Securing Proof-of-Work Ledgers via Checkpointing

Dimitris Karakostas
University of Edinburgh and IOHK
dimitris.karakostas@ed.ac.uk

Aggelos Kiayias
University of Edinburgh and IOHK
akiayias@inf.ed.ac.uk

Abstract

Distributed ledgers based on the Proof-of-Work (PoW) paradigm are typically most vulnerable when mining participation is low. During these periods an attacker can mount devastating attacks, such as double spending or censorship of transactions. Checkpointing has been proposed as a mechanism to mitigate such 51% attacks. The core idea is to employ an external set of parties that securely run an assisting service which guarantees the ledger's properties and can be relied upon at times when the invested hashing power is low. We realize the assisting service in two ways, via checkpointing and timestamping, and show that a ledger, which employs either, is secure with high probability, even in the presence of an adversarial mining majority. We put forth the first rigorous study of checkpointing as a mechanism to protect PoW ledgers from 51% attacks. Notably, our design is the first to offer both consistency and liveness guarantees, even under adversarial mining majorities. Our liveness analysis also identifies a previously undocumented attack, namely front-running, which enables Denial-of-Service against existing checkpointed ledger systems. We showcase the liveness guarantees of our mechanism by evaluating the checkpointed version of Ethereum Classic, a blockchain which recently suffered a 51% attack, and build a federated distributed checkpointing service, which provides high assurance with low performance requirements. Finally, we prove the security of our timestamping mechanism, build a fully decentralized timestamping solution, by utilizing a secure distributed ledger, and evaluate its performance on the existing Bitcoin and Ethereum systems.

1 Introduction

During the early '80s, the seminal work of Shostak, Pease, and Lamport introduced the consensus problem [29, 41]. 30 Years later, Bitcoin [34] accelerated research and development in this area by introducing an approach, frequently referred to as "Nakamoto consensus", and the blockchain data structure, which has been used in a wide range of distributed ledger systems in the past decade. Similar to classic (synchronous) consensus protocols, blockchain systems depend on an honest majority type of assumption. Contrary to them though, Bitcoin, and other Proof-of-Work (PoW) blockchain systems, assumes over 50% of hashing power backing correct protocol execution.

When the honest majority assumption is violated, an adversary can perform a wide range of attacks, that includes reverting *transaction finality*. This family of attacks is particularly devastating in this context, since it invalidates the immutability of the ledger. Specifically, finality ensures that transactions which are published on the ledger are stable after some time, i.e. cannot be reversed (unless with negligible probability). If finality is not ensured various problems arise, the most prominent being the "double spend" attack. Simply put, if the adversary can revert any transaction it wishes, then it can double spend the same assets by first issuing a payment and then reverting it, after it is presumed final by its counterparty.

Double spending attacks pose a grave threat against cryptocurrencies. Such attacks have been documented in cryptocurrency systems such as Horizen (formerly known as ZenCash) [47], Vertcoin [42], Bitcoin Gold [23], and Ethereum Classic [35]. The Ethereum Classic attack is an enlightening case of the dangers a cryptocurrency faces when the mining power drops. At its all time high, the system's mining difficulty, in other words the mining power which was protecting it, was approximately 248 Th/s[1]; 4 months later, at the time of the attack, it had dropped by more than half to 120 Th/s. Therefore, it may be stipulated that the attack was mounted exactly when the difficulty dropped sufficiently enough and was brought within the range of the adversary. In another telling case, protection against 51% attacks became prominent for *Bitcoin ABC*. ABC is an implementation of Bitcoin Cash, a top-5 cryptocurrency by market capitalization[2]. During the so-called "hash wars" [37], in fear of attacks against the system, ABC introduced a software patch which allows chain reorganizations only for the last 10 blocks [1], as a way to ensure transaction finality.

Importantly, Bitcoin ABC's solution introduces a major "network split" hazard, which is especially dangerous with a threshold as low as 10 blocks. Specifically, consider the following scenario. A node N goes offline for 5 hours, i.e. more than enough time for 10 Bitcoin Cash blocks to be created[3]. An adversary also creates 10 blocks in the span of these 5 hours[4]. When N re-joins the network, it obtains (and adopts) the adversarial chain before the honest. Now, N can never re-join the honest part of the network, since the honest chain is a fork older than 10 blocks compared to the adversarial chain, thus is rejected by N. In other words a network split occurs, where the (honest) node N joins and remains on the adversarial part.

In addition to double spending attacks, cryptocurrency ecosystems occasionally face Denial-of-Service (DoS) attacks. DoS attacks take various forms against different parts of the system, e.g. flooding of blockchain clients or attacking cryptocurrency exchanges. A more dangerous and nuanced DoS attack occurs when adversarial miners attempt to censor certain transactions. In that case, the adversarial miner never includes a certain transaction in its blocks and thus prevents the transaction from getting published, as long as no honest blocks are accepted for a sustained period of time, More critically, such attacks are nearly impossible to identify reliably. Therefore, it is highly desirable that blockchains offer strong censorship resistance guarantees; with foresight, we stress the importance of the *liveness* property in countering such DoS attacks on the blockchain level.

Evidently honest mining majority is not always a possible assumption for a PoW-based distributed ledger. To mitigate attacks in these cases, we can introduce an external set of parties, the majority of which is honest, to assist the system. The idea of external protection is rather old in the blockchain community. During the early days of Bitcoin, the need for faster client bootstrapping and protection against DoS attacks resulted in (centrally-issued) *checkpoints*. Interestingly, this mechanism[5], introduced by Bitcoin's creator Satoshi Nakamoto, was maintained until as late as 2014, with these old checkpoints remaining in Bitcoin's code until these days. However, the cryptographic literature lacks concise analyses regarding checkpointing solutions, thus often forcing developers to produce ad-hoc designs. The network split hazard in Bitcoin ABC exemplifies this problem.

Our work investigates such checkpointing mechanisms and provides a thorough analysis of how they can be implemented. We explore the caveats and security guarantees, aiming to inform practice on the formal guarantees achieved by checkpoint mechanisms w.r.t. the ledger's desired properties. Importantly, to enhance flexibility, our designs allow automatically activation, during vulnerable periods, and disabling, when the system reaches a given level of security.

Our Contributions and Roadmap. Our work provides, to the best of our knowledge, the first rigorous analysis of temporary assisting mechanisms for a Proof-of-Work blockchain that mitigate 51% attacks. Moreover our mechanism, to the best of our knowledge, is the first to provide *liveness guarantees* even under an adversarial mining majority. Our contributions are summarized as: i) a federated checkpointing mechanism which periodically commits the state of the ledger irreversibly; ii) a security analysis of checkpoints and the identification of front-running, a novel attack against all existing checkpointing implementations; iii) a prototype, signature-based checkpointing implementation which is provably secure and demonstrates high performance; iv) a decentralized timestamping-based checkpoint mechanism, which depends on

[1]The Ethereum Classic parameters are retrieved from BitInfoCharts: `https://bitinfocharts.com/comparison/difficulty-etc.html`

[2]`https://coinmarketcap.com/` [September 2019]

[3]A Bitcoin Cash block is created every 10 minutes on average.

[4]This is achievable with as low as $\frac{1}{3}$ of the total mining power.

[5]For a detailed discussion on checkpoints in the early versions of Bitcoin we refer to `https://bitcointalk.org/index.php?topic=437.msg3807`

a distributed ledger and is evaluated on Bitcoin and Ethereum.

Prior to introducing our solutions, Section 2 provides an overview of the preliminaries used in our work, including the execution model of our protocols, the threat model, as well as the distributed ledger's properties and block production mechanisms. Following that, we define a checkpointing "ideal functionality," i.e. the security definition of a ledger's chain resolution mechanism secured by checkpoints.

The security analysis of the checkpointing functionality shows that the checkpointed ledger satisfies the necessary properties, namely persistence and liveness, with high probability under adversarial majority. The liveness analysis was particularly challenging, needing to employ a Markov chain to achieve probabilistic guarantees that no prior work achieves. The analysis also highlights an attack against liveness, namely the "front-running" attack, which, to the best of our knowledge, has not been previously discussed in the context of checkpoints. The key idea that enables liveness to be preserved and mitigates front-running attacks is that our checkpointing mechanism also operates as an unpredictable "randomness beacon."

Next, we proceed with two implementations of the checkpointing functionality. The key interesting objective here is to obtain an implementation that does not trivialize the task of maintaining the ledger. Our implementation, described in Section 3.3, uses an updatable state that has (optimal) size $O(\kappa)$, κ being the security parameter. Our checkpointing service relies on an fail-stop protocol [38, 28] that enables checkpointing nodes to agree on a checkpoint and collectively compute an unpredictable nonce; Appendix B relaxes this assumption by tolerating byzantine faults via an interactive consistency [41] sub-protocol. We evaluate our scheme w.r.t. Ethereum Classic, thus demonstrating a way to integrate our solution into a PoW ledger which recently experienced double spending attacks. Finally, Section 3.4 provides a prototype implementation based on Raft [38], which achieves checkpoint issuance in the order of hundreds of milliseconds with standard commercial system requirements.

Our second implementation, *timestamping-based checkpointing*, is laid out in Section 4. The key idea is to use a second distributed ledger (presumably more reliable than the one we seek to protect) and, via a timestamping protocol we describe, facilitate the checkpointing functionality. This result demonstrates how to checkpoint a ledger with the help of another blockchain protocol, thus achieving a potentially higher level of decentralization compared to our first solution, which requires a (closed) set of dedicated servers. We also show a non-interactive (centralized or federated) timestamping mechanism (cf. Appendix E) which requires state of size only $O(\kappa)$.

2 Preliminaries

2.1 The Protocol's Execution Model

Our work is evaluated in a multiparty setting following Canetti's formulation of the "real world" [8]. Here an "environment" program $\mathcal{Z}$ drives the execution of a protocol Π by spawning an instance of an "interactive Turing machine" (ITI) which executes the protocol, instantiating a party P. The interaction between ITIs is controlled by a control program C, such that $(\mathcal{Z}, C)$ form a *system of ITMs*, cf. [8]. We restrict our setting both to "locally polynomial-bounded" systems of ITMs, thus ensuring polynomial-time execution, and to a sequential execution of parties by the environment, i.e. first activating the adversary $\mathcal{A}$ and then the parties $P_1 \dots P_n$ in order. The adversary $\mathcal{A}$ is an ITI which, upon activation, may "corrupt" a number of parties by sending a corruption message to C, after the environment has instructed it to do so. Next, whenever a corrupted party is supposed to be activated, $\mathcal{A}$ is activated instead. We also assume a *diffuse* functionality which allows the parties to broadcast messages without the need of a fully connected graph.

We assume that the ledger protocol is set in the synchronous setting. Specifically, it is executed in rounds, such that each party is activated and performs a number of operations in each round. We enforce that every message which is produced at round r is received by the other parties on round $r + 1$. Finally, we assume that the number of parties n is predefined and fixed for the duration of the execution, similar to prior work (cf. [15]). We note that, although the ledger protocol assumes a synchronous network, the checkpoint protocol may be asynchronous, as we show in Section 3.3. Future work will focus on relaxing both assumptions, i.e. exploring asynchronous networks as well as dynamic participation.

A party maintains two types of internal state, the local chain and the transaction memory pool (mempool). The local chain is chosen from a pool of chains available on the network, according to the chain decision rules. The mempool contains transactions which the party has received or created. We assume that a party removes a transaction from its mempool either if it is published on the chain or if u rounds pass since it received it. Also we impose no upper limit on the size of the blocks; this assumption is rather helpful in conjunction with the mempool strategy, since it implies that the honest miners include a transaction in the first block after the transaction is diffused on the network.

2.2 Eligibility Mechanisms for Block Production

Every distributed ledger hinges on the eligibility mechanism for updates, e.g. creating a new block and appending it to the existing chain. The core question, i.e. "which party is responsible for updating the ledger next?", has seen a number of answers, the most prominent being Proof-of-Work (PoW). PoW is based on the computational power of the parties. Each party is identified by an amount of hashing power and is elected to create a new block proportionally to it. In our model, each party can perform a number q of queries to a *random oracle*; in practice, a miner would repeatedly hash the block's payload with a pseudorandom nonce. A party is successful if the query's response, i.e. the produced hash, adheres to a protocol-defined limit, i.e. is in line with the protocol's difficulty parameter p. Here p denotes the probability that a single query to the random oracle is successful, thus the overall probability that a party produces a block at any given round is $q \cdot p$. Finally, there exist many alternatives to PoW, e.g. Proof-of-Stake and Proof-of-Space; although our work considers only PoW-based ledgers, there is no obvious barrier in applying our solutions to these mechanisms as well, as will be explored in future work.

2.3 The Ledger's Properties

In our analysis we employ the ledger's properties as defined in the Backbone model [15]. The first property, **persistence**, ensures that the honest parties converge to a single accepted chain, thus agreeing on the order of the transactions in the ledger. Therefore, if persistence holds, the probability of reverting an old, "stable" transaction, i.e. changing the ordering of stable transactions in the ledger, is negligible. The second property is **liveness**, which states that a transaction which is *valid*, i.e. is not in conflict with a stable transaction, and is produced and broadcast by an honest party for an adequate amount of time will eventually become stable. Intuitively, this property ensures that the adversary cannot censor an honest party's transaction. The concrete definitions of the two properties used to evaluate the protocols are as follows:

Definition 1 (Persistence)**.** *A transaction which is part of a block at least k blocks away from the ledger's head, i.e. a block which is part of the chain which results from removing the last k blocks of the current chain, is stable, i.e. every honest party reports it in the same position in the ledger.*

Definition 2 (Liveness)**.** *A transaction which is provided continuously as input to the parties is stable after a number u of rounds.*

Additionally, we assume that no insertions, copies, or predictions occur. These assumptions imply that the employed hash function is secure and will prove particularly useful when we try to minimize the state that the nodes maintain, in which case we rely on the assumption that the same block cannot extend two different chains.

