EDRAX: A Cryptocurrency with Stateless Transaction Validation

Alexander Chepurnoy[1], Charalampos Papamanthou[2], and Yupeng Zhang[3]

[1]Ergo Platform and IOHK, alex.chepurnoy@iohk.io
[1]University of Maryland, cpap@umd.edu
[1]UC Berkeley, zhangyp@berkeley.edu

October 14, 2018

Abstract

We present EDRAX, a general architecture for building cryptocurrencies with stateless transaction validation. In EDRAX, all cryptocurrency nodes, such as miners and validating nodes, can validate incoming transactions and subsequently update user balances simply by accessing the last confirmed block. This removes the current need for storing, off-chain and on-disk, order-of-gigabytes large validation state. We present and implement two instantiations of EDRAX, one in the UTXO-based model of Bitcoin-like cryptocurrencies, where we use sparse Merkle trees, and one in the account-based model of Ethereum-like cryptocurrencies, where we show that Merkle trees cannot be used and where algebraic vector commitments are needed instead. Towards this goal, we construct, prove secure and implement the first practical algebraic vector commitment with logarithmic asymptotic costs that can scale to millions of accounts, as required by cryptocurrencies today. Our evaluation of EDRAX shows that (i) for the current scale of Bitcoin and Ethereum our stateless transaction validation overhead is comparable to stateful transaction validation that requires gigabytes of local index data; (ii) while the scale increases, the performance of stateful validation deteriorates substantially due to expensive I/Os and our stateless validation is faster by up to approximately two orders of magnitude.

1 Introduction

Decentralized cryptocurrencies and smart contracts such as Bitcoin [19] and Ethereum [2] promise to remove trusted online parties (e.g., banks and escrows) in sake of faster and more secure financial transactions. Their underlying technology, the blockchain, is an ever-growing hashchain built on blocks of incoming transactions that is agreed upon by a dynamic set of nodes participating in the peer-to-peer cryptocurrency network. This ever-growing nature of the blockchain, however, can limit the cryptocurrency scalability, not only in terms of storage required to include all events since the genesis block, but also in terms increasing overheads for transaction validation, blockchain verification and initial synchronization.

In particular, most blockchain-based cryptocurrencies known to date consist of two kinds of parties, *clients* that own coins (e.g., a secret key to a Bitcoin address) and *nodes*[1] that validate transactions created by the clients. To decide if an incoming transaction is valid so that it can be included in the next block or

[1]Nodes are further distinguished into *miners* that propose new blocks and *validating nodes* that merely validate and propagate transactions and blocks in the network.

propagated to a peer, nodes store all the history of transactions so far—namely *the whole blockchain*. For example, if a new transaction appears requiring 5 bitcoins to be sent from address A to address B, a node must query the blockchain to decide whether A has at least 5 bitcoins in his account. Only if this is the case, is this transaction considered valid and candidate for appearing on the blockchain.

The blockchain data structure, however, is too large (e.g., Bitcoin blockchain is around 150 GB and Ethereum blockchain has exceeded 400 GB) and is growing continuously. Therefore naively querying it will simply take too long. For that reason, most cryptocurrency nodes are typically *stateful*, maintaining an appropriate index called *validation state* that is smaller than the blockchain and which is enough for deciding transaction validity. In some cryptocurencies (e.g., Bitcoin, ZCash, Komodo, Monero, Ergo) the validation state is a set of immutable coins called UTXO (unspent transaction outputs), in Bitcoin jargon. In this *UTXO-based* model, a transaction is valid if it spends coins which belong in UTXO. Other cryptocurrencies (e.g., Nxt, Ethereum, Bitshares, NEM, Tezos) organize the validation state as a set of mutable (and potentially long-living) accounts. In this *account-based* model, a transaction is valid if it is trying to spend no more tokens than the available balance. Advantages and disadvantages of both approaches are the focus of an ongoing debate in the cryptocurrency community [27].[2]

Challenges due to stateful validation. Locally maintaining the validation state, however, is quite cumbersome. In particular, the validation state is in the order of GBs (currently the UTXO set in Bitcoin is around 2.7 GB [16] and the authenticated Patricia trie in Ethereum is around 14 GB) and could grow substantially in the coming years. For example, approximately 86,000 Ethereum new accounts/addresses are currently generated every day [13] and at this rate the Ethereum validation state is expected to double in one year from now. For a new node to enter the network, the validation state needs to be either downloaded and verified or computed from scratch, making such synchronization an extremely slow process [14] (looking ahead, our approach will enable incoming miners to start validating transactions instantly, by just accessing the last block). Also, being in the order of GBs, the validation state is stored on disk (e.g., the `geth` Ethereum implementation stores the authenticated Patricia Trie using Google's levelDB [15]), leading to slow transaction validation due to expensive I/Os. This has facilitated various DoS attacks like the one that took place on Ethereum in 2016 [26], where adversarially-crafted transactions required a large number of disk accesses causing block validation times to reach 60 seconds! Finally, having to store such large state to verify transactions can potentially lead to disadvantaged miners that cannot dedicate large storage resources [7]. Several other practical issues and system-level components (e.g., storage rent and sharding) that would benefit via a complete erasure of the local validation state are analyzed extensively by Vitalik Buterin, Ethereum co-founder, in his recent blog post on the "stateless client concept" [14].

Due to the challenges above, building a cryptocurrency protocol where all nodes can *check the validity of transactions without having to store any local validation state* (namely validation state can be maintained as part of the cryptocurrency blocks on-chain) has been a challenging open problem in the cryptocurrency community.

1.1 EDRAX architecture

We address the above by designing and implementing EDRAX, a cryptocurrency supporting stateless transaction validation using *verifiable data outsourcing* methods such as authenticated data structures [25, 17] and vector commitments [11]. EDRAX comes in two versions, one supporting the UTXO-based model and one for the account-based model. EDRAX's architecture can be seen in Figure 1.

[2]One of the main advantages of the account-based model is the small transaction size. In particular, a public key can spend any amount of funds v associated with it with *a single transaction* whereas in the UTXO-based model a public key distributes its funds in many unspent outputs, meaning that spending an arbitrary amount of funds might require including many inputs in the transaction, thus increasing the transaction size.

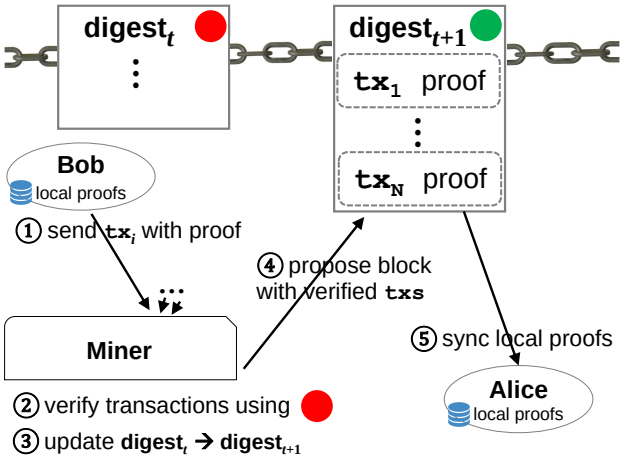

Figure 1: EDRAX architecture. Transactions include proofs. Miners verify transactions using a short digest stored at the last block t, an updated version of which is included in the next block $t+1$. Clients synchronize their local proofs after the new block is produced.

Validation digest, local proofs and transactions. In EDRAX, each block b includes a constant-size *validation digest* of the current validation state (the one that includes all transactions up to block b), computed with appropriate authenticated data structures. Clients, along with their coins, store short *local proofs* of their coins with respect to the aforementioned digest. A local proof is a proof that a specific coin can be spent given the current state of the blockchain (i.e., with respect to a certain validation digest). It is included, along with the traditional digital signature, in an EDRAX transaction, enabling miners and validating nodes to easily verify transactions by just accessing the latest validation digest.

Local proofs and digest synchronization. In EDRAX, Alice's local proof for a coin with respect to a validation digest digest_t at time t will be outdated at time $t+1$, after Bob's transactions are incorporated in the blockchain and digest_t changes to digest_{t+1}. EDRAX's authenticated data structures enable coin owners to synchronize their local proofs efficiently by accessing from the blockchain the updates that took place between t and $t+1$, so that they can spend their coins again at time $t+1$. Similarly, EDRAX allows miners to easily update the digest from digest_t to digest_{t+1} to incorporate new transactions added to the blockchain. The new validation digest will be part of the new block. See Figure 1.

1.2 Warm-up: EDRAX for UTXO-based model via sparse Merkle trees

As warm-up, we show in Section 3 how to provide stateless transaction validation in the UTXO-based model by just using Merkle trees. Recall that in the UTXO-based model, miners and validating nodes are maintaining a set S of unspent transaction outputs. Whenever a new transaction tx appears that has input x and output y, nodes must first check whether input x belongs in S, and if so, update set S by removing x and inserting the new output y. Our construction represents S with a *sparse* Merkle tree of 2^W leaves where 2^W is the maximum number of outputs that can ever be generated, e.g., $W = 40$ (in Section 7 we suggest an optimization with less leaves using authenticated red-black trees). At leaf i we store the i-th transaction output that was inserted into set S. To delete a leaf j, we just set the value of this leaf to be $null$. We then naturally define the validation digest to be the root of the underlying Merkle tree and local proofs as Merkle tree proofs. The above approach allows insertions and deletions to be performed by miners and validating nodes only if the whole Merkle tree is stored as validation state which is very

large. In Section 3 we show how to append a new output y to S by having nodes access only the local proof of the *most recent output* ever inserted in S (of size $O(W)$). Similarly, deletion of Alice's spent input x from S can be performed by having nodes process Alice's local proof used to prove membership of x in S.

Table 1: Comparison of our new vector commitment with existing work. While the lattice-based scheme has better asymptotics, the constants in the proof size are quite large and therefore it is not suited for our application (n is the size of the vector).

scheme	public key	update key	proof	Verify	Prove	UpdateDigest	UpdateProof
ECC-based [11]	$O(n^2)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
RSA-based [11]	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n \log n)$	$O(1)$	$O(n)$
lattice-based [21]	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
EDRAX	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$	$O(\log n)$

1.3 EDRAX for account-based model via algebraic vector commitments

In the account-based model, nodes maintain a vector of all account balances which serves as validation state. Whenever Alice wishes to send δ tokens to Bob, she posts a transaction including this information. To verify this transaction, nodes must access the balances vector to check whether Alice has at least δ tokens in her account.

