informatics *mathematics*

Ínría

Dietcoin: shortcutting the Bitcoin verification process for your smartphone

Davide Frey, Marc X. Makkes, Pierre-Louis Roman, François Taïani, Spyros Voulgaris

Dietcoin: shortcutting the Bitcoin verification process for your smartphone

Davide Frey[*], Marc X. Makkes[†], Pierre-Louis Roman[*],
François Taïani[*], Spyros Voulgaris[‡]

Abstract: Blockchains have a storage scalability issue. Their size is not bounded and they grow indefinitely as time passes. As of August 2017, the Bitcoin blockchain is about 120 GiB big while it was only 75 GiB in August 2016. To benefit from Bitcoin full security model, a bootstrapping node has to download and verify the entirety of the 120 GiB. This poses a challenge for low-resource devices such as smartphones. Thankfully, an alternative exists for such devices which consists of downloading and verifying just the header of each block. This partial block verification enables devices to reduce their bandwidth requirements from 120 GiB to 35 MiB.

However, this drastic decrease comes with a safety cost implied by a partial block verification. In this work, we enable low-resource devices to fully verify subchains of blocks without having to pay the onerous price of a full chain download and verification; a few additional MiB of bandwidth suffice. To do so, we propose the design of diet nodes that can *securely* query full nodes for shards of the UTXO set, which is needed to perform full block verification and can otherwise only be built by sequentially parsing the chain.

Key-words: blockchain, sharding, UTXO, distributed ledger, cryptocurrency, mobile computing

[*] Univ Rennes, Inria, CNRS, IRISA, France
[†] Vrije Universiteit Amsterdam, The Netherlands
[‡] Athens University of Economics and Business, Greece

Dietcoin: court-circuiter la vérification dans Bitcoin pour les téléphones mobiles

Résumé :

Les blockchains telles que Bitcoin passent mal à l'échelle, notamment du fait de leurs besoins important de stockage. Les besoins de stockage d'une blockchain typique ne sont pas limités et croissent indéfiniment. Par exemple, en août 2017, les données contenues dans la blockchain Bitcoin représentaient environ 120 GiB, contre 75 GiB un an auparavant, en août 2016. Pour bénéficier des garanties complètes de sécurité apportées par Bitcoin, un nœud qui rejoint le réseau doit télécharger et vérifier l'intégralité des 120 GiB de données. Cette nécessité pose un défi pour les appareils à faibles ressources tels que les smartphones. Heureusement, une alternative existe pour de tels dispositifs qui consiste à télécharger et à vérifier seulement l'en-tête de chaque bloc de la blockchain. Cette vérification partielle des blocs permet aux appareils de réduire leurs besoins en bande passante de 120 GiB à 35 MiB, mais cette diminution drastique ne permet qu'une vérification partielle des blocs, et diminue grandement les garanties de sécurité offertes aux nœuds qui l'utilisent.

Dans ce travail, nous proposons une approche qui permet aux appareils à faibles ressources de vérifier entièrement des sous-chaînes de blocs sans avoir à payer le prix onéreux d'un téléchargement et d'une vérification complète de la chaîne ; quelques MiB supplémentaires de bande passante suffisent. Pour ce faire, nous proposons d'introduire des nœuds Dietcoin qui sont capables *en toute sécurité* d'interroger des nœuds exécutant le protocole complet pour obtenir des fragments d'un ensemble appelé UTXO, nécessaire à la vérification complète des blocs.

Mots-clés : blockchain, partitionnement, UTXO, registre distribué, monnaie cryptographique, informatique mobile

Contents

1 Trustless Bitcoin

Within a decade, blockchains have become extremely popular, and have been used to implement several widely-used crytocurrencies [1], and smart-contract services [2]. A blockchain implements a *tamper-proof distributed ledger* in which public transactions can be recorded in a close-to-irrevocable manner. Recorded transactions are stored into *blocks*, which are then incrementally linked (or *chained*) in order to form an append-only list. The irrevocability of these chaining mechanisms exploits cryptographic mechanisms and peer-to-peer exchanges. This combination makes it in principle inconceivably hard for individual participants to revoke past transactions (due to the computational cost involved), while it remains possible for any participant to verify the validity of a blockchain's entire history.

Verifying a blockchain remains, however, a particularly costly process. The verifying node must first download the entire blockchain, which in many cases has reached a size beyond the communication capabilities of many mobile devices. The Bitcoin blockchain, for instance, had grown to 120 GiB as of August 2017 (Figure 1), and follows an exponential growth, implying the problem can only become more acute.

Once the blockchain has been downloaded, the verifying node must then check its consistency block by block, a lengthy process that can take hours on high-end machines. The exorbitant price of a full chain verification makes it unrealistic for low-resource devices to fully implement a blockchain protocol. Some blockchain systems, such as Bitcoin, therefore enable nodes to perform varying degrees of verification: *full nodes* verify everything while lightweight nodes only verify a small fraction of the data.