2.4 Threat Model

A core part in every security analysis is the adversarial model. We now define the assumptions which hold for all parties and the restrictions imposed on the adversary, as well as the communication network.

The adversary $\mathcal{A}$ controls $\mu_{\mathcal{A}}$ of the network's mining power. As mentioned, in the case of PoW $\mu_{\mathcal{A}}$ corresponds to hashing power; we stress that it is possible that $\mu_{\mathcal{A}} > 0.5$, i.e. the adversary might control the majority of the mining power. Our analysis w.r.t. the mining power is conducted in terms of the number of parties, so given the total amount parties n it holds that $\mu_{\mathcal{A}} = \frac{t}{n}$. Naturally, since the adversary controls t parties, the amount of honest parties is $n - t$. Additionally, the adversary is "adaptive", i.e. corrupts parties on the fly, and "rushing", i.e. at each round retrieves all honest parties' messages before deciding its strategy.

$\mathcal{A}$ tries to break either persistence or liveness (or both). Regarding the former, $\mathcal{A}$ tries to force two honest nodes to accept different chains as stable, i.e. to report different transactions as stable in the same position in their respective ledger; Theorem 1 shows that checkpoints ensure that such attacks are impossible. Regarding liveness, $\mathcal{A}$ attempts to prevent a transaction from becoming stable within u rounds. However, such censorship attacks are notoriously hard to detect. With hindsight, we note that a transaction becomes stable when it is reported in the checkpointed chain. Thus we ensure that *regardless of the adversarial strategy* at least one honest block, which by the assumptions of Section 2.1 includes the transaction, is checkpointed within u rounds.

3 The Checkpointed Ledger

Our first assisted scheme is the *checkpointed* ledger. Checkpoints are messages issued by a service and help the parties converge to a single chain, even in the presence of an adversary who controls a mining majority. In the following paragraphs we provide the necessary definitions and describe the checkpointing functionality. We focus on making the decision rules generic enough in order to accommodate any PoW blockchain, rather than being constrained to specific implementations. Finally, after analyzing the security of the checkpointed ledger, we realize it using standard cryptographic primitives. We note that, although outside of the scope of this paper, our mechanism can also help increase performance by minimizing the ledger's state and enabling clients to bootstrap faster.

3.1 The Checkpointing Functionality

As motivated in the introduction, our goal is to define a ledger which is resistant to attacks from an adversarial mining majority. In this section we achieve this via the checkpointing functionality $\mathcal{F}_{\mathsf{Checkpoint}}$. $\mathcal{F}_{\mathsf{Checkpoint}}$ achieves this by establishing *checkpoints*, i.e. irreversible chains. $\mathcal{F}_{\mathsf{Checkpoint}}$ maintains two chains: i) C, which is its local chain, and ii) C_c, which is the latest checkpoint. Upon retrieving a new candidate from the network, it decides whether to adopt it as its local chain by running $\mathsf{maxvalid}(\cdot, \cdot)$, i.e. the chain decision rule algorithm (cf. [15]); the inner workings of this algorithm depend on the blockchain, e.g. Bitcoin uses the *heaviest* chain, i.e. the chain with most hashing power. When $\mathcal{F}_{\mathsf{Checkpoint}}$ adopts a chain which is k_c blocks longer than its local checkpoint, it issues a new checkpoint. Following, any chain which is not an extension of the checkpoint is automatically rejected. Observe that, even if a fork occurs right before a checkpoint issuing, the functionality checkpoints only one block, thus all parties will converge to the canonical chain identified by this checkpoint.

k_c identifies the rate of checkpoint production. For instance, larger k_c results in sparse checkpoints, but also less stress on the network communications. On the other hand, smaller k_c allows the parties to synchronize faster and restricts the adversary's control over the chain's blocks, as shown in the security evaluation of Section 3.2. Additionally, checkpoints organize the protocol's execution in *epochs*, each beginning with the issuing of a checkpoint and consisting of a specific number of blocks. Finally, since the network is asynchronous, $\mathcal{A}$ can choose to discard candidate blocks and has full control of the scheduling of messages.

Every checkpoint is identified by an unpredictable nonce r. In order to model fail-stop faults, which will be needed for the protocol of Section 3.3, the functionality produces a list of nonces equal to the number of parties and allows $\mathcal{A}$ to discard a minority of them, before picking a random nonce from the remaining. We note that, if the adversary discards a majority of the messages (equivalently if a majority of checkpointing nodes crash), then the checkpointing operation halts. The unpredictable nonce is paramount to the security of the system. As we explore further in Section 3.2.2, if r is predictable or is not included in the chain, then an adversary with a majority of the mining power can break the liveness property by mounting a "front-running" attack and controlling the blocks of the chain in perpetuity.

Figure 1 defines the checkpointing ideal functionality $\mathcal{F}_{\mathsf{Checkpoint}}$, parameterized by $\mathsf{maxvalid}$ and k_c. Here $|C|$ denotes the length of a chain (in blocks), $|\mathbb{V}|$ the number of parties in $\mathbb{V}$, $||$ the concatenation of blocks, chains of blocks, and strings, $\prec$ the prefix operation, e.g. if $C = C'|| \cdots$ then $C' \prec C$, $\setminus$ the difference of two chains, e.g. if $C = C'||B|| \cdots$ then $C \setminus C' = B|| \cdots$, and $\mathit{tail}(C)$ the last block of a chain C.

Functionality $\mathcal{F}_{\mathsf{Checkpoint}}$

$\mathcal{F}_{\mathsf{Checkpoint}}$ interacts with a set of parties $\mathbb{V}$ and holds the local chain C and the checkpoint chain C_c, both initially set to ϵ. It is parameterized by k_c, which defines the number of blocks between two consecutive checkpoints, and the $\mathsf{maxvalid}(\cdot, \cdot)$ algorithm.

Upon receiving $(\textsc{CandidateCheckpoint}, C')$ from a party $\mathcal{V}$, forward it to $\mathcal{A}$. Upon receiving $(\textsc{CandidateCheckpoint}, C')$ from $\mathcal{A}$, if $C_c \prec C'$ set $C := \mathsf{maxvalid}(C, C')$. Next, if $|C \setminus C_c| = k_c$ compute a list R of $|\mathbb{V}|$ random values as $r_j \xleftarrow{\$} \{0,1\}^\omega$ and send $(\textsc{Nonce}, R)$ to $\mathcal{A}$. Upon receiving from $\mathcal{A}$ a response $(\textsc{Nonce}, R')$, such that R' is a list of at least $\frac{|\mathbb{V}|}{2}$ values from R, pick a value $r_i \in R'$, return $(\textsc{Checkpoint}, \mathit{tail}(C)||r_i)$ to $\mathcal{V}$ and set $C := C_c := C||r_i$.

Figure 1: The checkpointing ideal functionality.

3.2 Security of the Checkpointed Ledger

We now focus on the security of the functionality $\mathcal{F}_{\mathsf{Checkpoint}}$ and show that the checkpointed ledger satisfies *persistence* and *liveness* w.r.t. the following parameters:

- k: the persistence parameter, i.e. the number of blocks after which a transaction is stable;
- u: the liveness parameter, i.e. the amount of time that a transaction needs to be continuously provided to all parties before it becomes stable;
- k_c: the checkpointing interval, i.e. the epoch's length;
- q: the number of queries to the hashing oracle that a party can make during a single round;
- p: the block difficulty, i.e. the probability that a single query is successful in producing a block;
- n: the number of parties;
- t: the number of adversarial parties.

3.2.1 Persistence

First, we show that our scheme satisfies persistence, an expected result given that it is a direct outcome of the nature of checkpoints. Theorem 1 formally proves this intuition; $C^{\lceil k}$ denotes the chain which is output by removing the k last blocks from C.

Theorem 1 (Persistence)**.** *The checkpointed chain resolution protocol of Section 3.1 satisfies persistence (cf. Definition 1) for parameter $k \geq k_c$.*

Proof. It suffices to show that, at any round r, for two honest parties $\mathcal{V}_1, \mathcal{V}_2$ with chains C_1, C_2 respectively, where $|C_1| \leq |C_2|$, it holds that $C_1^{\lceil k} \prec C_2^{\lceil k}$. In that case, a transaction in $C_1^{\lceil k}$ is also reported by $\mathcal{V}_2$ in the same position, since $C_1^{\lceil k}$ is a prefix of its own chain. We observe that, if $k \geq k_c$, at least one of the last k blocks in both chains C_1, C_2 is a checkpoint. Assume that this checkpoint is the l_1-th block from the head of C_1 and the l_2-th from the head of C_2; by definition of the chain decision rule, $C_1^{\lceil l_1} = C_2^{\lceil l_2}$. □

As we see next, to provide adequate liveness guarantees, the parameter k_c typically needs to be small. Therefore, persistence is achieved for relatively low values of k, while our scheme ensures that blocks, and consequently transactions, are finalized after a short amount of time.

3.2.2 Liveness

We now focus on liveness in the checkpointed setting. The first step of our analysis is to determine the necessary conditions such that liveness holds. Theorem 2 shows that, as long as at least a single honestly-generated block is checkpointed, liveness holds regardless of the adversarial strategy.

Theorem 2. *For any execution of a checkpointed chain resolution protocol which securely realizes $\mathcal{F}_{\mathsf{Checkpoint}}$ of Section 3.1, a transaction τ is stable if at least one honestly-generated block, which is mined after the creation of τ, is part of the checkpointed chain after u rounds since τ is diffused on the network.*

Proof. Assume a block B which is honestly produced *after* τ is diffused on the network and extends a chain C. By definition of the model of Section 2.4, τ is part either of C or B. Next, assume that B is part of the checkpointed chain. By definition of the checkpointing functionality $\mathcal{F}_{\mathsf{Checkpoint}}$, the miners reject any chain which does not extend the checkpointed chain, i.e. which does not include B. Therefore, regardless of the adversarial strategy, after this point τ is necessarily in the checkpointed chain, i.e. is stable. □

As discussed in the introduction, liveness is pivotal in countering DoS attacks. Proving that the checkpointed ledger satisfies liveness against *every* adversary was significantly more challenging compared to the persistency analysis. A major difficulty here is that, if an adversary controls a mining majority, it has an inherent advantage over the honest parties regarding the chain's growth rate and which transactions are included in the ledger. Constraining this advantage is central in our analysis, as illustrated by the front-running attack of Figure 2. This attack is similar to Selfish Mining [14, 44]. An adversary that employs selfish mining can gain an advantage by mining in a "private" setting, i.e. withholding newly-mined blocks until it becomes necessary to publish them, e.g. until a competing block is observed. In our scheme, the unpredictable random value r mitigates this attack by preventing $\mathcal{A}$ from retaining an advantage and "refreshing" the execution with the issuing of every checkpoint, thus ensuring liveness is guaranteed with adequate probability. To the best of our knowledge, this attack has not been previously discussed or taken into consideration in existing checkpointing mechanisms, which thus fail to provide *any* liveness guarantees in the presence of adversarial mining majorities. Figure 3 also provides intuition on the front-running attack vector.

The Front-Running Attack

Assume an adversary $\mathcal{A}$ which controls $\mu_{\mathcal{A}}$ of the mining power. If $\mu_{\mathcal{A}} > 0.5$, i.e. if $\mathcal{A}$ controls a mining majority, $\mathcal{A}$ produces (on average) more blocks than the honest parties. Assume the following adversarial strategy: $\mathcal{A}$ extends a private chain, while not adopting any honest blocks, and, for every produced block which extends the longest honest chain, $\mathcal{A}$ reveals a block, while keeping all other adversarial blocks hidden. Since $\mu_{\mathcal{A}} > 0.5$, the adversary will eventually have produced more blocks than the honest parties, so it will be able to counter every honest block. Since $\mathcal{A}$ is rushing, the adversarial blocks are always adopted over the conflicting honest blocks. Thus, eventually all blocks in the chain are adversarial.

Figure 2: The front-running attack against liveness.

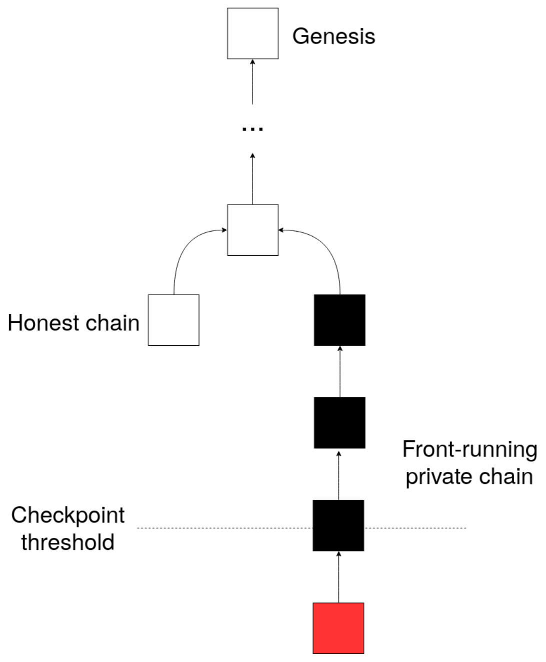

Figure 3: The front-running attack, where the adversary produces a private chain which spans multiple epochs. Upon publishing this chain, the adversary can checkpoint all black blocks. However, this chain also contains a red block, which extends the soon-to-be checkpointed chain prior to the issuing of the future checkpoint, allowing the adversary to build an advantage into the next epoch. Mitigating front-running would render the red block (and each subsequent block) invalid and restrict the advantage of the adversary.

