DESCAN

Censorship-resistant indexing and search for Web3

de Vos, Martijn; Ishmaev, Georgy; Pouwelse, Johan

DOI
10.1016/j.future.2023.11.008

Publication date
2024

Document Version
Final published version

Published in
Future Generation Computer Systems

Future Generation Computer Systems 152 (2024) 257–272

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

DeScan: Censorship-resistant indexing and search for Web3

Martijn de Vos *, Georgy Ishmaev, Johan Pouwelse

Distributed Systems, Delft University of Technology, Van Mourik Broekmanweg 6, Delft, 2624KM, Zuid-Holland, The Netherlands

ARTICLE INFO

Keywords:
Decentralized systems
Decentralized search
Web3
Blockchain technology
Censorship resistance
Skip graph

ABSTRACT

The popularity of blockchain technology has bootstrapped many "Web3" applications, e.g., Ethereum and IPFS, that apply distributed ledger technology to store transactions. The amount of transactions generated and stored in such Web3 applications is significant and, in its raw form, usually not searchable by users. Existing Web3 transaction indexing and search engines are predominantly centralized and, therefore, can manipulate search results or censor particular queries. With the proliferation of Web3 transactions and applications, a decentralized and censorship-resistant search primitive is becoming essential.

We present DeScan, a decentralized and censorship-resistant indexing and search engine for Web3. Users index their local Web3 transactions using custom rules that output triplets. Generated triplets are bundled in a distributed transaction graph that is searchable by other users. To coordinate search and distribute the storage of the transaction graph over peers in the network, we build upon a Skip Graph (SG) data structure. Since the Skip Graph does not provide any resilience against adversarial peers that censor searches, we propose four modifications to improve its robustness. We implement DeScan and conduct experiments with up to 12 800 peers and 10 million Ethereum transactions. Our experiments show that DeScan with our modifications enabled can tolerate 20% adversarial peers and 35% unresponsive peers without disruption. Moreover, we find that searches in DeScan are usually completed well within a second, even when the network grows. Finally, we show that storage and network costs are evenly distributed amongst peers as the network grows.

1. Introduction

Recently, there has been an stark increase in Web3 applications driven by success of decentralized ledger technologies [1,2]. While Web3 is a broad label, at the moment it de-facto refers to decentralized and permissionless applications such as blockchains and blockchain-based applications.

These Web3 applications can generate large volumes of transactions. For example, Ethereum, one of the most popular Web3 platforms, persists over 1 TB of transactions in its blockchain.[1] Web3 data generation outpaces the development of tools for indexing and search. Currently, indexing of and searching for Web3 data is predominantly handled by centralized services, e.g., Etherscan [3] and Infura [4]. This is contradictory given that independence of centralized parties is a key focus of Web3 in general. Even though many users critically rely on centralized services to track Web3 transactions, we argue that sustainable growth and success of Web3 ecosystems require data management solutions that provide the same degree of *decentralization and censorship resistance* as offered by Web3 application themselves.

Centralized Web3 search services may need to comply with regulatory requirements and may be forced to censor or taint specific transactions. This happened recently with Infura, which now actively blocks access to transactions in the Tornado Cash application, a mixer for Ethereum [5]. Generally speaking, *blockchain accessing services* with censorship capacity are a fundamental bottleneck in any permissionless solutions that aims to provide censorship resistance as a system property [6].

Censorship capacity can also invite manipulation for economic gains, for example, by withholding transactions to a smart-contract based marketplace [7]. This can also enable secondary markets for censorship called Censorship-as-a-Service [8]. Censorship can happen at different layers of a Web3 system: at the network and consensus layer (e.g., censoring transactions) or at the application layer (e.g., censoring transaction metadata).

Decentralization is often seen as a solution for the problem of censorship [9]. However, decentralization is a broad spectrum of system designs rather than one specific approach, and different types of solutions can present various trade-offs.

We find that in the context of transaction search and indexing on public blockchains there is a trade-off between censorship resistance and computational requirements. On the one hand, decentralized

* Corresponding author.
E-mail address: martijn.devos@epfl.ch (M. de Vos).

[1] See https://etherscan.io/chartsync/chaindefault.

https://doi.org/10.1016/j.future.2023.11.008
Received 9 December 2022; Received in revised form 21 June 2023; Accepted 7 November 2023
Available online 10 November 2023

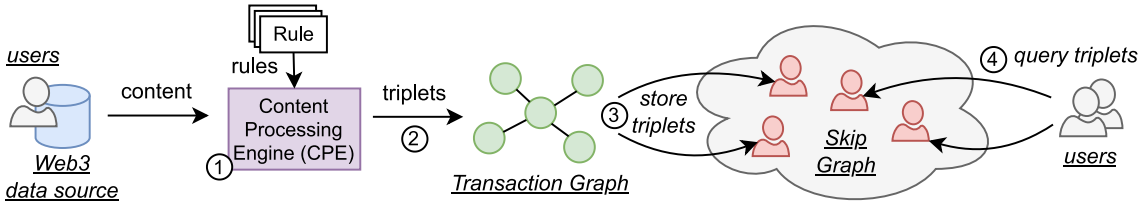


Fig. 1. Indexing and searching Web3 transactions with DeScan. The Content Processing Engine (CPE) indexes content retrieved from a local Web3 data source, e.g., an Ethereum RPC interface (step 1). The CPE generates triplets from Web3 content (step 2) that together form a transaction graph (TG). Each triplet is stored by multiple, random peers in a decentralized overlay (step 3). We use a Skip Graph data structure to assign triplets to peers. DeScan users can issue queries in the decentralized overlay to obtain triplets associated with particular content (step 4).

solutions based on local-first principles [10], require users to download all transactions in a particular Web3 application which can then be indexed and searched locally, e.g. by running a full blockchain node. While this sidesteps censorship concerns, it also requires users to download all these transactions and build local indices, which can introduce significant storage and computation overhead. On the other hand, other decentralized solutions for indexing and search based on peer-to-peer storage may allow peers to index transactions in a distributed manner [11]. This distributes the storage and computation burden over the peers in the network, enabling search functionality for low-resource devices, e.g. mobile phones. However, in this approach searching critically depends on other peers and thus is vulnerable to a censorship attack by adversarial peers. Ensuring censorship resilience in peer-to-peer decentralized networks for Web3 applications is still largely unexplored. This work therefore answers the following question: *How can we design, implement, and evaluate a Web3 indexing and search system that is decentralized and resistant to censorship by adversarial peers?*

We present DeScan, a decentralized indexing and search engine that is robust against adversarial peers that attempt to censor content, offers sub-second search latency, and is highly scalable as the network grows. Fig. 1 shows how users can index and search Web3 transactions with DeScan. DeScan includes a Content Processing Engine (CPE) that indexes content sourced from a local Web3 data source (step 1), such as an Ethereum RPC interface or an IPFS database. The CPE outputs triplets, which are three-tuples that denote relations between entities, e.g., "block X *contains* transaction Y". These triplets together form a *Transaction Graph*, which is a graph structure that stores transactions and their metadata. Developers can implement custom indexing logic, *rules*, that prescribe which triplets are generated from particular Web3 content. Rules enable DeScan to index transactions stored in different Web3 applications and enables DeScan to build a single, unified Transaction Graph. Triplets are digitally signed by their creator and users can verify the integrity of triplets retrieved from the network.

Triplets are stored by peers in a decentralized overlay (step 3 in Fig. 1). We build upon a *Skip Graph* data structure [12] to determine which peer should store particular triplets. A Skip Graph is a distributed data structure that offers searching functionality in peer-to-peer networks. Each peer in the Skip Graph maintains a local routing table and route incoming message to neighbouring peers in the routing table. Joining, leaving, and searching in a Skip Graph can be completed with logarithmic time and message complexity. We argue that these properties makes the Skip Graph a promising primitive to build a decentralized Web3 search engine. We consider the analysis and integration of incentive mechanisms to store and route data beyond the scope of this work. However, DeScan can be extended with different bandwidth and storage sharing incentives, including reciprocity-based mechanisms and transaction fees [13].

At the same time, the naive design of a Skip Graph is vulnerable to adversarial peers that attempt to censor searches performed by other peers [14]. Since each search is routed through a single path, any adversarial peer can, for example, refuse to forward an incoming search message for a particular transaction to other peers (censorship) or respond to the search initiator with a manipulated result. To address this key issue, we replicate triplets over multiple peers and have users issue multiple queries in multiple Skip Graphs simultaneously. We fully implement DeScan and our modifications. Our experiments demonstrate that these modification make DeScan robust against both adversarial and unresponsive peers, effectively addressing the issue of censorship when querying triplets in DeScan. These experiments also show that triplet queries can be completed within a second, and that network and storage overhead becomes evenly spread over the available peers as the network size grows. This makes it feasible to implement decentralized indexing in permissionless blockchains, such as Ethereum, on the basis of light clients [15]. This is a novel approach as at the moment indexing solutions are either centralized or require running a full node client with significant hardware requirements.

In summary, our contribution is four-fold:

1. We present DeScan, a decentralized and censorship-resistant indexing and search engine for Web3 (Section 4). DeScan indexes Web3 content into a Transaction Graph, which is stored by peers in the network. Search in DeScan is coordinated in a decentralized manner using a Skip Graph.
2. We identify and apply four improvements that make DeScan and the Skip Graph robust against both unresponsive peers and adversarial peers that attempt to censor access to Web3 transactions (Section 5).
3. We fully implement DeScan and make its implementation available on GitHub (Section 6.1).[2]
4. We conduct experiments with an Ethereum dataset to evaluate the robustness of DeScan against unresponsive and adversarial nodes, and to quantify query latency and overhead (Section 6.2–6.6). Our results reveal that DeScan tolerates 20% adversarial peers or 50% unresponsive peers with minimal degradation of quality-of-service.

2. Background and related work

2.1. Indexing and search in Web3

As Web3 applications generate and store increasing volumes of transactions, indexing and searching these transactions is also becoming increasingly important. Many blockchain fabrics provide an explorer that enables end-user to monitor transactions on the blockchain and search for them. Prominent examples are Etherscan [3] and the blockchain.com explorer [16]. These platforms continuously monitor on-chain transactions, index them, and make them searchable by end users. These platforms maintain central servers to index Web3 transactions and serve search queries, and therefore allow operators to censor the access to particular content. In comparison, peer-to-peer networks are considered more robust against censorship than centralized servers [7]

[2] See https://github.com/devos50/descan.

DeScan is related to four existing systems that index Web3 transactions: DeScan, The Portal Network, TrueBlocks, and The Graph. We describe each of these systems below and compare these solutions with DeScan.

DeSearch [17] is a search engine designed to index and search transactions in decentralized Web3 services. By using secure hardware, DeSearch enables users to verify the correctness and completeness of search results and provides privacy by keeping search queries hidden from workers. The DeSearch network consists of crawlers that fetch transactions from Web3 sources, indexers that index crawled transactions, and queries that process and respond to search requests. In line with this work, DeSearch focuses on censorship and manipulation of search results. An important difference between DeSearch and DeScan is that DeSearch critically depends on trusted hardware which is sparsely available in consumer hardware and therefore requires dedicated infrastructure. Another difference is that crawlers, indexers and peers in DeSearch require significant resource availability. DeScan is designed such that peers with lower resource capacities can also participate.

The Portal Network [18] is closely related to our DeScan approach. The project aims to build a fully decentralized network that enables lightweight access to historical Ethereum transactions by exposing the functionalities of the Ethereum RPC API. The Portal Network is in an early design stage and is built upon a custom DHT overlay. Even though DHTs have been widely used as underpinning primitive for storage and search of data in decentralized networks, it is known that without additional countermeasures, adversarial peers can sabotage the DHT [19,20]. In contrast to DeScan, The Portal Network does not focus on the attack-resilience of the indexing and search layer. Understanding and improving the attack resilience of different architectures other than DHTs, such as Skip Graphs, can eventually lead to more diverse and robust Web3 solutions. Another difference is that The Portal Network is exclusively built around Ethereum transactions, whereas DeScan enables transaction indexing across different Web3 applications.

