Paavolainen, Santeri; Carr, Christopher; Ghadafi, Essam

Adventures of a Light Blockchain Protocol in a Forest of Transactions

Published in:
IEEE Access

DOI:
10.1109/ACCESS.2021.3101717

Published: 30/07/2021

Document Version
Publisher's PDF, also known as Version of record

Published under the following license:
CC BY

Please cite the original version:

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.
Adventures of a Light Blockchain Protocol in a Forest of Transactions: A Subset of a Story

SANTERI PAAVOLAINEN1, (Member, IEEE), CHRISTOPHER CARR2, AND ESSAM GHADAFI3

1School of Electrical Engineering, Aalto University, 02150 Espoo, Finland
2Department of Accounting, Economics and Finance, University of the West of England, Bristol BS16 1QY, U.K.
3Department of Computer Science and Creative Technologies, University of the West of England, Bristol BS16 1QY, U.K.

Corresponding author: Santeri Paavolainen (santeri.paavolainen@aalto.fi)

This work was supported by the European Union’s Horizon 2020 Research and Innovation Programme under Agreement 779984.

ABSTRACT The Ethereum blockchain is one of the most popular permissionless blockchains. A consequence of its popularity has been the growth of processing and data storage requirements for any node participating in the Ethereum blockchain network. For constrained devices such requirements are often infeasible to meet. To cater for such nodes, a so-called light protocol has been proposed for Ethereum where the responsibility of maintaining a correct state representation is delegated to light protocol servers. Previous research has identified dependence on external state management as a potential security vulnerability that exclusively impacts light nodes. Although a simple mitigation strategy is available, it comes at the expense of increased latency. In this work, we propose a new Ethereum node type, which we call a subset node, as an extension of the light protocol. Our proposal allows subset nodes to gain a lower latency than a pure light node with comparable or even higher security assurances by tracking and evaluating only a subset of all of the transactions issued on the blockchain. We provide a formal proof on the correctness of the blockchain state used by the subset node under the proposed model. To evaluate the practical feasibility of the subset node model, we analyze one year of historical transaction data from Ethereum, and demonstrate that a subset node tracking the state of a single account can achieve a significant reduction in storage and computational requirements when compared to a full node.

INDEX TERMS Blockchain, Ethereum, transactions, transaction dependencies, light protocol, constrained devices.

I. INTRODUCTION

Since the introduction of Bitcoin [1], so-called second generation blockchains, such as Ethereum [2], have expanded the expressive power of blockchains through distributed programs embedded into the blockchain state, called smart contracts. Being theoretically Turing-complete, these programs allow blockchains to be extensible and adaptable for purposes other than pure account-to-account token transfers. The smart contract system makes Ethereum a popular choice for many commercial and research projects. It also appears that secure computations on a blockchain are desirable, as they have become almost universally included in more recent blockchains [3].

An unfortunate side effect of smart contracts is that the state of the blockchain is no longer explicitly represented by the set of its transactions (such as it is for the unsigned transaction output UTXO set in Bitcoin). Instead, the state is computed from transactions, and as a consequence, requires blockchain participants to continuously compute updates to a local copy of the computed state. For example, storing the full state representation of Ethereum requires several gigabytes of storage space [4]. In addition to the storage requirements, communication [5] and computing requirements may be prohibitive to resource-constrained devices. To circumvent these resource requirements, many blockchains include so-called light protocols to provide sufficient security to constrained devices without directly utilizing trusted intermediaries. However, constrained devices operating a light protocol have been identified previously as vulnerable to state injection attacks [6].

There are several proposals to increase light client security that would mitigate the identified vulnerability, but these either require: Partial connectivity with the honest network [7]; a decentralized reputation system [8]; replacement of the entire blockchain [9]; or introduction of a remote...
trusted party [10]. In short, they do not comprehensively address the issue within the context of the current Ethereum blockchain. Thus, it seems that in the context of existing Ethereum blockchain we only have a choice between being a full node—impossible for constrained devices—or being a light node and accepting the associated risks. A secure solution in this space would appear to offer large benefits, especially to IoT devices which are being increasingly deployed, but are often limited in the amount of computing and storage resources.

To address this gap, we propose a new type of Ethereum blockchain light client, which we call a subset node. A subset node differs from a regular light node in maintaining its local copy of the blockchain state—but only for a subset of all accounts recorded on the blockchain. Since a subset node does not rely on retrieving extracts of the blockchain state from a server, it is immune to state injection attacks by malicious light protocol servers. While a subset node is more secure against state injection attacks than a regular light node, the most important benefit is the expected considerable reduction in bandwidth, computing, and storage resources that are required.

However, it is not immediately obvious whether such a scheme is feasible either theoretically or practically, and what kind of benefits in resource savings it could offer. For example, it is not clear that “intuitively unrelated” transactions are, in reality, permanently independent in Ethereum’s transaction model. To provide a firm ground to subset nodes, we provide a formal proof of evaluation independence of independent transactions. To evaluate the feasibility of subset nodes, we analyze a year’s collection of Ethereum transactions, and determine the distribution of account and transaction dependencies in that data set. We base our analysis on a worst-case behavior, and conclude that the number of dependencies for a single account that occur on the Ethereum blockchain represent a very small fraction of the total number of transactions and number of active accounts within blocks that were analyzed.

In summary, the contribution of this article is to propose and describe subset nodes, to formally prove the correctness of the subset of the full blockchain state that the subset node uses, and to evaluate the practical feasibility of the subset node scheme against a year's worth of real-world transactions from the Ethereum network.

The rest of this work is structured as follow: we discuss and review related research and approaches to light protocol security in Section II, followed by an overview of how Ethereum processes transactions, manages state, and implements the light protocol in Section III. We describe the subset node model in detail in Section IV, including its operational characteristics and security properties — providing an overview of the formal proof. For the evaluation, we describe our analysis methods in Section V, and provide results and analysis in Section VI. We provide concluding remarks in Section VII, and give the full formal proof of transaction disjointness (a property referred to in Section IV-E) in Appendix.

II. RELATED WORK
Transactions in various blockchains, including Ethereum, have been examined from various angles. An early example is by Ron and Shamir [13] who looked at Bitcoin transaction dependencies as a graph, deriving several interesting statistical results. Transaction graphs have been used by many other researchers as well. For example, Guo et al. [14] characterized the Ethereum transaction network, finding that several transaction features exhibit a power-law distribution, and that most accounts are only weakly connected in the overall transaction graph. Similar findings have been made by various authors when analyzing the transaction or account graphs of Bitcoin [15]–[17] and Ethereum blockchains [17]–[21], although it has to be noted that it is still unclear whether these would be better matched by some other heavy-tailed distribution [22]. Furthermore, some specialized transaction sub-networks, such as token networks, do not always appear to follow a power-law distribution at all [23] and consequently, one needs to be cautious and not assume a power-law behavior of blockchain transaction-related graphs.

Most of the previously mentioned transaction graph analysis has focused on graphs constructed from user-generated transactions—“initiating transactions”—however, in some blockchains such as Ethereum, a transaction may result in subsequent internal transactions to be issued. These have been analyzed in detail, for example by di Angelo and Salzer [24], allowing a more detailed behavioral classification across “smart contract” accounts. Furthermore, the analysis of internal transactions has been used by Lee et al. [22] to build a more accurate graph of various different types of networks (e.g., transactions, contracts, and tokens) as defined by inter-contract transactions than would be possible based on initiating transaction analysis alone.

A more formal approach to blockchain transactions has been taken by Cachin et al. [25], who describe transaction graphs of Bitcoin, Ethereum, and Hyperledger Fabric. They present a general transaction graph model that can be used to describe transactions in different blockchains. Their approach views the blockchain state as an implicit concept, realized through the transaction graph. This coincides with the view of “state” in Bitcoin, but does not really address the explicit state representation (the state trie) of Ethereum.

Several researchers have noted the (unsurprising) property that most blockchain transactions are unrelated to each other, and this lack of relatedness can be exploited to increase transaction processing performance and scalability of the blockchain. The SAREK ordering framework, proposed by Li et al. [26], uses a predictor mechanism to distribute individual transactions to those partitions which host the relevant state, aiming to increase overall system scalability.

1 Conversely, a subset node is no less vulnerable than any other blockchain node to double-spending attacks, vulnerabilities in smart contracts, the incorrect application of other security measures [11], or any systemic failures such as misallocation of trust to various governing structures [12].
Others [27]–[30] have also noted that extending a blockchain from a linear structure into a directed acyclic graph (DAG) may improve overall system performance. In these works, however, the main concerns are not the dependencies among transactions in a block, but rather the dependency graph between different blocks. Perhaps the most explicit approach to block-level DAG has been taken by Sompolinsky et al. in their GHOSTDAG blockchain proposal [31].

While GHOSTDAG demonstrates the possibility of block-level parallelism, i.e., the lack of conflicts between blocks, researchers such as Dickerson et al. [32] and Anjana et al. [33] have examined whether transactions can be evaluated in parallel within a block to improve block processing performance. In their approach, transactions are speculatively evaluated in parallel and any conflicts are resolved into a serializable concurrent schedule. While their primary focus is on increasing block validation performance, it also coincidentally demonstrates that there exists exploitable parallelism as many transactions are independent of each other.

