Cuckoo's Nest: An Ultra-Lightweight DoS-Resilient Bitcoin Mempool

Hina Binte Haq*, Syed Taha Ali*, Siamak F. Shahandashti†
*National University Of Sciences and Technology, Pakistan
†University of York, UK

***Abstract*—The *memory pool (mempool)* plays a key role in processing and disseminating live transactions over the Bitcoin network. However, rising transaction loads and spam attacks significantly increase the mempool memory consumption which leads to dropped transactions, processing delays, and spikes in transaction fees, and exposes the network to sophisticated attacks. We present *Cuckoo's Nest*, a novel lightweight mempool design which provides resilience against spam attacks and contributes to the overall health of the network. Cuckoo's Nest reimagines the transaction pool using probabilistic data structures to fingerprint and forward live transactions. We implement Cuckoo's Nest in C++ and benchmark it using a unique 90-day Bitcoin transaction dataset. Our solution processes 300 MB worth of transaction load with only 12 MB RAM consumption with 99.999% fidelity and at three times the computational efficiency of the Bitcoin Core client. Cuckoo's Nest is an effective and efficient solution for lightweight and IoT-based Bitcoin clients; it does not require a hard fork; and its key design features can be adapted to other cryptocurrencies.**

***Index Terms*—cryptocurrency, Bitcoin, mempool, optimization, spam, denial-of-service**

I. Introduction

Bitcoin reigns the cryptocurrency landscape with over 56% market dominance and a compound annual growth rate of 63% [1]. Approximately 52% of institutional investors have reported holdings involving Bitcoin [2].

Bitcoin, however, continues to face various scalability and security challenges. One such concern is increasing transaction loads and spam attacks. These factors cause congestion in the Bitcoin transaction pool, referred to as the *mempool*. The mempool indexes unconfirmed or pending transactions in local memory (RAM) instead of disk for inventory and network-wide propagation. The mempool uses map data structures to organize transactions, resulting in memory usage of several hundred megabytes, typically three times the raw transaction size [3].

Mempool congestion correlates with network-wide spikes in transaction fees, delayed or dropped transactions, and also renders the Bitcoin network vulnerable to more complex attacks. Moreover, the growing transaction loads add considerably to the resource costs of operating Bitcoin nodes. Unlike Bitcoin miners, node operators are not financially incentivized to contribute resources to the Bitcoin network, and increasing costs can have a discouraging effect on their participation, and thereby the overall footprint size and decentralization level of the Bitcoin network.

Spam and dust attacks targeting the mempool pose a more dire threat [4]. Numerous spam and congestion incidents have occurred over the years. In October 2015, a Bitcoin spam campaign expanded the mempool to 1 GB (88,000 transactions), causing an estimated 10% of Bitcoin nodes to crash [4]. Dust attacks involve sending minute amounts of cryptocurrency, known as dust, to numerous wallet addresses, and are the most common type of spam. Dust attacks have various purposes including network disruption, advertisements, criminal activity, and deanonymization attempts [5] [6] [7]. Ethereum [8], Litecoin [9], Solana [10] and BinanceChain [11] have also experienced severe disruptions due to dust attacks.

To address this issue, researchers have proposed strategies to identify and evict spam transactions from the mempool. These metrics are primarily based on transaction age and fee thresholds that are characteristic of dust transactions [12] [13]. However, these approaches suffer from high false positive rates, act as inadvertent blacklists, and can lead to denial of service. Moreover, these solutions do not address the broader issue of mempool congestion and growing local memory consumption.

In this paper, we propose *Cuckoo's Nest*, a novel solution to rearchitect the Bitcoin mempool to increase its resilience to high traffic loads and network spam. *Cuckoo's Nest* uses probabalistic data structures to record live transactions. Our key insight is that the two key functions of the mempool, *inventory* and *forwarding* may be dissociated. The transaction inventory function is necessary for mining blocks and is of interest primarily to miners. Our solution enables other nodes to prioritize transaction verification and the forwarding functionality instead. *Cuckoo's Nest* is particularly suited to lightweight clients and IoT devices.

Our solution relies on a construction of sequential cuckoo filters to fingerprint live transactions instead of storing them in their entirety. The challenge here was to design our construction in a way to cater to Bitcoin's complex rules for live transactions: namely, devising mechanisms to expire transactions based on age, for expiry of transaction inputs, for tracking and limiting double-spends without a deterministic record, and ensuring resilience to DoS attacks.

Specifically, we make the following contributions:

1) We describe *Cuckoo's Nest*, a reimagined mempool construction leveraging time-shifting cascading cuckoo filters to replicate the Bitcoin mempool's core functions. With

This is an e-print of a paper accepted to the 7th IEEE International Conference on Blockchain and Cryptocurrency (IEEE ICBC 2025).

an overall memory footprint of 12 MB, *Cuckoo's Nest* accurately processes 99.999% of transactions over periods in which the default Bitcoin mempool was observed to routinely reach 300 MB, even going as high as 1 GB.

2) *Cuckoo's Nest* is implemented in C++ and evaluated using a custom dataset collected over 90 days from an instrumented Bitcoin node. The dataset was specifically gathered to assess our scheme. Both the code [14] and our dataset [15] are publicly available.
3) We evaluate our solution and provide extensive empirical results in multiple dimensions including security, error rates, memory usage, and compute time.

Cuckoo's Nest has some limitations: it does not explicitly remove spam and thereby does not resolve congestion in blocks. However, it can easily be integrated with spam filtering schemes. Our solution also results in false positives, but they are several orders of magnitude less than those reported for spam filtering solutions. Moreover, nodes running *Cuckoo's Nest* cannot participate in mining.

On the positive side, Cuckoo's Nest does not require a hard fork and being orthogonal to other light clients, can be aggregated with them to maximize benefits. Moreover, the cuckoo filter approach adopted by *Cuckoo's Nest* can be adapted to other cryptocurrencies. To the best of our knowledge, our work is the first to fundamentally redesign the mempool itself to prioritize security, efficiency, and operational costs.

We examine the requisite background in §II, followed by the proposed scheme in §III. We analyze and discuss empirical results in §IV. Conluding remarks are given in §VI.

II. Background

In this section we discuss the internals of the Bitcoin Core mempool, prior work and the mechanism of cuckoo filters.

A. Internals of the Bitcoin Core Mempool

The *memory pool (mempool)* is an in-memory staging area where transactions pending confirmation are temporarily stored. All incoming transactions for a node are *verified* for adherence to Bitcoin rules by the mempool and only admitted if they are valid transactions. The transactions are *stored* until they are included in a block (few transactions may be evicted before inclusion in a block for various reasons discussed ahead). Valid transactions are *advertised* to all nodes but are only *broadcast* upon request. This allows nodes to independently verify them and propagate them further within the Bitcoin network. A transaction is added to the mempool and *advertised* only when it is first received.