Why Merkle trees do not work here. One could consider using Merkle trees to provide stateless validation in the account-based model as well: Build a Merkle tree on top of the account balances, define the Merkle digest as the validation digest and have the owner of account i maintain a Merkle proof for the balance at position i as her local proof. However, there is a fundamental problem with such an approach. Assume Alice's balance is v_A tokens and Bob's balance is v_B tokens. Whenever Alice wants to send, say, 5 tokens to Bob, Alice needs to include a Merkle tree proof proving her current balance is $v_A \geq 5$. Once the miner verifies the proof, the miner can process the verified proof to efficiently update the new digest so as to reflect Alice's new account balance as $v_A - 5$. Unfortunately, due to the nature of Merkle trees, Alice's local proof does not suffice to update Bob's new balance to $v_B + 5$ as well unless Alice includes Bob's local proof in her transaction. However, this would require Alice to contact Bob and ask for his proof every time she sends money to him which is against the spirit of cryptocurrencies where Alice should be able to send money to Bob by just knowing a fixed public address. A similar approach was recently introduced by Reyzin et al. [14] where some cryptocurrency nodes must eventually store large state to circumvent the above fundamental problem.

Our approach. Our central observation is that any instantiation of *algebraic vector commitments*, as defined by Catalano and Fiore [11], provides a solution to stateless validation in the account-based model. An algebraic vector commitment is a way to compute a collision-resistant *digest* $dig(\mathbf{v})$ of a vector $\mathbf{v} = [v_1, \ldots, v_n]$ such that updates (of the digest) at arbitrary indices i by an amount δ (namely $v_i = v_i \pm \delta$) can be performed by an algorithm UpdateDigest that accesses only $dig(\mathbf{v})$, index i, difference δ and some fixed public parameters computed in the setup phase of the scheme—namely no proof is needed to update $dig(\mathbf{v})$ as in Merkle trees. Also, one can define a short proof π_i for the value v_i of vector $\mathbf{v}$ with respect to a digest $dig(\mathbf{v})$. This proof can be easily synchronized when an update (j, δ) takes place by just using j's *update key* upk_j. See Definition 1 for the formal definition of algebraic vector commitments.

Algebraic vector commitments are perfect fit for implementing EDRAX in the account-based model: The vector digest $dig(\mathbf{v})$ serves as validation digest; a SPEND transaction is of the form $[\pi_i, v_i, i \to j, \delta]$ meaning a client i owning v_i tokens wants to send $\delta \leq v_i$ tokens to client j; Proof π_i enable miners to

check that $\delta \leq v_i$; Auxiliary information $i \to j, \delta$ allows miners to update $dig(\mathbf{v})$ to reflect $v_i = v_i - \delta$ and $v_j = v_j + \delta$ so as to include it in the next block—it also allows all other clients to synchronize their local proofs π_k accordingly. The detailed protocol is described in Section 4.

1.4 A new algebraic vector commitment

We design and implement a new algebraic vector commitment for EDRAX—see Section 5. Our construction uses the ℓ-variate "multiplexer" polynomial $f(\mathbf{x})$, also called a multilinear extension, to represent a vector of $n = 2^\ell$ entries. E.g., for the vector $V = [5\ 2\ 8\ 3]$ the polynomial $f(x_1, x_2)$ is $5 \cdot (1 - x_2)(1 - x_1) + 2 \cdot (1 - x_2)x_1 + 8 \cdot x_2(1 - x_1) + 3 \cdot x_2x_1$, so that $f(0, 0) = V[0], f(0, 1) = V[1], f(1, 0) = V[2], f(1, 1) = V[3]$. Then the digest of the vector is computed as $g^{f(\mathbf{s})}$ where g is a generator of an elliptic-curve group and $\mathbf{s}$ is a random point that is kept secret. The proof size of our construction is exactly ℓ group elements. Our construction also features an $O(\ell)$-time algorithm[3] for synchronizing proof π_i for a point $i \in \{0, 1\}^\ell$ given un update (j, δ) where j is another point in $\{0, 1\}^\ell$—see Algorithm DELTAPOLYNOMIALS in Section 5.

Comparison to other vector commitments. Succinctly representing vectors using multilinear extensions was introduced by Zhang et al. [29, 28] for a different application, where no efficient proof synchronization algorithms were presented. Their proof was slightly larger ($2\ell - 1$ group elements) due larger domain of possible queries (as opposed to the hypercube $\{0, 1\}^\ell$ that we have here).

Other algebraic vector commitments that can be used to implement EDRAX were also introduced by Catalano and Fiore [11]: One based on elliptic curve cryptography (ECC) and one based on the RSA cryptosystem. Also Papamanthou et al. [21] presented a "streaming authenticated data structure" (that can be cast as an algebraic vector commitment) based on lattice assumptions, which was subsequently optimized and implemented in [24]. Unfortunately all these approaches are quite impractical. In particular, the ECC-based construction [11] has quadratic public key size (leading to hundreds of billions of group elements for public parameters in EDRAX) as well as linear update key size (meaning an otherwise constant-size EDRAX transaction $i \to j$ would suffer a linear blowup). Similarly the RSA-based construction [11] requires linear time for proof update, leading to very slow proof synchronization. Also, while the lattice construction has better asymptotics it suffers from high concrete parameters (as we analyze in detail in our evaluation in Section 6): Numbers from [24] indicate an increase in transaction size by at least two orders of magnitude (the size of a lattice hash is around 2.6 KB). A representative comparison is shown in Table 1.

1.5 Implementation and evaluation

In Section 6 we implement sparse Merkle trees and our new vector commitment scheme and use our implementation to simulate block processing in UTXO-based EDRAX and account-based EDRAX. Our main finding indicates that as the size of the validation state increases, storing validation state on-disk, as it happens today, is bound to considerably slow down block processing, even in the presence of caching. In such cases EDRAX's stateless transaction validation can yield up to 43× savings in block processing times.

2 Preliminaries

We now present background material on bilinear maps, multilinear extensions, sparse Merkle trees and vector commitments.

[3]Moreover, averaging over all $j \in \{0, 1\}^\ell$ the proof update complexity for i is $O(1)$.

Bilinear pairings. We denote by $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{Gen}(1^\lambda)$ generation of bilinear-map parameters, where $\mathbb{G}$, $\mathbb{G}_T$ are groups of prime order p, with g a generator of $\mathbb{G}$, and where $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is an efficient map, i.e., for all $P, Q \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$ it is $e(P^a, Q^b) = e(P, Q)^{ab}$. To prove security we will be using the q-Strong Bilinear Diffie-Hellman assumption[9] (q-SBDH) on the groups $\mathbb{G}$ and $\mathbb{G}_T$ that we formally define in the Appendix—see Assumption 1.

Multilinear extension polynomial of vectors. Let $\mathbb{F}$ be a field (one can think of it as $\mathbb{Z}_p$) and let $n = 2^\ell$. Let $i \in \{0, \ldots, n-1\}$ and let i_k denote its bit a position k. For a vector $\mathbf{a} = [a_0, \ldots, a_{n-1}]$ with elements in the field $\mathbb{F}$, we define its multilinear extension polynomial $f_{\mathbf{a}} : \mathbb{F}^\ell \to \mathbb{F}$ as a polynomial of ℓ variables that servers as a multiplexer for the vector $\mathbf{a}$, i.e.,

$$f_{\mathbf{a}}(x_1, \ldots, x_\ell) = \sum_{i=0}^{n-1} \left(a_i \cdot \prod_{k=1}^{\ell} \mathsf{select}_{i_k}(x_k) \right) , \tag{1}$$

where

$$\mathsf{select}_{i_k}(x_k) = \begin{cases} x_k & \text{if } i_k = 1 \\ 1 - x_k & \text{if } i_k = 0 \end{cases} . \tag{2}$$

Note that polynomial $f_{\mathbf{a}}$ is the unique multilinear polynomial such that for all i with binary representation $i_\ell, \ldots, i_1$ it is $f_{\mathbf{a}}(i_1, \ldots, i_\ell) = a_i$. For example, for the vector $\mathbf{a} = [5\ 2\ 8\ 3]$ it is

$$f_{\mathbf{a}}(x_1, x_2) = 5 \cdot (1 - x_2)(1 - x_1) + 2 \cdot (1 - x_2)x_1 + 8 \cdot x_2(1 - x_1) + 3 \cdot x_2 x_1 .$$

To simplify notation, we sometimes represent the point $(x_1, \ldots, x_\ell)$ as $\mathbf{x}$. The following polynomial decomposition from [28] is useful.

Lemma 1. *For any multilinear polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ and for $\mathbf{t} \in \mathbb{F}^\ell$, there exist polynomials q_i such that $f(\mathbf{x}) - f(\mathbf{t}) = \sum_{i=1}^{\ell} (x_i - t_i) q_i(\mathbf{x})$. Moreover, all q_i can be computed in $O(2^\ell) = O(|f|)$ time.*

Sparse Merkle trees. Sparse Merkle trees are Merkle trees [17] built over key-value pairs (k_i, v_i) whose keys k_i are drawn from a large domain $[0, 1, \ldots, 2^W - 1]$. In particular, data item (k_i, v_i) is stored at the k_i-th leaf of the tree (the tree has a total of 2^W leaves). We define a natural labeling for all nodes of the sparse Merkle tree: Root takes label ϵ, his left child takes label 0, his right child takes label 1, his leftmost grandchild takes label 00 and so on.

For W-bit leaf k_i that stores the data element (k_i, v_i) we define the digest of leaf k_i as $\mathsf{dig}(k_i) = k_i || v_i$. For leaves ℓ that do not store a key we set $\mathsf{dig}(\ell) = null$. For every internal node u of the Merkle tree we define the digest of u as $\mathsf{dig}(u) = H(\mathsf{dig}(v) || \mathsf{dig}(w))$, if either $\mathsf{dig}(v)$ or $\mathsf{dig}(w)$ is not null and $\mathsf{dig}(u) = null$ otherwise. Here, v is the left child of u and w is the right child of u and H is a collision-resistant hash function such as SHA-2.

Single item verification and deletion. For data item (k, v) let $path(k)$ be the ordered set of nodes on the path from k to the root ϵ and let $sib(k)$ be the ordered set of siblings of nodes on $path(k)$. Recall that the proof $\pi(k)$ for (k, v), with respect to the digest of the root $\mathsf{dig}(\epsilon)$ is the set $dig(k) \cup \{\mathsf{dig}(v) : v \in sib(k)\}$. In particular, to verify the proof, one can run a verification algorithm

$$d \leftarrow \mathsf{verifyMerkle}(k, v, \pi(k)) \tag{3}$$

that recomputes the digest of the root. If $d = \mathsf{dig}(\epsilon)$ the verification is successful and one can be assured (except with negligible probability) that (k, v) is the k-th leaf of the sparse Merkle tree. After a successful verification, the verification algorithm can also be used to update $\mathsf{dig}(\epsilon)$ when (k, v) is deleted from the tree. In particular one can run $\mathsf{verify}(k, null, \pi(k))$ to output the new digest d'. The proof size is $O(W)$ and the verification complexity is $O(W)$.