There are also proposals to improve the security of Ethereum light clients. One approach is to leverage Trusted Execution Enclaves (TEE) such as Intel’s SGX technology, where the TEE executed code is attested as authentic, and is used to provide secure updates on the ledger state to a light client [34]–[36]. There are other mechanisms as well that can increase the security of light clients albeit at the cost of relying on a trusted party [10]. It has been pointed out that the default Ethereum light protocol model provides no incentives for the servers to be honest, which has been addressed for example by Lu et al. who devise a model which incentivizes light servers to provide honest answers [37].

Some proposals move away from focusing on a single node, and look at how a community of loosely co-operating light nodes can provide higher security assurance and levy greater penalties against misbehaving nodes. For example, fraud proofs [7] allow a node to provide a succinct proof of incorrect behavior of a network node. Alternatively, a group of collaborative light nodes can distribute the resource costs associated with state verification [38]. Finally, instead of working with the confines of existing blockchain protocols, entirely new models have been proposed that have the potential to significantly reduce resource requirements [9], [37], [39].

It is possible to address the resource requirements of a blockchain directly instead of using light protocols. One common approach is to reduce the computation or storage requirements imposed on all nodes (not just light nodes) by the ledger itself [40], [41], with further improvements gained by reducing the communication bandwidth requirements [5], [42], [43]. Also, redacting (or rewriting) the history of a blockchain could be utilized to reduce the size of past history [44]–[46]. An emerging approach is to bridge high-resource ledgers with low-resource ledgers via inter-ledger approaches [47]–[49], allowing secure inter-operation from a ledger with high resource requirements to the constrained device via a more “IoT-friendly” ledger.

### III. BACKGROUND

This section contains the background material for this work, including a description of how Ethereum manages its state and the necessary information on Ethereum’s light protocol.

#### A. ETHEREUM TRANSACTIONS AND STATE

The state of the Ethereum blockchain includes information about all accounts, including account balances, the code of smart contracts, and the persistent variables of all smart contracts. The state of the blockchain changes with each block $n$, for which the state is represented as $S_n$. This state is stored as a Merkle-Patricia trie with individual accounts, identified by an address, as the leaves of the trie.\(^2\)

For smart contracts, the account includes a reference to the storage trie root and the contract code, as shown in Figure 1. The recursive inclusion of references to smart contract storage and the code within accounts, and the structure of the state trie $S_n$ allows the entire state at any block $n$ to be securely referenced by the root hash $r_n$. There are other data structures similarly referenced.

\(^2\)While the physical storage format for the blockchain state and contract storage is not necessarily in a trie format, for the purpose of secure references they are represented in the trie form.
by the block header, for example the list of transactions $T_n$ is referenced by the transaction trie root hash $h_n$. Finally, the block header contains a reference to the previous block by the parent block header hash $h_{n-1} = \text{hash}(B_{n-1})$. This forms a hash chain of blocks down to the genesis block $B_0$. Additionally, the block hash does—at least for most public Ethereum networks—play a part in the consensus protocol, although details of the consensus model are not relevant to this work.

The blockchain state is altered between any two sequential blocks $n - 1$ and $n$ primarily by transactions and block rewards. Although for completeness, one can view other attributes such as consensus protocol parameters (e.g., difficulty level) as part of the overall state. Here we focus solely on transactions (and rewards).

Transactions in a block form an ordered sequence $T_n = (t_1, \ldots, t_k)$, and the change incurred to the system state by them is given by $S_n = \Pi(S_{n-1}, T_n)$. $\Pi$ represents the state transition as a function of the previous block’s state $S_{n-1}$ and transactions $T_n$. Note that there are some additional steps (related to block rewards), but for our purposes we can ignore all changes to the blockchain state which do not occur as a result of transactions.

One can view the sequence of transactions $T_n$ from a block $n$ as a sequence of ordered operations on an intermediate, or partially complete, blockchain state $s_i$. The initial state in the evaluation is the state from the previous block $s_0 = S_{n-1}$, and the final state $s_k = S_k$. We define a single-transaction state transfer function $\pi$, such that $s_1 = \pi(s_0, t_1)$, thus $S_n = \Pi(S_{n-1}, T_n) = \pi(\pi(\ldots, \pi(S_{n-1}, t_1), \ldots, t_k-1), t_k)$.

Ethereum transactions may be either value transfers between regular accounts or calls to smart contracts. These differ from each other only marginally and one can view a transaction as a sequence of ordered operations on an intermediate blockchain state. Nevertheless, we can ignore all changes to the blockchain state which do not occur as a result of transactions.

One can view the sequence of transactions $T_n$ as a sequence of ordered operations on an intermediate, or partially complete, blockchain state $s_i$. The initial state in the evaluation is the state from the previous block $s_0 = S_{n-1}$, and the final state $s_k = S_k$. We define a single-transaction state transfer function $\pi$, such that $s_1 = \pi(s_0, t_1)$, thus $S_n = \Pi(S_{n-1}, T_n) = \pi(\pi(\ldots, \pi(S_{n-1}, t_1), \ldots, t_k-1), t_k)$.

Any transaction calling a smart contract results in the execution of the smart contract’s code which has access to the contract’s storage space. The contract may read from and write values to its allocated storage. Additionally, a smart contract may call other smart contracts, which in turn have access to their own storage and can change their own storage as a result. Since Ethereum smart contracts can arbitrarily call other smart contracts, there cannot be prior knowledge of which accounts a transaction will access or modify. However, it is possible to determine each transaction’s input set $\alpha$ (accounts which it accesses) and the output set $\omega$ (accounts which are modified) after it has been evaluated, and these will remain unchanged for any later evaluation of the same block due to the fully deterministic nature of Ethereum transaction semantics. Consequently, while the input and output sets from a transaction at a given block may be unknown prior to their first evaluation, the sets will be identical in all subsequent processing of the block.

**B. LIGHT ETHEREUM SUBPROTOCOL**

Ethereum’s light protocol is called the Light Ethereum Sub-protocol (LES). It is defined as an extension of the standard Ethereum peer-to-peer protocol [50]. LES is behaviorally a client-server protocol, and has been designed to reduce the resource requirements of the client by shifting the burden of maintaining the Ethereum state representation $S_n$ from the client to the server.

The LES protocol requires the client to make its own decision of the canonical chain, i.e., the client will query the chain of blocks down to the genesis block $B_0$, verify the block headers and the proof-of-work hash, and determine the chain with the most amount of “work” as the canonical one. The process is identical to full nodes with one exception: a light node is unable to calculate the block state $S_n$ and therefore is unable to verify the state root hash $r_n$.

When a light node requires an up-to-date representation of an account $a$, it will query the light server for the account data from the server. While it is possible that the client requests state from the latest block $n$, a more prudent strategy is to assume the most recent blocks may be reverted (or provided by an adversary from an intentional fork), and instead query for the account state from some depth $k$, i.e., from block $n-k$ instead. The block depth parameter $k$ is the most important security parameter that the light client controls, and higher $k$ values provide increased security. There is no simple rule for selecting the $k$ value, requiring its choice to be made based on risks related to the use case [51].

When a light protocol server receives a query for state of an account $a$ at a specific block $n-k$, it will return the account state $S_{n-k}$ and the path to the state root in the state Merkle trie, the proof of inclusion. The client will then calculate the hash of the account state, and iteratively calculate the intermediary path hashes up to the root hash $r_{n-k}$, which it then compares against the actual state root hash $r_{n-k}$ retrieved from the block header. The client will reject the server’s reply if there is a mismatch. This allows the client to use a block header $B_n$ to authenticate any subset of the state $S_n$.

If the account $a$ is a regular account, the light client has retrieved all necessary information from the server and can stop. However, if the account is a smart contract and the client wishes to extract information from the contract, it will query the light server for the contract’s program code. Since the smart contract code is referred to by its hash, the client can

---

3On an abstract level we could model block and uncle rewards as an implicit and final transaction in a block. However, such a transaction does not differ externally from “normal” transactions as addressed in this work and consequently has no impact on the formal results.

4There are other fields such as nonce, gas limit and gas price. These have no effect on our analysis, i.e., the nonce by definition must be valid for a transaction to be included in a block. The use of gas limit and gas price are internal to the state transition function $\pi$ and do not affect the relevant features of transactions.
use the hash to verify the authenticity of the returned code. The client will then invoke a so-called view method using the contract code, and follow up with queries to the contract’s storage trie when the contract reads data from its storage. The fetched storage data is returned with a proof of inclusion in the account’s state storage trie. Thus, assuming a valid block header \( B_k \), a light client can subsequently authenticate any data fetched from a light server.

LES has provisions to also query transactions and transaction receipts in a similar way, allowing the client to verify their security by comparing the returned proofs of inclusion against values in the block header. There are also other provisions for the client to receive notifications of new blocks, about completion of specific transactions etc., allowing a light client to be efficiently kept up-to-date on relevant changes to any account.

IV. SUBSET NODES

In this section we propose a new type of Ethereum light node which we refer to as a subset node. We describe its fundamentals, operation, changes needed to the LES protocol, and its security model as well as the necessary formalism. This section describes the light nodes at the protocol level. We emphasize that we purposefully do not consider other related blockchain topics for discussion, such as economics, incentives and consensus.

A. INTRODUCTION

When a regular light node wants to retrieve the state of an account \( a \) for block \( n \), it retrieves this information from the light protocol server. While the light node can check that the retrieved state \( S'_n|_a \) is securely referenced by the state root \( r'_n \) in block \( B'_n \), it is not able to verify that \( S'_n \) follows the Ethereum transaction rules itself, i.e., it is possible that \( S'_n \neq \Pi(S_{n-1}, T'_n) \).