The mempool may also index transactions, prioritizing them by fee, size, input age, etc., to aid block proposals. Transactions in the mempool may also be shared on request to populate a new node's mempool.

We briefly describe the Bitcoin mempool and its structure. Bitcoin mempool utilizes the class `CTxMemPoolEntry` to store the raw transaction data such as hash, size, fee, entry height, coinbase status, scripts, inputs, ancestor and descendant transactions [17]. Transaction metadata (time received, priority) and indexing data (data structure overhead allowing

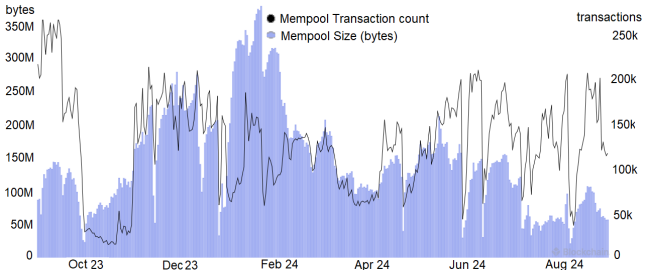


Fig. 1: Bitcoin mempool dynamics [16]

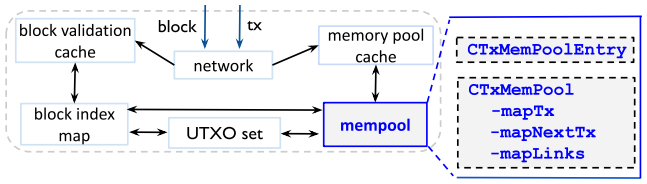


Fig. 2: Bitcoin Core Memory Schematic

efficient lookup and fetch transactions) are stored in the second class `CTxMemPool`. `CTxMemPool` has three components: 1) `mapTx` (`boost:: multi_index`) [18] sorts the mempool on five criteria: transaction hash, witness-transaction hash, descendant and ancestor fee rates, and time; 2) `mapNextTx` (`std::map`) [17] tracks the transaction inputs; and 3) `mapLinks` (`std::map`) [17] indexes in-mempool ancestor and descendant transactions data.

The size of the mempool depends on the number of transactions it contains and their individual size, as determined by the transaction content. `CTxMemPool` and `CTxMemPoolEntry` introduce up to 3 times memory overhead over the raw data [3]. This overhead in terms of pointers, indexes, and metadata is necessary for efficient transaction lookup and retrieval. Fig. 1 shows raw transaction data size, and the number of transactions in the Bitcoin mempool since August 2023. The mempool, typically 3 times the size of the raw data, commonly occupies several hundred megabytes in RAM. The mempool is allocated 300 MB by default [19], but can be reduced (`-maxmempool`) or disabled entirely (`-blocksonly`).

The minimum recommended storage for setting up a Bitcoin full-node is 2 GB RAM and 350 GB disk space [20]. As shown in Fig. 2, key memory components of a Bitcoin Core node besides the mempool include: partial *UTXO set* for transaction validation, *memory pool cache* for new transactions awaiting validation, *block validation cache* to store signature verification results, *block index map* for efficient block retrieval from disk, and *network* connections data.

B. Prior Work

The concept of light clients was introduced by Satoshi Nakamoto himself through Simplified Payment Verification (SPV) clients, which only download block headers and selective transactions to verify payments [21]. Since then, researchers have numerous optimizations that reduce the storage, memory, computation and communication demands on nodes while maintaining robust security.

We highlight key proposals for reducing resource consumption in Bitcoin: Efforts to reduce bootstrapping costs include pruned nodes [22], Non-interactive Proofs of Proof-of-Work (NIPoPoW) [23], FlyClient [24] and TXCHAIN [25]. Dietcoin [26] and Utreexo [27] aim to compress the UTXO set. Graphene [28], Erlay [29] and Compact Blocks [30] aim to reduce data exchanged in the Bitcoin network, while Segregated Witness (SegWit) [31] helped reduce computational overhead by optimizing transaction signature verification.

The mempool is an area of Bitcoin research that remains relatively underexplored. Default configurations allocate about 300 MB for the Bitcoin mempool [19], however custom configurations are permitted. Mempool schemes, particularly concerning spam attacks have been highlighted in various studies. Baqer et al. [4] analyzed a major spam attack on Bitcoin, identifying mempool vulnerabilities and recommending transaction evictions or the implementation of a dynamic fee model similar to Litecoin's to combat spam.

Boškov et al. [32] introduced the Set Reconciliation-Enhanced Propagation (SREP) algorithm to reduce bandwidth usage and speed up transaction pool synchronization using set reconciliation techniques, operating distributedly outside the network's block propagation channels.

Further studies, such as Contra by Saad et al. [12], introduced eviction strategies based on transaction age and fee thresholds, identifying and removing dust transactions. Wang et al. [13] developed Anti-dust, a model using Gaussian distributions to filter out low-value transactions, redirecting them to a separate dust pool. However, these approaches struggle to balance eviction thresholds; overly strict criteria can misclassify legitimate transactions as spam, leading to false positive rates much higher than the 1–2% threshold discussed by Baqer et al [4]. If multiple nodes were to deploy these filters, they may function as inadvertent blacklists. Attackers can modify spam transactions to evade filters, while filters lack dynamic and real-time adaptation to filter spam. Implementing real-time filters and separate pools for spam transactions, also incurs computation costs and increased local memory consumption which remain to be evaluated.

Moreover, solutions such as adopting a fee-per-output policy, which charges transaction fees based on the number of outputs rather than just size, require a hard fork to implement. Similarly, dynamic block sizes, which allow blocks to scale based on network demand, also necessitate a hard fork, which can be a contentious process.

Existing approaches focus primarily on identifying and evicting spam, rather than addressing the broader issue of mempool congestion and memory consumption. Our approach, emphasizing mempool resilience to high transaction volumes without spam filtering, is orthogonal to these strategies. This enables integration with existing solutions, enhancing Bitcoin's scalability and resource efficiency.

Building on Neonpool [33] and Carbyne [34], which used Bloom filter variants for a memory-efficient transaction pool in Bitcoin, we extend this work with a detailed study of cuckoo filter-based mempool construction. Our approach reduces false positives and negatives with minimal computational expense while enhancing resilience to DDoS attacks.

C. Cuckoo Filter

A cuckoo filter [35] is a 2D bit matrix. It has a fixed number of buckets m, where each bucket is further divided into b slots, and each slot may hold a fixed number of bits f. Such a filter may hold a maximum of $b \cdot m$ elements, and the number of elements currently inserted is denoted by n. Hence the load factor at any time may be calculated by $\alpha = n/(b \cdot m)$.

