifically for Bitcoin's proof-of-work, small modifications make it applicable to most proof-of-work cryptocurrencies.

Current pooled mining protocol [43]. The pool operator creates a candidate block by aggregating and creating a Merkle tree of transactions to be included, adding a coinbase transaction and linking the proposed candidate to the previous block. The coinbase transaction is the first transaction verified transaction on the list of verified transactions in a proposed block. Its value corresponds to the current block reward. The miners participating in the pool then iterate over different values of the time and nonce fields as specified in the block hash algorithm (in Section 3) to find proof-of-work solutions.

Our proposed pooled mining protocol. The main change we propose is that the coinbase transaction should be modified by the miners. The coinbase transactions have extra fields for data and extraNonce which can be modified to provide upto 100 bytes [44] of extra space to iterate over for miners. The goal of this suggested change is to allow miners the flexibility of picking the inputs to their mining devices. As demonstrated by the trigger example for the direct input model (Section 5.1), allowing the pool operator to pick the exact string input to the mining device allows for the load-shedding and leeching as described in the previous sections. Modifications to the coinbase are not a new concept and enforcing such changes can be done by the mining software. We spare the details here as similar changes have been suggested in the mining protocol by SlushPool [45], although due to reasons such as efficient mining work allotment etc. We also recommend that all proof-of-work cryptocurrencies instruct users to only use pools with open-source mining software. Similar ASAs can be mounted at the software level as well and there is no clear way to prevent it if the mining software is closed source.

Theorem 8.4. *proof-of-work pre-processing as specified above satisfies the unpredictability property as defined in definition 8.1.*

Proof sketch. If the number of bits modified by the miner, before inputting the resulting hash to the mining device is some linear function $f(\lambda)$ in the security parameter, then probability of the pool operator correctly estimating the exact input to the mining device is $\mathsf{negl}(\lambda) = \frac{1}{2^{f(\lambda)}}$. Considering that the proof-of-work pre-processing as specified above allows upto 100 bytes of entropy to be used by the miners, it satisfies the unpredictability property as defined in definition 8.1.

8.3 Masking VDFs

Verifiable delay functions (VDFs) are now being utilized by multiple blockchain protocols[4] [21, 29]. We now discuss a method to mask the VDF input from a puzzle solving device in case the input was adversarially biased to cause input-induced attacks. This is one way to ensure unpredictable puzzle pre-processing for VDFs. We demonstrate a masking protocol using Pietrzak's VDF construction [30]. Our masking protocol changes the original scheme minimally and the extra computation incurred is also minimal, as demonstrated in Table 2. Our implementation is based on C++ code from Hosszejni's work [46]. The numbers reflect computation times on a machine running Intel's i7-8565U CPU with a 16GB RAM. Note that these should be used for reference as when such a protocol is utilized at scale, it will be using specialized, blazing fast hardware [47] for the computation.

Halving Subprotocol from [30]:
On input (N, x, T, y)

Set $\mathcal{V}_{\mathsf{dec}} = \mathsf{nil}$
While $\mathcal{V}_{\mathsf{dec}} == \mathsf{nil}$ {
If $T = 1$ and $y = x^{2^T}$:
    $\mathcal{V}_{\mathsf{dec}} = 1$
Else If $T > 1$:
    $\mathcal{P}$ computes $\mu = x^{2^{T/2}}$
    $\mathcal{P}$ sends μ to $\mathcal{V}$
    If $\mu \notin QR_N^+$:
        $\mathcal{V}_{\mathsf{dec}} = 0$
        break
    Else:
        $\mathcal{V}$ samples a random $r \xleftarrow{\$} \mathbb{Z}_{2^\lambda}$
        $\mathcal{V}$ sends r to $\mathcal{P}$
        If var is even:
            $\mathcal{P}, \mathcal{V}$ output $(N, x', T/2, y')$
        Else:
            $\mathcal{P}, \mathcal{V}$ output $(N, x', \frac{T+1}{2}, y'^2)$
Set $T = T/2$
}

Here $x' := x^r \cdot \mu \left(= x^{r+2^{T/2}}\right)$ and $y' := \mu^r \cdot y \left(= x^{r \cdot 2^{T/2} + 2^T}\right)$

Masked Halving:
$\mathcal{P}$ on input (N, x, T, y):

