Aardvark: An Asynchronous Authenticated Dictionary with Applications to Account-based Cryptocurrencies

Derek Leung
MIT CSAIL[*]

Yossi Gilad
Hebrew University of Jerusalem

Sergey Gorbunov
University of Waterloo

Leonid Reyzin
Boston University

Nickolai Zeldovich
MIT CSAIL

Abstract

We design Aardvark, a novel authenticated dictionary with short proofs of correctness for lookups and modifications. Our design reduces storage requirements for transaction validation in cryptocurrencies by outsourcing data from validators to untrusted servers, which supply proofs of correctness of this data as needed. In this setting, short proofs are particularly important because proofs are distributed to many validators, and the transmission of long proofs can easily dominate costs.

A proof for a piece of data in an authenticated dictionary may change whenever any (even unrelated) data changes. This presents a problem for concurrent issuance of cryptocurrency transactions, as proofs become stale. To solve this problem, Aardvark employs a versioning mechanism to safely accept stale proofs for a limited time.

On a dictionary with 100 million keys, operation proof sizes are about 1KB in a Merkle Tree versus 100–200B in Aardvark. Our evaluation shows that a 32-core validator processes 1492–2941 operations per second, saving about 800× in storage costs relative to maintaining the entire state.

1 Introduction

Ensuring the integrity of vast record collections is a fundamental problem in computer security. In 1991, Blum et al. [10] described a scenario where a trustworthy computer executes programs on data in untrusted storage. To prevent the storage from tampering with the data, they introduced a data structure, now known as a two-party *authenticated dictionary* [24, 37], which allows the trusted computer to ensure data integrity by storing a small *commitment* to the data. When the computer reads from the data, it requires the untrusted storage to provide a *proof* guaranteeing correctness; this proof is verified using the commitment. On writes, the computer also receives a proof and, using the proof and the current commitment, computes the new commitment to the modified data.

Authenticated data structures have been proposed for dramatically reducing storage costs in *cryptocurrencies* [12, 15, 33, 47, 50]. In a cryptocurrency, any user can submit a transaction (such as sending money to another user). Transactions get distributed to numerous validators, who run a fault-tolerant protocol to authenticate them, check their validity against the system's current state (e.g., to prevent overspending), and serialize them. In the typical deployment, each stores the entire state of the system in order to validate transactions.

We present Aardvark, an authenticated dictionary for cryptocurrencies that substantially reduces validator storage (to enable so-called *stateless validation*). Aardvark supports many validators and many concurrently-issued transactions per second. We specifically target *account-based* cryptocurrencies (e.g., Ethereum [51], Coda [13], Algorand [22]), which maintain a dictionary that maps account identifiers (e.g., public keys) to account data (e.g., balances). Our design solves several key challenges present in this environment.

First, network bandwidth is at a premium because transactions and their proofs must be observed by many validators. Supporting short proofs is a particular challenge. In Aardvark, proofs are roughly 10× shorter than in typical Merkle-tree-based authenticated data structures because its design is based on pairing-based vector commitments with the shortest known proofs. Vector commitments do not provide a dictionary interface (instead, they provide authenticated fixed-length arrays), so much of our work involves building a dictionary interface from this more limited functionality.

Second, the ability of untrusted clients to issue transactions requires the data structure to remain efficient no matter how it is used. Aardvark enforces tight upper bounds on resource use even under adversarial access patterns.

Third, multiple clients may issue transactions concurrently, but validators must order them in a sequence, which presents a problem for the validity of the proofs. For example, suppose that one user issues a transaction that modifies a record, and then another user issues a separate transaction before the validators process the first transaction. If both transactions operate on the same record, then once the first transaction is

[*]The authors completed much of this work at Algorand, Inc.

processed, the data present in the second transaction becomes stale. Moreover, even if both transactions access different records, the proofs present in the second transaction may be invalidated after the first transaction is processed since a proof may depend on the entire dictionary state. Recomputing or updating proofs as they become outdated while a transaction is queued up (as suggested, for example, by [15]) can lead to congestion collapse: more transactions will be queued up when the system is under heavy load, causing the system to slow down further as it updates the proofs of these enqueued transactions. Instead of attempting to reduce these costs, Aardvark avoids them entirely by *versioning* the dictionary. This mechanism enables the system to safely and efficiently check stale proofs against an old dictionary state by maintaining a small cache of recent modifications.

Contributions. We present the following contributions.

1. An authenticated dictionary, backed by vector commitments, which possesses tight upper-bounds on excess resource use even under adversarial access patterns (§5).
2. A versioning mechanism for this dictionary, so that it supports concurrent transaction issuance (§6).
3. Techniques to improve the dictionary's availability (§7).
4. An implementation of our design, integrated into the storage backend of a cryptocurrency, and its evaluation (§8).
5. A rigorous definition and security analysis of the soundness and completeness properties of our versioned authenticated dictionary (§3.2 and Appendix A).

We survey related work next (§2), present an overview of the system's architecture (§3), review vector commitments (§4), and then describe our contributions in detail.

2 Related Work

Our work focuses on short proofs since bandwidth is often a bottleneck for transaction throughput in cryptocurrencies [18]. We focus on concrete gains in efficiency, working towards making stateless validation practical for a high-throughput cryptocurrency: proof sizes are comparable to transaction sizes (100–200B per operation), validator storage costs are reduced by a large constant fraction (800×) regardless of adversarial behavior, and computational costs on the critical path are mostly parallelizable and allow a 32-core machine to process more than 1000 operations per second (§8).

Stateless validation trades validator storage for bandwidth (taken up by proofs) and computation (required for proving and verifying). These tradeoffs do not reduce aggregate system costs in all settings. We now survey prior work, which targets various other points on the tradeoff curve.

Related work in the UTXO Model. Cryptocurrencies that work in the UTXO model organize information around transactions rather than around accounts. The state of the system at a given time is characterized by a set: namely, the set of transaction outputs that have not yet been used as inputs to new transactions. Thus, each transaction needs to reference a prior transaction and, for stateless validation, provide proof that this prior transaction's output has not yet been used. Stateless validation in such designs requires an authenticated version of the dynamic set data structure (often called a dynamic accumulator). There have been many proposals to utilize some version of the Merkle hash tree [32] for this purpose — see, for example, [17, 21, 33, 45, 46].

Merkle-tree-based constructions add substantial network bandwidth costs to the cryptocurrency, which may become a bottleneck in transaction throughput [18]. For example, Merkle proofs at 128-bit security for a dynamic set of 100 million entries, the approximate size of the Bitcoin UTXO set as of the time of writing, are about 1KB in size (the exact number depends on the underlying tree structure used). In contrast, a small transaction (e.g., transferring money in a cryptocurrency) may be as short as 100B, so proofs using Merkle trees impose a bandwidth overhead of 10x. On the other hand, Merkle proofs are fast to produce and verify.

To mitigate the problem of long proofs, [21] proposes some optimizations based on proof aggregation and local caching of the tree nodes, adding some state back to the clients. Other cryptographic accumulators are suitable as well; a discussion of options is available in [20]. In particular, a design based on groups of unknown order (such RSA and class groups) is proposed in [12, §6.1]. (Each proof in this design is over 1KB, but it allows for aggregating proofs for multiple transactions.)

Related work in the Account-Based Model. Because account-based cryptocurrencies maintain a mapping of accounts (usually identified by public keys) to account data (such as balances), it is natural to use authenticated dictionaries to reduce the client state. Most constructions of authenticated dictionaries [15, 24, 34, 37, 40, 50] are based on some variant of Merkle trees [32] and also suffer from the problem of long proofs, imposing 10–20× bandwidth overhead for a simple transaction that involves accessing two keys in the dictionary (e.g., source and destination accounts).

Some constructions of authenticated dictionaries that are not based on Merkle trees are static, without provisions for efficient updates to the data (e.g., [27]). Some early dynamic constructions not based on Merkle trees (e.g., [39] and [19, §4]) require the verifier to keep secrets from the prover to prevent the prover from cheating.[1] Thus, they are not suitable for cryptocurrency applications, as cryptocurrency validators are public and thus cannot be trusted with secrets.

Concerto [4] defers verification to the end, incurring a linear cost (in the dictionary size) once instead of small costs all the time. This approach is not suitable for cryptocurrencies as transactions need to be processed in real time.

[1] To be exact, [39] can work without verifier secrets, but dictionary updates become computationally expensive if proof sizes are kept small (on the order of seconds per update for proof sizes under 1KB per [39, Tables 2 and 3]).

Boneh, Bünz, and Fisch [12, §5.4, §6.1] propose the first authenticated dictionaries that require no verifier secrets and do not use Merkle trees; instead, they rely on factoring-based (or class-group-based) accumulators and vector commitments. They also suggest (but do not investigate the costs of) using such dictionaries to implement stateless validation. The proof length in their construction is over 1KB and thus comparable to Merkle trees (at 128-bit security). However, it is possible to batch proofs for multiple operations into a single constant-size proof. Follow-up works [1,49] improve various dimensions of [12] in important ways, adding functionality and efficiency; however, the proofs are still over 500 bytes long.

Tomescu et al. [48] address a somewhat different problem of append-only authenticated sets and dictionaries using bilinear pairings; their proofs (for different operations than simple reads and writes) are a few kilobytes in length for reasonable security parameters and number of accounts.

An elegant solution called EDRAX that avoids authenticated dictionaries was proposed by Chepurnoy et al. [17]. EDRAX assigns sequential integers (indices into an array) as account identifiers, thus simplifying the necessary data structure from an authenticated dictionary to a vector commitment. The approach adds complexity around new account registration: instead of simply issuing a transaction that sends money to a new public key (as in Aardvark), EDRAX requires every new public key to register via an initialization transaction (preventing the use of this transaction as a denial-of-service vector is discussed in [47, §4.2.4]). Other transactions to this public key must know the index and cannot be processed until this registration happens; at the same time, registration is computationally expensive, as we discuss later in this section.

Public Parameters and Commitment Size. Some pairing-based vector commitment schemes require a trusted set of public parameters. In known pairing-based vector commitments, these parameters are at least proportional to the vector length. EDRAX (and its improvement [47]) requires public parameters that are proportional to the number of accounts in the system because all the data is stored as a single vector. In contrast, Aardvark uses multiple vectors (which share the same parameters). This approach avoids an *a priori* bound on the number of accounts in the system and does not require lengthy parameter generation. However, Aardvark's validators need to store multiple vector commitments instead of just one. For reasonable vector lengths, this allows validators to store a small fraction of the total state.