Cuckoo filters use two independent hash functions: the fingerprint hash f_x and the bucket hash H. These functions are part of partial-key cuckoo hashing, which identifies the two buckets where a fingerprint $\phi(x)$ of an item x may be stored. The fingerprint hash determines the item's fingerprint, while the bucket hash selects the two candidate buckets i_1 and i_2 for storage. If neither has space, one bucket (i_1 or i_2) is chosen randomly, triggering the recursive cuckoo eviction process.

For the fingerprint function $\phi(x) = f_x(x)$, the partial-key cuckoo hashing is calculated as follows:

$$i_1 = H(x) \mod m \tag{1}$$

$$i_2 = i_1 \text{ XOR } \phi(x) \mod m \tag{2}$$

i_1 and i_2 denote the indices of the buckets and are limited to the range of valid buckets from $[0, m-1]$. When m is a power of two, the modulo operation simplifies to a bit-wise AND, thus leading to greater efficiency on modern hardware. The partial-key alternative bucket calculation given in Eq. 2 is free of false negatives only when m is a power of two.

Cuckoo filters use f bits to fingerprint each item, and the minimal fingerprint size for a given false positive rate ϵ and bucket size b is

$$f \geq 1 + \log_2 b - \log_2 \epsilon \tag{3}$$

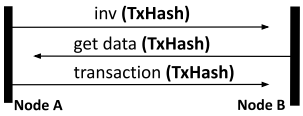
III. Proposed scheme

In this section, we provide a detailed explanation of our proposed scheme, *Cuckoo's Nest*, which does not necessitate the storage of complete transactions. Instead, it only stores transaction fingerprints, effectively disassociating the processes of transaction forwarding and inventory management. *Cuckoo's Nest* comprises two primary components. The first, `CuckooTxFilter`, utilizes the transaction hash `txHash` to map valid entry transactions. The second component, `CuckooTxInputsFilter`, ensures that duplicate or potential double-spend transactions are identified and discarded.

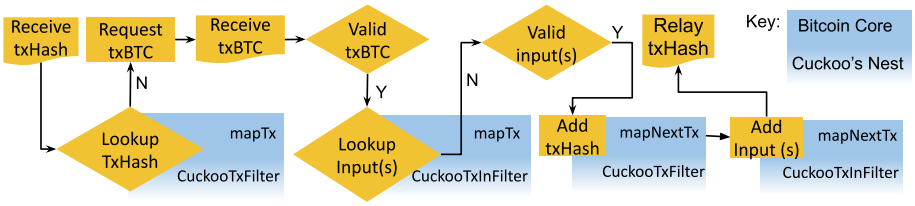
Here, we describe the transaction entry and exit process for the Bitcoin Core mempool and *Cuckoo's Nest*.

A. Entry

In both Bitcoin Core and *Cuckoo's Nest*, a sending node (Node A) makes a transaction announcement through an `inv` message. At the receiving node (Node B), `txHash` is used to query the mempool to determine if the transaction already exists in the mempool. In *Cuckoo's Nest*, the `txHash` is used to query the `CuckooTxFilter`. Nodes may receive a transaction announcement multiple times, but only accept it the first time they receive it. In both Bitcoin and *Cuckoo's Nest*, if a transaction with the same `txHash` has already been



(a) Mempool synchronization



(b) Transaction entry process for Bitcoin Core mempool and Cuckoo's Nest (Where only Y(es) or N(o) is shown, the other implies the transaction is dropped.)

Fig. 3: Mempool synchronization and Entry

received, it is discarded. If the transaction is determined to be new, the complete transaction is requested from the sending node via a `transaction` message, ensuring that both nodes eventually converge on a consistent view of the mempool. *Cuckoo's Nest* does not alter the number of round trips required for mempool synchronization compared to Bitcoin core. The synchronization protocol is summarized in Fig. 3a.

The transaction is checked for potential double-spends. In both Bitcoin and *Cuckoo's Nest*, each *input*, comprising the `inputtxHash` and `index`, is scanned for double-spends. Inputs are validated using the UTXO set, and transactions with invalid or spent inputs are discarded. It is also checked that none of the inputs exist in the mempool in Bitcoin. For *Cuckoo's Nest*, the `CuckooTxFilter` is queried with the tuple ¡`inputtxHash`, `index`¿ to ensure the input is not already in the filter. If any input is found in the mempool or `CuckooTxFilter`, the transaction is dropped. If two transactions with the same inputs are circulating, the first seen by a node is regarded as safe, while the second is dropped. If any input (parent or ancestor) is missing, the transaction is added to the orphan pool and will be re-processed once the ancestor is received. If the transaction passes verification, it is added to the mempool in Bitcoin and the `CuckooTxFilter` and `CuckooTxInputsFilter` in *Cuckoo's Nest*. Finally, the `txHash` is relayed to connected peers. This process is summarized in Fig. 3b.

B. Exit

In Bitcoin Core, transactions are removed from the mempool for various reasons, such as inclusion in a block, limited pool capacity, transaction expiry, fee priority, replacement by a newer version with a higher fee, invalid or conflicting transaction, or chain reorganization at the node.

Similarly, in Cuckoo's Nest, transactions are removed when a block arrives, as it contains confirmed transactions that should be cleared from the mempool. In Bitcoin Core, the mempool is updated by removing transactions from `mapTx`, `mapNextTx`, and `mapLinks`, while Cuckoo's Nest removes them only from the `Cuckoo TxFilter`. Transactions needing removal for reasons other than block inclusion accumulate and are removed by clearing the filter at periodic intervals. For this purpose, we cascade an additional filter to `CuckooTxFilter` to work in rotation. This configuration enables us to separate mempool transactions based on age.

We employ two identical cuckoo filters, a primary and a secondary, working together in rotation, which switch status after a predefined interval. All queries are directed to the primary filter first. If a transaction is not present there, then the secondary filter is queried. Transactions are removed from the filter that first reports them to be present. However, insertions only occur into the primary filter. After every predefined interval, the secondary filter is reset, and status of the two filters is switched again, effectively simulating transaction expiry. We implement this mechanism using Bitcoin's default 14-day expiry, as discussed in detail in § IV-D.

`CuckooTxInputsFilter` is periodically cleared to avoid overflow. Batch deletion removes the need to store individual transaction-input mappings and avoids indivdual deletions. This process is detailed in §IV-E.

