Designated-Verifier Dynamic zk-SNARKs with Applications to Dynamic Proofs of Index

Weijie Wang
Yale University
New Haven, CT, USA
weijie.wang@yale.edu

Charalampos Papamanthou
Yale University
New Haven, CT, USA
charalampos.papamanthou@yale.edu

Shravan Srinivasan
Lagrange Labs
New York, NY, USA
ShravanSrinivasan1@gmail.com

Dimitrios Papadopoulos
Hong Kong University of Science and Technology
Hong Kong, China
dipapado@cse.ust.hk

Abstract

Recently, the notion of dynamic zk-SNARKs was introduced. A dynamic zk-SNARK augments a standard zk-SNARK with an efficient update algorithm. Given a valid source statement-witness pair (x, w) together with a verifying proof p, and a valid target statement-witness pair (x', w'), the update algorithm outputs a verifying proof p' for (x', w'). Crucially, p' is not recomputed from scratch; instead, the update algorithm takes time roughly proportional to the Hamming distance between (x, w) and (x', w'), analogous to how dynamic data structures update the result of a computation after a small change.

In this paper, we initiate the study of designated-verifier dynamic zk-SNARKs: dynamic zk-SNARKs in which only a designated verifier, holding secret verification state, can be convinced by a proof. Following recent advances in designated-verifier zk-SNARKs—such as efficient post-quantum designated verifier SNARKs (CCS 2021) and designated verifier SNARKs with very small proofs (CRYPTO 2025)—we construct a designated-verifier dynamic zk-SNARK with $O(\log n)$ update time, constant proof size, and concrete efficiency. Our construction significantly outperforms Dynalog (both asymptotically and concretely), the only publicly verifiable dynamic zk-SNARK with polylogarithmic update time (Wang et al., 2024).

The concrete efficiency of our construction enables, for the first time, an efficient implementation of a dynamic proof of index: Given a digest d of an arbitrary set and a digest d' of its sorted index (e.g., binary search tree), we produce a SNARK proof certifying the consistency of d and d'. More importantly, this proof can be updated in sublinear time when the underlying set changes—for example, when an element is modified or inserted, potentially altering the sorted order. We demonstrate applications of designated-verifier dynamic proofs of index to verifiable dynamic database outsourcing, where a client outsources a database and later maintains verifiable indices for efficient query answering, even under arbitrary database updates.

Keywords

Zero-knowledge proofs

1 Introduction

Given an NP language L, a zero-knowledge proof is a protocol between two parties, a prover and a verifier, in which the prover provides a publicly-verifiable proof π that a statement $x \in L$, without revealing anything about the respective NP witness w. Since they were first introduced by Goldwasser, Micali and Rackoff [15], zero-knowledge proofs, and in particular succinct, non-interactive arguments of knowledge also known as zk-SNARKs (e.g., [17]), have found many applications, such as in privacy-preserving blockchains and cryptocurrencies, zk-rollups, privacy-preserving authentication and verifiable outsourced computation, among others.

Dynamic zk-SNARKs. A recently-introduced direction in zk-SNARKs is that of *dynamic* zk-SNARKs [27], which is the focus of our work. A dynamic zk-SNARK is a zk-SNARK that has an additional update algorithm: The update algorithm takes as input a valid source statement-witness pair (x, w) along with its proof π and a target statement-witness pair (x', w'), and outputs the proof π' for (x', w'). Importantly, updating the proof with a dynamic SNARK takes sublinear time, ideally proportional to the Hamming distance between the source and target statement-witness pair. In other words, a dynamic zk-SNARK is to SNARKs what a data structure is to traditional native computation. Dynamic zk-SNARKs have been shown to yield recursion-free IVC and sparse zk-SNARKs [27] and can be applied to compute proofs faster in any setting that involves maintaining zero-knowledge proofs over rapidly-evolving data, such as in zk-coprocessors, e.g., [23].

Designated-verifier dynamic zk-SNARKs. In this paper we study the problem of building dynamic zk-SNARKs in the weaker *designated-verifier setting*, where the produced zk-SNARK proof π is *not* publicly-verifiable: It can only be verified by a designated party who keeps a secret as part of a local state; If this secret leaks, the zk-SNARK is no longer secure. Designated verifier zk-SNARKs have found important applications in verified outsourced data and computation, e.g., [29], where a client stores private data with an untrusted server and wishes to later verify that a certain computation has been performed correctly, as well as in private authentication [25] where a server authenticating users, serving as the zk-verifier, can afford storing some secret verification key for every user.

Recent works proposed (static) designated-verifier zk-SNARKs with higher levels of concrete efficiency (a natural tradeoff one would expect). In particular, in CCS 2021, Ishai et al. [18] proposed a post-quantum designated-verifier zk-SNARK that is fast despite using lattice assumptions and in CRYPTO 2025 Arnon et al. [1] proposed a designated-verifier SNARK whose proof consists of

a single proof element! More theoretical constructions (such as compilers from simpler primitives to designated-verifier SNARKs) have been also proposed by Bitansky et al. [6] and Li and Zhang [21].

1.1 Delphus: An efficient designated-verifier dynamic zk-SNARK

Our main contribution is Delphus, an efficient designated-verifier dynamic zk-SNARK. To the best of our knowledge, Delphus has the best asymptotic update time among all existing dynamic zk-SNARKs. See Table 1. In particular, compared to the only-known polylog-update dynamic zk-SNARK, Dynalog [27], Delphus improves the update time to $O(\log n)$, while achieving constant-size proofs! When compared to constant-size proof dynamic zk-SNARKs (but with no polylog updates), e.g., Dynaverse [27] and Dynark [28], Delphus achieves faster update time.

Concrete efficiency of Delphus. As we show in our evaluation section, not only does Delphus achieve excellent asymptotic complexities, it is also concretely efficient. Our open-source implementation shows that the time required for an update when $n = 2^{24}$ is less than 30ms. In addition, by counting the precise number of operations (such as scalar multiplications in $\mathbb{G}_1$ and $\mathbb{G}_2$), we find that Delphus's update is approximately 10 times faster than Dynark's update and 40 times faster than Dynalog's update (We only compare with Dynark since its concrete cost is better than Dynaverse due to Dynaverse being universal, as opposed to Dynark.) For larger statements, the improvement approaches two orders of magnitude. We however note that Delphus's proof, while being much smaller than Dynalog's proof, is approximately 2.5 times larger than Dynark's proof (480 bytes v. 192 bytes). This is due to the fact that our proof requires a polynomial commitment in the form of an element in the target group $\mathbb{G}_T$, due to the designated verifier setting.

Technical highlights of Delphus. Recall that in order to build a dynamic zk-SNARK, one commits, for example, using KZG commitments [19], to a left-inputs vector $\boldsymbol{a}$, to a right-inputs vector $\boldsymbol{b}$ and to an outputs vector $\boldsymbol{c}$ (all vectors being of size n), and one of the main goals of a zk-SNARK system is to show that

$$\boldsymbol{a}[i] \cdot \boldsymbol{b}[i] = \boldsymbol{c}[i] \text{ for all } i = 0, \ldots, n-1 .$$

In polynomial-based SNARKs (like Groth16 [17] and PLONK [14]), this checks if $a(X) \cdot b(X) - c(X)$ is divisible by a vanishing polynomial $Z(X)$, yielding a quotient polynomial $q(X)$.

One fundamental bottleneck in dynamic SNARKs is that a local change to a single index i changes the coefficients of $q(X)$ globally, requiring expensive $O(n)$ or $O(n \log n)$ recomputation. In Section 3, we address this by decomposing $q(X)$ into a component $q_c(X)$ (dependent linearly on c) and a cross-term component $Q(X)$ (dependent on $O(n^2)$ cross terms). We observe that while computing the commitment $[Q(\tau)]_1$ is expensive, we can efficiently maintain its image in the target group, $[Q(\tau)]_T$. We construct an auxiliary binary tree over the commitments of $\boldsymbol{a}$ and $\boldsymbol{b}$ that allows us to update $[Q(\tau)]_T$ using only $O(\log n)$ pairings and group operations.

Finally, in Section 4, we integrate this approach into the Groth16 protocol [17]. We partition the Groth16 proof terms into those that are linearly updatable and those involving the quotient polynomial $q(X)$, replacing the latter with our previous decomposition $q(X) = q_c(X) + Q(X)$. This yields the first dynamic Groth16 with $O(\log n)$ update time.

1.2 Proof of index validity

To demonstrate the application of dynamic zk-SNARKs in verifiable databases (Section 6), we construct circuits in Section 5 that establish the validity of dynamic indexes. Specifically, this circuit takes two inputs: the digest of a data table (modeled as a multiset) and the digest of an index (modeled as a binary search tree or its generalizations). Its purpose is to verify the consistency between the table and the index—guaranteeing that the index is indeed a correctly organized representation of the table. To fully leverage the efficiency of dynamic zk-SNARKs, these circuits must be *locally updatable*: a single update to the circuit's input (e.g., a change in the index) must affect only a polylogarithmic number of wire values within the circuit.

The primary challenge in constructing such a circuit lies in validating the structure of the index itself. Consider the case where the index is a Binary Search Tree (BST). The circuit must verify that the tree is well-organized according to a specified key. However, maintaining this verification efficiently is non-trivial, since rebalancing the tree may connect nodes whose values were widely separated in the previous circuit state. We address this by proposing two constructions, both of which reduce the problem to checking the equality of two multisets that define the connectivity of the tree nodes. The first construction utilizes a cryptographic *multiset hash function* [4, 9] to verify set equality. The second construction employs a *sorting circuit* (specifically a variant of the Beneš network [5]) to prove that the sorted list is a permutation of the input, checking if the two sequences are identical. Crucially, we show that both constructions satisfy our requirement for efficient updates.

1.3 Application: Index-agnostic verifiable databases

In Section 6, we apply our designated-verifier dynamic zk-SNARK and our dynamic proof of index validity to construct an efficient Verifiable Database (VDB) [7, 23, 24, 29, 30]. In a VDB system, a resource-constrained client outsources a database to an untrusted server. The client wishes to perform efficient queries (such as range queries) and updates while maintaining a small local digest. This system naturally aligns with the designated-verifier model.

Existing VDB solutions often struggle to balance efficient query performance with efficient update handling. Maintaining a sorted index—such as B-trees—enables range queries returning k rows in $O(\log n + k)$ time, but it raises a fundamental challenge regarding how the index should be stored and managed. Clearly, the client cannot maintain the entire index locally. One possible approach is for the client to store a cryptographic digest of the index; however, this introduces substantial overhead. Specifically, even a single insertion or modification requires the client to verify and update the digests of all affected indexes stored locally. Moreover, in scenarios involving continuous or streaming updates, the client must still interact[1] with the server to update these digests, even if no queries are

[1]This interaction is not fundamentally unavoidable: for example, if the client maintains a simple multiplicative hash [4, 9] of the database, it can update this local state independently without communicating with the server.

Table 1: Asymptotic comparison of Delphus with all existing dynamic zk-SNARKs—Dynalog, Dynaverse **and** Dynark**. In the table below, $\mathcal{G}$ is the key generation algorithm, $\mathcal{P}$ is the prove algorithm, $\mathcal{V}$ is the verification algorithm, $|\pi|$ is the proof size, $|\mathsf{pk}|$ is the prover key size, $|\mathsf{vk}|$ is the verification key size, k is the number of updates between source/target statements and $n = |\mathbb{x}| + |\mathbb{w}|$.**

scheme	$\mathcal{G}$	$\mathcal{P}$	$\mathcal{U}$	$\mathcal{V}$	$\lvert\pi\rvert$	$\lvert\mathsf{pk}\rvert$	$\lvert\mathsf{vk}\rvert$	publicly-verifiable
Delphus	$n \log n$	$n \log n$	$k \cdot \log n$	1	1	$n \log n$	1	NO
Dynalog [27]	$n \log^3 n$	$n \log^2 n$	$k \cdot \log^3 n$	$\log^3 n$	$\log^3 n$	$n \log^3 n$	$\log^2 n$	YES
Dynaverse [27] and Dynark [28]	n	$n \log n$	$k \cdot \sqrt{n \log n}$	1	1	n	1	YES

issued during that period. This requirement significantly increases client-side communication and computation costs.

We propose an *index-agnostic* VDB where the integrity of the auxiliary index is certified by a persistent, dynamic SNARK proof. *In this system, the client only needs to verify the integrity of an index relevant to the current query.* When the client issues an UPDATE command, the server updates the database and the corresponding SNARK proofs using the Delphus update algorithm. The efficiency of Delphus ensures that the server's overhead is minimal (almost linear in the number of updates performed), significantly outperforming prior approaches that rely on static SNARKs.

2 Preliminaries

Sets and vectors. Let $[n] = \{1, \ldots, n\}$ and $[m, n] = \{m, \ldots, n\}$. We use bold letters to denote vectors of field elements (indexed from 0), e.g., bold small letters for vectors of field elements, such as $\boldsymbol{r} = (r_0, \ldots, r_{\ell-1}) \in \mathbb{F}^\ell$.

Pairings. Let $\mathbb{G}_1, \mathbb{G}_2$ denote cyclic groups of prime order p with generators G_1, G_2 respectively, and a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Here, the target group $\mathbb{G}_T$ is also of order p with generator $G_T = e(G_1, G_2)$. Using additive notation, the following hold for $e(\cdot, \cdot)$: (i) $\forall U \in \mathbb{G}_1, \forall W \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$, $e(a \cdot U, b \cdot W) = (ab) \cdot e(U, W)$. (ii) $e(U, H) + e(V, H) = e(U + V, H)$, $\forall U, V \in \mathbb{G}_1, H \in \mathbb{G}_2$. We use $\mathsf{pp}_{\mathsf{bl}} := (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G_1, G_2) \leftarrow \mathsf{GroupGen}_{\mathsf{bl}}(1^\lambda)$ to denote the pairing parameters.

Univariate Lagrange polynomials. Let $n = 2^\ell$ and ω be a primitive n-th root of unity. The Lagrange polynomial of degree $n-1$ for $j \in [0, n-1]$ is defined as

$$\mathcal{L}_j(X) = \prod_{\substack{0 \le i < n \\ i \ne j}} \frac{X - \omega^i}{\omega^j - \omega^i} = \frac{\omega^j}{n} \prod_{\substack{0 \le i < n \\ i \ne j}} (X - \omega^i) = \frac{\omega^j (X^n - 1)}{n(X - \omega^j)},$$

Note that $\mathcal{L}_j(\omega^j) = 1$, $\mathcal{L}_j(X) - 1$ has a root $X = \omega^j$, so $X - \omega^j \mid \mathcal{L}_j(X) - 1$. Denote $\mathcal{T}_j(X) = \frac{\omega^j (\mathcal{L}_j(X) - 1)}{n(X - \omega^j)}$.

