Black-box Approaches to Authenticated Dictionaries: New Constructions and Lower Bounds

Francesca Falzon[1], Harjasleen Malvai[2], and Emanuel Opel[1]

[1] ETH Zürich
ffalzon@ethz.ch, emanuel@eopel.ch
[2] UIUC/IC3
hmalvai2@illinois.edu

Abstract. Authenticated dictionaries (ADs) enable secure lookups to a dictionary hosted by an untrusted server and are a key component of various real-world applications, including transparency systems and cryptocurrencies. Despite significant overlap in techniques for building ADs and related primitives, such as memory checkers and accumulators (i.e., authenticated sets), these relationships have yet to be formalized.

In this work, we give a rigorous treatment of ADs and prove their precise connection to the latter two cornerstone primitives. We start by laying out the minimal algorithms and security properties needed in practice and introduce a new security notion for ADs called write-committing, which requires update proofs to guarantee an exact count of changes.

We prove that any AD built from a black-box authenticated set (AS) makes at least $\Omega(\log n)$ AS calls per lookup and obeys a trade-off between lookups and updates. With optimal lookups, such a scheme requires at least $\Omega(\log n/\log\log n)$ AS calls per update. We also resolve the open question of constructing a secure AD from only black-box access to an AS and present two schemes adhering to the trade-off: one with optimal lookup overhead and the other with higher lookup complexity, but which only requires two AS calls for an update.

Finally, we make strides towards unifying memory checkers and ADs. To this end, we present two constructions for memory checkers with black-box access to an AD: one that incurs constant overhead (but needs write-committing) and a second that only requires the AD to be lookup-secure but incurs logarithmic overhead. We then give a simple AD construction using a memory checker as a black-box, with $\mathcal{O}(1)$ overhead.

Our results demonstrate the inherent limitations of ADs built from accumulators but lay the foundation for extending existing results on memory checkers and other primitives, such as vector commitments, to ADs.

Keywords: Authenticated Dictionaries · Memory Checking · Accumulators.

1 Introduction

Authenticated dictionaries (ADs) enable one to verify the correctness of lookup queries to a dictionary hosted by an untrusted server. Ensuring the integrity of lookup queries is an important requisite for many real-world use cases including transparency logs (e.g., [25,52,53,58]), cloud storage (e.g., [28,42]), and cryptocurrencies (e.g., [55,92]). Most recently, WhatsApp has deployed an AD to enable clients to verify proofs of the key directory's correctness [53], and steps are also being taken toward standardizing key transperancy [60].

To appreciate both the impact of ADs today and their potential applications, consider these flagship deployments—some already built on AD primitives and others poised to benefit from upgrading to full ADs:

– **Key Transparency (KT) Logs.** WhatsApp's AD lets users fetch and verify their own public key bindings without leaking metadata [53].

– **Account-Model Blockchains.** Ethereum [92] and Algorand [3] store state in Merkle–Patricia tries (a kind of AD), letting light clients fetch balances and nonces via $\mathcal{O}(\log n)$ proofs.
– **Package Repositories.** Sigstore's Rekor and other similar transparency logs issue inclusion and consistency proofs per artifact, but cannot efficiently prove "all versions" or metadata for a given package name (e.g., the history of updates made to the package). Without AD support, auditors face $\mathcal{O}(n)$ scans or must trust centralized mirrors, leaving gaps in verifiable software distribution. Existing academic work [37,62,69] already lays the groundwork for improving the security guarantees and performance of these deployments, using AD constructions as building blocks.
– **Database Outsourcing in the Cloud.** Amazon QLDB [4] and Azure Confidential Ledger [64] provide append-only tamper-evident journals with Merkle proofs for cloud databases, but lack key-value lookups with completeness guarantees. Clients must still re-scan the Merkle tree or fully trust the service to ensure that no entries are missing, which undermines either scalability or integrity. Existing work (e.g. [38,44,83]) lays the groundwork for solutions that can overcome the limitations of these deployments.

Is a simpler data structure enough? An authenticated set (AS)–which we use interchangeably with the term accumulator–is arguably the simplest authenticated data structure and enables one to commit to a set and prove whether an item is or is not a member of the set. Given the aforementioned applications, it is natural to ask whether an authenticated set would suffice. The following examples illustrate the drawbacks of using accumulators in certain applications.

Certificate Transparency (CT) logs offer append-only, publicly auditable membership proofs for individual certificates, but they do not provide efficient key-value mappings from a domain name to the current set of valid certificates. This essentially makes a CT log an accumulator. As a result, domain owners or auditors must download or reconstruct the entire certificate log–an $\mathcal{O}(n)$ operation–to verify all certificates for a given domain. In contrast, KT systems balance verifiability with privacy and frequent key updates, allowing users to query their own key state in $\mathcal{O}(\log n)$ time without downloading the full log.

UTXO-based blockchains, such as Bitcoin, basically log a set of coins, effectively making them an accumulator–similar to the CT example above. In this setting, light clients employ Simplified Payment Verification (SPV) to verify that a transaction is included in a block by downloading only block headers and Merkle proofs. SPV, however, cannot prove the completeness of an address's unspent transaction outputs (UTXOs), i.e., that no UTXOs have been omitted, so wallets must scan the entire chain or trust a full node to enumerate all UTXOs for an address [93]. Early proposals for embedding an authenticated UTXO commitment in the blockchain aim to address this gap but are still in nascent stages and would require consensus-breaking changes [15]. Meanwhile, account-model blockchains emerged to address exactly these shortcomings of Bitcoin by providing efficient AD functionality.

Existing black-box compilers? The gap in performance between CT and KT or UTXO and account model blockchains illustrates the importance of using ADs over authenticated sets in certain settings. Given the long line of work on authenticated sets (see e.g., [9,10,11,13,14,17,19,20,21,33,34,43] [49,56,68,72,77,81,95,96]), it is natural to ask whether an authenticated set can be compiled into an AD in a black-box manner. Prior work has toyed with turning a generic accumulator into an AD; see, for example, the work of Tomescu et al.[85], which observes the limitations of the black-box approach and resorts to a non-black-box solution. Recent work on oblivious accumulators [12] used an accumulator as a black-box to build an AD, but only while also using a more powerful primitive–vector commitments–as an additional building block. Since all known approaches either break modularity or blow up performance, finding efficient black-box approaches (or proving the lack thereof) remains an open problem.

In other works such as SEEMless [25] and OPTIKS [54], "zero-knowledge sets" and "append-only sets" are used to build ADs in a black-box manner. This appears promising, but upon closer inspection, their use of the term accumulators is a departure from standard terminology. In fact, the primitives they refer to as a "set" are not really an accumulator but rather an authenticated dictionary.

Relation with other primitives. Other cryptographic primitives including memory checkers, vector commitments, proofs of data posession (PDP) and proofs of retrievability (POR) have used techniques similar to those used in the AD literature. Memory checkers enable a user to outsource a RAM program to a third-party server and later verify integrity of the memory with small local storage. POR and PDP are more restricted forms of memory checking, with similar security requirements. Vector commitments allow a prover to commit to a vector of elements and prove values for individual positions; the functionality of VCs is equivalent to treating each position in the vector as a memory location and then checking correctness of read (or write, in case of updates) operations.

As we discuss in Sec. 5, the security requirements of memory checking are strictly stronger than those for ADs–and this also turns out to be the case for VCs. Thus, the relationship between memory checking and ADs is thought-provoking but has seen little exploration in the literature to date. Given the line of lower-bound results about memory checking, the relationship between memory checking and ADs is especially interesting to study.

The limitations of prior work thus demand the following: (1) standardization of terminology for authenticated sets and ADs, (2) investigation of whether the overhead of compiling primitives such as accumulators into and ADs is inherent, and (3) exploration of the relationship between memory checking and ADs.

1.1 Our Contributions

This work formalizes the relationship of authenticated dictionaries to other important authenticated data structures. We outline our contributions below.

(C1) Formalizing ADs and their security. We start by formalizing ADs using a minimal set of definitions to capture the functionality and security notions required by real-world applications. We also define a new notion of security for ADs, which we term *write-committing*; this property proves critical for building one of our two memeory checking constructions (Sec. 2 and Sec. 5.1)

(C2) Two AD schemes. We resolve an open question posed in [85], about whether an AD can be built from black-box use of an authenticated set. The authors of [85] note the challenge of ensuring that all values associated with a queried label are returned. We answer the question affirmatively and describe two constructions for ADs built from an AS that are able to do just this and offer different tradeoffs; our first scheme is more efficient for small value sets, and our second scheme is more amenable to efficient updates. (Sec. 3)

(C3) AD lower bounds. We demonstrate the optimality of our constructions by proving lower bounds on the number of calls to the underlying AS scheme for both lookups and updates. We show that any such AD must make, on average, $\Omega(\log n)$ invocations to the underlying AS for each lookup, where n is the size of the universe of values. This bound is tight and met by our first scheme. We further show that when this lookup bound is met, at least $\log n / \log \log n$ AS operations are needed to update a key-value pair in the AD. In contrast, our second scheme trades off increased lookup overhead for constant update overhead and requires at most two AS operations for each update. Our lower bound proofs recast the problem to a counting argument in multi-dimensional space, and these techniques may be of independent interest. (Sec. 4)

(C4) Two memory checking schemes. We also demonstrate the relationship between ADs and memory checkers. The typical security notion for memory checkers, read-over-write (RoW)

security, requires that any value read from an array location be the last value written to that location. Given only black-box access to an updateable AD, we present two constructions for a memory checker, each assuming different security notions for the underlying AD. (Sec. 5)

(C5) An AD built from a memory checker. Finally, we provide a simple, constant-overhead construction for an AD using a memory checker together with Cuckoo hashing. (Sec. 6)

1.2 Implications of our Results

Our results on building ADs from authenticated sets and memory checkers extend to other lines of work on cryptographic commitments of data structures.