Shrinking Long Proofs Generically with SNARKs. Any long proof for any authenticated data structure can, in principle, be shrunk to constant size by an application of succinct non-interactive arguments of knowledge (SNARKs), which are general-purpose tools that can apply to any computation.

Current state-of-the-art SNARKs [26] have proofs that are quite short—just three points on a pairing-friendly elliptic curve, or about 144 bytes for 128-bit security. This approach is taken in Zcash [8] for UTXO-based cryptocurrencies (note that Zcash has privacy as its primary goal, which introduces additional complexity into the design). EDRAX [17] suggests it for account-based cryptocurrencies. SNARK verification is quite efficient, requiring a number of elliptic curve point multiplications proportional to the size of the statement being proven and just one product of three elliptic curve pairings [26, Tables 1 and 2]. The drawback of SNARK-based approaches is in the cost of proving. Computing a SNARK is time-consuming. For example, a SNARK for the (admittedly complex) statement proving a Zcash transaction is reported to take 3s [9, §1], and a SNARK for the vector commitment proof of EDRAX is reported to take 77s [36]. (See §8.1 for a more detailed comparison of Aardvark and EDRAX).

With the advantages of near-constant verification time, several proposals (often called "Rollups") batch multiple transactions together in a single proof (e.g., see [16,23,31]). While providing significant savings in verifier time, such proposals incur high prover latency because a prover must collect many transactions before producing a proof, and the per-transaction proving costs are on the order of a second [28, Fig. 7] and hard to parallelize.

SNARKs are an active research area, and SNARK-based approaches for stateless validation are likely to improve, potentially outperforming Aardvark in some settings. In particular, Ozdemir et al. report [35, Fig. 6] that a single Merkle proof (when a hash function is chosen to be particularly SNARK-friendly) can be converted into a SNARK in about 100ms.

Who supplies the proofs? Any approach to stateless validation must address the question of who is responsible for maintaining the outsourced data and supplying the proofs necessary for each transaction. We reiterate that proofs are not static: even if a user's account data does not change, the proof of correctness of that data changes as other accounts change (and cause the commitment to change).

Zcash and EDRAX require users to store proofs and keep them fresh by synchronizing them against the cryptocurrency's transactions. Implicitly, clients depend on untrusted storage, as if they miss an intervening transaction while offline, they must obtain it from the storage to catch up.

This design presents problems for new account registration in EDRAX described earlier. There is no one to keep a proof up to date for an index that is not yet attached to an account. A new user must either compute this proof by reading every transaction from the beginning or rely on a server who knows the entire current vector to provide the proof [36]. Either way, the cost is at least linear in the number of accounts. Aardvark's approach of using multiple short vector commitments may also be applied to EDRAX to reduce this cost.

Since EDRAX is based on vector commitments with long public parameters (proportional to the number of accounts in the system), an important feature of EDRAX's design is ensuring that each user needs only a small subset of the public parameters to synchronize a proof. Tomescu et al. [47] im-

prove this vector commitment, achieving constant-size proofs and requiring only constant-size portions of the public parameters for the relevant operations.

A disadvantage of this approach in the account-based model is that it limits the kinds of changes a transaction can cause to the recipient account. A sender of the transaction cannot supply a proof for the state of the recipient's account (neither the old one nor the updated one), which means that the only modifications to the recipient account that can be made by a transaction are ones that do not require knowledge of the account state. Modifying a committed state without knowing it requires some kind of homomorphism of the commitments. EDRAX relies on the additive homomorphism of its vector commitments to enable the addition of funds to an account with unknown state, so a transaction can do nothing but add funds to the recipient. Thus, more sophisticated operations (e.g., smart contracts) are not supported. Similarly, UTXO-based Zcash can only give a recipient a transaction output, rather than modify the recipient's state in a more general way.

In contrast, Aardvark outsources the problem of computing and supplying the proofs to untrusted archival nodes. This design enables us to have arbitrary updates to account information of both senders and recipients.

Out-of-order transaction processing and stale proofs. Recall that transactions are issued asynchronously and then ordered by the cryptocurrency consensus mechanism. Since the correctness of account data changes as other accounts change, the proofs may become stale by the time transactions with proofs reach a validator. Mechanisms for handling stale proofs are necessary for ensuring that transactions do not get rejected unnecessarily, which may severely limit the cryptocurrency's throughput—especially if the time to compute and transmit the proofs is significant. At the same, these mechanisms cannot be too permissive because they must prevent double-spending. Thus, validators cannot simply accept an account state authenticated by a stale proof, even if it is valid with respect to a recent commitment value, because the state could have changed since the proof was issued.

Most related work does not address this problem, instead assuming that the proofs reach the validators before becoming obsolete (e.g., [17] and [8, §8.3.2]). Aardvark's caching mechanism addresses this problem (§6).

3 Overview

In a cryptocurrency, a collection of *validators* jointly maintain the state. The current state is defined by a well-known ("genesis") initial state modified by a sequence of atomic *transactions*. Clients submit transactions, and the validators execute a protocol to verify these transactions and append them to the sequence if they are valid. This protocol is designed to ensure consensus, so that all validators agree on the same public sequence of transactions. Transactions are called *confirmed* when they are appended to this sequence.

In order to decide whether to accept a transaction, the validators need to know the current state of the system. For instance, if Alice issues a transaction that spends more money than she has, the validators must reject this transaction; in order to do that, they need to know Alice's current balance.

Aardvark enables the validators to use the system state without requiring them to store it. Instead, untrusted *archives* maintain this state, while validators maintain only a small commitment that evolves with the state. To issue a transaction, a client queries an archive for data related to the transaction. The archive returns the data and authenticating information called a *context*[2]. The client submits the transaction, along with the data and the contexts, to the validators, who check the data and the contexts against the commitments and verify the transaction. Once this transaction is confirmed, the validators use the contexts to update their commitments, and archives update their copies of the system state.

Note that if a transaction T is issued at state s_1 and comes with contexts that correspond to s_1, other transactions may be confirmed after s_1 but before the validators get to processing T. At the time the validators process T, their commitments may correspond to some later state s_2. In order to process T, validators must be able to use contexts from s_1 in order to validate T and update their commitments (which correspond to s_2) based on the results of T once T is confirmed. Aardvark addresses this challenge of out-of-order transaction processing, as long as no more than τ transactions have been confirmed between s_1 and s_2 (where τ is a system parameter).

System Components. Our system is built on vector commitments, described in §4. We first implement our basic dictionary operations (Get, Put, and Delete below) using vectors (§5). We next enable Aardvark to execute transactions out of order by developing a versioning scheme for the dictionary (§6). We then introduce a mechanism for Aardvark to continue providing service even if archives fail (§7).

3.1 Transaction Interface

Aardvark presents a *dictionary* interface which associates *keys* (account identifiers) with *values* (account balances and other information). In a transaction, clients interact with the state by issuing a sequence of operations against the dictionary to read and modify the key-value mappings.

Specifically, Aardvark supports the following operations: Get, Put, and Delete. Get(k) returns the value v associated with the key k or a special value which denotes that k is absent. Put(k,v) associates a new value v with k, overwriting any old value. Delete(k) disassociates k from any value.

In Aardvark either all of a transaction's operations execute, or none do. A transaction specifies a set of keys (a bitstring

[2]We use the word "contexts" instead of "proofs" for the authenticated dictionary in order to avoid confusion with the proofs in the underlying vector commitments. Vector commitment proofs are elements of contexts.

which identifies a single value) and a list of operations which involve those keys. All keys referenced by a transaction must be specified up front. In particular, operations such as dynamic key reads are unsupported. (See §6.3 for more details.)

Let `alice` and `bob` denote keys for two balances. (For simplicity, assume Get returns 0 for a missing key.) Then a transfer of p units of money from `alice` to `bob` may be implemented via a transaction, in pseudocode.

```
with Transaction(alice, bob) as txn:
    a = txn.Get(alice)
    b = txn.Get(bob)
    assert a >= p
    if a-p == 0:
        txn.Delete(alice)
    else:
        txn.Put(alice, a-p)
    txn.Put(bob, b+p)
```

If the validators were to store the system state, they would have enough information to process the transaction. Since in Aardvark they do not, the client asks archives to provide contexts for every dictionary operation in the transaction. Given a key and an operation, the archive computes the context σ. For every dictionary operation in the transaction, the client supplies the context obtained from an archive.

3.2 Security

The cryptocurrency depends on a proportion (e.g., a majority) of validators to faithfully conform to the rules of its protocol. Aardvark assumes that these same validators also follow its protocol. In contrast, the archives and clients are allowed to arbitrarily deviate from correct behavior.

Aardvark aims to provide functionality identical to that of a ideal dictionary with respect to a computationally-bound adversary. This functionality manifests itself in Aardvark in two ways: soundness and completeness. Here we state these properties informally and motivate their correctness. Appendix A formalizes and proves them.

Soundness. The effect of every transaction for a validator is the same as it would be if the validator had large persistent storage and stored the entire state on its own, without any archives. This property is precisely specified in §A.1.2. At a high level, this property is guaranteed by induction, as follows. The commitment initially reflects the genesis state by design. If the commitment reflects the current state, then a validator will accept transactions only with correct data and will update its commitment to correctly reflect the new state. We formalize this proof, which relies on the security of underlying vector commitments, in §A.3.

Completeness. A validator will accept a transaction as long as it includes correct contexts from the archives, and not too many other transactions have been confirmed since these contexts were created. This property is precisely specified in §A.1.1 and is evident by inspection of our design.

3.3 Availability

Aardvark operates in an environment where clients and archives may be malicious. Two aspects of Aardvark's design allow it to provide availability in such scenarios.

Efficiency under adversarial transactions. Under adversarial modification patterns, storage cost for validators is upper-bounded by a small fraction of the state size, which allows the system to charge storage fees to prevent denial-of-service attacks. Specifically, the cryptocurrency may require that each account holds a fixed minimum balance so that such an attack would require a substantial, ongoing investment. §8 evaluates this upper bound analytically.

Archive-free operation. Even if no archive functions correctly, clients can monitor the public transaction sequence to maintain a current context for any given key. §7 describes how clients synchronize their contexts without interacting with archives.