In contrast, a subset node will compute the account’s state locally. For this, it needs to be able to retrieve only those transactions \( X_n \) which either directly or transitively modify or participate in modification of \( S_a|_a \). Assuming it is possible to determine \( X_n \), then a subset node will update its state representation for \( a \) as:

\[
S_n|_a = \Pi(S_{n-1}|_{A(S_{n-1},X_n)}, X_n)|_a
\]  

(1)

This computation depends on transitive input dependencies \( A(S_{n-1},X_n) \) of transactions \( X_n \) and their state from the previous block. This state may have to be recursively calculated to determine the state representations for \( S_{n-1}|_{A(S_{n-1},X_n)} \).

In theory, the subset node would recurse all the way to genesis state \( S_0 \), but in practice the search has to be terminated at some depth \( K \) prior to that. A subset node will use the recursion limit parameter \( K \) to limit the storage and computation requirements. This is used in conjunction with the regular block depth parameter \( k \). If the current chain head is at block \( n \), the goal of the node is to calculate state of the account \( a \) at block \( n-k \), and recurse until it hits the block \( n - k - K \). At this depth, a subset node will terminate the recursion and retrieve the state representation \( S_{n-k-K} \) from the subset server, just like a regular light node would do. At this level, the subset node assumes that the block is not from an adversarial chain.

Thus, for the subset node, the \( k \) parameter determines the latency, i.e., the number of blocks it takes for a transaction to become visible to the node. However, unlike a regular light node which would be susceptible to a state injection attack at block depth \( k \), a subset node is secure against this attack as if was a light node using block depth of \( k + K > k \).

This broad description is elaborated below, with a description of the subset node’s operational model, the necessary extensions to the LES protocol, security of the scheme and an introduction to the necessary formalism that is fundamentally required to prove the correctness of the resolved blockchain subset state.

B. OPERATION

A subset node is fundamentally still a light node, and it will operate similarly by maintaining its own view of the canonical chain. It either retrieves or receives notifications of new blocks, verifying their proof of work, and determines locally the canonical chain and its head \( B_n \). Like a light node, a subset node uses a block depth parameter \( k \) to guard against fork reversals, but in addition it also has the recursion limit parameter \( K \).

The subset node may update its locally stored \( S_n|_a \) either continuously or on demand. The basic operation for both is the same, although if continuous operation is used, the node may benefit from caching intermediary states for non-\( a \) accounts. The detailed description is shown in Algorithm 1, w.r.t. which we will discuss some details below:

- The algorithm reduces to a light node behavior when \( K = 0 \) (i.e., retrieve the implicitly trusted state from block \( n - k \)).
- The node verifies that the input and output sets of returned transactions match what it itself sees when evaluating transactions. Any discrepancy will lead to rejection—the server would be marked as untrusted and would be no longer used.
- Similarly, all invalid transactions must be rejected. Any transaction that would not be allowed in a mined block, such as those with invalid signatures, non-monotonic sequence numbers, value transfers exceeding source’s balance etc. leads to rejection.

In our scheme \( K \) is fixed, but it could be dynamically adjusted, or the algorithm could be allowed to run until either no further dependencies are found (\( I^* \) is empty) or a pre-set maximum for number of referenced accounts \( n_A \) or transactions \( n_T \) is exceeded.

C. PROTOCOL EXTENSION AND SERVER BEHAVIOR

The operation of the subset node as described above contains one previously undefined request sent to the light protocol server (Algorithm 1, line 4). Similar to other LES requests,
Algorithm 1 Subset Node Updating the State of Account \( a \), With Blockchain Head at Block \( n \), Block Depth \( k \) and the Recursion Limit at \( K \). Using the Subset Protocol Server to Query for Dependent Accounts and Transactions. On Abort, No Changes Are Committed and the Server Is Marked as Dishonest:

1: \( d \leftarrow n - k \) \quad \triangleright \text{block level we’re looking at} \\
2: \( I^*_{d} \leftarrow \{a\} \) \quad \triangleright \text{accounts to determine at depth } d \\
3: \text{while } d \neq n - k - K \text{ do} \quad \triangleright \text{K times} \\
4: \quad X_d, A_d, \Omega_d \leftarrow \text{retrieve from server for accounts } I^*_d \\
5: \quad \text{abort if proof of inclusion for } X_d \text{ is invalid} \\
6: \quad I^*_{d - 1} \leftarrow A_d \quad \triangleright \text{need states of } A_d \text{ from depth } d - 1 \\
7: \quad d \leftarrow d - 1 \\
8: \text{end while} \\
9: \quad S^0_d | I^*_d \leftarrow \text{retrieve from server} \quad \triangleright \text{If } K = 0, d = n - k \\
10: \quad \text{abort if proof of inclusion for } S^0_d \text{ is invalid} \\
11: \text{while } d \neq n - k \text{ do} \quad \triangleright \text{K times} \\
12: \quad d \leftarrow d + 1 \quad \triangleright d \in [n - k - K + 1, n - k] \\
13: \quad S^0_d \leftarrow \Pi(S^0_{d - 1} | A_d, X_d) \\
14: \quad \text{abort if } \Pi \text{ detects an invalid transaction} \\
15: \quad \text{abort if } \Pi \text{ refers account } \notin A_d \\
16: \quad \text{abort if } \Pi \text{ does not refer all accounts } \in A_d \\
17: \quad \text{abort if } \Pi \text{ modifies account } \notin \Omega_d \\
18: \quad \text{abort if } \Pi \text{ does not modify all accounts } \in \Omega_d \\
19: \text{end while} \\
20: \quad S_{n-k-1}^{n-k} \leftarrow S^0_{n-k-1} | a \quad \triangleright d = n - k \\
21: \quad \text{commit results}

This would refer to the block \( d \) by its block hash and include a list of account identifiers. The response would contain a list of transactions \( X_d \), list of accounts forming the input set \( A_d \), and a list of accounts forming the output set \( \Omega_d \). We assume the \( X_d \) response is either accompanied with proof of inclusion in the transaction trie or this is retrieved independently by the client.

The additional task required of the server is to maintain \( \alpha \) and \( \omega \) values for each transaction in a block. When the client queries for modifications to accounts \( I^*_d \), it will compute the set of transactions which modify directly or indirectly those accounts. This set may, of course, be empty. An alternative approach would be to precompute the transitive dependencies of all modified accounts in a block, and just perform an union of dependent accounts and transactions over that set. While the former one would result in smaller storage requirements, the latter one would require less computing at a request time.

D. SECURITY

The relevant comparison for subset nodes is against light nodes as a baseline over which it provides improvements, and against full nodes which provides an upper boundary that it can strive towards. The basic security claim for subset nodes is that it provides equivalent or higher security for a given \( k \) value than a pure light node achieves. This is trivial to show. Firstly, if \( K = 0 \), the behavior of a subset node is identical to a light node. When \( K > 0 \), the “ground truth” state is retrieved from block \( n - k - K \). The same block would be used by another light node with \( k' = k + K \). Consequently a subset node has the same “base” level of assurances as a client using block depth \( k + K > k \).

This does not, though, mean that a subset node has the same level of security assurances as a full node using \( k \) as its block depth. The security of subset node is somewhere between a light client and a full node for a given \( k \) value (for \( K > 0 \)). If we assume that an adversary is unable to provide incorrect state at depth \( k - K \), but is still able to provide a forked block at some intermediate block \( d \in (n - k - K, n - k) \), it may provide the client with a transaction list \( X_d \) of its own choosing. In this case the adversary can:

- Generate arbitrary transactions for accounts the adversary controls. If these transactions are valid starting from state \( S_{n - k - K} \), the client sees these as valid, even if they are never sent to the honest network. This is just a variant of the double-spending attack, and does not change the security assurances of a subset node when compared to other node types.
- Choose which transaction to include if the sender has sent multiple transactions with the same sequence number (“nonce”) that were not yet part of the transaction history at block \( n - k - K \). Issuing a new transaction with already-used sequence number is often used to cancel pending transactions. Thus, an adversary may be able to disregard the cancel transaction in the history it provides to the client.
- Choose to stop the transaction stream from a sender on any point after those included in block \( n - k - K \). The adversary can only choose to omit any further transactions from a point as Ethereum mandates strict monotonicity for the transaction sequence number. This could mean, for example, that the adversary can withhold a transaction from the client to its advantage, such as a “revoke” operation sent by a party upon realizing a mistake they have made.

In contrast, the adversary is not able to generate fake transactions for accounts it does not have the signing key for. A subset node will validate the signature of any transaction it receives, and reject any transaction with an invalid signature.

While the options are severely limited for an adversary, they may still be able to exploit either operational issues (withholding transactions addressing mistakes made to the adversary’s advantage) or transaction ordering bugs in smart contracts in a way that remains invisible to the honest network participants. Note that many of the problems facing a subset node are similar to smart contracts faced with frontrunning attacks.

Finally, as with a normal light client, if the adversary is able to provide invalid state representation at depth \( k + K \) so that the provided \( S_{n - k - K} \neq \Pi(S_{n - k - K - 1}, T_{n - k - K}) \) then the client is at a loss. Thus, the same caveats discussed earlier relating to the selection of a suitable \( k \) (and now also \( K \)) based on the level of risk still apply.


