A Decentralized Blockchain with High Throughput and Fast Confirmation

Chenxing Li, Peilun Li, and Dong Zhou, *Tsinghua University;* Zhe Yang, Ming Wu, and Guang Yang, *Conflux Foundation;* Wei Xu, *Tsinghua University;* Fan Long, *University of Toronto and Conflux Foundation;* Andrew Chi-Chih Yao, *Tsinghua University*

https://www.usenix.org/conference/atc20/presentation/li-chenxing

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

A Decentralized Blockchain with High Throughput and Fast Confirmation

Chenxing Li*, Peilun Li*, Dong Zhou, Zhe Yang†, Ming Wu†,
Guang Yang†, Wei Xu, Fan Long‡†, Andrew Chi-Chih Yao
Tsinghua University †*Conflux Foundation* ‡*University of Toronto*

Abstract

This paper presents Conflux, a scalable and decentralized blockchain system with high throughput and fast confirmation. Conflux operates with a novel consensus protocol which optimistically processes concurrent blocks without discarding any as forks and adaptively assigns weights to blocks based on their topologies in the Conflux ledger structure (called Tree-Graph). The adaptive weight mechanism enables Conflux to detect and thwart liveness attack by automatically switching between an optimistic strategy for fast confirmation in normal scenarios and a conservative strategy to ensure consensus progress during liveness attacks.

We evaluated Conflux on Amazon EC2 clusters with up to 12k full nodes. The consensus protocol of Conflux achieves a block throughput of 9.6Mbps with 20Mbps network bandwidth limit per node. On a combined workload of payment transactions and Ethereum history transactions, the end-to-end system of Conflux achieves the throughput of up to 3480 transactions per second while confirming transactions under one minute.

1 Introduction

Following the success of cryptocurrencies [2, 23], blockchain has evolved into a technology powering secure, decentralized, and consistent transaction ledgers at Internet-scale. Newer blockchain platforms such as Ethereum [2, 35] support customized transaction rules as smart contracts, which greatly extend the capability of blockchain ledgers beyond value transfers.

Blockchain platforms like Bitcoin [23] use *Nakamoto consensus*. It organizes transactions into an ordered list of blocks, each of which contains multiple transactions and a link to its predecessor. Participants (miners) solves *proof-of-work* (PoW) puzzles to compete for the right of generating the next block. To prevent an attacker from reverting previous transactions, honest participants agree on the longest chain of blocks as the correct history. Each new block is appended at the end of the longest chain to make the chain longer and therefore harder to revert.

However, the performance remains one of the most critical issues of blockchains. Nakamoto consensus is bottlenecked by its slow block generation rate. For example, Bitcoin generates one 1MB block every 10 minutes and can therefore only process 7 transactions per second. Users have to wait for typically one hour (i.e., six blocks) to obtain high confidence on the finality of a transaction.

An ideal blockchain has the following four desirable properties, *security*, *decentralization*, *high throughput*, and *fast confirmation*. The key challenge of building such a blockchain system is the threat of security attacks. To obtain high performance, the system typically has to operate with a fast block generation rate. Because block propagation takes time, the system may therefore generate many concurrent blocks (i.e., forks). In Nakamoto consensus, concurrent blocks waste PoW computation because they do not contribute to the finality of the longest chain. They make the system vulnerable to *double spending attacks* that attempt to revert history transactions.

Moreover, a high block generation rate can make several recently proposed protocols vulnerable to *liveness attacks* [17, 31, 32]. An attacker can simultaneously generate blocks at two competing branches and strategically withhold/release these blocks to maintain the balance of the two branches. The attacker with little PoW computation power can stall the consensus progress [36].

Conflux: We present Conflux, the first blockchain system that achieves all of the four desirable properties. Conflux can process thousands of transactions per second while confirming each transaction with within one minute on average. With its novel consensus protocol, the consensus layer of Conflux is no longer the performance bottleneck, i.e., the throughput saturates its underlying gossip network bandwidth and the confirmation speed is within the same order of magnitude as the gossip network propa-

*The first two authors contributed equally.

gation delay. Conflux is provably secure (see our formal proof in [19]). It is also as decentralized and permissionless as Bitcoin — participants can join and leave the consensus process at any time and there is no privileged committee or super-node dictating the process. Conflux also implements a modified version of Ethereum Virtual Machine (EVM) [35] and most smart contracts in Ethereum can be directly ported to Conflux.

To address the security attack challenge, Conflux organizes blocks into a novel Tree-Graph structure, which is a tree embedded inside a direct acyclic graph (DAG). In Tree-Graph, concurrent blocks are not considered harmful and they contribute to the Conflux ledger as well. Their PoW solutions will improve the finality of all of their ancestors and their transactions will be optimistically included into the ledger total order. This secures Conflux against double spending attacks and improves the Conflux throughput. To address liveness attacks, the consensus protocol of Conflux inherently encodes two different block generation strategies: an optimistic strategy that allows fast confirmation and a conservative strategy that ensures the consensus progress. Conflux uses its novel *adaptive weight* mechanism to combine these two strategies into a unified consensus protocol.

Adaptive Weight: Conflux assigns a weight to each block, which indicates the amount of finality that the block contributes to its ancestors. For each new block, Conflux analyzes its topology in the Tree-Graph, decides whether a liveness attack is potentially going on (e.g., whether there are old ancestor blocks that are not finalized yet), and then adaptively assigns weights to blocks in the Tree-Graph to switch between the two strategies. In normal scenarios, the mechanism assigns weights in one way that enables the optimistic strategy to confirm transactions fast. When a liveness attack happens, the mechanism assigns weights in another way that enables the conservative strategy to thwart the attack.

Deferred Execution: The execution order of recently packaged transactions may oscillate temporarily in a system with fast block generation. A naive implementation of the transaction execution engine would have to roll back executions many times and waste computation resources. Conflux addresses this challenge with its *deferred execution* mechanism. Instead of executing transactions in every received block immediately, Conflux waits for several blocks so that the order is relatively stabilized. Our observation is that users need to wait for the stabilization of the total order anyway to confirm a transaction with high confidence. Therefore the deferred execution does not harm the user experience at all.

Link-Cut Tree: An efficient consensus implementation is important to the performance of Conflux. To maintain the Conflux Tree-Graph, a naive implementation has the time cost of $O(n)$ for processing a new block on average, where n is the number of existing blocks. To address this challenge, Conflux uses *link-cut tree* to maintain weight values in Tree-Graph efficiently. It reduces the processing time from $O(n)$ to $O(\log n)$ per block.

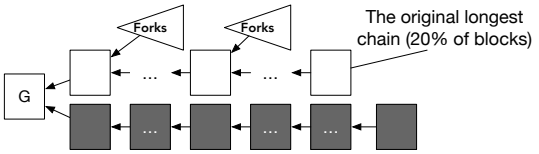
Experimental Results: We implemented Conflux and evaluated it with Amazon EC2 machines under the same experimental setup as previous work like Algorand and OHIE [11, 36]. Our experimental results show that with the bandwidth limit of 20Mbps and the simulated real world network latency setting, Conflux achieves a transaction throughput of 9.6Mbps and a confirmation latency of 47.75-51.54 seconds when running 3000-12000 nodes. For a combined workload of payment transactions and Ethereum history transactions, Conflux achieves up to 3480 transactions per second and confirms transactions within one minute with high confidence. Comparing to Algorand [11], Conflux has more than 4x higher throughput and comparable confirmation speed. Comparing to OHIE [36], Conflux has the same throughput and one order of magnitude faster confirmation speed.