In the case of Bitcoin, this lightweight verification is known as *Simplified Payment Verification* (SPV for short). SPV nodes only download and verify a much reduced version of the Bitcoin blockchain, comprised only of its block headers, which today only weights 35 MiB (a reduction
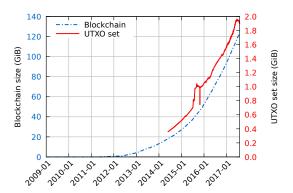
Figure 1: Both the Bitcoin blockchain and the UTXO set have almost tripled in size in the past two years.

by three orders of magnitude). This summary version however only contains the chaining information making up the blockchain, not the recorded transactions. This information is sufficient for SPV nodes to verify that the chain's structure is valid (and hence very unlikely to have been created by malicious nodes), but not that a past transaction does exist in the chain. As a result, SPV nodes are vulnerable to attacks in which an attacker leads an SPV node to believe a transaction t has occurred, while t is later on rejected by the system because the funds transferred by t have in fact already been spent (known as a *double-spend attack*).

To protect themselves against double-spend attacks, full nodes keep track of unspent funds in a structure known as the set of *Unspent TransaCTion Outputs* (UTXO set). The UTXO set is unfortunately costly to construct (as this construction requires the entire blockchain), to exchange (currently weighing 1.9 GiB, see Figure 1), and to maintain, which explains why SPV nodes do not use it.

In this report, we propose to bridge the gap between full nodes and SPV nodes by introducing *diet nodes*, and their associated protocol, *Dietcoin*. Dietcoin strengthens the security guarantees of SPV nodes by bringing them close to those of full nodes. Dietcoin enables low-resource nodes to verify the transactions contained in a block without constructing a full-fledged UTXO set. In our protocol, diet nodes download from full nodes only the parts of the UTXO set they need in order to verify a transaction of interest. This selective download mechanism must, however, be realized with care. Diet nodes must be able to detect any tampering of the UTXO set itself, at a cost that remains affordable for low-resource devices, both in terms of communication and computing overhead.

The rest of this report is structured as follows. We first present the Bitcoin protocol in more detail (Section 2), and explain the workings of full and SPV nodes. We then detail the design of Dietcoin and diet nodes and discuss the security guarantees they provide (Section 3). Finally, we present related work (Section 4), and conclude (Section 5).

2   The Bitcoin system

A blockchain is a *decentralized ledger* composed of blocks containing transactions. The transactions, the blocks, and the resulting chain obey a few core rules that ensure the system remains *tamper-proof*. Great care is required when modifying these rules, as even minor changes might break the blockchain's properties and its security guarantees.
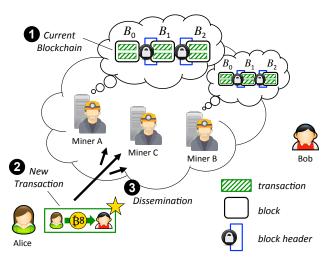
Figure 2: A blockchain is formed of a sequence of blocks containing transactions. The current state of the blockchain (here (B_0, B_1, B_2)) is stored by each individual miner.

In the following, we first detail the default workings of the Bitcoin blockchain and its rationale[1]. We then build upon these explanations to introduce and justify the changes we are proposing.

2.1 Overview

In a blockchain system such as Bitcoin, the blockchain proper $((B_k)_{k\in\mathbb{Z}_{\geq 0}}$, label ❶ in Figure 2) is maintained by a peer-to-peer network of miners. Each block B_k links to the previous block B_{k-1} by including in its header a cryptographic hash that is *(i)* easy to verify, but *(ii)* particularly costly to create (this second point is one of the central element of blockchains with open membership, which we will discuss in detail just below). The leftmost block B_0 is known as the *Genesis Block*: it is the first and oldest block in the blockchain, and it is the only block with no predecessor.

2.1.1 Recording a new transaction

To transfer 8 bitcoins from herself to Bob, the user Alice must first create a valid transaction (label ❷) that contains information proving she actually owns the 8 bitcoins (with a cryptographic signature using asymmetric keys), and encode the resulting transaction output with Bob's public key (such that, in turn, only Bob will be able to demonstrate ownership of the transaction's output).

Alice then broadcasts this new transaction to the network of miners ❸, in order for it to be included in the blockchain. Before adding Alice's transaction into the blockchain, Miner A first verifies that the transaction is valid (label ❹ in Figure 3, details on the transaction verification process will follow in Section 2.2.1-(**BV2**)).

Miner A then includes Alice's transaction together with other transactions received in parallel into a new block $(B_3,$ ❺), and attempts to link it to the current tip of the blockchain. This linkage operation requires Miner A to solve a probabilistically difficult cryptopuzzle ❻ that regulates the frequency at which blocks are created (or *mined*) by the whole network. (In Bitcoin, this periodicity is set to one block every 10 minutes.) If Miner A succeeds, the new block B_3 is now

---

[1]Blockchains with closed membership or different consensus protocols are not discussed in this section
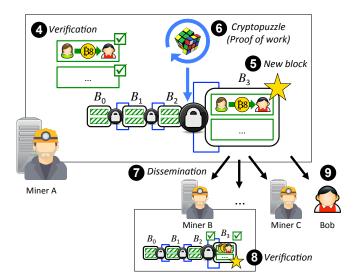
Figure 3: To add a new transaction to the current blockchain, a miner first verifies the validity of the transaction. It must then solve a costly cryptopuzzle to encapsulate this transaction in a new block (here B_3), before disseminating this block to other miners.

linked to the existing block chain (B_0, B_1, B_2), and is disseminated to the other miners ❼. When a miner receives a new block (here Miner B), it checks that the transactions included are valid, along with the cryptopuzzle ❽, before including the new block in its local copy of the blockchain. The new block ultimately reaches Bob ❾, who can check that the transaction has been properly recorded (and can then, for example, sell some goods to Alice).