Batch verification and deletion. The above approach can be generalized for verifying a set of data items $S = \{(k_1, v_1), \ldots, (k_t, v_t)\}$ at once. In particular let $path(S)$ be the union of $path(k_i)$ and let $sib(S)$ be the union of $sib(k_i)$. The proof $\pi(S)$ for all $(k_1, v_1), \ldots, (k_t, v_t)$ in S, with respect to the digest of the root $\mathsf{dig}(\epsilon)$ is the set $\cup_i dig(k_i) \cup \{\mathsf{dig}(v) : v \in sib(S)\}$. In this case we can call $\mathsf{verifyMerkle}(S, \pi(S))$ to recompute the digest of the sparse Merkle tree and verify all elements in S, as well as $\mathsf{verifyMerkle}(\{(k_1, null), \ldots, (k_t, null)\}, \pi(S))$ for the batch deletion of all data elements in S.

Vector commitments. We now give the definition of *vector commitment*, introduced by Catalano and Fiore in [11]. We have changed the syntax a bit, to reflect our application better. In particular we distinguish between the prover public key prk, the verifier public key vrk and the update public key upk_i.

Definition 1 (Vector commitment scheme)**.** *A vector commitment scheme $\mathcal{V}$ consists of the following* PPT *algorithms:*

1. $(\mathsf{prk}, \mathsf{vrk}, \mathsf{upk}_0, \ldots, \mathsf{upk}_{n-1}) \leftarrow \mathsf{KeyGen}(1^\lambda, n)$*: Given security parameter λ and vector length n, it outputs a prover key* prk*, a verifier key* vrk *and update keys* $\mathsf{upk}_0, \ldots, \mathsf{upk}_{n-1}$.[4]
2. $\mathsf{dig} \leftarrow \mathsf{Setup}(a_0, \ldots, a_{n-1}, \mathsf{prk})$*: Given prover key* prk *and vector* $\mathbf{a} = (a_0, \ldots, a_{n-1})$*, it outputs a digest* dig.
3. $(a_i, \pi_i) \leftarrow \mathsf{Prove}(i, \mathbf{a}, \mathsf{prk})$*: Given prover key* prk*, vector* $\mathbf{a}$ *and index i, it outputs element a_i and proof π_i.*
4. $\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{dig}, i, a, \pi, \mathsf{vrk})$*: Given verifier key* vrk*, digest* dig*, an index i, a value a, it outputs a bit denoting either accept or reject.*
5. $\mathsf{dig} \leftarrow \mathsf{UpdateDigest}(\mathsf{dig}, u, \delta, \mathsf{upk}_u)$*: Given update key* upk_u*, a digest* dig*, an index u, an update δ*[5] *at index u, it outputs the updated digest* dig.
6. $\pi_i \leftarrow \mathsf{UpdateProof}(\pi_i, u, \delta, \mathsf{upk}_u)$*: Given update key* upk_u*, a proof π_i for a value at index i, an update δ at index u, it outputs the updated proof π_i.*

The correctness definition for vector commitments is in the Appendix—Definition 3. We now present the soundness definition.

Definition 2 (Soundness of vector commitment scheme)**.** *Consider the following experiment that takes as input the security parameter λ and outputs vector* $\mathbf{a}$*, index i, value a and a bit b.*

- *Let n be output by the adversary $\mathcal{A}$;*
- $(\mathsf{prk}, \mathsf{vrk}, \mathsf{upk}_0, \ldots, \mathsf{upk}_{n-1}) \leftarrow \mathsf{KeyGen}(1^\lambda, n)$*;*
- *Let* $\mathbf{a} = [a_0, \ldots, a_{n-1}]$ *be output by adversary $\mathcal{A}$;*
- $\mathsf{dig} \leftarrow \mathsf{Setup}(\mathbf{a}, \mathsf{prk})$*;*
- *for* $i = 1, \ldots, t = poly(\lambda)$
  - *Let update (u, δ) be output by the adversary $\mathcal{A}$ and let* $\mathbf{a}$ *be the updated vector;*
  - $\mathsf{dig} \leftarrow \mathsf{UpdateDigest}(\mathsf{dig}, u, \delta, \mathsf{upk}_u)$*;*

[4] For an update (u, δ) on index u, update key upk_u is required to update either the digest or some other proof π_i where $i = 1, \ldots, n-1$. Note that all update keys are public and can be regarded as the pubic key of the system.

[5] Value δ can be either positive or negative indicating credit or debit for account u.

Algorithm 1 Algorithm for updating most recent proof when a new output $[\mathsf{PK}, v]$ is generated.

procedure $\pi(cnt+1) \leftarrow$ UPDATEMOSTRECENTPROOF($\pi(cnt), \mathsf{PK}, v$)
  Parse $\pi(cnt)$ as $d_0, \ldots, d_W$ where d_0 is $cnt||[pk, v']$;
  Initialize $\pi(cnt+1)$ as $\delta_0, \ldots, \delta_W$ where δ_0 is $(cnt+1)||[\mathsf{PK}, v]$ and $\delta_i = null$ for $i > 0$;
  Let the binary representations of cnt and $cnt+1$ be $B||b_k, \ldots, b_0$ and $B||\beta_k, \ldots, b_0$ respectively, where B is their common prefix;
  Set $q = |B| - 1$;
  Copy the last q hashes from $\pi(cnt)$ to the last q positions of $\pi(cnt+1)$;
  Let π contain the first $W - q$ hashes of $\pi(cnt)$;
  Run $d \leftarrow \mathsf{verifyMerkle}(cnt, [pk, v], \pi)$;
  Set $\delta_{W-q+1} = d$ and $\delta_i = null$ for all $0 < i < W - q + 1$;
  return $\pi(cnt+1)$;

- *Let i, a, π be output by the adversary and let*

$$b \leftarrow \mathsf{Verify}(\mathsf{dig}, i, a, \pi, \mathsf{vrk})\,.$$

- **return** $(\mathbf{a}, i, a, b)$*;*

A vector commitment scheme is sound if for all PPT *adversaries $\mathcal{A}$, the probability $b = 1$ and $a \neq a_i$, where a_i is the value at i is negligible.*

3 EDRAX in UTXO-based model

As we mentioned in the introduction, the UTXO-based model follows the design of Bitcoin-like cryptocurrencies, where validating a transaction tx depends on whether its inputs belong to a set of *unspent transaction outputs* (or UTXO) that is maintained by the miners. Once this condition is verified, the spent inputs of tx are removed from UTXO and the new outputs of tx are added to UTXO. We first describe a version of EDRAX in this model.

Representing UTXO as a sparse Merkle tree. We will represent the UTXO set as a sparse Merkle hash tree. A similar approach has been used in Zcash [8]—unlike Zcash, however, here we remove a transaction output from the Merkle tree after it is spent by marking is as $null$, see Figure 2. In particular each element of the UTXO is of the form $(i, [pk, v])$ where i is the an increasing timestamp/counter indicating when this output was added to the UTXO (and serves as the "key" in the sparse Merkle tree) and $[pk, v]$ contains the public key pk and EDRAX units v that this output can be spent to (and serves as the "value" in the sparse Merkle tree).

Validation digest. Every block b at time t (t refers to the rank of the block in the blockchain) in UTXO-based EDRAX contains the following information as validation digest.

1. The UTXO digest digest_t which is the roothash of the sparse Merkle tree built on transaction outputs that have been generated up to block b, block b included (for outputs that have already been spent we mark them as $null$);

2. The Merkle tree proof of the most recent entry in the UTXO, i.e., the Merkle tree proof corresponding to the last output of the last transaction in the last block b. We call this proof *most recent proof* and we denote it as π_t.

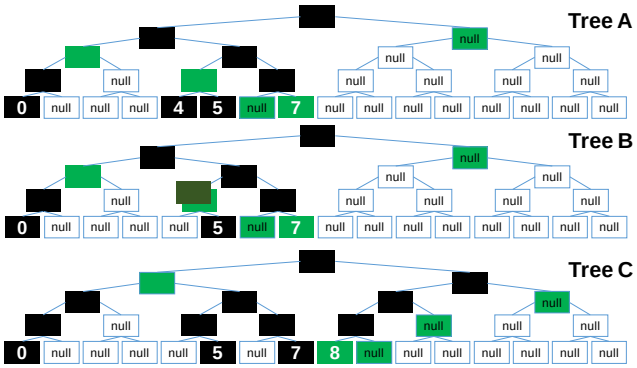

Figure 2: Evolution of the sparse Merkle tree on the UTXO set. In Tree A, the UTXO set contains 4 unspent outputs that were added with timestamps 0, 4, 5 and 7. The outputs that were added with timestamps 1, 2, 3 and 6 have already been spent (and thus deleted/nullified). We highlight with green color the nodes that constitute the most recent proof $\pi(7)$. In Tree B a new transaction tx with input 4 appears in the system (thus 4 must be deleted from the tree), causing the most recent proof $\pi(7)$ to change one of its hashes to $H(null||dig(5))$, indicated with dark green color. The output of tx is added in Tree C at the next position 8, causing the update of the most recent proof from $\pi(7)$ to $\pi(8)$. Note that $\pi(8)$ in Tree C can be easily computed from $\pi(7)$ in Tree B using Algorithm 1.

Client state. A EDRAX client stores the list $\mathcal{L}$ of his unspent transaction outputs $(i, [pk_i, v_i])$ (i.e., the ones for which he knows respective secret keys sk_i) as well as respective Merkle proofs $\pi(i)$.

`SPEND` transaction. For simplifying exposition, suppose Alice wants to create a transaction tx that spends a transaction output $(x, [pk, v])$ in her local list $\mathcal{L}$ to a specific public key PK (we can trivially generalize the `SPEND` transaction for multiple inputs and outputs). Let sk be the corresponding secret key to pk. Alice constructs and signs, using sk, the following transaction:

$$[(x, [pk, v]), \pi(x), \mathsf{PK}] .$$

To be valid, $\pi(x)$ must refer to the last block, i.e., block at time t.

New block creation. Suppose the last block that was computed is block t and miners are competing to compute block $t + 1$. To do that miners collect incoming `SPEND` transactions of the type

$$[(x, [pk, v]), \pi(x), \mathsf{PK}], sig$$

and decide using the validation digest stored at block t whether to include a transaction in block $t + 1$ by performing the following:

1. (transaction signature verification) Check that signature sig is valid under public key pk;
2. (verifying membership of transaction input in UTXO) Run

   $$\mathsf{verifyMerkle}(x, [pk, v], \pi(x))$$

   as in Relation 3 to output a digest d. If d equals digest_t (part of the validation digest), then the miner is assured $(x, [pk, v])$ exists in the UTXO set and thus can be spent.

The time required for verifying a transaction is $O(W)$ since one Merkle tree proof must be verified per transaction. The transactions that satisfy the above checks are candidates for the next block. The block

has size $O(m \cdot W)$ where m is the number of transactions in the block. Finally, in the new block $t+1$, miners must also include the updated validation digest, i.e., UTXO digest digest_{t+1} and the new most recent proof π_{t+1}. We describe this procedure next.