Accumulators. The terms accumulator and zero knowledge set [26,57,63] have been used in the literature to refer to what we call an authenticated set. While adhering to the lower bounds in Sec. 4, existing and future accumulator constructions can be used to obtain additional properties for ADs, such as short proofs [14,57], obliviousness [12] or batched proofs [23]. We note however, the following: depending on the underlying assumptions, for each accumulator construction's asymptotic performance on updates and membership/non-membership proofs, there exists a corresponding AD construction which matches its performance on updates and the analogous lookup proofs. Our results on the logarithmic overhead of black-box AD compilers using an accumulator show that building an AD from an accumulator is strictly worse than non-black-box constructions. Thus, our lower bounds urge researchers to redirect the work for applications such as blockchains and key-transparency away from accumulators.

Vector commitments (VCs). A vector commitment [24] allows a prover to commit to a vector and prove to a verifier the values committed at different positions in the vector. This is, in fact, equivalent to memory checking! As we discuss in Sec. 7, it is possible to build an AD from memory checking with $\mathcal{O}(1)$ overhead, and therefore the same applies to vector commitments. This further streamlines the AD contruction of Baldimtsi, Karantaidou, and Raghuraman [12], which uses a vector commitment but with an additional accumulator. Thus, our results give directions for future work to compile ADs from vector commitments and to allow the ADs to inherit any additional properties of the underlying VC such as efficient aggregation of proofs [80,46,84], efficient incrementability [22] and easily updated pre-generated proofs [91].

1.3 Related Works

Authenticated dictionaries. Most prior work on authenticated dictionaries proposes constructions for specific use cases such as cryptocurrency [66,55,92], key distribution [25,58,61], outsourced storage [31,65,30,55,43], and transparency [52,1,85,86,48,87]. Many of these schemes require $\mathcal{O}(\log n)$ computation at both the server and client, where n is the number of key-value pairs, e.g., [5,31,65,52]. Other constructions only require $\mathcal{O}(1)$ runtime for client lookups [31,86,55], at the expense of an $\mathcal{O}(n)$ factor on the server side.

Memory checking. Memory checkers were introduced by Blum et al. [16] and can be viewed as a special case of ADs in which the keys are memory locations and the data stored at a given location is the corresponding value. They present schemes for various data structures and prove a $\log N$ lower bound for the size of trusted storage required for a data structure of size N. Ajtai [2] showed that a memory checker with small trusted memory must be invasive, i.e., the number of physical locations at untrusted storage must have a blowup over the logical memory locations. Correspondingly, the queries to logical storage must suffer a blowup in physical storage accesses (or queries).

Naor and Rothblum [67] provided various results about the relationship between memory checking and other primitives, and showed that for an information-theoretically secure online memory checker, $q \times s \in \Omega(n)$, where n is the number of locations in the physical memory. Dwork et al.

[35] refined this result for deterministic, non-adaptive memory checkers. Recent work by Boyle et al. [18] generalized the bounds of [35] to any setting of online memory checking (adaptive, information-theoretic, computationally secure, etc.). Wang et al. [90] entended the lower bounds to lower bounds about the locality of memory checking.

Papamanthou and Tamassia [75] and more recent work [59], consider an expanded model for parallel reads/writes. Other constructions, e.g., [28,42,47,88,70], achieve various efficiency metrics such as smaller server storage or amortized read/write time.

PDP and POR. Ateniese et al. [7] introduced provable data possession (PDP) as a restrictive form of memory checking that allows a client to outsource data to an untrusted server and later verify possession of the original data without retrieval. Follow-up schemes were proposed (e.g., [8,6,89]) including ones supporting updates [27,36], distributed storage [39], and multiple replicas [32]

Proofs of Retrievability (POR) was concurrently introduced by Juels and Kaliski [50] and enables an untrusted server storing a file on behalf of a client to prove to the client that it holds the data and that the client can successfully retrieve the file in its entirety. POR offers stronger security guarantees than PDP. While PDP and POR are related to ADs, they offer less functionality.

Other data structures. ADs can be implemented using authenticated hash tables [76,78]. Other authenticated data structures have also been proposed, including for graph connectivity and geometric queries [45], shortest path queries on graphs [94], pattern matching queries [74] and authenticated databases [79]. Such data structures are outside the scope of this work.

2 Background and Definitions

Notation. We denote by $[\![x, n]\!] : \mathcal{U} \times \mathbb{N} \to \{0,1\}^n$ the representation of some element x in $\mathcal{U}$ as an n-bit string. Note that all definitions in this section are algorithmic, which is the convention in existing literature.

2.1 Authenticated Dictionary

We consider authenticated dictionaries (ADs), which we define below.

Definition 1. *Let $\mathbb{L}$ be the set of possible labels, $L \subseteq \mathbb{L}$, and $\mathbb{V}$ be the set of possible values. A **dictionary** is a mapping $D : L \to \mathcal{P}(\mathbb{V})$ that maps each label $\ell \in L$ to a set of values $V \in \mathcal{P}(\mathbb{V})$ (here, $\mathcal{P}$ denotes the power set).*

The sets $\mathbb{L}$ and $\mathbb{V}$ are application specific and, for the sake of generality, we abstract away specific descriptions of these sets. We use the term *labels* instead of keys to distinguish them from *cryptographic* keys.

Definition 2. *A **static AD**, denoted* StAD*, is a data structure that maps labels $\ell \in \mathbb{L}$ to sets of values $V \subseteq \mathbb{V}$ and comprises of the following algorithms:*

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{sp})$*: Takes a security parameter λ and setup parameters* sp*, and outputs public parameters* pp*.*
- $(d, \mathsf{st}) \leftarrow \mathsf{Commit}(\mathsf{pp}, D)$*: Takes public parameters* pp *and a dictionary D, and outputs a digest d and state* st*.*
- $(V, \pi^{lkup}_{\ell,V}) \leftarrow \mathsf{Lookup}(\mathsf{pp}, \mathsf{st}, D, \ell)$ *: Takes the public parameters* pp*, state* st*, dictionary D, and a label ℓ. It outputs a set of values V and a proof $\pi^{lkup}_{\ell,V}$.*
- $b \leftarrow \mathsf{VerLookup}(\mathsf{pp}, d, \ell, V, \pi^{lkup}_{\ell,V})$*: Takes public parameters* pp*, a digest d of D, a label ℓ, a set of values V, and a proof $\pi^{lkup}_{\ell,V}$. It returns a bit $b \in \{0,1\}$.*

This basic definition captures important standard functionalities across various authenticated dictionaries by allowing flexible interpretations of the different parameters. For example, the public parameters pp might contain a verification key, which can then be used at verification time to verify that a lookup query is answered correctly by a trusted data source. Alternatively, the public parameters might be empty, the digest be the root hash of a Merkle tree, and the lookup proof be an authentication path.

Definition 3. *An **updatable AD**, denoted* UpdAD*, is a* StAD *with the following two additional algorithms, as well as an associated predicate* Pred*:*

- $(\mathsf{st}_{i+1}, d_{i+1}, \pi^{updt}_{i,i+1})/\perp \leftarrow \mathsf{Update}(\mathsf{pp},\ d_i,\ \mathsf{st}_i,\ U)$*: Takes as input public parameters* pp*, a state* st_i *and a corresponding digest* d_i*, a set of updates* U *consisting of tuples of the form* (op, ℓ, V)*, where* $\mathsf{op} \in \{\mathsf{add}, \mathsf{update}, \mathsf{delete}\}$*. It outputs the updated state* st_{i+1}*, updated digest* d_{i+1}*, and proof* $\pi^{updt}_{i,i+1}$*, or an error* $\perp$*.*
- $b \leftarrow \mathsf{VerifyUpdate}(\mathsf{pp}, d_i, d_{i+1},\ \pi^{updt}_{i,i+1})$*: Takes as input public parameters* pp*, digests* d_i *and* d_{i+1} *and a proof* $\pi^{updt}_{i,i+1}$*. It returns a bit* $b \in \{0,1\}$

We require correctness, as well as a number of security properties for an AD. First, we define these properties for a static authenticated dictionary, i.e., one without updates.

Definition 4. *Let* $D : L \subseteq \mathbb{L} \to \mathcal{P}(\mathbb{V})$ *be a dictionary and* $\lambda \in \mathbb{N}$*.* StAD *is said to have **lookup correctness** if for all labels* $\ell \in L$*:*

$$\Pr\left[\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{sp}) \\ (d, \mathsf{st}) \leftarrow \mathsf{Commit}(\mathsf{pp}, D) \\ (V, \pi^{lkup}_{\ell,V}) \leftarrow \mathsf{Lookup}(\mathsf{pp}, \mathsf{st}, D, \ell) : \\ V = D[\ell] \\ \wedge\ \mathsf{VerLookup}(\mathsf{pp}, d, \ell, V, \pi^{lkup}_V) = 1 \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

Definition 5. *Let* $\lambda \in \mathbb{N}$*.* StAD *has **lookup security** if for all probabilistic* $\mathsf{poly}(\lambda)$*-time adversaries* $\mathcal{A}$ *and any* $\ell \in \mathbb{L}$

$$\Pr\left[\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (d, \ell, V \neq W, \pi^{lkup}_V, \pi^{lkup}_W) \leftarrow \mathcal{A}(\mathsf{pp}) : \\ \mathsf{VerLookup}(\mathsf{pp}, d, \ell, V, \pi^{lkup}_V) = 1 \\ \wedge\ \mathsf{VerLookup}(\mathsf{pp}, d, \ell, W, \pi^{lkup}_W) = 1 \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Definition 6. *Let* $\lambda \in \mathbb{N}$*.* UpdAD *is said to be **value-binding** if, for all probabilistic* $\mathsf{poly}(\lambda)$*-time adversaries* $\mathcal{A}$ *and any* $\ell \in \mathbb{L}$

$$\Pr\left[\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (t, n, j, \ell, V \neq W, \pi^{lkup}_V, \pi^{lkup}_W, \\ d_t, (d_i, \pi^{updt}_{i-1,i})_{i=t+1}^{t+n}) \leftarrow \mathcal{A}(\mathsf{pp}) : \\ \forall\ i \in [t+1, t+n], \\ \mathsf{VerifyUpdate}(\mathsf{pp}, d_{i-1}, d_i, \pi^{updt}_{i,i+1}) = 1 \\ \wedge\ t < j \leq t+n \\ \wedge\ \mathsf{VerLookup}(\mathsf{pp}, d_j, \ell, V, \pi^{lkup}_V) = 1 \\ \wedge\ \mathsf{VerLookup}(\mathsf{pp}, d_j, \ell, W, \pi^{lkup}_W) = 1 \end{array}\right] \leq \mathsf{negl}(\lambda).$$

2.2 Authenticated Set

An authenticated set (AS) is a data structure that commits to a set $S \subseteq \mathbb{V}$ and supports membership queries.