**E. TRANSACTION SUBSETS**

The correct behavior of subset nodes rests on being able to evaluate only a subset of transactions from a block while still being able to accurately and permanently determine the correct state $S_a |_a$ of a target account $a$. For a full node this process is straightforward when a new block $B_n$ arrives: take the preceding state $S_{n-1}$ and transactions $T_n$ from the current block, compute $S_n = \Pi(S_{n-1}, T_n)$, and retrieve the state for the target account from that as $S_n |_a$.

In the earlier description of the subset node behavior we assumed that there exists a subsequence $X_a \subseteq T_n$ that we use to correctly determine $a$’s state. Formally, we need to show that $X_n$ alone affects $a$’s state, and evaluation of other valid subsequences from the same block will not change $a$’s state as determined from $X_n$ alone. The following theorem is necessary for this property (the formal proof of this theorem is given in Appendix).

**Theorem 1:** For a sequence of transactions $T$ in block $B$, a sequence $X \subseteq T$ that is disjoint from remaining transactions $T' = T \setminus X$ in $S$, and any valid interleaving of transactions $T''$ from $T$ which includes $X$ the following equality holds for all $a \in \Omega(S, X)$:

$$\Pi(S, X) |_a = \Pi(S, T'') |_a$$  \hspace{1cm} (2)

The theorem states that when no dependencies exist between two transaction sequences (they are disjoint), any valid subsequence from either one can be interleaved arbitrarily into the other without affecting the outcome. Whilst this statement is intuitively true, to the best of our knowledge this has not been previously formally proven for the Ethereum transaction model.

This property is important for subset nodes to operate correctly. Consider what would happen if we evaluated a set of transactions $X$ from $T$ for an address $a$, and later we wanted to also calculate the state of address $b$. To calculate $b$, we could identify that the transaction sequence $Y$ has to be evaluated. If the outcome of calculating a valid interleaving $X \cup Y$ of the two results in difference in $a$’s state ($\Pi(S, X \cup Y) |_a \neq \Pi(S, X) |_a$), any decision or action based on $a$’s earlier value would have been based on an incorrect state. Theorem 1 disallows this from happening and is therefore crucial for the formal correctness of subset nodes.

The relevant outcome of the formalism regarding subset nodes is that it is possible to identify sequences of transactions in a block that allow us to correctly evaluate the state of an account. This sequence, when properly identified, may be smaller than the full sequence of transactions in the block. Furthermore, the calculated state of the selected account is guaranteed to be immutable with respect to the block, meaning it will not change due to the computation of other accounts state for the same block.

**V. METHODS**

In this section we describe the analyzed data set, our analysis methods, and their limitations.

---

**A. DATA PROCESSING**

Our transaction trace analysis was based on the public Google BigQuery Ethereum transaction trace and smart contract datasets. The traces were collected for a period of one year from June 2019 to June 2020.

The recorded execution traces includes information about calls across smart contracts (“internal transactions”), but it does not identify whether changes were actually made to the target contract state. Therefore, we take a conservative approach and assume any contract that is called modifies its account state or contract storage.

In addition to contract calls and their originating and target account addresses, we also extract the beneficiary addresses of block and uncle reward payments. From these we generate the input and output address sets for each transaction (and reward), and extract these as tuples of (block, index, address, input addresses, output addresses). The tuples are generated using a BigQuery query, and then transferred to a Postgres database where further data refinement and partial data analysis is performed. Final data analysis and graph plotting is performed using the R statistical program.

**B. DEPENDENCY GENERATION**

To analyze the growth of dependencies determining the state of an account, we start with a “mutation point”. This defines the set of all blocks where accounts are modified as (block, address) pairs. We only need to analyze these for any account, since if an account is modified at block $n$, then analyzing it at block $n+1$ is equivalent to decreasing $K$ by one when analyzing block $n$. Therefore, the upper bound for transaction and account dependencies can be determined from account mutation points.

For any such point we can calculate $(K', n_A, n_T)$ where $K'$ is the specific depth of the $K$ parameter ($1 \leq K' \leq K$), and $n_A$ and $n_T$ are the number of dependent distinct addresses and dependent number of transactions up to $K'$, respectively. Notice that $n_A$ counts only distinct addresses, as the same account may be referred to multiple times by different blocks within the $K'$ range. All transactions are unique, and no similar reduction for $n_T$ occurs.

Due to limited computing resources we sampled the full mutation point set with a Bernoulli trial sampling method set to 1% sampling rate. The final set of $K$-analysis values are calculated from this sampled set for values $1 \leq K \leq 12$.

**C. ACCOUNT CATEGORIZATION**

We adapted the labeling scheme used by di Angelo and Salzer for categorizing different account types based on how often they interact within the dataset [24, s. 6.2]. The original cut-off values were adjusted to match our smaller block range. We used the busy bee boundary from their work to divide regular accounts and contract accounts into either high or

---

5. This is the [bigquery-public-data:crypto_ethereum](https://bigquery-public-data.googleusercontent.com/query) dataset. Google blog posts provide further details on how the dataset is generated [52] and how it can be used [53].
other categories. The number of unique reward accounts is so low that they are all included as a single category. These categories and their limits are shown in Table 1.

### D. LIMITATIONS

There are two main limitations in our analysis originating from our dataset and how we process it. The first limitation stems from the fact that the Google dataset includes only calls either by a transaction or between smart contracts. It does not, however, identify whether the state of the recipient account was modified as a result. A transaction may fail to result in a change of its target account’s state for various reasons, e.g., lack of necessary permissions, incorrect parameters, etc. We cannot distinguish these from state-changing transactions and as a consequence they will be false positives in our results. The only exceptions are static and delegate calls, where we can usually infer that the target smart contract will not be modified. As a consequence our estimates for $n_A$ and $n_T$ dependencies are conservative i.e., upper bound estimates.

Another data processing issue is that we do not differentiate the block beneficiary account from uncles when determining reward dependencies. While this does lead to some excess dependencies, we did not see this as major issue as the pool of reward addresses is relatively small, and almost all reward addresses occur repeatedly within the data set.

### VI. RESULTS AND ANALYSIS

Below we describe the results of the data analysis, and evaluate them in the context of subset nodes. The results include overall description of the data set and its main statistics, consideration of several features that might reduce the applicability of the data to evaluation of subset nodes, and analysis of transaction dependencies as a function of the $K$ parameter.

#### A. OVERVIEW

We analyzed blocks starting from block 7,870,425 to block 10,176,689, spanning one year interval from the first block on 1 June 2019 UTC to the last block on 31 May 2020 UTC. The number of blocks, transactions, referenced accounts, reward accounts and other general metrics from the dataset are shown in Table 2.

There are a total of 430,580,405 account modifications within blocks (i.e., multiple different transactions on the same address in a single block are counted once). Out of this set, we stochastically sampled at 1% rate, resulting in 4,302,611 block-address pairs as the sample set from which dependencies were determined for different $K$ values.

The address categorization is shown in Table 3, showing the number of accounts and their portion of all accounts modified within the analysis range. The table also contains all account modifications by their address type, as well as the same categorization for the sample set. D. Angelo and Salzer noted that over 99% of all contracts were casual, in our dataset the ratio is even higher at 99.89%.

#### B. TRANSACTIONS, INPUTS AND OUTPUTS

The number of distinct addresses referenced by transactions in a single block varies, but has a well-defined peak at slightly over 250 addresses per block, as shown in Figure 2. It is a well-known phenomenon that some miners do not go to the trouble of computing and including any transactions in their blocks, and consequently there are a large number of blocks where the only account reference ($n_D = 1$) is the miner’s beneficiary account.

We can also analyze the frequency of address references, as shown in Figure 3 which exhibits—at least at a quick
TABLE 3. Summary of address categories, from left to right: all distinct modified addresses within the analysis block range, modifications by address category, and samples by category. The last one is not exactly 1% of previous as a stochastic sampling method was used.

<table>
<thead>
<tr>
<th>Category</th>
<th>of addresses</th>
<th>of modifications</th>
<th>of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
<td>%</td>
<td>n</td>
</tr>
<tr>
<td>Account</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>53,961</td>
<td>0.13</td>
<td>117,163,876</td>
</tr>
<tr>
<td>Other</td>
<td>28,576,372</td>
<td>70</td>
<td>162,887,492</td>
</tr>
<tr>
<td>Contract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>13,958</td>
<td>0.034</td>
<td>124,701,354</td>
</tr>
<tr>
<td>Other</td>
<td>12,347,147</td>
<td>30</td>
<td>22,527,645</td>
</tr>
<tr>
<td>Reward</td>
<td>518</td>
<td>0.0013</td>
<td>3,300,038</td>
</tr>
<tr>
<td></td>
<td>40,991,956</td>
<td></td>
<td>430,580,405</td>
</tr>
</tbody>
</table>

One consideration is whether any large-scale change in the behavior of Ethereum usage has occurred during our data range. There are several plausible reasons why such a change could have occurred (e.g., the COVID-19 pandemic). Therefore we look at how the number of transactions and the number of referenced accounts per block have developed over the data range, as shown in Figure 4. Due to the nature of Ethereum, some variability over time is to be expected. However, we observe that there was no significant change over time. There is an apparent change in the upper three sigma boundary on transaction inputs, but the mean and median values do not show any drastic changes. It may be that some of the changes are part of a yearly cycle, which would not be identifiable in our single-year sample. We therefore conclude that no obvious reasons to reject any further results is present in the broad usage patterns of the Ethereum blockchain.

