Winkle: Foiling Long-Range Attacks in Proof-of-Stake Systems

Sarah Azouvi[1,2], George Danezis[1,2], and Valeria Nikolaenko[1]

[1]Facebook Calibra
[2]University College London

Abstract

Winkle protects any validator-based byzantine fault tolerant consensus mechanisms, such as those used in modern Proof-of-Stake blockchains, against long-range attacks where old validators' signature keys get compromised. Winkle is a decentralized secondary layer of client-based validation, where a client includes a single additional field into a transaction that they sign: a hash of the previously sequenced block. The block that gets a threshold of signatures (confirmations) weighted by clients' coins is called a "confirmed" checkpoint. We show that under plausible and flexible security assumptions about clients the confirmed checkpoints can not be equivocated. We discuss how client key rotation increases security, how to accommodate for coins' minting and how delegation allows for faster checkpoints. We evaluate checkpoint latency experimentally using Bitcoin and Ethereum transaction graphs, with and without delegation of stake.

1 Introduction

A number of blockchains are considering proof-of-stake mechanisms in place of proof-of-work, attracted by faster and deterministic finality as well as lower energy costs. In proof-of-stake blockchains a set of validators run a consensus protocol between themselves and agree on the next block of ordered transactions by collectively signing the block. Such protocols rely on the long-term security of validators' signature keys and a compromise of validators' past keys threatens full auditability through *Long-Range Attacks* [11]. A long-range attack is considered successful if an adversary was able to create an alternative chain of transactions, starting with the same genesis block, that can not be distinguished from the real chain. A number of solutions have been proposed to foil long-range attacks such as publishing checkpoints off-chain in software updates or in a reputable archive (such as the front page of a national newspaper). Those solutions are difficult to deploy without introducing a highly unsatisfactory vector of centralization.

Validator key rotations help alleviate the problem, assuming secure destruction of older keys. However, validators might have auxiliary incentives to sell their old keys to an adversary, especially when real-world identities of validators are unknown in a permissionless system and reputation is not at risk. When dishonest behaviour of a validator becomes rational, real-world security of the whole system is at great risk. We notice that corrupting a significant number of coin holders, even after they have no more stake in the system, is far more challenging as they are much more numerous than validators. This observation brings us to introducing Winkle — a novel mechanism that leverages votes from clients creating a decentralized secondary layer of client-based validation to confirm checkpoints (snapshots of the blockchain) and to prevent long-range attacks on proof-of-stake protocols. The voting mechanism is very simple: each client augments their transaction with a single additional field — a hash of the previously sequenced block. Once this transaction gets signed by the client and submitted to the chain, it serves as a vote or a confirmation for a block weighted by the number of coins the client holds under their account. The block that gets a threshold of confirmations is called a "confirmed" checkpoint. We show that under plausible and flexible security assumptions confirmed checkpoints can not be equivocated and serve as irreversible snapshots of the blockchain.

Our contributions. We design Winkle to strengthen consensus protocols with dynamically changing validators against long-range attacks (Sec. 2-3). Though Winkle can secure other systems as well, it is mainly applicable to blockchains based on Byzantine Fault Tolerant (BFT) consensus such as PBFT [21], LibraBFT [3],

Tendermint [6], HotStuff [26], or SBFT [15]. Our solution does not require unincentivized validators to remain honest. Instead, Winkle allows each coin holder to augment a transaction with a vote for a previous block. We prove that after a critical mass of coins has voted for a block, the block becomes a "confirmed checkpoint" and cannot be equivocated even if validators who worked on constructing this block have leaked their keys to the adversary. Furthermore, in systems protected by Winkle, an adversary enacting a long-range attack and building a forking chain cannot freely replicate transactions from honest users since those commit to checkpoints on the real chain. In systems with probabilistic finality (not typical proof-of-stake systems) this mechanism also gives protection against replay attacks as was previously observed in [12].

We give plausible and flexible security assumptions (Sec. 4), showing trade-offs between the fraction of byzantine accounts and the quorum size. We notice that the assumptions are far more flexible than the byzantine bounds in the BFT protocols, *e.g.* if the disconnected users constitute a negligible fraction, the quorum size in Winkle can be just slightly above the bound on the byzantine accounts. Our assumptions are also tuned to key rotations, allowing accounts to heal from a compromise. We put up a definition for the long-term security of the validator-based consensus protocol (Sec. 5) and prove that Winkle satisfies the definition overcoming challenges of weighted-by-stake voting in the presence of constantly moving weights. We introduce delegation mechanism, where less active accounts can delegate their stake to more active accounts, in order to facilitate faster confirmation of checkpoints. We discuss how to safeguard minting of coins. Finally we simulate Winkle on the real-world datasets of Bitcoin and Ethereum and evaluate checkpointing delays with and without delegates (Sec. 6) showing that the block can be confirmed within several hours to a few days, allowing validators to safely leak their old keys after a few days of use. We discuss related work, other applications and future research directions (Sec. 7-9).

2 Background and Research Question

An account-based blockchain model. A blockchain maintains an evolving decentralized *database* that keeps track of the ownership of assets, and allows their transfer according to rules encoded in *transactions*. We represent the database, at any consistent point, as a key-value store, which maps account addresses to account states: $\mathbf{DB} = \{(\mathsf{A}_j, \mathsf{state}_j) \mid j = 1, 2, \ldots, N\}$, where $\mathsf{A}_j \in \{0,1\}^{256}$ is the account address, $\mathsf{state}_j \in \mathcal{S}$ is the value under the account, N is the number of accounts. The account state holds the following values:

- pk: public key for signature verification,
- seq: an incrementing sequence number, that prevents replay attacks,
- value: number of coins, that maps to the "voting power".

The account's state may also hold other meta-data or auxiliary information. To simplify notation we write $\mathbf{DB}[\mathsf{A}]$ to denote account A's state and use field notation to represent its values: $\mathbf{DB}[\mathsf{A}].\mathsf{pk}$, $\mathbf{DB}[\mathsf{A}].\mathsf{seq}$, $\mathbf{DB}[\mathsf{A}].\mathsf{value}$. We denote by $\mathbf{DB}.N$ the number of accounts in the database and by $\mathbf{DB}.S_{tot}$ the total number of coins, both numbers may be changing with the modifications to the database, *i.e.* when accounts and coins get created or destroyed.

Transactions. The database evolves, from one consistent state to the next, via processing transactions. A transaction (tx) is comprised of the following data:

- sender: the address of the transaction's sender,
- seq: sequence number that should match the sender's account seq plus one,
- program: an efficiently computable deterministic function that mutates the state of the database, *i.e.* $\mathsf{program}(\mathbf{DB}) \to \mathbf{DB}'$,
- σ: a cryptographic signature over the previous fields.

To clarify exposition we represent those values as $\mathsf{tx.sender}, \mathsf{tx.seq}$, $\mathsf{tx.program}$, $\mathsf{tx}.\sigma$. Furthermore, we assume that we can derive from tx.program a list of accounts that receive coins, denoted as tx.raddrs, and corresponding amounts. In this work we principally consider programs associated with asset transfers implementing a monetary system, relevant to cryptocurrencies, but we also allow transactions mutating other meta-data of user accounts (e.g. for key rotation and delegation introduced later).

Validator Based Consensus with Reconfiguration. We abstract a Validator Based Consensus with Reconfiguration (VBCR) as a mechanism that provides a chain of collectively signed blocks on which consensus has been reached. In this work a *block* refers to a long sequence of transactions (e.g. a day of activity

or an epoch of a different duration), rather than a short-term block within the low-level consensus protocol. The first block (genesis block) determines the initial state of the database: $\mathsf{B}_0 = \mathbf{DB}_0$. A *chain of blocks* $(\mathsf{B}_1, \mathsf{B}_2, \ldots)$ modifies the state of the database. Each *block* B_i for $i > 0$ is of the form $\mathsf{B}_i = [h_{i-1}, \mathbf{T}_i, \sigma_i]$, where h_{i-1} is the cryptographic hash of the previous block, *i.e.* $h_{i-1} = \mathsf{H}(\mathsf{B}_{i-1})$; $\mathbf{T}_i$ is an ordered list of transactions and σ_i is a signature over $(h_{i-1}, \mathbf{T}_i)$. Given a genesis state $\mathbf{DB}_0$ and the chain of blocks, we define recursively $\mathbf{DB}_i = \mathbf{T}_i(\mathbf{DB}_{i-1})$. Here $\mathbf{DB}_i$ is a result of application of programs in transactions' list $\mathbf{T}_i$ one by one in a given order to $\mathbf{DB}_{i-1}$.

The system is governed by the consensus protocol run between the validators. The validator set is determined by the set of verification keys: $\mathbf{V} = \{\mathsf{pk}_k\}_{k=1}^{|\mathbf{V}|}$. Each signature key sk_k is private to k-th validator. We assume there is a function $\mathsf{gov}(\cdot)$ on the previous block (or the genesis block initially)[1], that returns the current set of validators: $\mathbf{V}_i = \mathsf{gov}(\mathsf{B}_{i-1})$. Typically, we consider the signature σ_i in the block $\mathsf{B}_i = [h_{i-1}, \mathbf{T}_i, \sigma_i]$ as being valid if a certain threshold of validators $\mathsf{V}_i = \mathsf{gov}(\mathsf{B}_{i-1})$ have contributed to the aggregate signature σ_i.