Definition 7. *A **static authenticated set (AS)**,* StAS*, is a data structure that stores a set of values from* $\mathbb{V}$ *and comprises the following algorithms:*

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{sp})$*: Takes a security parameter* λ *and setup parameters* sp*, and outputs public parameters* pp*.*
- $(d, \mathsf{st}) \leftarrow \mathsf{Commit}(\mathsf{pp}, S)$*: Takes public parameters* pp *and a set* $S \subseteq \mathbb{V}$ *and outputs a digest* d *and corresponding state* st*.*
- $(b, \pi_v^{mbr}) \leftarrow \mathsf{Member}(\mathsf{pp}, \mathsf{st}, S, v)$ *: Takes public parameters* pp*, server state* st*, the set* S*, and an element* v*. It outputs a bit* $b \in \{0,1\}$*, and a proof* π_v^{mbr}*.*
- $b' \leftarrow \mathsf{VerMember}(\mathsf{pp}, d, v, b, \pi_v^{mbr})$*: Takes public parameters* pp*, a digest* d *of* S*, a value* v*, a bit* b*, and a proof* π_v^{mbr}*. It returns a bit* $b' \in \{0,1\}$*.*

Definition 8. *A **updatable authenticated set (uAS)**,* UpdAS*, is a static authenticated set with the following additional algorithms:*

- $(d', \mathsf{st}', S', \pi^{\mathsf{update}}) \leftarrow \mathsf{Update}(\mathsf{pp}, \mathsf{st}, S, v, \mathsf{type})$ *: Takes the public parameters, the server state* st*, the set* S*, an element* v *and a type for the update* $\mathsf{type} \in \{\mathsf{Insert}, \mathsf{Delete}\}$*. It outputs an updated digest, state, and set* d', st', S' *respectively and a proof* π^{update}*.*
- $b \leftarrow \mathsf{VerifyUpdate}(\mathsf{pp}, d, d', \pi^{\mathsf{update}}, \mathsf{type})$*: Takes public parameters* pp*, digests* d, d'*, a proof* π^{update}*, type of update* $\mathsf{type} \in \{\mathsf{Insert}, \mathsf{Delete}\}$ *and returns bit* b*.*

The setup parameters sp can specify additional information, e.g., an upper bound on the set or the number of bits used to encode each set element. We require that if $v \in S$ (resp. $v \notin S$), then Member outputs $b = 1$ (resp. $b = 0$) and a proof of membership (resp. non-membership) π_v^{mbr}.

Definition 9. *Let* $S \subseteq \mathbb{V}$ *be a set and* $\lambda \in \mathbb{N}$*.* StAS *has **membership correctness** if for all elements* $v \in \mathbb{V}$ *where* $b = 1$ *if* $v \in S$ *and* $b = 0$ *otherwise, then*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (d, \mathsf{st}) \leftarrow \mathsf{Commit}(\mathsf{pp}, S) \\ (b'', \pi_v) \leftarrow \mathsf{Member}(\mathsf{pp}, \mathsf{st}, S, v) : \\ b = b'' \wedge \ \mathsf{VerMember}(\mathsf{pp}, d, v, b'' \pi_v^{mbr}) = 1 \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

Definition 10. *Let* $\lambda \in \mathbb{N}$*.* StAS *is said to have **membership security** if for all probabilistic* $\mathsf{poly}(\lambda)$*-time adversaries* $\mathcal{A}$ *and any* $v \in \mathbb{V}$

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (d, v, \pi_v^{mbr}, \hat{\pi}_v^{mbr}) \leftarrow \mathcal{A}(\mathsf{pp}) : \\ \mathsf{VerMember}(\mathsf{pp}, d, v, 0, \pi_v^{mbr}) = 1 \\ \wedge \ \mathsf{VerMember}(\mathsf{pp}, d, v, 1, \hat{\pi}_v^{mbr}) = 1 \end{array}\right] \leq \mathsf{negl}(\lambda).$$

2.3 Memory Checker

Definition 11. *A memory checker* $\mathsf{MemCheck}$ *consists of a client* C*, a memory* M*, and the following algorithms:*

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{sp})$*: takes as input a security parameter* 1^λ *and a set of parameters* sp*. It outputs parameters* pp*.*

– $(T, U) \leftarrow \mathsf{Init}(\mathsf{pp}, \mathsf{st})$*: takes as input a set of parameters* pp*, a security parameter represented in binary, and the initial state of a database. It outputs the initial trusted memory state* T *and the untrusted memory state* U*.*
– $(v, \pi) \leftarrow \mathsf{Read}(i, U)$*: queries the untrusted memory for an index* i *and outputs value* v *and proof* π*.*
– $0/1 \leftarrow \mathsf{VerRead}(i, v, T, \pi)$*: takes as input a trusted memory state* T*, a location* i*, value* v*, and proof* π*, and outputs a bit* 0 *or* 1*.*
– $(\delta, \pi, U') \leftarrow \mathsf{Write}(i, v, U)$*: takes as input position* i*, a value* v *and untrusted memory* U*. It outputs updated untrusted memory* U'*, a proof* π*, and a set of trusted memory updates* δ*.*
– $(b, T') \leftarrow \mathsf{VerWrite}(i, v, T, \delta, \pi)$*: takes as input a location-value pair* (i, v)*, trusted memory state* T*, trusted memory updates* δ*, and a proof* π*. It outputs an updated trusted memory state* T' *and a bit* b*.*

We now define soundness for memory-checkers, called read-over-write security.

Definition 12. *Let* $\lambda \in \mathbb{N}$*.* $\mathsf{MemCheck}$ *is said to have* ***read-over-write*** *(RoW) security if for all probabilistic* $\mathsf{poly}(\lambda)$*-time adversaries* $\mathcal{A}$*:*

$$\Pr\left[\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{sp}), \\ (\ell, v, v^*, \pi, \pi^*, \\ (T_j, loc_{j-1}, val_{j-1}, \delta_{j-1}, \pi_{j-1})_{j\in[1,n]}, \\ k \in [0, n]) \leftarrow \mathcal{A}(\mathsf{pp}) : \\ v \neq v^* \\ \wedge\ \mathsf{VerRead}(\ell, v, T_0, \pi) = 1 \\ \wedge\ \mathsf{VerRead}(\ell, v^*, T_k, \pi^*) = 1 \\ \wedge\ \forall j \in [0, (n-1)], \\ \mathsf{VerWrite}(loc_j, val_j, T_j, \delta_j, \pi_j) = 1, T_{j+1} \\ \wedge\ \forall\, j \in [0, (n-1)],\ loc_j \neq \ell \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Def. 12 captures the notion that if v is the value of a location ℓ, then after a sequence of operations – excluding any write to ℓ – the subsequent read from ℓ must verify only if the value for ℓ is v. See [71] for more details about read-over-write (RoW) security.

2.4 Authenticated Dictionary Threat Model

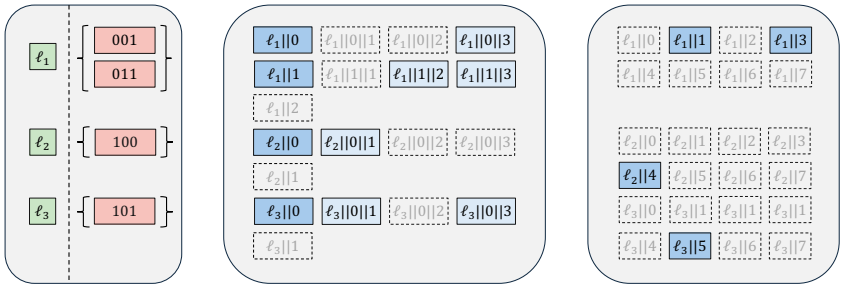
In this work, we consider two types of parties: (1) an untrusted client (or multiple untrusted clients) and (2) an untrusted server. No trusted data source exists, both parties can arbitrarily deviate from the protocol, and, in particular, an adversary may corrupt any subset of the parties. The server holds the entire dictionary and the client holds a short digest.

Security is only defined with respect to the honest (non-corrupt) clients; we require that if there are honest clients, then either verified answers to lookup requests are consistent across the honest clients or that misbehavior is eventually detected. Concretely, we require that honest clients see the same dictionary and require consistency between digests held by the honest clients, i.e., non-equivocation. This involves a dissemination mechanism such as a gossip protocol or a public bulletin board (e.g., blockchain).