C. ACCOUNT USAGE
Based on the difference on usage frequencies between regular and contract account types shown above, we take a closer look at the behavior of different account types (as defined in Table 1) over time. One metric to inspect is the inter-modification interval, i.e., the difference between two subsequent modifications of a single account. As shown in Figure 5, we can identify the reward and “high” category accounts are more frequently referenced than the “other” category. At first, this may appear as a direct result of our categorization—after all, an account that is referenced at most 334 times would by inference have on average \( \frac{333}{6900} \) blocks between each reference. However, it could be possible that infrequently accessed accounts and smart contracts are used as a cluster, i.e., all of the references are closely spaced in time. The fact that the intra-access interval density is relatively flat for the “other” account types is an argument against such clustering.

D. DEPENDENCY GROWTH
The main measure of subset node feasibility is how fast the number of accounts \( n_A \) and the number of transactions

FIGURE 3. The distribution of address occurrences for regular accounts and smart contracts.
$n_T$ grow as a function of $K$. For this, we calculated these values for a subset from all account modifications of the data range for values $1 \leq K \leq 12$. The subset consists of 4,302,611 unique (block, address) tuples. Using this data set, we can determine the portion of block-account modification pairs $p_A$ for which the number of referenced accounts is less than a given maximum value, i.e., $n_A \leq \max_A$. The same is done for the portion of transactions $p_T$ for which the number of transactions is less than the threshold $n_T \leq \max_T$. Thus, considering an account that is modified in a block, we can see in Figure 6 the number of account states that must be retrieved from an earlier block (including at least the modified account’s earlier state), and the number of transactions that must be evaluated to correctly determine the new state (including at least the transaction triggering the account update).

From the figure, we can make two relevant observations. First, as $p \to 1$, the $n_A$ and $n_T$ values converge towards approximately the same number for all account categories. Secondly, the absolute $n_A$ and $n_T$ values are relatively small, less than 10,000 for all cases. This indicates that almost all accounts—including smart contracts—are connected to only a limited number of accounts when a small $K$ block window is used.

Furthermore, as would be expected based on the earlier observation that different account types exhibit different access patterns, there are clear differences how quickly dependencies rack up. For most account types, the growth is shallow. Both regular account types (high and other) exhibit mostly very similar behavior, which is surprisingly similar to high contract category. For $n_A$—but not $n_T$ the contract other category stands out. Finally, the fact that reward accounts increase their $n_A$ dependencies quickly is not a surprise, as reward accounts intrinsically will form a dependency on all accounts which transact on a given block.

The minimum number of accounts $n_A$ and transactions $n_T$ required to consider for a given $p$ and $K$ values are summarized in Table 4. For example, from the table we can find that to fully evaluate 99% of all account modifications at $K = 12$, we need to determine a total of $\max_A = 4800$ account states and evaluate $\max_T = 3600$ transactions. However, the difference in values in Table 4 between the lowest and highest $K$ values is within a single magnitude, i.e., roughly similar when we put them into their proper context. There were over 41 million modified accounts, 260 million transactions, and 430 million distinct account modifications within the data range that a full node has to process and store. In this context the cost of processing a few thousand transactions and requiring storage for a similar number of accounts is tiny in comparison.

### E. SERVER RESOURCES

While we have so far looked at the proposed scheme from the viewpoint of the subset nodes, we need to consider also the cost from the server’s point of view. The increased network communication is difficult to characterize, and instead we will focus on the additional storage required to store...
FIGURE 6. \( \text{max}_A \) and \( \text{max}_T \) vs. the proportion of addresses whose \( n_T \) fall at or below, broken for each \( k \in (3, 6, 9, 12) \) separately. Please note that the vertical axis starts at one as, by definition, all modifications of an Ethereum account require access to at least that account’s previous state, and at least one transaction to initiate that change.

TABLE 4. The number of addresses \( n_A \) and transactions \( n_T \) that a subset node needs to be able to process for a given \( K \) value to be able to process a given portion \( p \) of all Ethereum account mutations within the analyzed data range. The limits are given either for all transactions, or broken down by the category to regular accounts, contracts, and reward addresses.

<table>
<thead>
<tr>
<th>Category</th>
<th>( K = 3 )</th>
<th>( K = 6 )</th>
<th>( K = 9 )</th>
<th>( K = 12 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( p = 50% )</td>
<td>( 90% )</td>
<td>( 99% )</td>
<td>( 50% )</td>
</tr>
<tr>
<td>Referenced distinct addresses (( n_A ))</td>
<td>680</td>
<td>1400</td>
<td>2100</td>
<td>1100</td>
</tr>
<tr>
<td>Account</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>500</td>
<td>1000</td>
<td>1800</td>
<td>730</td>
</tr>
<tr>
<td>Other</td>
<td>670</td>
<td>1400</td>
<td>2100</td>
<td>1100</td>
</tr>
<tr>
<td>Contract</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>350</td>
<td>680</td>
<td>1000</td>
<td>560</td>
</tr>
<tr>
<td>Other</td>
<td>400</td>
<td>740</td>
<td>910</td>
<td>690</td>
</tr>
<tr>
<td>Reward</td>
<td>470</td>
<td>860</td>
<td>1200</td>
<td>720</td>
</tr>
<tr>
<td>Transactions (( n_T ))</td>
<td>710</td>
<td>1300</td>
<td>1500</td>
<td>1100</td>
</tr>
<tr>
<td>Account</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>500</td>
<td>940</td>
<td>1300</td>
<td>740</td>
</tr>
<tr>
<td>Other</td>
<td>700</td>
<td>1300</td>
<td>1500</td>
<td>1100</td>
</tr>
<tr>
<td>Contract</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>320</td>
<td>610</td>
<td>880</td>
<td>510</td>
</tr>
<tr>
<td>Other</td>
<td>260</td>
<td>520</td>
<td>730</td>
<td>430</td>
</tr>
<tr>
<td>Reward</td>
<td>450</td>
<td>820</td>
<td>1200</td>
<td>690</td>
</tr>
</tbody>
</table>

The account and transaction dependency information for the server to be able to reply efficiently to the client’s requests.

To model the space requirement, we assume the server will store information per block about all the modified accounts, the list of referenced accounts and list of dependent
transactions for each modified account. This represents a fully “expanded” view of the data. Since there are more storage-efficient approaches, this model offers an approximation of the upper bound of storage requirements a server serving subset nodes would face.

To calculate the space requirement, we assume the server will store, for each transaction in the block (and for the block rewards, as well), a list of dependent accounts and transactions. Therefore, for each block there will a total of $s_d$ recorded dependent accounts and $s_t$ dependent transactions. We assume each account is referenced by its address (20 bytes), and each dependent transaction is referenced by a simple integer (4 bytes). Therefore, the size (in bytes) is $S = 20 \times s_d + 4 \times s_t$.

Over the whole analysis range, the average sum of the number of dependent accounts of all transactions in a single block is $s_d = 1440$ and for dependent transactions $s_t = 1183$, providing the average storage size per block $S = 33540$ bytes. These values do, however, vary significantly over the analysis range as shown in the leftmost graph in Figure 7, where the gray range shows the 99.7% range of values.

It is somewhat surprising that while the average storage requirement per block is less than a hundred kilobytes, there exist blocks where $S$ is several megabytes. We inspected a sample of these outliers, and found out that these are mostly due to mining pool paying out rewards to participants of the mining pool. Since each payment from a mining pool is dependent on the payee account’s balance, each such transaction consequently depends on all preceding dependencies from the same batch of payments. It is highly likely that in such cases a more compact representation would reduce the total storage requirement substantially, although we do not pursue this possibility.

Regardless, these rather infrequent anomalous blocks are a minority as can be shown in the graphs b and c in Figure 7. These show the results when the server stores the dependency information for either a window of 1000 blocks or 30,000 blocks (the former one is arbitrarily chosen, while the latter is the epoch size of Ethereum). The variability naturally reduces as the window increases. Regardless, we can see that the total storage size has always been under one gigabyte for a window of 1000 blocks, and less than 3.5 gigabytes for one-epoch long window. It is clear that smaller windows will have a lower average storage requirements, but with much higher variability.

Since the additional storage requirement for a server for storing transaction dependencies is “only” a few gigabytes even for the largest window of 30,000, we do not consider the additional storage requirements at the server to be an obstacle to this scheme.

VII. CONCLUSION

This work has proposed a new Ethereum light client node type called a subset node, describing its behavior, the necessary protocol changes, and a statistical analysis based on historical Ethereum transaction traces. We demonstrate that subset nodes attain greater security assurances than regular light clients with only a modest increase in resource requirements compared to light clients. We have also provided the theoretical formalism necessary to demonstrate the soundness of the subset node scheme in its handling of individual account states.

Our results are limited to providing an upper bound, or worst-case, estimates for the number of dependent accounts and transactions that a subset node has to store and evaluate when determining an accurate state for a single account. Therefore we expect that a more accurate tracing of an Ethereum smart contract’s load and store operations would greatly reduce the number of true dependencies, and thus reduce the estimated bounds of resource requirements. Regardless of these limitations, our results show a significant reduction in resource requirements compared to a full node when maintaining a locally computed version of an arbitrary account’s state.
We acknowledge that the subset node scheme may introduce new attack vectors, and we identify an adversary artificially inflating dependencies as one of those. This points to further work in this field, and we welcome such security analysis of our scheme from other researchers, as well as any potential mitigation strategies against those attacks.

