Securing Proof-of-Work Ledgers via Checkpointing

Dimitris Karakostas, Aggelos Kiayias
Bitcoin’s novelties

- Hash chain +
- Proof-of-Work +
- Incentives for participation
Bitcoin’s novelties

- Hash chain +
- Proof-of-Work +
- Incentives for participation

\[
\text{Distributed ledger} \downarrow \quad \text{Open (decentralised) consensus}
\]
Proof-of-Work

- A *compute cycle* is one identity
- Limit the amount of identities per person
  - Cannot create more identities than CPU cycles one controls
  - Sybil protection
Proof-of-Work

● A compute cycle is one identity
● Limit the amount of identities per person
  ○ Cannot create more identities than CPU cycles one controls
  ○ Sybil protection
● Core security assumption: 50%+1 CPU cycles are honest
51% attacks are real
Overview

- How can checkpoints secure an insecure ledger?
  - Checkpointing ideal functionality
  - Security guarantees
  - Ethereum Classic analysis
  - The protocol that realizes checkpointing functionality

- Distributed checkpointing prototype implementation

- Timestamping: decentralizing checkpoints
Our goals

- Secure a ledger *temporarily* against 51% attacks
- Avoid trivializing the ledger maintenance
- Minimize storage/time overhead

Core idea

- Introduce an external set of parties to guarantee security
Preliminaries

- Fixed number of parties (n)
- Round-based execution
- All messages are delivered by the end of a round (*synchronous*)
- Block size is unlimited
Preliminaries (cont.)

- Each party has $q$ queries to a random oracle (*hashing power*)
- Each query is successful with probability $p$
- The adversary $A$:
  - controls $t$ parties (equiv. $\mu_A = t/n$ hashing power)
  - adaptive: corrupts parties on the fly
  - rushing: decides strategy after (possibly) delaying honest messages
Ledger properties

- **Stable transaction** \( \tau \): each honest party reports \( \tau \) in the same position in the ledger

- **Persistence**: a transaction in a block at least \( k \) blocks away from the ledger’s head is stable

- **Liveness**: a transaction which is continuously provided to the parties becomes stable after at most \( u \) rounds
Checkpointing functionality

- The *ideal* definition of checkpoints
- An omnipresent entity
- Expresses the needed security properties
Checkpointing functionality

- The *ideal* definition of checkpoints
- An omnipresent entity
- Expresses the needed security properties

**Functionality \( F_{\text{Checkpoint}} \)**

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

Upon receiving \((\text{CANDIDATECHECKPOINT}, C')\) from a party \( \mathcal{V} \), forward it to \( \mathcal{A} \). Upon receiving \((\text{CANDIDATECHECKPOINT}, C')\) from \( \mathcal{A} \), if \( C_c < C' \) set \( C := \maxvalid(C, C') \). Next, if \( |C \setminus C_c| = k_c \) compute a list \( R \) of \( |\mathcal{V}| \) random values as \( r_j \overset{\$}{\leftarrow} \{0, 1\}^\omega \) and send \((\text{NONCE}, R)\) to \( \mathcal{A} \). Upon receiving from \( \mathcal{A} \) a response \((\text{NONCE}, R')\), such that \( R' \) is a list of at least \( \frac{|\mathcal{V}|}{2} \) values from \( R \), pick a value \( r_i \in R' \), return \((\text{CHECKPOINT}, \text{tail}(C)||r_i)\) to \( \mathcal{V} \) and set \( C := C_c := C||r_i \).
Security of the checkpointed ledger

Persistence

(a transaction in a block at least $k$ blocks away from the ledger’s head is stable)
Persistence

- $k$ (persistence parameter) $\geq k_c$ (checkpoint interval)
- At least one in the last $k$ blocks is a checkpoint
- Checkpoints cannot be reverted
- All blocks up to the last checkpoint are stable
Security of the checkpointed ledger

Liveness

(a transaction which is continuously provided to the parties becomes stable after at most $u$ rounds)
Liveness

- If an honest block B gets checkpointed after a transaction $\tau$ is created, then $\tau$ becomes stable
  - Proof: if $\tau$ is not in any block prior to B, then B will include it (because honest parties include all unpublished transactions and blocks are unlimited)

- Creating checkpoints is not enough; they need to be put in the chain
Front-running: An attack against liveness
Liveness analysis

- Separate the honest from the adversarial parties
- Argue about security wrt. honest parties (regardless of adversarial strategy)
- Stochastic Markov chain for protocol execution modelling
Liveness Markov chain

- Each state is identified by (i, j):
  - i: the number of blocks an honest party needs to produce to reach the next checkpoint
  - j: the number of blocks the adversary necessarily needs to produce to reach the next checkpoint

- 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 $i$ blocks at a given round, then $M^{(i)} = 1$, else $M^{(i)} = 0$