Contribution: This paper makes the following contributions: 1) we design and implement Conflux, a decentralized and smart-contract-enabled blockchain system with high throughput and fast confirmation; 2) we present a novel consensus protocol with the adaptive weight mechanism; 3) we present a set of novel and critical optimizations, including deferred execution and link-cut tree.

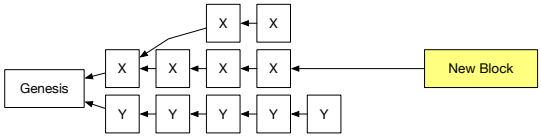
2 Related Work

Nakamoto Consensus in Bitcoin: Transactions are packed into blocks in Bitcoin. Each block has one predecessor block and all blocks form a tree structure with the genesis block as the root. Participants agree on the longest chain as the valid transaction history. Nakamoto consensus has to use a relatively slow block generation rate to avoid the generation of concurrent blocks, i.e., forks. This is essential for the safety against double spending attacks as shown in Figure 1a. More forks would mean relatively less blocks in the longest chain. In Figure 1a, due to forks, only 20% of blocks are on the longest chain so that an attacker with more than 20% of the network computation power can revert the longest chain to launch double spending attacks.

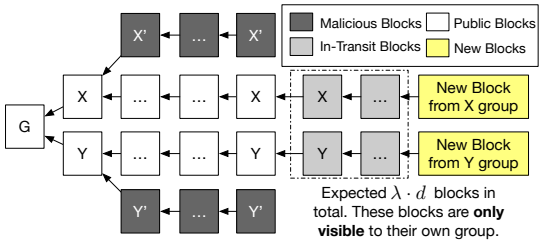
GHOST: GHOST is a previous proposal to replace the longest chain rule to improve the consensus safety under a fast block generation rate [32]. It is partially implemented in Ethereum [2]. Figure 1b presents an exam-



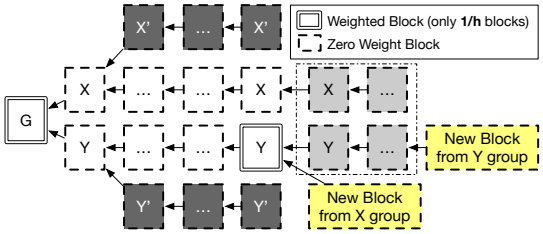
(a) An attacker with more than 20% of the network computation power can revert the longest chain.



(b) GHOST algorithm iteratively advances to the largest subtree to select the agreed chain.



(c) The attacker can strategically withhold or release his/her blocks to maintain the balance of two subtrees.



(d) If h is large enough, two groups will converge eventually.

Figure 1: Double Spending Attack, GHOST, Liveness Attack, and Structured GHOST

ple to illustrate the GHOST algorithm. GHOST starts from the genesis block and iteratively advances to the child block with the largest subtree to select the agreed chain [32]. In Figure 1b, the new block appends to the end of the agreed chain, which is in the subtree of X (containing 6 blocks) not in the subtree Y (containing 5 blocks). The difference between GHOST and the longest chain rule is that all blocks generated by honest participants will contribute to the finality of the agreed chain. Suppose G is an old enough block that is on the agreed chains of all honest participants. Future blocks generated by honest participants will all contribute the finality of G regardless of whether they are concurrent or not, because all of these blocks will be under the subtree of G. Unlike the longest chain rule, an attacker would need more than half of computation power to revert G from the agreed chain even with the presence of concurrent blocks [32].

Liveness Attack on GHOST: Unfortunately, GHOST is vulnerable to liveness attacks if the block generation rate is very fast. Figure 1c presents one example of such attacks. The example has the following settings: 1) the total block generation rate of honest participants is λ; 2) honest participants are devided into two groups with equal computation power (group X and group Y in Figure 1c); 3) blocks will transmit instantly inside each group, but the propagation between these two groups has a delay of d. In Figure 1c, each of the two groups extends its own subtree following the GHOST rule. Note that recent generated blocks within the time period of d are in-transit blocks (gray blocks in Figure 1c), which are only visible by the group who generates them. Therefore each group will believe its own subtree is larger until one group generates sufficiently more blocks than the other to overcome the margin caused by the in-transit blocks.

In normal scenarios, one of the two groups will get lucky to enable the blockchain to converge. However, an attacker can mine under two subtrees simultaneously to delay the convergence. The attacker can strategically withhold or release the mined blocks to maintain the balance of the two subtrees as shown in Figure 1c.

Theoretically, if honest participants evenly split due to network delay and the margin caused by in-transit blocks is significant, i.e., $\lambda d > 1$, a small portion of computation power is enough to launch attacks. Previous work [36] includes a simulation shows only 10% will do. In practice, even if honest participants do not have an even partition, the more computation power the attacker controls, the more likely the attacker will succeed. Consider the presence of mining pools, it is not rare to see one miner controlling more than 20% of computation power. Such a miner will be able to launch successful balance attacks without even partition.

DAG-based Structures: To improve the throughput and the confirmation speed, researchers have explored several alternative structures to organize blocks. Inclusive blockchain [17] extends the Nakamoto consensus and GHOST to DAG and specifies a framework to include off-chain transactions. In PHANTOM [31], participating nodes first find an approximate k-cluster solution for its local block DAG to prune potentially malicious blocks. They then obtain a total order via a topological sort of the remaining blocks. Unfortunately, when the block generation rate is high, inclusive blockchain and PHANTOM are all vulnerable to liveness attacks similar to Figure 1c. Therefore, unlike Conflux they cannot achieve both the security and the high performance.

Some protocols attempt to obtain partial orders instead of total orders for payment transactions. SPECTRE [30]

produces a non-transitive partial order for all pairs of blocks in the DAG. Avalanche [4] connects raw transactions into a DAG and uses an iterative random sampling algorithm to determine the acceptance of each transaction. Unlike Conflux, it is very difficult to support smart contracts on these protocols without total orders.

Hierarchical and Parallel Chains: Besides DAG, alternative ways to organize blocks include hierarchical chains and parallel chains. For example, in BitcoinNG [9], a macro block is generated every 10 minutes. The miner of such a block becomes the leader to generate micro blocks that contain actual transactions until the next macro block. Similarly, FruitChain [27] packs transactions first into fruits (i.e., micro blocks) and then packs fruits into blocks. OHIE [36] runs multiple parallel chains with the standard Nakamoto consensus and then deterministically sorts blocks to obtain a total order.

The shared property of these protocols is that only a small portion of blocks (e.g., macro blocks in BitcoinNG and FruitChain) influence the total order of the transaction ledger. It mitigates the liveness attack issue in Figure 1c because it reduces the chance of in-transit blocks influencing the total order. But these protocols have slow confirmation speed because they need to wait for more blocks to confirm transactions than other protocols. For example, BitcionNG has the same slow confirmation speed as Bitcoin [9]; OHIE confirms transactions in about 10 minutes on average only under an extremely fast block generation rate of 64 blocks per second [36]. In contrast, Conflux has a much faster confirmation speed in normal circumstances with no ongoing liveness attack.