In the remaining of this section we will analyze the conditions under which an adversary can break liveness. The tool we will use is an *absorbing Markov chain*, parameterized by k_c, where each state identifies the progress of the parties in producing enough blocks to reach a checkpoint. We then prove that reaching the absorption state translates into checkpointing an honest block, hence we can argue about the minimum liveness probability of the checkpointed protocol (using the chain's stochastic transition matrix, cf. Appendix A).

Each state of the Markov chain is identified by (i, j). i denotes the number of blocks that *an honest party needs to produce* in order to reach the next checkpoint, assuming the honest parties advance without any adversarial interference[6]. j is the number of blocks the adversary *necessarily* needs to produce in order to reach the next checkpoint without adopting any honest non-checkpointed blocks, thus restricting a transaction from getting checkpointed. In the first round of an epoch, all parties need to produce exactly k_c blocks to reach the next checkpoint, so each epoch starts on state (k_c, k_c).

The absorbing state is the state that compounds all pairs of the form $(0, j)$ with $j > 0$. Other pairs of the form (i, j) with $i > 0$ are transitional. Transitions represent the accumulation of honest and adversarial blocks, as per the execution model. Note that if honest parties simultaneously produce a block in a single round it will be counted once, therefore, the only allowed transitions from state (i, j) are towards states $(i - \alpha, j - b)$ with $\alpha \in \{0, 1\}, b \in [0, j]$. Before exploring the transition probabilities between any two states, we first provide some useful definitions. Similar to Bitcoin Backbone [15], we define the following random variables:

- H: if *at least one* honest party produces a block at a given round, then $H = 1$, else $H = 0$;
- $M^{(i)}$: if *all adversarial parties* produce exactly i blocks at a given round, then $M^{(i)} = 1$, else $M^{(i)} = 0$;

for which the following hold:

- $\mathbb{E}(H) = h = 1 - (1 - p)^{q \cdot (n-t)}$;
- $\mathbb{E}(M^{(i)}) = m^{(i)} = \binom{q \cdot t}{i} \cdot p^i \cdot (1 - p)^{q \cdot t - i}$ for any i.

Lemma 1 defines the transition probabilities from a state (i, j) to a state $(i - a, j - b), a \in \{0, 1\}, b \in [0, j - 1]$. Next, Lemma 2 explores the special cases, where either i or j of a state (i, j) is equal to 0. We denote $\hat{m}_l = \sum_{\phi=0}^{l} m^{(\phi)}$.

Lemma 1 (Transition Probabilities). *For any execution, the following hold for transitions from round (i, j) with $i, j > 0$:*

- *transition to $(i, j - b)$ occurs with probability $\bar{h} \cdot m^{(b)}$;*
- *transition to $(i - 1, j - b)$ occurs with probability $h \cdot m^{(b)}$;*

for every $b \in [0, j - 1]$ and $\bar{h} = 1 - h$.

Proof. The proof is straightforward by observing that, at state (i, j), the first coordinate is reduced by 1 if and only if at least one honest party exists that computes a block in a round, while the second coordinate is reduced by b if and only if all adversarial parties produce exactly b blocks. □

Lemma 2. *The following hold:*

i) the state $(0, 0)$ is equivalent to the state (k_c, k_c);

ii) transition from round $(1, j)$, where $j > 0$, to the absorbing state occurs with probability $h \cdot \hat{m}_{j-1}$;

iii) from round (i, j), where $i, j > 0$, the following hold:

- *transition to $(i, 0)$ occurs with probability $\bar{h} \cdot (1 - \hat{m}_{j-1})$;*

[6] Adversarial interference refers to censorship of queries to the hashing oracle or messages exchanged between the parties.

- *transition to $(i-1,0)$ occurs with probability $\bar{h} \cdot (1-\hat{m}_{j-1})$;*

iv) from round $(i,0)$, where $i > 0$, the following hold:

- *transition to $(i-1,0)$ occurs with probability h;*
- *transition to $(i,0)$ occurs with probability $1-h$.*

Proof. i) If both the honest parties and the adversary produce enough blocks, then, since the adversary is rushing, it can control the message delivery to the functionality and checkpoint its chain. So the round $(0,0)$ corresponds to checkpointing an adversarial chain and the beginning of a new epoch, i.e. the state (k_c, k_c).

ii) If the honest parties produce a block and the adversary produces strictly less than j blocks, then the execution reaches state $(0,l)$ with $l > 0$, i.e. the absorption state.

iii) If at round (i,j) the adversary produces at least j blocks, the reached state is $(i,0)$ if the honest parties don't produce a block (resp. $(i-1,0)$ if they do).

iv) When $j = 0$, the adversary has already produced enough blocks to reach the checkpoint. So, the only possible transition (i.e. to state $(i-1,0)$) depends on the honest block production probability h.

□

Algorithm 1 collects the above rules for the construction of the Markov chain. In order to produce the graph we run createGraph(k_c, k_c); this function is also parameterized by addEdge, which creates a new edge in the Markov chain given the source node, destination node, and transition probability. The connection of the Markov chain to liveness is established in Theorem 3.

Theorem 3 (Liveness)**.** *The Markov chain defined in Algorithm 1 has the property that, whenever it reaches the absorption state, an honest block is guaranteed to be checkpointed in the corresponding execution with error probability $L \cdot 2^{-\omega}$, L being the protocol's execution total length.*

Proof. The proof relies on two observations. We recall that the absorption state is defined as $(0,j)$ for $j > 1$, i.e. in the current execution, and since the last checkpoint, the honest parties have produced enough blocks to reach the next checkpoint, while the adversary has produced at least one block less. The first observation is that, whenever an honest block is produced in a round, the chain of all honest parties is guaranteed to advance irrespectively of the adversarial strategy; as a result, when the absorption state is reached, one honest party possesses a chain sufficiently long to be checkpointed. Next, we would like to show that such chain will have at least one honest block. We can derive this from the second observation, i.e. the fact that each checkpoint introduces unpredictable randomness $r \xleftarrow{\$} \{0,1\}^{\omega}$. Thus any adversarial blocks produced prior to the calculation of the last checkpoint cannot contribute to the chain that an honest party possesses (unless the adversary correctly guesses the random nonce r of the checkpoint, prior to its introduction, an event which is conveyed in the error term of the theorem). It follows that, by definition, the absorption state puts the adversary at a position where it lacks a sufficient number of blocks to match the blocks in an honest party's chain and thus at least one honest block will be checkpointed. □

Using the Markov chain of Algorithm 1, we can produce its stochastic transition matrix, in order to compute the liveness w.r.t. u, as well as the expected number of rounds before being absorbed. We also assume a sufficiently large value of ω, such that the probability of error (cf. Theorem 3) is negligible.

Liveness Evaluation of a Checkpointed Ledger. To evaluate our mechanism we took a snapshot of Ethereum Classic[7]. In order to realize our model, we assume that each hash

[7]All data regarding Ethereum Classic were provided by `https://bitinfocharts.com` [6 February 2019].

Algorithm 1 The absorbing Markov chain construction algorithm, defined by createMarkovChain and parameterized with k_c and the recursive helper function createGraph.

```
function createMarkovChain($k_c$)
    createGraph($k_c, k_c$)
    addEdge($\text{final}, \text{final}, 1$)
end function
function createGraph($i, j$)
    if $j > 0$ then
        for $l \in [0, j-1]$ do
            addEdge($(i,j), (i, j-l), \bar{h} \cdot m^{(l)}$)
            if $l > 0$ then
                createGraph($i, j-l$)
            end if
            if $i > 1$ then
                addEdge($(i,j), (i-1, j-l), h \cdot m^{(l)}$)
                createGraph($i-1, j-l$)
            end if
        end for
        addEdge($(i,j), (i,0), \bar{h} \cdot (1 - \hat{m}_{j-1})$)
        createGraph($i, 0$)
        if $i = 1$ then
            addEdge($(i,j), \text{final}, h \cdot \hat{m}_{j-1}$)
            addEdge($(i,j), (k_c, k_c), h \cdot (1 - \hat{m}_{j-1})$)
        else
            addEdge($(i,j), (i-1, 0), h \cdot (1 - \hat{m}_{j-1})$)
            createGraph($i-1, 0$)
        end if
    else
        addEdge($(i,j), (i,j), \bar{h}$)
        if $i = 1$ then
            addEdge($(i,j), (k_c, k_c), h$)
        else
            addEdge($(i,j), (i-1, j), h$)
            createGraph($i-1, j$)
        end if
    end if
end function
```

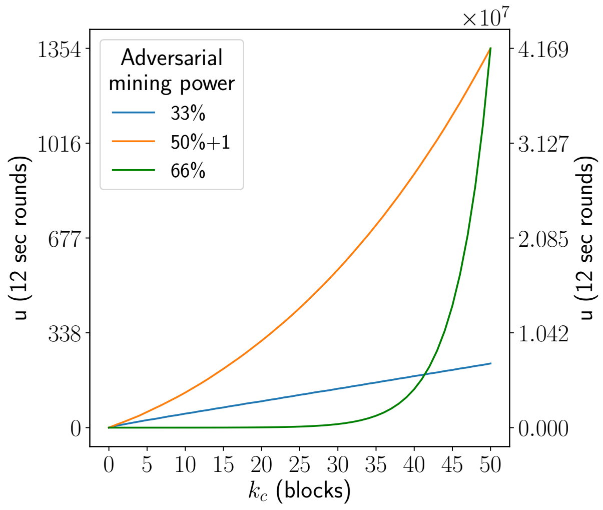

Figure 4: The expected number of steps before absorption for the checkpointed Ethereum Classic w.r.t. k_c, i.e. the expected u before liveness is achieved with probability at least $\frac{2}{3}$. The primary (left) axis identifies the liveness parameter u for 33% and $50\% + 1$ adversarial control, while the secondary axis corresponds to 66% adversarial power. Our model is parameterized with $q = 12 \cdot 237 \cdot 10^6, n = 33755, p = 9.1 \cdot 10^{-15}$, and 12-second rounds.

corresponds to a query and that each party performs 237 MH/s.[8] Furthermore, the total hash rate is on average 8 TH/s. Therefore, the total number of parties is $n = \frac{8 \cdot 10^{12}}{237 \cdot 10^6} = 33755$. We use 12 seconds as our round's length and also have $q = 12 \cdot 237 \cdot 10^6$. The final parameter is the difficulty of the network. The difficulty implies the threshold to which the hashes should comply in order to be accepted, specifically the required number of most-significant bits of the hash being equal to 0. In our case, the difficulty is 110 TH, i.e. 1 out of every $110 \cdot 10^{12}$ hashes is successful on average, so the probability that a single hash is successful is $p = 9.1 \cdot 10^{-15}$.

Figure 4 depicts the expected number of steps before absorption w.r.t. k_c, for Ethereum Classic's parameters above. This metric provides an estimation of the number of rounds, i.e. the value of u, needed to achieve liveness with probability at least $\frac{2}{3}$. As expected, the number of steps increases with the adversarial power. u increases linearly with k_c, as long as the adversary controls a minority of the mining power. However, if the adversary controls a mining majority, the expected number of steps increases exponentially with k_c, to the point where a 66% adversary is orders of magnitude more powerful (and we need a different axis to make the figure intelligible).

Figure 5 shows the liveness probability w.r.t. u for various values of k_c. Naturally, the liveness probability depends on the initial state of the graph, i.e. the state of the system when the transaction is published for the first time. Therefore, the minimum liveness probability can be extracted as the minimum probability over all possible transient states; our simulations have shown that, as expected, this state is $(k_c, 0)$, i.e. when the adversary has the biggest advantage.

In order to evaluate the liveness probability, we fix the adversarial mining power to $50\% + 1$. We observe that liveness is achieved with high probability after a relatively small amount of rounds for $k_c = 1$; specifically after 50 rounds, i.e. after 10 minutes, the liveness probability is 0.9975. However, the liveness probability decreases significantly as the epoch length increases; for instance, again for 10 minutes, when $k_c = 5$ the liveness probability is 0.5836, whereas when $k_c = 10$ it drops significantly to 0.1434. This behavior is expected since, as $k_c \to \infty$ the system downgrades to the standard non-checkpointed setting, where an adversary with majority can break liveness with probability 1.

[8]This corresponds to the popular mining hardware "PandaMiner B1 Plus", thus we assume each party is realized by a single such machine.