TrueBlocks [21] is a desktop-first client for indexing and searching Ethereum transactions. By default, it requires users to setup and maintain an Ethereum full node. This solution provides censorship resistance, assuming the Ethereum full node is also censorship-resistant. TrueBlocks also offers a compromise solution that uses indices stored on the IPFS network [22] for users unable or unwilling to maintain a full node. However, the speed of transaction indexing by this solution is slightly behind the main network, and the issue of censorship resistance is outsourced to IPFS. In contrast to DeScan, TrueBlocks is oriented towards power users, and being a desktop-first client makes TrueBlocks currently unsuitable for integration in low-resource clients such as mobile devices.

The Graph [23]is an indexing protocol for accessing on-chain transactions (e.g., Ethereum) and data in decentralized storage networks (e.g., IPFS). The Graph enables developers to write *subgraphs*, small scripts written in the GraphQL language that index transactions in particular Web3 applications. This idea is similar to building the Transaction Graph through rules in DeScan. A difference with DeScan is that subgraphs generate isolated information, whereas DeScan builds a unified Transaction Graph with all data stored in a single peer-to-peer network. Another distinction is that The Graph is built around monetary incentives and native tokens used to reward indexers (queried by users) and curators (verifying the quality of indexed data). When performing a search in The Graph, end users remunerate indexers and curators for their services with fees. The Graph in the current version is primarily oriented at smart contract developers and power users for whom paid query services are economically rational. Unlike The Graph, DeScan aims to provide free query functionality to end users.

2.2. Censorship resistance in Web3 systems

The problem of censorship resistance in Web3 is an under-explored topic. Wahrstätter et al. provide a comprehensive review of the current challenges regarding censorship of transactions in Ethereum [5]. Wang et al. consider some security implications of censorship for permissionless protocols [7]. Tithonus is a network-level protocol that builds on the Bitcoin protocol to provide censorship-resistant communication mechanisms [6]. It provides a communication protocol for Bitcoin clients in a setting where an adversary can control (inspect, inject, suppress) the Internet communications of users within an area. Redbelly is a leaderless Byzantine consensus mechanism that addresses censorship of transactions by misbehaving nodes in a permissionless blockchain [24]. To the best of our knowledge, however, no systems focus specifically on censorship-resistant peer-to-peer search and indexing of blockchain transactions.

2.3. Prior work on P2P overlay search

Indexing and search in peer-to-peer networks has received significant attention from the academic community. Gnutella, one of the earliest peer-to-peer networks, floods search queries through all nodes in the network [25]. This basic approach, however, turned out to have limited scalability and would quickly result in congestion and performance issues as more peers join the network [26]. This inspired the development of indexing and search solutions based on structured overlays such as Distributed Hash Tables (DHTs) [27].

Most traditional structured overlays such as Chord [28] and Pastry [29] are not designed to be censorship-resistant. Since their introduction there have been efforts to devise search algorithms in structured overlays. Fiat et al. present the design of an overlay network in which the probability of content availability remains high even after an adversary removes arbitrary large fractions of peers in the network [30]. Augustine et al. describe a distributed algorithm for the decentralized storage and retrieval of documents under adversarial churn [31].

Secure Routing Strategies. A key focus of our work is to achieve robust search in a Skip Graph structure in the presence of adversarial nodes. This is closely related to the problem of secure message routing in structured overlays [32]. Castro et al. discuss various solutions to deal with adversarial peers that attempt to disrupt a decentralized overlay by not forwarding incoming messages [20]. The ideas discussed in [20] are closely related to DeScan since dealing with adversarial peers during a search operation is the key focus of our work. Bypass is a mechanism that avoids sending messages to adversarial peers when routing in a DHT structure by routing queries to redundant nodes at each step of the search [33]. It then avoids forwarding subsequent messages to peers that are considered unreliable. Bypass uses iterative queries whereas DeScan uses a "receive-and-forward" message passing model that incurs lower latency and communication cost.

In Section 5, we improve the robustness of the Skip Graph with several modifications. Some of these modifications have been inspired by prior research on structured overlays. Several works leverage acknowledgement messages during a search operation to detect and bypass unresponsive nodes [34]. Other works route search requests over multiple, disjoint paths to increase resilience against failures and dishonest peers [35–37]. Extending the routing table of peers, i.e., having each peer track more neighbours in the network, has been used to reduce the latency of search operations [38,39]. While not all modifications proposed in Section 5 are unique, we are the first to apply the combination of these techniques to the Skip Graph data structure and theoretically and empirically analyse their robustness against unresponsive and adversarial peers.

2.4. Transaction graphs

DeScan uses a *transaction graph* to index and structure Web3 transactions. A transaction graph is a graph that embeds on-chain transactions, their metadata, and relations between transactions. More specifically,

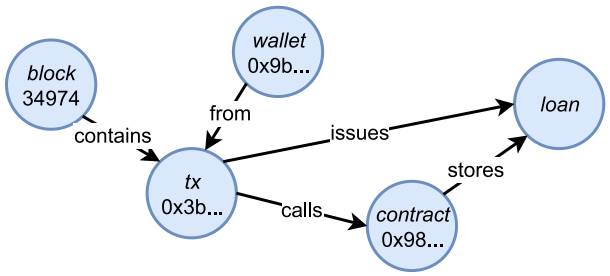


Fig. 2. A small transaction graph, reconstructed from transactions on the Ethereum blockchain. The nodes in this transaction graph describe entities such as blocks, wallet addresses, transactions, contracts, and loans. Edges between entities describe relations, e.g., a block *contains* a transactions. This information is extracted from the processed transactions.

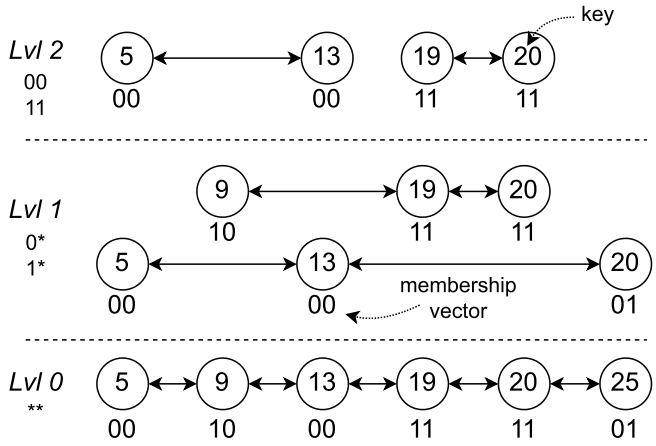


Fig. 3. A skip graph with six nodes. Nodes at level i are linked if the prefix of their membership vector is at least i bits. Level 0 contains all nodes, ordered ascendantly by their key.

the transaction graph is an acyclic directed graph (DAG) that consists of triplets in the form (*subject*, *relation*, *object*). Fig. 2 shows a small transaction graph constructed from transactions on the Ethereum blockchain. The nodes of this transaction graph contain entities such as blocks, transactions, and wallet addresses. The idea of the transaction graph is inspired by knowledge graphs in the domain of the semantic web where they are used as a means to structure knowledge and to make Internet pages machine-readable [40,41]. Knowledge graphs are also, for example, used to enhance results in search engines like Google and Bing [42].

We argue that a transaction graph is a suitable primitive to index Web3 transactions for the following two reasons. Firstly, it provides the means to bundle indexed transactions from different Web3 applications and in different formats into a single data structure. Besides data such as blocks and transactions, transaction graphs can also capture more semantically rich data such as market orders in particular smart contracts or even relations between transactions issued in different Web3 applications. Secondly, a graph in general is increasingly being used for advanced data analytics in the Web3 domain, e.g., for fraud detection [43], anomaly detection [44], and recommendation [45].

At the same time, constructing a transaction graph with accurate and meaningful information is challenging. In DeScan, triplets in the transaction graph are generated by small scripts that process specific Web3 transactions (also see Section 4.1). We assume that these scripts are peer-reviewed and audited by developers to generate accurate triplets. Since all generated triplets with DeScan are signed by their creator, one can build a reputation mechanism on top of DeScan to identify peers that made meaningful and accurate contributions to the transaction graph [46]. Additionally, several proposals utilize machine learning techniques to automatically add triplets to the transaction graph and remove erroneous triplets from it [47]. Blockchain technology has also been proposed to aid the construction of a transaction graph. For example, Wang et al. store a knowledge graph in an Ethereum smart contract and remunerate volunteers that propose correct modifications to this knowledge graph [48].

2.5. Skip graphs

DeScan uses a *Skip Graph* to distribute the storage of triplets amongst peers in the network, and to handle searches by peers for particular triplets [12]. The Skip Graph is a distributed data structure that is based on skip lists [49] and enables search in a decentralized overlay with logarithmic message and time complexity. It consists of one or more *nodes*[3] where each node n has a key, denoted by n_k, and a membership vector, denoted by n_v. The membership vector is usually a random bit string. A node can represent a physical machine or a peer in a peer-to-peer network, but it can also be more granular and describe some data that is stored by a particular peer. In DeScan, each node in the Skip Graph maps to a peer in the decentralized overlay. Nodes are linked together on different *levels* and each level contains one or more double-linked lists. Two nodes m and n are linked at level i if the prefix of m_v and n_v have at least i bits in common. Each peer maintains a routing table and is aware its immediate left and right neighbour on each level in the Skip Graph.

Fig. 3 visualizes a Skip Graph with six nodes and three levels. Membership vectors in Fig. 3 are depicted as bit strings. All nodes are included on level 0 and are linked ascendantly based on their key. The node with key 9 on level 1 references the node with key 19 since the length of their common membership vector prefix is at least one. Likewise, the nodes with key 5 and 13, respectively, are included in each other routing tables on level 2 since they have two bits of their membership vector prefix in common. In general, the distance between the keys of two linked nodes increases on higher levels and links become more "sparse".

Search. The Skip Graph structure enables peers to find nodes that have a particular key, or are close to one. A search in a Skip Graph for a particular key will either result in the node with the key being searched for, or in the node with largest key smaller than the search target. The search algorithm in a Skip Graph is similar to that of a Skip List, except that search requests are routed between peers in the network. A Skip Graph search initiated by node n for key k proceeds as follows. The search start at the top-most level in the routing table of n that includes n and traverses downwards through the levels in the Skip Graph without overshooting the search target. n starts by determining whether the search should be forwarded to a left or right neighbour. This depends on n_k: the search is forwarded to a right neighbour if $n_k < k$ and to a left neighbour if $n_k > k$. The search traverses downwards through the levels in the routing table of m until a neighbouring node b with key $b_k \leq k$ is encountered, when left-routing the search, or until a neighbouring node b with key $b_k \geq k$ is encountered, when right-routing the search. Assume that n finds that the search request should be forwarded to node m next. n then sends a `Search` message to m, containing the key being searched for, the current level of the search in the routing table, and the details of the search originator. If the search reaches a node p with key k, or if p is unable to forward the search request to a next eligible node, p sends a `SearchResponse` message to n that includes the details of node p.

We further illustrate a Skip Graph search with an example. Assume that in the Skip Graph in Fig. 3 node 20 initiates a search for the node

[3] When using the term *node*, we specifically refer to a node in a Skip Graph. We use the term *peer* to refer to a peer in the decentralized overlay.

with key 5. This search starts at level 2 in the routing table of node 20 and is forwarded to node 19 on level 2. Since node 19 does not have a left neighbour on level 2, the search drops down to level 1 in the routing table of node 19. Node 19 forwards the search to node 9 on level 1. Node 9 then drops down the search to level 0 since it does not have a left neighbour at level 1, and then forwards the search to node 5. Finally, since the key of node 5 is exactly the key being searched for, node 5 sends a search response to node 20. This completes the search.

A search in a Skip Graph can "skips" large parts of the key space at higher levels in the routing tables. Searches in a Skip Graph are expected to take $O(log\ n)$ time using $O(log\ n)$ messages, where n is the number of peers in the network [12]. Additionally, nodes only needs to store $O(log\ n)$ neighbours in their routing tables which makes the Skip Graph an efficient and scalable data structure for resource management in decentralized networks.

3. Problem formulation and system model

We now formulate the problem that this work addresses and present our system model.