    Picks a pre-evaluated b^{2^T} for a VDF instance b s.t.
    $\alpha = \{b^{2^T}, b^{2^{T/2}}, \ldots, b^2\}$,
    $\beta = \{(b^{2^T})^{-1}, (b^{2^{T/2}})^{-1}, \ldots, (b^2)^{-1}\}$
    is precomputed
    Sets $x_b = x.b$

The device $\mathcal{D}$ on input (N, x_b, T):
    Computes $\gamma = \{x_b^{2^T}, x_b^{2^{T/2}}, \ldots, x_b^2\}$
    Sends γ to $\mathcal{P}$

$\mathcal{P}$ computes $y = y_b.(b^{2^T})^{-1} = \gamma[0].\beta[0]$
$\mathcal{P}, \mathcal{V}$ now have (N, x, T, y)

At each halving step:
    $\mathcal{P}$ computes:
    $\mu = x_b^{2^{T/2}}.(b^{2^{T/2}})^{-1}$, $x' := x^r \cdot \mu$, $y' := \mu^r \cdot y$

$\mathcal{P}, \mathcal{V}$ interaction continues as specified

Figure 6: The halving subprotocol and our proposed masked halving protocol

Pietrzak's VDF construction builds on the RSW [48] time-lock puzzle construction. The RSW time-lock puzzle is a simple construction which takes an element $x \in \mathbb{Z}_N^*$ and the main goal of the puzzle evaluation algorithm is to calculate $y = x^{2^T} \pmod N$, where T specifies the puzzle difficulty. To convert this into a VDF, Pietrzak's work builds a protocol where a prover $\mathcal{P}$ convinces a verifier $\mathcal{V}$ that it solved an RSW puzzle. For reasons unrelated to our discussion, the VDF protocol utilizes the quadratic residue group $\left(QR_N^+, \circ\right)$ instead $(\mathbb{Z}_N^*, \cdot)$. We refer interested reader's to [30, Sec 2.2] for a detailed discussion.

[4] A full decription of Ethereum's Beacon Chain protocol is available in Appendix C.

The protocol proceeds as follows:

- The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ have an RSW puzzle (N, x, T) as common input along with the security parameter λ. Here $T \in \mathbb{N}$, $N = p \cdot q$ is the product of safe primes and the input $x \in QR_N^+$.
- The prover $\mathcal{P}$ computes T sequential squarings of the input x in the quadratic residue group QR_N^+ and sends $y = x^{2^T}$ to the verifier $\mathcal{V}$.
- The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ then iteratively engage in the "halving protocol" as described below. This subprotocol starts with the common input (N, x, T, y) and the output is either $(N, x', \lceil T/2 \rceil, y')$ or the verifier outputs $0/1$ and the protocol stops.

In the VDF hardware setting, the squarings and the halving protocol steps are computed in the hardware/computing device. The VDF computing device gets as input (N, x, T) and the randomness r for each round of the halving protocol. It outputs the puzzle solution y and the halving protocol outputs for each round. We present our masked halving protocol in Figure 6 and show its correctness. The goal is to ensure that no adversary who manufactured/subverted this device can feed in inputs and expect them to malfunction. Therefore, in the masked halving protocol there is another party the VDF computing device $\mathcal{D}$, other than the prover $\mathcal{P}$ and the $\mathcal{V}$. The masking ensures that each puzzle input to the VDF computing device is transformed unpredictably before the device sees it. The prover $\mathcal{P}$ wants to utilize the VDF computing device $\mathcal{D}$ to evaluate the VDF and an accompanying proof while ensuring that the device can not guess the original input.

$\lvert N\rvert$	2^T	Evaluation + Proving time (w/o Masking)	Pre-computation Cost ($\lfloor \log(T) \rfloor$ inverses)	Masking delay ($\lfloor \log(T) \rfloor$ multiplications)
512	2^8	1.7 ms	2.1×10^{-3} ms	3.3×10^{-5} ms
	2^{16}	31 ms	2.9×10^{-3} ms	4.1×10^{-5} ms
	2^{24}	2.1 s	3.8×10^{-3} ms	6.5×10^{-5} ms
	2^{32}	64 s	3.8×10^{-3} ms	6.5×10^{-3} ms
1024	2^8	4.6 ms	3.4×10^{-3} ms	4.8×10^{-5} ms
	2^{16}	92 ms	4.4×10^{-3} ms	6.1×10^{-5} ms
	2^{24}	7.4 s	6.1×10^{-3} ms	7.6×10^{-5} ms
	2^{32}	136 s	6.1×10^{-3} ms	7.6×10^{-5} ms
2048	2^8	21 ms	1.7×10^{-2} ms	2.2×10^{-4} ms
	2^{16}	656 ms	2.4×10^{-2} ms	3.1×10^{-4} ms
	2^{24}	19.5 s	4.1×10^{-2} ms	3.2×10^{-4} ms
	2^{32}	308 s	4.1×10^{-2} ms	3.2×10^{-4} ms