For each block B_i we define a function $\mathsf{seq}(\cdot)$ which returns the height of the block namely the number of blocks preceding it up to the genesis block B_0, the function implicitly takes the chain of blocks as input. By convention, we use a subscript of the block to denote the height, *e.g.* $i = \mathsf{seq}(B_i)$.

Chain validation. A public verification function $\mathsf{validate}(\cdot, \cdot)$ defines a predicate that takes a chain of blocks as a first argument and a new block as a second argument: $\mathsf{validate}(\mathbf{B}, \mathsf{B}_{i+1})$. The output indicates whether the block B_{i+1} can be successfully chained with the previous blocks $\mathbf{B} = (\mathsf{B}_0, \ldots, \mathsf{B}_i)$. The predicate encodes the business logic of the state machine describing the blockchain. A chain of blocks $\mathbf{B} = (\mathsf{B}_0, \mathsf{B}_1, \ldots, \mathsf{B}_i)$ is valid if either $\mathbf{B} = (\mathsf{B}_0)$ or recursively computed predicate is true, *i.e.* $\forall i > 0\ \mathsf{validate}((\mathsf{B}_0, \mathsf{B}_1, \ldots, \mathsf{B}_{i-1}), \mathsf{B}_i) = \mathsf{true}$. Auditing any chain of blocks $\mathbf{B}$, would start at block B_0, append the blocks one-by-one and test the validity of the chain incrementally. The predicate $\mathsf{validate}$ checks validators' signature on the new block, checks transactions' signatures within the new block and applies transactions one-by-one to the database state, checking the sequence numbers to prevent replay attacks. In more details, taking as input $\mathbf{B} = (\mathsf{B}_0, \ldots, \mathsf{B}_i)$ and B_{i+1}, the predicate assumes that $\forall j \leq i\ \ \mathsf{validate}((\mathbf{B}_0, \ldots, \mathbf{B}_{j-1}), \mathbf{B}_j) = \mathsf{true}$ and $\mathsf{H}(\mathsf{B}_{i-1}) = h_i$. Let the state of the database after applying all the transactions from the chain of blocks $\mathbf{B}$ in sequence be $\mathbf{DB}_i$. Let $\mathsf{B}_{i+1} = [h_i, \mathbf{T}, \sigma]$. First, the validators' public keys are retrieved from the previous block: $\mathsf{pk}_{i+1} = \mathsf{gov}(\mathsf{B}_i)$, the signature σ is verified under that key: $\mathsf{verify}(\mathsf{pk}_i, (h_i, \mathbf{T}), \sigma)$. If the verification fails, verify outputs false and halts, otherwise it continues. Transactions are processed from the list $\mathbf{T} = (t_1, \ldots, t_k)$ in order. Let $\mathbf{DB}_i^0 = \mathbf{DB}_i$ and for $j = 1, \ldots, k$ we verify that: (1) the sequence number is equal to the value under the account plus one: $t_j.\mathsf{seq} = \mathbf{DB}_i^{j-1}[t_j.\mathsf{sender}].\mathsf{seq} + 1$, (2) the signature $t_j.\sigma$ verifies under the public key stored under the sender's account $(\mathbf{DB}_i^{j-1}[t_j.\mathsf{sender}].\mathsf{pk})$, (3) additional checks can be applied to validate the transaction depending on the business logic of the blockchain, encoded in the transaction program. If any of the transactions fail at least one check then the procedure returns false and halts. If transaction t_j passed the checks successfully, it gets applied to the database to advance it's state: $\mathbf{DB}_i^j = t_j.\mathsf{program}(\mathbf{DB}_i^{j-1})$, the sequence number under the account is incremented by one. If the transaction was not applied successfully, the procedure outputs false and halts.

Safety of VBCRs. We say that the chain validation *audit is safe* if and only if any two parties that successfully completed the chain validation procedure would have a consistent view of the chain of blocks (*i.e.* the chains will be equivalent or a chain of one party will be a subchain of another party's chain).

Definition 1. *(Perpetually honest validator) A perpetually honest validator follows the protocol and maintains the secrecy of their signing keys in perpetuity (the adversary may never have access to them), and only signs a single block at a certain height.*

If sufficient number of validators are perpetually honest in keeping all their keys $(\mathsf{gov}(\mathsf{B}_i))$ unknown to the adversary in perpetuity, then the verification procedure ensures *audit safety.*

Long-Range Attacks on VBCRs. The safety of chain validation audit described depends on validators being perpetually honest. This imposes a heavy burden on the security of the validators' signature keys:

[1]In the VBCR abstraction block's boundaries capture events of validators' set change, as those are relevant to long-range attacks.

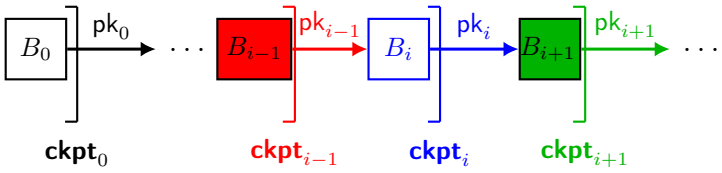


Figure 1: Each block commits the the full sequence of previous blocks, and forms a potential checkpoint that clients vote to confirm. Validator membership and keys may change across blocks (but not within them). Block B_i is signed by the key $\mathsf{pk}_{i-1} = \mathsf{gov}(\mathsf{B}_{i-1})$, and the key defined in $\mathsf{pk}_i = \mathsf{gov}(\mathsf{B}_i)$ signs B_{i+1}.

they need to remain secret forever, otherwise the validity of the sequence cannot be safely audited. This motivates us to define *Eventually compromised validators* as follows:

Definition 2. *(Eventually compromised validator) An eventually compromised validator after a block* B_i*, leaks all its previous signature keys* $\{\mathsf{sk}_{i'} \mid 0 < i' < i\}$ *to the adversary.*

Any VBCR protocol becomes unauditable under eventually compromised validators. For block B_i and any block $\mathsf{B}_{i'}$ in the history of B_i ($i' < i$), eventually the adversary will compromise sufficient number of validators and sign an alternative block to follow $\mathsf{B}_{i'}$ with a different set of transactions, defeating the Audit Safety property. This problem is referred to in the literature as a *long-range attack*, and Winkle strengthens Validator Based Consensus with Reconfiguration systems against those types of attack. We note that the assumption that eventually compromised validators provide *all* signing keys before height i represents a strong adversary, and subsumes any adversary that may have access to a subset of past keys instead. The compromise of the old keys in BFT or Proof-of-Stake consensus protocols is particularly devastating as creating an alternative chain is computationally inexpensive.

3 The Winkle Mechanism

Definitions. Without losing generality we assume that the set of validator keys may change at each block. Each block defines a *checkpoint* that coin holders can vote to confirm in the future, as illustrated (see Fig. 1). A checkpoint is a binding commitment to a chain of blocks sequenced by the VBCR: $\mathsf{ckpt}_i = \mathsf{H}(\mathsf{B})$, where B is the last block the checkpoint points to. We abuse the notation and define the height written as a subscript of a checkpoint as being equal to the height of a block it is committing to. In turn this height defines a global order on all checkpoints referring to a sequence of blocks produced by a secure VBCR.

We call two checkpoints ckpt_i, ckpt_j *consistent* if they commit to blocks B_i, B_j such that B_i is an ancestor of B_j (we write $\mathsf{ckpt}_i < \mathsf{ckpt}_j$)) or vice versa. Checkpoints that are not consistent commit to blocks that are on different sides of a fork and therefore cannot be part of a sequence produced by a safe VBCR. We assume everybody agrees on the genesis checkpoint: $\mathsf{ckpt}_0 = \mathsf{H}(\mathsf{B}_0)$.

3.1 Basic Winkle scheme

Checkpoint voting. Each account may vote for a checkpoint by including it in a signed transaction, and every coin carries the vote of its sender (we call this the *propagation rule*). The vote is also associated with the remaining value in the account.

For example, if an account A_1 has 10 coins and sends 1 coin to an account A_2 with a vote for checkpoint ckpt, then there is a system-wide vote with weight of 10 for checkpoint ckpt: 9 of which is associated with A_1 and 1 associated with A_2. This also means that an account could have different votes for different coins held, e.g., if A_2 received one coin from account A_1 with a vote for ckpt and one coin from another account A_3 that voted for a previous checkpoint ckpt', then A_2 carries one vote for ckpt and one vote for ckpt'. Whenever A_2 sends a new vote, then all of the coins in the account will be counted towards that new vote, and the previous votes are over-written. An account sending a transaction has to vote on the latest available checkpoint, otherwise its transaction will not get sequenced. As we show, there cannot be any two inconsistent checkpoints within one account due to new chain validation rules.

More formally, we define a *weighted vote* as a tuple $\mathsf{wVote} = (\mathsf{ckpt}, \mathsf{w})$, where ckpt is a checkpoint and w is a real positive number. We augment transactions and accounts with additional information: (1) we augment each transaction tx with a parameter $\mathsf{tx.vote}$ that we call a checkpoint vote: $\mathsf{tx.vote} = \mathsf{ckpt}$; (2) we change the structure of the database: for each account A in place of value $\mathbf{DB}[\mathsf{A}].\mathsf{value}$ we now store a set of weighted votes $\mathbf{DB}[\mathsf{A}].\mathsf{votes}$.