Prism operates with one proposer chain and many parallel vote chains, each of which casts vote to decide the total order of blocks in the proposer chain [6]. The theoretical simulation in [6] shows that Prism may achieve high throughput and fast confirmation speed similar to our Conflux results in Section 6. But the simulation assumes a block propagation delay of one second, which is too ideal (e.g., the measured block delay is 10-15 seconds in our experiments). It is therefore unclear how fast a blockchain system that implements Prism can confirm transactions when running under practical P2P networks.

Byzantine Fault Tolerance: ByzCoin [14] and Thunderella [28] propose to achieve consensus by combining the Nakamoto consensus with Byzantine fault tolerance (BFT) protocols. Algorand [11], HoneyBadger [22], and Stellar [21] replace the Nakamoto consensus entirely with BFT protocols. In practice, all these proposals run BFT protocols within a confined group of nodes, since BFT protocols only scale up to dozens of nodes. The confined group is often chosen based on their recent PoW computation power [14, 28], their stakes of the system [11], or external hierarchy of trusts [21, 22]. However, these approaches may create undesirable hierarchies among participants and compromise the decentralization of blockchain systems. In contrast, Conflux allows any participant to join and leave the network without permission. In addition, instead of *eagerly* deciding the total order of blocks as in BFT-based approaches, Conflux allows multiple blocks to be generated in parallel and finalizes their orders later, which leads to its higher throughput.

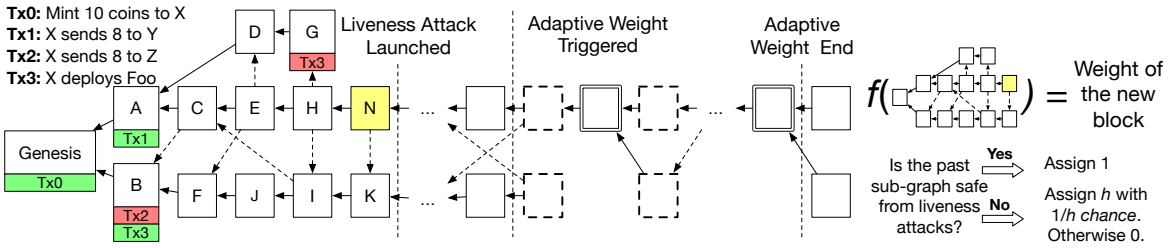
Sharding: Elastico [20], OmniLedger [15], RapidChain [37], and Monoxide [33] split the blockchain state into shards. Instead of having every node to verify all transactions, the systems select a small committee to maintain each shard to improve scalability. Unlike Conflux, such systems sacrifice security for scalability, i.e., the committee configuration can only be changed slowly (like days) due to reconfiguration overhead and therefore a small shard may be vulnerable to powerful attackers who can adaptively corrupt participants. This security issue is so important that Vault [16] chooses to only use sharding to mitigate storage cost with the trade-off of increased network bandwidth cost. Also despite the high combined throughput of all shards, the throughput of inter-shard transactions is still limited.

3 Overview

We will first present an example to illustrate a straw-man algorithm called *structured GHOST* that can defend against liveness attacks but has a sub-optimal confirmation speed. We will then present an overview of the Conflux consensus protocol, which uses the straw-man algorithm as a building block.

Structured GHOST: In our structured GHOST algorithm, only $1/h$ of blocks are weighted blocks that would count in the chain selection process. These blocks are selected randomly based on their PoW solution qualities (e.g., the number of leading zeros of the PoW hash). During the chain selection, the structured GHOST iteratively advances to the subtree with the largest number of weighted blocks, instead of considering all blocks.

Figure 1d shows an example to illustrate how structured GHOST can defend against the liveness attack we described before. With a sufficiently large h value, the expected number of in-transit weighted blocks ($\lambda d/h$) will be very small. As shown in Figure 1d, the generation of a weighted block of one group will very likely to make the two subtrees to converge, unless another group generates a concurrent weighted block. When $\lambda d/h \ll 1$, the chance of such concurrent generation is very unlikely. Without the margin caused by the in-transit blocks, the



(a) Tree-Graph Example. Solid blocks have weight one. Double-line blocks have weight 600. Dotted-line blocks have weight zero.

(b) Adaptive Weight

Figure 2: Examples of Conflux Consensus on Tree-Graph

liveness attack is not possible without significant computation power. Although structured GHOST is secure against liveness attacks, it sacrifices the confirmation speed — a user has to wait for the accumulation of enough weighted blocks to confirm a transaction.

Consensus with Two Strategies: The Conflux consensus protocol operates with two strategies, an optimistic strategy similar to the GHOST algorithm and a conservative strategy similar to the above straw-man algorithm. Our adaptive weight mechanism enables Conflux to encode these two strategies in a unified framework. In normal scenarios, Conflux would use the optimistic strategy to achieve high performance. If a liveness attack happens, the adaptive weight mechanism enables honest participants of Conflux to cooperatively switch to the conservative strategy to thwart the attack automatically.

Tree-Graph: The Conflux consensus protocol operates on the local Tree-Graph state of each individual node. Figure 2a presents a running example of the local Tree-Graph state of a node in Conflux. We will use this example in the remaining of this section to illustrate the high-level ideas of the Conflux consensus protocol. Each vertex in the Tree-Graph in Figure 2a corresponds to a block. In Figure 2a, Genesis is the predefined genesis block. Only Genesis, A, B, and G are associated with transactions. There are two kinds of edges in the Tree-Graph, parent edges and reference edges:

Parent and Reference Edges: Each block except Genesis has exactly one outgoing parent edge (solid line arrows in Figure 2a). For example, there is a parent edge from C to A. Each block can have multiple outgoing reference edges (dashed line arrows in Figure 2a). A reference edge corresponds to generated-before relationships between blocks. For example, there is an edge from E to D. It indicates that D is generated before E.

Pivot Chain: Note that all parent edges in the Tree-Graph together form a *parental tree* in which the genesis block is the root. In the tree, Conflux selects a chain from the genesis block to one of the leaf blocks as the *pivot chain*. Each block in the Tree-Graph may have a different weight determined by our novel adaptive weight mechanism. Conflux iteratively advances to the subtree with the heaviest total block weight to select the pivot chain. In Figure 2a before the liveness attack, Conflux selects Genesis, A, C, E, and H as the pivot chain to append the new block N. Note that Conflux does not selects the chain of Genesis, B, F, J, I, and K, because the subtree of A has heavier weights than the subtree of B.

Generating New Block: Whenever a node generates a new block, it first computes the pivot chain in its local Tree-Graph state and sets the last block in the chain as the parent of the new block. The node then finds all tip blocks in the Tree-Graph that have no incoming edge and creates reference edges from the new block to each of those tip blocks. For example, in Figure 2a, when generating N, the node chooses H as the parent of N and creates a reference edge from N to K.