2.1.2   Irrevocability of deep blocks

Because blocks are produced at a limited rate such that all the miners receive block B_k before they can successfully mine a concurrent block $B_{k'}$, honest miners are highly likely to extend the chain when producing a new block, ensuring a consistent system state with high probability. The views of individual miners may however diverge in problematic cases, causing branches to appear. When a branch occurs, miners resolve the divergence by choosing as valid branch the one that was the most difficult to create (details on block difficulty will follow in Section 2.2.1-(**BV2**)). Blocks that are left out of the chain are said to be *orphan*.

The risk of being made orphan decreases exponentially as a block lies deeper in a chain, ensuring the *practical irrevocability* of deep blocks and the transactions they contain. This is illustrated in Figure 4: consider an attacker who wishes to revoke a block B_{n-k} (targeted block), that lies k blocks away from the chain's tip B_n. For this attack to succeed, this attacker must produce an alternative subchain $(B'_{n-k}, .., B'_n, B'_{n+1})$ that is more difficult to create than the current chain. Producing this subchain is however extremely costly, and takes time which increases the odds that the legitimate chain grows (with a block B_{n+1}, thus requesting an even more difficult attack subchain) before the attacker succeeds. When the computing power of the attacker is less than half of that of the rest of the network, his probability of success drops exponentially with k.
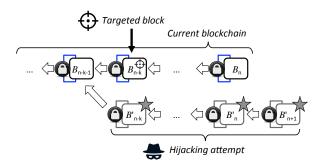
Figure 4: Revoking the content of a block B_{n-k} deep in the chain requires constructing a better alternative subchain, which becomes exponentially harder as the block lies deeper.

2.2   Transactions, Blocks, and UTXO set

To benefit from the full security of Bitcoin, Bob should verify the validity of the new block that contains Alice's payment to him (label ❾ in Figure 3) in addition to verifying the validity of Alice's transaction. This is because Alice could collude with a miner (or launch herself a miner), and produce an invalid block that she would advertise to Bob. Bitcoin relies on a number of built-in validity checks on blocks and transactions to conduct this verification. However, whereas *full nodes* exploit all of these checks, *Simple Payment Verification nodes* (SPV nodes) only perform a limited verification. In the following, we describe the details of these validity checks, we discuss the role of an intermediary set known as the set of *Unspent TransaCTion Outputs* (UTXO set), and the shortcomings of SPV nodes ensued by the limited verification they perform.

2.2.1   Checking block validity

A block is valid if and only if it meets the following two conditions.

- **(BV1)** Its header respects the blockchain's *Proof-of-Work predicate*.

- **(BV2)** It only contains valid transactions (which we discuss further below).

BV1: The Proof-of-Work predicate makes it very difficult for malicious actors to alter the blockchain in an attempt to edit the ledger. The Proof-of-Work predicate is used as a lock-in mechanism to anchor blocks in the chain. It is enforced on each block header, whose simplified structure is shown in Figure 5. The header of each block B_k points both to the header of the previous block B_{k-1} (using a hash function, ❶), and to the transactions contained in the current block B_k ❷. To fulfill the Proof-of-Work predicate ❹, a header must contain a *nonce* ❸ such that the hash of the header is less than a *difficulty target*. The difficulty target is set so that a new block is created every ten minutes by the miners as a whole, regardless of the computation power (the difficulty target is regularly adjusted to cope with changes in their computation power). Finding a nonce respecting the difficulty target is computationally very expensive, as every miner competes to create blocks. This computing cost prevents attackers from easily tampering the chain as they have to recompute fresh nonces for the blocks they wish to replace.

To establish a secure and verifiable link between the header of B_k and the corresponding block B_k, the pointer to B_k's transactions ❷ consists of the root of a Merkle tree. A Merkle tree is a hierarchical hashing mechanism for sets that enables a verifier to efficiently test whether an item (here a transaction) belongs to the set by reconstructing the root of the Merkle tree. Each leaf node in a Merkle tree consists of the hash of an item, while each internal node (including
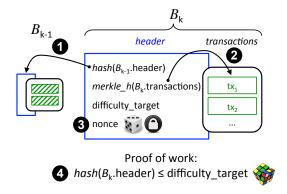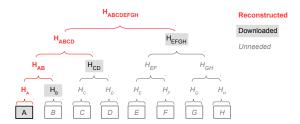
Figure 5: Content of a block header (simplified).

Figure 6: Example of a Merkle tree root reconstruction that only needs $log(n)$ hashes.

the root) consists of the hash of its children. This makes it possible to reconstruct the root, and thus verify set membership, using only a logarithmic number of intermediate hashes. In Figure 6 for example, a node can verify the presence of transaction A in the set by *(i)* downloading the root from a secured communication channel (e.g., the blockchain), and *(ii)* downloading A and the three intermediate hashes shown in red: H_B, H_{CD} and H_{EFGH}, and reconstructing the root from the downloaded hashes. The reconstructed root should match the downloaded one.