Accounts managed by honest and active users follow some constraints: (1) all their votes are for pairwise consistent checkpoints; and (2) each of their transactions always contains a vote for the latest available checkpoint. Condition (2) introduces a synchrony assumption, however this should not represent a bottleneck: first of all, we assume blocks are abstractions of long periods of consensus under the same set of validators (e.g. many hours or days); and, second, clients need to be aware of the latest checkpoint since they anyway need to know the latest validator set (from the previous block). The set is required in order to know which entities to direct transactions to, and also to be able to authenticate reads from the latest state of the database.

Chain validation. We augment the predicate $\mathsf{validate}$ executed on input $\mathbf{B} = (\mathsf{B}_0, \ldots, \mathsf{B}_i)$ and B_{i+1} with an additional check on each of the transactions in block B_{i+1} (see the 'Chain Validation' rules in Section 2). Each transaction tx_j in B_{i+1} for $j = 1, \ldots$ must contain a vote on ckpt_i, where $\mathsf{ckpt}_i = \mathsf{H}(\mathsf{B}_i)$. If the transaction passes the checks, given the previous set of the database $\mathbf{DB}_i^{j-1}$ ($\mathbf{DB}_i^0 = \mathbf{DB}_i$) the next state of the database is the same, except for the following changes. (1) For each of the receiving accounts $\mathsf{A} \in \mathsf{tx}_j.\mathsf{raddrs}$ that obtain some amount v with transaction tx_j, if the vote is already in the set: $(\mathsf{tx}_j.\mathsf{vote}, \mathsf{w}) \in \mathbf{DB}_i^{j-1}[\mathsf{A}].\mathsf{votes}$ for some w then v gets added to the weight w of the existing element, otherwise a new tuple $(\mathsf{tx}_j.\mathsf{vote}, v)$ is added to the set $\mathbf{DB}_i^{j-1}[\mathsf{A}].\mathsf{votes}$. (2) For the sending account A: $\mathsf{tx.sender} = \mathsf{A}$, the set $\mathbf{DB}_i^{j-1}[\mathsf{A}].\mathsf{votes}$ gets squashed into a single element $(\mathsf{tx}_j.\mathsf{vote}, v)$, where v is the sum of all the values in the set $\mathbf{DB}_i^{j-1}[\mathsf{A}].\mathsf{votes}$ minus the value sent in the transaction tx_j.

We define the *highest confirmed checkpoint* as a function $\mathsf{HighestCkpt}(\mathbf{DB})$ computed over all accounts in a database $\mathbf{DB}$. It represents the highest checkpoint ckpt_k ($k \leq i$) which precedes the checkpoint votes of $\mathbf{q}$ fraction of accounts' (weighted), i.e.

$$\sum_{\substack{\mathsf{A} \in \mathbf{DB} \\ \mathsf{wVote} \in \\ \mathbf{DB}[\mathsf{A}].\mathsf{votes}}} \mathsf{I}(\mathsf{wVote.ckpt} \geq \mathsf{ckpt}_k) \cdot \mathsf{wVote.w} \geq \mathbf{q} \times \mathbf{DB}.S_{tot}$$

The indicator function $\mathsf{I}(\mathsf{wVote.ckpt} \geq \mathsf{ckpt}_k)$ takes value 1 if the vote is for a checkpoint higher than k and consistent with ckpt_k, and otherwise takes value 0. The highest confirmed checkpoint of the database: $\mathsf{HighestCkpt}(\mathbf{DB})$ is the one with the largest height that is supported by a fraction $\mathbf{q}$ of the total stake.

Validator-free full audit. To validate a confirmed checkpoint, associated with a sequence of transactions $\mathbf{T}$ leading to it, a client performs the following steps: the client starts the validation process at the genesis state $\mathbf{DB}_0$, which needs to be known and to be authentic. The client then applies each transaction in the sequence in order, recomputes the state of the database and recomputes the $\mathsf{HighestCkpt}(\cdot)$ function, according to votes in each account, to determine the confirmed checkpoint. The client accepts the highest confirmed checkpoint.

We show that this audit process ensures safety, in the sense that the confirmed checkpoint returned is guaranteed to be consistent with the state of the honest chain (*i.e.* the chain of transactions built from the genesis state up to the confirmed checkpoint is guaranteed to be exactly the same as in the honest chain). However, it does not guarantee *freshness*, in the sense that the checkpoint returned may not be the highest one in existence. Clients can query multiple sources, and pick the highest confirmed checkpoint.

Determining the latest confirmed checkpoint is especially important to determine the latest set of validators currently maintaining the consensus protocol. Once the current set of validators is securely determined their (valid for some period) signatures can be used to track updates of the database state starting from the highest known confirmed checkpoint. Validation of the chain built prior to the known confirmed checkpoint does not involve any checks of validators' signatures.

3.2 Delegation

The basic Winkle mechanism requires a mass of account holders to vote for a checkpoint to become the highest confirmed checkpoint. Our experiments (see Section 6), suggest that many accounts within existing blockchains are dormant, and therefore would seldomly vote. In turn this leads to checkpoints being confirmed in months (up to three years for Bitcoin), during which the system is vulnerable to validator old key compromise. To reduce this latency we introduce a simple *delegation* model.

An account may be created with a 'delegate' field $\mathbf{DB}[\mathsf{A}].\mathsf{delegate}$ referencing the address of another account A_d with no delegate field. This indicates that the voting power of account A is delegated, and therefore contributes, to the weight of account A_d. This mechanism only allows for a single level of delegation. Accounts that delegate still need to include a vote for the latest block when they transact. When computing the highest confirmed checkpoint, only the most recent vote of either the account that delegates or its delegate is counted (i.e., the stake of accounts that delegate are counted only once, either with the delegate or independently if their transaction is more recent than the delegate's vote). Accounts that delegate may change their choice of delegate through a transaction. When an account changes its delegate, it also includes a vote and thus our propagation rule stays the same. For security, we assume that honest users delegate to honest delegates only. Note that Winkle compared to the delegated proof-of-stake systems requires a weaker trust assumptions from the clients. Clients need to trust the delegates which simply confirm checkpoints and are not running consensus protocol between themselves, any client can act as a delegate in Winkle and is not required to setup complex infrastructure in contrast to nodes running consensus.

Accounts that vote act as *pools*, for accounts that delegate. When pools vote often on behalf of dormant accounts, we expect checkpoints to be confirmed faster. To further speedup check-pointing a system of economic incentives may be set up to encourage pools to vote often, as well as for other accounts to delegate to those that do—all while preventing over-concentration of stake.

Economics of pools & decentralization. A key issue with delegation is the tendency towards centralization: delegating votes to well operated pools is simpler for a user than voting themselves (which requires sending transactions). At a logical extreme, one pool may emerge with a large amount of voting power, which may later have its key compromised and used to perform long-range attacks. To disincentives such concentration of voting power we adapt crypto-economic ideas from [5] and design an incentive scheme that should maintain close to a constant number of pools in the system.

We define as $U > 1$ a parameter to affect target number of pools we wish to incentivize in the system. Each pool incurs some fixed cost for operating, which we denote as E_i for pool i. E_i represents the operational pool cost, since it must be online to execute transactions and vote in an epoch, as well as an additional fee each pool must provide the system per epoch to vote. We denote as s_i the weighted vote delegated to a pool i, and as S the total vote weight in Winkle. For each pool i we define its incentive weight as $w_i = \min(s_i, \frac{\mathbf{q} \cdot S}{U})$. The value $\mathbf{q}$ is the fraction of the vote necessary to confirm a checkpoint.

We assume that during an epoch of length T time units some monetary value is set aside to reward users and pools that vote to confirm this checkpoint. We define the rate of reward per unit time for each coin in an epoch as d. When a checkpoint is confirmed we observe which fraction of $\mathbf{q}$ votes contribute to its confirmation. We split and assign the total reward, of total value $d \cdot T \cdot S$, to each pool i proportionately to its incentive weight w_i. Taking into account the costs advertised by each pool, its 'profit' for an epoch it contributed to confirming is:

$$R_i = \frac{w_i}{\mathbf{q} \cdot S} d \cdot T \cdot S - E_i = \frac{w_i}{\mathbf{q}} d \cdot T - E_i$$

However, if the pool did not contribute a vote to confirm the epoch, then the reward is zero – and for simplicity we assume that the cost E_i for this epoch is also zero. Assuming that a pool manages to participate in a fraction a' of confirmations then its expected reward per epoch is:

$$\bar{R}_i = \frac{a'}{\mathbf{q}} w_i \cdot d \cdot T - a' \cdot E_i$$

During each epoch any reward that is not distributed to pools is kept to reward future epochs (increasing future d values). We assume that each pool keeps a fee F_i for itself, and then distributes the remaining $R_i - F_i$ to the pool participants as an incentive to remain in the pool, according to their contribution in

terms of weighted votes. If a pool does not contribute to the vote we assume it does not keep any fee or distributes any incentive to participants.

We now determine a number of properties for the incentive scheme above. We assume that both pool operators and users are honest – in that they follow the Winkle protocol correctly – but are however rational in their choice of pools. We therefore stress (in line with recent thinking[2]) that the incentives mechanisms is there to protect honest users against perverse incentives creating a more brittle system, rather than argue that the incentives mechanism prevents malicious users from participating (such an argument is impossible without considering external incentives they may have to do so, with potentially unbounded rewards).