Adaptive Weight: Figure 2b illustrates the basic idea of the adaptive weight mechanism. The goal is to assign a different weight to each generated block so that Conflux can adaptively switch between the optimistic strategy with a fast confirmation and the conservative strategy that ensures the consensus progress. Conflux determines the weight of a new block based on its past sub-graph, i.e., all blocks that are reachable via a traversal from the new block. As shown in Figure 2b, if the past sub-graph is safe — every old enough ancestor of the new block in the past sub-graph is secured on the pivot chain with high probability, the weight of the new block will be one. If not, the new block will be assigned an adaptive weight — it gets a weight of h with the chance of $1/h$ (depending on the PoW quality) or zero otherwise. Note that we set $h = 600$ in Conflux. See Section 4.1.

Liveness Attack Resilience: Figure 2a shows how the adaptive weight mechanism stops liveness attacks. Suppose after the generation of N, an attacker launches a liveness attack similar to one described in Figure 1c to balance the subtree of A and B. After a while, all honest participants start to generate blocks with adaptive weights, because they find that the old ancestors of

their new generated blocks (e.g., A or B in Figure 2a) are still not secured on the pivot chain with high probability. This essentially enables the consensus protocol to operate in the conservative strategy similar to the structured GHOST algorithm. In Figure 2a, the heavy weight blocks make Conflux to converge to the the subtree of A. After more blocks being generated under A, the past sub-graph of new generated blocks will become safe. Participants therefore assign weight one to the new blocks, automatically swtiching back to the optimistic strategy.

Epoch and Block Order: Parent edges, reference edges, and the pivot chain together enable Conflux to split all blocks in a Tree-Graph into epochs. Every block on the pivot chain corresponds to one epoch. Each epoch contains all blocks 1) that are reachable from the corresponding block in the pivot chain via the combination of parent and reference edges (including the pivot chain block itself) and 2) that are not included in previous epochs. For example, in Figure 2a, J belongs to the epoch of H because J is reachable from H but not reachable from the previous pivot chain blocks.

Conflux then derives a total order of all blocks in the Tree-Graph with the following rules. Conflux first sorts blocks based on their epochs. For blocks within the same epoch, Conflux sorts them based on their topological order. Conflux break ties determinisitcally (e.g., with the PoW quality or the block hash). For example, in Figure 2a, Conflux will obtain the following block total order for all blocks before the liveness attack: Genesis, A, B, C, D, F, E, G, J, I, H, K, and N.

Transaction Order: Conflux first sorts transactions based on the total orders of their enclosing blocks. If two transactions belong to the same block, Conflux sorts the two transactions based on the appearance order in the block. Conflux checks the conflicts of the transactions at the same time when deriving the order. If two transactions are conflicting with each other, Conflux will discard the second one. If one transaction appears in multiple blocks, Conflux will only keep the first appearance and discard all redundant ones. In Figure 2a, the transaction total order is Tx0, Tx1, ~~Tx2~~, Tx3, and ~~Tx3~~. Conflux discards Tx2 because it conflicts with Tx1.

4 Consensus on Tree-Graph

The local state of a node in Conflux is $S = \langle B, g \rangle$, where B is the set of blocks and $g \in B$ is the genesis block. There are several fields associated with a block $b \in B$. b.parent denotes the parent block of b. b.pred_blocks denotes the set of predecessor blocks linked by the reference and parent edges from b. b.pow_quality is the quality of the PoW solution — for b to be valid, b.pow_quality must no

$$\mathsf{Child}(B,b) = \{b' \mid b' \in B, b'.\mathsf{parent} = b\}$$
$$\mathsf{SubT}(B,b) = (\cup_{i \in \mathsf{Child}(B,b)} \mathsf{SubT}(B,i)) \cup \{b\}$$
$$\mathsf{SubTW}(B,b) = \sum_{i \in \mathsf{SubT}(B,b)} i.\mathsf{weight}$$
$$\mathsf{Past}(b) = (\cup_{i \in b.\mathsf{pred\_blocks}} \mathsf{Past}(i)) \cup b.\mathsf{pred\_blocks}$$
$$\mathsf{PastW}(b) = \sum_{i \in \mathsf{Past}(b)} i.\mathsf{weight}$$

Figure 3: The definitions of utility functions.

Input : A set of blocks B and a starting block b.
Output : The pivot block for the subtree of b.

```
1 if Child(B,b) = ∅ then
2     return b
3 else
4     w ← max{SubTW(B,i) | i ∈ Child(B,b)}
5     a ← arg min_{i∈Child(B,b)} {i.hash | SubTW(B,i) = w}
6     return Pivot(B,a)
```

Figure 4: The definition of $\mathsf{Pivot}(B,b)$.

less than the PoW difficulty D. b.weight is the adaptive weight of b. We use b.hash to denote the hash of b – all nodes in Conflux share a predefined deterministic hash function that maps each block to a unique id.

Figure 3 defines several utility functions and notations. Child() returns the set of child blocks of a given block. SubT() returns the set of blocks in the subtree of a given block in the parental tree. SubTW() returns the sum of the weights in the subtree. Past() returns the set of blocks that are generated before a given block. PastW() returns the sum of the weights of the past block set of a block. Note that $\mathsf{Past}(b)$ and $\mathsf{PastW}(b)$ are determined at the generation time of b and remain constant afterwards. In this section, we use lists to denote chains and serialized orders. "$\circ$" denotes the concatenation of two lists.

4.1 Pivot Chain and Adaptive Weight

Pivot Chain: Figure 4 presents the pivot chain selection algorithm in Conflux. Given a set of blocks B and the starting genesis block g, $\mathsf{Pivot}(B,g)$ returns the leaf block of the selected pivot chain. The algorithm recursively advances to the child block whose corresponding subtree has the largest total weights (lines 4-6). To break ties, the algorithm selects the child block with the smallest unique hash id (line 5). The algorithm terminates until it reaches a leaf block (lines 1-2).

Adaptive Weight: Figure 5 presents how we calculate the weight of a block b. The algorithm first determines whether the block should have adaptive weight or not based on the past block set of b (lines 1-11). If not, the weight of the block will be one (lines 12-13). If so, the algorithm checks the PoW solution quality against a difficulty that is h times higher than the base validation difficulty. The weight of the block will be h if it passes the check and be zero if it fails (lines 14-17).

To determine whether b should have adaptive weight, the algorithm operates at a sub-Tree-Graph that only contains blocks in the past set of b. It inspects every block in

```
Input  : A new block b
Output : The adaptive weight of b
1  B ← Past(b)
2  a ← b.parent
3  adaptive ← False
4  Let f(x) =
     2·SubTW(B,x) − SubTW(B,x.parent) + x.parent.weight
5  Let t(x) = |TimerChain(b)| − |TimerChain(x.parent)|
6  Let g(x) = |SubT(B,x.parent)|
7  while a.parent ≠ Nil do
8      if f(a) < α and (t(a) > β or g(a) > γ) then
9          adaptive ← True
10         break
11     a ← a.parent
12 if not adaptive then
13     return 1
14 else if b.pow_quality ≥ h·D then
15     return h
16 else
17     return 0
```

Figure 5: The definition of AdaptiveWeight(b)

```
Input  : A block b.
Output : The timer chain of the past sub-graph of b.
1  if b.pred_blocks = ∅ then
2      return b
3  else
4      a ← arg max_{i∈b.pred_blocks} {|TimerChain(i)|}
5      if b.pow_quality > h_0·D then
6          return TimerChain(a) ∘ b
7      else
8          return TimerChain(a)
```

Figure 6: The definition of TimerChain(b).

the path from the genesis to b.parent. For each inspected block a, it determines 1) whether a is still not secure on the pivot chain with high probability – the subtree weight of a is not significantly larger than the weight of the sibling subtrees of a (i.e., $f(a) < \alpha$) and 2) whether a is old enough – there is an enough amount of timer ticks or an enough number of blocks in the subtree of its parent (i.e., $t(a) > \beta$ or $g(a) > \gamma$). If any inspected block satisfies these two conditions, b should have adaptive weight.