4 Background: Vector Commitments

To explain vector commitments, we recall hash functions. A cryptographic hash function $\mathcal{H}$, given an input X, produces $h = \mathcal{H}(X)$. Because it is infeasible to find another $X' \neq X$ such that $h = \mathcal{H}(X')$, we can say that h is a "commitment" to X: the storage of X can be outsourced to an untrusted server as long as we retain h. We can verify the data that the untrusted server sends us by comparing its hash value to h.

Vector commitments can be viewed as generalizations of cryptographic hash functions. A vector commitment scheme can commit to not just one input X, but an entire array V of data: the Commit procedure produces a short commitment c to the vector V. Unlike a mere hash function, a vector commitment scheme allows anyone who knows c to verify not only the entire V, but also individual elements $V[i]$ without seeing other elements if given a proof π_i, where π_i is produced by someone who knows V. Commitments and proofs are efficiently updatable when one value changes.

We first describe the interface and security properties required of a vector commitment scheme, and then we discuss the specific vector commitment scheme we use.

4.1 Interface and Security Properties

We consider schemes which fix the array size at parameter-generation time,[3] with the following black-box interface.

[3] There are vector commitment schemes that do not require this, but they are not efficient enough for our purposes.

$$\mathsf{ParamGen}(B) \to \mathrm{pp}$$
$$\mathsf{Commit}_{\mathrm{pp}}(V) \to c$$
$$\mathsf{Open}_{\mathrm{pp}}(V, i) \to v_i, \pi$$
$$\mathsf{Verify}_{\mathrm{pp}}(c, i, v_i, \pi) \to \mathsf{ok}/\perp$$
$$\mathsf{CommitUpdate}_{\mathrm{pp}}(c, (i, v_i, v_i')) \to c'$$
$$\mathsf{ProofUpdate}_{\mathrm{pp}}(\pi, j, (i, v_i, v_i')) \to \pi'$$

ParamGen generates parameters pp that can be used to commit to an array of size B. These parameters must be generated by a trusted procedure (for example, through multiparty computation) that does not reveal the secrets used. One set of parameters can be used for multiple arrays, each of size B. These parameters pp are used by all other algorithms; we will omit the subscript pp when it does not cause ambiguity.

Commit creates a cryptographic commitment to an B-length vector of variable-length data.[4] Commit is deterministic: if v_1 and v_2 are two B-sized vectors and $v_1 = v_2$, then $\mathsf{Commit}(v_1) = \mathsf{Commit}(v_2)$. We rely on the determinism of Commit to ensure that validator state is consistent.

Open creates a cryptographic proof that a particular vector element v_i (present at index $i < B$ in the vector v) is committed to in $\mathsf{Commit}(V)$. Verify checks that a proof is valid.

CommitUpdate returns $\mathsf{Commit}(V')$, where V' is obtained from V by changing the ith entry from v_i to v_i'. Similarly, ProofUpdate returns the proof $\mathsf{Open}(V', j)$ for element j in this V'. Note that CommitUpdate and ProofUpdate require only one element of V, unlike Commit and Open.

The *binding* property of the commitment[5] ensures that it is computationally infeasible to prove that any value other than v_i is at index i.[6]

4.2 Specific Choice: Vector Commitments of Libert and Yung

The specific choice of the underlying vector commitment is not crucial for Aardvark, as long as a commitment can be efficiently updated when one element in the vector changes. However, as stated before, our particular choice of commitment is motivated by our efficiency requirements. In particular, bandwidth costs are one factor that limits system throughput. We use vector commitments from Libert and Yung [30]. They have the smallest known proofs: only 48 bytes for 128-bit security, or about an order of magnitude shorter than Merkle proofs for reasonable data sizes and security parameters. Moreover, as shown by Gorbunov et al. [25], multiple proofs (even for different commitments) can be aggregated into a single 48-byte value and can be verified at once. They rely on pairing-based cryptography, which has seen adoption and deployment in the last decade [41].

These vector commitments require public parameters pp linear in B. (See [25, §4.1] for a discussion of how such parameters can be generated.) Commitments and proofs both take time linear in B to create and constant time to update (see Table 3), while verification runs in constant time but requires evaluation of a relatively expensive pairing. §8.1 evaluates our choice of vector commitments against those used by EDRAX.

These considerations affect our design. First, the choice of B allows Aardvark to trade off storage savings against computational overhead. Increasing B reduces storage costs by a constant factor and increases proof generation costs by a constant factor.

Second, verification costs remain relatively high and are not mitigated by scaling down B. While the proof generation work is done by untrusted archives and can be divided amongst many untrusted machines (e.g., on some cloud provider with cheap storage), proof verification must be done by the validator, so it lies on the critical path. Thus, to manage computational costs, Aardvark must allow proof verification to execute in parallel on independent threads and minimize the number of required verification operations.

5 Authenticated Dictionary Design

Aardvark maintains an authenticated mapping from keys, which are arbitrary bitstrings, to their values. Since vector commitments provide an array interface, we must find an efficient encoding of the dictionary's mapping in arrays. Our approach for maintaining such a mapping is storing all key-value pairs in an arbitrary order in arrays of size B (recall from §4.1 that B is the vector size). To prove that a key is associated with a given value, the prover runs Open on any index containing the key-value pair, which produces a correct result if the key appears exactly once in the dictionary. Modifying the value for a key involves running CommitUpdate.

Less obvious is how a prover shows that a key does *not* exist. What if the prover lies about a key's absence in a mapping? If we solve this problem, we can also ensure that every key appears at most once, because as part of the insertion proof, the prover would show that the key is absent from the dictionary before insertion.

Aardvark achieves this invariant by using two distinct orderings for the key-value pairs. First, Aardvark commits to the pairs in an arbitrary *sequential* ordering in the *slots* of the length-B vectors. Because this ordering is arbitrary, we can always append to the end of the last vector. By eagerly compacting the vector upon deletion, we minimize both the number of cryptographic operations and the amount of empty space. Second, Aardvark commits to an independent *lexicographic* ordering of its keys, allowing proofs of a key's ab-

[4] Variable-length data can always be converted to fixed-length data by hashing before applying Commit.

[5] Aardvark does not require a hiding property from the commitment scheme since the backing data stored by archives is public.

[6] Technically, we need binding to hold only when c was honestly produced; however, many vector commitment schemes ensure stronger binding, namely even when c is produced by the adversary.

sence from the dictionary. We do so by storing each key with its lexicographic successor. In this way, Aardvark commits to a linked list of keys in ascending lexicographic order.

Specifically, each slot contains a triplet $(k, v, \text{succ}(k))$. Given the set of all keys, we define the successor function $\text{succ}(k)$ for any key k as follows:

1. If k is the largest key in the dictionary, $\text{succ}(k)$ is the smallest key in the dictionary.
2. Otherwise, $\text{succ}(k)$ is the smallest key in the dictionary larger than k.

We can define a key's predecessor in a similar way.

Proving that k maps to v involves opening the vector commitment at the index containing the slot $(k, v, \text{succ}(k))$. Proving that k is absent from the dictionary involves opening the vector commitment at the index containing the slot $(k_0, v_0, \text{succ}(k_0))$, where $k_0 \neq k$ and $\text{succ}(k) = \text{succ}(k_0)$.

Initialization. Our dictionary is initialized with vector commitment parameters parameterized with a bucket size B, as specified in the previous section. In addition, it is initialized with a single key and value, as it must maintain the internal invariant that at least one key is present at any time.[7] The validator must compute the initial commitment value, as it cannot trust the archives to compute it correctly.

5.1 Contiguous Slot Packing

One challenge of implementing our dictionary is that users may wish to insert many arbitrary keys and then later remove these keys. Aardvark must efficiently scale up and down relative to these modifications, which could be adversarial in nature. Our allocation scheme for Aardvark guarantees a bounded worst-case storage cost regardless of access pattern.

Aardvark's slots are grouped consecutively into *buckets* of size B: the ℓth bucket holds hashes of slots numbered $[B\ell, B\ell + B)$. Denote the data in the ℓth bucket as $D[\ell]$. For each ℓ, archives store $D[\ell]$, while the validator stores $\mathsf{Commit}(D[\ell])$. The validator also stores the total size of the dictionary, s. Let D denote the sequence of all $D[\ell]$, and define the *dictionary commitment* to be the sequence of all vector commitments to all buckets and the value s.

To modify a slot, an archive computes the absolute index of the slot in the slot ordering, along with the proof of the slot's current value at that index. This data allows the validator to modify the slot in place.

```
def slot_write(db: handle, i: index,
               old: slot, new: slot):
    bucket, offset = i/B, i%B
    c0 = db.get_commit(bucket)
    c1 = CommitUpdate(c0, (offset, old, new))
    db.set_commit(bucket, c1)
```

[7] This may be achieved with a sentinel key which is never deleted.

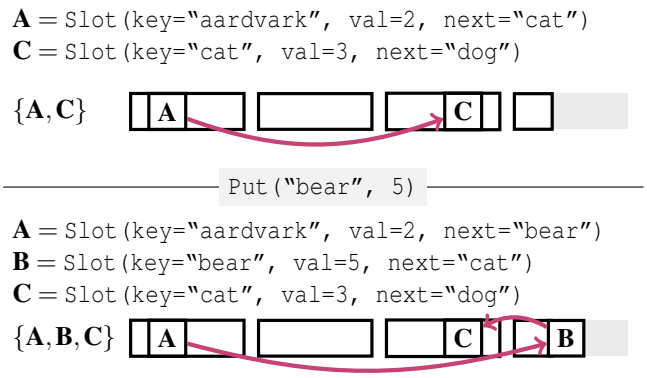

Figure 1: Key insertion in Aardvark consists of the following steps. (1) The key is inserted into the last slot of the dictionary. (2) Its predecessor is verified, and the new slot's next pointer is attached to the predecessor's next pointer. (3) Its predecessor's next pointer is updated to the inserted key. Observe that the operations shown here require only one update operation on a vector commitment (updating the predecessor's next pointer) since the tail is stored by the validator and not committed to.