Equilibrium implies $s_i \leq \mathbf{q} \cdot S/U$. Our first argument is that a user does not have incentives to participate in a pool with voting weight larger than $\mathbf{q} \cdot S/U$, and in fact a pool operator also has no incentive to operate such a pool. This is a straight forward implication of the incentive weight w_i being capped at $\mathbf{q} \cdot S/U$: any additional votes contributed to the pool have a zero marginal rate of return for the pool, and a negative rate of return for users (since distribution of the user incentives $R_i - F_i$ is done per contribution to the pool with no cap). A rational user will always have incentives to defect from a pool with voting weight larger than $\mathbf{q} \cdot S/U$ to a pool with equivalent characteristics E_i, F_i and a' with a lower voting weight. Therefore in an equilibrium allocation of user voting weights to pools it must be that $s_i \leq \mathbf{q} \cdot S/U$. As a result for the remaining of our analysis we consider that $w_i = s_i \leq a \cdot S/U$. The lack of incentive to operate larger pools is a key part for our argument that there will be about U/a pools in the system (the second part involves bounding the number from below).

Delay to confirm an epoch. For a pool to participate and vote it should expect at the very least a positive return. Since a vote for a later block also counts as a vote for earlier ones, a pool may chose to vote seldomly to reap high rewards per action. We would instead like pools to vote on each block to ensure blocks are quickly confirmed:

To generate a positive return on a single block it should hold that:

$$\frac{a'}{\mathbf{q}} w_i \cdot d \cdot T - a' \cdot E_i \geq 0$$

which in turns implies:

$$T \geq \frac{\mathbf{q}}{d} \cdot \frac{E_i}{s_i}$$

As the cost E_i of participating as a pool increases, so does the the minimum time a pool will wait before it votes in order to guarantee a positive return. On the other hand, the larger the voting power s_i (subject to the upper bound above) the lower the delay in voting to generate a return. This dynamic establishes a lower bound on the pool delay in voting.

Competitive pressures also bound the delay in voting from above. Pools observe the concentration of voting power amongst pools (or their delay in voting). They have to compete to be within the fraction $\mathbf{q}$ of voting power to confirm a block in order to generate any returns. Therefore no pool has an incentive to vote any later than the time a fraction $\mathbf{q}$ of voting power would have incentives to vote, since it would risk not being included in the reward for the confirmed blocks.

Characteristics of pools. Let's call t_m the maximum time at which the fastest set of pools (in terms of E_i and s_i above) comprising a fraction $\mathbf{q}$ of the voting power have incentives to vote. Any pool that finds it unprofitable to participate before this time will be excluded from the rewards systematically, and therefore will receive a zero return (its value a' will tend to zero). This establishes a couple of constraints in terms of the size of the pools and their costs to operate:

Assuming a pool expects to make a rate of return superior to the fee F_i, it must hold that:

$$\frac{a'}{\mathbf{q}} s_i \cdot d \cdot T_m - a' E_i \geq F_i$$

This constrains the fee pools can charge in relation to their voting power and costs. Obviously the higher the costs they incur the lower the fee; and the larger their voting power the higher the fees they can charge. This dynamic creates an upwards pressure on the size of pools.

[2]See "Rationality is Self-Defeating in Permissionless Systems" `https://bford.info/2019/09/23/rational/`

In fact small pools are simply not viable subject to the constraint (by rewriting the bound above):

$$s_i \geq \frac{\mathbf{q}}{a'} \frac{F_i + a' E_i}{d \cdot T_m}$$

Even by setting a fee at $F_i = 0$, and making no profit, small enough pools are not viable due to the fixed costs to participate E_i. Further, all other things being equal users always have incentives to delegate to larger pools, since those will make more profit and therefore provide a potentially higher rate of user incentive. This competitive pressure will ensure that we expect pools not only to be capped from above to $s_i \leq \mathbf{q} \cdot S/U$ but also from below, leading the system to having $S/\mathbf{q}$ pools of roughly equal size at equilibrium.

Competitions for user voting power. Pools have endogenous incentives to keep their costs E_i down, since any surplus can be turned into a larger profit F_i. Therefore if a pool has the options to lower its cost it will. We assume that the system should require a minimal payment of E to participate as a pool per epoch to ensure those costs do not become zero.

However, pools also compete for users, since it is user voting power that ultimately determines their voting power s_i. Users can change their delegates, and we assume that they will move to pools offering larger user incentives in terms of better rates of return per vote weight delegated. The rate of return for a user delegating one vote to pool i is:

$$u_i = \frac{\bar{R}_i - a' F_i}{s_i} = \frac{a'}{\mathbf{q}} d \cdot T - a' \left(\frac{E_i + F_i}{s_i} \right)$$

A pool can maintain an equal return to its users by either decreasing its fee F_i or by attracting more users' voting power to the pool. In a competitive setting users will change their delegation to pools offering higher returns. Therefore at equilibrium, for two pools the rate of return per vote must be equal ($u_i = u_j$) for users to not switch.

Assuming two pools within the faction that confirms blocks with likelihood $a' = \mathbf{q}$. Equality of rates of return reduces to the constraint:

$$\frac{s_j}{s_i} = \frac{E_j + F_j}{E_i + F_i}$$

where each of the pools can control only its own profit F, and can try to attract more user votes s. Attracting more user votes allows a pool to increase F, but the only tool available for doing so is increasing the rate of return per user vote which involves decreasing F (subject to fixed costs E). This puts a downward pressure on the fee F. Finally, for two pools with equal user vote share $s = s_i = s_j$, it holds that the difference in fees that are able to charge is:

$$F_i - F_j = (E_i - E_j)$$

Any fee differences are related to the difference in operational and pool participation costs. Therefore if one pool defects from a cartel and charges lower fees, all others will have to charge low fees, subject to their respective costs to ensure they provide a comparable rate of return to users, and to avoid them defecting. This dynamic supports a competitive ecosystem of pools.

3.3 Minting and Stake Bleeding Attacks

Another type of a long-range attack are Stake Bleeding Attacks [13]: an adversary, in a forking chain, may accumulate the rewards associated with creation of new blocks in order to inflate its stake until it accumulates enough to confirm an inconsistent checkpoint. To protect against such attacks we require every minting event to take effect only after the block containing it gets confirmed as a checkpoint.

Different Proof-of-Stake blockchains use different reward and minting mechanisms, and some also contract the monetary supply. For example, there can be a minting key capable of creating new coins, this key does not have to stay honest and secure in perpetuity. Since our mechanism guarantees that without minting an alternative forking block can not be accepted as a checkpoint, even if the old minting key is leaked to the adversary, the alternative minting transaction will never take an effect, as it will have to get checkpointed by the old money supply.

The amount of money allowed to get minted or destroyed at a time does not have to be limited as long as the fraction of stake in hands of the adversary remains bounded as per our assumptions (see Section 4).

3.4 Key Rotation and Account Healing

Accounts are controlled by signature keys. Key rotation operations shorten the lifetime of keys and prevent obsolete keys from issuing new transactions. However, this does not prevent a variant of long-range attacks on Winkle: an adversary that gains access to an account's old signature key that has been already rotated to a new key, may still use the stolen key to create past transactions, interfering with the safety of Winkle's audit. It is prudent to expect that any long term active account at some point may fail to protect a historic key. In the purely static compromise model this account would have to be considered under the control of the adversary for ever after, it is then likely that eventually the volume of stake under the control of such accounts would exceed any fractional threshold, threatening the security of Winkle. Therefore we adopt a more appropriate model where an adversary may compromise an account's key, but after the key is rotated may loose control over the account. Such an adversary may also compromise some old key that is not being currently active. An account holder A may include a special *key rotation* transaction t within block B_i, updating the public key associated with the account to pk'. The transaction is signed with the public key associated with the account (namely $\mathbf{DB}_i[\mathsf{A}].\mathsf{pk}$). Following the transaction t being applied, the database associates a new public key with account A.

We show, in our proof, that Winkle benefits from key rotations. In cases when some historic account key is compromised, but subsequent keys included in a confirmed checkpoint are not, the account does not have to be considered as contributing to the voting weight that the adversary commands to mount a long-range attack thereafter. This provides a forward security property, which we call *account healing*.

4 Security Assumptions

Eventually Compromised Validators. In our model, we assume that all validators are eventually compromised, and share their keys with the adversary after a confirmed checkpoint has been generated by Winkle. (In Section 6 we evaluate how long each block awaits to become a confirmed checkpoint). The *honest chain* or database is created by validators while they are honest and an *adversarial chain* is created by the adversary after the keys have been leaked. Without loss of generality we aggregate all validators into a single validator and we let the concrete instantiation of the consensus protocol to determine the number of validators, the exact form of keys, signatures and the conditions on their validity.

Honesty of accounts. Winkle leverages honest account holders for security. We assume account holders are harder to compromise than validators because there are more numerous. For example in Bitcoin, according to our experiments, as of September 2019 there are 889 accounts that hold one third of the total stake (see Fig. 3) compared to four miners that control more than half of the hashrate [4]. Accounts are of four types, and their type may change over time:

- active and honest ($f_{\mathbf{A}}$) accounts are connected to the true chain, they receive the latest checkpoint before the next checkpoint becomes available,
- byzantine ($f_{\mathbf{B}}$) accounts share keys with the adversary and arbitrarily deviate from the protocol,
- eclipsed ($f_{\mathbf{E}}$) accounts get eclipsed from the honest chain during the long-range attack and may transact on the adversarial chain while the adversary does not control their keys,
- dead ($f_{\mathbf{D}}$) accounts that do not transact during the attack and stay idle.