Table 2: Extra delay/computation required for Masked VDFs. For different lengths of N (RSA modulus) and different number of exponentiations T, the above estimates the extra costs incurred for masking. Masking delay is a very small fraction of online delay and the overhead $\frac{\log(T)}{T}$ decreases with increasing T.

Masked Halving Correctness. The only change to the original halving protocol by the masked halving protocol is that instead of directly computing $y = x^{2^T}$, masked halving computes $x_b = x.b$ and then computes $y_b = x_b^{2^T}$. To unmask the final result, we compute $y = y_b.(b^{2^T})^{-1}$. If b was a previous instance then b^{2^T} is known already. Furthermore, during the evaluation of b^{2^T} the intermediate steps can be stored

in memory as $\alpha = \{b^{2^T}, b^{2^{T/2}}, \ldots, b^2\}$. The terms in $\beta = \{(b^{2^T})^{-1}, (b^{2^{T/2}})^{-1}, \ldots, (b^2)^{-1}\}$ can be precomputed by the evaluator. In each halving now, to compute μ the evaluator computes $x_b^{2^{T/2}}.(b^{2^{T/2}})^{-1} = x^{2^{T/2}}$ which is the value of μ in the original halving protocol. This is the only change in the halving process, making masked halving correct and minimally taxing.

8.4 Testing Puzzle Evaluators

The best example of wide-scale use of puzzle evaluators in the real-world is Bitcoin and other proof-of-work mining hardware. Currently, there is no standardized testing against flaws, other than computing the number of operations per second. However, in light of our proposed attacks, owners of such devices might be interested in testing for subversions. However, every time period a mining device is not actually being used for mining, its incurring a loss to its owner. More testing in this sense also benefits mining device manufacturers who also use their manufactured devices. *Resulting in a paradox where distrust in the manufacturer somehow benefits them.* Following tests can be used for this:

- **Use existing puzzles with known solutions.** In any proof-of-work cryptocurrency, solutions at the required difficulty level are found at least every few minutes. To test the mining devices, one can simply supply nonce ranges for the known solution in the past blocks, along with the merkle tree root of proposed block and the previous block hash as described in Section 3.
- **Keep testing against latest proof-of-work solutions.** Using the same strategy as above, the devices can be tested for a low cost, against the latest proof-of-work solutions. This is to ensure that subversions which come into play only after the device computes a certain number of hashes can be noticed. If the previous testing strategy becomes widely used then subverting adversaries can move towards more sophisticated strategies to avoid being caught. Moreover, this also makes it possible to figure out if the device is under a leeching attack. If the latest solution is some (hashMerkleRoot, nonce, time) tuple already exhausted by the device under scrutiny without finding a solution then there is a high likelihood that the solution was exfiltrated. Especially if considering the fact that mining pools divide up work into pool members, at a given time, across all the devices participating in mining, the number of devices computing the hash for the same (hashMerkleRoot, nonce, time) tuple is not high.

9 Real-world Concerns

Beyond the attacks we propose and analyze, following are real-world concerns relevant to puzzle-solving devices:

Limits of testing. Previous work [9] models the oracles in the detection game as instantaneous response oracles. While they point out that the goal of their model is not to evade all forms of detection, these factors are crucial for real-world attack considerations. Keeping that in mind, our attacks still work in the model where the detector considers oracle response times. However, there are out of model attacks which still can not be detected via our testing regime.

Very strong attackers. Such attackers can come up with numerous attack strategies which are not input-induced. For example:

- **Simple timer-based attack.** The attacker fits a small clock in the puzzle solving device. After a certain time T, the device's puzzle solving ability drops by some fraction. Notice that such drops

are explained as device aging etc. It is not clear how this attack can be detected in the real world. Another attack in a similar vein is one where the device's puzzle solving ability drops after computing a certain number of puzzle solutions (at any difficulty level).