BV2: In addition to the Proof-of-Work predicate (**BV1**), all the transactions included in a block must also be valid for the overall block to be valid. Figure 7 shows the validity mechanisms included in a typical Bitcoin transaction. In this example, Alice uses 3 coins she owns (the transaction's inputs ❶) to pay 7 Bitcoins to Bob, and 4 to Tux (the transaction's outputs ❷).

Only coins created in earlier transactions may be spent: each of Alice's inputs therefore points back to the output of an earlier transaction ❸. To ensure that only the recipients (Bob and Tux) are able to spend the output, each new coin contains an ownership challenge (a hashed public key), that must be solved to spend this coin ❹.

Alice's transaction is only valid if the following three conditions are met:

- **(TV1)** The inputs do exist, and Alice owns them. She can prove her ownership of the inputs by providing a public key matching their ownership challenges ❺, and by signing the new transaction with the corresponding private key[2] ❻ ;

- **(TV2)** No money is created in the transaction. In effect, the total value of the transaction's

---

[2]Bitcoin uses a scripting language to encode challenges and proofs of ownership, enabling for more complex schemes, but for ease of exposition we limit ourselves to the typical case.
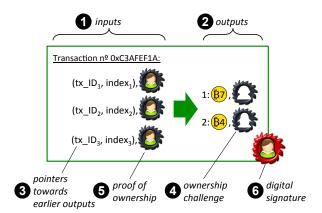
Figure 7: Structure of a transaction.

inputs must be greater than or equal to that of its outputs:

$$\sum_{in \in \mathsf{inputs}(t)} \mathsf{value}(in) \geq \sum_{out \in \mathsf{outputs}(t)} \mathsf{value}(out).$$

The difference $\sum \mathsf{value}(in) - \sum \mathsf{value}(out)$ is given as a fee to the miner of the block containing the transaction;

- **(TV3)** The transaction's inputs ($\mathsf{tx\_ID}_i, \mathsf{index}_j$) have not been spent yet (i.e., they do not appear as inputs of any earlier transaction, an attack known as a *double spend*).

2.2.2 The set of Unspent TransaCTion Outputs (UTXO set)

While the validity of a block's header (**BV1**) only requires access to the current block B_k, and to the header of its predecessor B_{k-1}, verifying transactions (**BV2**) requires a lot more information. Verifying the ownership challenges of input coins (**TV1**), and their amount (**TV2**) requires access to the transactions recorded in earlier blocks. Worse, verifying that inputs coins have not yet been spent (**TV3**) potentially requires parsing and verifying the entire blockchain.

To avoid performing such a costly operation for each new block, nodes that verify transactions maintain an intermediary set known as the set of *Unspent TransaCTion Outputs* (UTXO set). The UTXO set contains all the coins that have been created in the chain but not spent in later transactions, it thus contains all the spendable coins.

A node verifying transactions can prevent double spends (**TV3**) by simply ensuring that all the inputs of a transaction appears in its UTXO set. The UTXO set evolves as new correct blocks are added to the chain: transaction outputs are removed from the set when they are spent, and outputs of new transaction are added to the set.

2.3 The limitations of SPV nodes

In spite of its benefits, constructing a local UTXO set is costly: in order to obtain the set, a node must first download the entire chain (120 GiB as of August 2017, see Figure 1 and validate it (a lengthy process that can take hours on high-end machines), even if only the latest block is relevant to its interest.

Because of this cost, Bitcoin supports several levels of verification. *Miners* and users running *full nodes* construct the UTXO set and check both block headers (**BV1**) and transactions (**TV1,2,3** and as a result **BV2**). By performing all the possible checks, full nodes benefit from the maximum security that the Bitcoin system has to offer.

By contrast, *Simple Payment Verification* nodes (SPV nodes) do not construct the UTXO set. Instead, they only download the chain's block headers (rather than full blocks), and verify that these headers are valid (**BV1**). With the headers only, SPV nodes are able to verify the well-formedness of the blockchain including the crucial Proof-of-Work predicate that seals the links of the chain.

However, because this verification is only partial, SPV nodes are unable to detect if a new block contains an invalid transaction. This scenario, however probabilistically difficult to accomplish for an attacker, is a vulnerability of SPV nodes. To circumvent this vulnerability, SPV nodes typically wait until miners have created subsequent blocks extending the chain containing a block of interest, which implies that these miners have performed a full verification on it and consider this block as valid.

The need for SPV nodes to wait makes it particularly problematic to use Bitcoin on limited devices (i.e. mobile phones) for everyday transactions. SPV nodes are not even able to check whether the inputs used in a transaction do exist. It also limits the ability of SPV nodes to detect faulty transactions as early as possible, which is an important usability feature of modern payment systems.

In this work, we propose to overcome the inherent limitations of SPV nodes with *Dietcoin*. Dietcoin enables nodes with limited resources (*diet nodes*) to benefit from a security level that is close to that of full nodes, at a fraction of the cost required to run full security checks.