The proposed changes to the LES protocol are intentionally designed to be local, i.e., they can be deployed on a client and server basis without requiring changes to the overall Ethereum protocol. This has the disadvantage that transaction dependency information is unauthenticated and could be corrupted by a malicious node. While any subset node has to double-check the dependency information during its local processing, in our scheme any discrepancy is shown only at the time of its local validation, and server basis without requiring changes to the overall system.

### APPENDIX

#### PROOF OF TRANSACTION SUBSETS

Here we provide the formalism required to prove Theorem 1 as well as details of the used notation (see Table 5). We start by providing some definitions that are used to prove the main theorems. Theorem 2 proves that the order in which two disjoint transactions are evaluated does not alter the outcome of the final state. Theorem 3 proves that sequences of disjoint transactions can be evaluated independently of one another. Theorem 4 proves that interleaving disjoint sequences of transactions does not change the result. Finally, Theorem 5 demonstrates the immutability of subset state for disjoint transaction sequences.

#### A. TRANSACTIONS AND STATE

We define the Ethereum storage state (usually we write just \( \mathcal{S} \)) as a mapping from addresses \( A \) to values \( \mathcal{V} \), with the possible space of states being \( \mathcal{S} \) so that an individual system state is a mapping \( \mathcal{S} : \mathcal{A} \rightarrow \mathcal{V} \), and \( \mathcal{S} \in \mathcal{S} \).

**Definition 1 (State Equivalence):** States \( \mathcal{S} \in \mathcal{S} \) and \( \mathcal{S}' \in \mathcal{S} \) are equivalent, and we write \( \mathcal{S} = \mathcal{S}' \), if \( \text{dom}(\mathcal{S}) = \text{dom}(\mathcal{S}') \) and \( \forall a \in \text{dom}(\mathcal{S}) : (a) = (a) \).

**Definition 2 (State Complement):** For any state \( \mathcal{S} \), we define \( \mathcal{S} \setminus \mathcal{X} = \{ (a) : a \in \mathcal{S} \} \). This is a notational shortcut for a “pseudo-complement” so that \( \mathcal{S} \setminus \mathcal{X} = \mathcal{S} \) and does not extend to the full \( \mathcal{A} \). While each address \( a \in \mathcal{A} \) is “defined” in Ethereum (there is a default value), using this notation allows us to avoid overlapping domains when joining state subsets.

**Definition 3 (Extended \( \mathcal{T} \) and \( \mathcal{T}^* \)):** \( \mathcal{T}(\mathcal{S}) \) is the set of all possible valid transactions in state \( \mathcal{S} \) and \( \mathcal{T}^*(\mathcal{S}) \) is the set of all valid transaction sequences starting from state \( \mathcal{S} \).

For practical reasons, we use \( \mathcal{T} \) and \( \mathcal{T}^* \) as a shorthand for all valid transactions and transaction sequences in the current context. This is a necessary detail since in Ethereum the acceptability of a transaction to evaluation is dependent on the blockchain state just before the transaction is evaluated:

\[
\exists \mathcal{S} \in \mathcal{S}, \{a, b\} \subseteq \mathcal{T}(\mathcal{S}) : \langle a, b \rangle \in \mathcal{T}^*(\mathcal{S}) \land \langle b, a \rangle \notin \mathcal{T}^*(\mathcal{S})
\]  

**Definition 4 (State Transition Function):** \( \pi : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{S} \) is a function that evaluates a transaction \( t \) in state \( \mathcal{S} \), resulting in updated state \( \mathcal{S}' \), i.e., \( \mathcal{S}' = \pi(\mathcal{S}, t) \).

It suffices for our analysis to define \( \pi \) as evaluating the transaction and updating the state according to the Ethereum evaluation rules. Note that formally the blockchain state \( \mathcal{S} \) as we have defined it does not include information about the current block or a block that is being mined. Formally, this extra information \( E \), should be included and we would write \( \pi(\mathcal{S}, t, E) \), but for notational convenience, we assume \( \pi(\mathcal{S}, t) \) is aware of the context. All transaction evaluations occur, after all, within the scope of a single block, and a single distinct transaction, once it is part of the blockchain history, cannot be included in any future block.

**Definition 5 (Transaction Output):** \( \omega(\mathcal{S}, t) \), for \( \mathcal{S} \in \mathcal{S} \) and \( t \in \mathcal{T} \) is the set of outputs from the transaction \( t \). Formally:

\[
\omega(\mathcal{S}, t) = \{ a \mid a \in \mathcal{A} : (a) = (a) \} \neq \mathcal{S} \mid_{\mathcal{A}}
\]

\( \omega(\mathcal{S}, t) \) is the set of addresses whose mapping change between \( \mathcal{S} \) and \( \mathcal{S}' = \pi(\mathcal{S}, t) \).
Definition 6 (Transaction Input): $\alpha(S, t)$, for $S \in S$ and $t \in T$ is the set of inputs for the transaction. Formally,

$$\alpha(S, t) = \arg\min_{X \in p(A)} |X|$$ \hspace{1cm} (5)

For $\alpha(S, t)$ there is an infinite number of address sets with at least one address that is a true input (which, if missing, causes the result to differ), so we settle for the smallest such address set that must be included for the correct result.

Having defined $\alpha$ and $\omega$, we can define the state transition function $\pi$ in terms of those two functions as follows:

$$\pi(S, t) = \pi(S[S\alpha(S, t)], S[S\omega(S, t)])$$ \hspace{1cm} (6)

By definition, passing the $\alpha$ subset of state to transaction $t$ in state $S$ requires only that subset of states and no other state for a correct result. Similarly, the output of this evaluation will change only states in the $\omega$ set. Therefore we can extract non-$\omega$ states directly from the original state and use them in the final result without passing them to $\pi$.

Proposition 1: $\omega(S, t) \subseteq \alpha(S, t)$. The set of modified addresses is always included in the set of input addresses.

Proposition 2: $\text{from}(t) \in \omega(S, t)$. The sender’s address is always an output of the transaction.

First we note that Proposition 2 also implies $\text{from}(t) \in \alpha(S, t)$. Both propositions 1 and 2 require some explanation with regards to Ethereum transaction semantics. Any modification to Ethereum’s storage state requires access of the original state, thus any modified state must have its account address as input to the transaction. Furthermore, a transaction must always originate from a valid address, as a valid transaction must be digitally signed, and in Ethereum this will also indicate the address of the signer. A single transaction will — at least — increment the account nonce of the sender by one. Thus, the state of the sender’s account is always modified by the transaction, therefore the address of the sender is in both the $\alpha$ and $\omega$ sets.

Definition 7 (Bulk State Transition Function): $\Pi : S \times T^* \rightarrow S$ evaluates a sequence of transactions in order, passing updated state from one transaction to the next, with the result being updated state from the last transaction. We have that:

$$\Pi(S, \emptyset) = S$$

$$\Pi(S, (t)\Pi(T')) = \Pi(\pi(S, t), T')$$ \hspace{1cm} (7)

For example, two transactions $(a, b) \in T^*$ are evaluated in order, first $a$ and then $b$: $\Pi(S, (a, b)) = \pi(\pi(S, a), b)$.

B. TRANSACTION DISJOINTNESS

Definition 8 (Disjointness of Transactions): Two transactions $a \in T$, $b \in T$ are disjoint in state $S$ if for the sequence $(a, b)$: $\omega(S, a) \cap \alpha(\pi(S, a), b) = \emptyset$ and $\omega(\pi(S, a), b) \cap \alpha(S, a) = \emptyset$.

That is, the output from either transaction has no overlap with the input from the other transaction. We can now show, in Theorem 2, that evaluating two disjoint transactions from the same state can be performed in any order.

**Theorem 2:** For any disjoint transactions $a \in T$ and $b \in T$, the evaluation of $(a, b)$ and $(b, a)$ are identical.

**Proof:** First we can define some shorthand: $\alpha_a = \alpha(S, a)$, $\omega_a = \omega(S, a)$, $\alpha_b = \alpha(\pi(S, a), b)$, $\omega_b = \omega(\pi(S, a), b)$, and note that disjointness property expressed with this notation is $\omega_a \cap \alpha_b = \emptyset$ and $\omega_b \cap \alpha_a = \emptyset$.