Designated-verifier dynamic zk-SNARKs. As we mentioned, our definition of designated-verifier dynamic zk-SNARKs requires that the adversary does not have access to the secret verification key vk. We capture this in the following definition, which is a direct adaptation of the dynamic zk-SNARKs definition by Wang et al. [27].

Definition 2.1 (Designated-Verifier Dynamic zk-SNARKs.). *A designated-verifier dynamic zero knowledge succinct non-interactive argument of knowledge (dynamic zk-SNARK) for indexed relation $\mathcal{R}$ is a tuple of the following PPT algorithms* $\mathsf{DS} = (\mathcal{G}, \mathcal{P}, \mathcal{U}, \mathcal{V})$*:*

- $\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{upk}, \mathsf{vk})$: *Given 1^λ and index $\mathbb{i}$, outputs prover key* pk*, update key* upk *and verifier key* vk*.*
- $\mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}) \to (\pi, \mathsf{aux})$: *Given prover key* pk*, instance $\mathbb{x}$, and witness $\mathbb{w}$, outputs π and auxiliary information* aux*.*
- $\mathcal{U}(\mathsf{upk}, \mathbb{x}', \mathbb{w}', \mathbb{x}, \mathbb{w}, \pi, \mathsf{aux}) \to (\pi', \mathsf{aux}')$: *Given update key* upk*, new instance $\mathbb{x}'$, new witness $\mathbb{w}'$, the previous proof π for instance $\mathbb{x}$ and witness $\mathbb{w}$ and auxiliary information* aux*, outputs a new proof π' for $\mathbb{x}'$ and $\mathbb{w}'$, and new auxiliary information* aux′*.*
- $\mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to 0/1$: *Given verifier key* vk*, instance $\mathbb{x}$, and a proof π, outputs accept or reject.*

A designated-verifier dynamic zk-SNARK DS *should have polylog-sized proofs and satisfy the following properties.*

- Updatability*: We say that* DS *satisfies* updatability *if there is a function $f(|\mathbb{x}| + |\mathbb{w}|) = o(|\mathbb{x}| + |\mathbb{w}|)$ such that algorithm $\mathcal{U}(\mathsf{upk}, \mathbb{x}', \mathbb{w}', \mathbb{x}, \mathbb{w}, \pi, \mathsf{aux})$ runs in time $O(k \cdot f(|\mathbb{x}| + |\mathbb{w}|))$, where k is the Hamming distance of vectors $\mathbb{x}||\mathbb{w}$ and $\mathbb{x}'||\mathbb{w}'$.*[2]
- Completeness*: Let* $(\mathsf{pk}, \mathsf{upk}, \mathsf{vk}) \leftarrow \mathcal{G}(1^\lambda, \mathbb{i})$*. We say that* DS *satisfies* completeness *if for any $\mathbb{i}$, for any $(\mathbb{i}, \mathbb{x}_0, \mathbb{w}_0) \in \mathcal{R}, \ldots, (\mathbb{i}, \mathbb{x}_\ell, \mathbb{w}_\ell) \in \mathcal{R}$, if*

$$\mathcal{P}(\mathsf{pk}, \mathbb{x}_0, \mathbb{w}_0) \to (\pi_0, \mathsf{aux}_0), \text{ and}$$

$$\mathcal{U}(\mathsf{upk}, \mathbb{x}_{i+1}, \mathbb{w}_{i+1}, \mathbb{x}_i, \mathbb{w}_i, \pi_i, \mathsf{aux}_i) \to (\pi_{i+1}, \mathsf{aux}_{i+1}),$$

for $i = 0, \ldots, \ell - 1$, then $\mathcal{V}(\mathsf{vk}, \mathbb{x}_\ell, \pi_\ell) \to 1$.
- Knowledge Soundness*: We say that* DS *satisfies* knowledge soundness *if for any PPT adversary $\mathcal{A}$ and for any $\mathbb{i}$ there exists a PPT extractor $\mathcal{E}_\mathcal{A}$ such that*

$$\Pr\left[\begin{array}{l|l} \mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to 1 \wedge & \mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{upk}, \mathsf{vk}); \\ (\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R} & ((\mathbb{x}, \pi); \mathbb{w}) \leftarrow (\mathcal{A} || \mathcal{E}_\mathcal{A})(\mathsf{pk}, \mathsf{upk}) \end{array}\right]$$

is negligible.
- Zero Knowledge*: Fix any $\mathbb{i}$, $(\mathbb{i}, \mathbb{x}_0, \mathbb{w}_0) \in \mathcal{R}, \ldots, (\mathbb{i}, \mathbb{x}_\ell, \mathbb{w}_\ell) \in \mathcal{R}$ for some polynomially bounded ℓ. Let $\mathcal{D}$ be the distribution of $(\mathsf{pk}, \mathsf{upk}, \mathsf{vk}, \pi_0, \ldots, \pi_\ell)$ as output by the experiment below.*
  (1) $\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{upk}, \mathsf{vk})$.
  (2) $\mathcal{P}(\mathsf{pk}, \mathbb{x}_0, \mathbb{w}_0) \to (\pi_0, \mathsf{aux}_0)$.
  (3) $\mathcal{U}(\mathsf{upk}, \mathbb{x}_{i+1}, \mathbb{w}_{i+1}, \mathbb{x}_i, \mathbb{w}_i, \pi_i, \mathsf{aux}_i) \to (\pi_{i+1}, \mathsf{aux}_{i+1})$*, for all $i = 0, \ldots, \ell - 1$.*
  We say that DS *satisfies* zero-knowledge *if there exists a PPT simulator $\mathcal{S}$ such that, for every $\mathbb{i}$ and $(\mathbb{i}, \mathbb{x}_i, \mathbb{w}_i) \in \mathcal{R}$ as above, the distribution $\tilde{\mathcal{D}}$ of $(\mathsf{pk}, \mathsf{upk}, \mathsf{vk}, \tilde{\pi}_0, \ldots, \tilde{\pi}_\ell)$ output by the following experiment is computationally indistinguishable from $\mathcal{D}$.*

[2] Note that for $k = o\left(T(\mathcal{P})/f(|\mathbb{x}| + |\mathbb{w}|)\right)$, where $T(\mathcal{P})$ is the prover's runtime, this is $o(T(\mathcal{P}))$, as desired. Besides, for simplicity of notation, we write $\mathbb{x}', \mathbb{w}', \mathbb{x}, \mathbb{w}$ as explicit inputs of $\mathcal{U}$ but, in practice, it suffices to receive the old and new instance/witness elements at the k modified positions.

(1) $(t, \mathsf{pk}, \mathsf{upk}, \mathsf{vk}) \leftarrow \mathcal{S}(1^\lambda, \mathbb{i})$.
(2) $(\tilde{\pi}_0, \ldots, \tilde{\pi}_\ell) \leftarrow \mathcal{S}(t, \mathsf{pk}, \mathsf{upk}, \mathsf{vk}, \mathbb{x}_0, \ldots, \mathbb{x}_\ell)$.
This definition could also extend to statistical / perfect zero-knowledge.

KZG commitments. For a univariate polynomial $f(X)$, we write its KZG commitment [19] as $[f(\tau)]_b = f(\tau) \cdot G_b$ where $b \in \{1, 2, T\}$ and τ is the secret random field element picked for variable X—also called trapdoor.

Cryptographic assumptions. We prove security in the Algebraic Group Model (AGM) [13], using the (q_1, q_2)-**dlog** assumption. In AGM, adversaries are *algebraic*: whenever $\mathcal{A}$ outputs a group element, it also outputs its representation as a linear combination of all previously received group elements. In the asymmetric bilinear group setting, if $\boldsymbol{U}_1 = (U_{1,1}, \ldots, U_{1,m_1}) \in \mathbb{G}_1^{m_1}$ and $\boldsymbol{U}_2 = (U_{2,1}, \ldots, U_{2,m_2}) \in \mathbb{G}_2^{m_2}$ are all the group elements $\mathcal{A}$ received so far, then for any group element $V \in \mathbb{G}_b$ ($b \in \{1, 2\}$) that $\mathcal{A}$ outputs, it provides a vector $\boldsymbol{\mu}^V \in \mathbb{F}^{m_b}$ such that $V = \sum_{i \in [m_b]} \mu_i^V \cdot U_{b,i}$. Similarly, for any $V \in \mathbb{G}_T$ that $\mathcal{A}$ outputs, it provides the coefficients of V as a linear combination of the pairings $e(U_{1,i}, U_{2,j})$ and the $\mathbb{G}_T$ elements it received. The following (q_1, q_2)-**dlog** assumption is the *bilinear* version of the well-known q-**dlog** assumption.

Assumption 2.1 ((q_1, q_2)-**dlog**). *For any PPT $\mathcal{A}$, given* $\mathsf{pp}_{\mathsf{bl}} \leftarrow \mathsf{GroupGen}_{\mathsf{bl}}(1^\lambda)$ *and*

$$(G_1, \tau \cdot G_1, \ldots, \tau^{q_1} \cdot G_1,\ G_2, \tau \cdot G_2, \ldots, \tau^{q_2} \cdot G_2)$$

where $\tau \xleftarrow{\$} \mathbb{F}$*, the probability of $\mathcal{A}$ outputting τ is* $\mathsf{negl}(\lambda)$.

R1CS. *Rank-1 Constraint System (R1CS)* is an NP-complete language that generalizes arithmetic circuit satisfiability. The indexed relation $\mathcal{R}_{\mathrm{R1CS}}$ contains the tuples

$$(\mathbb{i}, \mathbb{x}, \mathbb{w}) = ((\mathbb{F}, n, m, k, \mathbf{A}, \mathbf{B}, \mathbf{C}), \mathbb{x}, \mathbb{w})$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^{n \times m}$, $\mathbb{x} \in \mathbb{F}^{k-1}$, $\mathbb{w} \in \mathbb{F}^{m-k}$, such that

$$(\mathbf{A} \cdot \boldsymbol{z}) \circ (\mathbf{B} \cdot \boldsymbol{z}) = \mathbf{C} \cdot \boldsymbol{z}$$

where $\boldsymbol{z} = (1, \mathbb{x}, \mathbb{w})$. Groth16 [17] is a celebrated zk-SNARK for $\mathcal{R}_{\mathrm{R1CS}}$. See the formal definition of zk-SNARKs and the construction of Groth16 in Section A.

3 A Dynamic zk-SNARK for the multiplication relation

In this section, we introduce a dynamic zk-SNARK for the multiplication relation $\mathcal{R}_\times$ that contains the tuples

$$(\mathbb{i}, \mathbb{x}, \mathbb{w}) = (n, ([a(\tau)]_1, [b(\tau)]_2, [c(\tau)]_1), (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}))$$

such that (i) $a(X), b(X), c(X)$ are Lagrange interpolations of $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{F}^n$ at roots of unity for $n = 2^\ell$; and (ii) $a(X)$, $b(X)$ and $c(X)$ satisfy the entry-wise product

$$a(\omega^i) \cdot b(\omega^i) = c(\omega^i) \text{ for all } i = 0, \ldots, n-1\,.$$

To build a zk-SNARK for this relation, one can compute a quotient polynomial $q(X)$ such that

$$a(X) \cdot b(X) - c(X) = q(X) \cdot (X^n - 1)$$

and output commitment $[q(\tau)]_1$ as the zk-SNARK proof. Then the verifier can just check

$$e([a(\tau)]_1, [b(\tau)]_2) - e([c(\tau)]_1, G_2) = e([q(\tau)]_1, [\tau^n - 1]_2)\,.$$

3.1 Update-friendly proof for the designated verifier setting

We propose an update-friendly SNARK proof for the designated-verifier setting. Since $a(\omega^i) \cdot b(\omega^i) = c(\omega^i)$ for all $i = 0, \ldots, n-1$, we can write the polynomial $a(X) \cdot b(X) - c(X)$ as

$$\begin{aligned} & \left(\sum_{j=0}^{n-1} \boldsymbol{a}[j] \cdot \mathcal{L}_j(X)\right) \cdot \left(\sum_{j=0}^{n-1} \boldsymbol{b}[j] \cdot \mathcal{L}_j(X)\right) - \sum_{j=0}^{n-1} \boldsymbol{c}[j] \cdot \mathcal{L}_j(X) \\ = & \sum_{j=0}^{n-1} \boldsymbol{c}[j] \cdot (\mathcal{L}_j^2(X) - \mathcal{L}_j(X)) + \sum_{i \neq j} \boldsymbol{a}[i] \mathcal{L}_i(X) \cdot \boldsymbol{b}[j] \mathcal{L}_j(X) \\ = & \ q_c(X) \cdot (X^n - 1) + Q(X) \cdot (X^n - 1)\,, \end{aligned} \tag{1}$$

where $q_c(X)$ is the quotient of the division

$$\sum_{j=0}^{n-1} \boldsymbol{c}[j] \cdot (\mathcal{L}_j^2(X) - \mathcal{L}_j(X))/(X^n - 1)$$

and $Q(X)$ is the quotient of the division

$$\sum_{i \neq j} \boldsymbol{a}[i] \mathcal{L}_i(X) \cdot \boldsymbol{b}[j] \mathcal{L}_j(X)/(X^n - 1)\,.$$

Note that both divisions above have no remainder. We now define our update-friendly proof for the designated verifier setting to consist of the following two elements.

(1) A commitment $[q_c(\tau)]_1 \in \mathbb{G}_1$.
(2) The target-group element $[Q(\tau)]_T \in \mathbb{G}_T$.

To verify this proof, the designated verifier can use trapdoor τ (used for KZG) and check that $e([a(\tau)]_1, [b(\tau)]_2) - e([c(\tau)]_1, G_2)$ equals

$$e([q_c(\tau)]_1, [\tau^n - 1]_2) + (\tau^n - 1) \cdot [Q(\tau)]_T\,.$$

We note that the reason we must use $[Q(\tau)]_T$ as a proof element, instead of $[Q(\tau)]_1$ is because $[Q(\tau)]_T$ is easily updatable (due to pairing properties), as opposed to $[Q(\tau)]_1$. As a matter of fact, it does not appear that it is easy to update $[Q(\tau)]_1$ without recomputation. We now describe how to update $[q_c(\tau)]_1$ and $[Q(\tau)]_T$ efficiently.

3.2 Computing and updating $q_c(X)$

Let us set $p_c(X) = \sum_{j=0}^{n-1} \boldsymbol{c}[j] \cdot (\mathcal{L}_j^2(X) - \mathcal{L}_j(X))$. Note that $p_c(X)$ has roots $1, \omega, \ldots, \omega^{n-1}$ and therefore $X^n - 1$ divides $p_c(X)$, in particular we have

$$p_c(X) = \left(\sum_{j=0}^{n-1} \boldsymbol{c}[j] \cdot \mathcal{T}_j(X)\right) \cdot (X^n - 1)\,,$$

where, recall, that $\mathcal{T}_j(X) = \omega^j(\mathcal{L}_j(X) - 1)/(n(X - \omega^j))$. Therefore $q_c(X) = \sum_{j=0}^{n-1} \boldsymbol{c}[j] \cdot \mathcal{T}_j(X)$, which is updatable in constant time whenever $\boldsymbol{c}$ changes and since $\mathcal{T}_j(X)$ is a fixed polynomial.