The intuition is that to make progress, for any pivot chain block a' in Tree-Graph, after a certain period of time, one of the child subtree of a' (e.g., the subtree of a) should become dominant. If the attacker attempts to maintain the balance between the subtrees of two (or more) children of a' for a long time, the condition at line 8 will become true for a. Therefore, all honest participants will start to generate blocks with adaptive weights. Conflux will essentially operate with a conservative strategy similar to the structured GHOST algorithm (see Section 3). This will thwart the attack to ensure the progress.

Timer Chain: Because an attacker with enough computation power may influence the subtree sizes of recent

```
Input  : The local state S = ⟨B,g⟩ and a new discovered
         block b
1  if b.pow_quality ≥ D then
2      Wait until Past(b) ⊆ B
3      if Pivot(Past(b),g) = b.parent then
4          b.weight ← AdaptiveWeight(b)
5          S ← ⟨B ∪ {b}, g⟩
```

Figure 7: The block validation procedure.

```
Input  : A block b
Output : An ordered list of all blocks in Past(b) ∪ {b}
1  if b.parent = Nil then
2      return ∅
3  L ← ConfluxOrder(b.parent)
4  B_Δ ← (Past(b) − Past(b.parent) − {b.parent}) ∪ {b}
5  while B_Δ ≠ ∅ do
6      B'_Δ ← {x | |x.pred_blocks ∩ B_Δ| = 0}
7      Sort all blocks in B'_Δ in order as a_1, a_2, ..., a_k
8          such that ∀1 ≤ i < j ≤ k, a_i.hash < a_j.hash
9      L ← L ∘ a_1 ∘ a_2 ∘ ... ∘ a_k
10     B_Δ ← B_Δ − B'_Δ
11 return L
```

Figure 8: The definition of ConfluxOrder().

pivot chain blocks via strategically withholding mined blocks, only counting the number of blocks under the subtree of a pivot block is not sufficient to detect whether the pivot block is old enough or not. To this end, Conflux uses a timer chain mechanism to obtain an attacker resilient estimation for the generation time of each block.

Figure 6 presents the definition of TimerChain(b), which is the longest chain of blocks in the past sub-graph of b whose PoW qualities are h_0 times higher than the normal difficulty. We then use the length of the timer chain as the timer tick of the generation time estimation of each block (i.e., line 5 in Figure 5). When h_0 is large enough respecting the network delay, the attacker cannot stop the growth of the timer chain, because honest participants will contribute to the timer chain almost synchronously. In Conflux we set $h_0 = 360$. See Section 6.1.

4.2 Block Validation and Total Order

Block Validation: Figure 7 presents the validation procedure for a new discovered block. It first checks whether the block has a PoW solution with a sufficient quality (line 1). The procedure will wait for all blocks in its past sub-graph being processed first (line 2). The procedure will then compute the pivot chain in its past sub-Tree-Graph to check whether it selects the right parent (line 3). If so, the procedure computes the weight of the new block and adds it to the local Tree-Graph state (lines 4-5).

Block Order: Figure 8 defines ConfluxOrder(), our block ordering algorithm. Given a block b, ConfluxOrder(b) returns the total order of all blocks in Past(b) $\cup$ $\{b\}$. The algorithm sorts the blocks based on their corresponding epochs, i.e., it first recursively orders all blocks in previous epochs. It then computes all blocks

in the epoch of b as B_Δ (line 4). It topologically sorts all blocks in B_Δ and appends them to the result list (lines 5-10), and uses the hashes to break ties (lines 7-8).

4.3 Correctness

We next discuss the intuitions behind the correctness of our consensus algorithm. Suppose the network together has a block generation rate of λ. The correctness of Conflux is based on the following assumptions: 1) attackers control at most δ of the total block generation power ($\delta < 0.5$); 2) the network is d-synchronous, i.e., if at time t one honest node broadcast a block via the gossip network, then before time $t+d$, all honest nodes will receive this block and add this block into their local states.

Adversary Model: The attacker can choose arbitrary strategies to disrupt honest nodes. We also assume 1) attackers immediately receive all blocks and transactions from the gossip network, 2) attackers can arbitrarily control the communication of honest nodes as long as the d-synchronous assumption holds. The attacker however does not have the capability to reverse cryptographic functions. Therefore honest nodes can reliably verify the integrity of a block in the presence of the attacker.

Safety: Conflux is safe against double spending attacks because of two facts: 1) to revert a transaction in an epoch, the attacker has to revert the pivot chain block associated with the epoch from the pivot chain; 2) reverting an old pivot chain block that is on the common pivot chain of all honest nodes requires the attacker to compete with all honest nodes together. Although honest nodes may generate blocks that are concurrent with each other, all of these blocks will be under the subtree of the common pivot chain block. As the time passes by, it will be impossible for the attacker to forge an alternative heavier subtree without the pivot chain block.

Liveness: Many previous consensus algorithms based on tree and DAG can only provide liveness guarantees if the block generation rate is significantly slower than the block propagation delay (i.e., $\lambda \cdot d \ll 1$) [13, 26]. In contrast, Conflux is safe against liveness attacks at the protocol level even when the block generation rate is fast, because if the consensus does not make progress for a certain period of time, all honest nodes will start to generate blocks with adaptive weights. In this scenario, only blocks with very high PoW quality will decide the total order and the block generation rate of these blocks is significantly slower than the block propagation delay (i.e., $\frac{\lambda \cdot d}{h} \ll 1$ for a large enough h). Because concurrent generation of such heavy weight blocks is rare, an attacker has to release a large number of previously withheld blocks to balance a new generated heavy block or all honest nodes will make progress and will recognize the heavy block as a common pivot chain block. Because block withholding capability of an attacker is limited by its block generation power, the attacker will eventually run out of blocks to continue the liveness attack. We prove in [19] that when $\delta < 1/2$, once a block in Conflux is seen by an honest node, its order will become irreversible with exception risk ε after $d \cdot O(\log(1/\varepsilon))$ time.

Confirmation Policy: Conflux confirms a block b if for any ancestor block of b, the corresponding subtree total weight is heavier than all of the subtrees of its siblings by a margin. This margin is not a preset value. It depends on the status of blockchain protocol. With the parameter setting used in our experiments, this margin is about 20∼30 in normal scenarios for obtaining the same confidence as waiting six blocks in Bitcoin. Specially, if Conflux is in the conservative mode and is generating blocks with adaptive weights, we need to wait six blocks with heavy weights instead. See [19] for the detailed formulas of the risk in confirming a block.