The final state $S'$ after evaluation of sequence $(a, b)$ can be expanded using Equation 6:

$$S' = \Pi(S, (a, b)) = \pi(\pi(S, a), b)$$

$$= \pi(\pi(\pi(S|\alpha_a, a)|\omega_a \cup S|\alpha_b, b)|\omega_b, b)|\omega_a \cup S|\alpha_b, b)|\omega_a \cup S|\alpha_b, b)$$

$$= \pi(\pi(S|\alpha_a, a)|\omega_a \cup S|\alpha_b, b)|\omega_b, b)|\omega_a \cup S|\alpha_b, b)$$

We note that for restrictions, $f|_{A \cap B} = f|_{A \cup B}$ and $(f \cup g)|_A = (f|_A \cup g|_A)$ and use these to move some of the restrictions inward:

$$S' = \pi(\pi(S|\alpha_a, a)|\omega_a \cap \alpha_b \cup S|\alpha_b \cap \alpha_a, b)|\omega_b, b)|\omega_a \cup S|\alpha_b, b)$$

Next use the disjointness property to eliminate the first innermost $\pi$ as $\omega_a \cap \alpha_b = \emptyset$, and simplify some of restrictions further by noting that $\omega_a \cap \alpha_b = \alpha_a$, $\omega_a \cap \omega_b = \omega_a$:

$$S' = \pi(S|\alpha_b, b)|\omega_b, b)|\omega_a \cup S|\alpha_b, a)|\omega_a \cup S|\omega_a \cup \omega_b)$$

Now we note that if we expand $\Pi(S, (a, b))$ we get an identical result. Consequently, given disjointness assumption of the two transactions, their order of evaluation does not matter and hence it holds that:

$$\Pi(S, (a, b)) = \Pi(S, (b, a))$$

□

C. TRANSACTION SEQUENCE DEPENDENCIES

Before progressing, we define $A$ and $\Omega$, which are similar to $\alpha$ and $\omega$, except they work over a transaction sequence instead of a single transaction.

**Definition 9 (Transaction Sequence Input):** $A : S \times T^* \rightarrow \varphi(A)$ returns the set of input addresses to a sequence of transactions evaluated within a transaction sequence.

For a (starting) state $S$, and two sequences of transactions $T$ and $X$ satisfying $\forall t \in X : t \in T$, we have the following properties for function $A(S, T, X)$:

$$A(S, (t)\Pi(T', X))$$

$$= \begin{cases} \emptyset & X = \emptyset \\ A(\pi(S, t), T', X) & t \notin X \\ \alpha(S, t) \cup A(\pi(S, t), T', X \setminus \{t\}) & t \in X \end{cases}$$ \hspace{1cm} (12)

It is not possible to define $A$ directly over $X$, since a sequence $T = (a, b, c)$ may differ from $X = (a, c)$ if transaction $b$ changes the behavior of transaction $c$. Thus, even if

110098
we are interested in the $A$-set of only a subset of transactions, we must evaluate $\alpha$ in the correct state with respect to the transaction sequence.

**Definition 10 (Transaction Sequence Output):** $\Omega : S \times T^* \times T^* \to \varphi(A)$ is the set of output addresses for a sequence of transactions evaluated within a transaction sequence.

We omit the detailed definition for brevity as $\Omega(S, T, X)$ is defined in a similar manner to $A(S, T, X)$, see Definition 9 above.

**Definition 11 (Transaction Sequence Disjointness):** For a transaction sequence $T \in T^*$, we say two subsequences $X \subseteq T, Y \subseteq T$ satisfying $\forall x \in X, y \in Y : x \notin Y \land y \notin X$ are disjoint sequences if $\Omega(S, T, X) \cap A(S, T, Y) = \emptyset \land \Omega(S, T, Y) \cap A(S, T, X) = \emptyset$.

### D. DISJOINT TRANSACTION SEQUENCES

The following theorem proves that any two disjoint subsequences of transactions can be evaluated independently of one another.

**Theorem 3:** If $X, Y \subseteq T \in T^*$ then those subsequences can be evaluated in isolation from each other.

**Proof:** We start by splitting $T$ into two sequences $X$ and $Y$ that are disjoint as by Definition 11:

$$X \subseteq T \quad Y = T \setminus X$$

$A_X = A(S, T, X) \quad A_Y = A(S, T, Y)$

$$\Omega_X = \Omega(S, T, X) \quad \Omega_Y = \Omega(S, T, Y)$$

$A_Y \cap \Omega_X = \emptyset \quad A_X \cap \Omega_Y = \emptyset$

We want to show that the following identity holds i.e., there is no specific ordering required between sequences $X$ and $Y$:

$$\Pi(S, T) = \Pi(S, X)|_{\Omega_X} \cup \Pi(S, Y)|_{\Omega_Y} \cup S|_{\Omega_X \cup \Omega_Y}$$  \hfill (13)

We proceed by showing that $\Pi(S, T)|_{\Omega_X} = \Pi(S, X)|_{\Omega_X}$.

The first step is to show that we can reduce the full sequence $T$ by removing its last element if it is not in $X$.

Let $T = T' \|(t) \quad S' = \Pi(S, T') \quad \alpha_t = \alpha(S', t) \quad \omega_t = \omega(S', t)$

we can take $\Pi(S, T)|_{\Omega_X}$ and expand the restriction:

$$\Pi(S, T'\|(t))|_{\Omega_X} = [\pi(S'|_{\alpha_t}, t)|_{\omega_t} \cup S'|_{\pi \tau}]|_{\Omega_X}$$

$$= \pi(S'|_{\alpha_t}, t)|_{\omega_t \cap \Omega_X} \cup S'|_{\pi \tau \cap \Omega_X}$$  \hfill (14)

Since $t \in Y$, $\omega_t \subseteq \Omega_Y \supseteq A_Y \Rightarrow \omega_t \cap \Omega_X = \emptyset$.

Therefore Equation 14 simplifies to:

$$\Pi(S, T'\|(t))|_{\Omega_X} = \Pi(S, T')|_{\Omega_X}$$  \hfill (15)

$T'$ now has one element less than $T$, while $X \subseteq T' \subset T$ stays invariant.

The next step is to show that we can swap two disjoint transactions that are in the middle of the $T$ sequence. This uses Theorem 2 regarding the flexibility of evaluation order on disjoint transactions. Assume $a \in X$ and $b \in Y$, and $T = T'\|(b, a)\|T''$. We consider the middle part $(b, a)$ of the transaction sequence $T$.

$$\Pi(S, T'\|(b, a)\|T'')|_{\Omega_X}$$

1. prefix    2. middle   3. suffix

$$= \Pi(\Pi(S, T'\|(b, a), \quad T''))|_{\Omega_X}$$

Since $X$ and $Y$ are disjoint sequences, all individual transactions across the two sequences are disjoint as well. This means we can freely swap $a$ and $b$ using Theorem 2:

$$\Pi(S, T'\|(b, a)\|T'')|_{\Omega_X} = \Pi(S, \Pi(S, T'\|(a, b), T''))|_{\Omega_X}$$  \hfill (16)

We can always remove the last element from $T$ if it is not in $X$ and we can always move the rightmost $t \in Y$ to the last position in the whole sequence. By repeating this process it is possible to remove all elements from $T \setminus X$ until:

$$\Pi(S, T)|_{\Omega_X} = \Pi(S, X)|_{\Omega_X}$$  \hfill (17)

Similarly, we can apply the same process to $Y$ resulting in Equation 13.

Note that we cannot swap dependent transactions, meaning we cannot change the order of transactions within the sequences $X$ or $Y$. So far, we have only shown that the two subsequences can be evaluated independently, i.e., if $X = \{x_1, \ldots, x_k\}, Y = \{y_1, \ldots, y_l\}$, both subsequences from $T$ so that $\forall t \in T : t \in X \lor t \in Y$, then we know that $\Pi(S, T) = \Pi(S, X||Y) = \Pi(S, Y||X)$. The following theorem demonstrates that interleaving disjoint sequences of transactions does not change the result.

**Theorem 4:** If $X, Y \subseteq T \in T^*$, then any interleaving of the two does not change the result.

**Proof:** Starting with Equation 16, we can remove the restriction and expand the evaluation to the whole of $T$ for $(a, b) \subseteq T$ and $(a \in X \land b \in Y) \lor (a \in Y \land b \in X)$:

$$\Pi(S, T'\|(b, a)\|T'')$$

$$= \Pi(\Pi(S, T'\|(b, a), T'')) \quad \text{(apply Equation 11)}$$

$$= \Pi(\Pi(S, T'\|(a, b), T''))$$  \hfill (19)

Through repeated swaps of disjoint transactions in $T$ we can generate all valid interleavings of disjoint sequences $X$ and $Y = T \setminus X$ while the final state stays invariant.

If—and only if—$X \setminus T$ is disjoint from $T \setminus X$, then $A(S, T, X) = A(S, X, X)$ and we can use shorthand $A(S, X)$, with the same requirement applying for the use of $\Omega(S, X)$ as a shorthand.

### E. IMMUTABILITY OF SUBSET STATE FOR DISJOINT SEQUENCES

There is now one final property we need to demonstrate. We need to show that the resulting subset state of any evaluation of a disjoint sequence $X$ containing a specific address
a ∈ Ω(S, T, X) is immutable. By immutable (or immutability) we mean that the evaluation of any other disjoint sequences from the set of transactions not included in X will not change the subset state of Ω(S, T, X) accounts.

Theorem 5: \( \Pi(S, X)|_{\Omega(S, T, X)} = \Pi(S, T')|_{\Omega(S, T, X)} \) for disjoint sequences \( X \subseteq T \) and \( T' \), where \( T' \) is any combination of valid disjoint sequences from \( T \) which contain the sequence \( X \).

Proof: Assume \( Y = T \setminus X \) is split into mutually disjoint subsequences \( Y^* \) that retain their disjointness from \( X \). At minimum \( Y^* = \{ Y \} \), and \( Y \) may have been originally empty as well.