To enforce tight bounds on the dictionary's space overhead, Aardvark's packing scheme maintains an invariant: the validator stores at most $\lceil s/B \rceil$ vector commitments plus a small, constant amount of metadata. When a new element is inserted, it is added to the last slot in the dictionary (at s).

```
def slot_append(db: handle, new: slot):
    s = db.size() + 1
    db.set_size(s)
    if s%B == 1:
        db.push_commit(Commit([new]))
        return
    bucket, offset = s/B, s%B
    c0 = db.get_commit(bucket)
    # zeros is the bit "0" repeated 256 times
    c1 = CommitUpdate(c0, (offset, zeros, new))
    db.set_commit(bucket, c1)
```

Similarly, when an element is deleted, it is overwritten with the element at the last slot (at s), and the last slot is cleared.

5.2 Authenticating Reads and Writes

To prove that a key k is associated with a particular value, an archive locates that key's slot and computes a proof of its membership with Open. Then, the archive transmits the slot, its index, and its proof to the validator as a *context* object. For a validator, reassigning the key to a new value is straightforward given its context: validate the context, use the index to locate the slot, and overwrite the slot with its new value. Updating the slot is implemented by `slot_write`, and context verification entails executing Verify.

```
def verify_ctx(db: handle, ctx: context):
    i, val, pf = ctx.index, ctx.slot, ctx.pf
    bucket, off = i/B, i%B
    c0 = db.get_commit(bucket)
    return Verify(c0, off, val, pf)
```

Proving that a key does not exist in the dictionary is similar, except that the archive proves the existence of the predecessor's slot. Thus, Get is simply implemented by checking which of these cases we are in.

Inserting a mapping for a new key k involves appending to the last bucket a slot which holds k and its value v. Moreover, we preserve the lexicographic ordering of the keys, similar to a linked-list insertion, which requires the context of k's predecessor. Thus, Put may be implemented as follows.

```
def Put(db: handle, key: key, val: value,
        ctx: context):
    assert verify_ctx(db, ctx)
    assert ctx.slot.key <= key < ctx.slot.next
    if ctx.slot.key == key:
        old, new = ctx.slot, ctx.slot
        new.val = val
        slot_write(db, ctx.index, old, new)
        return
    new_last = Slot(key=key, val=val,
                    next=ctx.slot.next)
    slot_append(db, new_last)
    old_pred, new_pred = ctx.slot, ctx.slot
    new_pred.slot.next = key
    slot_write(db, ctx.index,
               old_pred, new_pred)
```

Finally, Delete is the inverse of Put and requires three contexts: that of k itself, that of its predecessor (for the linked-list deletion), and that of the dictionary's last slot with respect to the *sequential* order (since deleting a key involves overwriting its slot's contents with the contents of the last slot).

5.3 Tail Operations

Insertion and deletion operations both affect the *tail bucket*, the last bucket in the dictionary. By batching cryptographic operations to this bucket, Aardvark reduces the cost of maintaining its commitment and obviates the need for slot insertions and deletions to provide a context for the last slot.

Specifically, no commitment to the tail bucket is maintained. Put operations appending slots do not cause the immediate creation of a new vector commitment. Instead, the new slots are stored directly on the validator until more than B slots have accumulated, at which point the validator performs a single Commit operation to generate the new commitment.

Similarly, for Delete operations, slots in the tail bucket with no commitment are removed without CommitUpdate operations. Once no such slots remain, the validator decommits to the entire bucket at once by obtaining the preimage of the last commitment and executing a single Commit operation. From here, the validator again stores the bucket slots in the preimage without a commitment.

Because the bandwidth cost of a single decommitment is relatively high but amortizes well over time, the validator caches the preimages of some suffix of the buckets in a *tail cache* and synchronizes this cache in the background. The validator forgets old buckets as insertion operations accumulate, and it requests buckets from archives as deletion operations accumulate. If a validator lacks the preimage to a tail bucket, it refuses to serve subsequent deletion requests until it can resynchronize. To reduce thrashing when insertions alternate with deletions, the validator maintains a minimum tail cache size of L but a maximum tail cache size of $2L$.

6 Out-of-order Transaction Processing

The dictionary design in §5 changes state on every commitment of a write-containing transaction, invalidating proofs against an older state. Specifically, executing a transaction that writes to some bucket will invalidate proofs for any concurrently-issued transaction which uses the same bucket.

In a high-throughput system, proofs may be invalidated frequently, requiring either clients or archives to constantly recompute and resubmit proofs for pending transactions. This problem worsens under heavy load, as the buildup of pending transactions further increases the cost of recomputing proofs. Moreover, an attacker may degrade service to a user by constantly modifying their key's bucket-neighbors, invalidating its old proofs. Unfortunately, our evaluation shows that ProofUpdate is expensive for our vector commitments (§8.1).

Aardvark addresses this problem by entirely eliminating the need to update stale proofs. Instead, Aardvark employs a *versioning* system which allows it to accept a transaction with stale proofs even while concurrent operations modify state referenced by that transaction.

First, the validator assigns each transaction it executes a sequence number, appending it to the sequence of completed transactions. Applying each transaction sequentially produces the current dictionary state. Thus, each prefix of this sequence determines a single *snapshot* of the dictionary state at a particular sequence number, and the sequence numbers enumerate the versions of the dictionary.

When issuing a transaction, clients declare the least version t_0 it may execute with. The contexts attached to the transaction refer to the snapshot of the dictionary as of t_0. The transaction may not execute past version $t_0 + \tau$, where τ is the system's *maximum transaction lifetime* and is a global constant set at initialization. A larger value of τ allows old transaction proofs to be accepted by the system for a longer period of time. When the dictionary is under heavy load, a client's transaction may expire before it is executed, forcing its contexts to be updated

before the transaction is resubmitted, so a larger τ reduces this extra work when the system is congested.

Second, the validator holds the dictionary commitment not as of the most recent version, but instead as of the *base snapshot* τ versions ago. For each transaction between the oldest state and the newest state, the validator caches the changes caused by the transaction, so a smaller τ reduces cache overheads. These changes modify the vector commitments over the buckets, so the validator caches the modified vector commitments as well. Thus the validator may validate proofs against any snapshot up to τ versions old by checking them against the cached modified vector commitments.

6.1 Completeness and Context Versioning

A client begins by querying an archive for the set of contexts necessary to execute a transaction, which consists of either Get, Put, or Delete operations. For each operation, the archive returns the slots required by the validator, their indices in the current version of the dictionary, and their openings with respect to their containing buckets, along with the current dictionary version t_0. When a validator receives a transaction along with contexts at some later version $t \geq t_0$, the validator must both determine the values of the slots relevant to the transaction's operations and also determine the changes necessary to update its dictionary commitment.

First, the validator authenticates the contexts, checking that they indeed correspond to slots which are valid at version t_0. This requires the validator to know the vector commitments as of t_0, since the archives opened the commitments with respect to that version. Second, once the validator has slots correct for t_0, it updates these slots so that they are correct for t. This requires the validator to know the updates which happened between version t_0 and version t. Once the validator has all slots at version t, it can validate and execute the transaction, which produces new updates to the current snapshot.

To achieve completeness, the validator must process proofs for all versions t_0 where $t - \tau \leq t_0 \leq t$. Thus, the validator's commitments correspond to the base snapshot at version $t - \tau$, and the validator must store the last τ transactions. Whenever the validator applies a new transaction to the state, it saves the new transaction and marks the oldest transaction eligible for deletion; old transactions are asynchronously applied to the commitments before they are garbage-collected.

6.2 Caching

As mentioned previously, the validator must authenticate proofs produced for any snapshot between version $t - \tau$ and version t, which means it must compute the vector commitment corresponding to that version. Because the validator must eventually compute all vector commitments anyway in order to update the commitments to the base snapshot, Aardvark maintains a cache of *vector commitment deltas*. This cache contains, for each version t' between $t - \tau$ and t, the old and new vector commitments corresponding to each bucket modified by the transaction with sequence number t'. This cache enables the validator to authenticate proofs without running CommitUpdate operations for each transaction.

Just as with vector commitments, the validator may also similarly cache *slot deltas* and *key deltas* to speed up lookups of slots (by index) and lookups of key-value pairs (by key), respectively, by holding both the old and new values of these slots and keys. Figure 2 illustrates how these caches enable a validator to process operations in transactions.

Determinism. Aardvark requires that updates are deterministic and that context verification is consistent across validators so that they compute matching commitments, which is particularly important in a cryptocurrency. Updates are made consistent by enforcing canonical orderings, such as by sorting keys before applying deltas and by processing key insertions and deletions in a well-defined order.

Block Batching. To reduce the in-memory overhead of maintaining deltas and the cost of searching deltas for keys, Aardvark supports batching transactions into *blocks*. With a block-commit optimization, versions (and τ) may refer to block sequence numbers as opposed to transaction sequence numbers. Once a block of transactions is processed, Aardvark merges all deltas in the block. By matching the block's key insertion operations with deletion operations, Aardvark reduces the total number of cryptographic operations required and expedites queries and updates to the base snapshot.

6.3 Transaction Expressiveness

Our choice of versioning scheme allows Aardvark transactions to be fairly expressive. For example, transactions applying commutative changes to values, such as the addition or the maximum function, interleave nicely in Aardvark. At the same time, authenticated dictionaries possess an inherent limitation. All keys must be fixed once a context is created.

To illustrate, suppose that the value of a key k_1 is another key k_2, and some transaction T executes $\mathsf{Get}(\mathsf{Get}(k_1))$. T has no way to compute the proof belonging to the key $k_2 = \mathsf{Get}(k_1)$ because some other transaction may execute $\mathsf{Put}(k_1)$ after T is issued but before T is confirmed.