Each account belongs to only one category, therefore the fractions representing each category add up to one $f_{\mathbf{A}} + f_{\mathbf{B}} + f_{\mathbf{D}} + f_{\mathbf{E}} = 1$. Those fractions are calculated at the boundary of the blocks, *i.e.* for any checkpoint ckpt we consider the corresponding state of the database $\mathbf{DB}$ and look into the fraction of accounts of each type in $\mathbf{DB}$. As explained later, the fractions are weighted by the amount of stake the accounts hold in database state $\mathbf{DB}$.

Winkle operates by accounts voting for *checkpoints*, which are commitments to blocks in the VBCR. Denote by $\mathbf{q}$ the fractional voting power (the quorum) required to create a checkpoint. For *safety* of the system, the eclipsed accounts and the adversarial accounts should not constitute enough voting power to create a forking chain: $f_{\mathbf{E}} + f_{\mathbf{B}} < \mathbf{q}$. For *liveness* byzantine and active users together should constitute a quorum: $f_{\mathbf{A}} + f_{\mathbf{B}} \geq \mathbf{q}$ (we assume byzantine nodes participate in the honest chain but also equivocate to participate in an adversarial fork). The three equations above can be satisfied in multiple ways, giving a trade-off between liveness and safety of the system. At a high level we want to minimize the voting power

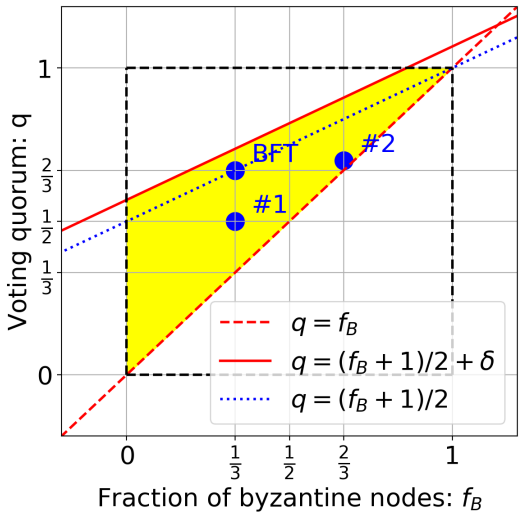


Figure 2: Generic trade-off between the fraction of byzantine users and the quorum size.

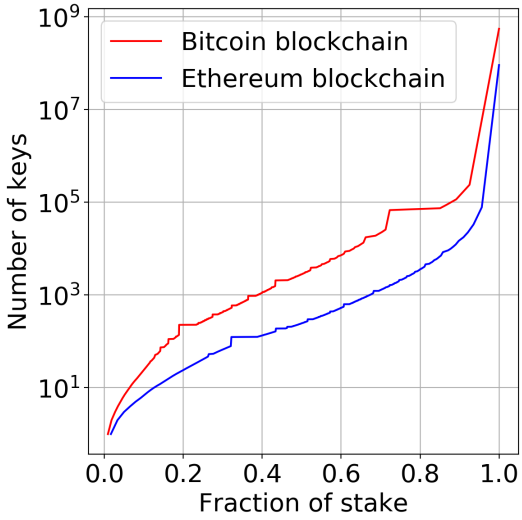


Figure 3: The number of keys holding a given fraction of stake in logarithmic scale.

needed to advance the checkpoint: $\mathbf{q} \to$ min and we want to maximize the number of byzantine and/or eclipsable nodes that the system can tolerate: $f_\mathbf{B}, f_\mathbf{E} \to$ max. We now discuss different solutions to this system.

Typical BFT. In BFT systems typically the trade-offs are chosen in the following way: $\mathbf{q} = 2/3$, $f_\mathbf{A} = 1/3 + \delta$, $f_\mathbf{B} = 1/3 - \delta$, $f_\mathbf{E} = 1/3$ (where δ represents a single user's fraction). More precisely: $N = 3f + 1$: $\mathbf{q} = (2f + 1)/N$; $f_\mathbf{A} = (f + 1)/N$; $f_\mathbf{B} = f_\mathbf{E} = f/N$; $f_\mathbf{D} = 0$. Note that this solution also leads to quorum intersection, where any two $\mathbf{q}$ quorums are guaranteed to intersect at some honest node. The solution also maximizes the portion of eclipsable users.

Generic trade-off. The BFT assumption is very strong: the adversary needs to compromise $1/3$ of all the coin holders and eclipse $1/3$ of coin holders from the network and make them believe the adversarial fork. It is based on the assumption that the network is adversarial and may separate honest users. In practice, it is quite unlikely that a user will connect to the chain through a fully adversarial network, and account holders are far more likely to query multiple validators about the state of the chain. So it is far more difficult to eclipse the accounts from the network [24], therefore we can lower the bound on the eclipsed accounts and get a generic trade-off between the number of byzantine users and the quorum size as illustrated by the yellow-filled area on Fig. 2. The selected points illustrate: Typical BFT solution (point "BFT", Fig. 2); the relaxed requirement on the quorum size: $f_\mathbf{B} = 1/3 - \delta$, $\mathbf{q} = 1/2$, $f_\mathbf{A} = 1/2 + \delta$, $f_\mathbf{E} = 1/6$ (point #1, Fig. 2); and the increased resilience to byzantine users (assuming those are not incentivised to break liveness): $f_\mathbf{B} = 2/3 - \delta$, $\mathbf{q} = 2/3$, $f_\mathbf{A} = 1/3 + \delta$, $f_\mathbf{E} \approx \mathsf{negl}$ (point #2, Fig.2).

Weighting accounts by value. To prevent Sybil attacks, where an adversary creates many empty accounts in order to gain control over more than $f_\mathbf{B}$ fraction of accounts at no cost, we weight accounts by the amount of coins that they hold — which we call *stake*. For the rest of the paper when we talk about accounts' fractions (e.g. $f_\mathbf{B}$, q), we implicitly assume that those accounts are weighted. Fig. 3 illustrates the number of keys protecting a given fractions of stake for both Bitcoin and Ethereum blockchains: $2/3$ of stake in Bitcoin is protected by 14,800 keys and same fraction of stake for Ethereum is protected by 800 keys (snapshot taken on 2019-08-01). Note that the flexibility of the assumption may be used to enlarge the bound on byzantine stake $f_\mathbf{B}$ thus increasing the number of keys required to be compromised in order to mount the long-range attack.

Limiting transfers. Due to transactions being processed the set of users and their stake is in constant flow. The adversary can potentially manipulate this flow in its fork and accumulate more than $f_\mathbf{B}$ stake in its chain, for example, by including all transactions where it is the recipient but no transactions where it is

the sender of coins. To mitigate this risk we additionally assume that at any state of the honest database the adversary controls at most f_y of accounts and that the amount of stake moving during any given block is at most f_x of the total amount of stake in the system. We then choose $f_y + f_x/2 = f_{\mathbf{B}}$ to make sure that the adversary can not accumulate more than $f_{\mathbf{B}}$ coins in his fork (see Section 5 (Lemma 1)). For example we could choose $f_y = 1/4$ and $f_x = 1/6$ or $f_y = 1/6$ and $f_x = 1/3$ depending on the power we wish to give to the adversary. For reference, based on simulations on Ethereum, one sixth of the money moves in roughly a week (3 days in Bitcoin) and one third in two weeks (one week for Bitcoin). This is consistent with the block time frames we consider.

Minting. Note that the adversary is only allowed to control $f_y S$ stake, where S is the total amount of stake in the system. But once the minting transaction takes an effect and increases the amount of stake to $S' = S + M$, the adversary is allowed to control more stake: $f_y S' > f_y S$. In our mechanism we require that minting happens only after the minting intent gets checkpointed: suppose that the minting intent was issued in block B_i and the block B_i got checkpointed in block B_j, where $j > i$, then the total stake increases from S to S' starting in block B_{j+1}. This means that by the end of block B_j, in state $\mathbf{DB}_j$, the adversary is still only allowed to hold at most $f_y S$ stake compromised.

5 Security Definition and Argument

We provide a game-based definition for the security of Winkle that closely mimics the interactive nature of the attacks. Without loosing generality we assume that there is a single validator, and that they use a fresh key to sign each new block. The adversary may adaptively compromise the users' accounts even retrospectively, but can not hold more than f_y of accounts' keys (and stake) at any given block, we define this requirement formally below. The adversary can submit transactions to be included in the next block and if the next block advances the confirmed checkpoint, then the adversary gets all the validator's keys prior to this newly confirmed checkpoint. The adversary wins if it outputs a forking chain of blocks, whose tip, B^*, generates a confirmed checkpoint inconsistent with the honest chain. We now put the game into more formal terms.