IV. Empirical Results and Discussion

A. Methodology, Dataset and Implementation

We record *entry* and *exit* transactions in the transaction pool in JSON format (for the raw transaction structure in Bitcoin see [36]) to allow us to reconstruct the transaction pool state at the client. Our data set also includes all transactions received over the network (for the Bitcoin inventory message structure see [37]), in CSV format, to help us replay network activity for simulation purposes. For Bitcoin, we run an instrumented version of Bitcoin Core modifying `txmempool.cpp`, to capture 30 million unique transactions (around 90 million transaction announcements over 30 days).

We develop a simulation of the *Bitcoin mempool* using map data structures, and of *Cuckoo's Nest* using cuckoo filters. We replay transactions in the data set to reconstruct the *Bitcoin mempool* over the 90 days. The simulated *Bitcoin mempool* acts as the ground truth, and running it in parallel with *Cuckoo's Nest* helps evaluate how our scheme performs.

We use C++ to implement Cuckoo's Nest and simulate the Bitcoin Core mempool. We use the cuckoo filter library by Efficient Computing at Carnegie Mellon [38] to implement cuckoo filters. The Cuckoo's Nest implementation consists of probabilistic data structures `CuckooTxFilter` and `CuckooTxInputsFilter`. We simulate Bitcoin Core mempool's key structures, `mapTx`, `mapNextTx` and `mapLinks`. The `CuckooTxFilter` in Cuckoo's Nest and the `mapTx` structure in Bitcoin Core are independently queried at every *entry*, *inv* and *exit* event in our dataset to check if the relevant transaction exists in the mempool or not. Due to its probabilistic nature, Cuckoo's Nest's `CuckooTxFilter` will sometimes deviate from the ground truth and yield false positives and negatives. Our code [14] and dataset [15] are both publicly available.

Cuckoo TxFilter	Buckets m	Slots b	Hash k	Fingerprint f bits	False Positive Rate Theor	No Exp	With Exp	Discarded Transactions No Expiry Num/(%)	With Expiry Num/(%)	Reprocessed Transactions No Expiry Num/(%)	With Expiry Num/(%)
1 MB	262,144	4	2	8	3.13×10^{-2}	5.57×10^{-2}	4.28×10^{-2}	914,608(0.815)	233,879(0.208)	8,070,080(9.11)	959(1.1×10^{-3})
2 MB	262,144	4	2	16	1.22×10^{-4}	2.80×10^{-2}	2.27×10^{-3}	441,161(0.393)	27,417(0.024)	7,360,622(8.32)	26(2.9×10^{-5})
4 MB	262,144	4	2	32	1.86×10^{-9}	1.94×10^{-3}	1.10×10^{-5}	206,384(0.184)	1,234(0.001)	5,054,885(5.74)	0(0)

TABLE I: Performance metrics for `CuckooTxFilter` of various sizes dimensioned for $n = 262,144$ transactions

B. Performance Metrics

The responses to mempool queries can be categorized into a confusion matrix. The outcomes as per event are as follows: For an $entry$ event the mempool is queried to add a received transaction: $\mathrm{TP}_{\mathrm{entry}}$: the transaction already exists in the pool and will be discarded; $\mathrm{TN}_{\mathrm{entry}}$: the transaction is new and will be added to the pool; $\mathrm{FP}_{\mathrm{entry}}$: the transaction is new and should be added to the pool but will erroneously be discarded; $\mathrm{FN}_{\mathrm{entry}}$: the transaction already exists in the pool, but will erroneously be added again.

For an inv event, the mempool is queried to check if a transaction is available in the mempool: $\mathrm{TP}_{\mathrm{inv}}$: the transaction already exists in the pool and the full transaction will not be requested; $\mathrm{TN}_{\mathrm{inv}}$: the transaction is new and the full transaction will be requested, to add to the pool; $\mathrm{FP}_{\mathrm{inv}}$: the transaction is new but the full transaction will not be requested, to be added to the pool; $\mathrm{FN}_{\mathrm{inv}}$: the transaction already exists in the pool but the full transaction will erroneously be requested, to add to the pool.

At $exit$, the mempool is queried to remove a transaction: $\mathrm{TP}_{\mathrm{exit}}$: the transaction exists and will be removed; $\mathrm{TN}_{\mathrm{exit}}$: the transaction does not exist and cannot be removed; $\mathrm{FP}_{\mathrm{exit}}$: the transaction does not exist, and another transaction is erroneously 'removed'; $\mathrm{FN}_{\mathrm{exit}}$: the transaction exists in the pool but is erroneously not removed.

Each filter-level outcome has different consequences for Cuckoo's Nest performance. False positives at all three events lead to transactions not being processed, reducing the overall accuracy of the system. Therefore, the overall false positive rate (FPR) is a vital metric. Specifically, any false positives at inventory and entry result in transactions being discarded, and the rate of discarding needs to be kept low. Regarding false negatives, any such outcome at inventory (or entry) will cause unnecessary reprocessing of transactions. False negatives at exit won't inflict immediate cost but will eventually increase the load factor, which may cause more false positives. Based on these insights, we define performance metrics:

- *False Positive Rate (FPR)* is a measure of accuracy, defined as the ratio of the false positives to the total number of queries (*entry*, *inv* and *exit*).

$$\mathrm{FPR} = \frac{\mathrm{FP}_{\mathrm{entry}} + \mathrm{FP}_{\mathrm{inv}} + \mathrm{FP}_{\mathrm{exit}}}{\mathrm{Queries}_{\mathrm{entry}} + \mathrm{Queries}_{\mathrm{inv}} + \mathrm{Queries}_{\mathrm{exit}}}.$$

- *Discarded Transactions* is a measure of the proportion of new transactions at $entry$ and $inventory$ that were erroneously discarded due to false positives.

$$\mathrm{DiscardedTxs} = \frac{\mathrm{FP}_{\mathrm{inv}} + \mathrm{FP}_{\mathrm{entry}}}{\mathrm{Queries}_{\mathrm{inv}} + \mathrm{Queries}_{\mathrm{entry}}}.$$

- *Reprocessed Transactions* is a measure of transactions processed twice due to false negatives at $inventory$.

$$\mathrm{ReprocessedTxs} = \frac{\mathrm{FN}_{\mathrm{inv}}}{\mathrm{Queries}_{\mathrm{inv}}}.$$

Note that circulating reprocessed transactions does not equate to actual double-spends, since nodes in the network, including Cuckoo's Nest nodes, will screen incoming blocks to prevent double-spend.