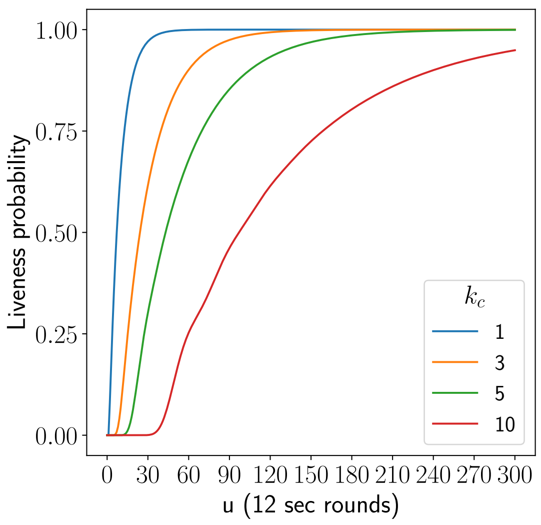

Figure 5: The liveness probability of a checkpointed Ethereum Classic w.r.t. u and k_c. The vertical axis identifies the liveness probability, while the horizontal axis defines u, i.e. the number of consecutive rounds for which the transaction is supplied to the miners. Our model is parameterized with rounds which last 12 seconds, $q = 12 \cdot 237 \cdot 10^6, n = 33755$, and $p = 9.1 \cdot 10^{-15}$.The adversarial mining power is $50\% + 1$ so $t = 16877$.

3.3 The Checkpointed Chain Resolution Protocol

In this section we realize the checkpointing authority as a federated service distributed among parties communicating over an asynchronous network. To make our design most efficient we assume that the checkpointing service runs among a set of parties that trust each other, while tolerating benign faults e.g. crashes, message reordering, *etc.* Although not fully decentralized, like the timestamping solution of Section 4, our scheme is in line with similar real-world checkpointing mechanisms, e.g. Bitcoin, Peercoin, and Feathercoin, where checkpoints were issued by the software's developers. In our prototype implementation of Section 3.4 we consider a scenario where 5 coordinating organizations, such as development companies and/or community foundations directly linked to the ledger's ecosystem, employ 3 nodes each (for redundancy purposes). In Appendix B we relax this trust assumption by defining a protocol that tolerates Byzantine Faults, although at the cost of using an expensive interactive consistency protocol over a synchronous network.

The checkpointing protocol is parameterized by a number of subroutines. First, it is parameterized by a validation predicate Validate. This predicate identifies whether a chain is valid, e.g. verifies the signatures and the Proof-of-Work of the chain's blocks, similar to the maxvalid function of the Bitcoin Backbone. Second, the parties coordinate via a fail-stop subprotocol π_{FS}, like RAFT [38] or Paxos [28]. This protocol enables the parties to both reach agreement on which block to checkpoint and the value of the unpredictable nonce r. Each party $\mathcal{V}_j$ inputs $\langle B_j, r_j \rangle$, where B_j is a (valid) block and $r_j \xleftarrow{\$} \{0,1\}^\omega$ is a random nonce. At the end of π_{FS}, each party outputs $\langle B', r' \rangle$, i.e. one of the input block and nonce. The checkpointing protocol is defined in Figure 6, where $C[i]$ denotes the i-th block of chain C. Theorem 4 formally shows that $\pi_{\mathsf{Checkpoint}}$ securely realizes $\mathcal{F}_{\mathsf{Checkpoint}}$, as long as the fail-stop protocol completes, i.e. a majority of parties is live. We note that the theorem restricts to environments that, when corrupting a party, they may force it to fail, rather than behave arbitrarily.

Theorem 4. *Protocol $\pi_{\mathsf{Checkpoint}}$ securely realizes $\mathcal{F}_{\mathsf{Checkpoint}}$ if a majority of parties in $\mathbb{V}$ is available, such that π_{FS} terminates successfully.*

Proof. Since we assume only crash faults, each party either follows the protocol or is unresponsive. Firstly, the blocks which are proposed for checkpointing are valid, since the protocol

Protocol $\pi_{\mathsf{Checkpoint}}$

A checkpointing party which runs $\pi_{\mathsf{Checkpoint}}$ is parameterized by the list $\mathbb{V}$ of n checkpointing parties, a (fail-stop) consensus protocol π_{FS}, a validation predicate $\mathsf{Validate}$, the function $\mathsf{maxvalid}$, and k_c. It keeps a local checkpointed block, B_c, initially set to ϵ.

Upon receiving $(\textsc{CandidateCheckpoint}, C')$ from a party $\mathcal{V}$, check:

- $\exists i : C'[i] = B_c$ (i.e. *if* C' *extends the checkpoint*);
- $\mathsf{Validate}(C') = 1$ (i.e. *if* C' *is valid*);
- $|C'| - i = k_c$ (i.e. *if* C' *is long enough*).

If all hold do:

1. pick $r_j \xleftarrow{\$} \{0,1\}^{\omega}$;
2. pick input $\langle C', r_j \rangle$ for the protocol π_{FS};
3. execute π_{FS} with the parties in $\mathbb{V}$ to agree on an input $\langle C', r' \rangle$, such that $\forall \langle \hat{C}, \hat{r} \rangle \in \mathbb{I} : \mathsf{maxvalid}(C', \hat{C}) = C'$ with $\mathbb{I}$ the set of inputs, i.e. choose the output according to $\mathsf{maxvalid}$;
4. set $B_c := \mathit{tail}(C') || r'$.

Finally, return $(\textsc{Checkpoint}, B_c)$ to $\mathcal{V}$.

Figure 6: The protocol run by the parties of the checkpointing authority.

employs the validation predicate $\mathsf{Validate}$ before providing them as input to π_{FS}. Secondly, the value r_j, which is picked by each party at random, is unpredictable and thus indistinguishable from the value r_i chosen by $\mathcal{F}_{\mathsf{Checkpoint}}$. Therefore, the inputs to the protocol π_{FS} are well-structured, i.e. indistinguishable from the inputs in $\mathcal{F}_{\mathsf{Checkpoint}}$. Finally, since a majority of parties is available, π_{FS} is guaranteed to output a checkpoint, which will be chosen according to $\mathsf{maxvalid}$ as in $\mathcal{F}_{\mathsf{Checkpoint}}$. We note that, in the ideal world, the simulator $\mathcal{A}$ can control the delivery of messages to $\mathcal{F}_{\mathsf{Checkpoint}}$, such that the first valid candidate block which is proposed is also the heaviest (according to $\mathsf{maxvalid}$). □

Naturally, if the availability guarantee fails, i.e. if a majority of parties is unavailable, then π_{FS} will stop. In that case, checkpoint consistency is guaranteed, i.e. no conflicting checkpoints will be issued, but also no new checkpoints are produced. In turn, the ledger downgrades to the plain execution model, so persistence is guaranteed only for the chain up to the last checkpoint, whereas liveness is no longer guaranteed, if the adversary controls a mining majority.

To incorporate checkpoints in the consensus protocol run by miners, we slightly adapt Bitcoin Backbone for the checkpointed setting. Specifically, instead of using $\mathsf{maxvalid}$ directly for chain resolution, a miner now utilizes the checkpointing mechanism. Next, we define the protocol which is run by the miners and realizes the chain resolution functionality. When a miner creates a new block, they submit it to all checkpointing parties via the $\mathsf{CandidateCheckpoint}$ interface of $\pi_{\mathsf{Checkpoint}}$. When the new checkpoint is issued, they accept it and, following, they adopt a new chain only if it contains a newly-issued checkpoint. The chain resolution protocol in the checkpointed setting is defined in Figure 7; in addition to previous notation, $C[: i]$ denotes the chain consisting of the first i blocks of C.

$\pi_{\mathsf{Checkpoint}}$ checkpoints *valid* chains, i.e. validates both the block's headers and transactions before selecting a chain. We can relax this assumption by requiring $\pi_{\mathsf{Checkpoint}}$ to validate only the block's headers. This change would reduce the computation requirements of $\pi_{\mathsf{Checkpoint}}$, but also allow invalid transactions in the chain; specifically, a block with invalid transactions may have valid headers, i.e. extend the hash chain per the PoW rules, thus a party which does

Protocol $\pi_{\mathsf{CheckpointMiningRes}}$

A party which runs $\pi_{\mathsf{CheckpointMiningRes}}$ is parameterized by $\mathsf{maxvalid}$, the n checkpointing parties $\mathbb{V}$ which run $\pi_{\mathsf{Checkpoint}}$, and k_c. It keeps a local chain C and the checkpoint index i_c, initially set to ϵ and 0.

Upon receiving $(\textsc{CandidateChain}, C')$, set $C := \mathsf{maxvalid}(C, C')$. If $|C| \geq i_c + k_c$ set $i_c := i_c + k_c$ and send $C[:i_c]$ to all parties in $\mathbb{V}$. Upon receiving $\lceil \frac{n}{2} \rceil$ messages $(\textsc{Checkpoint}, B||r)$ from different checkpointing parties, if $C[i_c] = B||r$ set $C := C[:i_c]||r$.

Upon receiving $(\textsc{Read})$ return $(\textsc{Chain}, C)$.

Figure 7: The checkpointed mining chain resolution protocol.

not validate each transaction would accept it. In this case, the ledger's consensus mechanism should be adapted to accept only the first of the potential conflicting transactions, rather than rejecting the chain which contains invalid transactions altogether, as is the case in current blockchain systems.

As motivated in the introduction, we aim to secure a blockchain only temporarily. Therefore, once the ledger can securely exist without assistance, the shut down of the checkpointing service is initiated. Shutdown is parameterized by a security threshold, such that, after it is reached, the checkpoint authority halts and the ledger transitions to the decentralized mode. This security threshold is outside of the scope of this work, though potential candidates include the network's *hash rate* or the *profitability* of attacks. The system though is at a risk if the service is compromised at a future point, as an adversary that compromises the service after the shut down could issue new checkpoints. This risk can be mitigated in two ways: i) if the security threshold is publicly computable, then the miners know whether it has been reached and ignore future checkpoints, ii) the authority produces a specific "shut down" message to alert the miners of the operation halting. Future work will explore additional mechanisms for future-proofing the checkpoint authority.

3.4 Prototype Implementation

We implemented a prototype of the checkpointing service and evaluated its performance. We assume a PKI for the checkpointing nodes which, after agreeing which block to checkpoint, produce and publish signatures. Given that the honestly-produced signatures are unpredictable, since otherwise an attacker could produce forged signatures, the necessary unpredictable nonce consists of the aggregated checkpointing signatures. Therefore, the miners can now verify in a non-interactive manner whether the checkpointing service has issued a checkpoint on a given block.

Our experiments ran on a private Ethereum network set up on Amazon's EC2 platform with `t2.micro` instances[9] running Ubuntu 18.04 LTS. The network consisted of 3 mining nodes, which coordinated via a "bootnode" node. Our network replaced the costly Proof-of-Work mechanism of the vanilla Ethereum with Parity's Proof-of-Authority (PoA) [13], which uses the *Clique* algorithm. All nodes were launched within the same geographical region (EU) and produced blocks on a 10 second interval (as opposed to the 15 second interval of the real-world Ethereum mainnet).

The checkpointing federation, consisting of 15 nodes, was built utilizing a number of existing tools. First, each node ran a full Ethereum client which connected to the private network and retrieved the newly-mined blocks from the mining nodes. Second, in order to coordinate the checkpointing nodes we used `etcd`[10], a distributed file system. `etcd` employs Raft [38] in order to resolve conflicts, such that all nodes maintain the same file contents. In our setting, we used `etcd` such that the federation nodes agree on which block to checkpoint and also to exchange

[9] `t2.micro` instances use 1 virtual CPU and 1 GB of memory.
[10] `https://etcd.io/`

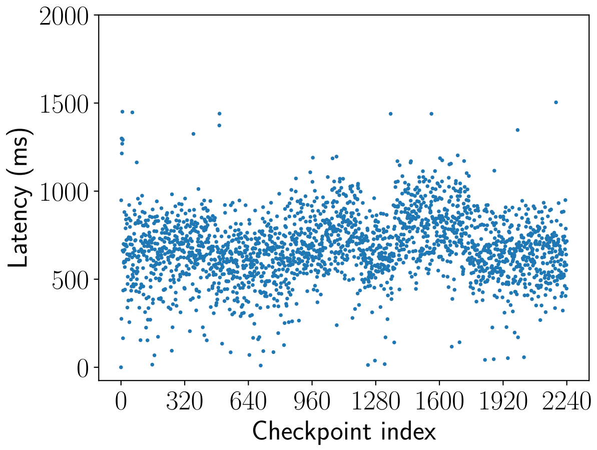

Figure 8: Evaluation of latency, i.e. the time between retrieving a block and accepting it as a checkpoint, for the prototype checkpointing implementation. Each graph point corresponds to an independent checkpoint, over a period of more than 24 hours.

and store signatures on newly-issued checkpoints. Third, each node was identified by a public key — we assume that the public keys are well-known, e.g. are part of the genesis block of the system. In order to generate the keys and the signatures we used the JavaScript cryptographic library `TweetNaCl`[11].

In order to checkpoint a block, a node connected to an Ethereum client and observed the blocks, until at least k_c blocks had been mined on top of the latest checkpoint (in the bootstrapping case since the genesis block); our implementation defined $k_c := 4$. When such block was observed, the node signed its hash and then stored on `etcd` both the hash and the signature. Therefore, a valid checkpoint consisted of at least 8 (i.e. a majority of) federation signatures on a block's hash. In case the federation nodes did not agree on which block to checkpoint, i.e. produced signatures on conflicting blocks such that no block was supported by a majority, the nodes dismissed the checkpoint and proceeded to checkpoint the next candidate block (after k_c blocks).

Our simulations focus on the following metrics: i) the network latency, i.e. the time between the transmission of a block and its acceptance as a checkpoint; ii) the storage overhead of checkpoints in the ledger.