- **Subverting only real-world difficulty parameters.** If we use the example of proof of work mining. The testing of mining devices currently includes testing the number of hashes computed per second and checking solutions outputted at lower difficulty levels. An attacker can ensure that the device is subverted only on puzzle instances of difficulty levels relevant in real-world applications. The detector could then try to verify the devices behavior on puzzles at real-world difficulty parameters where the solution is already known, for example, known bitcoin blocks (by feeding in a small nonce range and Merkle root of the proposed block). The attacker can easily counter such detection by ensuring the device never fails on existing blocks since this is a set of a few hundred thousand puzzle instances. The detector is then only left with the option to test the device on the freshly mined blocks available after the purchase of the device.

Stronger Input-induced attacks. The attacks we describe and analyze throughout the paper are input-induced attacks where the trigger inputs are a negligible fraction of the puzzle input domain. This is mainly to ensure that a PPT detector can detect all attacks which are triggered by a higher fraction of the puzzle input domain. An attacker might adopt a strategy where it has a long-enough time window to reap the benefits of the subversion before being detected. At the point of detection, the attacker has already succeeded and made a huge profit.

Real-world detection costs. We model our detector as a PPT party which detects all input-induced attacks except ones which succeed on a negligibly small set of inputs. However, in the real-world there are many other factors to consider. Polynomial sampling might result in detection but this implies a costly testing period in the real world. The puzzle solver is losing valuable evaluation time every testing cycle. This is economic disincentivization of testing means that in the real world, an attacker can increase its input-induced attack surface. There are also device life-cycles to consider, for example, a bitcoin mining rig has a life-cycle of less than a couple of years [49]. The attacker can mount attacks which are active only once in the life-cycle of the hardware.

10 Conclusion

The main focus of our work has been on attacks under the following assumptions:

1. **Strong detector.** The detector $\mathcal{D}$, as demonstrated in the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$, gets access to either the subverted or the unsubverted puzzle solving device and can query it on polynomial number of inputs. This ensures that any attack which subverts a high fraction (non-negligible in terms of the security parameter) of inputs in the domain, are detected with high probability.

2. **Blackbox device access.** The puzzle solving/evaluation device can be accessed as a blackbox. The detector can try to test it on different difficulty and security parameters.

3. **Input-induced subversion.** We study attacks where the attack behavior is triggered through a special input or set of inputs. The detector's goal therefore is to detect such triggers/attack behavior

Beyond the attacks we propose and analyze, there are several real-world concerns relevant to puzzle-solving devices such as the limits of testing, some out of model attacks which are hard to prevent and

other carefully crafted subversion strategies. We provide a discussion on all these concerns and consider stronger input-induced attacks and attackers in Section 9. We hope this discussion serves as motivation for future work.

Acknowledgments

The first and second authors are supported in part by NSF under awards CNS-1653110 and CNS-1801479 and the Office of Naval Research under contract N00014-19-1-2292. The second author is also supported in part by DARPA under Contract No. HR001120C0084. The authors would also like to thank Maximilian Zinkus for helpful discussions on error-correcting codes.

References

[1] J. I. Wong, "Research: Hackers could install backdoor in bitcoin cold storage," 2015, https://www.coindesk.com/markets/2015/01/16/research-hackers-could-install-backdoor-in-bitcoin-cold-storage/.

[2] L. Stefanko, "Crypto malware in patched wallets targeting android and ios devices," 2022, https://www.welivesecurity.com/2022/03/24/crypto-malware-patched-wallets-targeting-android-ios-devices/.

[3] "Hacker infects node.js package to steal from bitcoin wallets," 2018, https://www.trendmicro.com/vinfo/fr/security/news/cybercrime-and-digital-threats/hacker-infects-node-js-package-to-steal-from-bitcoin-wallets.

[4] A. Taylor, "Watch out for the 'rug pull' crypto scam that's tricking investors out of millions," 2022, https://fortune.com/2022/03/02/crypto-scam-rug-pull-what-is-it/.

[5] D. J. Bernstein, T. Lange, and R. Niederhagen, "Dual EC: A standardized back door," in *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, P. Y. A. Ryan, D. Naccache, and J. Quisquater, Eds., 2016.

[6] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R. Weinmann, E. Rescorla, and H. Shacham, "A systematic analysis of the juniper dual EC incident," in *ACM CCS*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., 2016.

[7] *Equation Group: Questions and answers.* Kaspersky Lab HQ, 2015.