C. Dimensioning `CuckooTxFilter`

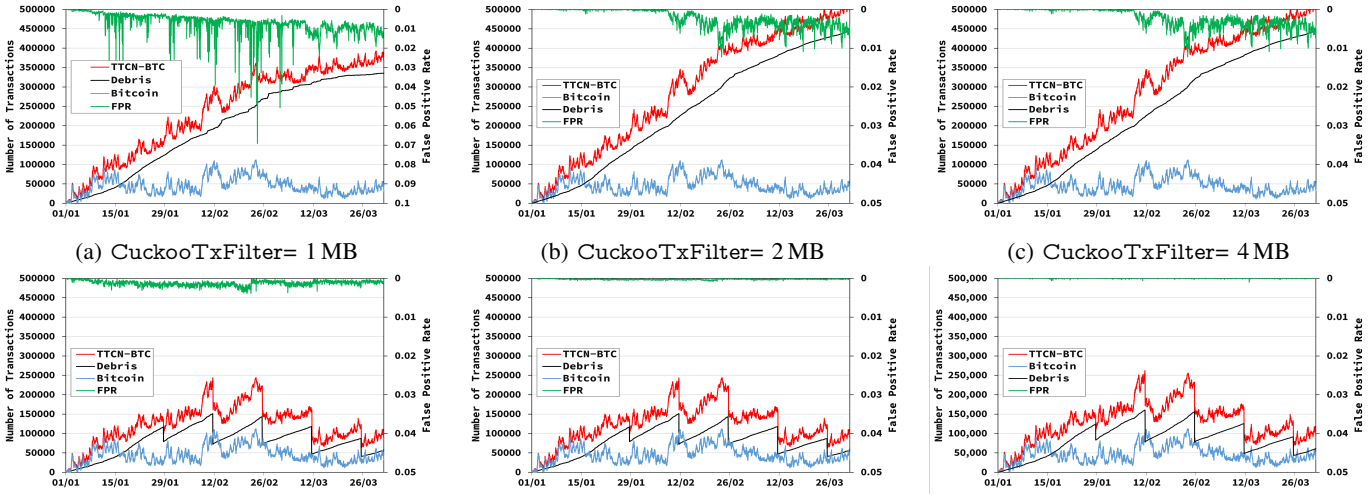
The highest transaction volumes observed to date are around 250,000, as shown in Fig. 1. We choose a maximum transaction load of 250,000 as a starting point to dimension `CuckooTxFilter`. Using Eq. 3, we derive a starting filter size of 1 MB with 262,144 rows (m, the number of rows, should be a power of 2 to avoid false negatives [39]), 4 buckets, an 8-bit fingerprint per item, and 2 hash functions. For comparison, we consider two other filter candidates, sized at 2 MB (medium) and 4 MB (large), with 262,144 rows, 4 buckets, 2 hash functions, and 16 and 32 bits per fingerprint, respectively. This is summarized in Table I.

D. Errors and expiry

We replay transactions in our dataset for the three month period through all filters. Table I shows the average FPR is highest for the 1 MB filter at 5.57×10^{-2}, reducing marginally to 2.80×10^{-2} for a 2 MB filter, and further to 1.94×10^{-3} for the 4 MB filter. For the 1 MB, 2 MB, and 4 MB filters. This translates to 0.815%, 0.393%, and 0.184% of transactions being erroneously *discarded* and 9.1%, 8.3%, and 5.7% of transactions being *reprocessed*, respectively. As expected, false positives and negatives decrease with larger filter sizes.

Figs. 4a–4f depict the number of transactions stored in these three filters in over time along with the FPR for the entire three-month period, plus the number of transactions in the Bitcoin Core mempool, the ground truth in our evaluation. In all three cases, the number of transactions closely tracks the pattern in the Bitcoin Core mempool, with an increasing offset.

This offset is due to extra load in the filter from two main sources: 1) transactions that should have been removed from the mempool because of age, limited transaction pool capacity, fee priority, replacement by a newer version that offers a higher fee, invalid or conflicting transaction or chain reorganization event at the node, but Cuckoo filters have no inherent mechanism to track these; 2) false negatives result in some transactions being erroneously added to the mempool at *entry* and some, due to be removed, to persist at *exit*.



(a) CuckooTxFilter= 1 MB (b) CuckooTxFilter= 2 MB (c) CuckooTxFilter= 4 MB

(d) CuckooTxFilter= 1 MB×2 (e) CuckooTxFilter= 2 MB×2 (f) CuckooTxFilter= 4 MB×2

Fig. 4: Number of transactions in Bitcoin vs Cuckoo's Nest, Debris transactions and False positive rate

We term these erroneous artifacts *debris*, which accumulates over time and corresponds to growing FPR, specially after the transaction count exceed 262,144, as depicted in Figs. 4a–4c.

Empirical false positive rates are significantly higher than theoretical, e.g., $2.80{\times}10^{-2}$ vs. $1.220{\times}10^{-2}$ for the 2 MB filter. First, real deployments often report higher false positives due to non-ideal hash functions and "clumped" data distribution, unlike theoretical models that assume perfect hashing and uniform data spread [35]. Fan et al. confirm that Eq. 3 provides a lower bound on the FPR [35].

Second, debris rapidly causes transactions in CuckooTxFilter to exceed the 262,144 mark it was provisioned for. With increased load factor, both false positives and negatives rise. While decent false positive rates or lookup performance are maintained up to 95% filter occupancy, beyond this, performance rapidly deteriorates. Boskov et al. [39] show that at load factors below 80%, the false negatives rate remains near 1%. However, cuckoo filters 95% full lead to up to 10% false negatives. These false negatives result from cuckoo evictions to incorrect buckets during inserts. We confirm this effect in our experiments.

As there is no inherent mechanism to remove debris, we propose periodically 'clean up' the filters: We employ two identical cuckoo filters, a primary and a secondary, working together in rotation, which switch status after a predefined interval, enabling us to separate transactions on the basis of age. All queries are directed to the primary filter first. If a transaction is not present there, the secondary filter is queried. Transactions are removed from the filter that first reports them to be present. However, insertions only occur into the primary filter. After every predefined interval, the secondary filter is reset, and status of the two filters is switched.

We implement this mechanism using Bitcoin's default 14-day expiry, as shown in Figs. 4d–4f. The filters first switch roles on 15 January, followed by expiry events every 14 days starting 29 January, corresponding to sharp drops in filter transaction numbers. Cuckoo's Nest now closely follows

CuckooTxInFilter	1 hour		3 hours	
	Num	FPR	Num	FPR
1 MB	166,338	$1.88{\times}10^{-3}$	41,585	$4.7{\times}10^{-4}$
2 MB	72,956	$8.23{\times}10^{-4}$	1,824	$2.1{\times}10^{-5}$
4 MB	72,537	$8.18{\times}10^{-4}$	1,813	$2.0{\times}10^{-5}$

TABLE II: CuckooTxInputsFilter, $n = 262,144$

Bitcoin Core transaction patterns. Average false positive rates, discarded, and reprocessed transactions for our three filters are significantly reduced. The expiry period here ranges from 14–28 days, as a filter is cleared every 28 days.