3.1. Problem formulation

The key focus of this work is on *censorship resistance* when storing and retrieving Web3 transactions in decentralized overlays. Censorship is a crucial issue in Web3 since individual peers can be motivated to prevent other users from accessing specific transactions. For example, if a peer stores a current-best order in a decentralized marketplace, it is in the economic interest of that peer not to share this information with other users. A user can then exploit its exclusive access to information, e.g., by front-running on the order [50]. Censorship is trivial in centralized platforms such as Etherscan since the organization behind these platforms controls all data flows. Since DeScan is a fully decentralized system, we define censorship as *the ability of an individual peer, or a group of colluding peers, to prevent other users from accessing particular Web3 transactions*.

We argue that there is a trade-off between censorship resistance and computational requirements. On the one hand, some existing solutions index all Web3 transactions locally and conduct searches in the local database. While this sidesteps censorship concerns, it also requires users to download on-chain transactions and build local indices, which can introduce significant storage and computation overhead. For example, indexing the entire history of on-chain Ethereum transactions incurs storage requirements in the order of terabytes. On the other hand, fully decentralized solutions for indexing have peers jointly build a distributed index. While this improves load balancing across peers, searching now directly depends on other peers and is still vulnerable to a censorship attack.

In DeScan, peers conduct searches in a Skip Graph to store and retrieve indexed Web3 transactions. Using a Skip Graph in this context is not trivial as naive implementations can be vulnerable to adversarial peers that disrupt incoming search requests from other peers [14]. An adversarial peer can, for example, refuse to forward an incoming search message for a particular transaction to other peers (censorship) or reply with a manipulated result. Skip Graphs are also vulnerable to unresponsive peers as any unresponsive peer on a "search path" will fail the search from reaching the target node being searched for. Therefore, the technical challenge of this work is to ensure that searches in a Skip Graph succeed with high probability, even in the presence of adversarial or unresponsive peers.

3.2. System model and assumptions

Peers in DeScan are connected together in a Skip Graph overlay. We assume that peers a bootstrap server (a common assumption) and join the Skip Graph after bootstrapping (also see Section 4.2). We use the stateless UDP networking protocol for all communication between peers. Each peer in the network possesses a cryptographic keypair. The public key is used to identify the peer and the private key is used to digitally sign outgoing network messages. We make standard assumptions and assume that all cryptographic operations performed by DeScan are secure. DeScan also assumes that the public key by peers are generated by a uniform distribution. This is required to ensure that network and storage overhead are balanced over the peers in the network. The membership vector of each peer is derived from its public key using a deterministic function.

Threat Model. The main focus of this work is on dealing with adversarial peers during search operations that attempt to block access to particular transactions. Adversarial peers are able to drop incoming messages, forward incoming messages to arbitrary other peers, or reply to the search originator with a wrong search result. They are, however, unable to impersonate other peers by forging digital signatures. We more formally define *censorship resistance* as the highest fraction of adversarial peers DeScan can tolerate before the query success rate drops below a particular target rate, say 95%.

A prominent attack in any decentralized system is the Sybil Attack in which an attacker joins the network under many different identities [51]. This attack can enable censorship in DeScan, e.g., by strategically positioning Sybil identities on the search paths to particular content. Since this attack is not the key focus of our work, we assume that there is a mechanism that prevents unlimited identity creation. For example, one can use a peer-to-peer reputation mechanism to achieve a desirable degree of Sybil tolerance [46]. Alternatively, DeScan could track the behaviour and properties of connected peers on the network layer and maintain reputation scores to identify suspicious behaviour [52]. There is also a possibility that peers will collude with many other peers to censor transactions. We do not consider this to be a realistic scenario, however, given practical difficulty of coordination if the network is large, and the lack of such incentives for peers in a heterogeneous permissionless network.

4. DeScan : Decentralized Web3 indexing and search

This section outlines the search and indexing algorithms of DeScan. We first explain in Section 4.1 how the collective indexing of Web3 transactions by peers results in a global transaction graph (Step 1 in Fig. 1). In Section 4.2 we then outline how peers store parts of the global transaction graph, and how peers can retrieve parts of the transaction graph. The system as described in this section is not censorship-resistant yet; we present the modifications to DeScan to achieve censorship-resistance later in Section 5.

4.1. DeScan : Web3 transaction indexing

The indexing process is performed by the *Content Processing Engine* (CPE). The CPE first loads raw transactions from local Web3 sources, for example, local databases or APIs, and then transforms these transactions into nodes and edges of a transaction graphs. This transformation is specified by *rules*, functions that take some content as input and outputs nodes and edges of the transaction graph. Fig. 4 shows an example with two rules (in green and blue, respectively) that indexes raw Ethereum block data into a transaction graph. One rule processes the "miner" field in the block data and the other rule processes the transactions contained in the block.

Rules are written by developers and their implementations can be shared on platforms like GitHub. Rules are also deterministic, meaning that the resulting transaction graph when applying a rule is always the same for the same input. We envision that each rule processes particular content, for example, blocks, transactions, or smart contracts on the Ethereum blockchain. Users running the DeScan software can import and enable rules to be applied during the indexing procedure.

Algorithm 1 The logic of the Content Processing Engine (CPE) executed by user u.

```
1:  Require: private key u_pk
2:  K ← LOADLOCALKNOWLEDGEGRAPH()
3:  P ← LOADPROCESSEDCONTENT()
4:  M ← LOADMERKLETREES()
5:  R ← LOADRULES()
6:  C ← LOADCONTENT()
7:
8:  for c in C do
9:      T_c ← {}                          ▷ the set T_c contains generated triplets
10:     for r in R do
11:         if (c.id, r.id) in P then
12:             continue                  ▷ c is already processed by r
13:         T_c^r ← r.EXECUTE(c)          ▷ apply rule r to c
14:         T_c.APPEND(T_c^r)
15:         P[(c.id, r.id)] ← T_c^r
16:
17:     K.ADD(T_c^r)
18:     m_c ← CREATEMERKLETREE(T_c)
19:     s ← SIGN(m_c.ROOT(), u_pk)
20:     M[c] ← m
21:     p ← SEARCHPEER(c.id)
22:     STORETRIPLETS(p, T_c, m, s)
```

We show the main logic of the CPE in Algorithm 1. The CPE operates on *content items* where each content item is a two-tuple with raw data and an identifier (e.g., a hash of the data). These identifiers end up as nodes in the transaction graph, as also shown in Fig. 4. When starting DeScan, the CPE first loads the local transaction graph (K), the identifiers of the content items that are already processed (P), the installed rules (R) and the content items available for processing (C). This is shown in line 2–4 in Algorithm 1. The CPE iterates over all content items $c \in C$ and applies all installed rules (line 9–21). If c has already been processed by a particular rule r, r is not applied again (line 10–11). Otherwise, r is applied to c by calling the `execute` method (line 12).

The `execute` method is expected to return a set of *triplets*, T_c^r. A triplet codifies a statement about an entity, for example, "The block with hash x contains the transaction with hash y". A triplet can be seen as two endpoint nodes and an edge in the transaction graph. Triplets are inspired by existing practices in the semantic web that are using RDF triplets to structure and express semantic data. A triplet t in DeScan consists of the following four fields:

$$t = \langle contentID, relation, tail, ruleID \rangle$$

For example, consider the triplet generated by rule 1 in Fig. 4 (in green). The `contentID` field of this triplet is "*0x3a...*", the `relation` field is "miner", and the `tail` field is "*0xf9...*". Finally, the `ruleID` field contains the identifier of the rule that generated t. For presentation clarity, the `ruleID` fields are not explicitly shown in Fig. 4.

In Algorithm 1 the triplets generated for content c by rule r are stored in the list T_c^r. All resulting triplets for content item c are merged into T_c and stored in P. When all rules have been applied to some content, the resulting triplets are appended to the local transaction graph K.

To ensure authenticity and prevent users from generating arbitrary triplets, each triplet will be digitally signed by its creator. Instead of embedding each signature in a triplet t, the CPE instead constructs a Merkle Tree for content item c with the hashes of each triplet in T_c, sorted ascendantly, as leaf node in the tree (line 18). A user u then signs the root hash of the Merkle Tree with their private key u_{pk}, which effectively acts as an irrefutable proof that u has generated these

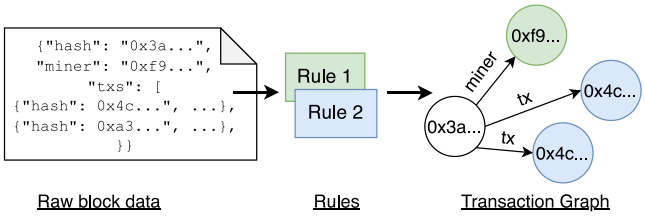


Fig. 4. Indexing raw Ethereum block data using two rules (shown in green and blue). In this example, one rule processes the miner field in the data and the other rule processes the transactions contained in the block. Applying these rules output a transaction graph.

triplets. The Merkle Tree m_c is shared together with the set of generated triplets T_c for content item c with other users. Signing a Merkle Tree only requires a single signature for each content item and significantly reduces storage requirements if a user generates millions of triplets, compared to when signing each triplet individually.

In contrast to related systems for Web3 transaction indexing, DeScan avoids network-wide replication of the transaction graph which can be costly in terms of storage and network overhead. Instead, the transaction graph is distributed over all peers in the network. More specifically, the triplets associated with each content item c are stored by a particular peer p. To determine which peer should store triplets associated with c, a user will conduct a search in the Skip Graph overlay (line 21). This process is further explained in Section 4.2 and for now we assume that peer p is responsible for storing triplets associated with c. The CPE then invokes the `storeContent` method that will store the triplets T_c at peer p.

4.2. DeScan : *Web3 data storage and search*

We now describe how DeScan distributes the triplets in the transaction graph over the different peers in the network, and how users search for particular triplets.

Constructing a Skip Graph. DeScan uses a Skip Graph to coordinate the storage of triplets and to route search queries to designated peers. Peers that index Web3 transactions join this Skip Graph overlay. The key of their Skip Graph node is the hash of their private key. Since the focus of this work is on censorship-resilient searching of Web3 transactions, we assume that DeScan uses established algorithms for the construction and maintenance of the Skip Graph overlay. For example, Aspen et al. present a Skip Graph construction algorithm where each node first inserts itself on the first level and then attempts to locate its neighbours on higher levels [12]. A peer u only needs to know one other peer for u to execute the joining algorithm. The algorithm described in [12] is also used in our implementation. Later work introduces more advanced construction algorithms that are also able to deal with unresponsive peers. For example, Riko et al. describe SKIP+, a self-stabilizing Skip Graph that uses redundant neighbour links and that can repair itself when there are inconsistencies in the Skip Graph [53]. Gguerraoui et al. introduce a fault-tolerant overlay construction approach based on message gossiping [54]. All of the above algorithms are capable of constructing the overlay with $O(log\ n)$ time and message complexity, where n is the total number of peers in the network.

Storing Triplets in the Network. A peer is responsible for storing triplets associated with some content item c if the identifier of c is the closest to their public key. Assume that a peer p has generated triplets T for c and is going to store these triplets in the network. To determine which peer should be storing T, we conduct a search in the Skip Graph overlay for $H(c.id)$, which algorithm is given in Section 2.5.

In general, a peer u with key k and with a right neighbour on level 0 with key k_n in the Skip Graph is responsible for storing items that map to a key in the range $[k, k_n)$. In other words, any search in the

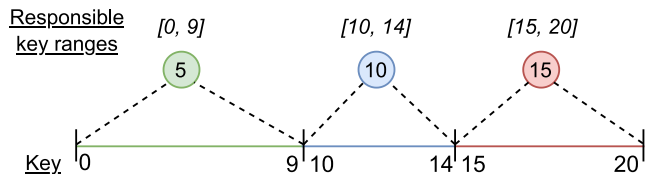


Fig. 5. Example mapping from key to peers for a key with range $[0, 20]$, and with peers nodes with keys 5, 10, and 15, respectively.

Skip Graph for a key in the range $[k, k_n)$ will result in peer u. If peer u has the lowest key of all peers, it is responsible for key range $[0, k]$ and likewise, if u has the highest key, it is responsible for key range $[k, k_m]$ where k_m is the highest possible key. We show an example of this in Fig. 5 that shows the responsible key ranges for each of the three peers with keys 5, 10 and 15. Since node 5 has the lowest key in the network, it is responsible for the key range $[0, 9]$. Depending on the distribution peer keys, a peer might be responsible for a larger or smaller key range than others.

Peer u stores its generated triplets T, associated with content c, at peer n as follows.