While many types of threat models exist in the AD literature, we focus on this model–which we refer to as the *transperancy model*–because it requires the fewest assumptions and is, therefore, the most general.

3 From Authenticated Sets to Authenticated Dictionaries

One natural approach to building ADs is to implement them using an AS. For example, [82] proposed an AS based on a Merkle tree and used it as a black box to construct an AD by storing



(a) Dictionary. (b) Set encoding 1. (c) Set encoding 2.

Fig. 1: An example of (a) a dictionary and the set encodings for (b) Compiler 1 and (c) Compiler 2. We allow the dictionary to map labels to multiple values e.g., label ℓ_1. In (b) and (c), the elements in the set are contained in blue boxes and the elements that we test for non-membership are in dashed boxes.

items of the form $H(H(\ell)\|H(v))$ in the set, where (ℓ, v) is a dictionary entry. Since they assume a trusted data source, they do not have to contend with multiple values for a label.

Tomescu et al. [85], on the other hand, assume that no parties are trusted; their scheme must be able to detect if a malicious server commits divergent value sets in the AD. They modify their AS scheme in a non-black-box manner and note that, while one could attempt to build an append-only AD from an append-only AS in a "black-box" manner by representing each key-value pair as a set element, this approach is not sufficient for proving to verifiers that all values associated with the queried label have been returned.

We ask and answer the following question affirmatively: *Is it possible to construct a static AD from a static AS in a black-box manner such that the AD can prove to the verifiers that all values associated with a label have been returned?* We are the first to present two general approaches for compiling an AS into an AD that achieves this. The query complexity of these constructions requires a multiplicative factor that is linear in the number of values returned and the bit length of the values.

3.1 Construction 1

Both of our constructions proceed by encoding the given dictionary as a set which can then be committed to using a static AS. At a high level, our first encoding is as follows. For each pair (ℓ, V) in the dictionary and each value $v_j \in V$, we add the element $\ell\|j$ to the set, where $j \in [|V|]$ denotes the index of the value in V. Then, for each bit of v_j's bit representation equal to 1, we add the string $\ell\|j\|k$ to the set, where k denotes the index of the bit in v_j. In the detailed construction, care is taken to encode all the indexes as bit strings of a length specified by the setup parameters.

We give an example dictionary and its set encoding in Fig. 1a and 1b, respectively. Proving that $\{001, 011\}$ is the set of values associated with ℓ_1 boils down to proving: (1) membership of $\ell_1\|0$ and $\ell_1\|1$, (2) membership of the corresponding bits equal to 1, (3) non-membership of its bits equal to 0, and (4) non-membership of $\ell_1\|2$. The final proof ensures that the set of returned values is exhaustive.

Compiler description. Let D be the dictionary we wish to commit to and let $S \leftarrow \varnothing$. The compiler starts by calling $\mathsf{StAS.Setup}$ to generate the public parameters $\mathsf{pp}_{\mathsf{AS}}$ for the underlying AS. For each pair $(\ell, V) \in D$, we associate a unique counter $\mathsf{ctr}_v \in [|V|]$ with each $v \in V$. For each value $v \in V$, we then add the following entry to S: $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!]\|[\![\mathsf{ctr}_v, \mathsf{len}_{\mathrm{ctr}}]\!]$. This element denotes a value with the index ctr_v. Let $\mathsf{len}_{\mathrm{ind}}, \mathsf{len}_{\mathrm{lab}}$, and $\mathsf{len}_{\mathrm{ctr}}$ denote the maximum bit length of a value, label, and counter, respectively. We encode the value as a set bitstring(s) as follows. For each index

$i \in \mathsf{len}_{\mathrm{ind}}$ such that $[\![v, \mathsf{len}_{\mathrm{val}}]\!][i] = 1$ add the string

$$[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}_v, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!].$$

to set S. Finally, we commit to the set S as $d, \mathsf{st}_{\mathsf{AS}} \leftarrow \mathsf{StAS.Commit}(\mathsf{pp}_{\mathsf{AS}}, S)$ and return $(d, \mathsf{len}_{\mathrm{ctr}}), (\mathsf{st}_{\mathsf{AS}}, S)$.

To generate a lookup proof for pair (ℓ, V), we initialize an empty dictionary Π, which will store the proofs for each value. For each $v \in V$, we generate membership proofs for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}_v, \mathsf{len}_{\mathrm{ctr}}]\!]$ and a nonmembership proof for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![|V| + 1, \mathsf{len}_{\mathrm{ctr}}]\!]$ which proves there are no more values present. In addition, for each $v \in V$ and each $i \in \mathsf{len}_{\mathrm{ind}}$ such that $[\![v, \mathsf{len}_{\mathrm{val}}]\!][i] = 1$, we add a membership proof for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}_v, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ and a nonmembership proof otherwise.

To verify Π for (ℓ, V), we initialize $W \leftarrow \varnothing$ and check that for each $v \in V$, the membership proof for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}_v, \mathsf{len}_{\mathrm{ctr}}]\!]$ verifies while ensuring the validity of the nonmembership proof for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![|V| + 1, \mathsf{len}_{\mathrm{ctr}}]\!]$. In addition, for each counter ctr_w and for each index $i \in [\mathsf{len}_{\mathrm{ind}}]$ for which proofs were supplied, we verify the (non-)membership proofs of the form $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}_w, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ and use them to reconstruct a value w (membership proofs correspond to a 1 and nonmembership proofs to a 0), which we then add to W. To verify that the supplied V is correct, we finally check that $\{[\![v, \mathsf{len}_{\mathrm{val}}]\!] : v \in V\} = W$.

The pseudocode can be found in Fig. 2 and 3.

Theorem 1. *If* StAS *is a static AS satisfying membership correctness (Def. 9), then the* StAD *scheme in Fig. 2 and 3 satisfies lookup correctness (Def. 4).*

Proof. Let D be a dictionary commited to using StAD in Fig. 2 and 3, i.e. $d, \mathsf{st} \leftarrow \mathsf{StAD.Commit}(\mathsf{pp}, D)$ where $\mathsf{pp} \leftarrow \mathsf{StAD.Setup}(1^\lambda, \mathsf{sp})$.

We proceed in two cases. First, suppose $(\ell, V) \in D$ and let $m = |V|$. By construction, for label ℓ and all counter values $\mathsf{ctr} \in [0, m-1]$, StAD.Commit adds the element $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!]$ to set S (lines 14, 15 and 21); note that no element is added to S for $\mathsf{ctr} \geq m$. Additionally, for each $\mathsf{ctr} \in [0, m-1]$ and for each bit position $i \in [\mathsf{len}_{\mathrm{val}}]$, we add $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ to S if and only if the ith bit of $[\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!]$ is 1 (lines 18, 19 and 21). Each of these elements is unique due to fixed-length bit encodings.

By definition, $\mathsf{StAD.Lookup}(\mathsf{pp}, \mathsf{st}, D, \ell)$ returns two elements: a set V and a proof $\Pi^{lkup}_{\ell,V}$. A correct lookup returns $V = D[\ell]$ (line 26), fulfilling the first part of the correctness requirement.

It remains to show that the returned proof verifies. StAD.Lookup first initializes an empty dictionary Π. For the given label ℓ and for all $\mathsf{ctr} \in [0, m-1]$, it adds a proof of membership for element $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!]$ to $\Pi[\mathsf{ctr}]$ and a proof of non-membership for $\mathsf{ctr} = m$ to $\Pi[m]$ (line 35). Additionally, for $\mathsf{ctr} \in [0, m-1]$ and $i \in [\mathsf{len}_{\mathrm{val}}]$, we add a proof of membership to $\Pi[(\mathsf{ctr}, i)]$ for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ if the ith bit of $[\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!]$ is 1 and a proof of non-membership otherwise (line 40).

Combining the above observations with the correctness of StAS, it follows that the **while** loop on line 6 iterates $m + 1$ times (assuming the second part of the loop does not return 0):

- In the first m iterations, we have $b^* = 1$ and $b^{**} = 1$, since for all $\mathsf{ctr} \in [0, m-1]$, $\Pi[\mathsf{ctr}]$ is a correct proof of membership for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!]$. Moreover, $\Pi[(\mathsf{ctr}, i)]$ contains a proof of membership for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ if the ith bit is 1 and a proof of non-membership otherwise. All proofs are correct with all but negligible probability.
- In iteration $m + 1$, $b^* = 0$ and $b^{**} = 1$, since $\Pi[m]$ is a correct proof of non-membership for $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![m, \mathsf{len}_{\mathrm{ctr}}]\!]$ with all but negligible probability. The conditional statement on line 10 is met and we thus exit the loop.