3   The Dietcoin system

To address the vulnerabilities of SPV nodes and to improve the confidence mobile users can have in recent transactions, we propose Dietcoin, an extension to Bitcoin-like blockchains. Although our proposal can be applied to most existing Proof-of-Work blockchains using the UTXO model for coins, we describe Dietcoin in the context of the Bitcoin system as presented in Section 2.

The core of Dietcoin consists of a novel class of nodes, called *diet nodes*, which provide low-power devices with the ability to perform *full block verification* with minimal bandwidth and storage requirements. Instead of having to download and process the entire blockchain to build their own copy of the UTXO set, diet nodes query the UTXO set of full nodes and use it to verify the legitimacy of the transactions they are interested in (as described in Section 2.2.1-(**BV2**)) and the correctness of the blocks that contain them. This gives diet nodes security properties that sit in between those of full nodes, and those of Bitcoin's SPV nodes.

Consider a user wishing to verify a transaction for the sale of some goods. The user's diet node will initially proceed like a standard SPV node. It will contact a full node to obtain the header of the block that supposedly contains its transaction as well as the corresponding branch of the transaction Merkle tree, to verify that the transaction indeed is included in the block. But while an SPV node would stop at this inclusion check, the diet node continues by verifying both the inclusion and the correctness of all the transactions in the block. To make this possible we introduce the possibility for diet nodes to access the state of the UTXO set of full nodes corresponding to the instant right before the block they want to verify.

Since downloading the entire UTXO set would result in prohibitive bandwidth overhead, as shown in Figure 1, Dietcoin-enabled full nodes split their UTXO set into small shards, enabling diet nodes to download only the shards that are relevant to the transactions in the block.

To prevent diet nodes from trusting maliciously forged shards, the shard hashes are used as leaves of a Merkle tree, which root is stored in each block. Having the Merkle root stored in blocks enables the UTXO shards to benefit from the same Proof-of-Work protection as transactions.

In addition to the full verification done on the block of their interest B_k, diet nodes can increase their trust in B_k by fully verifying its previous blocks. By doing so, diet nodes are ensured that none of the l verified blocks contain illegal transactions nor erroneous UTXO Merkle root. To make a diet node trust a forged transaction in block B_k, an attacker has to counterfeit the subchain of $l + 1$ blocks $(B_{k-l}, ..., B_k)$. Thanks to the Proof-of-Work protection, the cost of this attack increases exponentially as l increases linearly.

In the following, we detail the operation of Dietcoin by first describing how full nodes can provide diet nodes with verifiable UTXO shards. Secondly we discuss how miners link blocks to the state of their UTXO set. We then explain how diet nodes can extend the verification process from one block to a subchain of any length. Finally, we detail the operation of diet nodes when verifying transactions.

3.1 Sharding the UTXO set

To enable the operation of diet nodes, Dietcoin-enabled full nodes need *(i)* to provide diet nodes with shards of the UTXO set, while *(ii)* enabling them to verify that these shards are authentic.

To satisfy *(i)*, Dietcoin-enabled full nodes store the UTXO set resulting from the application of the transactions in each block in the form of shards with a predefined maximum size of 1 KiB (on average, across all shards). The use of shards enables diet nodes to download only the relevant parts of the UTXO set and also limits the storage requirements at full nodes, which only need to store the modified shards for each block to let diet nodes query older versions of the shards. Similarly, the limit of 1 KiB for the size of each shard limits the bandwidth employed by diet nodes in the verification process.

To satisfy *(ii)*, full nodes also maintain a Merkle tree that indexes all the shards of the UTXO set. Using shards also proves advantageous with respect to this Merkle tree. If nodes were to index UTXO entries directly, the continuous changes in the UTXO set would cause the Merkle tree to become quickly unbalanced, leading to performance problems or requiring a potentially costly self-balancing tree. The use of shards, combined with the right sharding strategy, gives the UTXO Merkle tree a relatively constant structure, enabling shards to be updated in place most of the time. Moreover, it makes it possible to predict the size of the UTXO Merkle tree and its incurred overhead, which enables us to better control and balance the storage overhead for full nodes and the bandwidth requirements of diet nodes.

A number of sharding strategies satisfy the requirement of a fixed number of shards. In this work, we use the simplest approach consisting of indexing UTXO entries by their first k bits. This strategy resembles a random approach since the first bits of an UTXO are the transaction hash it references, which value is expected to be random due to the uniformity property of the SHA-256 hash function. This strategy comes with the added advantages of obtaining shards of homogeneous size and a full binary Merkle tree with 2^k leaves.

Keeping in mind the size cap of 1 KiB per shard, k can be adapted locally by each node to cope with the growth of the UTXO set. When the average shard size breaches the cap of 1 KiB, k is incremented by one resulting in *(i)* halving the average shard size, and *(ii)* adding one layer to the Merkle tree, doubling its storage footprint in the process.
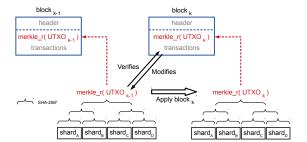
Figure 8: The UTXO set is updated every time a block is validated. For a counterfeited block to be validated by diet nodes, a malicious node has to forge at least two consecutive blocks: the first block B_{k-1} containing a fake Merkle root of $UTXO_{k-1}$, and the second block B_k spending fake coins validated by the fake $UTXO_{k-1}$.