Regarding (i), we deployed a checkpointing client outside of Amazon's service and connected it both to the Ethereum private network and the checkpoint federation. To estimate latency we measured the elapsed time between retrieving a block eligible for checkpointing and obtaining a majority of valid federation signatures for the block's hash. Our simulation lasted more than 24 hours, spanning over 2200 checkpoints. Figure 8 depicts our results, which are rather positive. Specifically, latency was on average 679 ms — occasionally, a checkpoint would need more time to be published, although no more than 1.5 seconds. Additionally, we observed cases when the federation nodes would not produce a checkpoint, with this failure rate being approximately 15%. Each fail resulted in a 40 second delay, i.e. until a following checkpoint was issued. However, we expect production-grade implementations to minimize such fails.

However, latency depends on the location of the node to which the client connected, as longer distance results in worse latency. Specifically, for a client residing in the United Kingdom latency was as follows, when connecting to checkpointing nodes in different geographical regions:

- London (EU): 557 ms
- N. California (US West): 620 ms
- São Paulo (South America): 711 ms

[11]`https://tweetnacl.js.org`

- Tokyo (Asia Pacific): 723 ms
- Singapore (Asia Pacific): 779 ms

Regarding (ii), a checkpoint consists of the concatenated signatures of federation nodes. Since each `TweetNaCl` signature consists of 64 bytes, each checkpoint amounts to $8 \cdot 64 = 512$ bytes, thus checkpointing results in a 0.6% increase in the ledger's size. [12] We expect production-grade implementations to offer better results, e.g. utilizing multi-signature schemes, like the ASM scheme of [4], to reduce the checkpoint's size.

4 The Timestamped Ledger

Our second scheme, timestamping, is motivated by the need to fully decentralize the checkpoint mechanism. As we show, timestamping allows us to relax our assumptions, while still achieving the same guarantees as above. Following, we first model timestamping as an ideal functionality and then realize it as an interactive decentralized service built on top of an existing distributed ledger.

4.1 The Timestamping Functionality

First, we define the *global* timestamping functionality in Figure 9. $\mathcal{F}_{\mathsf{Timestamp}}$ issues timestamps by keeping a monotonically increasing counter and a list of timestamped strings. It allows a party to timestamp a string s by submitting the message $(\textsc{Timestamp}, s)$; afterwards, every party can verify it via the Verify interface. We stress that the timestamping functionality is *global*, i.e. timestamps are not issued privately. Therefore, when a party timestamps a string, *every* other party can access both the string and its timestamp. We also note that the timestamp consists of both a counter and a random value. The latter helps mitigate the front-running attack (cf. Figure 3). In the decentralized implementation of Section 4.4, where the timestamping functionality is realized as a distributed ledger, the random value will be the hash of the block which timestamps a given string.

Functionality $\mathcal{F}_{\mathsf{Timestamp}}$

$\mathcal{F}_{\mathsf{Timestamp}}$ holds the following items:

- $T_{[]}$: an initially empty list of timestamped strings;
- τ: a counter initially set to 0;

Upon receiving $(\textsc{Timestamp}, s)$, if $\forall (s', \cdot) \in T_{[]} : s' \neq s$, set $\tau := \tau + 1$. Then compute a list R of $p(\kappa)$ random values as $r_j \overset{\$}{\leftarrow} \{0,1\}^{\omega}$ and send $(\textsc{Nonce}, R)$ to $\mathcal{A}$. Upon receiving a response $(\textsc{Nonce}, r_i)$, such that $r_i \in R$, add (s, τ, r_i) to $T_{[]}$.

Upon receiving $(\textsc{Verify}, s, \tau)$, if $\exists (s, \tau) \in T_{[]}$ then return $(\textsc{VerifyTimestamp}, \top)$.

Figure 9: The timestamping ideal functionality.

4.2 The Timestamped Chain Resolution Protocol

Using $\mathcal{F}_{\mathsf{Timestamp}}$ we can now construct the timestamped ledger. Similar to Section 3, we define the timestamped mining protocol, which leverages $\mathcal{F}_{\mathsf{Timestamp}}$ and picks a chain among the set of all possible candidates. A miner can timestamp a new block by submitting it to $\mathcal{F}_{\mathsf{Timestamp}}$;

[12]The size of each Ethereum block is on average 20 KB (cf. `https://etherscan.io/chart/blocksize`).

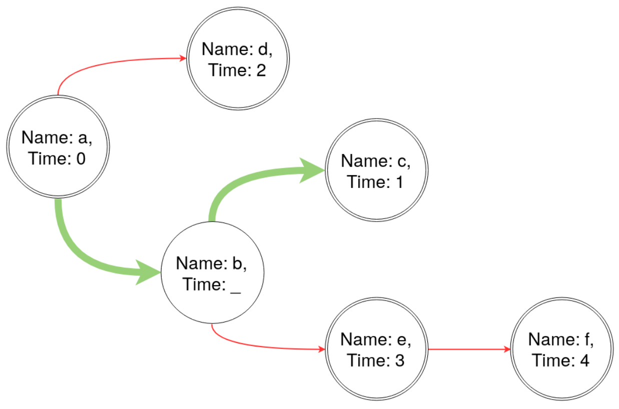

Figure 10: A graph of (potentially) timestamped blocks. A chain is chosen by traversing the graph, starting with a which has children b and d. Since b is not timestamped, the tree of b is traversed until timestamped blocks are encountered. So the decision is among $a||d$, $0||b||c$, and $a||b||e||f$; since block c has an older timestamp than d and e, $a||b||c$ is chosen. However, if block b is later checkpointed, the canonical chain will become $a||d$, which is the reason why timestamping as soon as possible is paramount.

a timestamped block is the tuple $B_t = (B, \tau)$, where B is the block created by the miner and τ is the timestamp issued by $\mathcal{F}_{\mathsf{Timestamp}}$; $B.\tau$ denotes the timestamp of the block B.

When a miner is given a new candidate chain, they compare it with their local chain. Starting from the genesis block, it parses both chains until it finds the *timestamped* position where the two diverge, i.e. the oldest timestamped block in each chain which does not exist in the other. If such point exists, then we adopt the chain with the oldest diverging block. Otherwise, i.e. if the last checkpointed block in both chains is the same, we employ the maxvalid algorithm. Finally, between timestamped and non-timestamped blocks, the former are preferred. Figure 10 provides intuition for the timestamped chain decision rules, showcasing a basic timestamped block graph, with the timestamped mining chain resolution protocol $\pi_{\mathsf{TimeMiningRes}}$ defined in Figure 11.

4.3 Security of the Timestamped Ledger

We analyze timestamping in Theorem 5, observing that it is equivalent to checkpoints with $k_c = 1$.

Theorem 5 (Timestamping). *The timestamped resolution protocol $\pi_{\mathsf{TimeMiningRes}}$ and the timestamping functionality $\mathcal{F}_{\mathsf{Timestamp}}$ of Section 4.1 guarantee persistence and liveness with parameter $k_c = 1$ (cf. Theorems 1 and 3).*

Proof. Assume a round where all parties hold the same chain C. It suffices to show that the first block which extends C (and gets timestamped) acts as a checkpoint. Assume the first such block B_1 extends C and is assigned a timestamp t_1. Any subsequent block B_i which extends C is assigned a timestamp t_i strictly larger than t_1, by definition of the timestamping mechanism. Thus, an honest miner will always adopt the chain $C||B_1$ over $C||B_i$. Regardless of the adversarial strategy, if the honest parties produce a block first, its timestamp is irreversible and older than any subsequent adversarial block. Also since the adversary cannot censor timestamp requests, honest blocks always get timestamped. Therefore, the timestamp produced for the first block that extends a chain acts as a checkpoint. Finally, the randomness r which mitigates front-running is constructed in a similar manner as $\mathcal{F}_{\mathsf{Checkpoint}}$, thus the adversary cannot produce multiple future blocks unless it can predict r, which is possible with probability $L \cdot 2^{-\omega}$, L being the protocol's execution length. □

There are two important caveats that need to be stressed. First, following the protocol and quickly timestamping new blocks is crucial for security. Given that timestamped blocks

Protocol $\pi_{\mathsf{TimeMiningRes}}$

A party which employs $\pi_{\mathsf{TimeMiningRes}}$ holds the local chain C, initially set to ϵ, and is parameterized by the $\mathsf{maxvalid}(\cdot,\cdot)$ algorithm.

Upon receiving a message $(\textsc{CandidateChain}, C')$, for every timestamped block $B \in C'$, send $(\textsc{Verify}, B, B.\tau)$ to $\mathcal{F}_{\mathsf{Timestamp}}$ and wait for $(\textsc{VerifyTimestamp}, \top)$. Next, do:

```
i) set $i := 0$;
ii) while $C[i] = C'[i]$ do $i := i + 1$;
iii) set $i' := i, c := i - 1$;
iv) while $C[i]$ is not timestamped and $i < |C|$ do $i := i + 1$;
v) while $C'[i']$ is not timestamped and $i' < |C'|$ do $i' := i' + 1$;
vi) if $i = |C|$ and $i' = |C'|$ set $C := \mathsf{maxvalid}(C \setminus C[: c], C' \setminus C'[: c])$,
vii) else if $i = |C|$ or $C'[i'].\tau < C[i].\tau$ then set $C := C'$.
```

Upon receiving $(\textsc{Read})$ return $(\textsc{Chain}, C)$.

Figure 11: The timestamped mining protocol for chain resolution.

are prioritized over non-timestamped, an adversary can discard an arbitrarily long, honestly-created, but non-timestamped chain. Additionally, the protocol which realizes $\mathcal{F}_{\mathsf{Timestamp}}$ needs prevent the adversary from producing forged timestamps. As we show next, we can create such decentralized mechanism via a secure distributed ledger. Second, it is necessary to timestamp the entire block, so that the honest parties can immediately access it. For example, timestamping only a hash does not suffice, as an adversary could timestamp the hash and keep the block secret. Following, either the honest miners would halt until the adversary reveals the block, i.e. resulting in a DoS, or they would extend a block with a newer checkpoint, in which case the adversary could drop the honest chain by simply revealing its own block, i.e. breaking persistence and liveness. Alternatively, we can timestamp only the block header and properly handle invalid transactions (cf. Section 3.3).

4.4 Decentralized Implementation

We now implement fully decentralized timestamping using a distributed ledger $\mathcal{L}$; Appendix E also provides a centralized solution similar to checkpoints.To timestamp a string d, a user submits a transaction τ to $\mathcal{L}$ which contains d. $\mathcal{L}$ is an append-only ledger, so a unique, monotonically increasing index can be assigned to every transaction in $\mathcal{L}$, thus providing a total ordering of transactions. We observe that, as long as $\mathcal{L}$ satisfies persistence, the ordering of stable transactions is irreversible. We can thus use this index as the timestamp of τ and, consequently, d. If $\mathcal{L}$ also satisfies liveness, then it is infeasible for the adversary to censor an honest party's request to timestamp their data. Finally, the hash of the block in $\mathcal{L}$ which timestamps d is used as the unpredictable nonce that mitigates front-running.

Here d is the *headers* of a block. Specifically, a miner publishes a transaction to $\mathcal{L}$ which contains the headers of a newly-mined block. Following, the block's timestamp can be retrieved by parsing $\mathcal{L}$ and identifying the index of the timestamping transaction in the ledger. It is evident that the length of the required public state is $O(n \cdot |d|)$, corresponding to the timestamped blocks' headers.

Naturally, the adversary may attempt to gain an advantage given the rate of timestamping of the assisting ledger. For example, it can use excessive fees to incentivize $\mathcal{L}$'s miners to prioritize its transaction over an honest transaction or attempt to perform a DoS attack using "spam transactions" [45]. Additionally, migrating to a non-timestamped setting is no longer as straightforward as with the federated checkpoints. For instance, if the honest parties simply stop timestamping their blocks, the adversary can continue to do so and thus revert any honestly-generated chain. Therefore, the transition to the non-timestamped setting necessitates a hard fork, i.e. hard-coding the final timestamped block, such that the honest parties ignore all subsequent timestamps.

Evaluation of Decentralized Timestamping. We implement the timestamping service on two major blockchain systems, Bitcoin and Ethereum. The results of our constructions are demonstrated in Table 1. Our Ethereum implementation is a contract which receives the header of a block to be timestamped and emits an event.[13]. $\mathsf{BTC}^\star$ denotes a ledger with Bitcoin-like block headers and $\mathsf{ETH}^\star$ a ledger with Ethereum-like headers; a Bitcoin header is 80 bytes[14], whereas an Ethereum header is about 600 bytes [46]. The timestamping cost is identified as the gas cost per operation and is evaluated in USD[15]. Deploying our contract costs 176569 gas (0.4 USD), whereas timestamping a $\mathsf{BTC}^\star$ and an $\mathsf{ETH}^\star$ header cost 30302 gas (0.07 USD) and 71137 gas (0.16 USD) respectively.

Cost	Ethereum	Smart contract deployment	\$0.4
		$\mathsf{BTC}^\star$ header timestamping	\$0.07
		$\mathsf{ETH}^\star$ header timestamping	\$0.16
	Bitcoin	$\mathsf{BTC}^\star$ header timestamping	\$0.45
		$\mathsf{ETH}^\star$ header timestamping	\$3.6
Latency	Ethereum	Stable timestamp	9 minutes
		Unstable timestamp	15 seconds
	Bitcoin	Stable timestamp	60 minutes
		Unstable timestamp	10 minutes
Proof size	Ethereum	Full node	181 GB
		SPV implementation	5 GB
		NIPoPoW implementation	6 MB
		FlyClient implementation	3 MB
	Bitcoin	Full node	240 GB
		SPV implementation	48 GB

Table 1: Decentralized timestamping performance, using Ethereum and Bitcoin to timestamp ledgers with Bitcoin-like ($\mathsf{BTC}^\star$) or Ethereum-like ($\mathsf{ETH}^\star$) headers.