3.3 Computing and updating $[Q(\tau)]_T$

To enable efficient updates of the group element $[Q(\tau)]_T$, we will express the group element $[Q(\tau)]_T$ as a sum of group elements corresponding to the nodes of a binary tree whose leaves i store the vector elements $\boldsymbol{a}[i]$ and $\boldsymbol{b}[i]$ for $i = 0, 1, \ldots, n-1$. In this binary tree, nodes v at level t are naturally labeled with a t-bit label as follows. The root is labeled with ϵ, its left child with 0, its right child with 1, its leftmost grandchild with 00 and so on. Let $\mathsf{lv}(v)$ be

the set of indices corresponding to leaves in v's subtree and $\overline{\mathsf{lv}}(v)$ be $\{0,\ldots,n-1\} \setminus \mathsf{lv}(v)$. Let also $\mathsf{in}(n)$ be the set of the $n-1$ labels corresponding to internal nodes, i.e., all nodes except the leaves. See Figure 1.

For every internal node $v \in \mathsf{in}(n)$, we store four KZG commitments $[Q^{\boldsymbol{a}}_{v0}]_1$, $[Q^{\boldsymbol{a}}_{v1}]_1$, $[Q^{\boldsymbol{b}}_{v0}]_2$, $[Q^{\boldsymbol{b}}_{v1}]_2$. Our final proof element $[Q(\tau)]_T$ in the target group will be defined as

$$[Q(\tau)]_T = \sum_{v\in\mathsf{in}(n)} \left(e([Q^{\boldsymbol{a}}_{v0}]_1, [Q^{\boldsymbol{b}}_{v1}]_2) + e([Q^{\boldsymbol{a}}_{v1}]_1, [Q^{\boldsymbol{b}}_{v0}]_2)\right).$$

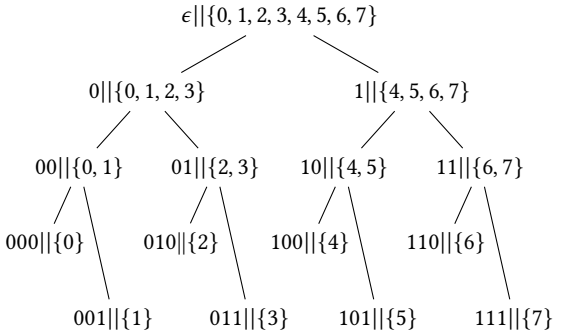


Figure 1: Our tree for $n = 8$. We show the pairs $v||\mathsf{lv}(v)$. Note that $\mathsf{in}(8) = \{\epsilon, 0, 1, 00, 01, 10, 11\}$.

Importantly, every KZG commitment $[Q^{\boldsymbol{a}}_{vb}]_1$ and $[Q^{\boldsymbol{b}}_{vb}]_2$ ($b \in \{0,1\}$) will be a linear combination of leaves in $\mathsf{lv}(vb)$. Therefore changing a value $\boldsymbol{a}[i]$ or $\boldsymbol{b}[i]$, requires updating those $\log n$ $[Q^{\boldsymbol{a}}_{vb}]_1$ and $[Q^{\boldsymbol{b}}_{vb}]_2$ on the path from i to the root—which can be done in $O(\log n)$ time. Once those are updated, the final group element can be updated with an additional $O(\log n)$ pairings and $O(\log n)$ operations in the target group $\mathbb{G}_T$, leading to an overall $O(\log n)$ time. What remains to show now is that indeed, $Q^{\boldsymbol{a}}_{vb}$ and $Q^{\boldsymbol{b}}_{vb}$ can be expressed as linear combinations of $\mathsf{lv}(vb)$. We summarize our central result in the following theorem.

Theorem 3.1. *Let $P(X) = \sum_{i\neq j} \boldsymbol{a}[i]\mathcal{L}_i(X)\cdot\boldsymbol{b}[j]\mathcal{L}_j(X)$ and let $Q(X)$ be the quotient $P(X)/(X^n-1)$. Then*

$$Q(X) = \sum_{v\in\mathsf{in}(n)} (Q^{\boldsymbol{a}}_{v0}(X)Q^{\boldsymbol{b}}_{v1}(X) + Q^{\boldsymbol{a}}_{v1}(X)Q^{\boldsymbol{b}}_{v0}(X)),$$

where, for $b \in \{0,1\}$, it is

$$Q^{\boldsymbol{a}}_{vb}(X) = \sum_{i\in\mathsf{lv}(vb)} \boldsymbol{a}[i]\cdot\mathcal{K}_{vb,i}(X) \tag{2}$$

and

$$Q^{\boldsymbol{b}}_{vb}(X) = \sum_{i\in\mathsf{lv}(vb)} \boldsymbol{b}[i]\cdot\mathcal{K}^*_{vb,i}(X), \tag{3}$$

where $\mathcal{K}_{vb,i}(X)$ and $\mathcal{K}^*_{vb,i}(X)$ are fixed polynomials.

Proof. Set

$$\mathcal{K}_{vb,i} = \frac{\mathcal{L}_i(X)}{\prod_{s\in\overline{\mathsf{lv}}(vb)}(X-\omega^s)}, \quad \mathcal{K}^*_{vb,i} = \frac{\mathcal{L}_i(X)}{\prod_{s\in\mathsf{lv}(v\bar{b})}(X-\omega^s)} \tag{4}$$

as our fixed polynomials. Then

$$\begin{aligned}
&(X^n-1)\cdot Q^{\boldsymbol{a}}_{v0}(X)\cdot Q^{\boldsymbol{b}}_{v1}(X)\\
=\ &(X^n-1)\cdot\frac{\sum_{i\in\mathsf{lv}(v0)}\boldsymbol{a}[i]\mathcal{L}_i(X)}{\prod_{s\in\overline{\mathsf{lv}}(v0)}(X-\omega^s)}\cdot\frac{\sum_{j\in\mathsf{lv}(v1)}\boldsymbol{b}[j]\mathcal{L}_j(X)}{\prod_{s\in\mathsf{lv}(v0)}(X-\omega^s)}\\
=\ &\left(\sum_{i\in\mathsf{lv}(v0)}\boldsymbol{a}[i]\mathcal{L}_i(X)\right)\left(\sum_{j\in\mathsf{lv}(v1)}\boldsymbol{b}[j]\mathcal{L}_j(X)\right),
\end{aligned}$$

since $\prod_{s\in\overline{\mathsf{lv}}(v0)}(X-\omega^s)\prod_{s\in\mathsf{lv}(v0)}(X-\omega^s) = X^n-1$. Similarly we have that

$$\begin{aligned}
&(X^n-1)\cdot Q^{\boldsymbol{a}}_{v1}(X)\cdot Q^{\boldsymbol{b}}_{v0}(X)\\
=\ &\left(\sum_{i\in\mathsf{lv}(v1)}\boldsymbol{a}[i]\mathcal{L}_i(X)\right)\left(\sum_{j\in\mathsf{lv}(v0)}\boldsymbol{b}[j]\mathcal{L}_j(X)\right).
\end{aligned}$$

Therefore

$$\begin{aligned}
&(X^n-1)\cdot Q(X)\\
=\ &\sum_{v\in\mathsf{in}(n)}\left(\sum_{i\in\mathsf{lv}(v0)}\boldsymbol{a}[i]\mathcal{L}_i(X)\right)\left(\sum_{j\in\mathsf{lv}(v1)}\boldsymbol{b}[j]\mathcal{L}_j(X)\right)\\
+\ &\sum_{v\in\mathsf{in}(n)}\left(\sum_{i\in\mathsf{lv}(v1)}\boldsymbol{a}[i]\mathcal{L}_i(X)\right)\left(\sum_{j\in\mathsf{lv}(v0)}\boldsymbol{b}[j]\mathcal{L}_j(X)\right)\\
=\ &\sum_{i\neq j}\boldsymbol{a}[i]\mathcal{L}_i(X)\cdot\boldsymbol{b}[j]\mathcal{L}_j(X),
\end{aligned}$$

as required. □

Theorem 3.2. *The protocol of Fig. 2 is a dynamic SNARK (per Definition 2.1) for the multiplication relation $\mathcal{R}_\times$ (on vectors of size n) with a designated verifier based on (q_1,q_2)-**dlog** (see Assumption 2.1) in the AGM. Its complexities are as follows.*

(1) $\mathcal{G}$ runs in $O(n\log n)$ time, outputs pk and upk of $O(n\log n)$ size, and vk of $O(1)$ size;
(2) $\mathcal{P}$ runs in $O(n\log n)$ time and outputs a proof π of $O(1)$ size;
(3) $\mathcal{U}$ runs in $O(k\log n)$ time, where k is the Hamming distance of $\mathbb{w},\mathbb{w}'$;
(4) $\mathcal{V}$ runs in $O(1)$ time.

Proof. Completeness and complexities follow naturally from the protocol. For knowledge soundness, we construct the following extractor $\mathcal{E}_{\mathcal{A}}(\mathsf{pk},\mathsf{upk})$ outputting a witness $\mathbb{w}$ and show that, under the (q_1,q_2)-**dlog** assumption in the AGM, if the verifier accepts, then $\mathbb{w}$ is a valid witness for $\mathbb{x}$ output by $\mathcal{A}$:

(1) Run the algebraic adversary $(\mathbb{x},\pi) \leftarrow \mathcal{A}(\mathsf{pk},\mathsf{upk})$.
(2) Note that since $\mathcal{A}$ is algebraic, it also outputs vectors of scalars showing how $\mathbb{x}$ can be computed from $(\mathsf{pk},\mathsf{upk})$. Based on them, $\mathcal{E}_{\mathcal{A}}$ reconstructs $\widetilde{a}(X)$, $\widetilde{b}(X)$, and $\widetilde{c}(X)$ such that $\mathbb{x} = ([\widetilde{a}(\tau)]_1, [\widetilde{b}(\tau)]_2, [\widetilde{c}(\tau)]_1)$.
(3) Output $\mathbb{w} = (\widetilde{\boldsymbol{a}},\widetilde{\boldsymbol{b}},\widetilde{\boldsymbol{c}})$ where $\widetilde{\boldsymbol{a}}[i] = \widetilde{a}(\omega^i)$, $\widetilde{\boldsymbol{b}}[i] = \widetilde{b}(\omega^i)$, $\widetilde{\boldsymbol{c}}[i] = \widetilde{c}(\omega^i)$, $\forall 0 \le i < n$.

Clearly, the extractor runs in polynomial time. To conclude the proof we only need to show that there exists a polynomial $q(X)$ such that

$$\widetilde{a}(X)\cdot\widetilde{b}(X) - \widetilde{c}(X) = q(X)(X^n-1),$$

- $\mathcal{G}(1^\lambda, n) \to (\mathsf{pk}, \mathsf{upk}, \mathsf{vk})$:
  - $\mathsf{pp}_{\mathsf{bl}} \leftarrow \mathsf{GroupGen}_{\mathsf{bl}}(1^\lambda)$.
  - Pick random $\tau \in \mathbb{F}$ for X.
  - Output $\mathsf{vk} = (G_2, \tau)$ and $\mathsf{pk} = \mathsf{upk}$ as
    - $[\tau^i]_1$, $[\tau^i]_2$ and $[\mathcal{T}_i(\tau)]_1$ for $i = 0, \ldots, n-1$.
    - $[\mathcal{K}_{v0,i}(\tau)]_1$ for $v \in \mathsf{in}(n)$ and $i \in \mathsf{lv}(v0)$.
    - $[\mathcal{K}_{v1,i}(\tau)]_1$ for $v \in \mathsf{in}(n)$ and $i \in \mathsf{lv}(v1)$.
    - $[\mathcal{K}^*_{v0,i}(\tau)]_2$ for $v \in \mathsf{in}(n)$ and $i \in \mathsf{lv}(v0)$.
    - $[\mathcal{K}^*_{v1,i}(\tau)]_2$ for $v \in \mathsf{in}(n)$ and $i \in \mathsf{lv}(v1)$.
- $\mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}) \to (\pi, \mathsf{aux})$:
  - Parse $\mathbb{w} = (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$.
  - Compute $[q_c(\tau)]_1$ and $[Q(\tau)]_1$ as in Eq. (1) and output $[q_c(\tau)]_1$ and $[Q(\tau)]_T = e([Q(\tau)]_1, G_2)$ as π.
  - For all $v \in \mathsf{in}(n)$, compute and output as aux the following commitments.
    - $[Q^{\boldsymbol{a}}_{v0}(\tau)]_1$ and $[Q^{\boldsymbol{a}}_{v1}(\tau)]_1$, as in Eq. (2).
    - $[Q^{\boldsymbol{b}}_{v0}(\tau)]_2$ and $[Q^{\boldsymbol{b}}_{v1}(\tau)]_2$, as in Eq. (3).
- $\mathcal{U}(\mathsf{upk}, \mathbb{x}', \mathbb{w}', \mathbb{x}, \mathbb{w}, \pi, \mathsf{aux}) \to (\pi', \mathsf{aux}')$:
  - Parse $\pi = ([q_c(\tau)]_1, [Q(\tau)]_T)$.
  - Parse $\mathbb{w} = (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ and $\mathbb{w}' = (\boldsymbol{a}', \boldsymbol{b}', \boldsymbol{c}')$.
  - $[q'_c(\tau)]_1 = [q_c(\tau)]_1 + \sum_{\boldsymbol{c}[i] \neq \boldsymbol{c}'[i]} (\boldsymbol{c}'[i] - \boldsymbol{c}[i]) \cdot [\mathcal{T}_i(\tau)]_1$.
  - Find the set V of affected tree nodes (due to the update) and for all $v \in V$ compute $[Q^{\boldsymbol{a}'}_{v0}(\tau)]_1$, $[Q^{\boldsymbol{a}'}_{v1}(\tau)]_1$, $[Q^{\boldsymbol{b}'}_{v0}(\tau)]_2$ and $[Q^{\boldsymbol{b}'}_{v1}(\tau)]_2$.
  - Compute

$$\begin{aligned}[Q'(\tau)]_T = [Q(\tau)]_T + \sum_{v \in V} \big(& e([Q^{\boldsymbol{a}'}_{v0}(\tau)]_1, [Q^{\boldsymbol{b}'}_{v1}(\tau)]_2) \\ & - e([Q^{\boldsymbol{a}}_{v0}(\tau)]_1, [Q^{\boldsymbol{b}}_{v1}(\tau)]_2) \\ & + e([Q^{\boldsymbol{a}'}_{v1}(\tau)]_1, [Q^{\boldsymbol{b}'}_{v0}(\tau)]_2) \\ & - e([Q^{\boldsymbol{a}}_{v1}(\tau)]_1, [Q^{\boldsymbol{b}}_{v0}(\tau)]_2) \big) .\end{aligned}$$

  - Output $\pi' = ([q'_c(\tau)]_1, [Q'(\tau)]_T)$ and aux' as aux with $Q^{\boldsymbol{a}'}_{v0}$, $Q^{\boldsymbol{a}'}_{v1}$, $Q^{\boldsymbol{b}'}_{v0}$ and $Q^{\boldsymbol{b}'}_{v1}$ replaced.
- $\mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to 0/1$:
  - Parse $\mathsf{vk} = (G_2, \tau)$ and $\mathbb{x} = ([a(\tau)]_1, [b(\tau)]_2, [c(\tau)]_1)$.
  - Parse $\pi = ([q_c(\tau)]_1, [Q(\tau)]_T)$.
  - Check if

$$\begin{aligned} e([a(\tau)]_1, [b(\tau)]_2) - e([c(\tau)]_1, G_2) \stackrel{?}{=} \\ e([q_c(\tau)]_1, [\tau^n - 1]_2) + (\tau^n - 1) \cdot [Q(\tau)]_T \end{aligned}$$

Figure 2: A dynamic SNARK for $\mathcal{R}_\times$ with a designated verifier.

which is true with overwhelming probability from the acceptance of the verifier.