3.2 Linking blocks with the UTXO set

With reference to Figure 8, consider a Dietcoin-enabled miner that is mining block B_k, and let $UTXO_k$ be the state of the UTXO set after applying all the transactions in B_k. The miner stores the root of the UTXO Merkle tree associated with $UTXO_k$ as an unspendable output in the first transaction of block B_k before trying to solve the Proof-of-Work as shown in Figure 8. Storing the root of the UTXO Merkle tree in the first transaction of the block does not require any modification to the structure of Bitcoin's blocks. Thus it is possible for Dietcoin-enabled full nodes and miners to co-exist with their legacy Bitcoin counterparts.

Dietcoin-enabled full nodes verify the value of the UTXO Merkle root against their own local copy when they verify a block, while legacy Bitcoin nodes simply ignore it.

The UTXO Merkle tree provides a computationally efficient way for diet nodes to verify whether the shards they download during the verification process are legitimate and correspond to the current state of the ledger. Still referring to Figure 8, let us consider a diet node d that wishes to verify a transaction in block B_k. Node d needs to obtain: block B_k, the UTXO Merkle root in block B_{k-1}, the shards of the UTXO set between the two blocks, and the elements of the associated Merkle tree that are required to verify their legitimacy. It can then verify the shards using the root stored in block B_{k-1}'s first transaction, and use them to verify the correctness of the transactions in B_k.

We observe that storing the Merkle-root referring to the state after the block into the block itself forces diet nodes to download two blocks to verify transactions. This makes it inherently harder for an attacker to provide a diet node with a fake block B_k because it would need to forge not only block B_k but also block B_{k-1}.

3.3 Extended verification

Diet nodes have the ability to extend their confidence in a block by iterating the verification process towards its previous blocks. By doing so, diet nodes ensure the correctness of the UTXO Merkle root present in block B_{k-1} used to verify the correctness of block B_k. The extended verification can be performed on a subchain of any length l provided that the verifying diet node can query UTXO shards of any age. A diet node fully verifying the subchain $(B_{k-l+1}, ..., B_k)$ can only be tricked into trusting a malicious transaction in block B_k if the attacker manages to counterfeit the $l+1$ successive blocks starting from B_{k-l}, which becomes exponentially more costly as l increases linearly.

The block B_{k-l} contains the UTXO Merkle root that serves as a basis for the verification of the consequent blocks. Since the block B_{k-l} is not verified, it is thus trusted by the diet node. A comparison can be made with full nodes where the first block of the chain, the genesis block, is hard-coded and is therefore trusted. By shifting the trust from the genesis block to the block B_{k-l} for diet nodes, Dietcoin effectively shortcuts the verification process.

Picking the value of l exhibits a trade-off between security and verification costs. On one hand choosing a great l draws the behavior of the diet node closer to that of a full node, while on the other hand choosing a small l draws it closer to that of an SPV node. The user can make her decision based on which block depth l is great enough in her opinion such that all blocks prior to B_{k-l} are unlikely to be counterfeited. For instance, a diet node user can choose l such that the trusted block has a block depth of 6 or greater since it is the de facto standard in Bitcoin to consider blocks of depth 6 or greater as secured[3].

In the case depicted in Figure 8, assuming $l = 2$, diet node d downloads block B_{k-1} to verify the transactions and the UTXO Merkle root in B_{k-1} to further increase its confidence in B_k. To verify B_{k-1}, the diet node uses the UTXO Merkle root in B_{k-2}.

3.4 Detailed Operation

Equipped with the knowledge of Dietcoin's basic mechanisms, we can now detail the verification process carried out by diet nodes. Algorithm 1 depicts the actions taken by a diet node when its user starts the application using Dietcoin and compares it with those taken by legacy SPV clients. Black dotted lines [●] are specific to diet nodes, while hollow dotted lines [○] are common to both diet and SPV nodes.

The algorithm begins when the application using Dietcoin starts and updates its view of the blockchain. In this first part of the algorithm, a diet node behaves exactly in the same manner as an SPV node. First, it issues a query containing the latest known block hash and an obfuscated representation of its own public keys in the form of a bloom filter (lines 4-5). A full node responds to this query by sending a list of all the block headers that are still unknown to the SPV/diet node, together with the transactions matching the SPV/diet node's bloom filter, and the information from the transaction Merkle tree that is needed to confirm their presence in their blocks. Using this information, the SPV/diet node verifies the received headers (including Proof-of-Work verification) and updates its view of the blockchain (line 6).

Since the response from the full node might contain false positives due to the use of a bloom filter for public key obfuscation, the next verification step consists in ensuring that the received transactions match one of the user's public keys (lines 8-9). Once false positives are discarded, the SPV/diet node verifies that the received transactions are in the blocks (line 10-12).

At this point, a standard SPV node simply returns the received transactions to the application (line 14). A diet node, on the other hand, continues the verification process. To this end, the diet node first computes which blocks to fully verify to ensure that *(i)* no block is fully verified twice (line 17), *(ii)* old blocks considered by the user as secured enough are not fully verified (parameter maxDepth$_p$, line 17) and *(iii)* only a subchain of limited length l is fully verified (parameter maxLength$_p$, line 19). If no block is selected for full verification, the diet node falls back to SPV mode (lines 20-21).