[8] A. L. Young and M. Yung, "Kleptography: Using cryptography against cryptography," in *EUROCRYPT 1997*, W. Fumy, Ed., 1997.

[9] M. Bellare, J. Jaeger, and D. Kane, "Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks," in *ACM CCS 2015*, 2015, pp. 1431–1440.

[10] G. Ateniese, B. Magri, and D. Venturi, "Subversion-resilient signature schemes," in *ACM CCS*, I. Ray, N. Li, and C. Kruegel, Eds., 2015.

[11] S. Berndt and M. Liskiewicz, "Algorithm substitution attacks from a steganographic perspective," in *ACM CCS 2017*, 2017, pp. 1649–1660.

[12] M. Fischlin and S. Mazaheri, "Self-guarding cryptographic protocols against algorithm substitution attacks," in *IEEE CSF 2018*, 2018, pp. 76–90.

[13] R. Chen, X. Huang, and M. Yung, "Subvert KEM to break DEM: practical algorithm-substitution attacks on public-key encryption," in *ASIACRYPT 2020*, 2020, pp. 98–128.

[14] P. Hodges and D. Stebila, "Algorithm substitution attacks: State reset detection and asymmetric modifications," *IACR Trans. Symmetric Cryptol.*, vol. 2021, no. 2, pp. 389–422, 2021.

[15] S. Berndt, J. Wichelmann, C. Pott, T.-H. Traving, and T. Eisenbarth, "Asap: Algorithm substitution attacks on cryptographic protocols," in *ASIACCS*, 2022.

[16] M. Bellare, K. G. Paterson, and P. Rogaway, "Security of symmetric encryption against mass surveillance," in *CRYPTO 2014*, 2014, pp. 1–19.

[17] M. Bellare, D. Kane, and P. Rogaway, "Big-key symmetric encryption: Resisting key exfiltration," in *CRYPTO*, M. Robshaw and J. Katz, Eds., 2016.

[18] B. Groza and B. Warinschi, "Cryptographic puzzles and dos resilience, revisited," *Des. Codes Cryptogr.*, vol. 73, no. 1, pp. 177–207, 2014.

[19] A. Juels and J. G. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," in *NDSS*, 1999.

[20] "Bitcoin's block hashing algorithm," 2021, https://en.bitcoin.it/wiki/Block_hashing_algorithm.

[21] J. Drake, "Minimal vdf randomness beacon," in *Ethereum Research Forum*, 2018. [Online]. Available: https://ethresear.ch/t/minimal-vdf-randomness-beacon/3566

[22] M. M. Labs, "51% attacks on cryptocurrencies," 2020, =https://dci.mit.edu/51-attacks.

[23] O. Williams-Grut, "Chinese bitcoin mining giant bitmain had revenues of $2.8 billion in the first half of the year," 2018, Link to article.

[24] "Bitmain," 2021, https://en.wikipedia.org/wiki/Bitmain.

[25] J. Têtu, L. Trudeau, M. V. Beirendonck, A. Balatsoukas-Stimming, and P. Giard, "A standalone fpga-based miner for lyra2rev2 cryptocurrencies," *IEEE Trans. Circuits Syst. I Fundam. Theory Appl.*, 2020.

[26] J. Tromp, "Cuckoo cycle: A memory bound graph-theoretic proof-of-work," in *BITCOIN workshop (FC)*, M. Brenner, N. Christin, B. Johnson, and K. Rohloff, Eds., 2015.

[27] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, "Proofs of space," in *CRYPTO 15*, 2015, pp. 585–605.

[28] H. Abusalah, J. Alwen, B. Cohen, D. Khilko, K. Pietrzak, and L. Reyzin, "Beyond hellman's time-memory trade-offs with applications to proofs of space," in *ASIACRYPT 2017*, 2017, pp. 357–379.

[29] B. Cohen and K. Pietrzak, "The chia network blockchain," 2019, https://www.chia.net/greenpaper/.

[30] K. Pietrzak, "Simple verifiable delay functions," in *ITCS 2019*, 2019, pp. 60:1–60:15.

[31] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *Financial Cryptography and Data Security FC 2014*, 2014.

[32] "Bitcoin's difficulty based on network hashrate," 2021, https://en.bitcoin.it/wiki/Difficulty#What_network_hash_rate_results_in_a_given_difficulty.3F.

[33] D. Stebila, L. Kuppusamy, J. Rangasamy, C. Boyd, and J. M. G. Nieto, "Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols," in *CT-RSA 2011*, 2011, pp. 284–301.