□

The protocol in Fig. 2 can easily be made zero-knowledge by a standard technique using mask polynomials. We omit the details here and present the following corollary.

Corollary 3.3. *There is a zero-knowledge version of the protocol in Fig. 2 with the same complexities.*

4 Delphus: A dynamic zk-SNARK for circuit satisfiability

In this section, we introduce Delphus, a designated-verifier dynamic zk-SNARK for the relation $\mathcal{R}_{\mathrm{R1CS}}$ by plugging the protocol from Fig. 2 into the Groth16 [17] (see details in Fig. 10). We reuse the definitions of the polynomials $u_j(X)$, $v_j(X)$, $w_j(X)$, $t_j(X)$, $a(X)$, $b(X)$, $c(X)$ from Section A.

For an index from $\mathcal{R}_{\mathrm{R1CS}}$ $\mathbb{i} = (\mathbb{F}, n, m, k, \mathbf{A}, \mathbf{B}, \mathbf{C})$, recall that the existence of $z = (1, \mathbb{x}, \mathbb{w})$ such that $(\mathbf{A} \cdot z) \circ (\mathbf{B} \cdot z) = \mathbf{C} \cdot z$ is equivalent to the existence of a quotient polynomial $q(X)$ such that

$$a(X) \cdot b(X) - c(X) = q(X) \cdot (X^n - 1) ,$$

which is exactly the equation established in Section 3. We now examine how the Groth16 proof $([A]_1, [B]_2, [C]_1)$ is computed:

$$[A]_1 = [\alpha]_1 + r[\delta]_1 + \sum_{0 \le j < m} z[j] \cdot [u_j(\tau)]_1 ,$$

$$[B]_2 = [\beta]_2 + s[\delta]_2 + \sum_{0 \le j < m} z[j] \cdot [v_j(\tau)]_2 ,$$

$$\begin{aligned}[C]_1 = & \, s[A]_1 + r[B]_1 - rs[\delta]_1 \\ & + \left[\frac{q(\tau)(\tau^n - 1)}{\delta}\right]_1 + \sum_{k \le j < m} z[j] \left[\frac{t_j(\tau)}{\delta}\right]_1 .\end{aligned}$$

To support dynamic updates of the proof, we partition these terms into the three categories:

- Terms involving $z[j]$. These terms can be updated linearly using the prover key.
- Terms involving r, s. These terms can be updated directly using freshly sampled randomness r', s'.
- The term with $q(\tau)$. Writing $q(\tau) = q_c(\tau) + Q(\tau)$, we can update them separately as in Section 3.

We present the full protocol in Fig. 3. One notable difference from the original Groth16 is the introduction of an additional random value η, which is used to ensure the polynomial $Q(X)$ can be computed only using $\mathcal{K}_{v,b,i}(X)$ and $\mathcal{K}^*_{v,b,i}(X)$.

Theorem 4.1. *The protocol of Fig. 3 is a dynamic SNARK (per Definition 2.1) for $\mathcal{R}_{\mathrm{R1CS}}$, restricted to indices in which each entry of z other than $z[0]$ appears in $O(1)$ constraints, with a designated verifier based on (q_1, q_2)-**dlog** (see Assumption 2.1) in the AGM. Its complexities are as follows.*

(1) $\mathcal{G}$ runs in $O(n \log n)$ time, outputs pk *and* upk *of $O(n \log n)$ size, and* vk *of $O(1)$ size;*
(2) $\mathcal{P}$ runs in $O(n \log n)$ time and outputs a proof π of $O(1)$ size;
(3) $\mathcal{U}$ runs in $O(k \log n)$ time (k is the Hamming distance of $\mathbb{x}||\mathbb{w}$ and $\mathbb{x}'||\mathbb{w}'$);
(4) $\mathcal{V}$ runs in $O(1)$ time.

Proof. Completeness and complexities follow naturally from the protocol. For knowledge soundness, we construct the following extractor $\mathcal{E}_{\mathcal{A}}(\mathsf{pk}, \mathsf{upk})$ outputting a witness $\mathbb{w}$:

(1) Run the algebraic adversary $(\mathbb{x}, \pi) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{upk})$.
(2) Note that since $\mathcal{A}$ is algebraic, it also outputs vectors of scalars showing how $\pi = ([A]_1, [B]_2, [C]_1, [D]_T)$ can be computed from pk (pk = upk). Specifically, $\mathcal{A}$ outputs $\boldsymbol{\mu}^A = (\mu^A_\sigma)_{\sigma \in \mathsf{pk}}$ such that $[A]_1 = \sum_{\sigma \in \mathsf{pk}} \mu^A_\sigma \cdot \sigma$.

- $\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{upk}, \mathsf{vk})$:
  - $\mathsf{pp}_{\mathsf{bl}} \leftarrow \mathsf{GroupGen}_{\mathsf{bl}}(1^\lambda)$. Pick $\alpha, \beta, \gamma, \delta, \eta, \tau \xleftarrow{\$} \mathbb{F}$.
  - Let $t_j(X) = \beta u_j(X) + \alpha v_j(X) + w_j(X)$ for $0 \le j < m$.
  - Output

$$\mathsf{pk} = \mathsf{upk} = \left(\begin{array}{c} [\alpha]_1, [\beta]_1, [\beta]_2, [\delta]_1, [\delta]_2, \\ ([u_j(\tau)]_1, [v_j(\tau)]_1, [v_j(\tau)]_2)_{0\le j<m}, \\ \left([t_j(\tau)/\delta]_1\right)_{k\le j<m}, ([\mathcal{T}_i(\tau)(\tau^n-1)/\delta]_1)_{0\le i<n}, \\ \left([\mathcal{K}_{v,b,i}(\tau)/\eta]_1, \left[\mathcal{K}^*_{v,b,i}(\tau)/\eta\right]_2\right)_{v,b,i} \end{array} \right),$$

$$\mathsf{vk} = \left(\eta^2(\tau^n-1), [\alpha]_1, [\beta]_2, [\gamma]_2, [\delta]_2, \left([t_j(\tau)/\gamma]_1\right)_{0\le j<k}\right).$$

- $\mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}) \to (\pi, \mathsf{aux})$:
  - Pick $r, s \xleftarrow{\$} \mathbb{F}$. Set

$$\mathsf{aux} = (r, s, [B]_1, \{[Q^{\boldsymbol{a}}_{vb}(\tau)/\eta]_1, [Q^{\boldsymbol{b}}_{vb}(\tau)/\eta]_2\}).$$

  - Output $\pi = \left([A]_1, [B]_2, [C]_1, [D]_T = \left[\frac{Q(\tau)}{\eta^2}\right]_T\right)$ where

$$[A]_1 = [\alpha]_1 + r[\delta]_1 + \sum_{0\le j<m} \boldsymbol{z}[j]\cdot[u_j(\tau)]_1,$$

$$[B]_2 = [\beta]_2 + s[\delta]_2 + \sum_{0\le j<m} \boldsymbol{z}[j]\cdot[v_j(\tau)]_2,$$

$$[C]_1 = s[A]_1 + r[B]_1 - rs[\delta]_1 + [q_c(\tau)(\tau^n-1)/\delta]_1 + \sum_{k\le j<m} \boldsymbol{z}[j]\left[t_j(\tau)/\delta\right]_1.$$

- $\mathcal{U}(\mathsf{upk}, \mathbb{x}', \mathbb{w}', \mathbb{x}, \mathbb{w}, \pi, \mathsf{aux}) \to (\pi', \mathsf{aux}')$:
  - Set $\boldsymbol{z} = (1, \mathbb{x}, \mathbb{w})$, $\boldsymbol{z}' = (1, \mathbb{x}', \mathbb{w}')$ and $\boldsymbol{z}^\Delta = \boldsymbol{z}' - \boldsymbol{z}$.
  - Parse $\pi = ([A]_1, [B]_2, [C]_1, [D]_T)$ and

$$\mathsf{aux} = (r, s, [B]_1, \{[Q^{\boldsymbol{a}}_{vb}(\tau)/\eta]_1, [Q^{\boldsymbol{b}}_{vb}(\tau)/\eta]_2\}).$$

  - Pick $r', s' \xleftarrow{\$} \mathbb{F}$ and update aux to aux′.
  - Output $\pi' = \left([A']_1, [B']_2, [C']_1, \left[\frac{Q'(\tau)}{\eta^2}\right]_T\right)$ where

$$[A']_1 = [A]_1 + (r'-r)[\delta]_1 + \sum_{0\le j<m} \boldsymbol{z}^\Delta[j]\cdot[u_j(\tau)]_1,$$

$$[B']_2 = [B]_2 + (s'-s)[\delta]_2 + \sum_{0\le j<m} \boldsymbol{z}^\Delta[j]\cdot[v_j(\tau)]_2,$$

$$[C']_1 = [C]_1 + s'[A']_1 - s[A]_1 + r'[B']_1 - r[B]_1 - (r's'-rs)[\delta]_1 + \sum_{0\le j<n} \boldsymbol{c}^\Delta[j]\cdot\left[\mathcal{T}_j(\tau)(\tau^n-1)/\delta\right]_1 + \sum_{k\le j<m} \boldsymbol{z}^\Delta[j]\cdot\left[t_j(\tau)/\delta\right]_1.$$

- $\mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to 0/1$:
  - Parse $\pi = ([A]_1, [B]_2, [C]_1, [D]_T)$ and check if

$$e([A]_1, [B]_2) \stackrel{?}{=} e([\alpha]_1, [\beta]_2) + e([C]_1, [\delta]_2) + e\left(\sum_{0\le j<k} \boldsymbol{z}[j]\cdot\left[t_j(\tau)/\gamma\right]_1, [\gamma]_2\right) + \eta^2(\tau^n-1)\cdot[D]_T.$$

Figure 3: Delphus, A dynamic SNARK for $\mathcal{R}_{\mathbf{R1CS}}$ with a designated verifier based on Groth16.

(3) Output $\mathbb{w} = \left(\mu^C_{[t_j(\tau)/\delta]_1}\right)_{k\le j<m}$.

Based on (q_1, q_2)-**dlog**, if the verifier accepts, then with overwhelming probability we have

$$A\cdot B = \alpha\beta + \delta\cdot C + \eta^2(\tau^n-1)D + \sum_{0\le j<k} \boldsymbol{z}[j]\cdot t_j(\tau)$$

where

$$\begin{aligned} A =& \mu^A_\alpha\cdot\alpha + \mu^A_\beta\cdot\beta + \mu^A_\delta\cdot\delta + \sum_{0\le j<m}\left(\mu^A_{u_j}\cdot u_j(\tau) + \mu^A_{v_j}\cdot v_j(\tau)\right) \\ &+ \sum_{k\le j<m} \mu^A_{t_j}\cdot\frac{t_j(\tau)}{\delta} + \sum_{0\le i<n} \mu^A_{q_c,i}\cdot\frac{\mathcal{T}_i(\tau)(\tau^n-1)}{\delta} \\ &+ \sum_{v,b,i} \mu^A_{\mathcal{K},v,b,i}\cdot\frac{\mathcal{K}_{v,b,i}(\tau)}{\eta} \end{aligned}$$

for some $\mu^A_{(\cdot)} \in \mathbb{F}$, and B, C can also be written in a similar way, and D as a linear combination of products of a $\mathbb{G}_1$ term and a $\mathbb{G}_2$ term of pk, with corresponding $\mu^B_{(\cdot)}, \mu^C_{(\cdot)}, \mu^D_{(\cdot)} \in \mathbb{F}$; hence following the same proof strategy in [17], the extracted $\mathbb{w}$ satisfies $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\mathrm{R1CS}}$. □

Similarly, Delphus can be made zero-knowledge by introducing masks between $q_c(X)$ and $Q(X)$ as well. We omit the details here and present the following corollary.

Corollary 4.2. *There is a zero-knowledge version of Delphus (Fig. 3) with the same complexities.*

Remark. We can also plug our dynamic zk-SNARK for $\mathcal{R}_\times$ into other existing dynamic zk-SNARKs for circuit satisfiability to improve their update complexity for proving multiplication relations. For instance, Dynalog [27], as a universal dynamic zk-SNARK with $O(\mathrm{poly}\log n)$ update time, relies on the Inner Product Arguments (IPAs [8]) to dynamically prove multiplication relations, which results in significant overhead and makes it less efficient in practice. This integration is in fact conceptually straightforward so we omit the details.

5 Circuits for index validity

In this section, we construct circuits that verify the validity of indexes for arbitrary tables. We will present an application of such circuits in Section 6. A key feature of these circuits is that they support dynamic updates, a property we formalize below.

Definition 5.1 (Circuit update locality). *We say an arithmetic circuit C of size n has update locality $t(n)$, if the number of gates whose output changes in response to a single input update is bounded by $t(n)$. If C has update locality $t(n) = O(\mathrm{poly}\log(n))$, then we say C supports dynamic updates, or C is dynamically updatable.*

Tables and indexes. A table *tbl* of size n is an *unordered* list of values $\{(i, val_i)\}_{0\le i<n}$, where val_i is any predefined type of data. Given any fixed way to compare values, we can build an index of this table to enable efficient search of values. In practice, such indexes could be a (balanced) binary search tree or any of its generalizations such as a B-tree. In this paper, we only consider using the basic binary search trees as indexes.