7 Archive-free Operations

The dictionary design presented above relies on archives for availability: if the archives go down or refuse to serve a client, the client cannot perform all operations. However, with small modifications, most operations may be made *archive-free*: they may be decoupled from archives, which allows them to execute even if all archives are unavailable.[8]

[8] One additional benefit of archive-free operations is that they reduce steady-state computational load on archives. This is significant when scaled

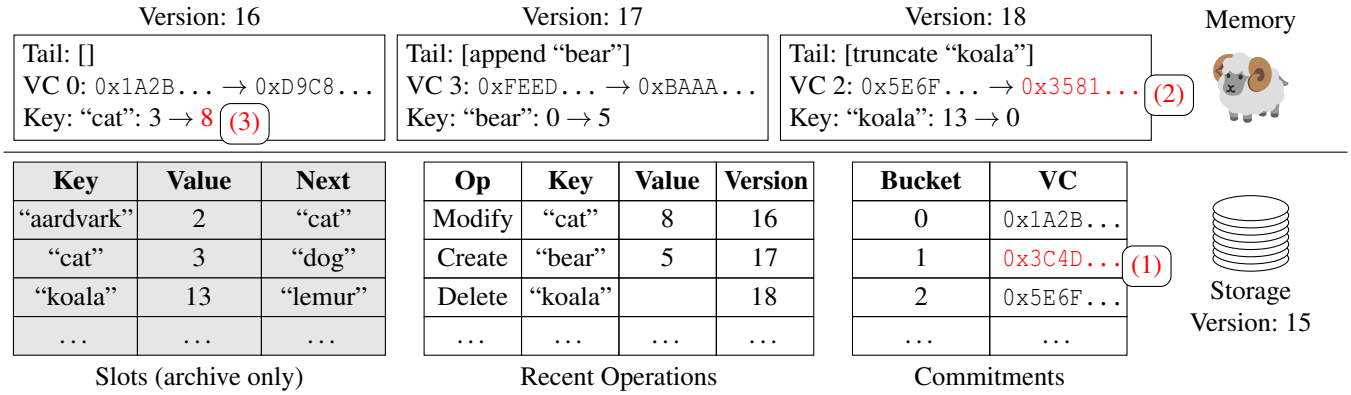

Figure 2: Three Gets in different Aardvark transactions ($\tau = 3$): (1) Get("aardvark", $\sigma = \{$val: 2, index: 1, version: 17$\}$) $\rightarrow$ 2, (2) Get("walrus", $\sigma = \{$val: 5, index: 3, version: 18$\}$) $\rightarrow$ 34, and (3), Get("cat", $\sigma = \{$val: 3, index: 0, version: 12$\}$) $\rightarrow$ 8. For (1), neither the key itself nor its bucket has been modified recently, so the archive's proof is guaranteed to be valid for the vector commitment on the validator's storage. For (2), the value itself has not been modified recently; however, a deletion operation relocated it. Since its proof was created after relocation, the commitment is in a vector commitment delta. For (3), the value was modified recently, so the proof is unnecessary; as a result, the dictionary can simply look up the value present in the key delta.

Specifically, consider a client which possesses a correct context for some key k at version t and then observes dictionary's update to version $t+1$. An update to the client's context must either overwrite (or delete) k, modify a bucket-neighbor of k, or move k into another slot (due to a Delete). In the first case, a proof update is necessary only if k is deleted, where it suffices for the client to store k's predecessor's context. The second case corresponds to a simple ProofUpdate operation. Finally, in the third case, the Delete operation contains the context of the new slot, which the client can store.

Thus, a client that maintains fresh proofs of a key and/or its predecessor can issue arbitrary valid Get, Put, or Delete operations without consulting an archive. Moreover, proof updates only need to be issued once every τ sequence numbers. There is one exception: at a certain point, a validator may run out of its tail cache because too many deletions have happened. In that case, no subsequent deletions may be executed because the tail cannot be moved to fill new slots.

8 Evaluation

Our evaluation answers the following questions:

- *Vector Commitments*: What are the performance characteristics of our choice of vector commitments, and how do they compare to alternatives?
- *Storage Savings*: How much storage does a validator require compared to an archive, which stores all data?
- *Bandwidth Costs*: What additional networking overhead do Aardvark's contexts introduce?
- *Processing Overhead*: What are the processing costs required of both validators and archives?

over the user base, as Open involves a non-trivial computation (§8.1).

Table 1: VC Operation Latency (mean $\pm$ SD μs)

Operation	Aardvark	Merkle Tree	EDRAX w/o SNARK
Commit	$40\,262 \pm 129$	$1\,317 \pm 4$	—
Open	$40\,277 \pm 444$	< 1	—
Verify	$3\,707 \pm 10$	9 ± 0	$3\,131 \pm 9$
CommitUpdate	62 ± 1	9 ± 0	13 ± 1
ProofUpdate	62 ± 1	< 1	27 ± 19

Table 2: VC Object Sizes (bytes)

Object	Aardvark and [47]	Merkle Tree	EDRAX w/o SNARK	EDRAX w/ SNARK
Com.	48	32	64	64
Proof	48	320	640	192

We give closed-form expressions for storage and bandwidth costs. For computation costs, we present an empirical evaluation on a prototype implementation, which is available at https://github.com/derbear/aardvark-prototype.

8.1 Vector Commitments

We evaluate the operations in our implementation of vector commitments, in a basic Merkle Tree, and in the implementation used in EDRAX, both with and without the SNARK [17, §4.1]. We compare to the vector commitments of [47] analyt-

Table 3: VC Curve Operations (scalar multiply, pairing)

Operation	Aardvark and [47]	EDRAX w/o SNARK	EDRAX w/ SNARK
Commit	$(O(B),0)$	$(O(B),0)$	$(O(B),0)$
Open	$(O(B),0)$	$(O(B),0)$	SNARK
Verify	$(O(1),2)$	$(O(1),\log B)$	$(O(1),3)$
CommitUpdate	$(O(1),0)$	$(O(1),0)$	$(O(1),0)$
ProofUpdate	$(O(1),0)$	$(O(\log B),0)$	SNARK

ically, because no implementation is available.

We implement our vector commitment scheme in Rust [3], using a pairing library for algebraic operations [2] (which is a fork of Bowe's library [14]).[9] We compare our implementation against that of EDRAX [52]. Although EDRAX presents an extension with SNARKs (§2), the implementation is not available. Moreover, the implementation supports neither Commit nor Open, which EDRAX does not require since clients incrementally update proofs and commitments.

We perform single-core microbenchmarks on a `m5.metal` Amazon EC2 instance having a Intel Xeon Platinum 2.5GHz CPU with sizes of 32KiB, 32KiB, and 1MiB for the L1, L2, and L3 caches, respectively. To compute operation throughput, we execute 100 iterations to warm up the machine state and then perform 1000 measurements. We precompute powers of base points in our vector commitments, the interior nodes of the Merkle Tree, and the update public keys in EDRAX. We set $B = 1024$, since EDRAX vector commitments support vector sizes which are a power of two.

Tables 1 and 2 show the results of our microbenchmarks. We do not provide operation latencies of EDRAX with a SNARK in Table 1, because we did not implement the very complex SNARK computation. We elaborate on those costs here. CommitUpdate remains the same. According to [17, §4.2], the cost of Verify is approximately 7ms. Understanding the cost of ProofUpdate with a SNARK is more subtle, because EDRAX requires updating *every* vector commitment proof (one per client) whenever any data in the vector changes but requires recomputing SNARKs only for the *one* proof (of the sender account value) that must be sent with a transaction. The cost of computing the SNARK is 77s [36], incurred by the client only when submitting a transaction; at all other times, ProofUpdate with a SNARK is the same as without.

The EDRAX implementation uses a different underlying elliptic curve than Aardvark. Since curve operations are roughly 5× faster on the EDRAX curve, the CommitUpdate and ProofUpdate operations are also faster. However, Verify times (without a SNARK) are roughly equivalent.

[9] More recent implementations of pairings [42, 43] are likely to provide a noticeable speed-up to our benchmarks.

In particular, while Aardvark uses the BLS12-381 [6] elliptic curve, EDRAX uses one of two BN curves [7] (depending on the presence of SNARKs). However, the specific curve used in the EDRAX implementation provides approximately 100 bits of security [5], which is less than the 128-bit security provided by our curve.

To compare the constructions independently of the curve choice, we count the number of both group and pairing operations in Aardvark and EDRAX in Table 3. (We also note that the vector commitments of [47] perform similarly to Aardvark.) Our analysis shows that Verify, Open, and ProofUpdate are slower in EDRAX's commitments than in Aardvark's.

8.2 Storage and Bandwidth Analysis

Storage Savings. Aardvark requires archives to store all records in the database. If s is the number of key-value pairs, and $|k|$ is the size of a key, and $|v|$ is the size of a value, the archive storage cost for the database is

$$S_A = s(|k| + |v|).$$

In the absence of Aardvark or some other authenticated storage mechanism, this is the storage cost which would be incurred by every validator.

In Aardvark, validators must store a commitment to a bucket every B records. Validators must also store the records in the tail bucket, and all records for all buckets in its cache. This cache is at most $2L$ buckets large (and at least L buckets small, unless the database is less than L buckets large). Moreover, validators must store the last τ transaction blocks to process contexts up to τ blocks stale. If the size of an encoded transaction block is $|T|$ and the size of a vector commitment is $|c|$, this means that the validator storage cost for the database is upper bounded as follows:

$$S_V < \lceil s/B \rceil |c| + (2L+1)(|k| + |v|)B + \tau|T|.$$

A natural choice for $2L$ is a small multiple of $\frac{|T|}{B|t|}$, where $|t|$ is the size of a transaction, which allows validators to execute several blocks of continuous deletion requests regardless of the availability of archives. If $\frac{|T|}{B|t|} = 10$, then a value of $2L = 30$ is sufficient. Even if blocks are confirmed very quickly (e.g., once a second), setting $\tau = 1000$ allows clients a considerable amount of concurrency (e.g., contexts remain valid for fifteen minutes). As a result, because the last two terms in the equation above are fairly small—around 1GB for $|T| = 1$MB—we can drop them as s grows large. Therefore, the ratio between a validator's cost and an archive's cost in the limit is

$$\frac{S_V}{S_A} \to \frac{B(|k| + |v|)}{|c|}.$$

As described in §4, $|c| = 48$ bytes in our particular commitment scheme. Choosing a value such as $B = 1000$ for keys of

size $|k| = 32$ and values of size $|v| = 8$ bytes gives Aardvark validators a savings factor of more than 800×; we note this savings factor increases with the size of $|v|$.

Bandwidth Costs. The bandwidth costs of a transaction consist of the cost of transmitting the transaction plus the costs of transmitting the values for each key k in the transaction's key set and the corresponding contexts σ_k.