[34] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions," in *CRYPTO 2018*, 2018.

[35] B. Wesolowski, "Efficient verifiable delay functions," in *EUROCRYPT 2019*, 2019.

[36] J. Redman, "Mining report highlights china's asic manufacturing improvements and dominance," 2020, Link to article.

[37] CoinMarketCap, "Top pow tokens by market capitalization," 2022, https://coinmarketcap.com/view/pow/.

[38] Chia, "Pooling faq," 2021, https://github.com/Chia-Network/chia-blockchain/wiki/Pooling-FAQ.

[39] AntMiner, "Antminer bitcoin mining device installation guide," 2020, https://www.antminerdistribution.com/wp-content/uploads/2020/05/S19-Pro-manual.pdf.

[40] SlushPool, "Bitcoin mining pools: Luck, shares, and estimated hashrate explained," 2021, https://braiins.com/blog/bitcoin-mining-pools-luck-shares-estimated-hashrate.

[41] C. Pierrot and B. Wesolowski, "Malleability of the blockchain's entropy," *Cryptography and Communications*, vol. 10, no. 1, pp. 211–233, 2018.

[42] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source," *IACR Cryptol. ePrint Arch.*, p. 1015, 2015.

[43] A. Antonopoulos, "Mastering bitcoin," 2017, https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/.

[44] "Bitcoin's transaction format," 2021, https://en.bitcoin.it/wiki/Transaction.

[45] SlushPool, "Stratum v1 docs: Mining protocol," 2022, https://braiins.com/stratum-v1/docs.

[46] D. Hosszejni, "Master's thesis: Verifiable delay functions and their applications," 2019.

[47] C. Kim, "Ethereum foundation and others weigh $15 million bid to build randomness tech," in *CoinDesk*, 2019.

[48] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," 1996.

[49] "Bitcoin mining producing tonnes of waste," 2021, https://www.bbc.com/news/technology-58572385.

[50] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin, "Randomness extraction and key derivation using the cbc, cascade and HMAC modes," in *CRYPTO 2004*, 2004, pp. 494–510.

A Malleability of Blockchain's Entropy

We adopt the model of an adversary who malleates a blockchain's entropy from the work of Pierrot and Wesolowski [41]. This model is specifically applicable to a proof of work based blockchain. They first start by assuming that the underlying hash function used for proof of work is secure. For d being the difficulty of the proof of work, each new block contains d bits of computational min-entropy, this was first justified in [42]. We know that there exists a cryptographic extractor [50] Ext, which maps each block's d bits of min-entropy to $\lfloor d/2 \rfloor$ of near-uniform bits.

The adversary's attempt to mine favorable blocks is then captured by the following game played between a set of honest miners and an adversary $\mathcal{A}$. The goal of the adversary is to bias the probability distribution of the extract of the *decisive block*, which is a future block of interest to the adversary. The adversary holds influence over a δ fraction of the blockchain. For $s_{\mathcal{A}}$ being the number of hashes per second the adversary can calculate and s being the number of hashes per second all miners combined (including $\mathcal{A}$) can calculate,

$$\delta = \frac{s_{\mathcal{A}}}{s}$$

The game starts when a fixed *initial block*, indexed 0 is received and ends when block height $n + f$ is reached. Here n is the block height of the decisive block and it takes f additional blocks to finalize the decisive block. This is to ensure that there are no forks of the blockchain which would alter the decisive block. The adversary wins if the extract x of the decisive block B, $\mathsf{Ext}(B) = x$ falls in a subset favorable to the adversary. We denote this via the indicator function, $\mathbb{I}$ where $\mathbb{I}(x) = 1$ if $x \in \mathcal{F}$ and $\mathbb{I}(x) = 0$ otherwise, for $\mathcal{F}$ being the favorable set for the adversary.

Our interest lies in the setting where the extractor Ext is private and set by the adversary $\mathcal{A}$. The extractor values are used by the adversary who mounts the load shedding attack as a trigger. The load shedding adversary therefore uses the blockchain to signal the mining evaluation hardware/software to malfunction.

B Real-world Cryptographic Puzzles

See figure B.