Now, consider the second part of the loop (lines 14-24). By the correctness of StAS, all (non-)membership proofs in Π are valid. Additionally, since $\Pi[(\mathsf{ctr}, i)]$ contains a proof of membership if the i-th bit of ctr-th value is one and a proof of non-membership otherwise, then for each $\mathsf{ctr} \in [0, m-1]$, at the end of the **for** loop on line 23 we have $w = [\![V[\mathsf{ctr}], \mathsf{len}_{\mathrm{val}}]\!]$, which we add

```
StAD.Setup(1^λ, sp) → pp
    len_lab, len_val ← sp // bitlength for labels/values
    pp_AS ← StAS.Setup(1^λ, sp)
    return (pp_AS, len_lab, len_val)
StAD.Commit(pp, D) → d, st
    (pp_AS, len_lab, len_val) ← pp
    S ← ∅
    c ← max{|V| : (ℓ, V) ∈ D}
    len_ctr ← ⌊log_2 c⌋ + 1) // bit length for counter
    len_ind ← ⌊log_2 len_val⌋ + 1 // bit index
    for (ℓ, V) ∈ D do
        ctr = 0
        for v ∈ V do
            s* ← [[ℓ, len_lab]] ‖ [[ctr, len_ctr]]
            S ← S ∪ {s*}
            for i ∈ [len_val] do
                if [[v, len_val]][i] = 1 then
                    s ← [[ℓ, len_lab]] ‖ [[ctr, len_ctr]] ‖ [[i, len_ind]]
                    S ← S ∪ {s}
            ctr ← ctr + 1
    d_AS, st_AS ← StAS.Commit(pp_AS, S)
    return d = (d_AS, len_ctr), st = (st_AS, S)
StAD.Lookup(pp, st, D, ℓ) → V, Π^lookup_{ℓ,v}
    (pp_AS, len_lab, len_val) ← pp
    (st_AS, S) ← st
    V ← D[ℓ]
    Π ← { } // initialize dictionary
    c ← max{|V| : (ℓ, V) ∈ D}
    len_ctr ← ⌊log_2 c⌋ + 1)
    len_ind ← ⌊log_2 len_val⌋ + 1
    ctr ← 0
    while True do
        s* ← [[ℓ, len_lab]] ‖ [[ctr, len_ctr]]
        b*, π* ← StAS.Member(pp_AS, st, S, s*)
        Π[ctr] ← (b*, π*)
        if b* = 1 then
            for i ∈ [len_val] do
                s ← [[ℓ, len_lab]] ‖ [[ctr, len_ctr]] ‖ [[i, len_ind]]
                b, π ← StAS.Member(pp_AS, st, S, s)
                Π[(ctr, i)] ← (b, π)
        else return V, Π
        ctr ← ctr + 1
```

Fig. 2: First construction for compiling a static AS into a static AD (Part 1). We assume that the setup parameters sp specify the label and value bit lengths.

to W. Thus, the last **if-else** statement must return 1, completing the final requirement for lookup correctness.

Now, consider the second case: that ℓ is not in D (this is equivalent to showing that $(\ell, \perp) \in D$ and $(\ell, \varnothing) \in D$). There is no restriction on V being non-empty. Thus, this case is also captured by the first case, and lookup correctness follows. □

Theorem 2. *If* StAS *is a static AS satisfying membership security (Def. 10), then the scheme* StAD *(Fig. 2 and 3) satisfies lookup security (Def. 5).*

Proof. Suppose that StAS is secure and there exists adversary $\mathcal{A}$ that runs in time $\mathsf{poly}(\lambda)$ and can output tuple $(d, \ell, V, W, \Pi_V^{lkup}, \Pi_W^{lkup})$ such that $V \neq W$ and

$$\begin{aligned}&\mathsf{StAD.VerLookup}(\mathsf{pp}, d, \ell, V, \Pi_V^{lkup}) = 1\\ \wedge\ &\mathsf{StAD.VerLookup}(\mathsf{pp}, d, \ell, W, \Pi_W^{lkup}) = 1\end{aligned}$$

with probability more than $\mathsf{negl}(\lambda)$. Let $\mathsf{pp}_{\mathsf{AS}}$ be the public parameters for StAS. We will show that if such an adversary exists, then there exists an adversary that breaks the security of StAS. We proceed by case distinction.

Case 1: $|V| \neq |W|$. Without loss of generality, let $|V| < |W|$. Then there exist proofs $\pi \in \Pi_V^{lkup}[|V|]$ and $\hat{\pi} \in \Pi_W^{lkup}[|V|]$ for $s = [\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![|V|, \mathsf{len}_{\mathrm{ctr}}]\!]$ such that

$$\begin{aligned}&\mathsf{StAS.VerMember}(\mathsf{pp}_{\mathsf{AS}}, d, s, 0, \pi) = 1\\ \wedge\ &\mathsf{StAS.VerMember}(\mathsf{pp}_{\mathsf{AS}}, d, s, 1, \hat{\pi}) = 1\end{aligned}$$

However, $d, s, \pi, \hat{\pi}$ were computed in polynomial time, but StAS is assumed to satisfy membership security. We have thus reached a contradiction.

Case 2: $|V| = |W|$**.** Since $V \neq W$, there exists $v \in V$ such that $v \notin W$. Since StAD.VerLookup succeeds for the pair (ℓ, V), then there exists some $\mathsf{ctr} \in [0, |V| - 1]$ which v corresponds to. We first define the following set.

$$S = \left\{ [\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!] : \begin{array}{c} i \in [\mathsf{len}_{\mathrm{val}}], \\ [\![v, \mathsf{len}_{\mathrm{val}}]\!][i] = 1 \end{array} \right\}.$$

For each $s = [\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!] \in S$ we must have

$$1 \leftarrow \mathsf{StAS.VerMember}(\mathsf{pp}_{\mathsf{AS}}, d_{\mathsf{AS}}, s, 1, \pi)$$

where $(1, \pi) \leftarrow \Pi_V^{lkup}[(\mathsf{ctr}, i)]$.

By a similar argument, since StAD.VerLookup succeeds for (ℓ, W), then each element in W corresponds to a unique set of proofs in Π_W^{lkup} each of which is indexed by a unique counter in $[0, |V| - 1]$. In particular, there exists $w \in W$, $w \neq v$, which corresponds to the same counter ctr for which S is defined above. We analogously define another set:

$$T = \left\{ [\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{ctr}, \mathsf{len}_{\mathrm{ctr}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!] : \begin{array}{c} i \in [\mathsf{len}_{\mathrm{val}}], \\ [\![w, \mathsf{len}_{\mathrm{val}}]\!][i] = 1 \end{array} \right\}.$$

Since $v \neq w$, then $S \neq T$. Without loss of generality, let $s \in S \setminus T$. Since s corresponds to some $i \in [\mathsf{len}_{\mathrm{val}}]$ such that $[\![v, \mathsf{len}_{\mathrm{val}}]\!][i] = 1$ and $[\![w, \mathsf{len}_{\mathrm{val}}]\!][i] = 0$, then there exist proofs $(1, \pi) \in \Pi_V^{lkup}[(\mathsf{ctr}, i)]$ and $(0, \hat{\pi}) \in \Pi_W^{lkup}[(\mathsf{ctr}, i)]$ such that

$$\begin{aligned}&\mathsf{StAS.VerMember}(\mathsf{pp}_{\mathsf{AS}}, d, s, 0, \pi) = 1\\ \wedge\ &\mathsf{StAS.VerMember}(\mathsf{pp}_{\mathsf{AS}}, d, s, 1, \hat{\pi}) = 1.\end{aligned}$$

Once again, $d, s, \pi, \hat{\pi}$ were computed in polynomial time, thus breaking the membership security of StAS. The argument for $s \in T \setminus S$ similarly holds.

To break the lookup security of the compiled StAD, the adversary needs to violate the membership security of the underlying StAS scheme at least once. The verification procedure for some

```
1:  StAD.VerLookup(pp, d, ℓ, V, Π) → 0/1
2:      (pp_AS, len_lab, len_val) ← pp
3:      len_ind ← ⌊log_2 len_val⌋ + 1
4:      W ← ∅, ctr ← 0
5:      (d_AS, len_ctr) ← d
6:      while True do
7:          (b*, π*) ← Π[ctr]
8:          s* ← [[ℓ, len_lab]] ‖ [[ctr, len_ctr]]
9:          b** ← StAS.VerMember(pp_AS, d_AS, s*, b*, π*)
10:         if b* = 0 and b** = 1 then
11:             break
12:         else if b** = 0 then
13:             return 0 // Invalid proof
14:         w ← ϵ
15:         for i ∈ [len_val] do
16:             (b, π) ← Π[(ctr, i)]
17:             s ← [[ℓ, len_lab]] ‖ [[ctr, len_ctr]] ‖ [[i, len_ind]]
18:             b' ← StAS.VerMember(pp_AS, d_AS, s, b, π)
19:             if b' = 0 then
20:                 return 0 // Invalid proof
21:             else
22:                 w ← w‖b // Reconstruct bits
23:         W ← W ∪ {w}
24:         ctr ← ctr + 1
25:     if {[[v, len_val]] : v ∈ V} = W then
26:         return 1 // Valid proof
27:     elsereturn 0 // Invalid proof
```

Fig. 3: First construction for compiling a static AS into a static AD (Part 2).

V requires verifying $|V|(\mathsf{len}_{\mathrm{val}}+1)+1$ membership proofs using the underlying AS, which

$$\Pr\left[\begin{array}{c}\mathsf{pp}\leftarrow\mathsf{Setup}(1^\lambda)\\ d,\ell,V\neq W,\Pi_V^{lkup},\Pi_W^{lkup}\leftarrow\mathcal{A}(\mathsf{pp}):\\ \mathsf{StAD.VerLookup}(\mathsf{pp},d,\ell,V,\Pi_V^{lkup})=1\\ \wedge\mathsf{StAD.VerLookup}(\mathsf{pp},d,\ell,W,\Pi_W^{lkup})=1\end{array}\right]$$

$\leq (\max(|V|,|W|)(\mathsf{len}_{\mathrm{val}}+1)+1)\cdot\mathsf{negl}(\lambda)$ □

3.2 Construction 2

We now describe a second construction that offers different trade-offs. Similarly to before, the goal is to encode the dictionary as a set which we can then commit to using a static AS. For this construction, we assume that there is an ordering on the universe of values $\mathbb{V}$. We define an index function $\mathsf{Pos}(v,\mathbb{V})=i$ that takes as input a value $v\in\mathbb{V}$ and a value set $\mathbb{V}$ and outputs the index i of v in $\mathbb{V}$. For each dictionary entry (ℓ,V) and each value $v_j\in V$, we then add the element $\ell\|i$ to the set, where $\mathsf{Pos}(v_j,\mathbb{V})=i$ in the ordering.