Definition 5.2 (Binary search tree as a list). *A binary search tree $bst_{\mathbf{cmp}}$ of size n under comparator* **cmp** *is an* ordered *list*

$$[node_i = (i, val_i, left_i, right_i)]_{0\le i<n}$$

which forms a rooted binary tree where all nodes are arranged in strict total order using **cmp** *to compare val_i. Specifically, for any $node_i$, all the values in its left sub-tree (rooted at $left_i$) are no more than val_i and all the values in its right sub-tree (rooted at $right_i$) are no less than val_i. We will omit* **cmp** *in the notation if it is implicitly used.*

Check the validity of indexes. Given a table *tbl* and a binary search tree *bst* as its index, let H_{tbl} and H_{bst} denote their respective digests. Here, we use a multiplicative hash function `MuHash` [4, 9] as the digest of an unordered list (set) and the standard Merkle hash [22] for an ordered list. The relation $\mathcal{R}_{\text{valid}}$, which captures the validity of indexes, consists of pairs $(\mathbb{x}, \mathbb{w})$ where $\mathbb{x} = (H_{tbl}, H_{bst})$ and $\mathbb{w} = bst = [(i, val_i, left_i, right_i)]_{0 \le i < n}$ such that

- *bst* is a valid binary search tree;
- $H_{tbl} = \texttt{MuHash}(\{(i, val_i)\}_{0 \le i < n})$;
- $H_{bst} = \texttt{Merkle}(bst, root)$, the Merkle hash of *bst* followed by the index *root* of its root.

Circuit `IndexValidity`
Public input: H_{tbl}, H_{bst}
Private input: bst, bst_{aux}

1: Check if $H_{tbl} = \texttt{MuHash}(\{(i, val_i)\}_{0 \le i < n})$.
2: Check if $H_{bst} = \texttt{Merkle}(bst, root)$.
3: Check for all $0 \le i < n$:
- if $left_i$ is not null, then $min_i = minLeft_i$ and $maxLeft_i \le val_i$; otherwise $min_i = val_i$.
- if $right_i$ is not null, then $max_i = maxRight_i$ and $val_i \le minRight_i$; otherwise $max_i = val_i$.
- $size_i = 1 + sizeLeft_i + sizeRight_i$, using 0 in place of $sizeLeft_i$ (resp. $sizeRight_i$) if $left_i$ (resp. $right_i$) is null.

4: Check if $\texttt{InputConsist}(bst, bst_{\text{aux}}) = 1$.
5: **return** 1 iff all checks above pass.

Figure 4: The `IndexValidity` circuit.

Circuit to prove $\mathcal{R}_{\text{valid}}$. We now construct circuits for $\mathcal{R}_{\text{valid}}$. The components for computing hashes H_{tbl} and H_{bst} are standard and dynamically updatable. The key component of the circuit is the validity check for *bst*, which relies on additional auxiliary input bst_{aux}. bst_{aux} includes for each $node_i$ $(0 \le i < n)$,

- min_i, max_i: the minimum and maximum values in the (sub)-tree rooted at node i;
- $minLeft_i, maxLeft_i, minRight_i, maxRight_i$: the minimum and maximum values in the sub-trees rooted at node $left_i$ and $right_i$ respectively;
- $size_i, sizeLeft_i, sizeRight_i$: the numbers of nodes in the sub-trees rooted at node i, $left_i$ and $right_i$ respectively;

as well as $(root, min_{root}, max_{root}, size_{root})$, corresponding to the root node of *bst*. The following lemma shows how to verify the validity of *bst* using these auxiliary inputs. Its proof is deferred to Section B.1.1.

Lemma 5.1. *bst is a valid binary search tree iff there exists bst_{aux} such that the following holds:*

- ***Node consistency.*** *For each $node_i$,*
  (1) if $left_i$ is not null, then $min_i = minLeft_i$ and $maxLeft_i \le val_i$; otherwise $min_i = val_i$.
  (2) if $right_i$ is not null, then $max_i = maxRight_i$ and $val_i \le minRight_i$; otherwise $max_i = val_i$.
  (3) $size_i = 1 + sizeLeft_i + sizeRight_i$, using 0 in place of $sizeLeft_i$ (resp. $sizeRight_i$) if $left_i$ (resp. $right_i$) is null.
- ***Input consistency.*** *The following two multisets are equal:*

$$\{(i, min_i, max_i, size_i)\}_{0 \le i < n}$$
$$= \{(left_i, minLeft_i, maxLeft_i, sizeLeft_i)\}_{0 \le i < n;\ left_i \text{ is not null}}$$
$$+ \{(right_i, minRight_i, maxRight_i, sizeRight_i)\}_{0 \le i < n;\ right_i \text{ is not null}}$$
$$+ \{(root, min_{root}, max_{root}, size_{root})\}$$

Based on Lemma 5.1, we build the circuit in Fig. 4, where two versions of the sub-circuit `InputConsist` will be introduced in Section 5.1. The following theorem summarizes the features of the circuit (see its proof in Section B.1.2).

Theorem 5.1. *The circuit C from Fig. 4 satisfies that*

(1) For any $\mathbb{x} = (H_{tbl}, H_{bst})$ and $\mathbb{w} = bst$, $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\text{valid}}$ if and only if there is bst_{aux} such that $C(H_{tbl}, H_{bst}, bst, bst_{\text{aux}}) = 1$.
(2) Suppose the sub-circuit `InputConsist` has size $s(n)$ and update locality $t(n)$, then C has size $s(n) + O(n)$ and update locality $t(n) + O(\log n)$.

5.1 Sub-circuits for input consistency

In this section, we construct dynamically updatable circuits for the input consistency check (see Lemma 5.1). We will provide two constructions. The first construction has linear size but requires heuristic instantiation of random oracles, whereas the second construction avoids this requirement at the cost of an additional $\log^2 n$ factor in the circuit size.

5.1.1 First construction. The input consistency check essentially verifies whether two multisets are equal. This can be achieved using a multiset hash function [4, 9]. Specifically, to check if two multisets S_1, S_2 are equal, one can test the following grand product equality:

$$\prod_{i \in S_1} \mathsf{H}(i) = \prod_{j \in S_2} \mathsf{H}(j),$$

where H is a random oracle. While one could consider using universal hash functions such as $\mathsf{H}(x) = x + r$ where r is randomly sampled every time as in [3, 14], the entire circuit will not be dynamically updatable, because all terms in the grand product will change whenever r changes. To support dynamic updates, we should pick a deterministic hash function.

Based on this idea, the circuit construction becomes straightforward. It selects the corresponding values from the input, forms tuples representing the multisets, and applies the multiset hash function. Note that the grand product can be computed in a tree structure. Our circuit is shown in Fig. 5.

5.1.2 Second construction. The main limitation of the first construction is that it requires heuristic instantiation of random oracles. To avoid that, we can employ a sorting-based approach. Note that the first multiset $\{(i, min_i, max_i, size_i)\}_{0 \le i < n}$ as an ordered list is already sorted by i. If we sort the second multiset as well by the first entry in each tuple, we obtain two identical sorted lists. This motivates the use of a data independent circuit for sorting. See the details of this sorting circuit in Section C, where we present the circuit in Fig. 11 and discuss its update locality in Theorem C.1.

Circuit `InputConsist1`
Public input: bst, bst_{aux}

1: For each $0 \le i < n$,
- Form tuples from bst, bst_{aux},

$$(i, min_i, max_i, size_i) \quad (left_i, minLeft_i, maxLeft_i, sizeLeft_i)$$
$$(right_i, minRight_i, maxRight_i, sizeRight_i)\,.$$

- Compute $H^{(i)} = \mathsf{H}(i, min_i, max_i, size_i)$.
- If $left_i$ is null, compute $H^{(i)}_{left} = 1$, otherwise

$$H^{(i)}_{left} = \mathsf{H}(left_i, minLeft_i, maxLeft_i, sizeLeft_i)\,.$$

- If $right_i$ is null, compute $H^{(i)}_{right} = 1$, otherwise

$$H^{(i)}_{right} = \mathsf{H}(right_i, minRight_i, maxRight_i, sizeRight_i)\,.$$

2: Use a product tree to compute the product of all $H^{(i)}$, $H^{(i)}_{left}$ and $H^{(i)}_{right}$ respectively as H, H_{left} and H_{right}.
3: **return** 1 iff $H = H_{left} \cdot H_{right} \cdot \mathsf{H}(root, min_{root}, max_{root}, size_{root})$.

Figure 5: The circuit `InputConsist1`.

Circuit `InputConsist2`
Public input: bst, bst_{aux}

1: Let $list = []$ and $list' = [(root, (min_{root}, max_{root}, size_{root}))]$.
2: For each $0 \le i < n$,
- Form tuples from bst, bst_{aux},

$$(i, min_i, max_i, size_i) \quad (left_i, minLeft_i, maxLeft_i, sizeLeft_i)$$
$$(right_i, minRight_i, maxRight_i, sizeRight_i)\,.$$

- Append $(min_i, max_i, size_i)$ to $list$, and the following to $list'$:

$$(left_i, (minLeft_i, maxLeft_i, sizeLeft_i))\,,$$
$$(right_i, (minRight_i, maxRight_i, sizeRight_i))\,.$$

3: Apply $list'$ to `SortWithRandomRouting` with proper additional inputs. Let $list_{\text{out}}$ denote its output.
4: **return** 1 iff $list = list_{\text{out}}[0 : n]$ and $0 \le left_i, right_i < n$ for every non-null $left_i, right_i$.

Figure 6: The circuit `InputConsist2`.

Our second construction, presented in Fig. 6, proceeds as follows. It forms tuples of multisets, sorts the second multiset, and checks if the first n tuples in the sorted result match the first multiset. To ensure that the sorting circuit has a fixed structure independent of the input, we set

$$(left_i, minLeft_i, maxLeft_i, sizeLeft_i) = (n + i, 0, 0, 0)\,,$$
$$(right_i, minRight_i, maxRight_i, sizeRight_i) = (2n + i, 0, 0, 0)\,,$$

as placeholders for the case that $left_i$ is null and $right_i$ is null, respectively.

We state the following theorem for our two constructions. Its proof follows directly from the construction and Theorem C.1, and is therefore omitted.

Theorem 5.2. *For our two sub-circuits for input consistency,*
(1) the first circuit from Fig. 5 has $O(n)$ size and $O(\log n)$ update locality;
(2) the second circuit from Fig. 6 has $O(n \log^2 n)$ size and $O(\log^2 n)$ update locality.

We have the following corollary from Theorem 5.2.

Corollary 5.3. *(1) The circuit from Fig. 4 has $O(n)$ size and $O(\log n)$ update locality if instantiating* `InputConsist` *with the circuit in Fig. 5;*
(2) The circuit from Fig. 4 has $O(n \log^2 n)$ size and $O(\log^2 n)$ update locality if instantiating `InputConsist` *with the circuit in Fig. 6.*

6 Index-agnostic verifiable database with dynamic proof of index validity

In this section, we present an application for the circuits of index validity from Section 5 in the verifiable database. In a verifiable database (VDB) system [23, 24, 29, 30], a client outsources a large database to an untrusted server while retaining only a small *digest* locally. For each subsequent query issued by the client, the server returns the query result, the new digest (if updated), and a proof to check the integrity of the result. To simplify notation, we only consider the following two types of queries, though our construction extends easily to others (e.g., `INSERT`, `DELETE`):

- *Selection query.* This is used to select rows from a table where values fall within a specific range. Example:
  `SELECT` *name* `FROM` T_{users} `WHERE` *age* < 35
- *Update query.* This is used to update rows from a specific table where values fall within a specific range. Example:
  `UPDATE` $T_{students}$ `SET` *award* $=$ \$10 `WHERE` *score* $= 100$

6.1 Limitations of existing approaches

One natural approach to build verifiable databases is that the server uses a linear size circuit to scan the entire database (e.g., Pantry [7], vSQL [29]). This circuit takes the digest and the query as public inputs and the whole database as the private input. For the integrity of the query, the server only needs to compute a SNARK proof for the circuit. The main issue of such an approach is that the server time is at least linear to the database size. Furthermore, the SNARK proof is not dynamic because every time with a new query, the check for each record will be modified, resulting in $\Omega(n)$ wire value changes.

In modern databases, fast queries are performed through indexes (e.g., binary search trees and their generalizations), where rows are sorted by specialized comparators. To leverage these in a VDB, the client could treat them as Authenticated Data Structures (ADS, e.g., [16]) — the client stores them in the server and keeps their digests locally as well so that it can interact with the server to query/modify them. However, this creates significant overhead when the number of modifications is large. Consider the following query

`UPDATE` T_{orders} `SET` *status* $=$ *done* `WHERE` *payment* $=$ *done*

which selects the rows with *payment* = *done* and sets their *status* = *done*. If this query affects k rows, then the client has to manually update each index at the server for all k rows. Although one might consider delegating the computation of the index changes to the server (e.g., [23]) and let the client verify the updates of index digests instead, the client state management is still complicated. Even the modification of one row in the database would require the client to verify the proofs of updates for each index digest maintained locally. Moreover, in scenarios involving streaming updates, the client must still interact with the server to update these digests, even if no queries are issued during that period.

6.2 Index-agnostic verifiable database construction

In this subsection, we present our index-agnostic verifiable database in which the client stores only the digest (multiplicative hash) to the database locally so that it can perform arbitrary streaming updates locally without queries to the server. The server time is almost linear to the query size, i.e., the number of selected/modified rows.

6.2.1 Model. For space limitations, we present a simplified model to show the core contribution. Following the notations from Section 5, we simplify our system such that each database has only one table, modeled as $tbl = \{(i, val_i)\}_{0 \le i < n}$, using a multiplicative hash function MuHash [4, 9] H_{tbl} as its digest. We emphasize again this can easily extend to support multiple tables. The server maintains a database state st, which includes the table *tbl* and indexes as desired. We formalize queries in the following forms:

- *Selection* (SELECT, l, r): return all the pairs (i, val_i) such that $l \le val_i \le r$. The comparator is implicitly used.
- *Update* (UPDATE, l, r, gt): replace all the pairs (i, val_i) such that $l \le val_i \le r$ with $(i, \mathsf{gt}(val_i))$. gt is predefined.

We present our model in Definition 6.1.