Definition 3. *A validator based consensus with reconfiguration* VBCR *is secure against* eventually compromised validators *if for any genesis state* B_0 *and for any polynomially bounded challenger who holds all the secret keys in the system and outputs valid answers for adversarial queries, for any probabilistic polynomial-time adversary* $\mathcal{A}$ *, there is a negligible function* $\nu(\lambda)$ *such that*

$$\Pr(\mathsf{EXP}_{\mathsf{VBCR},\mathcal{A}}(\lambda) = 1) \leq \nu(\lambda), \text{ for } \lambda \in \mathbb{N}.$$

The experiment $\mathsf{EXP}_{\mathsf{VBCR},\mathcal{A}}(\lambda)$ *is defined as the interaction with a challenger, where the challenger is comprised of two algorithms: key retrieval* GetKey *and next block creation* $\mathsf{NextBlock}$*, these are stateful algorithms that implicitly take the chain built up to this point as an input.*

$\underline{\mathsf{GetKey}(\mathsf{A}, i[, \mathbf{B}])}$ *outputs the account key: let* $\mathbf{DB}_i$ *be a state of the database built using the first* i *blocks of* $\mathbf{B}$*,* $\mathsf{sk} \leftarrow \mathbf{DB}_i[\mathsf{A}].\mathsf{sk}$*,* $Q = Q \cup \{(i, \mathsf{A}, \mathsf{sk})\}$*, output* sk*.*

$\underline{\mathsf{NextBlock}(\mathbf{T}^*_i[, \mathbf{B}])}$ *creates and outputs the next block and the secret keys of the validators used prior to the highest confirmed checkpoint: let the next block include adversarial transactions* $B_i \leftarrow [h_{i-1}, \mathbf{T}_i || \mathbf{T}^*_i, \sigma]$*, output* B_i *and the validator's keys:* $\{\mathsf{sk}_{t<\mathsf{HighestCheckpoint}(B_i).\mathsf{seq}}\}$

$\underline{\mathsf{EXP}_{\mathsf{VBCR},\mathcal{A}}(\lambda)}$ *experiment runs as follows starting with an empty* $\mathsf{state} = \emptyset$ *and* $i = 0$*:*

1. *the adversary creates transactions to inject:* $(\mathbf{T}^*_i, \mathsf{state}) \leftarrow \mathcal{A}_1^{\mathsf{GetKey}(\cdot,\mathsf{B}_i)}(\mathsf{B}_{i-1}, \mathsf{state})$*,*
2. *the adversary gets the next block:* $\mathsf{B}_i := \mathsf{NextBlock}(\mathbf{T}^*_i, \mathsf{state})$*,*
3. *the adversary decides if his forgery is ready:* $(b, \mathsf{state}) \leftarrow \mathcal{A}_2(\mathsf{B}_i, \mathsf{state})$*,*
4. *IF* $b = 1$*, THEN* $i \leftarrow i + 1$ *and GOTO (1),*
5. $\mathsf{B}^* \leftarrow \mathcal{A}_3(\mathsf{state})$ *and output 1 iff* $\mathsf{HighestCkpt}(\mathsf{B}^*)$ *is inconsistent with* B_i *and in the adversarial queries* Q *the total amount of stake in control of the adversary never exceeds* f_y*.*

Theorem 1. *The validator based consensus with reconfiguration* VBCR *as defined in Section 2 is secure against* eventually compromised validators *as per Definition 3.*

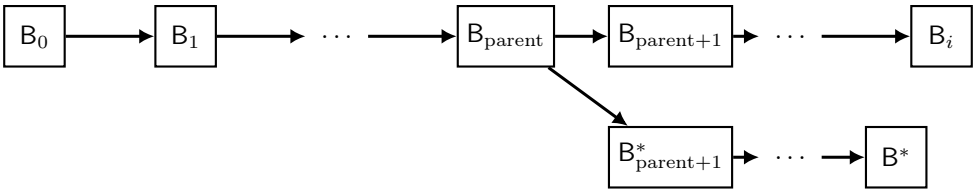
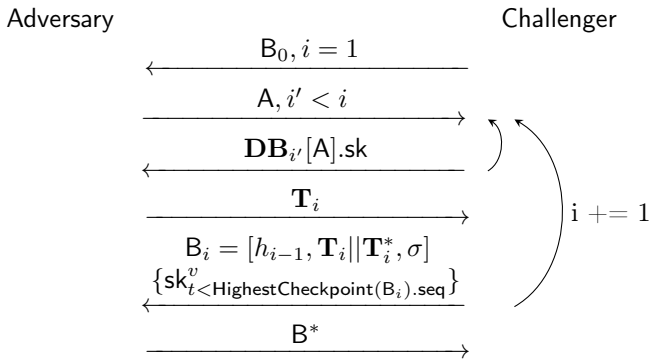


Figure 4: Illustration of an adversarial fork in an attempt to perform a long-range attack

Informally, the adversary plays the following game:



The formal security definition and the full description of the game can be found in Section 5. We first prove the following Lemma.

Lemma 1. *For any fraction f, if an adversary is only allowed to compromise keys that control at most f fraction of coins in each of the two states of the database $\mathbf{DB}_1$ and $\mathbf{DB}_2 = \mathbf{T}(\mathbf{DB}_1)$ (here $\mathbf{DB}_2$ is the result of applying the sequence $\mathbf{T}$ of transactions to $\mathbf{DB}_1$) and $\mathbf{T}$ moves at most p amount of stake between the keys, then for any sub-sequence of transactions $\mathbf{T}^* \subseteq \mathbf{T}$ the amount of stake in hands of the adversary in $\mathbf{DB}^* = \mathbf{T}^*(\mathbf{DB}_1)$ is at most $f + p/2$.*

Proof. Suppose in $\mathbf{DB}_1$ the adversary controls $f - \Delta$ of coins for some $\Delta \geq 0$. Suppose that in the sequence of transactions $\mathbf{T}$ adversarial addresses send $\mathcal{S}$ amount of coins and receive $\mathcal{R}$ amount of coins. To accumulate the maximum amount of stake in $\mathbf{T}^*$, the adversary needs to include transactions it receives and to drop transactions that it is sending out. Therefore the adversary will be able to accumulate at most $\mathsf{max}_{\mathsf{amount}} = (f - \Delta) + \mathcal{R}$. We consider two different cases. In the first case let's assume that the adversary receives less than Δ stake: $\mathcal{R} \leq \Delta$, then we trivially have $\mathsf{max}_{\mathsf{amount}} \leq f \leq f + p/2$.

In the second case we assume $\mathcal{R} > \Delta$. Let's write $\mathcal{R} = \Delta + \delta$ for some $\delta > 0$. In $\mathbf{DB}_2$ the adversary cannot hold more than f and thus we have: $(f - \Delta) + (\Delta + \delta) - \mathcal{S} \leq f$. Which in turns implies $\delta - \mathcal{S} \leq 0$, adding $(\Delta + \delta)$ to both parts equivalently get (*) $\Delta + 2\delta \leq \mathcal{S} + \Delta + \delta$. Since no more than p stake can move in $\mathbf{T}$, we have $\mathcal{R} + \mathcal{S} \leq p$, therefore (**) $\Delta + \delta + \mathcal{S} \leq p$. Inequalities (*) and (**) gives us that $\Delta + 2\delta \leq p$ or equivalently $\delta \leq \frac{p-\Delta}{2}$ (***).

Now going to $\mathsf{max}_{\mathsf{amount}}$, we have: $\mathsf{max}_{\mathsf{amount}} = f - \Delta + \Delta + \delta = f + \delta \leq f + \frac{p-\Delta}{2} \leq f + \frac{p}{2}$ by (***). And hence $\mathsf{max}_{\mathsf{amount}} \leq f + p/2$, therefore the adversary cannot obtain more than $f + p/2$ in its sub-sequence of transactions $\mathbf{T}^*$. □

Proof of Theorem 1. Suppose that the adversary has won the game and successfully produced a forgery B^*, s.t. $\mathsf{ckpt}^* := \mathsf{HighestCkpt}(\mathsf{B}^*)$ is inconsistent with B_i. Denote the latest common ancestor block of B^* and B_i as $\mathsf{B}_{\text{parent}}$. The adversary has the validator's key for $\mathsf{B}_{\text{parent}}$ to sign the conflicting descendant block creating a fork. By definition of the checkpoints at least $\mathbf{q}$ of weighted accounts in the adversarial chain has voted for ckpt^* or some later one. We prove that this event contradicts at least one of our assumptions. We do so by first considering a very simple variant of our system (Case 1) and incrementally adding additional features (*i.e.* key rotations, minting, delegation) (Case 2-4).

Recall some of the important details of Winkle's design described in Sections 3:

- Assumption 1: more than $1 - f_y$ keys are safe in the beginning of the block (the key is *safe* if all the keys under that account are not compromised within the block),

- Assumption 2: at most f_x stake moves within a block,

- Assumption 3: every transaction is included in some honest block only if it votes for the latest checkpoint (the previous most recent block).

We also assume that $f_y + f_x/2 = f_{\mathbf{B}}$.

Case 1. We first consider a simplified variant of our system with no key rotations (accounts never rotate keys and new account are not created), no delegation (no accounts have delegates), no minting (new coins are never minted, the only coins in the system are those created in B_0).