Consider the dictionary in Fig. 1a and let $\mathbb{V}=\{0,1\}^3$ be the value set (here, we use the natural ordering of bit strings i.e., 000 is 0, 001 is 1, etc). We depict the second set encoding in Fig. 1c. Proving that 001 and 011 are the values associated with ℓ_1 using our second compiler requires proving: (1) membership of $\ell_1\|1$ and $\ell_1\|3$ and (2) non-membership of $\ell_1\|i$ for $i\in[8]\setminus\{1,3\}$.

Compiler Description. The compiler calls $\mathsf{StAS.Setup}$ to generate the AS's public parameters $\mathsf{pp}_{\mathsf{AS}}$. To commit to a dictionary D, we initialize set $S \leftarrow \varnothing$ and, for each pair $(\ell, V) \in D$ and value $v \in V$, add $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{Pos}(v, \mathbb{V}), \mathsf{len}_{\mathrm{ind}}]\!]$ to S. As a reminder, $\mathsf{len}_{\mathrm{ind}}, \mathsf{len}_{\mathrm{lab}}$, and $\mathsf{len}_{\mathrm{ctr}}$ denote the maximum bit length of a value, label, and counter, respectively. We then commit to S by running $d, \mathsf{st}_{\mathsf{AS}} \leftarrow \mathsf{StAS.Commit}(\mathsf{pp}_{\mathsf{AS}}, S)$, and return $d, (\mathsf{pp}_{\mathsf{AS}}, S)$.

To generate a lookup proof for some pair (ℓ, V), we initialize an empty dictionary Π, which will store the proofs for each value. For each $i \in [|\mathbb{V}|]$, we compute the element $s = [\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ and generate the corresponding (non-)membership bit b and proof π for s using $\mathsf{StAS.Member}$. The proof $\Pi[i] = (b, \pi)$ is then updatded. In particular, if $v \in V$, then $b = 1$ and π is a membership proof. On the other hand, if $v \notin V$, then $b = 0$ and π is a (non-)membership proof.

To verify proof Π for (ℓ, V), we must check that for each $(i, (b, \pi)) \in \Pi$, $b = 1$ if and only if $i = \mathsf{Pos}(v, \mathbb{V})$ and $v \in V$. Else, Π is invalid, and we output 0. Otherwise, we check the validity of each individual proof in Π. For each $i \in [|\mathbb{V}|]$ and $(b, \pi) \leftarrow \Pi[i]$, we require that π is a valid proof of (non-)membership for $s = [\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ in set S. If for each $i \in [|\mathbb{V}|]$, the corresponding (non-)membership proof is valid, then Π is a valid lookup proof and we return 1. Otherwise, we output 0.

The pseudocode can be found in Fig. 4.

Theorem 3. *If* StAS *is a static AS satisfying membership correctness (Def. 9), then the* StAD *scheme in Fig. 4 satisfies lookup correctness (Def. 4).*

Proof. Let D be a dictionary commited to using the StAD in Figure 4, i.e. $d, \mathsf{st} \leftarrow \mathsf{StAD.Commit}(\mathsf{pp}, D)$ where $\mathsf{pp} \leftarrow \mathsf{StAD.Setup}(1^\lambda, \mathsf{sp})$.

We proceed in two cases. First, suppose $(\ell, V) \in D$ and let $m = |V|$. By construction, for label ℓ and all values $v \in V$, $\mathsf{StAD.Commit}$ adds the element $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![\mathsf{Pos}(v, \mathbb{V}), \mathsf{len}_{\mathrm{ind}}]\!]$ to set S (line 11 and following); where $\mathsf{Pos}(v, \mathbb{V})$ provides a unique position number for each $v \in \mathbb{V}$ (and therefore for each $v \in V$).

By definition, $\mathsf{StAD.Lookup}(\mathsf{pp}, \mathsf{st}, D, \ell)$ returns two elements: a set V and a proof $\Pi^{lkup}_{\ell,V}$. A correct lookup returns $V = D[\ell]$ (line 19), fulfilling the first part of the correctness requirement.

It remains to show that the returned proof verifies. $\mathsf{StAD.Lookup}$ first initializes an empty dictionary Π (line 19). For the given label ℓ and for all $i \in [|\mathbb{V}|]$, it adds a proof of (non-)membership for element $[\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ to $\Pi[i]$ (line 21 and following). Note that by the above, it is a membership proof if and only if $i = \mathsf{Pos}(v, \mathbb{V})$ for some $v \in V$, and a nonmembership proof otherwise.

From the above observation, it follows that we the condition on line 31 must never be true, as we have membership proofs ($b = 1$) for each $i \in P$ by definition of P, and nonmembership proofs ($b = 0$) for each $i \notin P$. Combining the above observations with the correctness of StAS, it follows that the verification of membership proofs on line 34 always returns 1, so the condition on line 35 must also always fail. Therefore, verification must return 1, as it cannot return 0.

Now, consider the second case: that ℓ is not in D (this is equivalent to showing that $(\ell, \perp) \in D$ and $(\ell, \varnothing) \in D$). There is no restriction on V being non-empty. Thus, this case is also captured by the first case, and lookup correctness follows. □

Theorem 4. *If* StAS *is a static AS satisfying membership security (Def. 10), then the scheme* StAD *(Fig. 4) satisfies lookup security (Def. 5).*

Proof. Suppose that StAS is secure and there exists an adversary $\mathcal{A}$ that runs in time $\mathsf{poly}(\lambda)$ and outputs a tuple $(d, \ell, V, W, \Pi^{lkup}_V, \Pi^{lkup}_W)$ where $V \neq W$ and

$$\mathsf{StAD.VerLookup}(\mathsf{pp}, d, \ell, V, \Pi^{lkup}_V) = 1$$
$$\wedge\ \mathsf{StAD.VerLookup}(\mathsf{pp}, d, \ell, W, \Pi^{lkup}_W) = 1$$

```
1:  StAD.Setup(1^λ, sp) → pp
2:      len_lab, len_val ← sp // bit length of labels and values
3:      pp_AS ← StAS.Setup(1^λ, sp)
4:      return (pp_AS, len_lab, len_val)
5:  StAD.Commit(pp, D) → d, st
6:      (pp_AS, len_lab, len_val) ← pp
7:      S ← ∅
8:      len_ind ← ⌈log_2 |𝕍|⌉
9:      for (ℓ, V) ∈ D do
10:         for v ∈ 𝕍 do
11:             i ← Pos(v, 𝕍)
12:             s ← [[ℓ, len_lab]] ‖ [[i, len_ind]]
13:             S ← S ∪ {s}
14:     d, st_AS ← StAS.Commit(pp_AS, S)
15:     return d, (st_AS, S)
16: StAD.Lookup(pp, st, D, ℓ) → V, Π^lkup_{ℓ,v}
17:     (pp_AS, len_lab, len_val) ← pp, (st_AS, S) ← st
18:     len_ind ← ⌈log_2 |𝕍|⌉
19:     Π ← { }
20:     for i ∈ [|𝕍|] do
21:         s ← [[ℓ, len_lab]] ‖ [[i, len_ind]]
22:         b, π ← StAS.Member(pp_AS, st, S, s)
23:         Π[i] ← (b, π)
24:     return D[ℓ], Π
25: StAD.VerLookup(pp, d, ℓ, V, Π) → 0/1
26:     (pp_AS, len_lab, len_val) ← pp
27:     len_ind ← ⌈log_2 |𝕍|⌉
28:     P ← {Pos(v, 𝕍) : v ∈ V}
29:     for i ∈ [|𝕍|] do
30:         (b, π) ← Π[i]
31:         if (b = 1 ∧ i ∉ P) ∨ (b = 0 ∧ i ∈ P) then
32:             return 0 // Invalid proof
33:         s ← [[ℓ, len_lab]] ‖ [[i, len_ind]]
34:         b* ← StAS.VerMember(pp_AS, d, s, b, π)
35:         if b* = 0 then
36:             return 0 // Invalid proof
37:     return 1 // Valid proof
```

Fig. 4: Our second construction for compiling a static AS to a static AD.

with probability more than $\mathsf{negl}(\lambda)$. Let $\mathsf{pp}_{\mathsf{AS}}$ be the public parameters for StAS. We will show that if such an adversary exists, then there exists an adversary that breaks the security of StAS.

Since $V \neq W$, without loss of generality, there must be a $v' \in V$, such that $v' \notin W$. Let $i' = \mathsf{Pos}(v', \mathbb{V})$, and $s' = [\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![i', \mathsf{len}_{\mathrm{ind}}]\!]$. Now, since $v' \in V$, by definition of P (line 28) we must have $i' \in P$ during the verification of V, and $i' \notin P$ during the verification of W.

This insight then implies that $(1, \pi_V) = \Pi[i']$ for some π_V during the verification of V, while $(0, \pi_W) = \Pi[i']$ for some π_W during the verification of W, as otherwise one or both verifications would trigger the condition on line 31. For both verifications to succeed as assumed, we then further require that $1 = \mathsf{StAS.VerMember}(\mathsf{pp}_{\mathsf{AS}}, d, s', 1, \pi_V)$ and $1 = \mathsf{StAS.VerMember}(\mathsf{pp}_{\mathsf{AS}}, d, s', 0, \pi_W)$, otherwise the verification will fail on line 35. However, note that this now directly contradicts mem-

bership security (definition 10), as we assumed a polynomial-time adversary and a non-negligible probability of success. □

3.3 A Hybrid Approach

Both constructions presented above have their advantages. For example, proving and verifying the lookup of a pair (ℓ, V) is more efficient using construction 1 (Fig. 2 and 3) when $\mathsf{len}_{\mathrm{val}} \cdot |V| + |V| + 2 < \lceil \log_2 |\mathbb{V}| \rceil$. This insight suggests a hybrid approach in which one chooses the more efficient encoding of the pair (ℓ, V) as set elements based on the size of $|V|$. On the other hand, construction 2 (Fig. 4) is more easily adapted to support efficient updates: adding a new label pair (ℓ, v) to D would only entail computing the index $i \leftarrow \mathsf{Pos}(v, \mathbb{V})$ and then adding $s \leftarrow [\![\ell, \mathsf{len}_{\mathrm{lab}}]\!] \| [\![i, \mathsf{len}_{\mathrm{ind}}]\!]$ to the set. Likewise, deleting a pair would involve deleting the respective set element. We leave exploring these trade-offs and formalizing a hybrid construction for future work.