Update of validation digest. We first show how to update the validation digest digest_t, π_t for one transaction $[(x, [pk, v]), \pi(x), \mathsf{PK}]$ with one input and one output and then we generalize to multiple transactions—see Figure 2. In particular, to compute digest_{t+1} and π_{t+1} given digest_t and π_t the miners perform the following steps:

1. (deleting transaction input from UTXO) Update the UTXO digest digest_t to d' to not contain spent input $(x, [pk, v])$ anymore by running $d' \leftarrow \mathsf{verifyMerkle}(x, null, \pi(x))$ as we described in Section 2;

2. (updating most recent proof due to deletion) Let cnt be the timestamp corresponding to the most recent proof π_t. For every node $v \in sib(cnt) \cap path(x)$ replace every hash $dig(v)$ in the most recent proof π_t with the new hashes $dig(v)$ as computed by running algorithm $\mathsf{verifyMerkle}(x, null, \pi(x))$ above leading to a new proof π'—see Tree B in Figure 2.

3. (adding transaction output to UTXO) The new transaction output should now be stored at leaf $cnt+1$ as $(cnt+1), [\mathsf{PK}, v])$. Because of the addition of the new leaf, the most recent proof $\pi' = \pi(cnt)$ computed above must be updated to $\pi_t = \pi(cnt+1)$. Intuitively this can be done since π_t is "to the right" of π'—see `UpdateMostRecentProof` (Algorithm 1) for the detailed pseudocode and Tree C in Figure 1. After π_{t+1} is computed, miners can finally update the digest d' by running $\mathsf{verifyMerkle}(cnt+1, [\mathsf{PK}, v], \pi_t)$ which will output the final digest digest_{t+1}.

Processing multiple inputs and outputs. To process a block with more than one transactions with more than one inputs and outputs (as it typically happens in practice), miners must perform *batch verification* and *batch deletion* to verify and delete the inputs from the UTXO, as described in Section 2. This not just an optimization, but it is needed for correctness (otherwise proofs will be out-of-sync). Finally, to add the new outputs to the UTXO, the miners run Step (5) above as many times as the number of new outputs generated in the block. Updating the validation digest with the above steps takes $O(m \cdot W)$ time where m is the total number of transaction inputs and outputs in the block.

Proof computation and synchronization. For an unspent output $(x, [pk, v])$ in the UTXO, let $\pi(x)$ be the proof stored locally by the client with respect to time t. To synchronize $\pi(x)$ for time $t+1$, the client must process all transactions in the block at time $t+1$ by performing the same steps as the miners above. But instead of outputting digest digest_{t+1} at time $t+1$ and most recent proof π_{t+1} at time $t+1$, he just replaces the affected hashes in his proof $\pi(x)$, due batch deletion and addition of the new outputs. In general, to synchronize between t_1 and t_2, he repeats this process $t_2 - t_1$ times.

4 EDRAX in account-based model

We now describe our version of EDRAX stateless cryptocurrency that uses balances (such as Nxt, Ethereum, Bitshares, NEM, Tezos). Recall in such systems the miners maintain a database with balances and transaction validity is checked against this database (instead of UTXO). To implement EDRAX in the account-based model, we will use any secure vector commitment scheme (as given in Definition 1) as a black box but in Section 5 we provide a concrete construction which is efficient both asymptotically and in practice.

Setup. Just like Zcash [8], EDRAX requires an one-time setup phase. In particular given an upper bound n on the number of accounts that EDRAX can support[6] and the security parameter λ, algorithm

[6] E.g., for Ethereum the number of accounts now is approximately 30 million; we will show experiments for 1 billion accounts.

$\mathsf{KeyGen}(1^\lambda, n)$ is executed outputting the prover key prk, the verification key vrk and update keys upk_i—these public parameters are hardcoded into the EDRAX reference software client. As an optimization one can just hardcode a Merkle tree digest of these parameters (since they can be quite large) and retrieve them as required during the build—this technique is used in Zcash, for example. Finally, to mitigate the risk of trapdoor leakage during execution of KeyGen, we can use a secure multiparty computation protocol as in [10].

Validation digest. Just like in other cryptocurrencies, EDRAX miners store the whole blockchain. Also for each block b at time t they include, along with transaction data, two constant-size values:

1. The account digest digest_t which is a summary (hash) of the account balances in the system up to block b, block b included. It is computed using a vector commitment scheme as in Definition 1. It is initialized by running $\mathsf{dig} \leftarrow \mathsf{Setup}(0, \ldots, 0, \mathsf{prk})$ which, in our implementation, is $g^0 = \mathbf{1}$, where $\mathbf{1}$ denotes the identity element of the bilinear group $\mathbb{G}$. In general the digest digest_t will be on a vector $\mathbf{a}$ that stores mappings of public keys to balances. Our implementation does that by storing mappings of the type
$$i \rightarrow [h(\mathsf{PK})||balance]$$
where i is in $\{0, 1, \ldots, n-1\}$ and is assigned by miners for a specific public key PK—this assignment is triggered via a special `INIT` transaction that serves as "registration" for a new user and is described in the following.[7] [8]

2. The account counter cnt_t that indicates how many `INIT` transactions have occurred up to block b, block b included—roughly speaking this indicates how many accounts are currently in the system. It is initialized as 0.

Client state. Apart from a public and a secret key required in other cryptocurrencies, an EDRAX client is required to store the local proof π for the value of his balance with respect to the account digest digest_t. In our implementation, proof π is particularly small, having only $\log n$ group elements. Also in our implementation each proof is initialized as $(\mathbf{1}, \ldots, \mathbf{1})$ where $\mathbf{1}$ is the group identity element.

`INIT` transaction. Just like in Bitcoin and Ethereum, the first time Alice ever wants to use EDRAX, she creates a pair of private and public keys (sk, pk) (e.g., using elliptic curve cryptography). Recall however that EDRAX represents accounts as integers in $\{0, 1, \ldots, n-1\}$ (where, in our implementation n is around 2^{30}) and therefore a mechanism to map Alice's public key pk to an integer $i \in \{0, 1, \ldots, n-1\}$ must be in place. To achieve that, EDRAX offers an `INIT` transaction that allows Alice to map her public key to the next available index i. In particular Alice constructs and signs, using sk, the transaction
$$[\texttt{INIT}, pk]\,.$$
Looking forward, after registering a mapping of the next available index i to public key pk, this transaction will implicitly define Alice's public key PK_a as $[pk||i||\mathsf{upk}_i]$ where upk_i is the update key of the vector commitment scheme.

`SPEND` transaction. Let us assume that Alice has public key $\mathsf{PK}_a = [pk_a||i||\mathsf{upk}_i]$, corresponding secret key sk_a and current balance equal to v' EDRAX units. She wants to send $v \leq v'$ EDRAX units to Bob that has public key $\mathsf{PK}_b = [pk_b||j||\mathsf{upk}_j]$. Alice constructs and signs, using sk_a, the following transaction:
$$[\mathsf{PK}_a, \mathsf{PK}_b, v, \pi_i, v']\,,$$

[7] EDRAX cannot map public keys directly to balances as the vector commitment supports only a polynomial number of indices and the domain of public keys is exponential.

[8] As in Ethereum, EDRAX stores the mapping $i \rightarrow [h(\mathsf{PK})||nonce||balance]$, where $nonce$ shows how many payments have been made out from PK. This is necessary to distinguish between two separate payments from the same public key and a replay attack of the same payment. For simplicity, we do not include $nonce$ in our exposition.

meaning that public key PK_a wishes to send v EDRAX units to public key PK_b and π_i is her local proof proving that PK_a has enough funds $v' \geq v$ (with respect to the latest account digest digest_t) to support this transaction.

New block creation. Again, assume the last block that was computed is block t and miners compete to compute block $t+1$. To do that miners collect new `INIT` and `SPEND` transactions of the type

$$[\texttt{INIT}, pk], sig$$

and

$$[\mathsf{PK}_a, \mathsf{PK}_b, v, \pi_i, v'], sig$$

respectively. For an `INIT` transaction to be candidate for inclusion in block $t+1$, it is enough that its signature verifies. To decide whether a `SPEND` transaction $[\mathsf{PK}_a, \mathsf{PK}_b, v, \pi_i, v'], sig$ can be included in the next block the miner needs to perform the following steps:

1. Parse PK_a as $[pk_a||i||\mathsf{upk}_i]$ and check whether sig is a valid signature under pk_a;
2. Check whether $v \leq v'$;
3. Check whether $1 \leftarrow \mathsf{Verify}(\mathsf{digest}_t, i, h(\mathsf{PK}_a)||v', \pi_i, \mathsf{vrk})$ where digest_t is the account digest of the current block (at time t).

The new block has size $O(m \log n)$ where n is the upper bound on the number of accounts and m is the number of transactions included in the block. Finally, in the new block $t+1$, miners must also include the updated validation digest, i.e., account digest digest_{t+1} and the new account counter cnt_{t+1}. We describe how miners compute these updated values next.

Update of validation digest. To update the validation digest the miners initially set $cnt \leftarrow \mathsf{cnt}_t$ and $digest \leftarrow \mathsf{digest}_t$. Then they consider `INIT` transactions first and `SPEND` transactions later.

In particular for every verified `INIT` transaction $[\texttt{INIT}, pk], sig$ sent by Alice to be included in block $t+1$ the miners set $cnt = cnt + 1$ and implicitly assign the updated index cnt to pk[9]. Then they set

$$digest \leftarrow \mathsf{UpdateDigest}(digest, cnt, \delta, \mathsf{upk}_{cnt})$$

where $\delta = h(\mathsf{PK})||0$ (we assume Alice begins with 0 balance) and where $\mathsf{PK} = [pk||cnt||\mathsf{upk}_{cnt}]$. This operation essentially registers Alice's public key to a specific index cnt.

Then for every verified `SPEND` transaction $[\mathsf{PK}_a, \mathsf{PK}_b, v, \pi_i, v'], sig$ the miners set

$$digest \leftarrow \mathsf{UpdateDigest}(digest, i, -v, \mathsf{upk}_i)$$

and then again

$$digest \leftarrow \mathsf{UpdateDigest}(digest, j, +v, \mathsf{upk}_j)\,,$$

where $\mathsf{PK}_a = [pk_a||i||\mathsf{upk}_i]$ and $\mathsf{PK}_b = [pk_b||j||\mathsf{upk}_j]$. This transaction updates the balances of the sender and the receiver accordingly. Finally the miners set $\mathsf{cnt}_{t+1} \leftarrow cnt$ and $\mathsf{digest}_{t+1} \leftarrow digest$ and output the new validation digest to be included in block $t+1$.

Proof synchronization. Let π be Alice's local proof that corresponds to the state of the system up until block t. Now assume some transactions are taking place and block $t+1$ is created. Alice's local proof π is no longer valid and Alice must synchronize her proof to make sure it incorporates all updates $(u_1, \delta_1), (u_2, \delta_2), \ldots, (u_p, \delta_p)$ that were included in block $t+1$. To do that she executes Algorithm 2 by running $\pi \leftarrow$ SYNCHRONIZEPROOF$((u_1, \delta_1), \ldots, (u_p, \delta_p), \pi)$ and outputs a new synchronized proof π. Note that Alice does not have to synchronize her proof at every new block. She just has to synchronize her proof whenever she wants to spend some EDRAX units to someone else in which case she must process all the blocks since her proof was last synchronized in the same way as above.