5 Implementation and Optimizations

We have implemented Conflux in Rust [1].

Difficulty Adjustment: For brevity, our algorithm in Section 4 assumes a constant PoW difficulty. Conflux operates with a difficulty adjustment mechanism tailored for Tree-Graph. Every 5000 epochs, Conflux counts the number of blocks generated in the last 5000 epochs and adjusts the difficulty accordingly to maintain a stable block generation interval. Instead of setting the weight of every normal block to one, Conflux sets the weight to the difficulty of the block. For a block with a heavy adaptive weight, its weight will be its difficulty multiplied by h. The rationale is to allow the block weight to align with the accumulated PoW as the difficulty changes.

Storage: A Conflux full node stores the blockchain account state as Merkle Patricia Trees [35] in a key-value DB. To save storage space, Conflux periodically forms checkpoints at specific epoch heights (e.g., every 200k epoch heights) when the confirmation risk of the pivot chain blocks at these heights become extremely low. After a checkpoint, all history transactions before it can be safely discarded — all full nodes treat the checkpoint block as the new genesis. Note that full nodes still store all block headers to help new nodes bootstrapping.

Bootstrap: To bootstrap a new full node to join the network, it first synchronizes all the block headers in the ledger from the peers and decides the latest confirmed checkpoint block based on the headers. It then fetches the corresponding checkpoint state from the peers and continues the execution from that state.

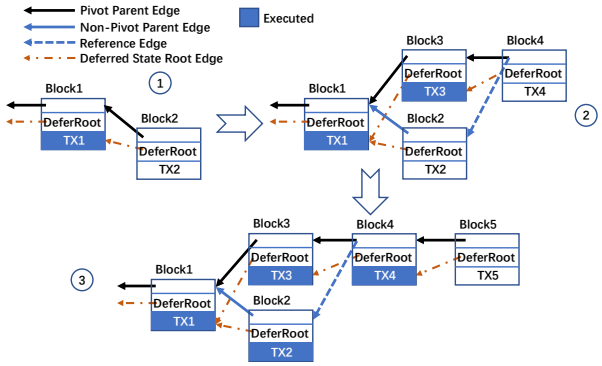


Figure 9: Save Redundant Execution by Deferred Execution

Transaction Relay: For a high performance blockchain like Conflux, it is critical to minimize the redundant transactions that are transferred. Ideally and optimally, each node should only receive each transaction exactly once. In current Bitcoin and Ethereum implementation, transactions are disseminated among nodes via flooding, which may waste network bandwidth resources. Erlay [24] tries to solve this issue of Bitcoin by letting peers exchange sets of unsent transactions that are encoded with PinSketch algorithm [8]. However, this method cannot be applied in Conflux, since it only works well when the difference between the transaction sets of two peers is small. Conflux is not this case because the transaction throughput of Conflux is orders of magnitude higher than Bitcoin. Conflux instead only floods 4-byte short transaction ids and pulls the missed transactions from peers. The short id is built by a SipHash [12] on the SHA-3 hash of the transaction and a peer-specific random nonce to significantly decrease the id conflict rate.

Signature Verification: Signature verification is computation-intensive and may become a bottleneck when the throughput is high. Conflux therefore uses a thread-pool to parallel the signature verification for different transactions to avoid bottlenecking other components.

Incentive Mechanism: For every mined block, Conflux assigns its block generation reward based on how many other blocks that are generated in parallel in Tree-Graph, i.e., blocks that are not in the past and future sets of the mined block. The more blocks are in parallel with the mined block, the smaller the block reward would be. This incentive mechanism penalizes malicious behaviors such as withholding mined blocks and not referencing other blocks. Because every block receives reward regardless of whether they are on the pivot chain or not, this mechanism nullifies selfish mining attack strategies [10, 25, 29]. See [7] for the details of the incentive mechanism.

5.1 Deferred Execution

In Conflux, when a block just enters the Tree-Graph structure, its position in the total order will change frequently. Although such oscillation will stop in a short time, it poses a challenge for the transaction execution engine. In typical blockchain systems, all transactions in a block immediately get executed in a node as soon as it is discovered. Such naive approach may execute transactions in a block many times as the order of the block oscillates. This incurs significant execution overhead. The top part of Figure 9 illustrates this problem. When a full node just gets the $Block_2$, it is on the pivot chain and the total order of transactions is $\{TX_1, TX_2\}$. TX_2 is then executed in this order. But, when the node later gets $Block_3$ and $Block_4$, $Block_2$ does not belong to pivot chain anymore and TX_3 is positioned between TX_1 and TX_2 in the newly decided total order. TX_2 has to be executed again.

Conflux uses a novel *deferred execution* mechanism to address this issue. The insight is that, just like users waiting for a period of time to confirm transactions, Conflux can wait for the total order position of a block to almost stabilize to execute its transactions. Conflux delays the execution by k-epochs. Specifically, unlike Ethereum where the header of each block b contains a merkle state root that corresponds to the execution results after processing all transactions in and before b, the header of b in Conflux contains a deferred root that corresponds to the execution results of the block that are k-hops older than b along its path to the root. We set k to five in Conflux so that Conflux can avoid re-execution of transactions in most cases. Five is also smaller than the typical number of epochs that an user needs to wait to confirm transactions and therefore it does not impact the user experience.

Figure 9 presents an example to illustrate the saving of redundant executions by using deferred execution. For illustration purpose, we set k to be one in this example. Therefore in Figure 9, the $Block_2$ stores the state root based on the execution of TX_1. When the full node gets $Block_3$ and $Block_4$, in order to verify the deferred state root of $Block_4$, the system needs to execute TX_3 but does not need to execute TX_2 because TX_2 is positioned after TX_3 in the decided total order. When getting and verifying $Block_5$, Conflux then needs to produce the state based on the execution of TX_4 which depends on TX_2. In the process, although the pivot chain oscillates between $Block_2$ and $Block_3$, TX_2 only gets executed once.

5.2 Link-cut Tree Optimizations

Maintaining pivot chain in Conflux is not trivial. Adding a new block to the Tree-Graph requires updating the subtree weights of all the blocks from this new block back

to the genesis. The naive approach will takes $O(n)$ time to complete, because the Tree-Graph height is usually linear to the number of blocks. To efficiently update the subtree weights, Conflux uses a data structure called *link-cut tree* [3]. Link-cut tree splits a tree structure into one or more paths, and represents each path using a splay tree, a form of balanced binary search tree invented by Tarjan et al. [5]. The link-cut tree is ideal for maintaining values like the subtree weights in the Conflux consensus protocol, because it enables the following operations at an amortized cost of $O(\log n)$: 1) increase or decrease values of all nodes along a path in the tree by a given value; 2) find the minimum or maximum value among values of all nodes along a path; 3) find the least common ancestor (LCA) of two nodes in the tree.