C Ethereum's Beacon Chain

Ethereum plans on using VDFs as a source of unpredictable randomness. An example of this requirement in Ethereum 2.0 is the following. A small of group of validators are required to progressively build a chain of randomness, this chain is termed the "Beacon-chain"[21]. Assuming a global clock and splitting time into contiguous 8-second blocks and 128-slot epochs, one value O is generated per epoch $\mathcal{E}$. This generated random value is used to select a validator who then gets the opportunity to propose the next block to be added to the blockchain and consequently reaps the block reward. In the ideal scenario, with unbiased randomness the frequency with which a validator is selected is directly proportional their stake in the system. However, if a malicious actor is successful in biasing the randomness they can sample strings such that they can select the one which benefits them most. Currently the randomness O is obtained from the reveals of a RANDAO commit-reveal scheme used to generate a random number where the commits are inputs produced by the validators during epoch $\mathcal{E}$. In a RANDAO commit-reveal scheme, every beacon

chain proposer is committed to 32 bytes of local entropy. (In practice a chain of commit-reveals is setup with a hash onion for validator registration.) Beacon chain proposers may reveal their local entropy by extending the canonical beacon chain with a block. Honest proposers are expected to keep their local entropy private until their assigned slot. The beacon chain maintains 32 bytes of on-chain entropy by XOR-ing the local entropy revealed at every block. However such a commit-reveal scheme is biasable: a malicious validator that controls the last reveal can choose to reveal or not, giving them some control over the choice of O based on their decision. Therefore, one proposed way to make O unbiasable is to pass O through a *verifiable delay function* (VDF) which is guaranteed to be slow to compute. This is done so that all validators must choose whether to reveal or not *before* they know the output of the VDF.

Let $B(\cdot, b_i)$ be a function which produces biasable randomness on input an epoch i and the 32 bytes of local entropy b_i from the beacon chain proposer selected for this round. Then the randomness beacon $R(\cdot)$ output for epoch i is computed as follows:

$$B(i, b_i) = B(i-1, b_{i-1}) \oplus b_i$$

$$R(i) = \mathsf{VDF.Eval}(pp, B(i-T, b_{i-T}))$$

Assuming it takes real world time T for the commodity hardware to compute the VDF output. This construction ensures that the VDF input is available at the same time to all parties interesting in computing its output. The Ethereum foundation may spend over $15 million [47] for the development of specialized hardware to provide to the validators, in order to achieve lowest evaluation time for any VDF parameters.

Definition B.1 (Bitcoin Proof of Work as a Cryptographic Puzzle)**.** Bitcoin proof of work can be formalized as a cryptographic puzzle as follows:

- $\mathsf{Setup}(1^\lambda, \Delta) \rightarrow \mathsf{pp}$: The security parameter 1^λ contains the specifications for the hash function (SHA-256) used for Bitcoin' proof of work including the number of bits of security it provides. The difficulty parameter is an integer which indicates minimum amount of leading zeroes required for a valid proof of work solution.
- $\mathsf{Pre}(x', \mathsf{aux}) \rightarrow x$: The unprocessed puzzle input x' is the previous block header/proof of work. The auxiliary input aux is the Merkle tree root of the transactions confirmed and included in the proposed block. The pre-processed input is computed as $x = H(\mathsf{aux}, x')$.
- $\mathsf{Eval}(x, \mathsf{aux}) \rightarrow y/\ \perp$: The evaluation algorithm checks if for any input r in the nonce range, the hash $H(r, x)$ has $\geq \Delta$ many preceding zeroes. If it finds such a solution it outputs $y = r$, otherwise outputs $\perp$.
- $\mathsf{Verify}(x, \mathsf{aux}, y) \rightarrow 0/1$: If $H(y, x)$ has $\geq \Delta$ many preceding zeroes, output 1 and 0 otherwise.

It is worth noting that in the real world, the pre-processing phase can possibly be completely predictable. Typical bitcoin mining logic is to include transactions with the highest miner fees. If this is the case then the input to the Eval algorithm become predictable. Since the inputs to Eval are the inputs to the mining rig, this opens up the possibility of input-induced attacks which we discuss in Sections 6 and 7. However, this does not imply that Bitcoin proof of work is a bad cryptographic puzzle construction. This particular issue can be easily fixed by introducing some entropy into the process of selecting transactions.