1. u first sends a `storageRequest` message to n. This message contains the identifier of c, denoted as $c.id$. Upon receiving the `storageRequest` message by u, n verifies that it is responsible for storing the triplets associated with c. It does so by inspecting the keys of its level-0 neighbouring nodes in its Skip Graph routing table and by establishing that these neighbours are not closer to $c.id$. If n is the designated peer to store triplets associated with c, n replies to u with a `storageAccept` message. Otherwise, n replies to u with a `storageRefusal` message. These acceptance and refusal messages prevents malicious users from placing content at arbitrary peers.
2. If u has received a `StorageAccept` message from n, it will send a `TripleHashes` message to n. This message contains $c.id$, a list with all hashes of the triplets in T, and the signature s of the root of the Merkle Tree associated with T. Upon reception of the `TripletHashes` message by n, n reconstructs the Merkle Tree from the hashes in the message and checks the validity of s using the public key of u. If s is invalid, n sends a `storageRefusal` message to u. Otherwise, n sends a `TripleHashes` message to u with a list of hashes that n does not have. This additional `TripleHashes` message avoids duplicate transmission of triplets to n if another user already indexed the same content with the same rules.
3. Upon receiving a `TripletHashes` message by u, u sends triplets that n is missing in serialized form to n in a `Triplets` message. n verifies the validity of incoming triplets and whether the hash of each of these triplets is included in the earlier-sent Merkle Tree. n then appends the incoming triplets to its local transaction graph and replies with a `storageComplete` message to u. The Merkle Tree and corresponding signature of the root hash are stored by n alongside with the incoming triplets. This finalizes the storage procedure.

Querying Triplets. Users can query the network to retrieve triplets associated with some content c. We use the term *query* to refer to the action of retrieving triplets from other peers, and we use the term *search* when we talk about a search in the Skip Graph. A triplet query consists of two steps: (1) a search in the Skip Graph and (2) a request to the peer that holds the triplets. We outline each of the two steps below with an example where peer u queries triplets associated with c.

1. *Skip Graph Search.* First, u performs a search in the Skip Graph for key $c.id$ to determine the peer that stores the triplets associated with c. This search can fail, for example, if a peer on the *search path* is unresponsive or adversarial. Therefore, each Skip Graph search has a timeout value Δt which depends on the network characteristics and acceptable search latency. If this timeout triggers, the query fails and results in an empty set. This step requires $O(log\ n)$ time and message complexity.
2. *Requesting Triplets.* Assume that the search resulted in peer n. To retrieve the triplets associated with c from, u sends a `Retrieve` message to n, with $c.id$. When n receives the `Retrieve` message, n inspects its local transaction graph for all triplets and Merkle Trees associated with c. n then sends a `Triplets` message back to u, containing both the requested triplets, the Merkle trees, and the signatures of their root hashes. Upon receiving this `Triplets` message, u verifies the authenticity of the incoming triplets using the included Merkle Trees and signatures. Note that u is also able to detect if n did not send a particular triplet to u, or if u sent a triplet which hash is not included in any of the Merkle Trees. This step requires two messages between peers n and u, one `Retrieve` and one `Triplets` message. u now has received the requested triplets which completes the query.

In summary, a triplet query can be completed with $O(log\ n)$ time and message complexity.

Relocating Triplets under Churn. Peers joining and leaving the Skip Graph gracefully (informing other peers) can invalidate the placement of triplets. This subsequently can result in failed triplet queries. For example, if in Fig. 5 peer 12 joins the Skip Graph, it takes over responsibility for the key range $[12, 15]$ from peer 10. Any Skip Graph search for key 12 will now result in peer 10 which does not store the requested triplets. Likewise, if peer 10 leaves the Skip Graph, the responsibility for the key range $[10, 14]$ is passed over to peer 5.

Therefore, restoring the validity of triplet placement under churn requires additional logic. DeScan relocates triplets when peers are joining the overlay as follows. When peer n with key n_k has joined the Skip Graph, it sends a `TripletStoreSync` message to its left level-0 neighbour, say peer v with key v_k. When node v receives the `TripletStoreSync` message, it iterates over all stored triplets and determine the triplets that should be stored by the newly-joined peer n instead. These triplets and their associated Merkle Trees are sent to n in a `Triplets` message for storage and are subsequently deleted from the local transaction graph of v. As peers join DeScan and existing peers relocate (some of) their stored triplets to newly-joined peers, the storage burdens are effectively spread out over peers in the network. We experimentally evaluate this property in Section 6.6.

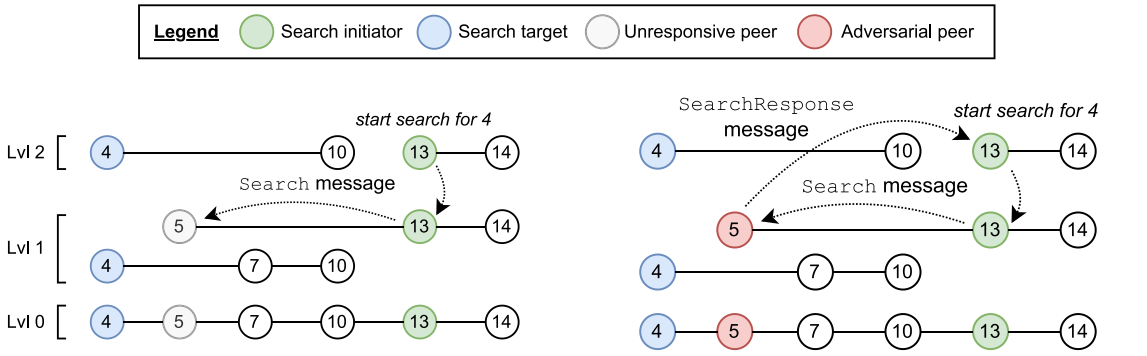
DeScan deals with leaving peers as follows. Before some peer n with key n_k leave the network, it first sends its stored triplets to the left level-0 neighbour, say node v with key v_k, in a `Triplets` message. This makes node v the new responsible node for the triplets priorly stored by n. Peer n then leaves the Skip Graph.

4.3. Extensions

We now discuss two extensions of DeScan. While these extensions are not required for the correct operation of DeScan, we believe they are able to improve the user experience and capabilities of DeScan.

Supporting Nodes with Limited Storage Capacities. DeScan stores triplets at random peers in the network. However, if a particular peer n is responsible for a large part of the key space, it might incur more storage overhead than available on n. Another possibility is that users have allocated a limited amount of disk space for DeScan storage, for example, 1 Gigabyte.

DeScan can perform a load-balancing operation when n is running out of storage, by moving some of its triplets to a level-0 neighbour in the Skip Graph, say node v. n then executes the triplet relocation algorithm described earlier and keeps track of the triplets that have been offloaded to node v. When n receives a `Retrieve` message from



(a) Unresponsive node 5 causing a timeout of the search initiated by node 13.

(b) Adversarial node 5 preventing the search initiated by node 5 from reaching node 4.

Fig. 6. The effect of unresponsive and adversarial nodes on a Skip Graph search.

node s for triplets that have been offloaded to node v, n informs s of node v. Subsequently, s then also sends a `Retrieve` message to v. Offloading triplets and querying offloaded triplets increases end-to-end latency of triplet retrieval since more round-trip messages are required to collect all triplets from multiple peers. A more similar algorithm that rebalances content in a Skip Graph structure as response to capacity limitations is provided in the work of Disterhöft et al. [55]. Even though this extension is not part of our implementation and experiments, dealing with storage limitations becomes important when devices with lower resource capacities, e.g., mobile phones, participate in DeScan.

Range Queries and Ordered Indexing. A distinct benefit of the Skip Graph structure is that it allows range queries, e.g., fetching all Skip Graph nodes in a key range [56]. This is possible since nodes are linked in incremental order on their key on level 0. For example, if a peer n wishes to retrieve all nodes in the key range $[40, 70]$, n first performs a search for node 40. n then iteratively queries the right level-0 neighbour of the resulting node, continuing until a node with a key equal to or higher than 70 is retrieved.

We can modify the indexing mechanism of DeScan to store ordered transactions at adjacent nodes in the Skip Graph, a mechanism we call *ordered indexing*. For instance, buy and sell orders can be stored based on an increasing price in nodes in the Skip Graph, and retrieved using the price range by end users.

5. Achieving censorship resilience and fault tolerance when querying triplets

The approach to query triplets, as described in Section 4, is not resistant against adversarial peers that attempt to censor particular triplets, nor has any robustness against unresponsive peers on a search path. We show how unresponsive and adversarial peers can make a Skip Graph search, and consequentially a triplet query, fail in Fig. 6. Fig. 6(a) shows how unresponsive node 5 causes the search for node 4, initiated by node 13, to timeout. Since peers forward incoming `Search` messages to other peers, any unresponsive peer will stall the search since a `Search` message will not reach its final destination. In Fig. 6(a), the `Search` message will never reach node 4.

Similarly, an adversarial peer can easily disrupt a Skip Graph search by sending a `SearchResponse` message to the search initiator with an incorrect search result, e.g., by reporting itself as the intended search result. This is shown in Fig. 6(b) where node 13 searches for node 4. Adversarial node 5 responds with a `SearchResponse` messsage to node 13, after which node 13 will send a `TripletsRequest` to node 5. Node 5 can now respond with a `Triplets` message containing no triplets, effectively preventing node 13 from obtaining the requested triplets.

To quantify the impact of adversarial peers on the success rate of queries, we experiment with the DeScan implementation (also see Section 6). We construct a Skip Graph with 10 000 nodes in which the key of each node is its numerical index, starting from 1. We make 10% of all joined peers adversarial, i.e., they will respond with an invalid search result as shown in Fig. 6(b). We conduct 1000 Skip Graph searches in total. Each query is initiated by a random honest peer and has another random peer as a search target. This experiment reveals that most Skip Graph searches, 58.4%, result in an incorrect node. This result shows that the usability of DeScan degrades quickly, even when a small fraction of peers is adversarial.

To achieve tolerance against both adversarial and unresponsive peers, we make four modifications to DeScan: (1) acknowledgements of `Search` messages, (2) extended routing tables, (3) replicating triplets on the storage layer, and (4) operating multiple Skip Graph. We describe each modification in the following subsections.

5.1. Acknowledgements of `Search` messages

Our first modification enhances the Skip Graph search algorithm described in Section 4.2. This modification allows peers to detect unresponsive peers during Skip Graph searches and update their routing tables accordingly. When a peer u forwards a `Search` message to a subsequent peer v, v will now also send a `SearchAck` message back to u. If v fails to reply to u with this message within a timeout, u will consider peer v unresponsive and remove it from its local routing table. This will prevent u from routing subsequent `Search` messages to v. To replace peer v in the routing table of u, u should locate another node to fill its place using the primitives provided by the Skip Graph construction algorithm. u will then send the `Search` message to the next eligible peer. This modification enables DeScan to deal with unresponsive peers. However, this modification might increase the total latency of triplet queries as any unresponsive peer on the search path requires a peer forwarding a `Search` message to wait for the timeout duration.

5.2. Extended routing tables

By default, a peer stores at most one left and right neighbour at each level in the local routing table. Our second modification extends the routing tables of peers such that each peer will maintain at most b left or right neighbours on every level. For example, in the Skip Graph shown in Fig. 6 and with $b = 2$, node 7 will store on level 0 both node 4 and 5 left as left neighbours, and node 10 and 13 as right neighbours. Maintaining multiple neighbours reduces the number of hops in Skip Graph searches since `Search` messages can now be routed quicker to a destination node. Consequentially, it also reduces the probability that a search hits an adversarial peer, assuming the fraction of malicious peers is fixed. This modification also provides increased re-routing opportunities of a search to other peers if unresponsive peers

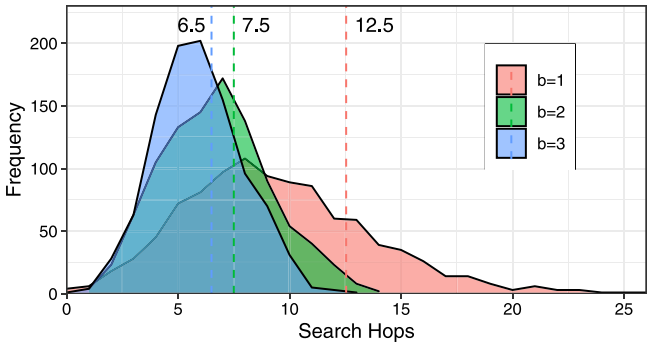


Fig. 7. The distribution of search hops for $n = 1600$ and $b = 1$, $b = 2$, and $b = 3$. For each value of b, we perform 1000 searches and annotate the average number of search hops with a dashed vertical line.

are encountered. However, this comes with additional storage and coordination overhead since additional nodes must be maintained in local routing tables.