Latency is expectedly worse in the decentralized case. Ethereum assumes (on average) 9 minutes until a transaction is stable, corresponding to 35 confirmations[16], i.e. new blocks mined "on top" of the transaction. However, users could accept unstable timestamps and mine immediately after they are issued, thus reducing latency to 15 seconds; in the off chance an unstable timestamp is reverted, they would update their chain and start afresh.

We also evaluate the storage verification requirements of timestamping. Typically the miner acquires a full copy of the ledger $\mathcal{L}$, parses it, and verifies the timestamps. However, running a

[13]The contract is deployed on the Ropsten testnet: `https://ropsten.etherscan.io/address/0xf95c1b1caefe5f2b5050844f64cac906f15a78f1`

[14]`https://bitcoin.org/en/glossary/block-header`

[15]$1\text{gas} = 1.3135247596 \cdot 10^{-8}$ ETH, 1ETH = 172 USD (cf. `https://etherscan.io/chart/gasprice` [September 2019])

[16]The reference numbers of confirmations used by the Coinbase exchange: `https://support.coinbase.com/customer/portal/articles/593836`

full node is an intensive task, e.g. the Ethereum chain is 181 GB[17], thus storing and parsing it assumes significant hardware requirements. Alternatively, we can utilize the Simplified Payment Verification (SPV) mode [34]. SPV clients retrieve only the blocks' headers, instead of the entire chain. Since the timestamp of a string is the index of its transaction in the ledger, in order to verify that the timestamp of a string d is τ, the miner verifies the chain's headers in a standard SPV fashion and additionally obtains the full block which contains the transaction of interest. As of September 2019, the data which an SPV Ethereum client needs to retrieve and parse amount to about 5 GB. Finally, we can utilize super-light client modes such as NIPoPoW [24] and FlyClient [5]. Super-light clients employ succinct proofs of synchronization, thus allowing a client to verify the timestamp of a block using a proof of $O(polylog(n))$ size on the chain's length n. A super-light Ethereum-based timestamping client retrieves about 6 MB and 3 MB for NIPoPoW and FlyClient respectively (cf. [5]).

Alternatively, we can use special transactions to coordinate timestamping. Every blockchain which allows arbitrary data inclusion in the transaction's payload can be used instead of Ethereum, although Ethereum's smart contracts do provide a helpful interface in tracking the timestamps. Such blockchain is Bitcoin, which allows via the `OP_RETURN` opcode [3] to timestamp arbitrary data up to 80 bytes[18]. Therefore, using the Bitcoin blockchain, timestamping would cost 0.45$ for $\mathsf{BTC}^\star$ headers and 3.6$ for $\mathsf{ETH}^\star$ headers (using multiple transactions). Finally, latency in Bitcoin is 10 minutes for unstable timestamps, i.e. the average block production time, and 60 minutes for stable, whereas the size for a full node is 240 GB and an SPV node is 48 GB[19].

5 Related work

Checkpointing precedes blockchains as a method of stabilizing a consensus protocol. In their seminal paper on Practical Byzantine Fault Tolerance [9], Castro and Liskov describe such mechanism in a replicated setting to bring up to date a replica "left behind". In the blockchain space, checkpoints are often used against network attacks and to enhance performance. Both Bitcoin and Bitcoin Cash introduced checkpoints at different stages of development, in order to speed up the bootstrapping of network nodes and mitigate DoS and 51% attacks. Bitcoin's checkpoints were issued in an entirely centralized manner, a method also employed by Peercoin [27] and Feathercoin [18]. Our work proposes a similar checkpointing solution, albeit federated rather than fully centralized, as well as a fully decentralized mechanism in the form of timestamping. In contrast, Bitcoin Cash ABC restricted chain re-orderings to a maximum depth of 10 blocks, while, similarly, Nxt [36] applies a maximum re-ordering depth of 720 blocks. However, these solutions introduce the major "network split" hazard described in Section 1. In comparison, our checkpointing mechanism formalizes these previous attempts, thus preventing hazards like the network split and enabling us to provide rigorous security proofs regarding persistence and liveness. Finally, RSK [30] proposes a checkpoint system similar to Bitcoin, which is constructed as a federation. Notwithstanding, all mechanisms fail to counter the front-running attack of Section 3.2.2, since the checkpoints exist outside of the chain; instead, our design provides probabilistic liveness guarantees even in the presence of an adversarial mining majority.

The blockchain academic literature has primarily considered checkpoints in the context of preventing long-range attacks in Proof-of-Stake (PoS), rather than protecting Proof-of-Work (PoW) systems. PoS protocols replace mining power with "stake", i.e. the subset of the coins that the block producer owns. Although such systems replace the costly mining operations with a more environmentally friendly mechanism, they also enable an adversary to produce blocks at effectively no-cost, which in turn results in a number of threats against PoS protocols, such as the nothing-at-stake [12], long range [6], and stake bleeding [16] attacks. Checkpoints are utilized to prevent such attacks in protocols like Ouroboros [25], Snow White [2], and Ouroboros Praos [10], in the latter also being used as a mechanism to mitigate adaptive corruptions. However, these

[17] `https://etherscan.io/chartsync/chaindefault` [September 2019]

[18] Although Bitcoin's consensus rules do not impose such limit, the 80 byte threshold is a relay standard, thus most miners enforce it. The Bitcoin dust fee for `OP_RETURN` transactions is 546 satoshis (0.05$) and the median transaction fee is 0.4$ (`https://bitinfocharts.com/comparison/bitcoin-median_transaction_fee.html` [September 2019]).

[19] `https://www.blockchain.com/charts/blocks-size` [September 2019]

works lack both our probabilistic analysis and, most importantly, provide no liveness guarantees against front-running attacks.

Checkpoints have also been used to improve *finality*, i.e. to reduce the time until a transaction is deemed stable. Casper the Friendly Finality Gadget [7] defines a fine-grained checkpointing mechanism, which is applied in conjunction with a PoW blockchain, allowing to both protect against block reversions and (financially) penalizing misbehaving parties. Afgjort [32] describes a generic finality layer, which is run by a sub-committee and can be applied on top of an arbitrary blockchain. However, both systems assume a secure underlying blockchain, i.e. a blockchain which already satisfies persistence and liveness, and aim at a performance boost in finality. In comparison, our analysis allows an adversary to hold a mining majority and aims at securing a temporarily insecure ledger.

The idea of employing sub-selection to run an agreement protocol that commits transactions irreversibly has also been explored in various settings. On the PoW side, notable works are hybrid consensus [39], which integrates a permissioned protocol with a decentralized blockchain to elect rotating committees, and Thunderella [40], which optimistically confirms transactions via an asynchronous consensus protocol that tolerates $\frac{1}{4}$ faults of an elected committee while using PoW as a fallback mechanism. On the PoS side, Algorand [17] uses a combination of a Verifiable Random Function to elect a committee which runs a Byzantine Agreement (BA) protocol to produce blocks. However, these systems assume a secure ledger, on top of which the BA protocol is run; in contrast, our mechanisms take effect in insecure environments, i.e. when an adversary controls more than 50% of the mining power. Additionally, our mechanism can both be seamlessly integrated in existing blockchains and also easily and automatically removed when the system matures.

An alternative design, Stellar [33, 31], operates based on quorum slices, i.e. subsets of users, within which specific validator nodes are selected. Validators assign trust relationships among each other and agreement is guaranteed between mutually trusted validators. Stellar tolerates *at best* 33% Byzantine Faults while, as shown by Kim *et al.* [26], the Stellar network is effectively centralized around a few validators operated by the same foundation. If a Stellar-like system is vulnerable, we could employ checkpointing by establishing a small number of quorums, each operated by a trusted entities to issue checkpoints as above. Honest nodes could join them, effectively forming a centralized setting as the one identified in [26], although transparent and strictly confined to issuing checkpoints.

Regarding our second scheme, secure timestamping is an old topic of interest in the cryptographic community. In a representative piece of literature, Haber and Stornetta [21] propose timestamping based on hashes and digital signatures; we utilize similar ideas in constructing a centralized and non-interactive timestamping solution in Appendix E. Blockchains increased interest in secure timestamping with distributed ledgers acting as timestamping services. For instance, Gipp et al. [19] explored trusted timestamping using the Bitcoin blockchain, while projects such as OriginStamp [22] aim to allow users to timestamp arbitrary data using Bitcoin's blockchain. Additionally, Veriblock [43] employs a mechanism, dubbed "Proof-of-Proof", which leverages Bitcoin's blockchain to secure its own chain. However, this system assumes an elaborate mechanism, while relying entirely on Bitcoin's security. In comparison, the scheme of Section 4.4 serves only as a temporary solution, while ensuring that transition to the non-timestamped setting is achieved securely.

6 Conclusion

This paper investigates securing distributed ledgers against attacks by an adversarial mining majority. The core idea is to introduce an external set of parties to guarantee the ledger's security. Motivated by the increasing rate of mining attacks and the threat they pose against the early stages of all systems, we provide a rigorous treatment of two mechanisms, checkpointing and timestamping, which ensure a good level of protection and performance under standard security assumptions. Our analysis highlights a novel attack against liveness, front-running, and our solutions are the first to achieve probabilistic liveness guarantees under adversarial majority. Our timestamping solution achieves a high level of decentralization by relying on an existing, secure distributed ledger.

Our work poses a number of questions that require further research. On the theoretical part, the ledger can be analyzed in the asynchronous setting under dynamic participation. In practice, we can also explore automatic shutdown for the checkpointing service, e.g. from a game theoretical perspective, while also future-proofing it. Regarding timestamping, future research can focus on thoroughly evaluating the implications of using a ledger for timestamping, e.g. tackling incentive-based and network attacks. Additionally, further analysis is needed regarding latency in decentralized timestamping vis-à-vis the probability of reversal of an unstable timestamp. Finally, we can explore performance enhancements, e.g. reducing the checkpoint length and improving client bootstrapping, as well as the applicability of checkpoints on non-PoW systems, e.g. PoS systems like Algorand and Ouroboros or complex designs like Stellar.

References

[1] Bitcoin ABC. Bitcoin abc 0.18.5 released, 2018. `https://www.bitcoinabc.org/2018-11-20-bitcoin-abc-0-18-5/`.

[2] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016.

[3] Bitcoin. Op_return, 2019. `https://en.bitcoin.it/wiki/OP_RETURN`.

[4] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. pages 435–464, 2018.

[5] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Super-light clients for cryptocurrencies. *IACR Cryptology ePrint Archive*, 2019:226, 2019.

[6] Vitalik Buterin. On stake. *Ethereum Blog*, 5, 2014.

[7] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[8] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/2000/067`.

[9] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[10] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

[11] Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Nikos Chondros, and Mema Roussopoulos. Interactive consistency in practical, mostly-asynchronous systems. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 752–759. IEEE, 2015.

[12] Ethereum. Proof of stake faqs, 2018. `https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs`.

[13] Parity Ethereum. Proof-of-authority chains, 2019. `https://wiki.parity.io/Proof-of-Authority-Chains`.

[14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[16] Peter Gaži, Aggelos Kiayias, and Alexander Russell. Stake-bleeding attacks on proof-of-stake blockchains. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 85–92. IEEE, 2018.

[17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[18] David Gilson. Feathercoin secures its block chain with advanced checkpointing, 2013. `https://www.coindesk.com/feathercoin-secures-block-chain-advanced-check-pointing`.

[19] Bela Gipp, Norman Meuschke, and André Gernandt. Decentralized trusted timestamping using the crypto currency bitcoin. *arXiv preprint arXiv:1502.04015*, 2015.

[20] Charles M Grinstead and J Laurie Snell. Markov chains. *Introduction to probability*, pages 405–470, 1997.

[21] Stuart Haber and W Scott Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[22] Thomas Hepp, Patrick Wortner, Alexander Schönhals, and Bela Gipp. Securing physical assets on the blockchain: Linking a novel object identification concept with distributed ledgers. In *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, pages 60–65. ACM, 2018.

[23] C. Edward Kelso. Bitcoin gold hacked for $18 million, 2018. `https://news.bitcoin.com/bitcoin-gold-hacked-for-18-million/`.

[24] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. Cryptology ePrint Archive, Report 2017/963, 2017. `http://eprint.iacr.org/2017/963`.

[25] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[26] Minjeong Kim, Yujin Kwon, and Yongdae Kim. Is stellar as secure as you think? *arXiv preprint arXiv:1904.13302*, 2019.

[27] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[28] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[29] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[30] Sergio Demian Lerner. Rsk white paper overview, 2015. `https://docs.rsk.co/RSK_White_Paper-Overview.pdf`.

[31] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96. ACM, 2019.

[32] Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Afgjort: A partially synchronous finality layer for blockchains. Cryptology ePrint Archive, Report 2019/504, 2019. https://eprint.iacr.org/2019/504.

[33] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, page 32, 2015.

[34] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[35] Mark Nesbitt. Deep chain reorganization detected on ethereum classic (etc), 2019. https://blog.coinbase.com/ethereum-classic-etc-is-currently-being-51-attacked-33be13ce32de.