Definition B.2 (Verifiable Delay Functions)**.** A verifiable delay function (VDF) consists of the following algorithms:

- $\mathsf{Setup}(1^\lambda, \Delta) \rightarrow \mathsf{pp}$: The VDF generation algorithm takes as input a security parameter 1^λ and a difficulty parameter Δ and outputs public parameters pp which fix the domain $\mathcal{X}$ and range $\mathcal{Y}$ of the VDF and other information required to compute a VDF or verify a solution.
- $\mathsf{Eval}(\mathsf{pp}, x) \rightarrow (y, \pi)$: The VDF evaluation algorithm takes as input the public parameters pp, an input from the domain x. It outputs a VDF solution y and a proof π.
- $\mathsf{Verify}(\mathsf{pp}, x, y, \pi) \rightarrow 0/1$: The VDF verification algorithm takes as input the public parameters pp, an input from the domain x, an input from the range y and a proof π. It outputs either 0 or 1.

Additionally, VDFs must satisfy the correctness, soundness and sequentiality definitions as defined below.

Definition B.3 (Correctness)**.** A verifiable delay function is correct if $\forall \lambda, \Delta, \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta)$, and $\forall x \in \mathcal{X}$ if $(y, \pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x)$ then $\mathsf{Verify}(\mathsf{pp}, x, y, \pi) = 1$.

Definition B.4 (Soundness)**.** We require that an adversary can not get a verifier to accept an incorrect VDF solution.

$$\Pr\left[\begin{array}{c|c} \mathsf{Verify}(\mathsf{pp}, x, y, \pi) = 1 & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta) \\ y \neq \mathsf{Eval}(\mathsf{pp}, x) & (x, y, \pi) \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Definition B.5 (VDF as a Cryptographic Puzzle)**.** A VDF can be formalized as a cryptographic puzzle as follows:

- $\mathsf{Setup}(1^\lambda, \Delta) \rightarrow \mathsf{pp}$: This algorithm has inputs and outputs exactly similar to as described in $\mathsf{VDF.Setup}()$.
- $\mathsf{Pre}(x', \mathsf{aux}) \rightarrow x$: Set $\mathsf{aux} = \mathsf{nil}$ and $x = x'$.
- $\mathsf{Eval}(x, \mathsf{aux}) \rightarrow y/\ \perp$: The evaluation algorithm proceeds exactly as described in $\mathsf{VDF.Eval}()$. The output y is a tuple consisting of a VDF solution s and its proof π, $y = (s, \pi)$.
- $\mathsf{Verify}(x, \mathsf{aux}, y) \rightarrow 0/1$: Outputs the result from computing $\mathsf{VDF.Verify}(\mathsf{pp}, x, s, \pi)$.

VDF security is described using a sequentiality game played between a challenger and an adversary as defined below:

Definition B.6 (Sequentiality)**.** The sequentiality game captures the notion that no adversary should be able to compute the output for Eval on a random challenge in time less than the requisite time t even with arbitrary parallelism. For an exact description of the game and more details we refer readers to Boneh et al's [34] work.

Definition B.7 (Proof of Space)**.** A proof of space is defined using the following algorithms:

- $\mathsf{Init}(N, pk) \rightarrow S$: The initialization algorithm takes as input a space parameter $N \in \mathcal{N}$ (where $\mathcal{N} \subset \mathbb{Z}^+$ is the set of valid parameters) and a public key pk for a signature scheme. It outputs $S = (S.\Lambda, S.N, S.pk)$ where S consists of the space taxing file the prover needs to store.
- $\mathsf{Prove}(S, c) \rightarrow \pi$: The prove algorithm on an input S and a challenge $c \in \{0,1\}^w$ in the challenge space, outputs a proof π.
- $\mathsf{Verify}(S, c, \pi) \rightarrow 0/1$: The verify algorithm accepts the proof and outputs 1 if it is a valid proof of space for the given input, challenge pair. Otherwise it outputs 1.

Additionally, a PoSpace must satisfy the completeness and security definitions as defined below.

Definition B.8 (PoSpace Completeness)**.** Perfect completeness implies that $\forall N \in \mathcal{N}, c \in \{0,1\}^w$,

$$\Pr[\mathsf{Verify}(S, c, \pi) = 1] = 1$$

where $S \leftarrow \mathsf{Init}(N, pk)$ and $\pi \leftarrow \mathsf{Prove}(S, c)$.

Definition B.9 (PoSpace Security)**.** Informally, proof of space security states that an adversary who stores a file of size considerably less than N bits should not be able to produce a valid proof when given a random challenge without using a significant amount of computation. For an exact description of the game and more details we refer readers to [27, 28].

Figure 7: Real world puzzle definitions under the Cryptographic Puzzle lens