To quantify the impact of this modification on the number of hops for each search, we experiment with $n = 1600$ and different values of b. We keep track of the number of peers each Skip Graph search traverses while excluding the search originator. Fig. 7 shows the impact of b on the number of hops in a search, for $b = 1$, $b = 2$, and $b = 3$. For each value of b we perform 1000 searches. We also annotate for each value of b the mean number of search hops with a vertical dashed line. Fig. 7 shows that the average number of search hops decreases significantly when increasing b from 1 to 2, namely from 12.5 to 7.5. Increasing b from 1 to 3 reduces the average number of search hops to 6.5, a reduction of 48.0% compared to $b = 3$. We also observe that increasing b further has diminishing returns on the number of hops per search.

The search algorithm discussed in Section 2.5 should be modified to support tracking multiple neighbours per level. Instead of forwarding a search to the immediate left or right neighbour at a particular level in the local routing table, a peer iterates over the list of left or right neighbours until it finds the best candidate to forward the search to. When left-routing the search, we iterate over the left neighbours until we find a node with the lowest key higher than or equal to the search target. When right-routing the search, we iterate over the right neighbours until we find a node with the highest key less than or equal to the search target.

5.3. Replicating triplets on the storage layer

Our third modification adds replication on the storage layer and ensures that triplets remain available when peers that store them become adversarial or unresponsive. By default, DeScan stores triplets associated with particular content on exactly one peer. Similar to unresponsive or adversarial peers encountered during a Skip Graph search, unresponsive or adversarial peers storing triplets will not respond with a `Triplets` message to a query initiator. Triplets stored on such peers then become unavailable.

To address this, we replicate triplets on $r > 1$ peer in the network, where r is the *triplet replication factor*. As r increases, the probability that at least one honest peer in the network stores particular triplets also increases. A peer querying triplets now conducts r parallel searches. Redundant storage improves resilience against unresponsive and adversarial peers but multiplies the storage overhead by r. Likewise, conducting multiple parallel searches in the Skip Graph increases network overhead.

To determine the r peers that are responsible for storing triplets associated with content c, a query initiator p compute $k_i = H(c.id \parallel i)$ where k_i is the i^{th} key of the peer responsible for storing these triplets, and i is an index in the interval $[0, r]$. p then conducts r parallel Skip Graph searches for each computed k_i and sends a `Retrieve` messages to each of the peers returned by the search.

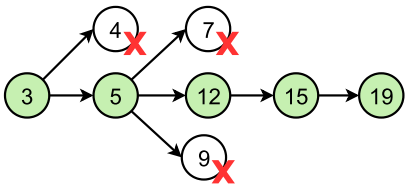


Fig. 8. The peers involved in a search operation initiated by node 3, searching for node 19. Nodes with a red cross are unresponsive.

5.4. Constructing multiple skip graphs

Our fourth modification involves the network constructing and maintaining s Skip Graphs. Each peer in the network derives s distinct membership vectors from their public key. When starting the DeScan software, a peer joins all s skip graphs with the same key but with different membership vectors. This modification makes it much more challenging for adversarial peers to place themselves strategically in the network since the links on level $x > 0$ differ significantly between the different Skip Graphs. Even though this requires peers to construct s Skip Graph overlays, we argue that these costs are reasonable since constructing the Skip Graph is an operation only performed upon startup of the DeScan software.

5.5. Analysis

We now analyse the algorithmic complexity and censorship resistance in DeScan. In our analysis, we assume that membership vectors are uniformly random, resulting in well-balanced Skip Graphs. First we derive the complexity of a Skip Graph search in terms of steps taken (i.e., the length of a search path), and then analyse the impact of our improvements on this complexity, which we denote by m. This complexity approximates the average number of messages we have to send to complete a search. To derive the number of steps required for a search, we do, analogue to [49], a backwards analysis of the search path. Let $C(j)$ denote the number of steps required to go up j levels in the Skip Graph. At each particular step during the search operation, the search was either coming from a left or right neighbour with 50% probability, or moved down from an upward level with 50% probability [49]. More formally, $C(j)$ is given by:

$$
\begin{aligned}
C(j) &= 1 + \frac{1}{2}C(j-1) + \frac{1}{2}C(j) \\
2C(j) &= 2 + C(j-1) + C(j) \\
C(j) &= 2 + C(j-1) \qquad (1)
\end{aligned}
$$

Eq. (1) shows that the expected number of steps at each level is 2. Analysis in [12,49] derives that a Skip Graph contains, on average, $log(n)$ levels where n is the total number of peers in the network, resulting in an expected number of steps $m = 2 \cdot log(n)$, or $O(log\ n)$. The message complexity, i.e., the number of messages sent between peers during a search operation in the Skip Graph, is also $2 \cdot log(n)$.

Acknowledging `Search` messages (see Section 5.1) increases the total number of messages required for a search operation when unresponsive peers are encountered but does not extend the length of the search path to reach the peer with the requested key. We show this in Fig. 8, visualizing how a search initiated by node 3 and looking for node 19 traverses through peers in the network. Node 4, 7 and 9 are considered unresponsive (indicated with a red cross) and the other peers are considered responsive (indicated in green). While the total number of messages increase compared to when all peers are responsive, the length of the search path to reach node 19 remains 5 regardless of encountering unresponsive peers during a search. In general, acknowledging `Search` messages requires two messages to be sent when taking a step during a search, and costs one message

when forwarding a `Search` message to an unresponsive peer (as unresponsive peers do not send acknowledgements). Let f_u denote the fraction of peers that are unresponsive. Assuming $m = 2 \cdot log(n)$, the expected number of messages required for a search becomes $2\ m + f_u \cdot m = 4 \cdot log(n) + 2f_u \cdot log(n)$, which still results in an overall $O(log\ n)$ message complexity.

Extending the routing tables (see Section 5.2) does not modify the expected number of levels but does enables a search to take "shortcuts" on a single level, therefore reducing the expected number of steps taken at each level. When deriving Eq. (1) we assumed that a search is equally likely to have originated from a left or right neighbour, as it originates from an upward level. Increasing b, however, decreases the number of peers visited on a single level and therefore increases the probability that a search came down from an upwards level. For a particular value of b, the probability that a search came from a left or right neighbour on the same level becomes $\frac{1}{2b}$. We rework Eq. (1) and integrate b as follows:

$$
\begin{aligned}
C(j) &= 1 + (1 - \frac{1}{2b})C(j-1) + \frac{1}{2b}C(j) \\
2b \cdot C(j) &= 2b + 2b(1 - \frac{1}{2b})C(j-1) + C(j) \\
&= 2b + (2b-1)C(j-1) + C(j) \\
(2b-1)C(j) &= 2b + (2b-1)C(j-1) \\
C(j) &= \frac{2b}{2b-1} + C(j-1) \qquad (2)
\end{aligned}
$$

Eq. (2) shows that the expected number of peers visited on each level is now $\frac{2b}{2b-1}$ which tends to go towards 1 as b increases. The expected total steps for a search operation now becomes $m = \frac{2b}{2b-1} \cdot log(n)$.

The modifications described in Sections 5.3 and 5.4 do not impact the structure of the Skip Graph, nor the flow of a single search operation. Instead, these modifications initiate r concurrent searches in s Skip Graphs to increase the probability that the search originator successfully retrieves requested information. The query success rate critically depends on the fraction of adversarial peers in the network which we denote by f_a. We first derive the probability $P_s(q)$ that a single search query q succeeds, which happens when there are no adversarial peers on the search path of q:

$$P_s(q) = (1 - f_a)^m \qquad (3)$$

Next, we derive the probability P_s that at least one of our $r \cdot s$ search queries succeed, which is:

$$
\begin{aligned}
P_s &= 1 - (1 - P_s(q))^{r \cdot s} \\
&= 1 - (1 - (1 - f_a)^m)^{r \cdot s}
\end{aligned} \qquad (4)
$$

Where m is given by:

$$m = \frac{2b}{2b-1} \cdot log(n) \qquad (5)$$

Combining Eq. (4) and (5) reveals that the probability of a search in DeScan succeeding depends on n, f_a, b, r and s. Increasing n and f_a reduce P_s whereas increasing b, r and s increase P_s. We remark that unresponsive peers do not affect the length of a search path (also see Fig. 8) but they do increase the overall number of messages being sent. Therefore, f_u does not influence P_s.

6. Experimental evaluation

We now describe our experimental setup and evaluation of DeScan. Our experiments answer the following questions:

1. What is the impact of unresponsive and adversarial peers on the success rate when querying triplets (Section 6.2)?
2. What is the effect of parameters n, b, s and r on the success rate when querying triplets, in the presence of unresponsive or adversarial peers (Section 6.3)?
3. What is the end-to-end latency of triplet queries in DeScan (Section 6.4)?
4. How does the network usage overhead of DeScan change when increasing the network size (Section 6.5)?
5. How does the storage overhead of DeScan change when increasing the network size and the volume of indexed transactions (Section 6.6)?
6. What is the storage requirement of DeScan when indexing and storing all transactions in the Ethereum blockchain (Section 6.7)?

Table 1
The parameters used by DeScan and during our experiments.

Notation	Parameter Description
n	Number of peers in the network.
b	Neighbours in the local routing table.
s	Number of Skip Graphs.
r	Triplet replication factor.
f_u	Fraction of unresponsive peers.
f_a	Fraction of adversarial peers.

6.1. Implementation and experimental setup

We implement all functionality of DeScan in the Python 3 programming language. DeScan is built using the IPv8 networking library that provides tools to build decentralized overlays.[4] We use the UDP network protocol for packet exchange and adopt an event-driven programming paradigm using the `asyncio` Python library. Triplets are exchanged between peers using a binary transfer protocol based on the Trivial File Transfer Protocol (TFTP). We open-sourced the full implementation of DeScan and associated artefacts such as tests and documentation.[5]

We use an overlay simulator already included in the IPv8 networking library to conduct experiments with thousands of peers. Each peer generates a random key pair and derives s membership vectors from their public key to join the Skip Graph. To evaluate the query duration of DeScan in a realistic environment with pairwise network latencies, we crawl ping times from WonderNetwork, providing estimations on the RTT between their servers in 277 cities [57]. We then assign each peer in our experiments to a city in a round-robin fashion and apply outgoing network latency accordingly. All experiments are conducted on a HPE DL385 Gen10 servers, equipped with 128 AMD EPYC 7452 CPUs, 512 GB of DDR4 memory, and running Debian 10. Even though our experiments are conducted in a simulated environment, it is straightforward to deploy DeScan since it is built with the IPv8 networking library, and its implementation requires minimal changes to the logic in order to run it in a non-simulator environment.

Each experiment run starts with all peers joining the Skip Graph and filling their routing tables. We then start the CPE on each peer (see Section 4.1) and index all locally available content. This will generate triplets and distribute them amongst the peers in the network. In each experiment, we perform exactly 1000 queries. A triplet query is always initiated by an honest, responsive peer. The triplets that these peers query are randomly chosen from the dataset. During all experiments, we monitor whether queries are successful, i.e., result in the requested triplets for the query originator. We repeat each experiment at least ten times and average all results. For reference, we have listed in Table 1 all the parameters considered during the evaluation of DeScan.

Dataset. Since we intend DeScan to be used in a Web3 context, we evaluate our mechanism using a subset of Ethereum transactions. We choose to evaluate with an Ethereum dataset since Ethereum is the most popular Web3 platform at the time of writing. Specifically, we have crawled the block with index 15 000 000 from the Ethereum mainnet

[4] See https://github.com/tribler/py-ipv8.
[5] See https://github.com/devos50/descan.