4 AD Lower Bounds

Here, we present lower bounds for lookups and updates for authenticated dictionaries built out of black-box authenticated sets. In Sec. 4.1, we present definitions and abstractions. We use these abstractions to prove our lower bounds for lookups in Sec. 4.2 and for updates in Sec. 4.3.

Assumptions. We assume that each value for a given label ℓ in the set of labels $\mathbb{L}$ is equally likely. We do not assume any specifics about the construction: the AD uses an AS in a black-box fashion by relying on a sequence of AS.Member invocations to conduct AD.Lookup and similarly for Update operations. Given a label $\ell \in \mathbb{L}$ in AD, let $\mathbb{V}$ be the set where any value associated with ℓ must lie. We also define the set $\mathcal{Q}$ as the set of items for which the underlying AS may be queried. Note that in this section, we consider bounds for *deterministic constructions*, i.e., constructions in which there exists a deterministic function $f : \mathbb{V} \to \mathcal{Q}^*$ such that once a value $v \in \mathbb{V}$ is purported to be the value associated with a label ℓ in the AD, the verifier runs AS.VerMember for the items in $f(v)$.

Lookup lower bound. Recall from Sec. 2.2 that the only authenticated operations for a given state of an authenticated set are AS.Member, which outputs an authenticated bit to signify membership or non-membership of an item in this set. Recall, also, that for a given state of an authenticated dictionary, for a given label $\ell \in \mathbb{L}$, the AD.Lookup function outputs a value V and a proof to authenticate that V is indeed the value associated with ℓ. If an AD is built in a black-box fashion out of an AS, each AD.Lookup must call AS.Member as a subroutine. The first bound in this section (formalized in Theorem 5) provides the minimum number of invocations to AS.Member, made by such an AD.Lookup as a function of the size of the space $\mathbb{V}$ of values V for each label ℓ. We prove, in fact, that, to provide a proof for the value of a given label ℓ, the AD.Lookup operation must make at least $\log(|\mathbb{V}|)$ invocations to AS.Member in expectation.

Update lower bound. Similarly to the case for AD.Lookup, when constructing an AD in a black-box fashion out of an AS, we consider the invocations made to AS.Update, made by AD.Update for a single update, i.e., an update set of size 1. In this case, we find a relationship between the number of AS.Member calls made by AD.Lookup and the number of AS.Update operations an AD.Update operation must invoke. In particular, we confirm the folklore that the complexity of AD.Lookup can be increased to achieve an AD.Update operation, making fewer calls to AS.Update.

4.1 Proof Machinery

We now provide the background needed to prove our lower bounds.

Encoding a dictionary using sets. Since constructions may consist of any "encoding" of a dictionary value $V \in \mathbb{V}$ as the membership or non-membership of a sequence of items in a set, we

Symbol	Meaning
$\mathbb{L}$	Set of labels for an AD.
$\mathbb{V}$	The set of values for an AD.
ℓ	Label in $\mathbb{L}$ being queried.
v	Purported value associated with ℓ. Sometimes called an *element*.
q	Input to $\mathsf{AS.Member}(\cdot, \cdot, q, \cdot)$.
$E_\ell(q, v)$	Output of $\mathsf{AS.Member}(\cdot, \cdot, q, \cdot)$ if v is the value associated with ℓ.
$E(q, v)$	Output of $\mathsf{AS.Member}(\cdot, \cdot, q, \cdot)$ if v is the val. associated with the given label.
$\mathcal{Q}_{\ell,v}$	Set of AS.Member queried items if v is the purported value for ℓ.
$\mathcal{Q}_\ell$	The set of potential AS.Member queries for ℓ for any value. $\mathcal{Q}_\ell = \cup_{v \in \mathbb{V}} \mathcal{Q}_{\ell,v}$.
m	$\lvert\mathcal{Q}_\ell\rvert$
n	$\lvert\mathbb{V}\rvert$

Table 1: Notation for AD-AS lower bound proofs.

define a general set of AS values for a given label $\ell \in \mathbb{L}$, as $\mathcal{Q}_\ell$. At a high level, $\mathcal{Q}_\ell$ is the entire set of items for which AS.Member may be invoked when AD.Lookup is called for the label ℓ. We also denote the set of values for the dictionary as $\mathbb{V}$. An encoding, which we define formally in Def. 13, is meant to capture, given a purported value v for a label ℓ, whether a particular AS.Member query is necessary and, if so, what it outputs.

Definition 13 (Encoding). *Let* AD *be any deterministically constructed authenticated dictionary built using black-box use of an authenticated set* AS*. Then for each label ℓ we define an* ***encoding*** $E_\ell : \mathcal{Q}_\ell \times \mathbb{V} \to \{0, 1, \perp\}$. *$\mathcal{Q}_\ell$ is defined as $\bigcup_{v \in \mathbb{V}} \mathcal{Q}_{\ell,v}$, where for each $v \in \mathbb{V}$, $\mathcal{Q}_{\ell,v}$ is the set of elements for which (non-)membership proof verifications are made to* AS.VerMember *when executing* AD.VerLookup *for ℓ when $D[\ell] = v$. $E_l(q, v) = 0$ means that we expect a non-membership proof for the given query q and value v, 1 a membership proof, and $\perp$ that the query is not relevant for the given value.*

In the case of $\perp$, this specifically means that the query provides no informartion about the value in question. Note that we often abbreviate E_ℓ as simply E since many properties that can be proven for a distinct label ℓ also hold overall, so the exact label is irrelevant.

Next, we formally define a valid encoding as one that allows the AD to inherit correctness and security from the underlying AS, where, given a purported value v for a label ℓ, is authenticated using a sequence of AS.Member calls.

Definition 14 (Valid Encoding). *Let* AD *be any AD constructed deterministically from black-box use of an AS,* AS*. If* AD *satisfies lookup security (Def. 5) and correctness (Def. 4) given* AS *satisfies membership security (Def. 10) and correctness (Def. 9), then the encodings E_ℓ for each $\ell \in L$ are* ***valid****.*

4.2 Lookups Lower Bound

In this section, we present proof that the number of StAS.VerMember invokations made by our construction is asymptotically optimal. Concretely, we show that for any *deterministic construction* of an AD from an AS, the clients need to verify $\Omega(\lceil \log n \rceil)$ queries to the AS to confirm a label-value pair in the AS in the worst case, where n is the size of the value space.

For a **deterministic construction**, the server provides the responses and proofs for a predetermined set of queries (these queries could be cached or not), and the client then needs to verify that for the given value, the server provides the expected queries and answers, and all the proofs verify.

Theorem 5. *Let* StAS *be a membership-secure static AS scheme (Def. 7). Let* StAD *be a deterministic construction of a lookup-secure static AD (Def. 2) built from* StAS*. For any such* StAS *and* StAD *schemes, there exists a dictionary* $D : L \to \mathbb{V}$ *such that for all labels* $\ell \in L$*,* StAD.VerLookup *must make* $\Omega(\lceil \log n \rceil)$ *invocations to* StAS.VerMember *where* $n = |\mathbb{V}|$*.*

Proof outline Each deterministic construction of an AD from an AS must mean that potential AD values for a label ℓ correspond to some encoding E_ℓ, as defined in Def. 13. We interpret the encoding function E_ℓ as a set of points in the boolean hyperspace, such that each query to the AS rules out specific values v but does not eliminate the possibility of others. We then show how lookup security Def. 5 is equivalent to the property that each orthant in the hyperspace has a single element (lemma 2). Finally, we show that any encoding violating the claimed lower bound fails this property of the hyperspace.

Proof. For $\ell \in L$ and $v \in \mathbb{V}$, let $\mathcal{Q}_{\ell,v}$ denote the set of elements for which (non-)membership proof verifications are made to StAS.VerMember when executing StAD.VerLookup for ℓ when $D[\ell] = v$. Let $\mathcal{Q}_\ell = \bigcup_{v \in \mathbb{V}} \mathcal{Q}_{\ell,v}$. Since StAD is a deterministic construction, for all $\ell \in L$, the value space $\mathbb{V}$ must admit an encoding $E : \mathcal{Q}_\ell \times \mathbb{V} \to \{0, 1, \perp\}$, as defined in Def. 13.

Claim. For any $\ell \in L$, $E : \mathcal{Q}_\ell \times \mathbb{V} \to \{0, 1, \perp\}$ is valid, per Def. 14, then E must satisfy the following: forall $v, v' \in \mathbb{V}$ where $v \neq v'$, there exists some query $q \in \mathcal{Q}_\ell$ such that $E(q, v) = b$ and $E(q, v') = 1 - b$ for $b \in \{0, 1\}$.

Proof of Claim: Suppose for $v \neq v' \in \mathbb{V}$, there exist no $q \in \mathcal{Q}_\ell$ such that $E(q, v) = b$ and $E(q, v') = 1 - b$ for $b \in \{0, 1\}$. Then, even if StAS has membership security, it is possible to provide proofs for both $D[\ell] = v$ and $D[\ell] = v'$, with respect to a single digest v, breaking membership security for StAD, making E invalid. □

Fig. 5b and 5c provide examples of valid encodings.