- Expectations:
  - $E(H) = h = 1 - (1-p)^{(n-t)}$
  - $E(M^{(i)}) = m^{(i)} = \binom{q}{i} \cdot p^i \cdot (1-p)^{qt-i}$

- Transition probabilities ($b \geq 0$):
  - To (i, j - b): $(1 - h) \cdot m^{(b)}$
  - To (i - 1, j - b): $h \cdot m^{(b)}$
Liveness Markov chain ($k_c = 1$)
Markov chain properties

- Stochastic transition matrix: matrix that defines the transition probabilities between two states

- Canonical form: $M = \begin{pmatrix} Q & R \\ 0 & I_r \end{pmatrix}$ (Q: transition states, R: absorption states)

- Probability of transition from $s_i$ to $s_j$ after $u$ rounds: $ij$-th column of $M^u$

- Expected number of steps before absorption: $t = \left[ \sum_{j=0}^{t} N_{ij} \right], (I - Q)^{-1} = N$
Liveness of a checkpointed Ethereum Classic

Liveness probability for 51% adversary
Liveness of a checkpointed Ethereum Classic

Expected number of steps before absorption
The checkpointing protocol

- Parameterized by a *fail-stop* protocol $\pi_{fs}$
- Every $k_c$ blocks:
  - Pick a random nonce (e.g., randomized signature)
  - Run $\pi_{fs}$ to agree on checkpoint
  - Append nonce to chosen block
The checkpointing protocol

**Protocol \( \pi_{\text{Checkpoint}} \)**

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

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

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

If all hold do:
1. pick \( r_j \leftarrow \{0,1\}^\mu \);
2. pick input \( \langle C', r_j \rangle \) for the protocol \( \pi_{\text{FS}} \);
3. execute \( \pi_{\text{FS}} \) with the parties in \( \mathcal{V} \) to agree on an input \( \langle C'', r' \rangle \), such that \( \forall (\hat{C}, \hat{r}) \in I : \text{maxvalid}(C'', \hat{C}) = C'' \) with \( I \) the set of inputs, i.e. choose the output according to \( \text{maxvalid} \);
4. set \( B_c := \text{tail}(C')||r' \).

Finally, return \( (\text{Checkpoint}, B_c) \) to \( \mathcal{V} \).
Proof strategy

- Show that ideal and real worlds are indistinguishable

**Functionality \( F_{\text{Checkpoint}} \)**

\( F_{\text{Checkpoint}} \) interacts with a set of parties \( V \) and holds the local chain \( C \) and the checkpoint chain \( C_e \), both initially set to \( \epsilon \). It is parameterized by \( k_e \), which defines the number of blocks between two consecutive checkpoints, and the maxvalid(\cdot, \cdot) algorithm.

Upon receiving \((\text{CANDIDATE-CHECKPOINT}, C')\) from a party \( V \), forward it to \( A \). Upon receiving \((\text{CANDIDATE-CHECKPOINT}, C')\) from \( A \), if \( C_e < C' \) set \( C := \text{maxvalid}(C, C') \). Next, if \(|C \setminus C_e| = k_e \), compute a list \( R \) of \(|V|\) random values as \( r_j \overset{\$}{\leftarrow} \{0, 1\}^n \) and send \((\text{NONCE}, R)\) to \( A \). Upon receiving from \( A \) a response \((\text{NONCE}, R')\), such that \( R' \) is a list of at least \(|V|\) values from \( R \), pick a value \( r_i \in R' \), return \((\text{CHECKPOINT}, \text{tail}(C)||r_i)\) to \( V \) and set \( C := C_e := C||r_i \).

**Protocol \( \pi_{\text{Checkpoint}} \)**

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

Upon receiving \((\text{CANDIDATE-CHECKPOINT}, C')\) from a party \( V \), check:

- \( \exists i : C'[i] = B_e \) (i.e. if \( C' \) extends the checkpoint);
- \( \text{Validate}(C') = 1 \) (i.e. if \( C' \) is valid);
- \(|C' - i| = k_e \) (i.e. if \( C' \) is long enough).

If all hold do:

1. pick \( r_j \overset{\$}{\leftarrow} \{0, 1\}^n \);
2. pick input \((C', r_j)\) for the protocol \( \pi_{FS} \);
3. execute \( \pi_{FS} \) with the parties in \( V \) to agree on an input \((C', r')\), such that \( \forall (C', r) \in 1 : \text{maxvalid}(C', C') = C' \) with \( \| \) the set of inputs, i.e. choose the output according to \( \text{maxvalid} \);
4. set \( B_e := \text{tail}(C')||r' \).