Regardless of whether a key k is present, σ_k includes k itself, the value v it is mapped to, the key's successor $\text{succ}(k)$, the key's index in the sequential ordering i, the context's version number τ_0 and the proof that results from opening the vector commitment π. In addition, for contexts inserting a new k, the transaction must also contain the key k itself, and contexts that delete a key k include k twice (once in the slot itself and once in the predecessor slot), so they may de-duplicate k. Finally, deleting keys requires decommitting to entire buckets; with amortization, a single net deletion requires decommitment to one slot (but no proof). If the rest of the transaction data (which contain its operations) are $|t|$ bytes long, then the number of bytes transmitted for a transaction inserting n_1 keys, modifying n_2 keys, and deleting n_3 keys is

$$\begin{aligned}&|t|+(3n_1+2n_2+3n_3)|k|\\&\quad+(n_1+n_2+2n_3)(|v|+|\pi|+|i|+|\tau_0|)\\&\quad+\max\{n_3-n_1,0\}(|k|+|v|).\end{aligned}$$

Recall that contexts in Aardvark are composed of a version number and an index in the dictionary's slot ordering, in addition to a 48-byte vector commitment proof (see §4), which means that a size of $|\pi|+|i|+|\tau_0| = 64$ bytes is sufficient for a context corresponding to a particular key. Since the cost of transmitting the transaction in the first place was

$$|t|+(n_1+n_2)(|k|+|v|)+n_3|k|.$$

we obtain, with $|k| = 32$ and $|v| = 8$, roughly 100B of overhead per Put operation and 200B per Delete operation.

As compared to transaction size, context overhead is greatest for small transactions that read and write many keys and least for large transactions that read and write few keys. Note that transactions include the key set read and written by the transaction, which also contributes to transaction size.

Note that by aggregating proofs, we can transmit only a single proof, which halves the marginal overhead to roughly 50 additional bytes per transaction per key, plus the fixed cost of the aggregated proof. Applications where transactions access many different keys benefit substantially from aggregation.

8.3 System Throughput

We perform an experimental evaluation of the computational overhead of Aardvark. We replace the storage backend of the Algorand cryptocurrency [22] with our open-sourced implementation of Aardvark [29], and we benchmark single-operation transactions involving a Put or Delete operation.

Table 4: Validator Processing Time for 100 000 Transactions

Scenario	Cores	Mean ± SD (s)
Put (modify)	1	342 ± 14
Put (insert)	1	349 ± 13
Delete	1	684 ± 38
Put (modify)	32	34 ± 1
Put (insert)	32	45 ± 2
Delete	32	67 ± 3

(Get involves processing similar to that of Put). We measure completion times on an archive and a validator separately to eliminate the effects of network latency.

We separate Put operations into two cases: (1) Put operations that insert a new key and (2) Put operations that modify an existing key in the dictionary; this allows us to measure the additional costs of updating key successors. For Delete operations, we pass in bucket preimages as they are needed for vector decommitment. We do not aggregate proofs for deletion operations but instead transmit both proofs separately.

We generate transactions as follows. Keys are 32 bytes long and values are 8 bytes long. For Put transactions that insert keys, the key is a random 32-byte string. For other transactions, the key is picked at random from the current set of keys. The maximum transaction lifetime is $\tau = 10$ blocks. The transaction's least valid block sequence number is chosen uniformly at random between b and $b - \tau$. Put transactions are 105 bytes in size, while Delete transactions are 87 bytes.

The dictionary is initialized with 1 million random key-value pairs, corresponding to a total state size of 47 MB.[10] Before running each workload, we execute $\tau + 2 = 12$ blocks, each with 10 000 transactions of the same type, in order to reach steady-state behavior.

We conduct our experiment on a `c5.metal` Amazon EC2 instance having a Intel Xeon Platinum 3.0GHz CPU, using `numactl` to restrict the physical CPUs available to the system. We call into our Rust implementation of the vector commitment scheme from our dictionary implementation, written in Go. We store commitments, recent transactions, and (for archives) slots in an in-memory SQLite database to reduce the effect of I/O latency costs in our measurements.

Validator Throughput. We pre-generate 100 000 transactions, partition them evenly into 10 blocks, and then issue each block to a validator. Validators perform cryptographic proof verification in parallel and verify proofs on all transactions—even those which only affect recently-modified keys—to simulate worst-case performance. We measure the time taken to validate and apply transactions in each block. We run ten

[10] We expect measurements of computational overhead to generalize to larger dictionaries as cryptographic costs constitute the main CPU bottleneck.

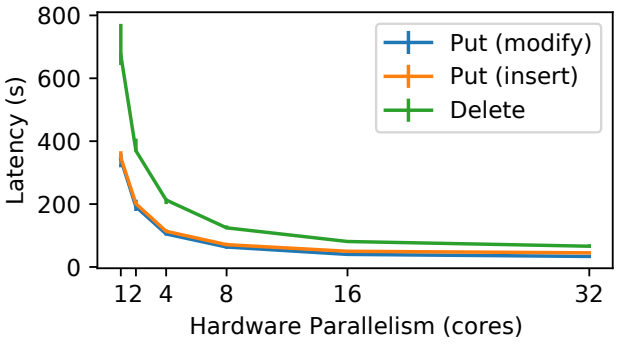

Figure 3: Scaling of Aardvark relative to number of cores. Points represent the median times taken to apply 100 000 transactions batched into 10 blocks, while the error bars represent the minima and maxima. Adding cores improves performance of proof verification, which is extremely parallel, but updating commitments must still be done in order.

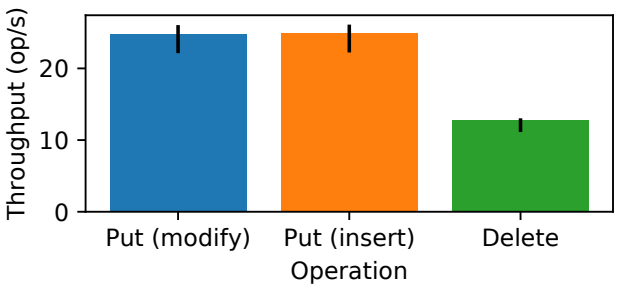

Figure 4: Transaction contexts created per second on a single archive with a single core. Put operations take roughly the same amount of time, regardless of whether they insert a new key or modify an existing one. Delete operations require executing both Read and DeleteContext, so producing a proof takes twice as long. This plot shows the median throughput, with error bars denoting the maxima and minima.

trials each for 1, 2, 4, 8, 16, and 32 cores. We summarize the results in Table 4 and plot them in Figure 3.

Our experiments show that transaction validation for blocks filled with Delete transactions take around twice as long as blocks filled with Put transactions for existing keys, since proof verification is expensive, and Delete transactions require two proof verifications while Put transactions require one (in addition to verification of tail slots).

Parallelization improves system throughput, but only up to a point: increasing from 1 to 32 cores raises throughput by only 8–10× for several reasons. First, in the worst case, each transaction may affect the next, forcing serialization of transaction processing. Second, processing all deltas in each block incurs a fixed cost as the deltas are reconciled with the stable storage. Third, although proof verification is completely parallelized, commitment updates are serialized in our experiments (we did not parallelize commitment updates). Although some parallelism is possible in the common case, an adversary could cause many modifications to affect different indexes of a single bucket, forcing serialization of these updates in the worst case.[11] As a result, while proof verification costs dominate when few cores are available, the overhead of updating slots and commitments becomes a bottleneck as parallelization increases.

Archive Throughput. Since archive operations are trivially parallelizable, we evaluate archive workloads on a single core. In each trial, we query a single archive for contexts for 10 000 transactions. We run 10 trials for each workload, measuring the throughput of context creation in each trial, and plot the results in Figure 4.

Recall that a Put operation which modifies a key requires looking up its context, a Put operation which inserts a new key requires looking up its predecessor's context, and a Delete operation requires one of each lookup. Our evaluation shows that a single-core archive serves around 12.8 Delete and around 22.1 Put requests per second, regardless of whether a key is inserted or not. This shows that proofs of membership and nonmembership in Aardvark take roughly the same amount of time as they are dominated by the cost of creating the cryptographic proof, as opposed to other costs.

[11] Batch commitment updates may improve performance in this worst case. This optimization is absent from the prototype and is left to future work.

9 Conclusion

This paper presents Aardvark, an authenticated dictionary suitable for high-throughput, distributed applications. We show that it is possible to create authenticated dictionaries with short proofs of membership and nonmembership from pairing-based vector commitments while enforcing tight bounds on extra resource use. We develop a versioning scheme that enables us to completely ignore expensive proof update costs while supporting concurrent transaction execution. Our evaluation shows that remaining costs reside in proof verification and in updating vector commitments and are partially addressed by parallelism.

Acknowledgments

The authors would like to thank Hoeteck Wee and Adam Suhl for their assistance with the analysis and implementation of vector commitments and Alin Tomescu for discussion on the paper's motivation and the security model. Moreover, the authors would like to thank David Lazar, Tej Chajed, Kyle Hogan, and Sacha Servan-Schreiber for feedback on a draft of this document. Yossi Gilad was supported by the Hebrew University cybersecurity research center, the Alon fellowship, and

Mobileye. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1745302.

References

[1] Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020*, pages 839–869, Cham, 2020. Springer International Publishing.

[2] Algorand. Pairing plus library, 2020. https://github.com/algorand/pairing-plus.

[3] Algorand. Source code for pointproofs, 2020. https://github.com/algorand/pointproofs.

[4] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. Concerto: A high concurrency key-value store with integrity. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *SIGMOD 2017*, pages 251–266. ACM, 2017.

[5] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *Journal of Cryptology*, 32:1298–1336, 2019.

[6] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, editors, *Security in Communication Networks*, pages 257–267, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[7] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 319–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

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

[9] Alex Biryukov, Daniel Feher, and Giuseppe Vitto. Privacy aspects and subliminal channels in zcash. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *CCS 2019, London, UK, November 11-15, 2019*, pages 1795–1811. ACM, 2019.

[10] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *FOCS 1991, San Juan, Puerto Rico, 1-4 October 1991*, pages 90–99. IEEE Computer Society, 1991. Later appears as [11], which is available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991.

[11] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991.

[12] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, 2019.

[13] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. *IACR Cryptol. ePrint Arch.*, 2020:352, 2020.

[14] Sean Bowe. pairing, 2019. https://github.com/zkcrypto/pairing.

[15] Vitalik Buterin. The stateless client concept, 2017. https://ethresear.ch/t/the-stateless-client-concept/172.

[16] Vitalik Buterin. An incomplete guide to rollups, 2021. https://vitalik.ca/general/2021/01/05/rollup.html.

[17] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968, 2018.

[18] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On scaling decentralized blockchains. In *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[19] Scott A. Crosby and Dan S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In Michael Backes and Peng Ning, editors, *ESORICS 2009*, volume 5789 of *LNCS*, pages 671–688. Springer, 2009.