Update Pivot Chain: Conflux tracks the last pivot chain block of the current Tree-Graph state. Conflux uses the link-cut tree to maintain the total subtree weights of each block. When Conflux discovers a new block b, it inserts b into the link-cut tree and increases the total subtree weights of all blocks along the path from b to the root (i.e., genesis) by b.weight. Note that adding b may trigger a pivot chain change — instead of running the chain selection algorithm from the root, Conflux uses the link-cut tree to calculate the LCA of b and the current last pivot block p. If the weight of the subtree p belongs to is still heavier than the one b belongs to, no pivot chain update occurs. Otherwise, Conflux re-runs the selection algorithm from the LCA block. Because long range pivot chain reorganization is extremely rare, rerunning the algorithm from the LCA block is not expensive in practice.

6 Experimental Results

We next present a systematic evaluation of Conflux on its throughput, confirmation speed, and scalability. We also evaluate important design aspects of Conflux, e.g., the adaptive weight mechanism for defending against liveness attacks, the deferred execution for optimizing the transaction execution, as well as link-cut tree for optimizing the Tree-Graph maintainance.

We deployed Conflux on up to 800 Amazon EC2 m5.2xlarge virtual machines (VM), each of which has 8 cores and 1Gbps network throughput. By default, we run one Conflux full node in each VM. To model the network latency, we use the intercity latency measurements [34] and assign each VM to one of 20 major cities. We emulate the intercity delay by inserting artificial delays. For each full node, the gossip network of Conflux connects it to an average of 10 randomly selected peers.

When we measure the confirmation speed of a transaction, we count a transaction as confirmed if we can obtain the same confidence as empirically confirming a transaction in Bitcoin after waiting six Bitcoin blocks. In our experiments, unless otherwise noted, we limit the bandwidth of each full node to 20Mbps and we assign each full node with an equal block generation power.

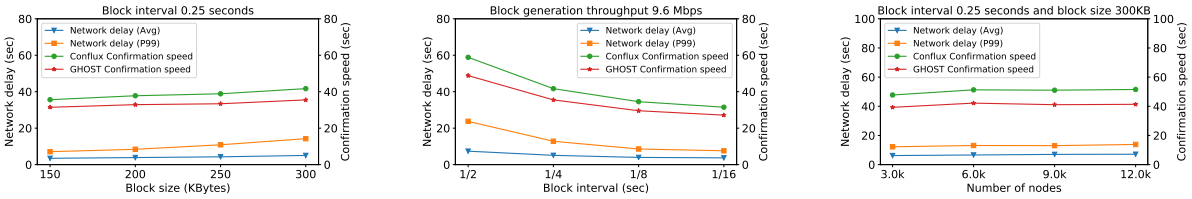
6.1 Protocol Parameter Calibration

To calibrate Conflux consensus protocol parameters, we run a set of experiments with 200 Conflux full nodes on Amazon EC2 with one full node per VM. We run Conflux with a set of different combinations of block size limits and block generation rates to measure the block propagation delays. For each setting, we run Conflux from the genesis for 10 minutes and fill each block to full with randomly generated simple payment transactions.

Figure 10a and 11a presents the experimental results of Conflux where we fix the block generation rate at four blocks per second and change the block size limit. Average network delay corresponds to the number of seconds on average for a generated block to reach more than 50% of participants. Network delay (99%) corresponds the number of seconds for all blocks to reach more than 99% of participants. In our experiments, we use the network delay (99%) number as the network diameter d for calculating parameters. There are often one or two machines lagging behind for some blocks and we can tolerate them as temporary failure nodes.

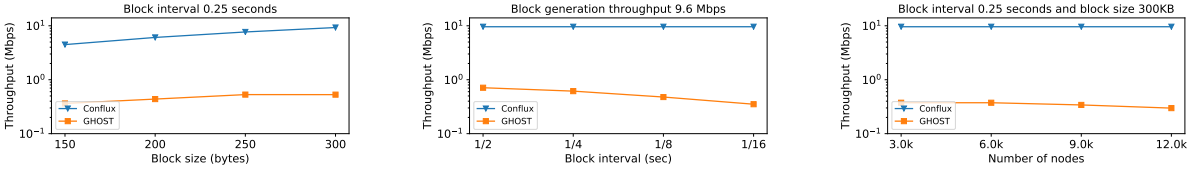
Conflux achieves the throughput of 9.6Mbps at the setting of 300K $\times$ 4 blocks per second. We find that Conflux almost saturates its underlying gossip network capability, considering that we limit the bandwidth of each full node to 20Mbps, which is only enough to send each block twice on average. With block sizes of 350K and beyond, full nodes start to experience significantly higher delay and may not be able to catch up new blocks.

Figure 10b and 11b presents the experimental results of Conflux where we fix the block generation throughput at 9.6Mbps and change the block generation rate from 2 blocks/s to 16 blocks/s. Our results show that as Conflux operates with faster block generation rate and smaller blocks, the network propagation delay decreases. But the delay no longer decreases much as it approaches the latency limit of the network. Smaller network propagation delay will improve the confirmation speed of transactions, but there are additional costs for using high generation rate. 1) Conflux full nodes have to store all block headers (block content could be pruned away with checkpoint techniques) and the average header size of Conflux is 300$\sim$500 bytes; 2) high block generation rates incur more blocks in parallel and these blocks cannot process transactions with dependencies.



(a) Latency vs. block sizes. (b) Latency vs. block intervals. (c) Scalability

Figure 10: Network Delay and Confirmation Speed



(a) Throughput vs. block sizes. (b) Throughput vs. block intervals. (c) Scalability

Figure 11: Throughput

Based on the above trade-off, we choose the block generation rate of 4 blocks per second and the block size limit of 300K. With the measured network propagation delay, we determine the adaptive weight algorithm parameters following suggestions in our theory analysis [19]. 1) $h = 600$ is large enough to enable Conflux to tolerate liveness attacks from a powerful attacker that controls 40% of the network computation power; 2) $\beta = 160$ and $\gamma = 10000$ so that the confirmation policy gives a desirable margin; 3) $\alpha = 1800$ since it requires $\alpha \geq 3h$; 4) $h_0 = 360$ so the timer chain will have rare forks. Figure 10 plots the average confirmation speed under this set of parameters.

Compared to GHOST (by only considering pivot blocks as valid), Conflux achieves the similar confirmation latency while provides significantly higher throughput. As Figure 11b shows, the transaction throughput of GHOST decreases with increasing block generation rate since more concurrently generated blocks with smaller size lead to less valid transactions.

6.2 Performance Results

Consensus Scalability Results: We next evaluate the consensus protocol performance as the number of nodes increases. Due to our resource limitation, we have to run 15 full nodes per VM. Because we are evaluating the consensus protocol only, we turn off signature verification and transaction execution to ensure enough computation resources for 15 full nodes sharing each VM.

Figure 10c presents the network propagation delay and the average transaction confirmation speed when running Conflux with different numbers of full nodes. In

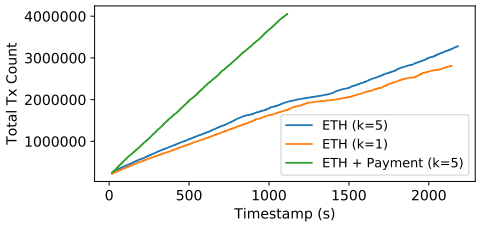


Figure 12: End-to-end Results for ETH Workload

all the experiments, Conflux successfully operates with the block throughput of 9.6Mbps (300K $\times$ 4 blocks per second). Our results highlight the fast confirmation speed of Conflux, it confirms transactions on average in 47.75-51.54 seconds when running 3000-12000 full nodes. Our results also show that the consensus protocol of Conflux scales well. As the number of nodes increases, the network propagation delay only increases slightly so does the confirmation speed.