Table II shows the average FPR is highest for the 1 MB filter at $4.28{\times}10^{-2}$, reducing to $2.270{\times}10^{-3}$ for a 2 MB filter, and further to $1.10{\times}10^{-5}$ for the 4 MB filter. For the 1 MB, 2 MB, and 4 MB filters this translates to 0.208%, 0.024%, and 0.001% of transactions being erroneously *discarded* and $1.1{\times}10^{-3}$%, $2.9{\times}10^{-5}$%, and 0% of transactions being *reprocessed*. As expected, false positives and negatives decrease with larger filter sizes.

E. CuckooTxInputsFilter Dynamics

CuckooTxInputsFilter scans inputs of incoming transactions to prevent double-spends as described in §III. The implications are $\mathrm{TP}_{\mathrm{inputs}}$: a transaction bearing that input was added to CuckooTx InputsFilter and the new transaction should be discarded; $\mathrm{TN}_{\mathrm{inputs}}$: a transaction bearing that input does not exist in CuckooTxInputsFilter and the new transaction should be added; $\mathrm{FP}_{\mathrm{inputs}}$: a transaction bearing that input does not exist in CuckooTxInputsFilter but the new transaction was erroneously discarded; $\mathrm{FN}_{\mathrm{inputs}}$: a transaction bearing that input already exists in CuckooTxInputsFilter, but the new transaction was erroneously added.

In our dataset, incoming transactions average 40,000 inputs per hour, peaking at 191,947. We dimension the CuckooTx InputsFilter for 262,144 transactions, similar to CuckooTx-

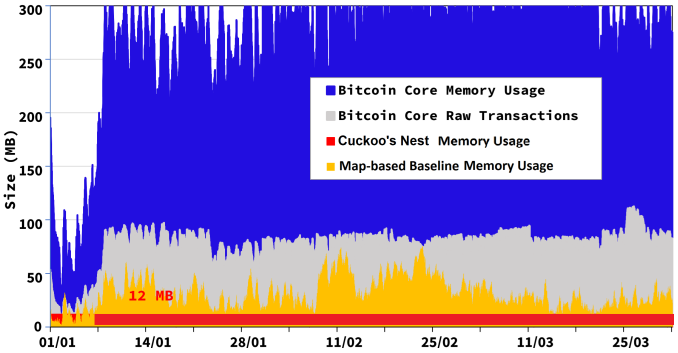


Fig. 5: Memory footprint for Bitcoin Core vs Cuckoo's Nest

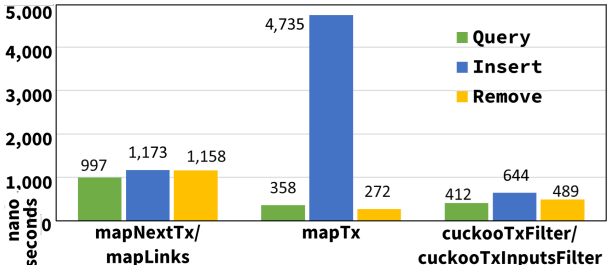


Fig. 6: Computation time: Bitcoin Core vs Cuckoo's Nest

Filter, and reset CuckooTxInputsFilter every hour and every three hours.

This removes the need to track individual transaction inputs. With a three-hour reset and CuckooTxInputsFilter sizes of 1 MB, 2 MB, and 4 MB, the FPR is $4.7{\times}10^{-4}$, $2.1{\times}10^{-5}$, and $2.0{\times}10^{-6}$, respectively, as shown in Table II. Thus, an attacker can resend a transaction with the same input after the expiry interval. However, since Cuckoo's Nest maintains complete UTXO information, nodes will reject blocks containing double-spend transactions.

F. Memory Footprint

We compare the memory footprint of Cuckoo's Nest and Bitcoin Core. Fig. 5 shows the raw transaction size versus the mempool size over the 3-month experimental period.

Cuckoo's Nest results are immensely promising: to process an equivalent transaction volume with 99.999% accuracy, Cuckoo's Nest requires only 12 MB of memory. The CuckooTxFilter (4 MB×2) and CuckooTxInputsFilter (4 MB) discard only 0.001% of transactions, achieving 99.999% accuracy. Users can adjust parameters to balance accuracy and memory usage.

As baseline, we consider a straightforward mempool optimization schemes which uses standard deterministic data structures such as maps), storing minimum transaction data. Rudimentary calculations indicate that for transaction validation, we would need to store the transaction ID (32 bytes), input hash (32 bytes) and index (4 bytes), amounting to $32 + 36n$ bytes for each raw transaction where n is the number of transaction inputs. Memory overhead will be 3 times this value. This approach reduces memory consumption, but the disadvantages are still significant. It is less resilient to congestion events and spam attacks, and scales linearly. It is as limited as Cuckoo's Nest from an inventory perspective, i.e., it cannot bootstrap mempools of other nodes.

G. Computation footprint

We next calculate computational overheads. Bitcoin Core components mapTx, mapNextTx, and mapLinks perform query, insertion, and deletion in $O(\log n)$ time, where n is the number of stored transactions, corresponding to the internal nodes of the red–black binary search tree used in the reference implementation. Counting cuckoo filters in Cuckoo's Nest operate in constant time using 2 hashes for query, insertion, and deletion, regardless of filter cardinality.

As depicted in Fig. 6, we perform benchmarks using an Intel Core i7 8700 CPU @3.2GHZ ×12 and 32 GB RAM, running Ubuntu 18.04 with GCC 5.4.0. We replicate Bitcoin Core structures mapTx (Boost multi-index), mapNextTx and mapLinks (C++ STL maps). We use the *cuckoo filter* library by Efficient Computing at Carnegie Mellon [38] to instantiate counting cuckoo filters and perform query, insert and delete operations, averaging over 1 million iterations.

a) CuckooTxFilter vs. mapTx: Query, insert and delete operations in mapTx take 358 ns, 4,735 ns, and 272 ns, respectively, whereas CuckooTxFilter (for $k = 2$) requires 412 ns, 644 ns, and 489 ns, respectively. For a complete transaction life-cycle, mapTx requires 5,365 ns compared to 1,545 ns for CuckooTxFilter.

b) CuckooTxInputFilter vs. mapNextTx: The query, insert and delete operations for mapLinks and mapNextTx require 997 ns, 1,173 ns, 1,158 ns, respectively. CuckooTxInputsFilter takes 412 ns and 544 ns, for query and insert operations, while deletion is replaced by batch reset. Over the transaction life-cycle, mapTx requires 3,328 ns compared to 956 ns for CuckooTxInputsFilter.

c) mapLinks: Maintaining unconfirmed transaction chains in mapLinks helps prioritize transactions but is computation- and memory-intensive, and a potential DDoS vector. To cap these costs, Bitcoin Core 0.12 introduced a default policy limiting unconfirmed chains to 25 transactions and 101 kB total size. Cuckoo's Nest avoids storing these mappings as explained in §III, eliminating these costs.