[20] Justin Drake. Accumulators, scalability of UTXO blockchains, and data availability. https://ethresear.ch/t/accumulators-scalability-of-utxo-blockchains-and-data-availability/176, 2017.

[21] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the bitcoin UTXO set. *IACR Cryptol. ePrint Arch.*, 2019:611, 2019.

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP 2017, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.

[23] Alex Gluchowski. Introducing zkSync: the missing link to mass adoption of Ethereum. https://medium.com/matter-labs/introducing-zk-sync-the-missing-link-to-mass-adoption-of-ethereum-14c9cea83f58.

[24] Michael T. Goodrich, Michael Shin, Roberto Tamassia, and William H. Winsborough. Authenticated dictionaries for fresh attribute credentials. In Paddy Nixon and Sotirios Terzis, editors, *Trust Management, First International Conference, iTrust 2003, Heraklion, Crete, Greece, May 28-30, 2002, Proceedings*, volume 2692 of *LNCS*, pages 332–347. Springer, 2003. Available at http://cs.brown.edu/cgc/stms/papers/itrust2003.pdf.

[25] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. *IACR Cryptol. ePrint Arch.*, 2020:419, 2020.

[26] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, 2016.

[27] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, 2010.

[28] Jonathan Lee, Kirill Nikitin, and Srinath T. V. Setty. Replicated state machines without replicated execution. In *SP 2020*, pages 119–134. IEEE, 2020.

[29] Derek Leung, Leonid Reyzin, and Nickolai Zeldovich. Aardvark prototype artifact, 2020. https://github.com/derbear/aardvark-prototype.

[30] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 499–517. Springer, 2010.

[31] Loopring: zkRollup exchange and payment protocol. https://loopring.org/.

[32] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1989. Available at http://www.merkle.com/papers/Certified1979.pdf.

[33] Andrew Miller. Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage), 2012. https://bitcointalk.org/index.php?topic=101734.msg1117428.

[34] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In Suresh Jagannathan and Peter Sewell, editors, *POPL '14*, pages 411–424. ACM, 2014. Project page and full version at http://amiller.github.io/lambda-auth/paper.html.

[35] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020, August 12-14, 2020*, pages 2075–2092. USENIX Association, 2020.

[36] Babis Papamanthou. Private Communication.

[37] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceedings*, volume 4861 of *LNCS*, pages 1–15. Springer, 2007. Available at http://www.ece.umd.edu/~cpap/published/cpap-rt-07.pdf.

[38] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 91–110. Springer, 2011.

[39] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.

[40] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In Aggelos Kiayias, editor, *FC 2017*, volume 10322 of *LNCS*, pages 376–392. Springer, 2017.

[41] Y. Sakemi, T. Kobayashi, T. Saito, and R. Wahby. Pairing-friendly curves. IETF Draft, https://tools.ietf.org/id/draft-irtf-cfrg-pairing-friendly-curves-07.xml#applications-of-pairing-based-cryptography.

[42] SCIPR-Lab. Zexe, 2020. https://github.com/scipr-lab/zexe.

[43] Supranational. blst, 2020. https://github.com/supranational/blst.

[44] Roberto Tamassia and Nikos Triandopoulos. Certification and authentication of data structures. In Alberto H. F. Laender and Laks V. S. Lakshmanan, editors, *AMW 2010*, volume 619 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[45] Peter Todd. Making UTXO set growth irrelevant with low-latency delayed TXO commitments, 2016. https://petertodd.org/2016/delayed-txo-commitments.

[46] Peter Todd, Gregory Maxwell, and Oleg Andreev. Reducing utxo: users send parent transactions with their merkle branches. https://bitcointalk.org/index.php?topic=314467, 2013.

[47] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *LNCS*, pages 45–64. Springer, 2020.

[48] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *CCS 2019*, pages 1299–1316. ACM, 2019.

[49] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis)aggregation. *IACR Cryptol. ePrint Arch.*, 2020:1239, 2020.

[50] Bill White. A theory for lightweight cryptocurrency ledgers. Available at https://github.com/bitemyapp/ledgertheory/blob/master/lightcrypto.pdf (see also code at https://github.com/bitemyapp/ledgertheory), 2015.

[51] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[52] Yupeng Zhang. vector commitment scheme with efficient updates, 2019. https://github.com/starzyp/vcs.

A Security Analysis

Agrawal and Raghuraman [1] define security of key-value commitments, which are very similar in functionality to our authenticated dictionaries. To define security, we first generalize their notion of key binding [1, §3.2] and then apply the generalization to Aardvark.

Aardvark operations, which are Get, Put, and Delete, differ from those by Agrawal and Raghuraman, which are Insert and Update. Instead of a piecemeal approach of simply replacing these operations in the authenticated dictionary definition, we follow the approach of [38, 39, 44] and define a generic authenticated data structure for *any* set of operations, which encompasses different possible dictionary interfaces as well as other data structures with completely different interfaces (e.g., range queries, searching by rank). After defining the generic data structure, we specialize it to Aardvark, using a map instead of Agrawal and Raghuraman's set of tuples, and defining both Put and Delete in terms of a map update.

A.1 General Authenticated Data Structure Definitions

Let $\mathcal{F} : \mathcal{S} \times O \times X \to \mathcal{S} \times Y$ denote an *ideal* data structure functionality. ("Ideal" refers to the data structure when it behaves as expected, without any adversarial, or even accidental, deviation.) Here $\mathcal{S}$ is the set of *ideal states* of some data structure, with some *initial* state $s_{\mathsf{init}} \in \mathcal{S}$, which supports a set of *operations* identified by a set of *operation codes* (or *opcodes*) O. Given a state and input from the set X, an operation produces a new state and an output in the set Y.

We now turn to the authenticated ("real", as opposed to "ideal") data structure that implements $\mathcal{F}$. Let $\Delta : \mathbb{N} \to \mathbb{N}$ be a *delay function parameter* (which will determine the exact definition of completeness). Recall that an authenticated data structure involves a prover and verifier. The prover (e.g., the Aardvark archive) holds a large *real state* drawn from some set W, and a verifier (e.g., the Aardvark validator) holds a small *commitment* drawn from some set C_Δ. The verifier's objective is to maintain its commitment $c \in C_\Delta$ given reads from and writes to the data structure, and the prover enables the verifier to do this by producing a *context* σ[12] drawn from some set of contexts Σ.

We wish to model an application (e.g., a cryptocurrency) in which the order that the prover produces operation contexts may differ from the order that verifiers execute operations. In particular, because some parts of the operation will be fixed (e.g., a dictionary key) and some parts will be variable (e.g., the value the key is mapped to at some point in time), we decompose the input X into a *static* component X_{static} and a *dynamic* component X_{dynamic} (i.e., $X = X_{\mathsf{static}} \times X_{\mathsf{dynamic}}$).

For the prover, define two functions. $P_\Delta : W_\Delta \times O \times X_{\mathsf{static}} \to \Sigma$ is a *proof generation function* which consumes a real state, opcode, and static input and produces a *context* for the operation. $G : W \times O \times X \to W$ is the corresponding *real update*

[12] Contexts provide all the information necessary to compute the result of the operation and update the commitment. They in particular include information that is traditionally thought of as a proof.

function which mutates the real state.

Let Init be some probabilistic algorithm, which runs in time polynomial with respect to a *security parameter* $\lambda \in \mathbb{N}$, such that $\mathsf{Init}(\lambda) = (c_{\mathsf{init}}, w_{\mathsf{init}}, \mathsf{pp})$, where $c_{\mathsf{init}} \in C_\Delta$ is an *initial commitment*, $w_{\mathsf{init}} \in W$ is an *initial state*, and pp are public *parameters* for the authenticated data structure.

Define $V_{\mathsf{pp}} : C_\Delta \times O \times X_{\mathsf{static}} \times \Sigma \to \{0,1\}$ to be a *verification* algorithm which maps a (commitment, opcode, input, context) tuple to a bit indicating whether the context is valid, given some fixed initial parameters.

Define the *commitment update function* $F_{V,\mathsf{pp}} : C_\Delta \times O \times X \times \Sigma \to C_\Delta \times Y$ to be an algorithm which maps (commitment, opcode, input, context) tuples to (commitment, output) pairs.

A.1.1 Completeness

1. Let $(c_{\mathsf{init}}, w_{\mathsf{init}}, \mathsf{pp}) \leftarrow \mathsf{Init}(\lambda)$
2. Set $s := s_{\mathsf{init}}$, $c := c_{\mathsf{init}}$, $w := w_{\mathsf{init}}$, $i := 0$.
3. Repeat indefinitely:
   (a) Set $w_i := w$.
   (b) Pick any arbitrary $o \in O$, $x = (x_s, x_d) \in X$, and $j := \{0, 1, \ldots, \Delta(i)\}$ nondeterministically. Set $\sigma := P(w_{i-j}, o, x_s)$.
   (c) If $V_{\mathsf{pp}}(c, o, x_s, \sigma) = 0$, output FAIL and halt.
   (d) Set $(s, y) := \mathcal{F}(s, o, x)$, $(c, \bar{y}) = F(c, o, x, \sigma)$, $w := G(w, o, x)$, and $i := i + 1$

We say that F satisfies Δ-*synchronous completeness* for some Δ if for all nondeterministic choices in the algorithm above, the algorithm never outputs FAIL. It is natural to say that *synchronous completeness* corresponds to $\Delta(n) = 0$ and *asynchronous completeness* corresponds to $\Delta(n) = n$.

A.1.2 Soundness

Define $\mathcal{A}$ to be an *adversary*, which is some probabilistic process that runs in time polynomial in λ. Define $\mathcal{G}_{\mathcal{F},\lambda,\mathcal{A}}$ to be the value returned by the following algorithm.

1. Let $(c_{\mathsf{init}}, w_{\mathsf{init}}, \mathsf{pp}) \leftarrow \mathsf{Init}(\lambda)$
2. Set $c := c_{\mathsf{init}}$, $s := s_{\mathsf{init}}$.
3. For some number of iterations polynomial in λ, do the following in a loop:
   (a) Query $\mathcal{A}$ to get $(o, x, \sigma) \leftarrow \mathcal{A}(\mathsf{pp}, \lambda)$ where $o \in O, x = (x_s, x_d) \in X, \sigma \in \Sigma$.
   (b) Set $(s', y) := \mathcal{F}(s, o, x)$ and set $(c', \bar{y}) = F(c, o, x, \sigma)$.
   (c) If $V_{\mathsf{pp}}(c, o, x_s, \sigma) = 1$ and $y \neq \bar{y}$, output FAIL and halt.
   (d) If $V_{\mathsf{pp}}(c, o, x_s, \sigma) = 1$, set $c := c'$ and $s := s'$.
4. Output OK and halt.