[36] Nxt. Nxt whitepaper, 2014. https://nxtwiki.org/wiki/Whitepaper:Nxt.

[37] Stephen O'Neal. Bitcoin cash hard fork battle: Who is winning the hash war, 2018. https://cointelegraph.com/news/bitcoin-cash-hard-fork-battle-who-is-winning-the-hash-war.

[38] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[39] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. Cryptology ePrint Archive, Report 2016/917, 2016. http://eprint.iacr.org/2016/917.

[40] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. Cryptology ePrint Archive, Report 2017/913, 2017. http://eprint.iacr.org/2017/913.

[41] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[42] Jamie Redman. Vertcoin network suffers 300-block reorg following 51% attacks, 2018. https://news.bitcoin.com/vertcoin-network-51-attacked-and-suffers-from-a-reorg-300-blocks-deep/.

[43] Maxwell Sanchez and Justin Fisher. Veriblock whitepaper, 2018. https://www.veriblock.org/wp-content/uploads/2018/03/PoP-White-Paper.pdf.

[44] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 515–532. Springer, 2016.

[45] Bitcoin Wiki. Spam transactions, 2019. https://en.bitcoin.it/wiki/Spam_transactions.

[46] Gavin Wood. Ethereum yellow paper. *Internet: https://github.com/ethereum/yellowpaper,[Oct. 30, 2018]*, 2014.

[47] ZenCash. Zencash statement on double spend attack, 2018. https://blog.zencash.com/zencash-statement-on-double-spend-attack/.

A Mathematical Background

In this section we cover the basic mathematical tools used for our security analysis, namely the properties of the absorbing Markov chains cf. [20].

The Markov chain. A Markov chain is identified by a set of *states* $S = \{s_1, s_2, \dots\}$. An execution starts at one of the states in S and progresses in steps, each corresponding to a *transition* from a state s_i to a (different or the same) state s_j. Each transition is identified by a *probability* p_{ij}, which is independent of the history of the execution, but only depends on the

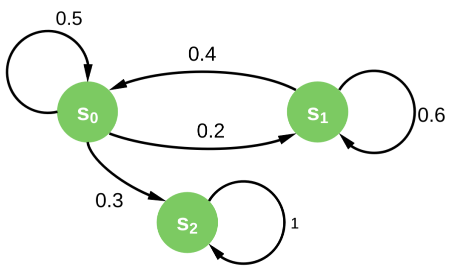

Figure 12: The example Markov chain.

	s_0	s_1	s_2
s_0	0.5	0.2	0.3
s_1	0.4	0.6	0
s_2	0	0	1

Table 2: Example transition matrix.

current state s_i. Figure 12 depicts an example Markov chain with 3 states, which we will use to provide intuition in the following paragraphs.

The absorbing state. A state s_i is *absorbing* if for the transition probabilities it holds $p_{ii} = 1$ and $p_{ij} = 0, i \neq j$; in other words, if the execution reaches an absorbing state it will never transition to a different state after. Every state which is not absorbing is *transient.*

The transition matrix. The stochastic transition matrix of a Markov chain is a matrix which comprises of the transition probabilities between any two states of the Markov chain. Specifically, it is a $n \times n$ square matrix M, where n is the number of states of the Markov chain, such that the entry $M_{ij} = p_{ij}$; in other words, the ij-th entry in M contains the probability of transition from state s_i to s_j. The *canonical form* of the transition matrix M is:

$$M = \begin{pmatrix} Q & R \\ \mathbb{O} & I_r \end{pmatrix}$$

where Q is the $t \times t$ matrix, where each column corresponds to one of the t transient states, and R is the $r \times t$ matrix, where each column corresponds to one of the r absorption states; $\mathbb{O}$ is the $r \times t$ zero matrix and I_r is the $r \times r$ identity matrix.

Table 2 depicts the canonical form of the transition matrix of an example Markov chain with 3 states, where s_2 is the absorption state and s_0, s_1 are transient states.

Absorption probability after u rounds. Assume a transition matrix M of an (absorbing) Markov chain. The ij-th entry of matrix M^u identifies the probability that starting from state s_i the execution is at state s_j after exactly u steps.

Table 3 depicts the canonical form of the transition matrix of the above Markov chain after 5 steps. Therefore, if the execution starts from the state s_0, the probability of absorption in state s_2 after 5 steps is 0.68661.

The fundamental matrix and expected number of steps until absorption. For an absorbing Markov chain with transition matrix M as above, it holds $(I - Q)^{-1} = N = I + Q + Q^2 + \cdots$; the matrix N is called the *fundamental matrix* for M. The ij-th entry in N denotes the expected number of times that the execution is in state s_j having started from state s_i.

Given the fundamental matrix N as above, the expected number t of steps before absorption is $t = \lceil \sum_{j=0}^{t} N_{ij} \rceil$, when the execution starts from the state s_i.

Table 4 depicts the fundamental matrix of the example Markov chain. The expected number of steps until absorption from the initial state s_0 is 5.

	s_0	s_1	s_2
s_0	0.17061	0.14278	0.68661
s_1	0.28556	0.242	0.47244
s_2	0	0	1

Table 3: Example transition matrix after 5 steps.

	s_0	s_1
s_0	3.333	1.667
s_1	3.333	4.167

Table 4: Example fundamental matrix.

B A Checkpointed Protocol That Tolerates Byzantine Faults

In this section we relax the trust assumption between the parties that realize the checkpointing authority. In Section 3.3 we assumed that a party may only fail by crushing. In order to allow arbitrary behavior, i.e. Byzantine Faults, instead of a fail-stop protocol we now employ an interactive consistency subprotocol π_{IC}, such as the schemes of [11]. This protocol enables the parties to both reach agreement on which block to checkpoint and also collectively produce the unpredictable nonce r.

Now, we need to slightly modify the ideal functionality $\mathcal{F}_{\mathsf{Checkpoint}}$ to express the byzantine behavior. In $\mathcal{F}_{\mathsf{CheckpointBFT}}$ of Figure 13, the adversary has more power by choosing among a polynomial number of potential random values r_j. This change models the ability to produce a (polynomial-bounded) number of random values to pick from and participate in the interactive consistency protocol.

Functionality $\mathcal{F}_{\mathsf{CheckpointBFT}}$

$\mathcal{F}_{\mathsf{Checkpoint}}$ interacts with a set of parties $\mathbb{V}$ and holds the local chain C and the checkpoint chain C_c, both initially set to ϵ. It is parameterized by k_c, which defines the number of blocks between two consecutive checkpoints, and the $\mathsf{maxvalid}(\cdot,\cdot)$ algorithm.

Upon receiving $(\textsc{CandidateCheckpoint}, C')$ from a party $\mathcal{V}$, if $C_c \prec C'$ set $C := \mathsf{maxvalid}(C, C')$. Next, if $|C \setminus C_c| = k_c$ compute a list R of $p(\kappa)$ random values as $r_j \xleftarrow{\$} \{0,1\}^\omega$ and send $(\textsc{Nonce}, R)$ to $\mathcal{A}$. Upon receiving from $\mathcal{A}$ a response $(\textsc{Nonce}, r_i)$, such that $r_i \in R$, return $(\textsc{Checkpoint}, \mathit{tail}(C)||r_i)$ to $\mathcal{V}$ and set $C := C_c := C||r_i$.

Figure 13: The checkpointing ideal functionality that tolerates byzantine faults.

As before, the checkpointing protocol is also parameterized by a validation predicate $\mathsf{Validate}$, which identifies whether a chain is valid. Each party $\mathcal{V}_j$ inputs $\langle B_j, r_j\rangle$, where B_j is the block which it wishes to checkpoint and r_j is a random nonce. At the end of π_{IC}, each party outputs an ordered list $[\langle B_1, r_1\rangle, \ldots, \langle B_n, r_n\rangle]$, which contains the inputs of all parties; in case a party aborts a default value $\langle\perp,\perp\rangle$ is chosen as its input. Following, the parties pick the block that has plurality among the blocks that are output, breaking ties in lexicographical order. Additionally, they produce the collective nonce as $r = \mathsf{H}(r_1|| \ldots ||r_j)$, where $\mathsf{H} : \{0,1\}^\star \to \{0,1\}^\omega$ is a hash function.

The checkpointing protocol is defined in Figure 14 and Theorem 6 shows that $\pi_{\mathsf{CheckpointBFT}}$ securely realizes $\mathcal{F}_{\mathsf{CheckpointBFT}}$.

Protocol $\pi_{\mathsf{CheckpointBFT}}$

A checkpointing party which runs $\pi_{\mathsf{Checkpoint}}$ is parameterized by the list $\mathbb{V}$ of n checkpointing parties, an interactive consistency protocol π_{IC}, a hash function H, a validation predicate $\mathsf{Validate}$, and k_c. It keeps a local checkpointed block, B_c, initially set to ϵ.

Upon receiving $(\textsc{CandidateCheckpoint}, C')$ from a party $\mathcal{V}$, check:

- $\exists i : C'[i] = B_c$ (i.e. *if* C' *extends the checkpoint*);
- $\mathsf{Validate}(C') = 1$ (i.e. *if* C' *is valid*);
- $|C'| - i = k_c$ (i.e. *if* C' *is long enough*).

If all hold do:

1. pick $r_j \xleftarrow{\$} \{0,1\}^{\omega}$;
2. execute protocol π_{IC} with the parties in $\mathbb{V}$ with input $\langle tail(C'), r_j\rangle$ and wait for its output $[\langle B_1, r_1\rangle, \ldots, \langle B_n, r_n\rangle]$;
3. find the block B_j which has plurality among the output blocks (breaking ties lexicographically) and set $B_c := B_j || \mathsf{H}(r_1 || \ldots || r_n)$.

Finally, return $(\textsc{Checkpoint}, B_c)$ to $\mathcal{V}$.

Figure 14: The protocol which is run by the parties of the checkpointing authority.

Theorem 6. *Protocol $\pi_{\mathsf{CheckpointBFT}}$ securely realizes $\mathcal{F}_{\mathsf{CheckpointBFT}}$, assuming a secure interactive consistency protocol π_{IC}, which successfully terminates, and a hash function H.*

Proof sketch. π_{IC} is an interactive consistency protocol, so the honest parties agree on the same checkpoint block and produce the same nonce r. Since at least one honest party contributes to the output of the protocol and, since H is secure, r is pseudorandom and unpredictable, like r of $\mathcal{F}_{\mathsf{CheckpointBFT}}$. □

C Liveness for Epochs of Random Length

As shown above, longer epochs act in favor of the adversary. In order to sidestep this advantage, we will try instead to hide the epoch's length. The core idea here is, if the adversary can no longer plan when to publish its chain, it is in its best interest to publish it immediately, although allowing the honest parties to catch up; the following example showcases this argument.

Consider the case when the honest parties have produced only a single block and the adversary has produced 2 blocks. Since the adversary does not know the epoch's length, by withholding its chain it risks the possibility that the honest parties produce a second block and, if $k_c = 2$, reach the checkpoint. Additionally, observe that now the epoch's length becomes known only *after* the checkpoint has been issued, when it is too late for the adversary. Therefore, the only way for the adversary to be completely sure that it beats the honest parties to the checkpoint is to directly publish any new block it produces.

In order to apply this idea to our model we make a small change in the checkpointing functionality of Section 3.1. Now the functionality, defined in Figure 15, is parameterized by an upper bound $k_c^{\top}$, which is known to the adversary. Additionally, it holds an internal variable k_c *unknown to the adversary*, which is drawn from $(0, k_c^{\top}]$ uniformly at random and is updated when a checkpoint is issued.

In order to evaluate liveness we again consider the adversarial strategy and the Markov chain which results from it. As mentioned above, the plain strategy that an adversary follows is to immediately publish every block it produces. Indeed, this strategy gives the adversary the best

Functionality $\mathcal{F}_{\mathsf{CheckpointRand}}$

$\mathcal{F}_{\mathsf{CheckpointRand}}$ interacts with a set of parties $\mathbb{V}$, is parameterized by $k_c^{\top}$ and $\mathsf{maxvalid}(\cdot,\cdot)$, and holds the local chain C and the checkpointed chain C_c, both initially set to ϵ, and k_c, the current epoch's length, initially set to $k_c \xleftarrow{\$} (0, k_c^{\top}]$.
Upon receiving a message $(\textsc{CandidateChain}, C')$, if $C_c \prec C'$ set $C := \mathsf{maxvalid}(C, C')$. If $|C| - |C_c| \geq k_c$ then pick $r \xleftarrow{\$} \{0,1\}^{\omega}$, and set $C := C_c := C||r$ and $k_c \xleftarrow{\$} (0, k_c^{\top}]$.
Upon receiving a message $(\textsc{Read})$ from a party $\mathcal{V} \in \mathbb{V}$ return $(\textsc{Chain}, C)$.