Definition 6.1 (Index-agnostic verifiable database (IAVDB)). *An index-agnostic verifiable database (IAVDB) is a tuple of algorithms* $\mathsf{VDB} = (\mathcal{G}, \mathcal{P}, \mathcal{V})$ *with the following interface:*

- $\mathcal{G}(1^\lambda, tbl) \to (\mathsf{pk}, \mathsf{vk}, \mathsf{st}, H_{tbl})$: *Outputs server key* pk*, client key* vk*, initial server state* st*, and initial digest* H_{tbl}.
- $\mathcal{P}(\mathsf{pk}, \mathsf{st}, qry) \to (\mathsf{st}', res, H'_{tbl}, \pi)$: *Given server key* pk*, state* st*, and query* $qry \in \{\mathsf{SELECT}, \mathsf{UPDATE}\}$*, outputs a new server state* st′*, query result res, digest* H'_{tbl}*, and proof* π.
- $\mathcal{V}(\mathsf{vk}, H_{tbl}, H'_{tbl}, qry, res, \pi) \to 0/1$: *Given client key* vk*, old digest* H_{tbl}*, new digest* H'_{tbl}*, query qry, query result res, and a proof* π*, outputs accept or reject.*

A VDB *should have the following properties.*

- Completeness*: A* VDB *is complete, if for any* λ*, any* tbl_0*, any* $t \ge 0$*, and any queries* $qry_1, \ldots, qry_t$ *and* qry^**, the following experiment outputs* 1*:*
  - $\mathcal{G}(1^\lambda, tbl_0) \to (\mathsf{pk}, \mathsf{vk}, \mathsf{st}_0, H_0)$.
  - *For* $1 \le i \le t$*, run* $\mathcal{P}(\mathsf{pk}, \mathsf{st}_{i-1}, qry_i) \to (\mathsf{st}_i, res_i, H_i, \pi_i)$*. If* $\mathcal{V}(\mathsf{vk}, H_{i-1}, H_i, qry_i, res_i, \pi_i)$ *rejects then outputs* 0*. If* $qry_i =$ UPDATE*, then set* tbl_i *as the new database after running* qry_i *on* tbl_{i-1}*; otherwise, set* $tbl_i = tbl_{i-1}$.
  - *Run* $\mathcal{P}(\mathsf{pk}, \mathsf{st}_t, qry^*) \to (\mathsf{st}^*, res^*, H^*, \pi^*)$*. Outputs* 0 *if rejected in* $\mathcal{V}(\mathsf{vk}, H_t, H^*, qry^*, res^*, \pi^*)$*. Outputs* 1 *iff* res^* *is the result of query* qry^* *on* tbl_t.
- Soundness*: A* VDB *is sound, if for any* t*, PPT* $\mathcal{A}^*$*, the probability that the following experiment outputs* 1 *is negligible:*
  - $\mathcal{A}^*(1^\lambda) \to tbl_0$.
  - $\mathcal{G}(1^\lambda, tbl_0) \to (\mathsf{pk}, \mathsf{vk}, \mathsf{st}_0, H_0)$.
  - *For* $1 \le i \le t$*,* $\mathcal{A}^* \to qry_i$*. Run* $\mathcal{A}^*(\mathsf{pk}, \mathsf{st}_{i-1}, qry_i) \to (\mathsf{st}_i, res_i, H_i, \pi_i)$*. Outputs* 0 *if* $\mathcal{V}(\mathsf{vk}, H_{i-1}, H_i, qry_i, res_i, \pi_i)$ *rejects. If* $qry_i =$ UPDATE*, set* tbl_i *as the new database after running* qry_i *on* tbl_{i-1}*; otherwise, set* $tbl_i = tbl_{i-1}$.
  - $\mathcal{A}^* \to qry^*$*. Run* $\mathcal{A}^*(\mathsf{pk}, \mathsf{st}_t, qry^*) \to (\mathsf{st}^*, res^*, H^*, \pi^*)$*. Outputs* 0 *if* $\mathcal{V}(\mathsf{vk}, H_t, H^*, qry^*, res^*, \pi^*)$ *rejects. Outputs* 1 *iff* res^* *is* not *the result of query* qry^* *on* tbl_t.

The working flow based on Definition 6.1 is as follows. VDB.$\mathcal{G}$ is called in the setup phase. The server receives the server key pk and the initial state st, and the client receives the client key vk and the initial digest H_{tbl}. To perform a query, the client sends *qry* and the server runs VDB.$\mathcal{P}$ to get the new state st′ for itself, the query result *res*, the new digest H'_{tbl} and the proof of query π. It returns res, H'_{tbl}, π to the client. Finally, the client runs VDB.$\mathcal{V}$ to check the result and, if it accepts, updates its local digest to H'_{tbl} after an UPDATE query and retains its existing digest after a SELECT query.

6.2.2 Construction. **Indexes and circuits.** We use balanced binary search trees (formalized below) as our indexes. Standard examples include Red-Black trees and AVL trees.

Definition 6.2 (Balanced binary search tree). *A balanced binary search tree* $bst = [node_i = (i, val_i, left_i, right_i)]_{0 \le i < n}$ *is a binary search tree (per Definition 5.2) of height* $O(\log n)$ *such that if any* val_i *is modified, then we only need to modify* $O(\log n)$ *nodes, through changes to* $left_i$ *and* $right_i$*, to form a new valid binary search tree of height* $O(\log n)$.

We use Merkle hash H_{bst} as the digest of the index *bst*. The following circuits are used in our construction:

- IndexValidity ($\mathbb{i}_{\mathsf{Ind}}$, see Fig. 4 and details in Section 5). This circuit checks if H_{bst} is the digest of a valid index, i.e., if $((H_{tbl}, H_{bst}), bst) \in \mathcal{R}_{\mathsf{valid}}$.
- $\mathsf{RangeQuery}_k$ ($\mathbb{i}^{(k)}_{\mathsf{Rng}}$, see Fig. 7). This circuit checks if $res = [(id_i, val_{id_i})]_{0 \le i < k}$ with Merkle hash H_{res} is the correct result of query (SELECT, l, r) based on the index *bst*. This circuit has size $O(k \log^2 n)$ because it requires $O(k \log n)$ necessary nodes in *bst* to perform the query, where each of them requires an $O(\log n)$-size Merkle proof to show its validity according to H_{bst}. We assume k is a power of two and regard it as a parameter of the circuit. Note that the size of the query result is not known in advance, so $\log n + 1$ such circuits with parameters $2^0, 2^1, \ldots, 2^{\log n}$ are preprocessed.
- $\mathsf{ModifyArray}_{k,\mathsf{gt}}$ ($\mathbb{i}^{(k,\mathsf{gt})}_{\mathsf{Mod}}$, see Fig. 8). This circuit is designed for UPDATE. It first checks if $res = [(id_i, val_{id_i})]_{0 \le i < k}$ with Merkle hash H_{res} is in the database with digest H_{tbl}. If so, it checks if applying the gate gt to every pair in *res* will result in a new database with digest H'_{tbl}. It has size $O(k \log n)$, mostly from the Merkle hash check and recomputation. k and gt are parameters of the circuit.

High-level idea. The server maintains a database state st, which includes the database *tbl* and its digest H_{tbl}, the required index *bst* and its digest H_{bst}, and the proof of index validity π_{Ind} for *bst*. Whenever there is a new query *qry*, the server first queries the index to get a target list *res*, and computes a proof π_{Rng} for circuit `RangeQuery` to show that *res* is the correct result according to the index. If *qry* = `SELECT`, then its job is done. If *qry* = `UPDATE`, the server then modifies the target list as requested, and updates its state (the database and the index accordingly). Furthermore, it computes another proof π_{Mod} for circuit `ModifyArray` to show that *tbl* is correctly updated. Our construction is presented in Fig. 9.

Circuit $\mathtt{RangeQuery}_k$
Public input: H_{bst}, l, r, H_{res}
Private input: *res*, *nds*, *root*, π_{nds}

1: Check if $H_{res} = \mathtt{Merkle}(res)$.
2: Use π_{nds} to check if H_{bst} opens to *nds* and *root*.
3: Use *nds* to check *res* is the result for range query $[l, r]$ in the binary search tree rooted at *root*.
4: **return** 1 iff all checks above pass.

Figure 7: The circuit $\mathtt{RangeQuery}_k$.

Circuit $\mathtt{ModifyArray}_{k,\mathrm{gt}}$
Public input: $H_{tbl}, H'_{tbl}, H_{res}$
Private input: *res*, *res*′

1: Check if $H_{res} = \mathtt{Merkle}(res)$.
2: Parse $res = [(id_i, val_{id_i})]_{0\le i<k}$, $res' = [(id_i, val'_{id_i})]_{0\le i<k}$. Check if $\mathrm{gt}(val_{id_i}) = val'_{id_i}$ for $0 \le i < k$.
3: Check that H_{tbl} will update to H'_{tbl} if *res* is modified to *res*′.
4: **return** 1 iff all checks above pass.

Figure 8: The circuit $\mathtt{ModifyArray}_{k,\mathrm{gt}}$.

The proof of the following theorem is straightforward from the construction, the security of S and DS and the definition of balanced binary search trees (Definition 6.2), and is thus omitted.

Theorem 6.1. *The protocol of Fig. 9 is an index-agnostic verifiable database (per Definition 6.1). Its complexities are as follows (if using* `InputConsist` *from Fig. 5, Groth16-based protocols for* S, DS, *and balanced binary search trees for bst)*

(1) $\mathcal{G}$ runs in $O(n \log n)$ time, outputs pk *of $O(n \log n)$ size, and* vk *of $O(\log n)$ size;*
(2) $\mathcal{P}$ runs in $O(k \log^3 n)$ time and outputs π of $O(1)$ size;
(3) $\mathcal{V}$ runs in $O(k)$ time.

7 Evaluations

In this section, we provide evaluations on Delphus (see Fig. 3).

Comparison with other dynamic zk-SNARKs. We compare Delphus with Dynark [28] and Dynalog [27]. Dynark is publicly verifiable with $\tilde{O}(k\sqrt{n})$ update time and $O(1)$ proof size. Dynalog is also publicly verifiable with $O(k \cdot \mathrm{poly}\log n)$ update time and $O(\mathrm{poly}\log n)$ proof size. We calculate concrete lower bounds of the numbers of their operations and compare them in Table 2. The

- $\mathcal{G}(1^\lambda, tbl) \to (\mathsf{pk}, \mathsf{vk}, \mathsf{st}, H_{tbl})$:
  - For each power of two $k \in [1, n]$, predefined gt, call
    $$\mathsf{S}.\mathcal{G}(1^\lambda, \mathbb{i}^{(k)}_{\mathsf{Rng}}) \to (\mathsf{pk}^{(k)}_{\mathsf{Rng}}, \mathsf{vk}^{(k)}_{\mathsf{Rng}}),$$
    $$\mathsf{S}.\mathcal{G}(1^\lambda, \mathbb{i}^{(k,\mathrm{gt})}_{\mathsf{Mod}}) \to (\mathsf{pk}^{(k,\mathrm{gt})}_{\mathsf{Mod}}, \mathsf{vk}^{(k,\mathrm{gt})}_{\mathsf{Mod}}).$$
  - Call $\mathsf{DS}.\mathcal{G}(1^\lambda, \mathbb{i}_{\mathsf{Ind}}) \to (\mathsf{pk}_{\mathsf{Ind}}, \mathsf{upk}_{\mathsf{Ind}}, \mathsf{vk}_{\mathsf{Ind}})$.
  - Compute $H_{tbl} = \mathtt{MuHash}(tbl)$.
  - For each required index *bst*, compute *bst*, bst_{aux}, $H_{bst} = \mathtt{Merkle}(bst, root)$ (stored in $\mathsf{aux}_{\mathsf{mkl}}$), and call
    $$\mathsf{DS}.\mathcal{P}(\mathsf{pk}_{\mathsf{Ind}}, (H_{tbl}, H_{bst}), (bst, bst_{\mathsf{aux}})) \to (\pi_{\mathsf{Ind}}, \mathsf{aux}_{\mathsf{Ind}}).$$
  - Output $(\mathsf{pk}, \mathsf{vk}, \mathsf{st}, H_{tbl})$ where
    $$\mathsf{pk} = \left(\{\mathsf{pk}^{(k)}_{\mathsf{Rng}}\}_k, \{\mathsf{pk}^{(k,\mathrm{gt})}_{\mathsf{Mod}}\}_{k,\mathrm{gt}}, \mathsf{pk}_{\mathsf{Ind}}, \mathsf{upk}_{\mathsf{Ind}}\right),$$
    $$\mathsf{vk} = \left(\{\mathsf{vk}^{(k)}_{\mathsf{Rng}}\}_k, \{\mathsf{vk}^{(k,\mathrm{gt})}_{\mathsf{Mod}}\}_{k,\mathrm{gt}}, \mathsf{vk}_{\mathsf{Ind}}\right),$$
    $$\mathsf{st} = (tbl, H_{tbl}, \{bst, bst_{\mathsf{aux}}, H_{bst}, \pi_{\mathsf{Ind}}, \mathsf{aux}_{\mathsf{Ind}}\}_{bst}, \mathsf{aux}_{\mathsf{mkl}}).$$
- $\mathcal{P}(\mathsf{pk}, \mathsf{st}, qry) \to (\mathsf{st}', res, H'_{tbl}, \pi)$:
  - Parse l, r from *qry*, and pick the required *bst*.
  - Compute *res* from *bst*, $k = |res|$, *nds* (necessary from *bst*); fetch Merkle proofs from $\mathsf{aux}_{\mathsf{mkl}}$ for *root* and each node in *nds* according to H_{bst} to form π_{nds}.
  - Compute $H_{res} = \mathtt{Merkle}(res)$.
  - $\mathsf{S}.\mathcal{P}(\mathsf{pk}^{(k)}_{\mathsf{Rng}}, (H_{bst}, l, r, H_{res}), (res, nds, root, \pi_{nds})) \to \pi_{\mathsf{Rng}}$.
  - If $qry = (\mathtt{SELECT}, l, r)$,
    - Output $(\mathsf{st}, res, H_{tbl}, \pi = (k, H_{bst}, \pi_{\mathsf{Ind}}, \pi_{\mathsf{Rng}}))$.
  - If $qry = (\mathtt{UPDATE}, l, r, \mathrm{gt})$,
    - Perform gt on *res* to get *res*′, and update *tbl*′, H'_{tbl} based on *res*′; call
      $$\mathsf{S}.\mathcal{P}(\mathsf{pk}^{(k,\mathrm{gt})}_{\mathsf{Mod}}, (H_{tbl}, H'_{tbl}, H_{res}), (res, res')) \to \pi_{\mathsf{Mod}};$$
    - For each index *bst*,
      · Update *bst*′, bst'_{aux}, H'_{bst} based on *res*′;
      · Call $\mathsf{DS}.\mathcal{U}$ to update $(\pi'_{\mathsf{Ind}}, \mathsf{aux}'_{\mathsf{Ind}})$;
    - Update st′ accordingly; output
      $$(\mathsf{st}', [], H'_{tbl}, \pi = (k, H_{bst}, H_{res}, \pi_{\mathsf{Ind}}, \pi_{\mathsf{Rng}}, \pi_{\mathsf{Mod}})).$$
- $\mathcal{V}(\mathsf{vk}, H_{tbl}, H'_{tbl}, qry, res, \pi) \to 0/1$:
  - Check if $\mathsf{DS}.\mathcal{V}(\mathsf{vk}_{\mathsf{Ind}}, (H_{tbl}, H_{bst}), \pi_{\mathsf{Ind}}) = 1$.
  - If $qry = (\mathtt{SELECT}, l, r)$,
    - Compute $H_{res} = \mathtt{Merkle}(res)$;
    - Check if $\mathsf{S}.\mathcal{V}(\mathsf{vk}^{(k)}_{\mathsf{Rng}}, (H_{bst}, l, r, H_{res}), \pi_{\mathsf{Rng}}) = 1$.
    - Check if $H'_{tbl} = H_{tbl}$.
  - If $qry = (\mathtt{UPDATE}, l, r, \mathrm{gt})$,
    - Check if $res = []$.
    - Check if $\mathsf{S}.\mathcal{V}(\mathsf{vk}^{(k)}_{\mathsf{Rng}}, (H_{bst}, l, r, H_{res}), \pi_{\mathsf{Rng}}) = 1$ and if $\mathsf{S}.\mathcal{V}(\mathsf{vk}^{(k,\mathrm{gt})}_{\mathsf{Mod}}, (H_{tbl}, H'_{tbl}, H_{res}), \pi_{\mathsf{Mod}}) = 1$.
  - Output 1 iff all checks above pass.