Finally, return \((\text{CHECKPOINT}, B_e)\) to \( V \).
Chain decision using checkpoints

- Every $k_c$ blocks, send the last block to checkpoint authority
- Retrieve checkpoint, append it to the chain, and then keep mining

---

Protocol $\pi_{\text{CheckpointMiningRes}}$

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

Upon receiving $(\text{CANDIDATECHAIN}, C')$ do:

- if $|\text{chains}| - |C'| < k_c$ set $C := \text{maxvalid}(C, C')$
- else set $i_c := i_c + k_c$ and send $C'[i_c]$ to all parties in $V$. Upon receiving $\left\lceil \frac{n}{2} \right\rceil$ messages $(\text{CHECKPOINT}, B||r)$ from different checkpointing parties, if $C'[i_c] = B||r$ set $C := C'$, else if $C'[i_c] = B$ set $C := C'[i_c]$.

Upon receiving $(\text{READ})$ return $(\text{CHAIN}, C)$.
Prototype implementation

- PKI for checkpointing nodes
- 15 Amazon EC2 t2.micro nodes
- Raft: fail-stop consensus protocol
- $k_c = 4$
- Checkpoints are aggregated signatures
- Test blockchain: Private Ethereum Proof-of-Authority
Prototype evaluation

Storage (size of checkpoints):

- $8 \text{ (nodes)} \cdot 64 \text{ (bytes of a single signature)} = 512 \text{ bytes}$
- $0.6\%$ increase in ledger’s size
Prototype evaluation

Latency

(time between retrieval of block and issuing of signed checkpoint)

- London (EU): 557 ms
- N. California (US West): 620 ms
- São Paulo (South America): 711 ms
- Tokyo (Asia Pacific): 723 ms
- Singapore (Asia Pacific): 779 ms
Timestamps: Decentralized checkpoints

**Functionality $F_{Timestamp}$**

$F_{Timestamp}$ holds the following items:

- $T[]$: an initially empty list of timestamped strings;
- $\tau$: a counter initially set to 0;

Upon receiving $(\text{TIMESTAMP}, s)$, if $\forall (s', \cdot) \in T[]: s' \neq s$, set $\tau := \tau + 1$ and add $(s, \tau)$ to $T[]$.

Upon receiving $(\text{VERIFY}, s, \tau)$, if $\exists (s, \tau) \in T[]$ then return $(\text{VERIFY\_TIMESTAMP}, \top)$. 
Chain decision using timestamps

Diagram:
- Name: a, Time: 0
- Name: b, Time: _
- Name: c, Time: 1
- Name: d, Time: 2
- Name: e, Time: 3
- Name: f, Time: 4
Timestamping security

- Security guarantees: Same as checkpoints with $kc = 1$
- Timestamping *every* block is important:
  - A chain segment that follows a non-timestamped block can be removed in the future
- The entire block header needs to be timestamped:
  - Timestamping a hash is not enough, as the adversary can keep a timestamped block secret
## Decentralized timestamping

<table>
<thead>
<tr>
<th>Cost</th>
<th>Ethereum</th>
<th>Bitcoin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Smart contract deployment</td>
<td>BTC* header timestamping</td>
</tr>
<tr>
<td></td>
<td>0.4$</td>
<td>0.07$</td>
</tr>
<tr>
<td></td>
<td>BTC* header timestamping</td>
<td>ETH* header timestamping</td>
</tr>
<tr>
<td></td>
<td>0.45$</td>
<td>3.6$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Latency</th>
<th>Ethereum</th>
<th>Bitcoin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stable timestamp</td>
<td>Stable timestamp</td>
</tr>
<tr>
<td></td>
<td>Unstable timestamp</td>
<td>Unstable timestamp</td>
</tr>
<tr>
<td></td>
<td>9 minutes</td>
<td>60 minutes</td>
</tr>
<tr>
<td></td>
<td>15 seconds</td>
<td>10 minutes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proof size</th>
<th>Ethereum</th>
<th>Bitcoin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full node</td>
<td>Full node</td>
</tr>
<tr>
<td></td>
<td>SPV implementation</td>
<td>SPV implementation</td>
</tr>
<tr>
<td></td>
<td>NIPoPoW implementation</td>
<td>NIPoPoW implementation</td>
</tr>
<tr>
<td></td>
<td>FlyClient implementation</td>
<td>FlyClient implementation</td>
</tr>
<tr>
<td></td>
<td>181 GB</td>
<td>240 GB</td>
</tr>
<tr>
<td></td>
<td>5 GB</td>
<td>48 GB</td>
</tr>
<tr>
<td></td>
<td>6 MB</td>
<td>3 MB</td>
</tr>
</tbody>
</table>
Future work

- Byzantine Fault Tolerant checkpointing service
- Randomized checkpointing (intervals)
- Non-rushing adversaries
- Non-interactive (but centralised) timestamping
- Checkpoints for Proof-of-Stake
Conclusion

● In case of adversarial majority, an external set of honest parties needs to be introduced

● Checkpoints need to become part of the chain to ensure liveness
  ○ Front-running attack

● Checkpoints can be decentralized via distributed ledger-based timestamping

Thank you!