Note that our experimental setup is as same as the Algorand and OHIE papers [11, 36], therefore we can directly compare our results with their results. Compared to Algorand [11], Conflux achieves more than 4x throughput and similar confirmation latency. Compared to OHIE [36], Conflux achieves similar throughput but one order of magnitude faster confirmation.

End-to-end Results: With the calibrated parameters, we run Conflux on 400 VMs (one node per VM) to measure the throughput and the confirmation speed of Conflux. To obtain a representative workload, we collected the first four million transactions from the Ethereum blockchain [2]. This includes both payment transactions and smart contract transactions. We converted these col-

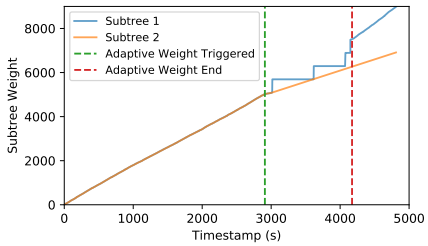


Figure 13: Subtree Weights under Liveness Attack

lected transactions into Conflux transaction format. We run two experiments, one experiment with the collected Ethereum transactions only and another experiment with a combined workload of the collected Ethereum transactions and randomly generated payment transactions. We terminate an experiment once Conflux processes four million transactions in total.

Figure 12 presents the number of processed transactions overtime. Our experimental results show that Conflux achieves a throughput of 1392 transactions per second for Ethereum workload and 3480 transactions per second for the combined workload. The average confirmation latencies are under one minute for both the Ethereum workload and the combined workload. Note that in the combined workload experiment, 14% of processed transactions are from Ethereum history.

Conflux achieves higher transaction throughput on the combined workload than on the Ethereum history workload. A primary reason is that Ethereum is a much slower blockchain and its transaction history does not have enough parallelism to saturates the Conflux consensus layer. We find that during the execution, future transactions in the Ethereum history often depends on previous transactions and full nodes of Conflux often do not have enough pending transactions ready to pick up so concurrent blocks pack duplicate transactions. A secondary reason is that Ethereum history contains more smart contract transactions which are more expensive to process than payment transactions.

Deferred Execution: Conflux by default defers the execution of transactions by five epochs $k = 5$. To illustrate the effect of the deferred execution optimization, we run a modified version of Conflux on the Ethereum history workload with $k = 1$. The results in Figure 12 show that this causes a 11.6% slowdown of Conflux on the transaction throughput because of the frequent re-execution of transactions during order oscillation.

6.3 Liveness Attack and Link-cut Tree

Liveness Attack: We conducted a liveness attack experiment to evaluate the security of Conflux. The experiment includes three nodes, two honest nodes and one attacker node. They keep a four block per second block generation rate for entire network. The block propagation network delay between the two honest nodes are 20 seconds and the attacker node does not relay honest blocks. We use only two honest nodes with significant delay to simulate the ideal liveness attack scenario in Figure 1c — a powerful attacker that evenly splits honest nodes in two groups and honest nodes with no latency inside a group and maximum latency between the two groups. The attacker node controls 30% of the total computation power. It launches the attack by finding the first fork between the two honest nodes and try to mine blocks under the lighter subtree to keep the two subtrees balanced.

Figure 13 shows how the weights of the two forked subtrees change along the time. The attack starts at the timestamp 0. During the time when the attack is performed and no adaptive weight is triggered, the weights of the two forked subtrees are almost perfectly balanced. The adaptive weight mechanism triggers the conservative strategy at timestamp 2,909 s. After that, the liveness attack quickly fails and the two honest nodes can then agree on the same pivot block and generate blocks under its subtree. After another 1,264 s, the two honest nodes come back to the optimistic strategy.

Link-cut Tree: To evaluate the benefits of link-cut tree, we run experiments on a micro-benchmark, a Tree-Graph that contains 1.5 million blocks, to measure the block processing throughput of the naive as well as our optimized approaches. Our experimental results show that the naive appraoch slows down to less than 4 blocks per second when the number of blocks in the Tree-Graph grows to one million, while our approach processes 5000 blocks per second on average.

7 Conclusion

Conflux is a scalable and decentralized blockchain platform with high throughput and fast confirmation. Its novel consensus protocol makes Conflux secure against both double spending attacks and liveness attacks, even if Conflux operates with a fast block generation rate. Conflux provides a promising solution to address the performance bottleneck of blockchains and opens up a wide range of blockchain applications.

Acknowledgments

We thank Bo Qiu and Yanpei Liu for their helps on the experiments. We thank Xi Wang and the anonymous reviewers for their insightful comments on the early draft of this paper. We note that [18] describes an early version of the Conflux system.

References

[1] Conflux-Rust. https://github.com/Conflux-Chain/conflux-rust.

[2] Etherum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper.

[3] Link/cut tree. https://en.wikipedia.org/wiki/Link/cut_tree.

[4] Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV.

[5] Splay tree. https://en.wikipedia.org/wiki/Splay_tree.

[6] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 585–602, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Yuxi Cai, Fan Long, Andreas Park, and Andreas Veneris. Engineering economics in the conflux network. *arXiv preprint arXiv: 2004.13696*, 2020.

[8] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 523–540, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59, 2016.

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

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

[12] Aumasson JP. and Bernstein D.J. Siphash: A fast short-input prf. *Progress in Cryptology - INDOCRYPT*, 2012.

[13] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. *IACR Cryptology ePrint Archive*, 2015:1019, 2015.

[14] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, 2016.

[15] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[16] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. Vault: Fast bootstrapping for cryptocurrencies. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[17] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[18] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint*, 1805.03870, 2018.

[19] Chenxing Li, Fan Long, and Guang Yang. GHAST: Breaking confirmation delay barrier in nakamoto consensus via adaptive weighted blocks. *arXiv preprint arXiv:2006.01072*, 2020.

[20] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

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

[22] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[24] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. Erlay: Efficient transaction relay for bitcoin. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 817–831, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Kartik Nayak, Srijan Kumar, Andrew Edmund Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *IEEE European Symposium on Security and Privacy*, pages 305–320, 2016.

[26] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

[27] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 315–324. ACM, 2017.

[28] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[29] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. *CoRR*, abs/1507.06183, 2015.

[30] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: Serialization of proof-of-work events: confirming transactions via recursive elections, 2016.

[31] Yonatan Sompolinsky and Aviv Zohar. Phantom, a scalable blockdag protocol. https://eprint.iacr.org/2018/104.pdf.

[32] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[33] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, 2019. USENIX Association.

[34] WonderNetwork. Global ping statistics: Ping times between WonderNetwork servers. https://wondernetwork.com/pings, Apr. 2018.

[35] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07), 2017. Accessed: 2018-01-03.

[36] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. OHIE: Blockchain scaling made simple. *arXiv preprint arXiv:1811.12628*, 2018.

[37] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: A fast blockchain protocol via full sharding.