Figure 9: An index-agnostic VDB based on a zk-SNARK S **and a dynamic designated-verifier zk-SNARK** DS**.**

comparison involves a random circuit of size n where k gates are modified with $3k$ wire values changed.

Based on the benchmarks from zkalc [12], for a single gate change ($k = 1$) and $n = 2^{16}$, the update time in Delphus is $\sim 2\times$ faster than Dynark and at least 19× faster than Dynalog. For $n = 2^{24}$, the update time in Delphus is $\sim 10\times$ faster than Dynark and at least 40× faster than Dynalog. For $n = 2^{31}$, the update time in Delphus is $\sim 100\times$ faster than Dynark and at least 60× faster than Dynalog.

Table 2: Comparison of numbers of operations. Calculations are performed on a random circuit with n gates. For updates, we consider k modified gates with $3k$ wire values changed in the circuit. Numbers for Dynark and Dynalog are concrete lower bounds. $\mathsf{FFT}(n)$ has $O(n \log n)$ field operations.

Scheme	Delphus	Dynark [28]	Dynalog [27]
Prove	$2n \log n + 4n$ $\mathbb{G}_1$ $2n \log n + n$ $\mathbb{G}_2$ 6 $\mathsf{FFT}(n)$	$4n$ $\mathbb{G}_1$ n $\mathbb{G}_2$ 6 $\mathsf{FFT}(n)$	$n \log^2 n$ $\mathbb{G}_1$ n $\mathbb{G}_2$ 6 $\mathsf{FFT}(n)$
Update	$k \log n + 9k$ $\mathbb{G}_1$ $k \log n + 3k$ $\mathbb{G}_2$ Pairing$(2k \log n)$	$3k\sqrt{n \log n} + 15k$ $\mathbb{G}_1$ $3k$ $\mathbb{G}_2$	$4k \log^3 n$ $\mathbb{G}_1$ $k \log^3 n$ $\mathbb{G}_2$
Verify	Pairing(4) 1 $\mathbb{G}_T$	Pairing(4)	Pairing$(4 \log n)$
Proof	480 bytes	192 bytes	$192 \log n$ bytes

Microbenchmarks. We implement[3] Delphus in Rust, using arkworks [11] and BLS12-381 for bilinear pairings. We run our code on an Ubuntu 24.04 with Intel Core i7-1185G7 with 4.8GHz, 4 cores (8 threads) and 16 GB of RAM. We benchmark our implementation in Table 3 on random circuits of size $n = 2^\ell$ with one random gate updated.

Table 3: Single-thread execution for Delphus.

$\ell = \log_2 n$	7	8	9	10	11	12
Prove (s)	0.43	0.88	1.81	3.71	7.51	15.32
Update (ms)	11.98	13.11	10.91	11.68	12.62	17.96
Verify (ms)	2.19					
Proof size	480 bytes					

Acknowledgments

Dimitrios Papadopoulos was supported by the Hong Kong Research Grants Council under grant GRF-16213625. Charalampos Papamanthou was supported by the National Science Foundation and Protocol Labs.

References

[1] Arnon, G., Dujmovic, J., Ishai, Y.: Designated-verifier SNARGs with one group element. In: Kalai, Y.T., Kamara, S.F. (eds.) Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part VII. Lecture Notes in Computer Science, vol. 16006, pp. 192–224. Springer (2025). https://doi.org/10.1007/978-3-032-01907-3_7

[2] Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the AFIPS Spring Joint Computer Conference. vol. 32, pp. 307–314 (1968). https://doi.org/10.1145/1468075.1468121

[3] Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012. pp. 263–280. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[4] Bellare, M., Micciancio, D.: A new paradigm for collision-free hashing: incrementality at reduced cost. In: Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques. pp. 163–192. EUROCRYPT'97, Springer-Verlag, Berlin, Heidelberg (1997)

[5] Beneš, V.E.: Optimal rearrangeable multistage connecting networks. Bell System Technical Journal **43**(4), 1641–1656 (1964)

[6] Bitansky, N., Chiesa, A., Ishai, Y., Ostrovsky, R., Paneth, O.: Succinct non-interactive arguments via linear interactive proofs. In: Sahai, A. (ed.) Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7785. Springer (2013). https://doi.org/10.1007/978-3-642-36594-2_18

[7] Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP). pp. 341–357. ACM (2013)

[8] Bünz, B., Maller, M., Mishra, P., Tyagi, N., Vesely, P.: Proofs for inner pairing products and applications. Cryptology ePrint Archive, Paper 2019/1177 (2019), https://eprint.iacr.org/2019/1177

[9] Clarke, D., Devadas, S., van Dijk, M., Gassend, B., Suh, G.E.: Incremental multiset hash functions and their application to memory integrity checking. In: Laih, C.S. (ed.) Advances in Cryptology - ASIACRYPT 2003. pp. 188–207. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

[10] Clos, C.: A study of non-blocking switching networks. The Bell System Technical Journal **32**(2), 406–424 (1953)

[11] arkworks contributors: `arkworks` zkSNARK ecosystem (2022), https://arkworks.rs

[12] Ernstberger, J., Chaliasos, S., Kadianakis, G., Steinhorst, S., Jovanovic, P., Gervais, A., Livshits, B., Orrù, M.: zk-Bench: A toolset for comparative evaluation and performance benchmarking of SNARKs. In: Galdi, C., Phan, D.H. (eds.) Security and Cryptography for Networks. pp. 46–72. Springer Nature Switzerland (2024)

[13] Fuchsbauer, G., Kiltz, E., Loss, J.: The algebraic group model and its applications. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 33–62. Springer International Publishing, Cham (2018)

[14] Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953 (2019), https://eprint.iacr.org/2019/953

[15] Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract). In: Sedgewick, R. (ed.) Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA. pp. 291–304. ACM (1985). https://doi.org/10.1145/22145.22178

[16] Goodrich, M.T., Tamassia, R., Schwerin, A.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: Proceedings DARPA Information Survivability Conference and Exposition II (DISCEX'01). pp. 68–82. IEEE (2001)

[17] Groth, J.: On the size of pairing-based non-interactive arguments. In: Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II. pp. 305–326 (2016)

[18] Ishai, Y., Su, H., Wu, D.J.: Shorter and faster post-quantum designated-verifier zkSNARKs from lattices. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. pp. 212–234. ACM (2021). https://doi.org/10.1145/3460120.3484572

[19] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) Advances in Cryptology - ASIACRYPT 2010. pp. 177–194. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[20] Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann, San Mateo, CA (1992)

[21] Li, C., Zhang, F.: Designated-verifier zk-SNARKs made easy. Cryptology ePrint Archive, Paper 2024/1153 (2024), https://eprint.iacr.org/2024/1153

[22] Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. In: Pomerance, C. (ed.) CRYPTO '87. pp. 369–378. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)

[23] Papamanthou, C., Srinivasan, S., Gailly, N., Hishon-Rezaizadeh, I., Salumets, A., Golemac, S.: Reckle trees: Updatable Merkle batch proofs with applications. In: Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1538–1551. ACM (2024). https://doi.org/10.1145/3658644.3670354

[24] Peng, Y., Du, M., Li, F., Cheng, R., Song, D.: FalconDB: Blockchain-based collaborative database. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 637–652. ACM (2020)

[25] Sherman, A.T., Lanus, E., Liskov, M., Zieglar, E., Chang, R., Golaszewski, E., Wnuk-Fink, R., Bonyadi, C.J., Yaksetig, M., Blumenfeld, I.: Formal methods analysis of

[3] https://github.com/wangnick2017/Delphus

the secure remote password protocol (2020), https://arxiv.org/abs/2003.07421
[26] Valiant, L.G.: A scheme for fast parallel communication. SIAM Journal on Computing **11**(2), 350–361 (1982)
[27] Wang, W., Papamanthou, C., Srinivasan, S., Papadopoulos, D.: Dynamic zk-SNARKs (with applications to sparse zk-SNARKs and IVC). Cryptology ePrint Archive, Paper 2024/1566 (2024), https://eprint.iacr.org/2024/1566
[28] Zhang, T., Ouyang, Y., Zhang, Y.: Dynark: Making Groth16 dynamic. Cryptology ePrint Archive, Paper 2025/1897 (2025), https://eprint.iacr.org/2025/1897
[29] Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 863–880 (2017)
[30] Zhang, Y., Katz, J., Papamanthou, C.: IntegriDB: Verifiable SQL for outsourced databases. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1480–1491. ACM (2015)

Generative AI Usage

Gemini 3 is used for grammar editing and the generation of Fig. 11. AI-generated content is validated by manual verification.

Open science

The implementation of Delphus, a designated-verifier dynamic zk-SNARK for circuit satisfiability can be found at https://github.com/wangnick2017/Delphus.

Ethical considerations

This work contains only cryptographic protocols (zk-SNARKs) with application to verifiable databases. The ethical implication is not beyond the research of cryptography.

A Additional preliminaries

zk-SNARKs. We now present the definition of circuit-specific zk-SNARKs [17]. For zk-SNARKs with universal setup, algorithm $\mathcal{G}$ below is separated into two algorithms, a universal generation $\mathcal{G}(1^\lambda, |\mathbb{i}|) \to \mathsf{pp}$ and an indexer $\mathcal{I}(\mathsf{pp}, \mathbb{i}) \to (\mathsf{pk}, \mathsf{vk})$. In both circuit-specific and universal zk-SNARKs, algorithm $\mathcal{G}$ must be trusted.

Definition A.1 (zk-SNARKs). *A zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) for an indexed relation $\mathcal{R}$ is a tuple of PPT algorithms $\mathsf{S} = (\mathcal{G}, \mathcal{P}, \mathcal{V})$ as follows:*

- $\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{vk})$: *Outputs prover key* pk*, verifier key* vk*.*
- $\mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}) \to \pi$: *Given proving key* pk*, instance* $\mathbb{x}$*, and witness* $\mathbb{w}$*, outputs proof* π*.*
- $\mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to 0/1$: *Given verification key* vk*, instance* $\mathbb{x}$*, and a proof* π*, outputs accept or reject.*

A zk-SNARK S *should have polylog-sized proofs and satisfy:*

- Completeness*: Let* $\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{vk})$*. We say that* S *satisfies* completeness *if for any* $\mathbb{i}$*, for any* $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$*, if* $\pi \leftarrow \mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w})$*, then* $\mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to 1$*.*
- Knowledge Soundness*: We say that* S *satisfies* knowledge soundness *if for any PPT adversary* $\mathcal{A}$ *and for any* $\mathbb{i}$ *there exists a PPT extractor* $\mathcal{E}_\mathcal{A}$ *such that the following is negligible:*

$$\Pr\left[\begin{array}{c:c} \mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to 1 \wedge & \mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{vk}); \\ (\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R} & ((\mathbb{x}, \pi); \mathbb{w}) \leftarrow (\mathcal{A} || \mathcal{E}_\mathcal{A})(\mathsf{pk}, \mathsf{vk}) \end{array}\right]$$

- Zero Knowledge*: Fix any* $\mathbb{i}$ *and* $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$*. Let* $\mathcal{D}$ *be the distribution of* $(\mathsf{pk}, \mathsf{vk}, \pi)$ *as output by the experiment below. (1)* $\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{vk})$*. (2)* $\pi \leftarrow \mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w})$*.*
  We say that S *satisfies* zero-knowledge *if there exists a PPT simulator* $\mathcal{S}$ *such that, for every* $\mathbb{i}$ *and* $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$ *as above, the distribution* $\tilde{\mathcal{D}}$ *of* $(\mathsf{pk}, \mathsf{vk}, \tilde{\pi})$ *output by the following is computationally indistinguishable from* $\mathcal{D}$*.*
  (1) $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{S}(1^\lambda, \mathbb{i})$*. (2)* $\tilde{\pi} \leftarrow \mathcal{S}(\mathsf{pk}, \mathsf{vk}, \mathbb{x})$*.*
  The zero-knowledge definition naturally extends to statistical/perfect zero-knowledge.

Groth16 [17]. Groth16 is a zk-SNARK (see Definition A.1) for $\mathcal{R}_{\mathsf{R1CS}}$. For $\mathbb{i} = (\mathbb{F}, n, m, k, \mathbf{A}, \mathbf{B}, \mathbf{C})$ and $\mathbb{z} = (1, \mathbb{x}, \mathbb{w})$, define for each $0 \le i < n$ and $0 \le j < m$,

$$u_j(X) = \sum_{0 \le i < n} \mathbf{A}[i][j]\mathcal{L}_i(X)\,,\ a(X) = \sum_{0 \le j < m} \mathbb{z}[j] \cdot u_j(X)\,,$$
$$v_j(X) = \sum_{0 \le i < n} \mathbf{B}[i][j]\mathcal{L}_i(X)\,,\ b(X) = \sum_{0 \le j < m} \mathbb{z}[j] \cdot v_j(X)\,,$$
$$w_j(X) = \sum_{0 \le i < n} \mathbf{C}[i][j]\mathcal{L}_i(X)\,,\ c(X) = \sum_{0 \le j < m} \mathbb{z}[j] \cdot w_j(X)\,.$$

Note that $a(\omega^i) = \sum_{0 \le j < m} \mathbb{z}[j] \cdot \mathbf{A}[i][j]$ (and similar for b and c). The existence of $\mathbb{z}$ such that $(\mathbf{A} \cdot \mathbb{z}) \circ (\mathbf{B} \cdot \mathbb{z}) = \mathbf{C} \cdot \mathbb{z}$ is equivalent to the existence of a quotient polynomial $q(X)$ such that $a(X) \cdot b(X) - c(X) = q(X) \cdot (X^n - 1)$. Based on these, the whole Groth16 protocol is presented in Fig. 10.