Figure 15: The Randomized Checkpointing Functionality

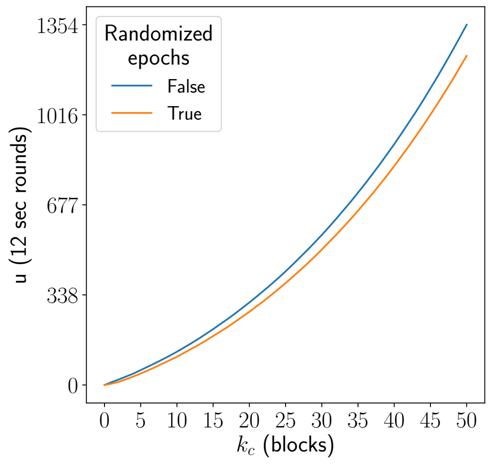

Figure 16: Comparison of the expected number of steps before absorption in the non-randomized and the randomized epoch length settings. The adversarial power is fixed to 51%.

chances of getting checkpointed, *assuming* that its chain is only 1 block shy of reaching the epoch's limit. However, if the checkpoint is further away, then publishing the chain will only allow the honest parties to "catch up". We note that this strategy is straightforward, but not necessarily the *optimal*; future work will explore alternative strategies which might produce better results for the adversary, such as taking into account the probability that k_c is equal to some value given $k_c^{\top}$ and choosing whether to publish the chain accordingly.

Similar to Section 3.2.2, the adversary will not adopt any of the honestly-generated blocks. However, it cannot anymore gain an advantage over the honest parties. Therefore, the states (i, j) where $i > j$ are now merged with the state (j, j). Algorithm 2 defines the updated chain generation mechanism; following the notation of Section 3.2.2, we set $m = (1 - m_0^{\Sigma})$, i.e. the probability that the adversary produces *at least* 1 block, and $\bar{m} = m^{(0)}$, i.e. the probability that the adversary does not produce *any* blocks.

Our simulations have shown that the behavior of the liveness probability and the expected steps is the same as in Section 3.2.2. Specifically, the liveness probability decreases significantly as the epoch length increases, while u increases roughly linearly with k_c, when the adversary controls a minority, and exponentially when it controls a large majority. However, randomizing the epoch lengths does improve both the liveness probability and the expected rounds compared to the plain setting of Section 3.2.2. Figures 16 and 17 depict the comparison of the expected rounds and the liveness probability respectively between the non-randomized and the randomized epoch length settings. For comparison, in the randomized setting after 300 steps for $k_c = 3$ the liveness probability is 0.871, compared to 0.7173 in the non-randomized setting.

Algorithm 2 The Markov chain construction algorithm for randomized epoch lengths.

```
function createMarkovChain($k_c$)
    createGraph($k_c, k_c$)
end function
function createGraph($i, j$)
    if $i = 0$ then
        addEdge(final, final, 1)
        return
    end if
    addEdge($(i, j), (i, j), \bar{h} \cdot \bar{m}$)
    if $i = j$ then
        if $i = 1$ then
            addEdge($(i, j), (k_c, k_c), m$)
            addEdge($(i, j)$, final, $h \cdot \bar{m}$)
        else
            addEdge($(i, j), (i - 1, j - 1), m$)
            createGraph($i - 1, j - 1$)
            addEdge($(i, j), (i - 1, j), h \cdot \bar{m}$)
            createGraph($i - 1, j$)
        end if
    else
        addEdge($(i, j), (i, j - 1), \bar{h} \cdot m$)
        createGraph($i, j - 1$)
        if $i = 1$ then
            addEdge($(i, j)$, final, $h$)
        else
            addEdge($(i, j), (i - 1, j - 1), h \cdot m$)
            createGraph($i - 1, j - 1$)
            addEdge($(i, j), (i - 1, j), h \cdot \bar{m}$)
            createGraph($i - 1, j$)
        end if
    end if
end function
```

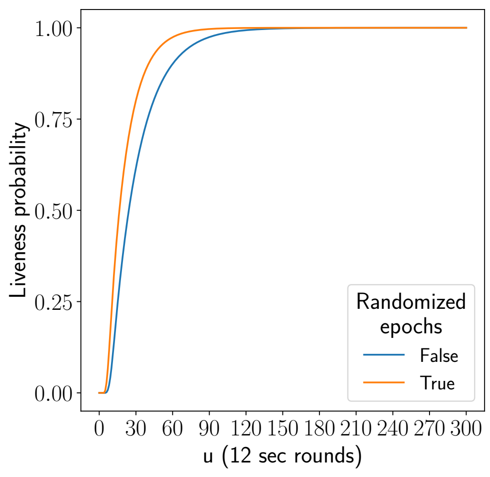

Figure 17: Comparison of the liveness property in the non-randomized and the randomized epoch length settings. The adversarial power is fixed to 51% and the epoch length is set to $k_c = 3$.

D Liveness for Non-Rushing Adversaries

In this section we slightly modify our model, in an attempt to both make it more realistic and achieve better liveness. Specifically, we no longer assume that the adversary is rushing, so now the adversary can no longer plan its strategy with the knowledge of the honest parties' messages during a round. More importantly, if, for a specific round, both an adversarial and an honest chain are published, it is no longer the case that the adversarial chain will be adopted. Instead we introduce the *network adoption parameter* γ as follows:

- γ: the probability that an adversarial chain is adopted in a round over an honest chain.

This change affects the Markov chain production algorithm. Algorithm 3 defines the construction of the updated Markov chain, taking γ into account. Specifically, in case both the adversary and the honest parties produce chains which can be checkpointed, there exists now a probability $1-\gamma$ that honest parties' chain is checkpointed; it is evident that, when $\gamma = 1$, the algorithm produces the same chain as Algorithm 1.

Algorithm 3 The absorbing Markov chain construction algorithm for the optimistic setting, defined by the chain construction function createMarkovChainOptimistic, parameterized by k_c, and the recursive helper function createGraph.

```
function createMarkovChainOptimistic($k_c$)
    createGraph($k_c, k_c$)
    addEdge(final, final, 1)
end function
function createGraph($i, j$)
    if $j > 0$ then
        for $l \in [0, j-1]$ do
            addEdge($(i, j), (i, j-l), \bar{h} \cdot m^{(l)}$)
            if $l > 0$ then
                createGraph($i, j-l$)
            end if
            if $i > 1$ then
                addEdge($(i, j), (i-1, j-l), h \cdot m^{(l)}$)
                createGraph($i-1, j-l$)
            end if
        end for
        addEdge($(i, j), (i, 0), \bar{h} \cdot (1 - m^{\Sigma}_{j-1})$)
        createGraph($i, 0$)
        if $i = 1$ then
            addEdge($(i, j)$, final, $h \cdot m^{\Sigma}_{j-1} + \bar{m} \cdot h \cdot (1 - m^{\Sigma}_{j-1})$)
            addEdge($(i, j), (k_c, k_c), m \cdot h \cdot (1 - m^{\Sigma}_{j-1})$)
        else
            addEdge($(i, j), (i-1, 0), h \cdot (1 - m^{\Sigma}_{j-1})$)
            createGraph($i-1, 0$)
        end if
    else
        addEdge($(i, j), (i, j), \bar{h}$)
        if $i = 1$ then
            addEdge($(i, j), (k_c, k_c), m \cdot h$)
            addEdge($(i, j)$, final, $\bar{m} \cdot h$)
        else
            addEdge($(i, j), (i-1, j), h$)
            createGraph($i-1, j$)
        end if
    end if
end function
```

However, now Algorithm 3 does not necessarily model a minimal execution. Specifically, it might be in the adversary's benefit to avoid risking a checkpoint of the honest parties' chain, and instead follow a conservative strategy. This strategy defines that the execution does not reach the state $(1, 0)$, i.e. the adversary publishes its chain when the honest parties are only one block short of reaching the checkpoint. Observe that, if the execution is at state $(i, 0), i > 1$ then the adversary will always checkpoint its chain. Algorithm 4 is a slightly modified version of Algorithm 3 which accommodates this change.

Algorithm 4 The absorbing Markov chain construction algorithm of the "conservative" strategy for the optimistic setting, defined by the chain construction function createMarkovChainOptimistic, parameterized by k_c, and the recursive helper function createGraph.

```
function createMarkovChainOptimisticConservative($k_c$)
    createGraph($k_c, k_c$)
    addEdge(final, final, 1)
end function
function createGraph($i, j$)
    if $j > 0$ then
        for $l \in [0, j-1]$ do
            addEdge($(i, j), (i, j-l), \bar{h} \cdot m^{(l)}$)
            if $l > 0$ then
                createGraph($i, j-l$)
            end if
            if $i > 1$ then
                addEdge($(i, j), (i-1, j-l), h \cdot m^{(l)}$)
                createGraph($i-1, j-l$)
            end if
        end for
        if $i = 1$ then
            addEdge($(i, j), (k_c, k_c), \bar{h} \cdot (1 - m^{\Sigma}_{j-1}) + \gamma \cdot h \cdot (1 - m^{\Sigma}_{j-1})$)
            addEdge($(i, j)$, final, $h \cdot m^{\Sigma}_{j-1} + \bar{\gamma} \cdot h \cdot (1 - m^{\Sigma}_{j-1})$)
        else
            addEdge($(i, j), (i, 0), \bar{h} \cdot (1 - m^{\Sigma}_{j-1})$)
            createGraph($i, 0$)
            if $i = 2$ then
                addEdge($(i, j), (k_c, k_c), h \cdot (1 - m^{\Sigma}_{j-1})$)
            else
                addEdge($(i, j), (i-1, 0), h \cdot (1 - m^{\Sigma}_{j-1})$)
                createGraph($i-1, 0$)
            end if
        end if
    else
        addEdge($(i, j), (i, j), \bar{h}$)
        if $i = 2$ then
            addEdge($(i, j), (k_c, k_c), h$)
        else
            addEdge($(i, j), (i-1, j), h$)
            createGraph($i-1, j$)
        end if
    end if
end function
```

In our analysis, in order to find the minimum liveness probability, we take into account both strategies. Specifically, for every execution we simulate both strategies and find the strategy which is best for the adversary, i.e. results in worse liveness probability. Figures 18 and 19 show the comparison between the optimistic setting and the standard execution of Section 3.2.2. The results in the optimistic setting are better both in terms of liveness probability and expected

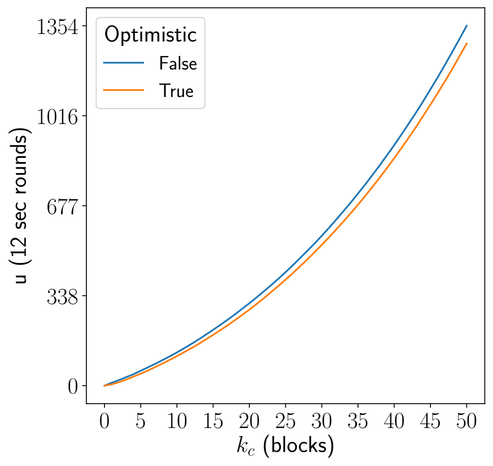

Figure 18: Comparison of the expected number of steps before absorption in the non-optimistic and the optimistic settings. The adversarial power is fixed to $50\% + 1$ and the network adoption parameter to $\gamma = 0.5$.

steps before absorption, which is expected since the adversary is now in a disadvantage compared to the standard setting.

E Centralized and Non-Interactive Timestamping

Similarly to checkpoints, the most straightforward way of realizing the timestamping functionality is as a centralized authority. The timestamping service is now parameterized by a EUF-CMA signature scheme and identified by a public key vk. Additionally, it keeps an internal counter c, which increases when a timestamp is issued. Interestingly, this counter can be removed under the assumption of a global clock which allows all parties to coordinate.

The timestamped object is the tuple $\langle r||c, \mathsf{Sign}(sk, r||c||m)\rangle$, consisting of the (monotonically increasing) time counter, the randomness r (cf. the checkpointing functionality $\mathcal{F}_{\mathsf{Checkpoint}}$), and the service's signature on the timestamped message m. In order to construct the authority as a federation of parties, a Byzantine Agreement protocol can again be deployed, similar to Section 3.3.

The major benefit of this mechanism lies in the non-interactive nature of signatures. A miner can broadcast the timestamped signature, along with the new block, and a validator can check it non-interactively; naturally, the security of the mechanism relies on the underlying signature scheme's security. Additionally, the timestamping authority does not need to maintain a list of timestamped objects; instead, the miners always choose the oldest, when provided with multiple timestamps for the same message. Therefore, the state that the timestamping service needs to maintain is $O(|c| + \kappa)$, whereas, assuming a global clock, the state is non-updatable and only $O(\kappa)$ long, comprising only of the signing key.

A further benefit of this approach is the ease of migration to a non-timestamped setting. When the blockchain achieves an adequate level of security and assistance is no longer needed, the timestamping service can simply halt its operation. In this case, the miners continue participating in the protocol uninterrupted, even though the chains are no longer timestamped. Therefore, the transition to the non-timestamped setting is seamless and without the need for extra effort, such as a hard fork of the blockchain.

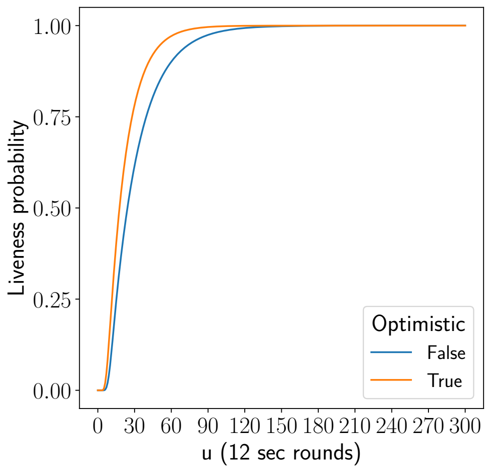

Figure 19: Comparison of the liveness property in the non-optimistic and the optimistic settings. The adversarial power is fixed to 50% + 1, the epoch length is set to $k_c = 3$, and the network adoption parameter to $\gamma = 0.5$.