To bootstrap the full verification process, the diet node must first download the UTXO Merkle root present in the block prior to the first block to verify (line 23). For each of the selected blocks B_k, the diet node downloads from Dietcoin-enabled full nodes *(i)* the block B_k itself (line 25), *(ii)* the state, before B_k, of the UTXO shards associated with both the block's transactions' inputs

---

[3]`https://bitcoin.org/en/developer-guide#verifying-payment`

and outputs (line 26), and *(iii)* the partial UTXO Merkle tree required to prove the integrity of the downloaded shards (line 26).

Once it has all the data, the diet node proceeds with the verification process. For each block B_k, it first verifies that the downloaded UTXO shards match the UTXO Merkle root from the previous block B_{k-1} (lines 27-29). Then it verifies that the transactions in each block use only available inputs from these shards (lines 32-33), and computes the new state of these shards based on the transactions in B_k (lines 34, 36). Finally it verifies that the updated shards lead to the UTXO Merkle root contained in B_k (lines 37-39). Once the verification process has terminated, the diet node returns the transactions associated with the local user to the application (line 14).

4   Related work

Making the UTXO set available for queries between nodes has been discussed several times in the Bitcoin community over the past few years. Bryan Bishop published a comprehensive list of such proposals [3] that share some of the following goals: *(i)* enabling faster node bootstrap, *(ii)* strengthening the security guarantees of lightweight nodes, and *(iii)* scaling the UTXO set to reduce its storage cost. The primary goal of Dietcoin is to strengthen the security of lightweight nodes. Dietcoin's strongest feature is the ability for diet nodes to efficiently perform subchain verification, as described in Section 3.3, which offers stronger security guarantees than the referenced proposals made by the community. Moreover, even though we focus in this report on the security of lightweight nodes, it is also possible to bootstrap full nodes faster with Dietcoin.

Sharing similar goals with Dietcoin, Andrew Miller [4] suggests to store in blocks the root of a self-balancing Merkle tree built on top of the UTXO set. In such a system, lightweight nodes only download the UTXOs they need, which results in a lower bandwidth consumption than with shards as we propose it, but at the cost of a greater storage overhead since the stored Merkle tree is larger. Moreover, Dietcoin combines a full, and thus always balanced, Merkle tree built on top of 2^k shards that can each be updated in place as blocks are appended to the chain. This combination of tree stability and updatable shards enables efficient subchain verification, as described in Section 3.3, and adapting this feature to a system using a self-balancing Merkle tree does not seem trivial.

Vault [5] also proposes to use Merkle trees to securely record the state of the distributed ledger in recent blocks, and shards this state across nodes, to reduce storage costs. Contrarily to Dietcoin, however, Vault targets balance-based schemes, such as introduced by Ethereum, in blockchains relying on Proof-of-Stake consensus. Vault further stores individual accounts in the Merkle trees, rather than UTXO shards as we do. This represents a different and to some extent orthogonal trade-off to that of Dietcoin, in that Vault chooses to increase the size of Merkle tree witnesses that must be included in transactions, but removes the need for lightweight nodes to download UTXO shards.

Whereas we focus on sharding the resulting state of the blockchain, other systems propose to shard the verification process. Both Elastico [6] and OmniLedger [7] proposes a permissionless distributed ledger using multiple classical PBFT consensus protocols each executing within a subset (shard) of nodes.

Elastico limits the number of shards a malicious nodes may join (under different identifies) by tying the shard of a node to the result of a Proof-of-Work puzzle. OmniLedger [7] extends the ideas proposed by Elastico [6] to increase the size of the shards (and thus reduce the probability of failures), and allow for cross-shards transactions thanks to a Byzantine shard-atomic commit protocol called *Atomix*. Both Elastico and OmniLedger use sharding to increase the transaction processing power of a distributed ledger, rather than to improve access and verification of the

UTXO set, as we do.

Chainiac [8] combines the ideas of skiplists and blockchains to realize *skipchains*, an authenticated log with both back and forward long-distance links to implement a distributed authenticated software-release ledger. Chainiac relies on digital collective signatures to implement forward links, which are not available in permissionless Proof-of-Work chains such as Bitcoin. Long distance links are particularly well adapted to navigate a well-identified subset of a blockchain (such as a package's releases). They were not however directly designed to handle the kind of dependencies captured by the UTXO model.

An entirely different approach to scaling blockchains for lightweight nodes is the use of Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) [9] that enable constant size queries. NIPoPoWs strive for minimal cost of proof of inclusion of a transaction in a chain, thus reducing to a minimum the bandwidth requirements of lightweight nodes. NIPoPoWs however do not aim at offering improved security for lightweight nodes as we do with Dietcoin.

5 Conclusion

In this report, we have presented the design of Dietcoin, that proposes a new form of Bitcoin nodes that strengthens the security guarantees of lightweight SPV nodes by bringing them closer to those of full Bitcoin nodes. The Dietcoin protocol enables low-resource nodes to verify the transactions contained in blocks without constructing a full-fledged UTXO set. In our protocol, diet nodes download from full nodes parts of the UTXO set they need in order to verify a block, or a subchain of blocks, of interest. Diet nodes are able to detect any tampering of the UTXO set itself, at a cost that remains affordable for low-resource devices, both in terms of communication and computing overhead. In our approach, Dietcoin-enabled full nodes split their UTXO set into small shards, and enable diet nodes to download only the shards that are relevant to the transactions in the block, while verifying that these shards do indeed corresponds to the state of the UTXO set for the block they are verifying.