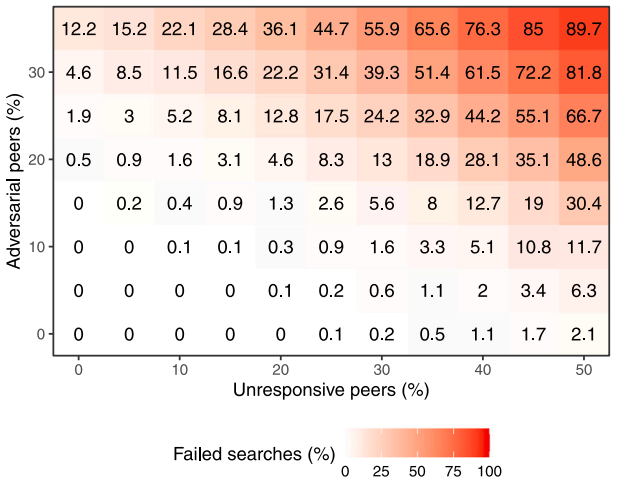


Fig. 9. The fraction of failed searches for different fractions of unresponsive and adversarial peers. We fix $n = 1600$ and $b = s = r = 5$.

and the 55 000 subsequent blocks, which together contain just over 10 million Ethereum transactions. These blocks were created between June 21, 2022 and July 1, 2022. We implemented two rules in DeScan to process this data. The first rule analyses a block and creates triplets using the Web3 library.[6] This rule also queues any transaction data contained in the block for processing. The second rule generates triplets from Ethereum transactions. Except for the experiments that analyse storage overhead, we experiment with a subset of our crawled dataset, namely the first 1000 blocks that include 187 987 transactions in total.

6.2. The impact of unresponsive and adversarial peers

Setup. We first analyse the impact of unresponsive and adversarial peers on the success rate of triplet queries. We model the behaviour of unresponsive and adversarial peers as described in Section 5 and as shown in Fig. 6. We fix $n = 1600$ and $b = s = r = 5$ to enable our four modifications presented in Section 5. As a result, a peer initiating a triplet query will issue $5 \cdot 5 = 25$ Skip Graph searches in parallel, five in each of the five Skip Graphs. We evaluate DeScan under different combinations of fractions of unresponsive peers (f_u) and adversarial peers (f_a). We experiment with up to 50% unresponsive peers and 35% adversarial peers.

Results. Fig. 9 shows the query success rate for different combinations of f_u and f_a, with percentages in each cells. We observe that queries remain highly successful with $f_u = 0$ and $f_a \leq 0.2$. In comparison, without our modifications enabled ($b = s = r = 1$) and with $f_u = 0$ and $f_a = 0.2$, 82.7% of all queries is unsuccessful. For $f_u = 0$ and $f_a = 0.35$, 12.2% of all triplet queries fail. In extreme circumstances with $f_u = 0.5$ and $f_a = 0.35$ (i.e., merely 15% of all peers are honest and responsive), we observe that 10.3% of all queries still are successful and result in triplets.

Fig. 9 also shows that DeScan better tolerates unresponsive peers than adversarial peers. Even when half of all nodes are unresponsive ($f_u = 0.5$), 97.9% of all queries are successful. This is because unresponsive peers can be detected by other peers and be removed from the routing tables. It is more difficult to verify whether the search result returned by an adversarial peer to a query initiator is correct [14]. We believe that detecting adversarial peers is an interesting extension of our work and can further increase the robustness of DeScan against adversarial peers.

[6] See https://web3js.readthedocs.io/en/v1.8.0/.

Conclusion. Our experiment reveals that DeScan and the proposed modifications in Section 5 exhibit some robustness against unresponsive and adversarial peers. With $n = 1600$ and $b = s = r = 5$, DeScan can deal with 20% adversarial peers or 50% unresponsive peers without significant disruption. This robustness, however, comes at an increased overhead of storing triplets, additional computational costs, and increased network overhead. We analyse this overhead in Sections 6.5 and 6.6.

6.3. The impact of DeScan parameters

We now explore the impact of different DeScan parameters listed in Table 1 on the query failure rate. We analyse the effect of these parameters on adversarial and unresponsive peers separately.

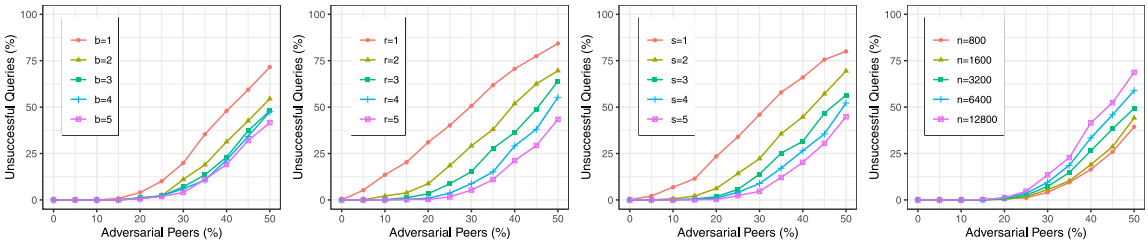
Adversarial Peers. We run experiments to analyse the effect of b, s, r, and n while increasing f_a up to $f_a = 0.5$ and while fixing $f_u = 0$. Fig. 10 shows the query failure rate as f_a increases, and each subplot shows the impact of a different system parameter on this failure rate. Unless otherwise stated, we fix $b = r = s = 5$ and $n = 1600$. Fig. 10(a) reveals that storing additional left and right neighbours on each level in the routing table (e.g., increasing b) decreases the query failure rate. This is because increasing b decreases the number of peers that a Skip Graph search has to traverse, as also shown in Fig. 7. Fig. 10(a) shows that increasing b from 1 to 2 has the most effect, and the gains by increasing b beyond that are marginal.

Fig. 10(b) highlights the effect of storing triplets on multiple (r) peers. With $r = 1$, the query failure rate increases roughly linear with the increase of f_a. This linear relation is expected since with $r = 1$, the probability that an adversarial peer stores some triplets is f_a. As such, the query failure rate is at least f_a. Additionally, there is a chance that the Skip Graph search, performed before retrieving triplets from a particular peer, hits an adversarial peer and fails. Increasing r reduces the query failure rate: with $r = 5$ and $f_a = 0.5$, there is already a probability of 96.9% that at least one honest peer stores some triplets. Fig. 10(b) shows that increasing r reduces the query failure rate: for $f_a = 0.25$, we see a decrease in unsuccessful queries for $r = 1$ and $r = 5$ from 40.2% to 1.6%, respectively.

Fig. 10(c) shows how constructing s Skip Graphs reduces the query failure rate in the presence of adversarial peers. For $f_a = 0.25$, increasing s from 1 to 5 reduces the query failure rate from 34.0% to 2.3%. We observe that increasing s shows comparable effects as increasing r (see Fig. 10(b)), probably because they both result in more parallel Skip Graph searches being performed. However, r controls the number of searches performed within a single Skip Graph, whereas s determines the number of unique Skip Graphs the network constructs.

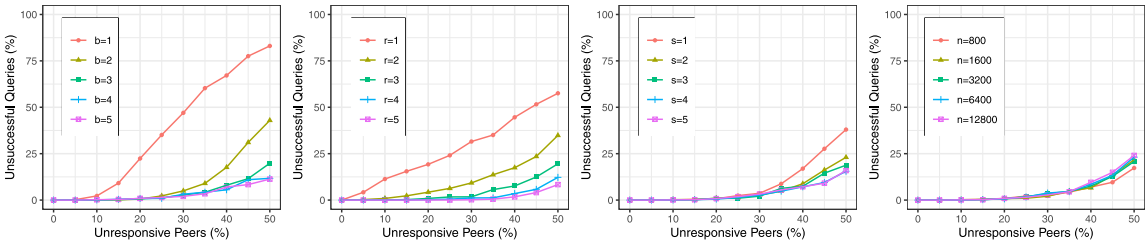
Finally, Fig. 10(d) shows how the query failure rate behaves as more nodes join the network. Increasing the network size harms the success of DeScan queries. This is because more peers in the network will result in longer search paths for individual Skip Graph searches. Thus the likelihood of hitting an adversarial peer during a search also increases. The effect of increasing n becomes more pronounced when f_a also increases. For $f_a = 0.5$, the query failure rate increases from 39.4% to 68.7% for $n = 800$ and $n = 12\,800$, respectively. This effect is less for $f = 0.25$ in absolute terms, where the same increase is from 1.2% to 4.65%.

Unresponsive Peers. We repeat the above experiments but vary the fraction of unresponsive nodes, f_u and fixing $f_a = 0$. Since Fig. 9 indicates that DeScan is highly robust against unresponsive nodes with parameters $b = s = r = 5$, we lower b, s and r to 3. This also helps to distinguish results in the plots better. The results of these experiments are visible in Fig. 11. Fig. 11(a) shows that increasing b has a high impact on the query failure rate: 83.1% of all queries fail with $f_u = 0.5$ and $b = 1$. This number is reduced to 11.3% for $b = 5$. Extending the routing tables with more nodes decreases the query failure rate since a `Search` message can always be forwarded to another neighbour on level i if an unresponsive peer is detected.



(a) The effect of b. (b) The effect of r. (c) The effect of s. (d) The effect of n.

Fig. 10. The effect of parameters b, r and n on the query failure rate, while increasing the fraction of adversarial peers, f_a. Unless otherwise stated, we fix $b = r = s = 5$ and $n = 1600$.



(a) The effect of b. (b) The effect of r. (c) The effect of s. (d) The effect of n.

Fig. 11. The effect of parameters b, r and n on the query failure rate, while increasing the number of unresponsive peers, f_u. Unless otherwise stated, we fix $b = r = s = 3$ and $n = 1600$.

Fig. 11(b) shows the effect of increasing r on the query failure rate. When increasing r from 1 to 5 for $f_u = 0.35$, the query failure rate reduces significantly, from 35.1% to 0.5%. Comparing Figs. 11(b) and 10(b), we find the effect of increasing r in the presence of unresponsive peers is more than when having adversarial peers. This is because adversarial peers have a more profound impact on the failure of Skip Graph searches than unresponsive peers. Specifically, an unresponsive peer on a search path does not necessarily fail the search — the unresponsive peer can simply be removed from the routing table, and the `Search` message can be forwarded to another peer. In contrast, any adversarial peer on a search path can fail a search and conducting searches in multiple Skip Graphs is an effective approach to deal with such nodes. Fig. 11(c) highlights that increasing s also increases the query success rate, although this effect is less pronounced than in the presence of adversarial peers as in Fig. 10(c). For $s = 1$ and $f_u \geq 0.3$, we see that the query failure rate quickly increases.

Fig. 11(d) shows that the network size has little impact on the query failure rate. Even though a larger network size increases the number of peers that a Skip Graph search traverses, with $b = r = s = 3$, there is sufficient redundancy in the routing tables to handle unresponsive peers and to ensure that most queries are still successful.

Conclusion. We have analysed the effect of b, s, r and n on the query failure rate for different values of f_a and f_u. In general, increasing b, s, and r reduces the query failure rate, both in the presence of adversarial and unresponsive peers. To resist adversarial peers, increasing r and s is the most effective since that increases the number of Skip Graph searches performed for each query. To resist unresponsive peers, increasing b is the most effective since it allows a Skip Graph search to be routed to other peers in the routing table when an unresponsive peer is detected.

6.4. Latency of triplet queries

The ability for users to quickly retrieve relevant triplets is a key property of DeScan. Our next experiments measure the duration of triplet queries, for different network sizes and in the presence of adversarial and unresponsive nodes.

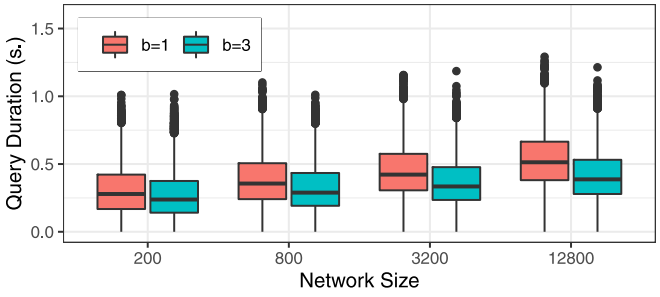


Fig. 12. The distribution of query durations as the network size increases. We fix $s = r = 5$ and experiment with $b = 1$ and $b = 3$.