- $\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{pk}, \mathsf{vk})$:
  - $\mathsf{pp}_{\mathsf{bl}} \leftarrow \mathsf{GroupGen}_{\mathsf{bl}}(1^\lambda)$.
  - Pick $\alpha, \beta, \gamma, \delta, \tau \xleftarrow{\$} \mathbb{F}$.
  - Let $t_j(X) = \beta u_j(X) + \alpha v_j(X) + w_j(X)$ for $0 \le j < m$.
  - Output

$$\mathsf{pk} = \begin{pmatrix} [\alpha]_1, [\beta]_1, [\beta]_2, [\delta]_1, [\delta]_2, \\ \left([u_j(\tau)]_1, [v_j(\tau)]_1, [v_j(\tau)]_2\right)_{0 \le j < m}, \\ \left(\left[t_j(\tau)/\delta\right]_1\right)_{k \le j < m}, \left(\left[\tau^i(\tau^n - 1)/\delta\right]_1\right)_{0 \le i \le n-2} \end{pmatrix},$$
$$\mathsf{vk} = \left([\alpha]_1, [\beta]_2, [\gamma]_2, [\delta]_2, ([t_j(\tau)/\gamma]_1)_{0 \le j < k}\right).$$

- $\mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}) \to \pi$:
  - Form $\mathbb{z} = (1, \mathbb{x}, \mathbb{w})$. Pick $r, s \xleftarrow{\$} \mathbb{F}$.
  - Output $\pi = ([A]_1, [B]_2, [C]_1)$ where

$$[A]_1 = [\alpha]_1 + r[\delta]_1 + \sum_{0 \le j < m} \mathbb{z}[j] \cdot [u_j(\tau)]_1\,,$$
$$[B]_2 = [\beta]_2 + s[\delta]_2 + \sum_{0 \le j < m} \mathbb{z}[j] \cdot [v_j(\tau)]_2\,,$$
$$[C]_1 = s[A]_1 + r[B]_1 - rs[\delta]_1 + \left[\frac{q(\tau)(\tau^n - 1)}{\delta}\right]_1 + \sum_{k \le j < m} \mathbb{z}[j] \left[\frac{t_j(\tau)}{\delta}\right]_1.$$

- $\mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to 0/1$:
  - Parse $\pi = ([A]_1, [B]_2, [C]_1)$ and check if

$$e([A]_1, [B]_2) \stackrel{?}{=} e([\alpha]_1, [\beta]_2) + e([C]_1, [\delta]_2) + e\left(\sum_{0 \le j < k} \mathbb{z}[j] \cdot \left[\frac{t_j(\tau)}{\gamma}\right]_1, [\gamma]_2\right).$$

Figure 10: Groth16 [17]: A zk-SNARK for $\mathcal{R}_{\mathsf{R1CS}}$.

Lemma A.1 ([13, 17]). *Groth16 (Fig. 10) is a zk-SNARK (per Definition A.1) for $\mathcal{R}_{R1CS}$ on (q_1, q_2)-**dlog** (per Assumption 2.1) in AGM.*

B Security proofs

B.1 Proofs for Section 5

B.1.1 Proof of Lemma 5.1.

Proof. The direction $\Rightarrow$ is trivial. Consider the other direction. Let S_L be the multiset on the LHS of the *Input Consistency* equation, and S_R be the multiset on the RHS. The equality $S_L = S_R$ implies that for every tuple $(i, min_i, max_i, size_i)$ in S_L $(0 \le i < n)$, there is exactly one matching tuple in S_R.

- If the matching tuple comes from the root, then $i = root$.
- If the matching tuple comes from a left child term of some parent p, then $i = left_p$. Crucially, we have:

$$min_i = minLeft_p \quad \text{and} \quad max_i = maxLeft_p \tag{5}$$

- Otherwise (from a right child), $i = right_p$, and:

$$min_i = minRight_p \quad \text{and} \quad max_i = maxRight_p \tag{6}$$

Since every node appears exactly once on the RHS, every node $i \neq root$ has a unique parent p and $root$ has none, and the matching also gives $size_i = sizeLeft_p$ if $i = left_p$ (resp. $size_i = sizeRight_p$ if $i = right_p$). Let U be the set of nodes not reachable from $root$; every parent and every child of a node in U is in U. Summing Node Consistency (3) over all $i \in U$ and replacing each claimed child size by the size of the child it names, every $size_j$ with $j \in U$ appears exactly once on each side, so all sizes cancel and $|U| = 0$ in the field. Since the characteristic of the field exceeds $n \ge |U|$, $U = \emptyset$, and the structure forms a rooted tree covering all n nodes.

Now we prove by induction that for every node i, the sub-tree rooted at i satisfies the BST property (Definition 5.2) and that min_i, max_i are true minimum/maximum values of that sub-tree.

Base Case (Leaf Node): Suppose i is a leaf ($left_i$, $right_i$ are null). By Node Consistency condition (1)(2), $min_i = max_i = val_i$.

Inductive Step: Assume it holds for the children of a node p.

If p has a left child $L = left_p$, by the inductive hypothesis, L is a valid BST with values in range $[min_L, max_L]$. From Eq. (5), we know the parent correctly views the child's attributes:

$$minLeft_p = min_L, \quad maxLeft_p = max_L$$

From Node Consistency (1) for p:

$$min_p = minLeft_p = min_L,\ maxLeft_p \le val_p \implies max_L \le val_p$$

Since max_L is the true maximum of the left subtree, all values in the left subtree are $\le val_p$. This satisfies the left-side BST ordering condition. Furthermore, $\min_p$ is correctly set to the minimum of the left subtree, which is the minimum of the subtree rooted at p.

Similar and symmetric analysis applies if p has a right child.

Conclusion: For any node p, the values in its left subtree are $\le val_p$ and the values in its right subtree are $\ge val_p$. Thus, the ordered list forms a valid Binary Search Tree according to Definition 5.2. □

B.1.2 Proof of Theorem 5.1.

Proof. First, we establish the equivalence between the circuit acceptance and the relation $\mathcal{R}_{\text{valid}}$.

($\Rightarrow$): Assume $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\text{valid}}$. By the definition of $\mathcal{R}_{\text{valid}}$, we have (1) bst is a valid binary search tree; (2) $H_{tbl} = \texttt{MuHash}(\{(i, val_i)\})$; (3) $H_{bst} = \texttt{Merkle}(bst, root)$. In the circuit C (Fig. 4), Lines 1 and 2 verify the digests. Since conditions (2) and (3) hold, these checks pass. Line 3 verifies *Node Consistency* for all i. Since bst is a valid BST, the local properties hold by definition. Line 4 verifies *Input Consistency*. Since bst is a valid BST, by Lemma 5.1, a valid witness bst_{aux} exists for `InputConsist`. Therefore, C outputs 1.

($\Leftarrow$): Assume C outputs 1. Lines 1-2 imply H_{tbl} and H_{bst} are the correct digests. Line 3 implies the list satisfies *Node Consistency* (local parent-child constraints). Line 4 implies *Input Consistency* (multiset equality). By Lemma 5.1, a structure satisfying both *Node Consistency* and *Input Consistency* is a valid binary search tree whose root is $root$, since no child tuple names $root$. Thus, all conditions for $\mathcal{R}_{\text{valid}}$ are met.

Complexities. The size of C is the sum of its components:

- `MuHash` and `Merkle`: The product tree and the Merkle tree over n entries require $O(n)$ gates.
- The loop at Line 3 runs n times. Each iteration performs $O(1)$ arithmetic comparisons. Total: $O(n)$.
- `InputConsist`: Given size $s(n)$.

The total size is $s(n) + O(n) + O(n) = s(n) + O(n)$.

For the update locality, consider a single update to the input bst (changing one node $node_i$):

- `Merkle`: Changing one leaf in a Merkle tree affects only the path to the root. This updates $O(\log n)$ gates.
- The loop at Line 3: Changing $node_i$ only affects the i-th check. This updates $O(1)$ gates.
- `InputConsist`: Given locality $t(n)$.

The total is $t(n) + O(\log n) + O(1) = t(n) + O(\log n)$.

□

C Sorting Circuit via Randomized Routing

Our goal is to build a circuit that sorts a list of pairs $(id_i, data_{id_i})_i$ by id_i, such that the circuit supports dynamic updates (Definition 5.1). Many existing sorting circuits (e.g., Batcher's bitonic sorter [2]) suffer from instability: a single change in the input set may necessitate $O(n)$ wire changes. This is because they are based on comparisons. Hence, we consider building circuits based on radix and assume for simplicity that each id is unique and $0 \le id < n$.

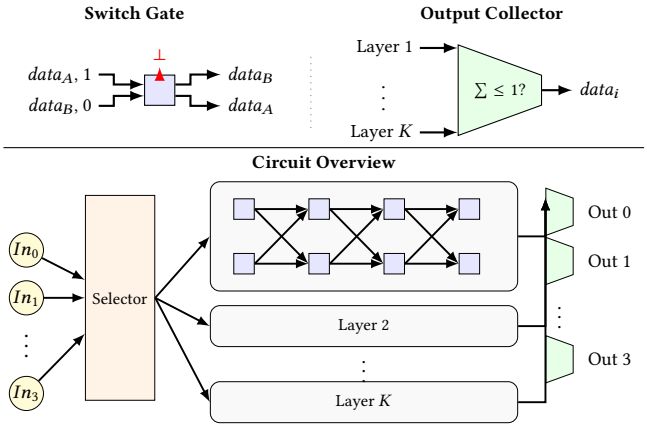
Butterfly/Beneš networks [5, 20] offer an efficient routing mechanism ($O(\log n)$ depth), but suffer from either collisions in the worst case or instability for the control bits. According to [10], simple deterministic strictly non-blocking networks (e.g., Clos) often require $\Omega(n\sqrt{n})$ size. To break this barrier, inspired by Valiant's randomized routing scheme [26], we propose a sorting circuit via randomized routing where the prover picks some random bits by itself and handles the collisions.

Our circuit is presented in Fig. 11. Specifically,

- *Key primitive: Switch gate.* It takes as input $data_A$, $data_B$ and two routing bits b_A, b_B. The routing bit indicates whether the data should go up (0) or down (1), e.g., if $b_A = 1$ and

$b_B = 0$, then the gate outputs $data_B$ in the up wire and $data_A$ in the down wire. If both data are not NULL and $b_A = b_B$, then a collision occurs and this gate triggers "ERROR".

- *Main component: Beneš network [5].* On input n data, it consists of $2\log n$ sub-layers of $n/2$ identical *Switch* gates, each with two routing bits as additional input, arranged symmetrically. The first $\log n$ sub-layers form a butterfly-like network that recursively routes inputs to an intermediate stage, while the remaining $\log n$ sub-layers mirror this structure in reverse. The output is a permutation of the input. A fundamental property is that for any desired output, there exists a configuration of the routing bits that realizes it (see [5] for details). Importantly, circuit structure is fixed and independent of the data being routed.
- *Layer selection.* We build $K = 4\log n$ independent copies of the *Beneš network*, each as one layer. For i-th global input of the circuit $data_{id_i}$, it has a layer selection $s_i \in [1, K]$ as additional input. There is a selector that forwards $data_{id_i}$ to the i-th input of layer s_i and sets the i-th input of other layers as NULL.
- *Output collector.* This collects the i-th output of all layers, checks that at most one of them is not NULL and forwards it as the i-th output of the whole circuit.



Circuit `SortWithRandomRouting`
Global input: $data_{id_i}$ for $0 \le i < n$
Additional input: (all in binary bits) s_i, ρ_i, id_i for $0 \le i < n$

1: Selector dispatches $data_{id_i}$ to layer s_i for $0 \le i < n$.
2: Beneš network routing: For each $0 \le i < n$, layer s_i routes $data_{id_i}$ to ρ_i in the intermediate stage and to id_i as output. Abort with "ERROR" if collision occurs at any *Switch* gate.
3: Check and collect outputs from all layers: If there is exact one layer with non-NULL i-th output, take it as the i-th output of the circuit.

Figure 11: Sorting Circuit via Randomized Routing.

The prover prepares circuit inputs: for each $(id_i, data_{id_i})$,

- *The global input* $data_{id_i}$. This is the i-th input of the selector.
- *Additional inputs as a path.* The prover randomly picks $s_i \in [1, K]$ for layer selection and $\rho_i \in [0, n-1]$ as the location in the intermediate stage. This leads to a unique path for $data_{id_i}$ in the circuit:

$$i \text{ at input} \to i \text{ at Layer } s_i \to \rho_i \text{ at Layer } s_i \to id_i \text{ at Layer } s_i \to id_i \text{ at output}$$

If this path collides with any other data at any *Switch* gate, retry to pick new s_i and ρ_i; otherwise put s_i in the selector and fill all the routing bits on the path using ρ_i and id_i.

Theorem C.1. *The circuit C in Fig. 11 satisfies:*

(1) ***Completeness:*** *For valid $(id_i, data_{id_i})_i$, with $K = \Omega(\log n)$ layers, there exists a routing configuration that C accepts.*
(2) ***Soundness:*** *If the circuit outputs a result (i.e., does not trigger "ERROR"), the output is a correct sorting of $(id_i, data_{id_i})_i$.*
(3) ***Prover complexity:*** *The expected number of trials for the prover to successfully route a specific packet (data) is $O(1)$.*
(4) *C has size $O(n\log^2 n)$ and update locality $O(\log n)$.*

Proof. For completeness, see the analysis of randomized routing on Hypercube-derivative networks (e.g., the Butterfly/Beneš) initiated by Valiant [26]. The soundness relies on the deterministic verification logic embedded in the *Switch* gates and the *Output Collector*.

- The *Switch* gate logic ensures that data flows according to the routing bits without collision.
- The Beneš network topology guarantees that a path from the control bits is correctly routed to the destination.
- The *Output Collector* at output j verifies that at most one layer has provided a non-NULL data, ensuring that exactly one data claiming destination j has arrived.

Thus, if the circuit does not abort, the output at port j is $data_j$.

Prover complexity. We analyze the probability that an honest prover fails to find a valid routing for $data_{id_i}$, assuming the circuit already contains $n-1$ other routed data.

Let the routing path for $data_{id_i}$ be determined by the intermediate destination ρ_i, chosen uniformly at random from $[0, n-1]$. This path, denoted as $P(\rho_i)$, traverses the Beneš network which consists of $2\log n$ stages. Let X be the random variable representing the number of existing data that share at least one wire with path $P(\rho_i)$. Following the analysis of randomized routing on Beneš networks (e.g., [20, 26]), for any specific wire e in the network, the expected number of paths passing through e when routing a permutation is 1. By linearity of expectation, the expected total number of conflicts is the sum of the expected congestion on each edge of the path:

$$\mathbb{E}[X] \le \sum_{e \in P(\rho_i)} 1 = 2\log n - 1 .$$

The prover selects a layer $s_i \in [1, K]$ uniformly at random. A collision occurs if the selected layer s_i is already occupied by any of the X conflicting packets. Let C be the event that a collision occurs. The probability of collision is:

$$\Pr[C] = \Pr[s_i \in \{s_j \mid j \text{ conflicts with } i\}] \le \frac{\mathbb{E}[X]}{K}.$$

Substituting the bound for $\mathbb{E}[X]$, we have $\Pr[C] < \frac{2\log n}{K} = \frac{1}{2}$. Since the probability of success is strictly greater than $1/2$, the expected number of trials to find it is upper bounded by $2 = O(1)$.

□