[9] This defines Alice's public key as $[pk||cnt||\mathsf{upk}_cnt]$.

Algorithm 2 Algorithm for synchronizing the proof so that updates $(u_1, \delta_1), \ldots, (u_t, \delta_t)$ are included.

procedure $\pi \leftarrow$ SYNCHRONIZEPROOF$((u_1, \delta_1), \ldots, (u_p, \delta_p), \pi)$
  for $i = 1, \ldots, p$ **do**
    $\pi \leftarrow \mathsf{UpdateProof}(\pi, u_i, \delta_i, \mathsf{upk}_{u_i})$;
  return π;

Algorithm 3 Algorithm for computing the polynomials that are required to update the proof at position i on update (u, δ).

procedure $[\Delta_\ell(\mathbf{X}), \ldots, \Delta_1(\mathbf{X})] \leftarrow$ DELTAPOLYNOMIALS(u, δ, i, ℓ)
  if $\ell > 0$ **then**
    if msb of u is 0 and msb of i is 1 **then return** $\left[-\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k), 0, \ldots, 0\right]$;
    if msb of u is 1 and msb of i is 0 **then return** $\left[+\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k), 0, \ldots, 0\right]$;
    if msb of u is 0 and msb of i is 0 **then return**

$$\left[-\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k), \text{DELTAPOLYNOMIALS}(u \bmod 2^\ell, \delta, i \bmod 2^\ell, \ell - 1)\right] ;$$

    if msb of u is 1 and msb of i is 1 **then return**

$$\left[+\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k), \text{DELTAPOLYNOMIALS}(u \bmod 2^\ell, \delta, i \bmod 2^\ell, \ell - 1)\right] ;$$

5 Vector Commitment Construction

We now present our new vector commitment construction that we use in the implementation of account-based EDRAX. We present all algorithms in detail, as defined in Definition 1 and then we prove correctness as defined in Definition 3 in the Appendix and soundness, as required by Definition 2.

$(\mathsf{prk}, \mathsf{vrk}, \mathsf{upk}_0, \ldots, \mathsf{upk}_{n-1}) \leftarrow \mathsf{KeyGen}(1^\lambda, n)$: Let $(p, \mathbb{G}, \mathbb{G}_T, e, g)$ be output by running $\mathsf{BilGen}(1^\lambda)$. Let $\ell = \log n$ and let $\mathcal{S}$ be the powerset of $\{1, 2, \ldots, \ell\}$. Select $s_1, \ldots, s_\ell$ randomly from $\mathbb{F}$ and set

$$\mathsf{prk} = \left\{ g^{\prod_{i \in S} s_i} : S \in \mathcal{S} \right\} \text{ and } \mathsf{vrk} = \{g^{s_1}, \ldots, g^{s_\ell}\} .$$

Also for all $u = 0, \ldots, n-1$, we have that the update key for position u contains ℓ group elements, i.e.,

$$\mathsf{upk}_u = \left\{ g^{\prod_{k=1}^{t} \mathsf{select}_{u_k}(s_k)} : t = 1, \ldots, \ell \right\} = \left\{ \mathsf{upk}_{u,t} : t = 1, \ldots, \ell \right\} .$$

where $\mathsf{select}_{u_k}(s_k)$ is defined in Equation 2.

$\mathsf{dig} \leftarrow \mathsf{Setup}(a_0, \ldots, a_{n-1}, \mathsf{prk})$: Set digest

$$\mathsf{dig} = g^{f_{\mathbf{a}}(s_1, \ldots, s_\ell)} ,$$

where $f_{\mathbf{a}}$ is the multilinear extension polynomial of the vector $\mathbf{a} = [a_0, \ldots, a_{n-1}]$ as defined in Equation 1.

$(a_i, \pi_i) \leftarrow \mathsf{Prove}(i, \mathbf{a}, \mathsf{prk})$: Let $i_\ell, \ldots, i_1$ be the binary representation of i. As $a_i = f_{\mathbf{a}}(i_1, \ldots, i_\ell)$, using polynomial decomposition, compute polynomials $q_1, \ldots, q_\ell$ such that

$$f_{\mathbf{a}}(\mathbf{x}) - f_{\mathbf{a}}(i_1, \ldots, i_\ell) = \sum_{k=1}^{\ell} (x_k - i_k) q_k(\mathbf{x}) \,.$$

Output the proof π_i as $\{g^{q_1(\mathbf{s})}, \ldots, g^{q_\ell(\mathbf{s})}\}$. To compute polynomials $q_1, \ldots, q_\ell$, we divide $f_{\mathbf{a}}(\mathbf{x}) - f_{\mathbf{a}}(i_1, \ldots, i_\ell)$ by $x_\ell - i_\ell$ and set q_ℓ as the quotient polynomial of the division, which is a multilinear polynomial with variables $x_1, \ldots, x_{\ell-1}$. The remainder is a multilinear polynomial with variables $x_1, \ldots, x_{\ell-1}$, which we divide by $x_{\ell-1} - i_{\ell-1}$ to get $q_{\ell-1}$. We repeat recursively until we get q_1.

$\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{dig}, i, a, \pi, \mathsf{vrk})$: Parse π as $w_1, \ldots, w_\ell$ and output 1 iff $e(\mathsf{dig}/g^a, g) = \prod_{k=1}^{\ell} e(g^{s_i - i_k}, w_k)$.

$\mathsf{dig} \leftarrow \mathsf{UpdateDigest}(\mathsf{dig}, u, \delta, \mathsf{upk}_u)$: Compute the new digest as

$$\mathsf{dig} = \mathsf{dig} \cdot \left[g^{\prod_{k=1}^{\ell} \mathsf{select}_{u_k}(s_k)} \right]^{\delta} = \mathsf{dig} \cdot \left[\mathsf{upk}_{u,\ell} \right]^{\delta} \,.$$

$\pi_i \leftarrow \mathsf{UpdateProof}(\pi_i, u, \delta, \mathsf{upk}_u)$: Parse π_i as $w_1, \ldots, w_\ell$. For $i = 1$ to ℓ set

$$w_i = w_i \cdot g^{\Delta_i(\mathbf{s})} \,,$$

where $\Delta_\ell(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ are the polynomials computed by calling DELTAPOLYNOMIALS(u, δ, i, ℓ) (see Algorithm 3). Note that it is very easy to modify DELTAPOLYNOMIALS to output the terms $g^{\Delta_i(\mathbf{s})}$ directly by allowing it to access the update key upk_u of u (so instead for computing, for example, the polynomial $-\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k)$ it can just directly output $[\mathsf{upk}_{u,\ell-1}]^{-\delta}$).

5.1 Correctness of DELTAPOLYNOMIALS

The proof of correctness of our new vector commitment scheme lies upon proving the correctness of DELTAPOLYNOMIALS used to update the proof for an index i, π_i, when another update (u, δ) takes place on an index u. To see why DELTAPOLYNOMIALS correctly performs this task, note that before the update (u, δ), proof π_i consists of $\{g^{q_1(\mathbf{s})}, \ldots, g^{q_\ell(\mathbf{s})}\}$ where polynomials $q_i(\mathbf{x})$ satisfy $f_{\mathbf{a}}(x_1, \ldots, x_\ell) - a_i = \sum_{k=1}^{\ell} (x_k - i_k) \cdot q_k(\mathbf{x})$. Due to the update (u, δ), the digest $f_{\mathbf{a}}(x_1, \ldots, x_\ell)$ increases by $\delta \cdot \prod_{k=1}^{\ell} \mathsf{select}_{u_k}(x_k)$ and therefore polynomials $q_i(\mathbf{x})$ should be adjusted to $q_i(\mathbf{x}) + \Delta_i(\mathbf{x})$ to accommodate this change, as described in the following lemma:

Lemma 2. *Algorithm DeltaPolynomials(u, δ, i, ℓ) correctly computes polynomials $\Delta_\ell(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ such that*

$$\delta \cdot \prod_{k=1}^{\ell} \mathsf{select}_{u_k}(x_k) = \sum_{k=1}^{\ell} (x_k - i_k) \cdot \Delta_k(\mathbf{x}) \,, \text{ if } u \neq i \tag{4}$$

or

$$\delta \cdot \prod_{k=1}^{\ell} \mathsf{select}_{u_k}(x_k) - \delta = \sum_{k=1}^{\ell} (x_k - i_k) \cdot \Delta_k(\mathbf{x}) \,, \text{ if } u = i \,, \tag{5}$$

where i_k is the k-th bit of i.

Proof. By induction on ℓ. For the base case, note that Algorithm DELTAPOLYNOMIALS$(u, \delta, i, 1)$ outputs $\Delta_1(\mathbf{x}) = -\delta$ in case u is 0 and i is 1 or both u and i are 0 and $\Delta_1(\mathbf{x}) = \delta$ in case u is 1 and i is 0 or both u and i are 1. Indeed $\Delta_1(\mathbf{x})$ does satisfy the relations above as we prove in the following by considering all four possible cases.

1. $u = 0$ and $i = 1$. In this case Relation 4 is indeed satisfied as $\delta \cdot (1 - x_1) = (x_1 - 1) \cdot (-\delta)$.

2. $u = 0$ and $i = 0$. In this case Relation 5 is indeed satisfied as $\delta \cdot (1 - x_1) - \delta = x_1 \cdot (-\delta)$.

3. $u = 1$ and $i = 0$. In this case Relation 4 is indeed satisfied as $\delta \cdot x_1 = (x_1 - 0) \cdot \delta$.

4. $u = 1$ and $i = 1$. In this case Relation 5 is indeed satisfied as $\delta \cdot x_1 - \delta = (x_1 - 1) \cdot \delta$.

For the inductive hypothesis, assume DELTAPOLYNOMIALS$(u, \delta, i, \ell-1)$ outputs polynomials $\Delta_{\ell-1}(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ that satisfy either Relation 4 or Relation 5 (depending whether $u = i$ or not). We prove the same claim for DELTAPOLYNOMIALS(u, δ, i, ℓ) by considering the following cases.

1. If msb of u is 0 and msb of i is 1, then the algorithm returns $\Delta_\ell(\mathbf{x}) = -\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k)$ and $\Delta_i(\mathbf{x}) = 0$ for all $i < \ell$. Since $u \neq i$, these polynomials must satisfy Relation 4 which can be rewritten as

$$\delta \cdot (1 - x_\ell) \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) = (x_\ell - 1) \cdot \Delta_\ell(\mathbf{x}) + \sum_{k=1}^{\ell-1} (x_k - i_k) \cdot \Delta_k(\mathbf{x}) \,.$$

It is easy to see that this is indeed the case by simple substitution.

2. If msb of u is 0 and msb of i is 0, then the algorithm returns $\Delta_\ell(\mathbf{x}) = -\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k)$ along with $\Delta_{\ell-1}(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ as output by DELTAPOLYNOMIALS$(u \bmod 2^\ell, \delta, i \bmod 2^\ell, \ell - 1)$. We distinguish two subcases.