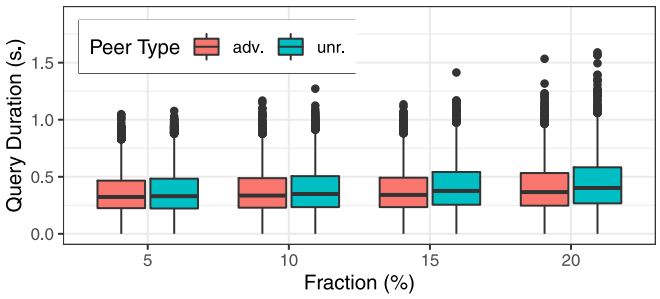


Fig. 13. The distribution of query durations as the fractions of adversarial and unresponsive peers (f_a and f_u) increases. We fix $b = r = s = 5$ and $n = 1600$.

Setup. We first analyse the duration of individual queries as the number of peers in the network (n) increases. This allows us to analyse

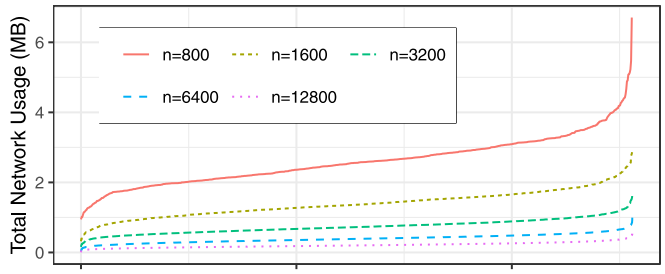


Fig. 14. The total network usage per peer, for increasing values of n. The horizontal axis is a dimensionless unit denoting the peer indices, ascendantly ordered by total network usage. We fix $b = s = r = 5$.

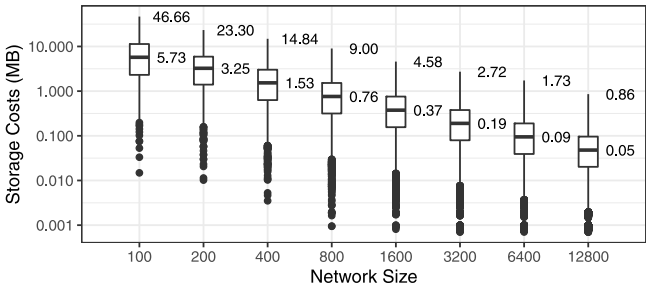


Fig. 15. The storage overhead by individual peers as n increases. For each value of n we annotate the median, and maximum storage overhead. We fix $r = 5$.

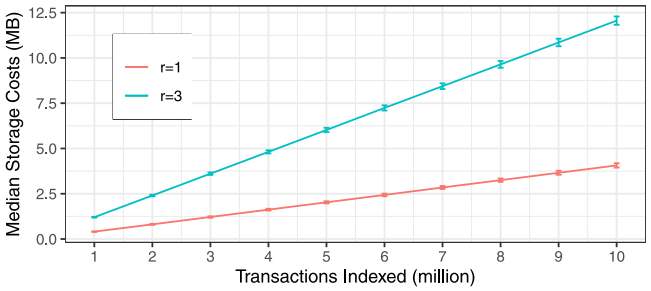


Fig. 16. The evolution median storage overhead when indexing and storage Ethereum transactions with DeScan and $n = 1600$, for $r = 1$ and $r = 3$. The error bars indicate the standard deviation across runs.

the performance of DeScan as the network grows. We keep track of the duration of each query, which is the elapsed time between initiating the query until retrieving all edges from at least one other peer. We increase n, starting from 100 and up to 12 800, multiplying n by four every step. Then we analyse the effect of unresponsive and adversarial peers on the query durations.

Results. Fig. 12 shows the distribution of query durations in seconds for increasing network sizes, and for $b = 1$ and $b = 3$. We find that query durations in DeScan scale very well with the network size: there only is a marginal increase in the average query duration when the network size grows. For $b = 1$, the average query duration increases from 0.31s for $n = 200$ to 0.53s for $n = 12\,800$, an increase of just 225 ms. For $b = 1$ and $n = 12\,800$, 99% of all queries are answered within 1.1s. Fig. 12 also shows that increasing b reduces query durations since Skip Graph searches can be completed more efficiently when maintaining more neighbourhood information in local routing tables. For $b = 3$ and $n = 12\,800$, queries are answered in 0.42s on average, a reduction of 21.3% compared to the experiment with $b = 1$.

Fig. 13 shows the impact of increasing the fractions of adversarial and unresponsive peers (f_a and f_u) on the duration of successful queries. For $u_a = 0.05$ and $f_u = 0.05$ the average query duration is 360 ms and 364 ms, respectively. Average query durations increase to 406 ms and 439 ms for $u_a = 0.2$ and $u_f = 0.2$, respectively. As such, adversarial and unresponsive peers have a low impact on the query durations. Fig. 13 does reveal that unresponsive peers increase query durations slightly more than adversarial peers.

Conclusion. Our experiment demonstrate that queries in DeScan are usually completed well within a second, even with adversarial or unresponsive peers, and when the network size increases.

6.5. Network usage

Our following experiments quantify the network usage induced by DeScan.

Setup. During this experiment, we measure the amount of incoming and outgoing network traffic for every peer when conducting 10 000 searches. We repeat the experiment for increasing values of n to determine how network traffic is distributed over peers in the network.

Results. Fig. 14 shows the total network usage in MB per peer when increasing the network size. For each value of n, we have sorted all peers based on their network usage in ascending order. For $n = 800$, we see significant differences in network utilization amongst peers. The peer with the lowest utilization incurs 0.94 MB of network traffic, whereas the peer with the highest utilization incurs 6.71 MB of network traffic. As n increases, we also find that the network usage per peer decreases and the variation between peers also lowers. In other words, the network load is more evenly balanced amongst peers. For $n = 12\,800$, the average network usage per peer is just 0.21 MB and the peer with the highest network utilization processed 0.59 MB of traffic. This indicates that DeScan effectively disperses the communication burdens amongst peers as the network grows.

Further analysis reveals that 92.7% of all network traffic is related to Skip Graph searches. The remaining 7.3% of traffic is attributed to transferring triplets to the query initiator. With $n = 12\,800$, each query results in roughly 252.8 KB of network traffic. This number can be reduced by lowering s and r. In comparison, with $s = r = 1$, each query only results in 10.1 KB network traffic.

Another insight is that many of the parallel searches performed are effectively "wasted" since they might result in triplets after a search originator already received triplets from a faster parallel search. To address this, we could perform Skip Graph searches in linear instead of in parallel and stop when one of the searches and subsequent triplet retrievals are successful. Linear execution of searches increases query latency since one has to wait for a Skip Graph search to complete or for the timeout trigger before the next one is initiated.

Conclusion. As the DeScan network grows, the communication overhead becomes more evenly distributed over participating peers.

6.6. Storage cost

We now explore the storage cost of peers when increasing the network size or the number of indexed transactions.

Setup. After each experiment ends, we compute the storage requirements of DeScan for each peer by summing the size of all stored triplets and Merkle Trees in serialized form. When increasing the network size n, we fix $r = 5$. We note that changing b and s does not affect the storage requirements of individual peers.

Results. Fig. 15 shows a plot with the number of peers in the network on the horizontal axis and the storage overhead of each peer in Megabyte on a logarithmic vertical axis. This figure indicates that the storage overhead per peer reduces as more peers join the network. With $n = 12\,800$, the median storage cost per peer is only 48.1 KB. In comparison, if the network only had a single peer ($n = 1$), this single peer would incur a storage overhead of 174.8 MB. Fig. 15 shows

Table 2
The estimated number of peers necessary to ensure various average storage costs when storing all indexed transactions on the Ethereum blockchain for different values of r.

Average storage cost	Peers necessary		
	$r = 1$	$r = 3$	$r = 5$
1 GB	1824	5472	9120
100 MB	18 672	56 016	93 360
10 MB	186 714	560 142	933 570
1 MB	1 867 137	5 601 411	9 335 685

that DeScan effectively distributes the storage burdens over peers as the network grows.

We also observe from Fig. 15 that triplets are unevenly balanced amongst peers. We observe considerable differences in storage cost between peers for a fixed n. For example, with $n = 100$, one peer is required to store 46.6 MB of data, which is 8.2 times the median storage cost (5.7 MB). At the same time, several peers are storing less than 100 KB.

This imbalance in storage cost has two causes. First, the number of triplets generated by rules varies for different content items, which can result in significant storage imbalances. Second, node keys are unevenly distributed within the key space. This is because each peer in our experiment generates a uniformly random cryptographic keypair. It is likely that a peer is "responsible" for a larger or smaller key range compared to other peers. This issue can, for example, be alleviated by having a coordinator assign keys to joining peers, or by using a load-aware key assignment algorithm that considers load balancing. Offloading triplets to adjacent nodes in the Skip Graph, as described in Section 4.3, can also alleviate this issue. These results show that as the network grows, storage burdens decrease and it becomes possible for low-resource devices (e.g., mobile wallets) to join the system and persist indexed transactions.

Fig. 16 shows the number of indexed transactions on the horizontal axis and the median storage cost across peers in the network on the vertical axis for $r = 1$ and $r = 3$. We index up to 10 million Ethereum transactions and fix $n = 1600$. The error bar indicates the standard deviation of the median storage costs across the ten experiment runs, indicating the sensitivity of this metric to the key space. Fig. 16 shows that the median storage cost increases linearly with the number of stored transactions. After indexing all ten million transactions, the average median storage cost is 4.1 MB and 12.1 MB for $r = 1$ and $r = 3$, respectively. We also observe that the deviation of median storage costs across runs (e.g., the error bars length) increases when more transactions are indexed.

Conclusion. DeScan evenly distributes the storage costs over peers as the network grows.

6.7. Indexing and storing the full Ethereum blockchain

Finally, we estimate the storage costs with DeScan when storing the entire Ethereum blockchain. Ethereum is currently the most popular blockchain in terms of transaction volume, and at the time of writing contains close to 2 billion transactions. The experimental data in Section 6.6 reveals that an indexed transaction, on average, results in a storage overhead of 979 bytes when storing it at a single peer in the network without replication. We estimate that, to store 2 billion transactions with a replication factor of 5, we require a network storage capacity of 1.8 TB.

Table 2 lists the estimated number of peers necessary to ensure particular average storage costs when storing *all* indexed transactions on the Ethereum blockchain for different values of r. To ensure an average storage cost of 1 MB while replicating each triplet across five peers, we require the involvement of approximately 9.3 million peers. While this number might seem high, we also suspect that there is an abundance of lightweight Internet devices that can easily store 1 MB of content or more. If each peer can store 100 MB, we require 93 360 peers to store the entire Ethereum blockchain. In practice, the storage capabilities of participating peers are likely to differ significantly, resulting in the situation where some peers store more triplets than others. The extension introduced in Section 4.3 can help to rebalance triplet storage across the network.

We also note that the rules used by the CPE are relatively simple and only process high-level metadata of each Ethereum block and transaction. More advanced rules that index content items with higher granularity generate additional triplets, resulting in additional storage overhead per indexed content item. The above analysis also ignores the situation where Web3 data from multiple applications are indexed and stored with DeScan. Nonetheless, our experiments hint that DeScan is a scalable solution in which the storage costs decrease as the network grows.

7. Conclusions

We have presented DeScan, a fully functional decentralized and censorship-resistant Web3 system for indexing and search. DeScan uses rules to transform Web3 transactions into triplets, which together form a transaction graph. The storage of this transaction graph is distributed over peers in the DeScan network. We leverage a Skip Graph data structure to coordinate this storage process and to enable users to search for triplets. Through a series of modifications, we improve the robustness of the DeScan against both adversarial and unresponsive peers. In particular, we extend local routing tables, replicate triplets over multiple peers and conduct parallel searches in multiple Skip Graphs.

We provide a formal analysis of the algorithmic complexity and censorship resistance of DeScan, which establishes a relation between the probability of a search succeeding and the DeScan system parameters. We provide an experimental evaluation of this analysis. It shows that the increase of two specific parameters, namely the triplet replication factor (r) and the number of Skip Graphs (s), serve as an effective countermeasure against adversarial peers. Our experiments with a real-world transaction trace from the Ethereum blockchain showed that DeScan, with our modifications, can deal with up to 25% adversarial peers and 35% unresponsive peers without significant system degradation. We show that queries for triplets are completed well within a second, even when there are over 10 000 peers. As the network grows, the communication overhead becomes more evenly distributed among peers. Finally, we establish that the network and storage overhead induced by individual peers decreases as the network grows.