We want to show that we can take any combination of sequences from \( Y^* \), and include them in any valid interleaving with \( X \) while the subset of states \( \Omega_X = \Omega(S, T, X) \) stays invariant. Therefore the goal is to prove that:

\[
\forall Y' \in \rho(Y^*): \Pi(S, X)|_{\Omega_X} = \Pi((\Pi(S, X)|_{\Omega_X} \cup S)|_{\Omega_X}, Y')|_{\Omega_X} \quad (20)
\]

Let us rewrite the rightmost part with \( S' = \Pi(S, X)|_{\Omega_X} \cup S|_{\Omega_X} \) and then expand the evaluation of \( Y' \) with short-hands \( A_{Y'} = A(S, T, Y') \) and \( \Omega_{Y'} = \Omega(S, T, Y') \):

\[
\Pi(S', Y')|_{\Omega_X} = \left[ \Pi(S'|_{A_{Y'}}, Y')|_{A_{Y'}, \Omega_{Y'}} \cup S'|_{A_{Y'} \cap \Omega_{Y'}} \right]|_{\Omega_X}
\]

\[
= \Pi(S'|_{A_{Y'}, \Omega_{Y'}})|_{\Omega_X \cap \Omega_{Y'}} \cup S'|_{\Omega_X \cap \Omega_{Y'}}
\]

\[
= S'|_{\Omega_X} \quad (21)
\]

Now we substitute \( S' \) back into the equation:

\[
\Pi(S', Y')|_{\Omega_X} = S'|_{\Omega_X}
\]

\[
= \left[ \Pi(S, X)|_{\Omega_X} \cup S|_{\Omega_X} \right]|_{\Omega_X}
\]

\[
= \Pi(S, X)|_{\Omega_X \cap \Omega_X} \cup S|_{\Omega_X \cap \Omega_X}
\]

\[
= \Pi(S, X)|_{\Omega_X} \quad (22)
\]

This demonstrates that evaluation of \( Y' \) does not change the subset state \( \Omega_X \). By employing our earlier results, we can show that we can generate all valid interleavings between \( X \) and \( Y' \) while retaining the invariance of the subset state of accounts in \( \Omega_X \). Therefore Equation 20 holds, and extends to all valid interleavings as well.

The usefulness of the last proof comes from the fact that someone may, at some point in time, evaluate only some of the disjoint sequences in a block’s transactions and store the updated state. If they later need to evaluate any other disjoint sequence from the same block—to retrieve state of an account not part of the \( \Omega \) set of the previously evaluated sequences, for example—then the original state subset will not be modified. Consequently, any decisions made on the previous subset state are equally valid—as long as they originally referenced only the subset state, of course.

F. PARTIALLY DISJOINT TRANSACTION SEQUENCES

Definition 12: Partially disjoint transaction sequences \( X \) and \( Y \) (from \( T \)) are such that \( X \) can be evaluated independent of \( Y \) (\( A_X \cap \Omega_Y = \emptyset \)), but \( Y \) is dependent on \( X \): \( A_Y \cap \Omega_X \neq \emptyset \).

This definition differs from Definition 11—where sequences are mutually disjoint—in that only sequence \( X \) is sequentially independent from \( Y \), but not vice versa. Because of this, the following hold:

\[
\begin{align*}
\Pi(S, T)|_{\Omega_X} &= \Pi(S, X)|_{\Omega_X} \\
\exists T': \Pi(S, T)|_{\Omega_Y} &\neq \Pi(S, Y)|_{\Omega_Y}
\end{align*}
\]

The usefulness of partially disjoint transaction sequence is that it is possible to omit some transactions from \( X \) that would have to be included for fully mutually disjoint \( X \) and \( Y \). If we want to maintain only a subset of state of the whole blockchain then we are interested in evaluating only those transactions that modify that subset—we have no interest in irrelevant transactions that cannot modify that subset—even if they have some addresses from the subset as inputs.

For example, in Figure 8 transaction sequences \( (a, b) \) and \( (c) \) are disjoint as they share no input and output addresses. However, \( (a) \) and \( (b) \) are partially disjoint with regards to address 1 whose state can be deduced for the full sequence \( (a, b, c) \) by just evaluating transaction \( a \). The correct state of address 3 cannot however be determined from \( b \) alone, as \( a \) is required to calculate the correct input for transaction \( b \).

G. CONSTRUCTING DISJOINT AND PARTIALLY DISJOINT SEQUENCES

While we have been discussing disjoint and partially disjoint sequences, no constructive method has been defined. Here we address that gap and provide methods for constructing such sequences.

Definition 13 (Transitive Inputs/Outputs): \( \alpha^*: S \times T^* \times T \) is the transitive input dependencies of a transaction in a sequence of transactions, forming the set of input addresses that a transaction may depend either directly, or indirectly via earlier transaction that produces output that the transaction or its previous dependencies uses. \( \omega^*: S \times T^* \times T \) is respectively the transitive output addresses of a transaction.
For example, looking at Figure 8, with three transactions \((a, b, c)\), the input set of transaction \(b\) is \(\omega_b = \{1, 3\}\). However since it depends on state of address 1 which is modified by transaction \(a\), the transitive input dependency includes address 2 as well: \(\omega'_b = \omega^*_{(S, (a, b, c), b)} = \{1, 2, 3\}\). In similar manner, transaction \(a\) modifies only a single address \(\omega_a = \{1\}\), but via another transaction \(b\) it can have impact on state of address 3, thus \(\omega'_a = \{1, 3\}\).

Assuming we have a set of addresses we are interested in \(I \subseteq A\), and we are strictly interested in the subset state \(S_j\). How do we determine the set of transactions that must be included from a sequence \(T\) that define a disjoint partition \(X\) and \(Y\) where \(\forall a \in I: a \notin \Psi(S, T, Y)\), i.e., all transactions that refer to addresses in \(I\) are in the \(X\) set.

**Definition 14 (Constructing a Disjoint Sequence):** \(\Psi: S \times \mathcal{T}^* \times \varphi(A) \to \mathcal{T}^*\) partitions the transactions into two disjoint sequences ensuring that one of them contains all transactions that are dependent on a given set of addresses.

\[
\Psi(S, T, I) = \{t \mid t \in T : (\alpha^*(S, T, t) \cap I) \neq \emptyset\}
\]  
(24)

Since all transactions—and subsequently, all transaction sequences—have their outputs as subsets of inputs, the sequence \(T \backslash \Psi(S, T, I)\) will out of necessity lack any transactions which have any of \(I\) as a transitive output, and consequently the input set of \(\Psi(S, T, I)\) will be disjoint from rest of the transactions. This definition can also be used to determine all disjoint sequences from a sequence of transactions \(T\). This is trivially accomplished by picking an address from the set of untouched transactions, and using \(\Psi\) to pick out the disjoint sequence that contains that address, until all transactions have been identified as part of some disjoint sequence.

**Definition 15 (Constructing a Partially Disjoint Sequence):** \(\psi: S \times \mathcal{T}^* \times \varphi(A) \to \mathcal{T}^*\) identifies the partially disjoint sequence of transactions that modifies a given set of addresses.

\[
\psi(S, T, I) = \{t \mid t \in T : (\omega^*(S, T, t) \cap I) \neq \emptyset\}
\]  
(25)

Unlike \(\Psi\), \(\psi\) may omit transactions that do not modify an address in \(I\) either directly or potentially indirectly. For example, in Figure 8, \(\psi_1 = \{a, b\}\) but \(\psi_1 = \{a\}\) as while transaction \(b\) depends on transaction \(a\), only transaction \(a\) modifies state of address 1. \(\Psi\) and \(\psi\) differ in their definition only whether they look at the transitive input dependencies \(\alpha^*\) or the transitive output dependencies \(\omega^*\).

**REFERENCES**


S. Paavolainen et al.: Adventures of Light Blockchain Protocol in Forest of Transactions: Subset of Story


**SANTERI PAAVOLAINEN** (Member, IEEE) received the B.Sc. degree in computer science from the University of Helsinki, Helsinki, Finland, in 2016, and the M.Sc. degree in engineering physics from the School of Science, Aalto University, Finland, in 2019, where he is currently pursuing the Ph.D. degree with the School of Electrical Engineering. He enrolled to the University of Helsinki, in 1993, but soon after got sidetracked and worked in the software industry for over two decades. During this time, he has worked extensively on scalable internet service design, cloud computing, and technology strategy development. Eventually, he heard the siren song of academia. His research interests include the IoT devices and distributed ledger (i.e., blockchain) integration.

**CHRISTOPHER CARR** received the master’s degree in cryptography and mathematics from Royal Holloway, University of London, in 2012, and the Ph.D. degree from the Norwegian University of Science and Technology, in 2018. His thesis was selected for an outstanding doctoral thesis award. During his time as a Ph.D. candidate, he took part in a televised research dissemination competition, was awarded a scholarship at the Queensland University of Technology and won a championship for a blockchain startup project that was subsequently established into its own company. He works at the University of the West of England, Bristol, where he is currently a Research and Teaching Fellow. His research interests include distributed ledger technology, amongst other areas.

**ESSAM GHADAFI** received the B.Sc. degree in computer science from the University of Tripoli, in 1998, and the M.Sc. degree (Hons.) in advanced computing and the Ph.D. degree in cryptography and information security from the University of Bristol, in 2008 and 2012, respectively. He worked as a Postdoctoral Researcher with the University of Bristol and University College London. He has also extensive industrial experience through working in industry for eight years. He is currently a Senior Lecturer in computer science with the University of the West of England, Bristol, U.K. His research interests span over various aspects of cyber security. He has supervised dozens of students, both at the undergraduate and postgraduate levels.