   (a) $u \neq i$. In this case polynomials $\Delta_\ell(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ must satisfy Relation 4 which can be rewritten as

$$\delta \cdot (1 - x_\ell) \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) = x_\ell \cdot \Delta_\ell(\mathbf{x}) + \sum_{k=1}^{\ell-1} (x_k - i_k) \cdot \Delta_k(\mathbf{x}) \,.$$

   By substituting the output polynomials and by using our inductive hypothesis that states $\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) = \sum_{k=1}^{\ell-1} (x_k - i_k) \cdot \Delta_k(\mathbf{x})$ it is easy to see that this is indeed the case.

   (b) $u = i$. In this case polynomials $\Delta_\ell(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ must satisfy Relation 5 which can be rewritten as

$$\delta \cdot (1 - x_\ell) \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) - \delta = x_\ell \cdot \Delta_\ell(\mathbf{x}) + \sum_{k=1}^{\ell-1} (x_k - i_k) \cdot \Delta_k(\mathbf{x}) \,.$$

   By substituting the output polynomials and by using our inductive hypothesis that states $\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) - \delta = \sum_{k=1}^{\ell-1} (x_k - i_k) \cdot \Delta_k(\mathbf{x})$ it is easy to see that this is indeed the case.

3. If msb of u is 1 and msb of i is 0, then algorithm returns $\Delta_\ell(\mathbf{x}) = \delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k)$ and $\Delta_i(\mathbf{x}) = 0$ for all $i < \ell$. Since $u \neq i$, these polynomials must satisfy Relation 4 which can be written as

$$\delta \cdot x_\ell \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) = x_\ell \cdot \Delta_\ell(\mathbf{x}) + \sum_{k=1}^{\ell-1} (x_k - i_k) \cdot \Delta_k(\mathbf{x}) \,.$$

It is easy to see that this is indeed the case by simple substitution.

4. If msb of u is 1 and msb of i is 1, then the algorithm returns $\Delta_\ell(\mathbf{x}) = \delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k)$ along with $\Delta_{\ell-1}(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ as output by DELTAPOLYNOMIALS$(u \bmod 2^\ell, \delta, i \bmod 2^\ell, \ell - 1)$. We distinguish two subcases.

(a) $u \neq i$. In this case polynomials $\Delta_\ell(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ must satisfy Relation 4 which can be rewritten as

$$\delta \cdot x_\ell \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) = x_\ell \cdot \Delta_\ell(\mathbf{x}) + \sum_{k=1}^{\ell-1}(x_k - i_k) \cdot \Delta_k(\mathbf{x})\,.$$

By substituting the output polynomials and by using our inductive hypothesis that states $\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) = \sum_{k=1}^{\ell-1}(x_k - i_k) \cdot \Delta_k(\mathbf{x})$ it is easy to see that this is indeed the case.

(b) $u = i$. In this case polynomials $\Delta_\ell(\mathbf{x}), \ldots, \Delta_1(\mathbf{x})$ must satisfy Relation 5 which can be rewritten as

$$\delta \cdot (1 - x_\ell) \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) - \delta = x_\ell \cdot \Delta_\ell(\mathbf{x}) + \sum_{k=1}^{\ell-1}(x_k - i_k) \cdot \Delta_k(\mathbf{x})\,.$$

By substituting the output polynomials and by using our inductive hypothesis that states $\delta \cdot \prod_{k=1}^{\ell-1} \mathsf{select}_{u_k}(x_k) - \delta = \sum_{k=1}^{\ell-1}(x_k - i_k) \cdot \Delta_k(\mathbf{x})$ it is easy to see that this is indeed the case.

□

5.2 Security and complexity analysis

Our vector commitment scheme can be viewed as an application of the selectively-secure verifiable polynomial delegation scheme in [20], for the multilinear polynomial of Relation 1. While selective security is very weak in general, we show it is enough for our application. This is because a vector commitment requires evaluating the polynomial on a fixed number of points, i.e., on the hypercube $\{0,1\}^\ell$ and not on arbitrary (exponentially-many) points in $\mathbb{Z}_p^\ell$. We provide the detailed proof in the following.

Theorem 1. *The vector commitment scheme presented above is sound according to Definition 2 and under Assumption 1.*

Proof. For the proof of soundness we will use the following technique. First an ℓ-SBDH instance

$$((p, \mathbb{G}, \mathbb{G}_T, e, g), g^s, \ldots, g^{s^\ell})$$

is given to adversary $\mathcal{A}_1$. Then $\mathcal{A}_1$ picks random $b \in \{0,1\}^\ell$ (such that 2^ℓ is $poly(\lambda)$) and implicitly computes $s_1 = s$ and $s_i = r_i \cdot (s - b_1) + b_i$ where r_i are chosen at random. Note now that, given

$$g^s, \ldots, g^{s^\ell}\,,$$

$\mathcal{A}_1$ can easily compute prk, vk and upk_u for all $u = 0, 1, \ldots, n-1$, which he all gives to $\mathcal{A}_2$. Moreover, all of these keys are indistinguishable from the output of KeyGen since the r_i's have been picked at random. We now show that if $\mathcal{A}_2$ is able to break soundness as defined in Definition 2 (by providing a forgery (x, a, π) to $\mathcal{A}_1$) then $\mathcal{A}_1$ will be able to use that forgery and break Assumption 1.

Indeed, given a vector $\mathbf{a}$ and the corresponding digest dig, suppose $\mathcal{A}_2$ is able to output a forgery (x, a, π) such that it holds $1 \leftarrow \mathsf{Verify}(\mathsf{dig}, x, a, \pi, \mathsf{vrk})$ and $a \neq a_x$, where a_x is the current value at index x of $\mathbf{a}$ after a possible sequence of updates. Let us assume that the index x that $\mathcal{A}_2$ chose to forge is the index b that $\mathcal{A}_1$ picked previously to compute the secrets s_i—namely $x = b$. Note that the probability of that event is $1/2^\ell = 1/poly(\lambda)$. Then the following should hold (note that $\pi = (w_1, w_2, \ldots, w_\ell)$)

$$e(g^{f_{\mathbf{a}}(s_1,\ldots,s_\ell)-a}, g) = \prod_{i=1}^{\ell} e(g^{s_i - b_i}, w_i)$$

$$\Leftrightarrow e(g^{f_{\mathbf{a}}(s_1,\ldots,s_\ell)-a_x+(a_x-a)},g)=\prod_{i=1}^{\ell}e(g^{s_i-b_i},w_i)$$

$$\Leftrightarrow e(g^{\sum_{i=1}^{\ell}(s_i-b_i)q_i(s_1,\ldots,s_\ell)+(a_x-a)},g)=\prod_{i=1}^{\ell}e(g^{s_i-b_i},w_i)$$

$$\Leftrightarrow e(g,g)^{(a_x-a)}=\prod_{i=1}^{\ell}e\left(g,\frac{w_i}{g^{q_i(s_1,\ldots,s_\ell)}}\right)^{s_i-b_i}$$

$$\Leftrightarrow e(g,g)^{a_x-a}=e\left(g,\frac{w_1}{g^{q_1(s_1,\ldots,s_\ell)}}\right)^{s-b_1}\prod_{i=2}^{\ell}e\left(g,\frac{w_i}{g^{q_i(s_1,\ldots,s_\ell)}}\right)^{r_i(s-b_1)}$$

$$\Leftrightarrow e(g,g)^{\frac{a_x-a}{s-b_1}}=e\left(g,\frac{w_1}{g^{q_1(s_1,\ldots,s_\ell)}}\right)\prod_{i=2}^{\ell}e\left(g,\frac{w_i}{g^{q_i(s_1,\ldots,s_\ell)}}\right)^{r_i}$$

$$\Leftrightarrow e(g,g)^{\frac{1}{s-b_1}}=\left[e\left(g,\frac{w_1}{g^{q_1(s_1,\ldots,s_\ell)}}\right)\prod_{i=2}^{\ell}e\left(g,\frac{w_i}{g^{q_i(s_1,\ldots,s_\ell)}}\right)^{r_i}\right]^{\frac{1}{a_x-a}}.$$

Therefore $\mathcal{A}_1$ can compute $e(g,g)^{\frac{1}{s-b_1}}$ with probability $1/poly(\lambda)$ which breaks Assumption 1.

□

Asymptotic costs of vector commitment algorithms. KeyGen runs is time $O(n)$ assuming it outputs the update keys upk_u on a binary tree (some parts of the update keys are the same) and Setup runs in time $O(n)$ assuming it takes as input only the necessary parts of the update keys. The size of prk is $O(n)$, the size of vrk is $O(\log n)$ and the size of each upk_i is also $O(\log n)$. The running time of Prove is $O(n)$, due to Lemma 1. Algorithm Verify runs in $O(\log n)$ time. UpdateDigest runs in $O(1)$ time and UpdateProof runs in $O(\log n)$ time in the worst case (if one amortizes over all indices the time is $O(1)$). Note that unlike previous approaches, UpdateProof input depends only on the index that is being updated, and not on the index of the proof itself. In other words to update *any* proof π_i with respect to index u, algorithm UpdateProof takes as input the same information upk_u (that depends on this index u).

6 EDRAX Evaluation

In this section, we present an evaluation of EDRAX. In Sections 6.1 and 6.2, we first evaluate the two primitives used in EDRAX: sparse Merkle trees in the UTXO-based model (see Section 6.1) and our new vector commitment scheme in the account-based model (see Section 6.2). Then, in Section 6.3, we compare block validation times and block size between stateless validation in EDRAX and traditional stateful clients in both UTXO and account-based models.

Experimental setup. We implement our schemes in C++. We implement the sparse Merkle tree scheme using the Scrypto library [6]. We use the GMP library [3] for field arithmetic, the ate-pairing [1] on a 254-bit elliptic curve for pairings in our vector commitment and LevelDB [5] to simulate the performance of a stateful client.

We run the experiments of our schemes on an Amazon EC2 c4.4xlarge machine with 30GB of RAM and an Intel Xeon E5-2666v3 CPU with 16 2.9GHz virtual cores. To simulate the cost of disk I/O for stateful clients, we use a laptop with Intel Core i7, 16 GB RAM and 1TB HDD of 5400 RPM (SATA interface, 128 GB SSD cache). We perform 10 runs and report their average for each data point of running time, unless stated otherwise. Note that for stateful clients, the state is never larger than 128 GB

so all state in these experiments is stored on the SSD, which means faster (in comparison with hard disk) reported times for the case of stateful clients.

6.1 Evaluation of sparse Merkle tree

Table 2 shows the performance of our sparse Merkle tree for different values of W (recall W is the height of the tree). Sparse Merkle trees are quite efficient due to the lightweight SHA-2 hash function used. For example for $W = 40$, it takes 0.063ms to verify a proof and the proof size is 0.35KB, which is enough for supporting the total number of transaction outputs ever generated in Bitcoin—right now this number is about 765 million, as computed using the BlockSci tool [4]. We implemented the algorithms to update the most recent proof as well as the algorithm for the local proof synchronization presented in Section 3. For a transaction of one input and one output the time to update the most recent proof is 0.081ms and the time to synchronize the local proof is 0.011ms for $W = 30$. All numbers above scale linearly with W (logarithmically to number of leaves).