Acknowledgments

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: http://www.cryptovest.co.uk/resources/Bitcoin%20paper%20Original.pdf

[2] T. D. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *ACM Symposium on Principles of Distributed Computing, PODC*, 2017, pp. 303–312. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087835

[3] B. Bryan, "[bitcoin-dev] Protocol-Level Pruning," Nov. 2017. [Online]. Available: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-November/015297.html

[4] A. Miller, "Storing UTXOs in a Balanced Merkle Tree (zero-trust nodes with O(1)-storage)," Aug. 2012, https://bitcointalk.org/index.php?topic=101734.0.

[5] D. Leung, A. Suhl, Y. Gilad, and N. Zeldovich, "Vault: Fast bootstrapping for cryptocurrencies," Cryptology ePrint Archive, Report 2018/269, 2018, https://eprint.iacr.org/2018/269.

[6] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A Secure Sharding Protocol For Open Blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16.   New York, NY, USA: ACM, 2016, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978389

[7] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, San Francisco, May 2018, pp. 19–34. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2018.000-5

[8] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds," in *26th USENIX Security Symposium (USENIX Security 17)*.   Vancouver, BC: USENIX Association, 2017, pp. 1271–1287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin

[9] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive proofs of proof-of-work," Tech. Rep. 963, 2017. [Online]. Available: https://eprint.iacr.org/2017/963

bloomFilter(keys)	Compute a bloom filter from keys
buildMRoot(hashes)	Compute the Merkle root from hashes
getShardKey(txId)	Apply the sharding algorithm to txId
height(header)	Index of header starting from the genesis block
updateMTreeInPlace(MTree, dataset)	Update the hashes of MTree with the new value of dataset
verifyHeaders(headers)	Add headers to the chain, return the hash of the new chain tip

---

Algorithm 1 – SPV [○] and Diet [○●] node p (compacted)

1: **parameters:** ○ pubKeys_p, ● maxDepth_p, ● maxLength_p
2: **global variables:** ○ headerStore_p, ○ tipId_p, ● highestVerified_p

○3: **procedure** UPDATECHAIN( )   ▷ *App interface*
○4:   filter ← bloomFilter(pubKeys_p)
○5:   ({header, txMTree, txs}) ← **send** QUERYMERKLEBLOCKS(tipId_p, filter)
○6: tipId_p ← verifyHeaders((header))
○7:   **for all** {header, txMTree, txs} ∈ ({header, txMTree, txs}) **do**
○8:   **if** $\forall k \in \text{pubKeys}_p : k \notin$ {tx.inputs ∪ tx.outputs} **then**
○9:   **continue**   ▷ *Ignore bloom filter false positives*
○10:   **assert**($\forall \text{tx} \in \text{txs} : \text{HASH(tx)} \in \text{txMTree}$)
○11:   builtTxMRoot ← buildMRoot(txMTree)
○12:   **assert**(builtTxMRoot = header.txMRoot)
●13:   VERIFYBLOCKSUPTO(height(header))
○14:   **callback**(txs, header)   ▷ *Callback to app*

●15: **procedure** VERIFYBLOCKSUPTO(last)
●16:   ▷ *Do not verify blocks twice or below* maxDepth_p
●17:   first ← **max**(highestVerified_p, height(tipId_p) − maxDepth_p)
●18:   ▷ *Verify up to* maxLength_p *blocks*
●19:   first ← **max**(first, last − maxLength_p)
●20:   **if** first ≥ last **then**
●21:   **return**   ▷ *Fallback to SPV mode*
●22:   ▷ *The first UTXO Merkle root is not verified*
●23:   utxoMRoot ← **send** QUERYUTXOMROOT(HASH(headerStore_p[first]))
●24:   **for all** blockId of height ∈ [first + 1, last] **do**
●25:   block ← **send** QUERYBLOCK(blockId)
●26:   {shards, utxoMTree} ← **send** QUERYUTXOS(blockId)
●27:   **assert**($\forall \text{shard} \in \text{shards} : \text{HASH(shard)} \in \text{utxoMTree}$)
●28:   builtUtxoMRoot ← buildMRoot(utxoMTree)
●29:   **assert**(builtUtxoMRoot = utxoMRoot)
●30:   **for all** tx ∈ block.transactions **do**
●31:   **for all** i ∈ tx.inputs **do**
●32:   shard ← shards[getShardKey(i)]
●33:   **assert**(i ∈ shard ∧ valid proof of ownership of i)
●34:   shard.remove(i)
●35:   **for all** o ∈ tx.outputs **do**
●36:   shards[getShardKey(o)].add(o)
●37:   utxoMTree ← updateMTreeInPlace(utxoMTree, shards)
●38:   utxoMRoot ← buildMRoot(utxoMTree)
●39:   **assert**(utxoMRoot = block.utxoMRoot)
●40: highestVerified_p ← height(blockId)