Denote the first pair of diverging blocks $\mathsf{B}_{\text{parent}+1}$ and $\mathsf{B}^*_{\text{parent}+1}$. By design of the system (assumption 3), transactions can be sequenced in the block only if these transactions vote on the previous immediate block, thus only a subset of transactions from non-adversarial accounts in $\mathsf{B}_{\text{parent}+1}$ can be replayed in the fork as only those sign $\mathsf{B}_{\text{parent}}$ as a checkpoint, moreover these transactions can only get placed in the block $\mathsf{B}^*_{\text{parent}+1}$. No other transactions from the honest chain can be replayed in the adversarial fork. Having that under Assumption 1 the adversary can control at most f_y keys and eclipse at most $f_{\mathbf{E}}$ keys and by Assumption 2 in block $\mathsf{B}_{\text{parent}+1}$ at most f_x of stake moves, by applying Lemma 1, we get that there are no subset of transactions of the honest block $\mathsf{B}_{\text{parent}+1}$ such that if they are applied to $\mathbf{DB}_{\text{parent}}$ more than $f_{\mathbf{E}} + f_y + f_x/2 = f_{\mathbf{E}} + f_{\mathbf{B}}$ of weight gets concentrated jointly under the compromised and eclipsed accounts in the resulting state of the database. Thus since $f_{\mathbf{E}} + f_{\mathbf{B}} < \mathbf{q}$ the adversary will be able to get more than $\mathbf{q}$ fraction of stake and thus will not be able to create a new checkpoint on a forking chain. Therefore the forgery is not possible and moreover the eclipsed users can distinguish an adversarial chain from a true chain by the fact that new blocks do not become confirmed checkpoints on an adversarial chain.

Case 2: Case 1 + key rotation. We now prove the theorem in the presence of key rotations. Let's recall that our assumption in that case is that $1 - f_y$ of the accounts are safe between two checkpoints, meaning that none of the keys from those accounts are ever leaked to the adversary during any given block.

Let's assume that the adversary has corrupted f_y accounts within block $\mathsf{B}_{\text{parent}+1}$. To corrupt a new account, A, account A has to perform a key rotation operation in block $\mathsf{B}_{\text{parent}+2}$ otherwise this account would be considered unsafe in block $\mathsf{B}_{\text{parent}+1}$ as well, making the number of unsafe accounts go above the allowed threshold f_y. Due to assumption 3, the key rotation for account A will include a vote for $\mathsf{B}_{\text{parent}+1}$, making it impossible to replay this key rotation transaction in the adversarial fork. This ensures the adversary cannot control more than f_y in any block since it cannot accumulate corrupted accounts from multiple blocks (see Figure 5 for a visual illustration).

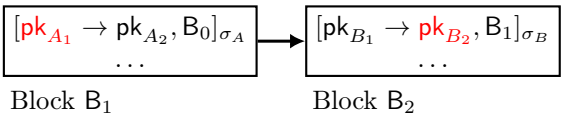


Figure 5: The adversary that knows pk_{A_1} and pk_{B_2} cannot use both keys in its forking chain because the second key rotation includes a commitment to block B_1 which include the first key rotation, healing the compromised key.

Case 3: Case 2 + delegation. We now prove the result in the case where delegation is enabled. As a reminder, in the case of delegation, we assume that honest users delegate to honest delegates (assumption 4). Additionally, we require that each delegate is active (as they are incentivized to be).

The adversary cannot control more than f_y in the honest chain by assumption 1 and 4 and cannot receive more than $f_x/2$ of the coins by a similar argument as in Lemma 1. The main difference between the previous case is that now, eclipsed users may vote on the adversarial chain, but have their delegate vote on the honest chain and thus equivocate. Even in that case, under the assumption that no more than $f_{\mathbf{E}}$ users are eclipsed, we still have that the adversary can get at most $f_{\mathbf{B}} + f_{\mathbf{E}} < \mathbf{q}$ of the vote in its chain.

Similarly as in case 2, if the adversary was to corrupt a previously honest delegates, then they will have to make a key rotation that includes a vote for the latest checkpoint, preventing the adversary, as before, from accumulating stake over many blocks.

Case 4: Case 3 + Minting. We now prove that an adversary cannot obtain more than f_y of the stake

in its chain even when money can be minted. As explained in Section 3, the minting transaction is only effective once the block that contains it has been checkpointed. We will thus refer to the mint transaction as a *mint-intent* transaction.

If the minting key has been leaked to the adversary, the adversary can include in its chain a mint intent that creates money for itself. However as per Cases 1-3 of this proof the adversary cannot checkpoint the alternative minting intent using the old stake. Though the adversary can not create a minting transaction of its own it can potentially take advantage of omitting the minting transaction which we explore next.

Note that the adversary is only allowed to control $f_y S$ stake, where S is the total amount of stake in the system. But once the minting transaction takes an effect and increases the amount of stake to $S' = S + M$, the adversary is allowed to control more stake: $f_y S' > f_y S$. We now show that the adversary can not leverage this fact, omit the minting transaction and hold compromised more than $f_y S$ of stake in its forking chain.

Indeed, suppose that the minting intent was issued in block B_i and the block B_i got checkpointed in block B_j, where $j > i$. If the adversary tries to omit the checkpointing of the minting intent, it has to modify block B_j or an earlier one and therefore can not include any of the transactions from the subsequent blocks $\mathsf{B}_{>j}$ of the honest chain, but only those later transactions (e.g. key rotations) actually allow the adversary to compromise more stake, if the adversary can not leverage those, it can not accumulate more than $f_y S$ stake.

In more details, after the minting intent got checkpointed the adversary can either keep the same accounts compromised (and receive more money on it) or compromise new accounts. In the first case, the additional stake that the adversary receives in order to own more than $f_y S$ of the stake must happen after B_j and thus, by assumption 3, this stake includes a vote for checkpoint B_j. This bounds the adversary to mint new coins. Thus if the adversary wants to increase its stake it has to include the minting transactions and cannot inflate its relative stake. Similarly, if the adversary compromises new accounts after B_j, then by assumption these new accounts need to be safe at state B_j and thus have operated a key rotation after checkpoint B_j. This is because otherwise these accounts would be unsafe in the previous block which contradicts our assumption. Thus these new accounts that the adversary compromises also have to include a vote for B_j by a similar argument as before. Therefore, all the new stake that the adversary gets past the block B_j has to include a vote for B_j and therefore the adversary has to include the minting intent and the minting transaction to keep a valid chain. □

6 Evaluation of Checkpointing Delay

The key barrier to reducing Winkle's checkpoint confirmation delay is users' idleness, which is encouraged within some cryptocurrency communities[3]. We estimate the checkpoint confirmation delay (the *finality time*) of Winkle in the "real world", using workloads from the most popular cryptocurrency projects namely Ethereum and Bitcoin. We pre-process the transactions graph for these projects by assuming that each transaction in a block votes for the block that precedes it; i.e., if a transactions appears in block B_i, we consider that it votes for block B_{i-1}. For each block we then retroactively determine the confirmed checkpoint as the block that has just received $\mathbf{q}$ fraction of the votes. For our simulations we choose $\mathbf{q} = 2/3$ in line with the usual BFT assumptions. The *finality time* shows how long the block awaits to get checkpointed by $\mathbf{q}$ fraction of stake, *i.e.* the time validators need to stay honest.

We use Google BigQuery on the Ethereum [14] and the Bitcoin [28] blockchains to perform simulations. We build a database with all addresses (or accounts) and the date of their last transaction, which corresponds to their latest vote, voteSent. The value under the account right after the last sent transaction is counted towards the vote voteSent. We also list transactions received per account after the last sent transaction, we apply the propagation rule (see Sec. 3.1) and append them to the list of the account's votes. This gives us a global list of weighted votes. We rank the weighted votes by decreasing checkpoint's vote (i.e., most recent one first) and retroactively determine the highest confirmed checkpoint.

Fig. 6 plots the confirmed checkpoint delays for databases at different times. For Ethereum a checkpoint takes between 50 days and up to a year to be confirmed (with mean 204 days), and for Bitcoin a checkpoint takes between 4 months and up to three years to be confirmed (with mean 601 days).

[3]A practice known as HODLing `https://en.wikipedia.org/wiki/Hodl`

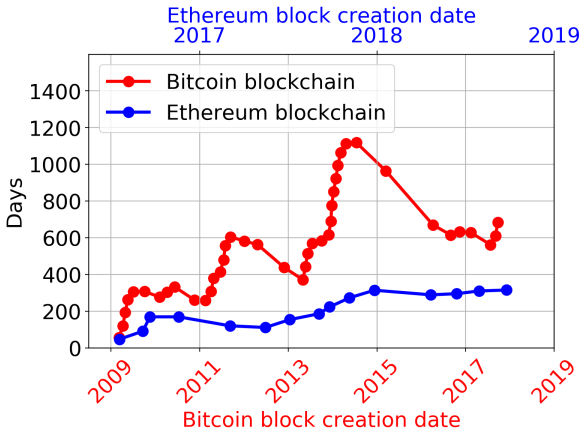


Figure 6: Checkpoint confirmation delay without delegates

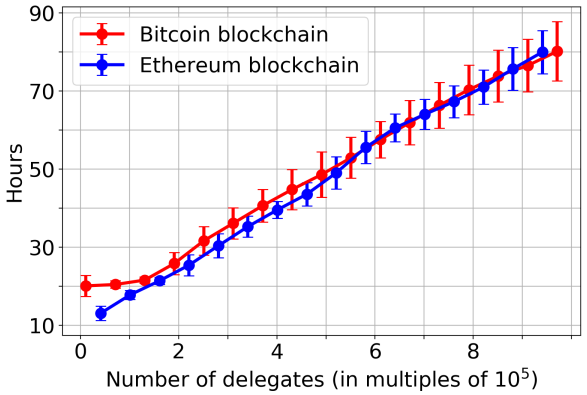


Figure 7: Checkpoint confirmation delay with different number of delegates

Delegation. As shown in Fig. 6, Winkle without delegation would lead to finality times of months or years. We simulate simple delegation to estimate the improvement in the finality time. In our delegation scheme, we incentivize the creation of k pools [5] of roughly equal weight. To average out monthly fluctuations, we take 12 measurements, plot the mean as a line and standard deviation as error bars on Fig. 7. For each month between Sep'18 and Aug'19 we choose delegates as the top k accounts that have been the most frequently transacting during that month. More precisely, these are the accounts (or addresses) whose maximum delay between two consecutive transactions in the given month is the smallest. We assume those as our delegates with equal voting power. We retroactively compute the latest checkpoint confirmed on the last day of the month, by taking the vote from the two third of delegates who sent the newest vote. The results, depending on the number of delegates k between a few and one million are presented in Fig. 7. To better understand the x-axis scale, note the number of transacting Ethereum accounts between Sep'18 and Aug'19 is 21 million and transacting distinct Bitcoin addresses is 113 million. As expected, the delegation scheme drastically reduces finality time: for one thousand delegates the finality time is about 4 hours, for 10 thousand delegates it is 20 hours, and for 100 thousand delegates it is 3.5 days.