Thus, Cuckoo filter-based Cuckoo's Nest is at least three times faster than Bitcoin Core mempool.

V. Security Analysis

We assess an adversary's potential to exploit Cuckoo's Nest for DoS attacks via spam, censorship, or other methods:

A. DDoS resilience

We next discuss how *Cuckoo's Nest* copes with surge in transaction loads such as those in spam based DDoS attack. Our approach, emphasizes the resilience of the mempool to high transaction volumes without spam filtering, which acts as an inadvertent blacklist. Cuckoo's Nest can easily withstand such an attack by preventively dimensioning a bigger filter or recursively generating additional cuckoo filters recursively

Transaction count	Bitcoin Core mempool	Cuckoo's Nest
262,144	300 MB	12 MB
524,288	600 MB	24 MB
1,048,576	1200 MB	48 MB

TABLE III: Cuckoo's Nest amidst DDoS attack

to take the load. Since our approach does not differentiate between benign/spam transactions, our data set does no include spam transactions. However, incorporating spam into the dataset should yield similar results.

a) Preventive Dimensioning: The filters in Cuckoo's Nest are proactively dimensioned to handle an abnormal amount of transactions such as 524,288, over double the amount of transactions ever witnessed on the Bitcoin network. Such a filter would be 8 MB in size. The primary filter holds the first wave, but when overloaded, a secondary filter activates. The system remains stable, handling the load without disruption and with minimal errors.

b) Remedial Response: We initialize a counter to track the total number of transactions. As the transaction count exceeds the filter capacity of 262,144, recursive filter-generation kicks in and creates additional filters of the same capacity and size. These filters expire after a set period, freeing up space. This method dynamically scales based on load, keeping memory usage low while maintaining accuracy. As congestion subsides, additional filters are no longer needed.

The highest mempool volumes have bloated it to around 1 GB, while the highest transaction load is around 250,000. Table III illustrates the transaction counts, corresponding memory usage in Bitcoin Core's mempool, and memory consumption of Cuckoo's Nest ($2\times$`CuckooTxFilter` + $1\times$`CuckooTxInputsFilter`) to withstand these attacks. We defer empirical evaluation as future work.

B. Adversarial Resilience

Can an adversary craft transaction to inject or censor transactions at individual nodes? Literature shows cuckoo filters can be efficiently transformed to be adversarially resilient by applying a pseudo-random permutation of the input [40] [41], i.e. applying a sufficiently large (128 bit) random salt before forwarding it to the cuckoo filter. Thus, for an adversary to trigger a false positive at a specific node, they would need both the filter's initialization seed and its current state. Obtaining the seed is difficult as adversary only has oracle access to nodes. Nodes can regularly update their seeds (e.g., biweekly). Thus, adversaries can only broadcast random transactions, unable to craft transactions that cause false positives.

Can an adversary achieve network-wide injection or censorship of specific transactions? Each Cuckoo's Nest node initializes its filters independently using secret, randomly generated 128-bit seeds [40]. Given a FPR of 0.0001 per node, even if an adversary manages to trigger a false positive at one node, the probability of the same transaction being censored by two nodes is negligible $(0.0001)^2$. Therefore, adversarial attempts to target individual nodes do not scale across the Bitcoin network, and there is no feasible low-cost method for network-wide transaction censorship.

Thus, the adversary only has oracle access to the bloom filter i.e. does not know its contents or seed. We situate this assumption within established practices in the cryptocurrency ecosystem and light client security models as identified by Chatzigiannis et al. [42]. Further common assumptions that underpin light client designs like ours, including trusted genesis block, reliable consensus, secure underlying cryptographic primitives, weak synchrony (i.e. no long network partitions), trusted setup, peer-to-peer communication for relaying information, and rational behaviour of participants.

Can an adversary be successful in a double-spend attack? Cuckoo's Nest preserves Bitcoin's transaction verification and validation mechanisms. While nodes may forward transactions with conflicting inputs, they screen blocks to reject double-spends. Since Cuckoo's Nest retains full UTXO data, it rejects blocks with double-spend transactions. Hence, false negatives in the `CuckooTxFilter` only cause reprocessing, not double-spends. Replay transactions are dropped as already seen transactions will trigger a positive, indicating that the transaction is already present.

VI. Future Work and Conclusion

We propose *Cuckoo's Nest*, a novel cuckoo filter based design for the Bitcoin Core mempool that drastically reduces the mempool memory consumption, reducing the cost of running a full node, and increasing its reliability and survivability in the wake of network spam.

Our results thus far are immensely promising: with an overall memory footprint of 12 MB, Cuckoo's Nest accurately processes 99.999% of transactions while Bitcoin Core mempool routinely hits 300-500 MB, and requires at least three times less computational effort than Bitcoin Core mempool. We are working on developing a functional prototype for live deployment and launch a Bitcoin Improvement Proposal.

We foresee some challenges in deployment: a key challenge is bootstrapping new nodes storing random subsets of transactions in RAM. Our extra analysis shows that if a new node connects to 4, 8, or 12 Cuckoo's Nest nodes, each storing 10% of transactions, it can recover 30%, 55%, or 70% of the mempool, respectively. Other challenges include: mempool changes affecting layer-2 protocols e.g. Lightning network, securing consensus among the Bitcoin community, etc.

Cuckoo's Nest can be adapted for other cryptocurrencies, but implementation requires significant modifications tailored to each protocol. For example, adapting it for Bitcoin derivatives like Bitcoin Cash, Bitcoin Gold, Litecoin, and Dogecoin is relatively straightforward since they follow the UTXO model. In these cases, only specific parameters—such as transaction expiration times and cuckoo filter sizes—need adjustment based on network traffic and block intervals.

In contrast, adapting Cuckoo's Nest for Ethereum requires addressing key differences from Bitcoin. Ethereum's account-based system relies on state tracking for double-spend prevention, its transaction and transaction pool structure and transaction broadcast protocol differ. This warrants a dedicated study using Ethereum-specific data. To that end, we are currently adapting Cuckoo's Nest for Ethereum.

REFERENCES

[1] "Cryptocurrency Prices, Charts And Market Capitalizations | CoinMarketCap," 2024. https://coinmarketcap.com.

[2] Coinweb, "How Many People Hold Bitcoin in 2024? - Coinweb." https://coinweb.com/trends/how-many-people-hold-bitcoin, 2024.

[3] "The 300 MB default maxmempool Problem." https://b10c.me/blog/001-the-300mb-default-maxmempool-problem, Dec. 2017.