CRediT authorship contribution statement

Martijn de Vos: Conceptualization, Methodology, Software, Validation, Formal analysis, Writing – original draft, Funding acquisition. **Georgy Ishmaev:** Conceptualization, Writing – review & editing, Funding acquisition. **Johan Pouwelse:** Resources, Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Martjn de Vos reports financial support was provided by Ethereum Foundation. Georgy Ishmaev reports financial support was provided by Ethereum Foundation.

Data availability

The link to the research data and code is available as URL in the article.

Acknowledgements

This work is funded by the Ethereum Foundation under grant FY22-0833.

References

[1] L.V. Kiong, Web3 Made Easy: A Comprehensive Guide to Web3: Everything you need to know about Web3, Blockchain, DeFi, Metaverse, NFT and GameFi, Liew Voon Kiong, 2022.
[2] A. Pentland, Building a new economy: data, AI, and Web3, Commun. ACM 65 (12) (2022) 27–29, http://dx.doi.org/10.1145/3547659, URL https://dl.acm.org/doi/10.1145/3547659.
[3] Etherscan, Etherscan: The ethereum blockchain explorer, 2022, URL https://etherscan.io.
[4] Infura, Infura, 2022, URL https://infura.io.
[5] A. Wahrstätter, J. Ernstberger, A. Yaish, L. Zhou, K. Qin, T. Tsuchiya, S. Steinhorst, D. Svetinovic, N. Christin, M. Barczentewicz, et al., Blockchain censorship, 2023, arXiv preprint arXiv:2305.18545.
[6] R. Recabarren, B. Carbunar, Tithonus: A bitcoin based censorship resilient system, Proc. Priv. Enhanc. Technol. 1 (2019) 68–86.
[7] Z. Wang, X. Xiong, W.J. Knottenbelt, Blockchain transaction censorship:(in) secure and (in) efficient? Cryptol. ePrint Arch. (2023).
[8] S. Brown, MEV driven centralization in ethereum: Part 1, 2022, URL https://simbro.medium.com/mev-driven-centralization-in-ethereum-ec829a214f18.
[9] A.E. Gencer, S. Basu, I. Eyal, R.v. Renesse, E.G. Sirer, Decentralization in bitcoin and ethereum networks, in: International Conference on Financial Cryptography and Data Security, Springer, 2018, pp. 439–457.
[10] M. Kleppmann, A. Wiggins, P. van Hardenberg, M. McGranaghan, Local-first software: you own your data, in spite of the cloud, in: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, ACM, Athens Greece, 2019, pp. 154–178, http://dx.doi.org/10.1145/3359591.3359737, URL https://dl.acm.org/doi/10.1145/3359591.3359737.
[11] C. Böhm, D. Hefenbrock, F. Naumann, Scalable peer-to-peer-based RDF management, in: Proceedings of the 8th International Conference on Semantic Systems, 2012, pp. 165–168.
[12] J. Aspnes, G. Shah, Skip graphs, ACM Trans. Algorithms 3 (4) (2007) 37–es.
[13] V.H. Lakhani, L. Jehl, R. Hendriksen, V. Estrada-Galinanes, Fair incentivization of bandwidth sharing in decentralized storage networks, in: 2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops, ICDCSW, IEEE, 2022, pp. 39–44.
[14] S.T. Boshrooyeh, O. Ozkasap, Guard: Secure routing in skip graph, in: 2017 IFIP Networking Conference (IFIP Networking) and Workshops, IEEE, 2017, pp. 1–2.
[15] P. Chatzigiannis, F. Baldimtsi, K. Chalkias, Sok: Blockchain light clients, in: Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers, Springer, 2022, pp. 615–641.
[16] Blockchain.com, Blockchain.com, 2022, URL https://blockchain.com.
[17] M. Li, J. Zhu, T. Zhang, C. Tan, Y. Xia, S. Angel, H. Chen, Bringing decentralized search to decentralized services, in: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), 2021, pp. 331–347.
[18] The Portal Network, The portal network, 2022, URL https://github.com/ethereum/portal-network-specs.
[19] G. Danezis, C. Lesniewski-Laas, M.F. Kaashoek, R. Anderson, Sybil-resistant DHT routing, in: European Symposium on Research in Computer Security, Springer, 2005, pp. 305–318.
[20] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, D.S. Wallach, Secure routing for structured peer-to-peer overlay networks, Oper. Syst. Rev. 36 (SI) (2002) 299–314.
[21] TrueBlocks, TrueBlocks: Lightweight indexing for any EVM-based blockchain, 2022, URL https://trueblocks.io.
[22] J. Benet, Ipfs-content addressed, versioned, p2p file system, 2014, arXiv preprint arXiv:1407.3561.
[23] The Graph Foundation, The Graph Network, 2022, URL https://thegraph.com/blog/transitioning-to-decentralized-graph-network/.
[24] T. Crain, C. Natoli, V. Gramoli, Red belly: A secure, fair and scalable open blockchain, in: 2021 IEEE Symposium on Security and Privacy, SP, IEEE, 2021, pp. 466–483.
[25] M. Ripeanu, Peer-to-peer architecture case study: Gnutella network, in: Proceedings First International Conference on Peer-to-Peer Computing, IEEE, 2001, pp. 99–100.
[26] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, S. Shenker, Making gnutella-like p2p systems scalable, in: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2003, pp. 407–418.
[27] M. Castro, M. Costa, A. Rowstron, Should we build gnutella on a structured overlay? ACM SIGCOMM Comput. Commun. Rev. 34 (1) (2004) 131–136.
[28] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, ACM SIGCOMM Comput. Commun. Rev. 31 (4) (2001) 149–160.
[29] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, in: IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, 2001, pp. 329–350.
[30] A. Fiat, J. Saia, Censorship resistant peer-to-peer content addressable networks, in: SODA, Vol. 2, Citeseer, 2002, pp. 94–103.
[31] J. Augustine, A.R. Molla, E. Morsy, G. Pandurangan, P. Robinson, E. Upfal, Storage and search in dynamic peer-to-peer networks, in: Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2013, pp. 53–62.
[32] R. Villanueva, M.d. Pilar Villamil, M. Arnedo, Secure routing strategies in dht-based systems, in: International Conference on Data Management in Grid and P2P Systems, Springer, 2010, pp. 62–74.
[33] M. Sanchez-Artigas, P.G. Lopez, A.F.G. Skarmeta, Bypass: providing secure DHT routing through bypassing malicious peers, in: 2008 IEEE Symposium on Computers and Communications, IEEE, 2008, pp. 934–941.
[34] B. Heep, R/kademlia: Recursive and topology-aware overlay routing, in: 2010 Australasian Telecommunication Networks and Applications Conference, IEEE, 2010, pp. 102–107.
[35] K. Hildrum, J. Kubiatowicz, Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks, in: Distributed Computing: 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003. Proceedings 17, Springer, 2003, pp. 321–336.
[36] I. Baumgart, S. Mies, S/kademlia: A practicable approach towards secure key-based routing, in: 2007 International Conference on Parallel and Distributed Systems, IEEE, 2007, pp. 1–8.
[37] S. Serbu, P. Felber, P. Kropf, HyPeer: Structured overlay with flexible-choice routing, Comput. Netw. 55 (1) (2011) 300–313.
[38] H. Nagao, K. Shudo, Flexible routing tables: Designing routing algorithms for overlays based on a total order on a routing table set, in: 2011 IEEE International Conference on Peer-to-Peer Computing, IEEE, 2011, pp. 72–81.
[39] M. Hojo, R. Banno, K. Shudo, Frt-skip graph: A skip graph-style structured overlay based on flexible routing tables, in: 2016 IEEE Symposium on Computers and Communication, ISCC, IEEE, 2016, pp. 657–662.
[40] D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, A. Wahler, Introduction: what is a knowledge graph? in: Knowl. Graphs, Springer, 2020, pp. 1–10.
[41] P.A. Bonatti, S. Decker, A. Polleres, V. Presutti, Knowl. graphs: New directions for knowledge representation on the semantic web (dagstuhl seminar 18371), in: Dagstuhl Reports, Vol. 8, No. 9, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
[42] X. Zou, A survey on application of knowledge graph, J. Phys. Conf. Ser. 1487 (1) (2020) 012016.
[43] H. Kanezashi, T. Suzumura, X. Liu, T. Hirofuchi, Ethereum fraud detection with heterogeneous graph neural networks, 2022, arXiv preprint arXiv:2203.12363.
[44] V. Patel, L. Pan, S. Rajasegarar, Graph deep learning based anomaly detection in ethereum blockchain network, in: International Conference on Network and System Security, Springer, 2020, pp. 132–148.
[45] R. Mars, A. Abid, S. Cheikhrouhou, S. Kallel, A machine learning approach for gas price prediction in ethereum blockchain, in: 2021 IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC, IEEE, 2021, pp. 156–165.
[46] B. Nasrulin, G. Ishmaev, J. Pouwelse, MeritRank: Sybil tolerant reputation for merit-based tokenomics, in: 2022 4th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), IEEE, Paris, France, 2022, pp. 95–102, http://dx.doi.org/10.1109/BRAINS55737.2022.9908685, URL https://ieeexplore.ieee.org/document/9908685/.
[47] Z. Chen, Y. Wang, B. Zhao, J. Cheng, X. Zhao, Z. Duan, Knowledge graph completion: A review, IEEE Access 8 (2020) 192435–192456.
[48] S. Wang, C. Huang, J. Li, Y. Yuan, F.-Y. Wang, Decentralized construction of knowledge graphs for deep recommender systems based on blockchain-powered smart contracts, IEEE Access 7 (2019) 136951–136961.
[49] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, Commun. ACM 33 (6) (1990) 668–676.
[50] S. Eskandari, S. Moosavi, J. Clark, Sok: Transparent dishonesty: front-running attacks on blockchain, in: International Conference on Financial Cryptography and Data Security, Springer, 2019, pp. 170–189.
[51] J.R. Douceur, The sybil attack, in: International Workshop on Peer-to-Peer Systems, Springer, 2002, pp. 251–260.
[52] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, Y. Psaras, GossipSub: Attack-resilient message propagation in the Filecoin and ETH2. 0 networks, 2020, arXiv preprint arXiv:2007.02754.
[53] R. Jacob, A. Richa, C. Scheideler, S. Schmid, H. Täubig, SKIP+ A self-stabilizing skip graph, J. ACM 61 (6) (2014) 1–26.
[54] R. Guerraoui, S.B. Handurukande, K. Huguenin, A.-M. Kermarrec, F. Le Fessant, E. Riviere, Gosskip, an efficient, fault-tolerant and self organizing overlay using gossip-based construction and skip-lists principles, in: Sixth IEEE International Conference on Peer-to-Peer Computing, P2P'06, IEEE, 2006, pp. 12–22.

[55] A. Disterhöft, A. Funke, K. Graffi, Packetskip: Skip graph for multidimensional search in structured peer-to-peer systems, in: 2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems, SASO, IEEE, 2017, pp. 21–30.
[56] A. González-Beltrán, P. Milligan, P. Sage, Range queries over skip tree graphs, Comput. Commun. 31 (2) (2008) 358–374.
[57] WonderNetwork, Global ping statistics, 2022, https://wondernetwork.com/pings. (Accessed 12 August 2022).

Martijn de Vos (m.a.devos-1@tudelft.nl) is a postdoctoral researcher at the Distributed Systems section of TU Delft. His research focusses on developing lightweight decentralized solutions and decentralized machine learning.

Georgy Ishmaev (g.ishmaev@tudelft.nl) is a postdoctoral researcher at the Distributed Systems section of Software Development department of TU Delft. His research is focused on the ethics of decentralized systems, with specific focus on blockchain technology, identity management, and data ethics.

Johan Pouwelse (j.a.pouwelse@tudelft.nl) is an associate professor at the Distributed Systems section of Software Development department of TU Delft, specialized in large-scale cooperative systems. He is the founder of Tribler, a living laboratory and proving ground for next generation self-organizing systems research and ledger technology.