Table 2: Performance of the sparse Merkle tree.

height W	proof size	verification	update most recent proof	update local proof
30	0.341KB	0.055 ms	0.071ms	0.005ms
40	0.357KB	0.063 ms	0.081ms	0.011ms
50	0.363KB	0.074 ms	0.1ms	0.014ms

6.2 Evaluation of our new vector commitment

In this section, we evaluate the performance of our new vector commitment scheme. We also justify why we did not use the similarly-performant, from an asymptotic perspective (see Table 1 in the introduction), lattice-based vector commitment from [24] for the EDRAX implementation, by directly comparing with [24] and showing our vector commitment is much more practical.

One-time setup. The costliest part of our vector commitment scheme is the one-time setup to generate the prover, update and verification keys. Such an expensive setup is not required by the lattice-based scheme whose setup involves just a constant-time sampling of two lattice-based hash functions. Results for our scheme are provided in Table 3. As shown in the table, it takes 2,301s to generate the keys for a vector of 32 million elements, which is close to the current number of accounts in Ethereum [12]. Most of the time in key generation is spent on computing exponentiations in the base group, which can be easily parallelized. The key generation time is reduced by $7.6\times$ to 303s with EC2 16 virtual cores. The running time scales linearly with the number of elements in the vector, and it takes 42,508s for $\ell = 32$, i.e., 4 billion elements, with parallelization.

Moreover, our implementation stores the prover key on disk, so it can scale to a larger ℓ as long as the disk size is larger than the proving key size. The overhead for disk I/O is already included in the key generation time reported in Table 3. Finally, the verification key size and the update key size for one element are less than 1KB for $\ell \leq 32$ and grows logarithmically with the number of elements in the vector. Note the EDRAX nodes do not need to store the prover key, just the update key and verification key.

Proof size and verification time. Figure 3 shows the proof size and the verification time of our vector commitment scheme and the lattice-based scheme in [24]. Though the asymptotics are the same in the two schemes, the concrete performance in our scheme is better. The proof size is less than 1KB for $L \leq 32$ (same as the size of prk and vrk) in our scheme, while it is 62–78KB in the lattice-based scheme.

Table 3: One-time setup in our vector commitment scheme for various values of ℓ. The notation * means estimation due to long running times.

ℓ	KeyGen (single)	KeyGen (multi)	memory usage	prk size	vrk & upk size
25	2,301s	303s	14GB	4.1GB	0.78KB
26	4,611s	609s	25GB	8.2GB	0.81KB
28	18,756 s	2,738s	25GB	16.3GB	0.87KB
30	74,000s*	9,941s	25GB	65GB	0.93KB
32	295,000s*	42,508s	25GB	263GB	0.99KB

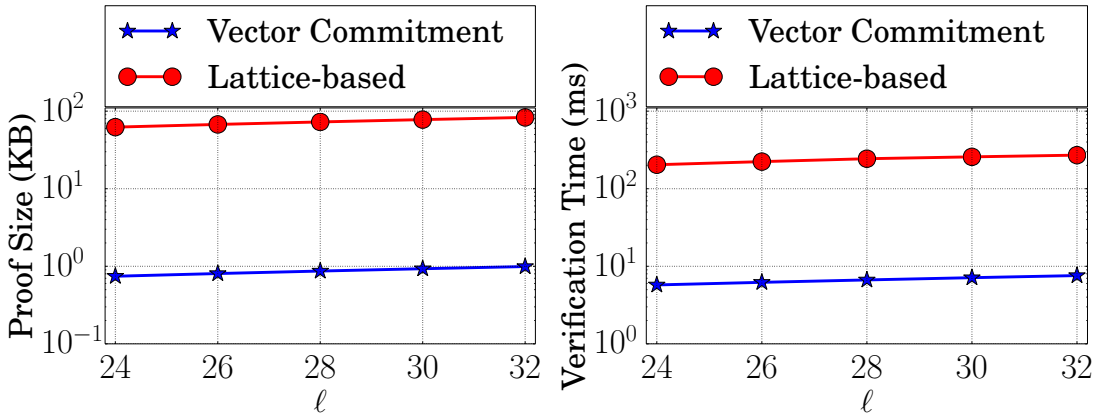

Figure 3: Proof size and verification time for our vector commitment scheme and the lattice scheme [24].

As in EDRAX we need to attach a proof to each transaction, the transaction size would increase by two orders of magnitude using the lattice-based scheme, which is impractical for our applications. In addition, the verification time for $\ell = 25$ is 210ms in the lattice-based scheme, while it takes 6ms in our scheme, $35\times$ faster than the lattice-based scheme.

Digest update and proof synchronization. Figure 4 shows the time to update the digest and to perform proof synchronization in the two schemes. It takes 8.3μs to update the digest and 15.1μs (amortized) to synchronize the proof of one element, regardless of the value of ℓ. In the lattice-based scheme, both updating the digest and updating the proof takes around 790ms for $\ell = 25$, and the time grows logarithmically with the number of elements in the vector. Note that in the lattice-based scheme, one can also precompute the "partial labels" ([24, Definition 14], similar to the update key). The time to update the digest and the proof can be reduced significantly (both $O(1)$ asymptotically, and tens of microseconds concretely), but it would require an additional setup phase that is linear to the number of elements in the vector and is slower than our key generation time. We omit the comparison to this alternative since the proof size would still be the bottleneck.

6.3 Stateless validation vs. stateful validation

Using the primitives evaluated in Sections 6.1 and 6.2, we simulate stateless block processing for both UTXO-based and account-based EDRAX. Then we compare our results with simulated stateful clients. In all the experiments in this section we do not measure time spent on digital signature verification as well

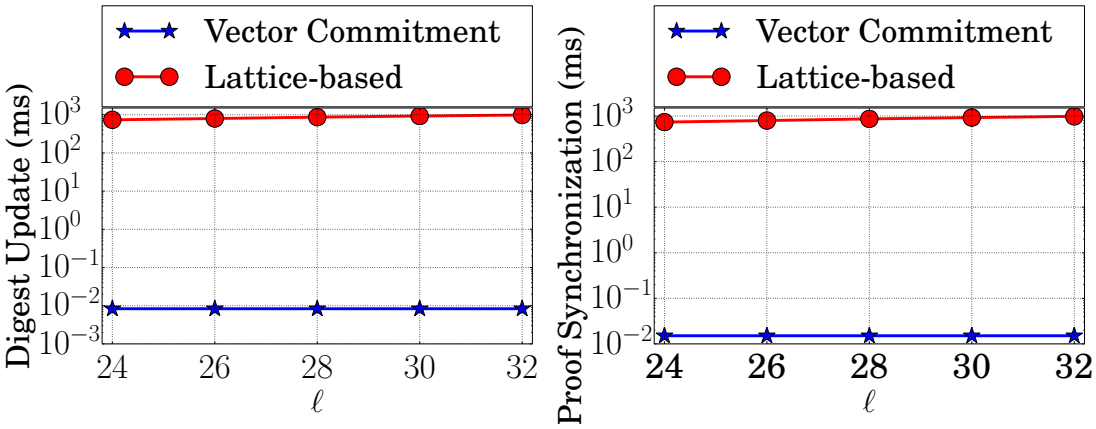

Figure 4: Time for updating the digest and for proof synchronization of vector commitment vs. the lattice scheme [24].

as other stateless checks.

Blocks and transactions. In our simulation, each block contains 1,000 transactions. Recall that in the UTXO-based model, a SPEND transaction has the following fields: amount of the transaction, sender's and recipient's public keys, sender's unspent output index and balance, and a proof of the output presence in the UTXO set. In the account-based model, a SPEND transaction contains amount of the transaction, sender's and recipient's public keys, the current balance in sender's account, the proof of the balance and the sender's and recipient's update keys.

Simulating stateful clients. We simulate the cost to process blocks for stateful clients storing the UTXO set or account balances on disk. We use the LevelDB database with disk persistence, under default settings and single synchronous batch write per block—note that LevelDB is used in Bitcoin's core reference implementation to store the UTXO. For the UTXO-based model, we perform 1 read and 1 delete for the same key, and 2 writes with two new consecutive keys for each transaction (to read an output of the sender, remove it from the UTXO, and create two new outputs). For the account-based model, we perform 2 reads and 2 writes on two randomly chosen keys (to get sender and recipient balances and to update them). As the time to access the database varies a lot because of the disk I/O, we report the average of processing 10,000 blocks in the figures for the stateful client.

UTXO-based model. The comparison of the performance between EDRAX and stateful clients in the UTXO-based model is shown in Figure 5 (left). As 2^W is an upper bound on the total number of transaction outputs in the system, we set $W = 40$, which is enough to support even more transaction outputs than those that have ever been generated in Bitcoin as of now. With this fixed W, the time to process one block does not change with the number of transactions in the UTXO. As shown in the figure, it takes 534ms to process one block in EDRAX, while it takes 213ms for a stateful client with 2^{26} unspent transactions in the UTXO, which is the similar to the scale of the UTXO in Bitcoin now. As the size of the UTXO grows, EDRAX starts to outperform the stateful client. For example, it takes 14.18s for a stateful client to process one block when the UTXO contains 1 billion transaction outputs, $26.5\times$ slower than EDRAX.

Note that the processing time of the stateful client increases dramatically when the UTXO size goes from 2^{27} to 2^{28}. This is because the cache in LevelDB and the operating system seem to be becoming much less effective at these scales. When the number of transactions is less than or equal to 2^{27}, we observe that most blocks are processed within hundreds of milliseconds, while some blocks take significantly longer time, 1-4 seconds, due to cache misses and disk accesses. When the number of transactions is equal to or larger than 2^{28}, most elements in the database are not cached, and the processing time for

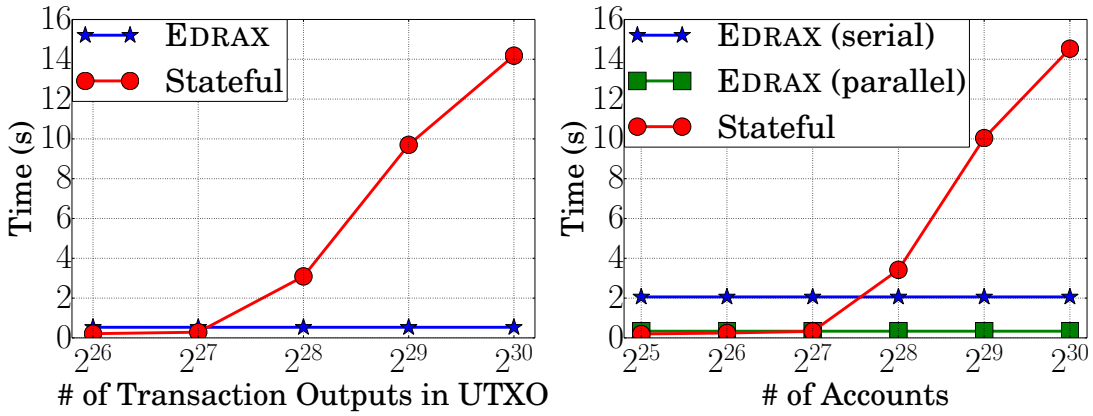

Figure 5: EDRAX vs. stateful (on-disk) validation. Left: UTXO-based model ($W = 40$); right: account-based model ($\ell = 30$).

every block in our experiment is higher than 1s, which results in a much longer preprocessing time than the time required for 2^{27} transactions.