We say that F is a *secure* implementation of $\mathcal{F}$ if for all such $\mathcal{A}$, $\Pr[\mathcal{G}_{\mathcal{F},\lambda,\mathcal{A}} = \text{FAIL}]$ is negligible in λ.

Note that our notion of soundness assumes that c_{init} is generated correctly. In an alternative scenario where $\mathcal{A}$ generates c_{init}, this notion would demand that $\mathcal{A}$ cannot cause the verifier to accept two operations with contradictory outcomes. Our application does not consider adversarially-generated commitments as the validator knows the correct c_{init} and updates this commitment itself after every operation.

A.2 Ideal Dictionary Definition

Now we specialize our general definition to a dictionary.

Let $\mathcal{K}$ be a set of keys and $\mathcal{V}$ be a set of values. Let $\perp$ be a sentinel value denoting absence with $\perp \notin \mathcal{K}$ and $\perp \notin \mathcal{V}$. Denote $\mathcal{V}_\perp = \mathcal{V} \cup \{\perp\}$.

Define $\mathcal{M} : \mathcal{K} \to \mathcal{V}_\perp$ to be the *dictionary map* from $\mathcal{K}$ to $\mathcal{V}_\perp$. Let $\mathcal{S} = \mathcal{M}$. Define the *empty dictionary* to be $m_{\text{empty}} \in \mathcal{M}$ where $m(k) = \perp$ for all $k \in \mathcal{K}$.

Call read a *read opcode* and write a *write opcode*. Let $O = \{\mathsf{read}, \mathsf{write}\}$. An invocation of $\mathsf{Get}(k)$ corresponds to the functionality $\mathcal{F}(m, \mathsf{read}, (k, \perp))$. Observe that since $\perp$ denotes absence, $\mathsf{Put}(k, v)$ for any m and $v \neq \perp$ corresponds to the functionality $\mathcal{F}(m, \mathsf{write}, (k, v))$; likewise, $\mathsf{Delete}(k)$ corresponds to $\mathcal{F}(m, \mathsf{write}, (k, \perp))$.

Define $\mathcal{F}$ as follows on the ideal state $m \in \mathcal{M}$, opcode $o \in O$, and the input $(k, v) \in \mathcal{K} \times \mathcal{V}_\perp$.

- $\mathcal{F}(m, \mathsf{read}, (k, \perp)) = (m, m(k))$.
- $\mathcal{F}(m, \mathsf{write}, (k, v)) = (m', \perp)$, where $m'(k) = v$ and $m'(k') = m(k')$ for all $k' \neq k$.

In this scenario, $X_{\mathsf{static}} = \mathcal{K}$ and $X_{\mathsf{dynamic}} = Y = \mathcal{V}_\perp$.

To simplify our analysis, we analyze the security of Aardvark with respect to the operations and not transactions. Given a secure dictionary of operations with a verification algorithm V and update function G, we can define a corresponding secure transactional dictionary with a verification algorithm V' and update function G' where G' is the composition of G over each operation in the transaction and V' returns 1 if and only if V returned 1 for each intermediate state in the composition.

A.3 Security of Aardvark

Theorem 1. *Let F be implemented by Aardvark, C_Δ be the set of validator states, and Σ be the corresponding operation contexts, with* Init *as the function which creates initial vector commitment parameters (§4) and $c_{\mathsf{init}} = \mathsf{Commit}(m_{\text{empty}})$. Then the Aardvark implementation of an ideal dictionary satisfies soundness and τ-synchronous completeness.*

Completeness follows from the argument presented in §6.

To show that soundness holds, consider the Aardvark representation of the ideal data structure m, which we call a *snapshot*. While m simply maps keys to values, the snapshot represents this map as a vector $\bar{e}$ of slots, with each slot containing a key, a value, and the successor key. We argue that

two invariants hold after every operation: the current snapshot faithfully represents the current m, and validators hold commitments to the past τ snapshots.

These invariants are sufficient to show that soundness holds. Indeed, in order to win, $\mathcal{A}$ must output some $(o,(k,v),\sigma)$ such that both $V_{\mathsf{pp}}(c,o,k,\sigma)=1$ and $F(c,o,(k,v),\sigma)$ outputs some $\bar{y}$ inconsistent with $\mathcal{F}$, implying $o=\mathsf{read}$ and $\bar{y}\neq m(k)$.

The context σ must be for a snapshot $\bar{e}'$ that is no more than τ operations old. By the binding property of the vector commitment, σ will contain the correct slot information for k for $\bar{e}'$, or else Verify (which gets correct inputs because the validator has the correct commitment to $\bar{e}'$ by the second invariant) will reject with all but negligible probability.[13] Because the validator stores the past τ operations, the validator can then correctly compute slot information for k according to the current snapshot which, by the first invariant, gives $m(k)$. This contradicts $\bar{y}\neq m(k)$.

To argue that the invariants hold, we need to reason about how operations change m, the snapshot, and the validator state. In §A.3.2 we prove that after each operation the validator continues holding commitments to the past τ snapshots. Intuitively, the security of vector commitments ensures that the context σ corresponds to correct slot information, and therefore the vector commitments will be updated by the validator to faithfully commit to the new snapshot.

In §A.3.3, we argue, based on data structure design, that the snapshots continue to faithfully represent the ideal map m after every operation. In particular, we show the existence of a function that maps each snapshot to a single m, and which respects updates performed by write.

A.3.1 Modeling Validator Behavior

We begin by more precisely describing the (abstract) behavior of Aardvark.

Let $e_i=(k_i,v_i,k_i')$ for some $k_i,k_i'\in\mathcal{K},v_i\in\mathcal{V}$ be a slot, and $\bar{e}=e_0,\ldots,e_{S-1}$ be the snapshot. We assume that the initial snapshot consists of a single slot $(\perp,\perp,\perp)$, where $\perp$ is a sentinel key which is the greater than every key.

We define a *commitment* to a snapshot to be the list $\bar{c}=c_0,\ldots,c_{\lceil S/B\rceil-1}$ that commits to $\bar{e}$, where $c_j=\mathsf{Commit}(e_{jB},\ldots,e_{\min((j+1)B-1,S)})$ (the last vector is padded to length B with zero-slots).

We say that a Aardvark *operation* is the tuple $\gamma=(o,(k,v),\sigma)$. For any operation γ, we define the function $A(\bar{e},\gamma)=\bar{e}'$, which *applies* the operation γ to the snapshot $\bar{e}$ and produces a new snapshot $\bar{e}'$, in the same way as defined by Aardvark (§5).

[13] Formally, if $\mathcal{A}$ can cause the validator to accept inaccurate information, $\mathcal{A}$ can be used to violate the binding property of the vector commitment in polynomial time by running the transactions from the beginning.

A.3.2 Validators are Bound to Snapshots

Lemma 1. *After t operations, the validators hold commitments to the past $\min(t,\tau)$ snapshots, as well as $\min(t,\tau)$ most recent operations, with all but negligible probability.*

Proof. We prove the lemma by induction on t.

The base case holds by design, so consider the inductive hypothesis. Any operation γ for a key k must come with a context relative to a snapshot $\bar{e}_{t'}$ with $t-\tau\leq t'\leq t$, or else the validator will reject. By the binding property of the vector commitment, the context will contain accurate information on the slot that holds k (and its predecessor in case of deletes) for the snapshot $\bar{e}_{t'}$, or else the vector commitment verification algorithm (which gets correct inputs by the inductive hypothesis, because the validator has correct $\bar{c}_{t'}$) will reject with all but negligible probability. If the validator rejects the operation then neither the snapshot nor the validator state changes, and the inductive case holds.

If the validator accepts the operation, it will use CommitUpdate to compute the new commitment for the snapshot $\bar{e}_{t+1}$. Because the validator has the information relevant to the operation from $\bar{e}_{t'}$ and all the operations that happened since t', the validator can correctly compute the difference between the latest snapshot $\bar{e}_t$ and the new snapshot $\bar{e}_{t+1}$ produced by the operation. By the correctness of CommitUpdate, the validator will compute the correct commitment to this new snapshot, and the inductive case holds. □

A.3.3 Snapshots Encode the Ideal Map

The second invariant we must show is that snapshots encode the ideal map in a manner consistent with read and write operations. Since read changes neither the snapshot nor the map, we focus only on write. We define a snapshot $\bar{e}$ to be *well-formed* if the following conditions hold.

- For any k, a triple $(k,v_1,\mathrm{succ}(k))$ appears in $\bar{e}$ at most once.
- If $k\neq\perp$ and $(k,v,k')\in\bar{e}$, $v\neq\perp$. In other words, all keys (not the sentinel key $\perp$) which are in a slot are mapped to a value which is not $\perp$.
- For all $(k,v,k')\in\bar{e}$, it is the case that $k'=\mathrm{succ}(k)$ (relative to the list of keys in $\bar{e}$). In other words, the successor-references in $\bar{e}$ form a valid circularly-linked list of keys.

From this definition, it follows that there exists a natural *decoding* function D from well-formed snapshots to ideal maps m. Namely, since for all slots $(k,v,k')\in\bar{e}$, (k,v) appears uniquely, it follows that D defines the map m where $m(k)=v$ if (k,v) appears in $\bar{e}$ and $m(k)=\perp$ otherwise.

We will now show that the invariant holds by induction. The base case follows by our initialization. The inductive step is given by the following lemma.

Lemma 2. *Suppose $\bar{e}$ is well-formed, $D(\bar{e})=m$, and $\gamma=(\mathsf{write},(k,v),\sigma)$. Let $\bar{e}'=A(\bar{e},\gamma)$. Let m' be the map given*

by $m'(k) = v$ and $m'(k') = m(k')$ for all $k' \neq k$. Then $\vec{e}'$ is well-formed, and $D(\vec{e}') = m'$.

Proof. The lemma follows by inspection of A (for the case of $v = \perp$ and the case of $v \neq \perp$) as defined in Section A.3.1. □