7 Related Work

An abstract VBCR can be instantiated through a Byzantine agreement protocol such as PBFT [21], or HotStuff [26]. Systems such as Tendermint [6], SBFT [15], and Libra [3] implement such mechanisms and we have designed Winkle to be relevant to those designs. Long-range attacks do not occur in Nakamoto consensus based protocols such as Bitcoin [22] or Ethereum [25] that rely on Proof-of-Work, instead of validator signatures. However, those only offer probabilistic finality and Winkle may act as a finality layer.

Previous approaches to defeating long-range attacks. Two approaches have been proposed in the literature to foil long-range attacks, besides using secure hardware [19]. *Software checkpointing* [18] periodically includes in client software a 'checkpoint'. Clients never accept any past contradicting blocks. This relies on centrally trusting the software distribution process and can be modeled as a VBCR system, secured by software developers keys. These becoming known to the adversary allows for long-range attacks. The second approach is based on validators *locking deposits* [7, 27] that are not returned in case of equivocation. It is difficult to determine how long the deposits should be locked, after a validator becomes inactive. And locking deposits for any fixed time cannot prevent validators from losing their key after that period suffering no penalty.

Winkle can be interpreted as a refinement of those two techniques: it allows all to determine appropriate checkpoints in a highly decentralized manner, and allows validators that were honest to recover their deposits after their blocks become confirmed through a checkpoint.

Background and related techniques. The core of Winkle is based on confirming checkpoints using a simplified Byzantine Consistent Broadcast [16] based on stake. This primitive is always safe but may not lead to a checkpoint in case the initiator (the VBCR in our case) provides two equivocating checkpoints — hence we assume that validators are only eventually compromized.

Winkle borrows ideas from Proof-of-Stake systems proposed as early as 2012 in the context of the Ppcoin [18], and then Snow White [9] and Ouroboros [17]. Winkle makes similar security assumptions, namely that a fraction of the stake is controlled by honest clients. However, Winkle is not a consensus mechanism but rather a finality layer, similar to Casper [8]. Fantômette [1] and Afgjort [20] are finality layers that resemble Winkle; they rely however on validators rather than general users to provide finality. Ouroboros Genesis [2] has a finality layer that allows new parties to bootstrap the correct blockchain without checkpointing.

Another type of long-range attack, stake-bleeding attacks, have been proposed by Gazi et al [13]. An adversary works on its own chain and the adversary's stake grows due to block rewards whereas the honest stake stays still, until it overtakes it. Winkle addresses this attack as argued in Section 3.3.

A number of blockchains employ client-led validation. In Iota [23] clients include references to multiple past transactions. Consensus is secure if honest clients make more transactions than dishonest ones. EOS consensus [10] is based on clients delegating stake to one of a set of validators, and uses 'Transactions as Proof of Stake (TaPoS)' similar to Winkle. Winkle differs in that it is a finality layer immune to long-range attacks, rather than a full consensus protocol, and therefore makes weaker assumptions for its security.

8 Discussion

Winkle provides a finality layer based on account stake. It may also be adapted to provide finality to Proof-of-Work based blockchains, that otherwise can only achieve probabilistic finality. In such systems blocks are generated through a Proof-of-Work and a fork choice rule that privileges forks with the most work. Clients vote for the fork they consider authoritative, after sufficient time to constitute an epoch and to ensure they are likely to be correct. Once $\mathbf{q}$ of stake confirms a checkpoint all clients accept it, and no matter how much work an adversary commits in the future it may never be reverted.

A full Proof-of-Stake system may be bootstrapped through Winkle. Clients may indicate on-chain whether they are prepared to act as validators. At each block, a certain number of clients are selected that represent the most stake in the system, either directly or through delegation. Their stake at this point is locked, and they act as validators to produce a long-term block. Once the stake has shifted significantly, or after some number of blocks, they are rotated and a new set is selected. Stake is released once the last checkpoint they facilitated is confirmed by a fraction $\mathbf{q}$ of the stake as per Winkle.

9 Conclusions

Proof-of-Stake systems based on validators are a promising way to scale up blockchain, but are based on fundamentally different security assumptions to Proof-of-Stake. They are susceptible to validators eventually becoming compromised, and to a long-range attack. Winkle provides a decentralized approach for clients to determine and validate checkpoints, that rests only on the short term honesty of validators, and the longer term honesty of a more numerous set of all the stake holders. We show that using delegation, and assuming usage similar to Bitcoin or Ethereum, Winkle checkpoints with a delay of hours or a few days. Thus validators need only to keep their signature keys safe for this short window of time, assuming validators frequently rotate the keys. A stake locking mechanism incentivizing validators to stay honest can unlock stake after a confirmed checkpoint giving tolerable delays.

Finally, Winkle as presented, has a key shortcoming: we assume that an honest account votes for the latest checkpoint. However, this creates a boundary race condition between blocks; it represents a challenge for cold wallets; and it makes transactions in mempools invalid between blocks. We took a pragmatic approach and assumed those issues are alleviated by blocks representing periods that are relatively long — in the order of a day or more. However, a key open research question relates to relaxing this condition, while preserving the security and simplicity of the Winkle scheme.

Acknowledgements. The authors would like to thank Ben Maurer (Facebook Calibra) for suggesting naming the system after the popular story of Rip Van Winkle. In contemporary times he would have fallen asleep and missed the cryptocurrency boom, and upon waking up would have to do a full chain validation, subject to Long Range Attacks, to catch up with the world. The authors would also like to thank Kostas Chalkias for useful comments on the idea behind the project. Sarah Azouvi worked on Winkle while being a research intern with Facebook Calibra.

References

[1] S. Azouvi, P. McCorry, and S. Meiklejohn. Betting on blockchain consensus with fantomette. *arXiv preprint arXiv:1805.06786*, 2018.

[2] C. Badertscher, P. Gaži, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[3] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino. State machine replication in the libra blockchain. `https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf`, 2019.

[4] Bitcoin pools. `https://www.blockchain.com/pools`. Accessed: 2019-09-16.

[5] L. Brünjes, A. Kiayias, E. Koutsoupias, and A. Stouka. Reward sharing schemes for stake pools. *arXiv preprint arXiv:1807.11218*, 2018.

[6] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[7] V. Buterin. Slasher: A punitive proof-of-stake algorithm. Ethereum Blog URL: `https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/`, 2014.

[8] V. Buterin and V. Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[9] P. Daian, R. Pass, and E. Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *International Conference on Financial Cryptography and Data Security*. Springer, 2019.

[10] Eos.io technical white paper v2. `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`.

[11] Proof of stake faq. `https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ`.

[12] Replay attack protection: Include blocklimit and blockhash in each transaction. `https://github.com/ethereum/EIPs/issues/134`.

[13] P. Gaži, A. Kiayias, and A. Russell. Stake-bleeding attacks on proof-of-stake blockchains. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 2018.

[14] Google bigquery. `https://console.cloud.google.com/bigquery`.

[15] G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019.

[16] D. Imbs and M. Raynal. Simple and efficient reliable broadcast in the presence of byzantine processes. *arXiv preprint arXiv:1510.06882*, 2015.

[17] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO'17*. Springer, 2017.

[18] S. King and S. Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. https://www.peercoin.net/whitepapers/peercoin-paper.pdf, 2012.

[19] W. Li, S. Andreina, J. Bohli, and G. Karame. Securing proof-of-stake blockchain protocols. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2017.

[20] B. Magri, C. Matt, J. Nielsen, and D. Tschudi. Afgjort–a semi-synchronous finality layer for blockchains. 2019.

[21] C. Miguel and L. Barbara. Practical byzantine fault tolerance. In *OSDI*, 1999.

[22] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.

[23] S. Popov. The tangle. https://iota.org/IOTA_Whitepaper.pdf.

[24] A. Singh, T. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *IEEE INFOCOM'06*, 2006.

[25] G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger EIP-150 Revision. http://gavwood.com/paper.pdf, 2016.

[26] M. Yin, D. Malkhi, M. K Reiter, G. G. Golan, and A. Ittai. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[27] V. Zamfir. Introducing casper "the friendly ghost". Ethereum Blog URL: https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost, 2015.

[28] Zielak. Bitcoin historical data. https://www.kaggle.com/mczielinski/bitcoin-historical-data. processed via Google bigquery: https://console.cloud.google.com/bigquery.