Account-based model. Figure 5 (right) shows the comparison in the account-based model. Again as 2^{ℓ} is an upper bound on the number of accounts in the system, we fix $\ell = 30$, supporting more accounts than that currently in Ethereum. It takes 2,059ms to process one block in EDRAX, while it takes 197ms for a stateful client with 2^{25} accounts, the current scale in Ethereum. As the number of accounts grows, the performance of EDRAX is comparable or faster than the stateful client. For example, with 2^{28} accounts, the stateful client requires 3.42s to process one block, and with 2^{30} accounts, the stateful client takes 14.53s, 7.3× slower than EDRAX. Moreover, most time in EDRAX is spent on verifying proofs of our vector commitment scheme, which can be easily parallelized. With 16 Amazon Ec2 virtual cores, the processing time is reduced by 6.1× to 337ms. On the contrary, we do not observe any speedup when parallelizing the stateful client, as the bottleneck is on disk I/O. Therefore, the performance of the parallel of EDRAX is similar the stateful client with 2^{25} accounts, and 43× faster with 2^{30} accounts.

Main finding. The experiments above indicate that EDRAX not only removes the large storage requirement on the client, but also achieves comparable processing time under the current scale of cryptocurrencies, and will be much faster when the scale increases, due expensive I/Os of the database maintained by the stateful client.

Block size and proof synchronization. Finally, though EDRAX significantly reduces the storage of a client, it also introduces overhead on the block size and proof updates. In the UTXO-based model, an additional sparse Merkle tree proof is included for each transaction output, which is 357 Bytes for $W = 40$. In the account-based model, the client needs to attach a proof, the sender's and recipient's update keys, which is 2.86KB in total for $\ell = 30$. Also the clients must synchronize their proofs using the information on the blockchain. It takes 11ms to synchronize one's proof per block with 1,000 transactions in the UTXO model, and 30.2ms in the account-based model. We believe these overheads are reasonable in practice, given the significant savings on storage and processing time.

7 Conclusions and discussion

In this paper we presented EDRAX, an architecture (and two different implementations thereof) for stateless transaction validation in cryptocurrencies. Our concrete implementations are first steps—in the future

more practical authenticated data structures or vector commitment schemes can be used as a drop-in replacement, leading to even better performance. Here we outline some directions for future research and some observations.

Authenticated red-black trees instead of sparse Merkle trees. In UTXO-based version of EDRAX, presented in Section 3, we used sparse Merkle trees as our basic authenticated data structure. Sparse Merkle trees have fixed structure, allowing very flexible updates, especially when not all the tree is stored locally. However, they have large proofs, $O(W)$, *irrespective of the current size of the UTXO*. We believe one can implement the same algorithms for an authenticated red-black tree (or in general for an authenticated balanced tree) and maintain a proof size that is always $O(\log n)$ where n is the *current size of the UTXO*. We note here that while authenticated red black trees can be derived generically using the work of Miller et al. [18], it is not clear how these algorithms can be used in EDRAX setting, since updates must be performed without storing the whole authenticated tree, but *just the most recent proof*. Similar algorithms, however, have been studied by Papamanthou and Tamassia [22], for the case of authenticated skip lists.

Local proof synchronization by miners. In the general description of EDRAX we assume (for ease of exposition) that a client wishing to perform an EDRAX transaction synchronizes her proof to be valid with respect to the digest stored in the last (confirmed) block t of the blockchain. However, when the transaction arrives (or when a miner wishes to include it in the next block), block t might not be the last anymore (e.g., the last block might be $t + 5$) and therefore the proof originally computed by the client will not be valid. This is not a problem since miners can easily update proofs and produce valid proofs for $t + 5$ by processing transactions that took place between t and $t + 5$ exactly in the same way that clients would do it (recall algorithms SYNCHRONIZEPROOF and DELTAPOLYNOMIALS do not require any secrets for their execution and therefore can be easily executed by miners).

Introducing proof-serving nodes. One of the main differences of the EDRAX architecture with current cryptocurrencies is the fact that clients must perform local proof synchronization (with the blockchain) before they post their transactions. This introduces a moderate cost for the clients that did not exist before, and also changes the user experience. To alleviate this cost, we can extend the EDRAX architecture to contain *proof-serving nodes* (that can run for example on an Amazon machine) which *do not participate in the blockchain protocol*. These nodes are responsible for storing and serving *up-to-date* proofs for all the nodes of the cryptocurrency (and can be incentivized through some form of reward). Note that there is no requirement that they serve correct proofs. In particular, a client that wants to post a transaction can request his proof from a proof-serving node. If this proof is valid (which he can check by using the digest from the blockchain), then he can use it, otherwise he can always use the "default" setting and synchronize his proof by using the algorithms we presented before.

Supporting smart contracts in the stateless setting. We can envision an extension of EDRAX to support stateless verification of smart contracts as well. Recall that in the smart contract setting, the flow of money will depend of the execution of some contract code on the current contract state, which is updated after the contract execution. Therefore for Alice to post a contract-triggering transaction she must provide a proof of correctness of the current contract state for EDRAX nodes to execute on. For that, we can again use a Merkle tree whose leaves store hashes of the contract state and have clients provide the respective Merkle proofs. Two challenges that arise in this setting are (i) who is storing the contract state since any client can post transactions that will trigger a contract execution; (ii) how to avoid including the contract state as part of the transaction (the contract state might be too large). For both of these challeges, proof-serving nodes (as introduced above) and SNARKs (e.g., [23]) might help. A complete treatment of these challenges, however, is left as future work.

Adding privacy to account-based model. We note here that the problem of adding privacy in the presented models (UTXO and account-based) is only partially solved, *even in the stateful setting*. In particular, while Zcash [8] solves the problem of privacy in the UTXO-based model, there is no protocol

(to the best of our knowledge) for the account-based model. Of course one could consider using a zero-knowledge version of our new vector commitment to add privacy in the account-based model. However, even if we could manage to add zero-knowledge on our construction, other more fundamental challenges remain. In particular, note that in order to update the local proof, one needs to know index u of the account whose balance changes. However this is precisely what a privacy-preserving account-based cryptocurrency must hide! We leave it as an open problem to design a privacy-preserving cryptocurrency (either stateful or stateless) in the account-based model.

References

[1] Ate pairing. `https://github.com/herumi/ate-pairing`.

[2] Ethereum white paper," https://github.com/ethereum/wiki/wiki/white-paper.

[3] The GNU multiple precision arithmetic library. `https://gmplib.org/`.

[4] https://github.com/citp/blocksci.

[5] LevelDB. `https://github.com/google/leveldb`.

[6] Scrypto. `https://github.com/input-output-hk/scrypto`.

[7] Utxo uh-oh...," http://gavinandresen.ninja/utxo-uhoh.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474, 2014.

[9] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 56–73. Springer, 2004.

[10] Sean Bowe, Ariel Gabizon, and Matthew Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *International Conference on Financial Cryptography and Data Security*. Springer, 2018.

[11] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, pages 55–72, 2013.

[12] Ethereum Unique Address Growth Chart. https://etherscan.io/chart/address.

[13] https://etherscan.io/chart/address.

[14] https://ethresear.ch/t/the-stateless-client concept/172.

[15] https://medium.com/cybermiles/diving-into-ethereums-world-state c893102030ed.

[16] https://statoshi.info/dashboard/db/unspent-transaction-output set.

[17] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 218–238, 1989.

[18] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*.

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

[20] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In *Theory of Cryptography*, pages 222–242. Springer, 2013.

[21] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming authenticated data structures. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 353–370, 2013.

[22] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *Information and Communications Security, 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceedings*, pages 1–15, 2007.

[23] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252, 2013.

[24] Yi Qian, Yupeng Zhang, Xi Chen, and Charalampos Papamanthou. Streaming authenticated data structures: Abstraction and implementation. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*, pages 129–139, 2014.

[25] Roberto Tamassia. Authenticated data structures. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 2–5, 2003.

[26] 2016. https://blog. ethereum.org/2016/09/22/transaction-spam-attack-next-steps/ Vitalik Buterin. Transaction spam attack: Next steps.

[27] Joachim Zahnentferner. Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. Cryptology ePrint Archive, Report 2018/262, 2018. `https://eprint.iacr.org/2018/262`.

[28] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 203–220.

[29] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary sql queries over dynamic outsourced databases. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 863–880. IEEE, 2017.

Appendix

Assumption

Assumption 1 ([9] q-Strong Bilinear Diffie-Hellman (q-SBDH)). *For any* PPT *adversary $\mathcal{A}$, the following probability is negligible:*

$$\Pr\left[\begin{array}{c}(p,\mathbb{G},\mathbb{G}_T,e,g)\leftarrow \mathsf{Gen}(1^\lambda);\\ s\xleftarrow{R}\mathbb{Z}_p^*;\\ \sigma=((p,\mathbb{G},\mathbb{G}_T,e,g),g^s,\ldots,g^{s^q}):\\ (x,e(g,g)^{\frac{1}{s+x}})\leftarrow\mathcal{A}(1^\lambda,\sigma)\end{array}\right].$$

Correctness definition of a vector commitment scheme

Definition 3 (Correctness of vector commitment scheme). *We say that a vector commitment scheme is correct, if for all $\lambda\in\mathbb{N}$ and $n=\mathsf{poly}(\lambda)$, for all public keys $(\mathsf{prk},\mathsf{vrk},\{\mathsf{upk}_u\})\leftarrow\mathsf{KeyGen}(1^\lambda,n)$, for all vectors $\mathbf{a}=[a_0,\ldots,a_{n-1}]$, if dig is a commitment to vector $\mathbf{a}$ computed by $\mathsf{Setup}(\mathbf{a},\mathsf{prk})$ and π_i, for all i, is a proof generated $\mathsf{Prove}(i,\mathbf{a},\mathsf{prk})$, then for a polynomial number of updates (u,δ) if dig and π_i, for all i, are produced by calls to $\mathsf{UpdateDigest}(\mathsf{dig},u,\delta,\mathsf{upk}_u)$ and $\mathsf{UpdateProof}(\pi_i,u,\delta,\mathsf{upk}_u)$ respectively then for all i it is $\Pr[1\leftarrow\mathsf{Verify}(\mathsf{dig},i,a_i,\pi_i,\mathsf{vrk})]=1$, where a_i is the value at index i after all updates took place.*