[4] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver, "Stressing out: Bitcoin "stress testing"," in *International Conference on Financial Cryptography and Data Security*, pp. 3–18, Springer, 2016.

[5] F. Memoria, "$700 million stuck in 115,000 unconfirmed bitcoin transactions, CCN." https://www.ccn.com/700-million-stuck-115000-unconfirmed-bitcoin-transactions/, 2017.

[6] K. Sedgwick, "200,000 Unconfirmed Transactions Pile Up in Another Crazy Day for Bitcoin." https://news.bitcoin.com/200000-unconfirmed-transactions-pile-another-crazy-day-bitcoin/, Dec 2019.

[7] A. Zmudzinski, "Bitcoin's Mempool Saw an Anomalous Number of Big Transactions on Friday- Coin Telegraph." https://cointelegraph.com/news/bitcoins-mempool-saw-an-anomalous-number-of-big-transactions, Nov 2019.

[8] B. Dale, "Mempool Manipulation Enabled Theft of 8M in MakerDAO Collateral on Black Thursday: Report - CoinDesk," *CoinDesk*, Jul 2020.

[9] "Litecoin - Open source P2P digital currency." https://litecoin.org, 2024.

[10] B. Quarmby, "Solana reportedly hit by DDoS attack, but network remains online," *Cointelegraph*, 12 2021.

[11] "Binance Is Facing Issues with Solana Withdrawals - Financial and Business News | Finance Magnates," Dec 2024.

[12] M. Saad, J. Kim, D. Nyang, and D. Mohaisen, "Contra-*: Mechanisms for countering spam attacks on blockchain memory pools," *arXiv preprint arXiv:2005.04842*, 2020.

[13] Y. Wang, J. Yang, T. Li, F. Zhu, and X. Zhou, "Anti-Dust: A Method for Identifying and Preventing Blockchain's Dust Attacks," in *ICISCAE 2018*, pp. 274–280, IEEE, 2018.

[14] H. Bhaq, "Cuckoo's Nest - BTC." Available on GitHub: https://github.com/hbhaq/CuckoosNestBTC, Mar. 2025.

[15] "Mempool state Bitcoin." Available on Kaggle: https://www.kaggle.com/datasets/mempoolstate/mempool-state-bitcoin, Mar. 2025.

[16] Blockchain.com, "Blockchain charts, Bitcoin mempool." https://www.blockchain.com/explorer/charts, 2024.

[17] "Txmempool.h, Bitcoin source code, Github." https://github.com/bitcoin/bitcoin/blob/master/src/txmempool.h, 2024.

[18] "Boost.MultiIndex Documentation - Performance." https://cs.brown.edu/~jwicks/boost/libs/multi_index/doc/performance.html, 2024.

[19] "P2P Network Guide - memory pool limit - Bitcoin." https://bitcoin.org/en/p2p-network-guide#memory-pool, Oct 2024.

[20] "Running a full-node - Bitcoin." https://bitcoin.org/en/full-node, 2024.

[21] S. Nakamoto, "Bitcoin p2p e-cash paper," *The Cryptography Mailing List*, 2008.

[22] "Full node - Bitcoin Wiki." https://en.bitcoin.it/wiki/Full_node, 2024.

[23] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive proofs of proof-of-Work," in *International Conference on Financial Cryptography and Data Security*, pp. 505–522, Springer, 2020.

[24] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "Flyclient: Super-light clients for cryptocurrencies," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 928–946, IEEE, 2020.

[25] A. Zamyatin, Z. Avarikioti, D. Perez, and W. J. Knottenbelt, "TxChain: Efficient Cryptocurrency Light Clients via Contingent Transaction Aggregation," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 580, 2020.

[26] D. Frey, M. X. Makkes, P.-L. Roman, F. Taïani, and S. Voulgaris, "Dietcoin: hardening bitcoin transaction verification process for mobile devices," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 12, no. 12, pp. 1946–1949, 2019.

[27] T. Dryja, "Utreexo: A Dynamic Hash-Based Accumulator Optimized for the Bitcoin UTXO Set.," *IACR Cryptol. ePrint Arch. 2019/611*, 2019.

[28] A. P. Ozisik, G. Andresen, B. N. Levine, D. Tapp, G. Bissias, and S. Katkuri, "Graphene: efficient interactive set reconciliation applied to blockchain propagation," in *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 303–317, Springer, 2019.

[29] G. Naumenko, G. Maxwell, P. Wuille, A. Fedorova, and I. Beschastnikh, "Erlay: Efficient transaction relay for bitcoin," in *Proceedings of the 2019 ACM CCS*, pp. 817–831, 2019.

[30] M. Corallo, "Compact block relay." https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki, 2016.

[31] E. Lombrozo, J. Lau, and P. Wuille, "Segregated witness (consensus layer)." BIP 141: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki, 2015.

[32] N. Boškov, S. Simsek, A. Trachtenberg, and D. Starobinski, "Srep: Out-of-band sync of transaction pools for large-scale blockchains," in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–9, 2023.

[33] H. Binte Haq, S. T. Ali, A. Salman, P. McCorry, and S. F. Shahandashti, "Neonpool: Reimagining cryptocurrency transaction pools for lightweight clients and IoT devices." arXiv preprint arXiv:2412.16217, 2025.

[34] H. Binte Haq, S. T. Ali, A. Salman, P. McCorry, and S. F. Shahandashti, "Carbyne: An ultra-lightweight DoS-resilient mempool for Bitcoin." https://eprints.whiterose.ac.uk/id/eprint/225617, 2025.

[35] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014.

[36] "Raw Transactions format - Bitcoin developer reference." https://developer.bitcoin.org/reference/transactions.html#raw-transaction-format.

[37] "P2P Network — Bitcoin, Inventory Messages - Bitcoin developer reference." https://developer.bitcoin.org/reference/p2p_networking.

[38] Efficient, "cuckoofilter - GitHub." https://github.com/efficient/cuckoofilter, 2024.

[39] N. Boskov, A. Trachtenberg, and D. Starobinski, "Birdwatching: False negatives in cuckoo filters," in *Proceedings of the Student Workshop*, pp. 13–14, 2020.

[40] D. Clayton, C. Patton, and T. Shrimpton, "Probabilistic data structures in adversarial environments," in *Proceedings of the 2019 ACM CCS*, pp. 1317–1334, 2019.

[41] M. Naor and Y. Eylon, "Bloom filters in adversarial environments," *ACM Transactions on Algorithms (TALG)*, vol. 15, no. 3, pp. 1–30, 2019.

[42] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias, "SoK: Blockchain light clients," in *International Conference on Financial Cryptography and Data Security*, pp. 615–641, Springer, 2022.