Encoding to hyperspace. To prove the lower bound, we map each encoding E to a hyperspace of $m = |\mathcal{Q}_\ell|$ dimensions as follows. Each query $q \in \mathcal{Q}_\ell$ corresponds to a hyperplane. The hyperplanes are pair-wise orthogonal, and each value $v \in \mathbb{V}$ corresponds to a set of points $\mathcal{P}_v$ such that $|\mathcal{P}_v| \geq 1$. Each hyperplane divides the hyperspace into two. For any query q and value $v \in \mathbb{V}$:

- If $E(q, v) = 0$, then the corresponding points $\mathcal{P}_v$ lie on one side of the hyperplane corresponding to q.
- If $E(q, v) = 1$, then the points in $\mathcal{P}_v$ must all lie on the other side of the hyperplane corresponding to q.
- If $E(q, v) = \perp$, then $\mathcal{P}_v$ contains points on both sides of the hyperplane corresponding to q.

Figure 5 gives a pictorial example. Taking the above claim into account and disregarding points of the same element v in the same orthant, the number of points $|\mathcal{P}_v|$ for each element v depends only on how many queries q are not relevant for v, so $|\{q | E(q, v) = \perp\}|$. We formulate this as a lemma:

Lemma 1. *Let* $v \in V$ *and* $q_v = |\{q | E(q, v) \neq \perp\}|$*, i.e., the number of queries relevant to* v*. Then* $|\mathcal{P}_v| = 2^{m - q_v}$*, where* m *is the number of queries.*

Proof. There are m hyperplanes. For each $v \in V$, there are $m - q_v$ queries irrelevant to verifying the given value. The number of points $|\mathcal{P}_v|$ starts at 1, doubling for each non-relevant query ($E(q, v) = \perp$). It follows that $|\mathcal{P}_v| = 2^{m - q_v}$. □

By Lemma 1, if we have $m = |\mathcal{Q}_\ell|$ distinct queries, and only q_v are necessary for some element v i.e., $q_v = |\{q | E(q, v) \neq \perp\}|$, then $|\mathcal{P}_v| = 2^{m - q_v}$. Each point in $\mathcal{P}_v$ lies in a distinct orthant of the m-dimensional space. Note that each orthant corresponds to a configuration of the AS for the elements in $\mathcal{Q}_\ell$ (whether these elements are present or absent), and each point in an orthant corresponds to a value v that can be verified successfully. Now consider the distribution of points to orthants:

Lemma 2. *For any valid encoding E, there is at most one point per orthant.*

Proof. Assume we have in total m queries which are relevant to verifying an element for a given label l. As described, we represent each query as a hyperplane and each element as one or multiple points in hyperspace (where the points cannot be part of any hyperplane). We can now consider each orthant a unique combination of elements in the underlying AS, so as a configuration of the AS. E.g., in Figure 5, the bottom-left orthant would represent that the element q_1 is present in the AS, while q_2 is absent. Towards a contradiction, assume that some orthant contains two points elements $v_1 \neq v_2$ (note that by construction, points of the same element cannot be in the same orthant). By construction of the hyperspace, note that, therefore, the server can provide correct proofs for the client to verify v_1 and v_2 by using an AS in the configuration represented by that orthant, which would break lookup security (Def. 5). □

Lower bound. To prove the lower bound on the necessary number of verifications, we will show that it is impossible to provide an encoding E for $n = 2^k + 1$ elements in which each element requires the verification of at most k queries.

Towards a contradiction, assume a valid encoding E exists for some label ℓ. Let $m = |\mathcal{Q}_\ell|$ and $u_{max} = \max_{v \in \mathbb{V}} |\{q | E(q, v) \neq \perp\}|$. By assumption, it holds that $u_{max} \leq k$. We have 2^m distinct orthants (each corresponding to a configuration of the underlying AS regarding these queries). If $m \geq k$, then by Lemma 1, $|\mathcal{P}_v| \geq 2^{m-u_{max}}$ for each element $v \in \mathbb{V}$. There are thus at least $(2^k + 1)2^{m-u_{max}} \geq 2^m + 2^{m-u_{max}}$ points in total, which is more than the number of orthants (2^m). By the pigeon-hole principle, this contradicts Lemma 2, which requires that each orthant contains at most one point. If $m < k$, then the number of elements $(2^k + 1)$ is greater than the number of orthants (2^m), again resulting in a contradiction.

Thus, if there are $n = 2^k + 1$ elements for some $k \in \mathbb{N}$, then at least one element requires $k + 1$ verifications, and the theorem follows. □

Theorem 6. *An optimal encoding has exactly k queries for each element.*

We start by showing a supporting corollary.

Corollary 1. *Let m be the number of queries and $q_v = |\{q | E(q, v) \neq \perp\}|$. Then, for any valid encoding we have that $\sum_{v \in V} 2^{m-q_v} \leq 2^m$.*

Proof. By Lemma 1, for each value v, we have 2^{m-q_v} points the hyperspace. Therefore, we have $\sum_{v \in V} 2^{m-q_v}$ in total. By Lemma 2, we know that for a valid encoding E, there can be at most one point per orthant. There are 2^m orthants, which yields $\sum_{v \in V} 2^{m-q_v} \leq 2^m$. □

Now we can forumlate a lower ound on the expected number of queries:

Lemma 3. *If the set of possible values for a given label ℓ is 2^k, then, for any* StAD *built using a* StAS*, the expected number of queries to* StAS*, to authenticate the value for ℓ is at least k, assuming each value is equally likely.*

Proof. For a given label l, let q_v be the number of queries made to StAS if label ℓ maps to the element (=value) v. From Lemma 1, we know the number of points $|p_v| = 2^{m-q_v}$. Thus, $q_v = m - (m - q_v) = m - \log(|p_v|)$. Also, from Corollary 1, $\sum_{v \in V} 2^{m-q_v} \leq 2^m$. $E[q_v] = E[m - \log(|p_v|)] = m - E[\log(|p_v|)]$. We also know that $E[\log(|p_v|)] \leq \log(E[|p_v|])$, by Jensen's inequality. Also, by definition, $\log(E[|p_v|]) = \log(\frac{1}{2^k} \sum_v |p_v|)$, by Corollary 1, $\log(E[|p_v|]) \leq \log(\frac{1}{2^k} 2^m) = m - k$. Thus, $E[q_v] = m - E[\log(|p_v|)] \geq m - (m - k) = k$. □

Additionally, we can also show that using less than k queries is not optimal:

Lemma 4. *If any element has strictly less than k queries, then the expected number of queries will be strictly larger than k.*

Proof.

$$E_{v\neq v^*}[q_v] = m - E_{v\neq v^*}[\log_2(\#copies)]$$
$$= m - \log_2 2^m - \log_2\left(\frac{1-2^{-k+i}}{2^k-1}\right) = \log_2\left(\frac{2^k-1}{1-2^{-k+i}}\right)$$

Therefore, we need to show that

$$k < \frac{k-i}{2^k} + \frac{2^k-1}{2^k}\log_2\left(\frac{2^k-1}{1-2^{-k+i}}\right) \iff i < \frac{2^k-1}{2^k}\log_2\left(\frac{2^k-1}{2^{k-i}-1}\right).$$

To prove that the above inequality holds, we will show that (1) the limit is i and (2) the term on the right is monotonically decreasing. To prove (1), we start by applying the rule that $\lim f(x)\cdot g(x) = \lim f(x)\cdot\lim g(x)$.

$$\lim_{k\to\infty}\frac{2^k-1}{2^k}\log_2\left(\frac{2^k-1}{2^{k-i}-1}\right) = 1\cdot\lim_{k\to\infty}\log_2\left(\frac{2^k-1}{2^k2^{-i}-1}\right)$$

$$= \lim_{k\to\infty}\log_2\left(\frac{2^k-1}{2^k2^{-i}-1}\cdot\frac{2^i}{2^i}\right) = i + \lim_{k\to\infty}\log_2(1) = i+0 = i.$$

Since i is constant, as k tends to infinity, the quantity inside the log tends to 1, and the entire log term tends to 0. Thus, the limit is i, and (1) holds.

We now show that (2) the function is monotonically decreasing (i.e., the function approaches $y = i$ from above). This is equivalent to showing that the first derivative is always negative for all $c \geq 1$ and valid k. We first simplify the expression by substituting $u = 2^k$ and taking the derivative with respect to u:

$$\frac{\partial}{\partial u}\left(\frac{u-1}{u}\log_2\left(\frac{u-1}{u2^{-i}-1}\right)\right) = \frac{(u-2^i)\ln\left(\frac{u-1}{u2^{-i}-1}\right)+(1-2^i)u}{u^2(u-2^i)\ln 2}$$

We now want to show that the derivative is always less than 0. Since the denominator is positive for all $u = 2^k > 2^i$ (and note that $k > i$ is a valid inequality to consider since the expected number of queries, k, must be greater than the number of queries we are removing, i, from one of the elements), then:

$$0 > \frac{(u-2^i)\ln\left(\frac{u-1}{u2^{-i}-1}\right)+(1-2^i)u}{u^2(u-2^i)\ln 2}$$
$$\iff 0 > (u-2^i)\ln\left(\frac{u-1}{u2^{-i}-1}\right)+(1-2^i)u$$
$$= (u-2^i)\left(\ln\left(\frac{u-1}{u-2^i}\right)+\ln i\right) - u(2^i-1)$$

Now observe the following. First, for all $i \geq 1$, $(u-2^i) < u$. Second, for all $i \geq 1$ and all $u \geq 2^i$, it holds that $0 \leq \ln\left(\frac{u-1}{u-2^i}\right) < 1$. We thus have that for all $i \geq 1$,

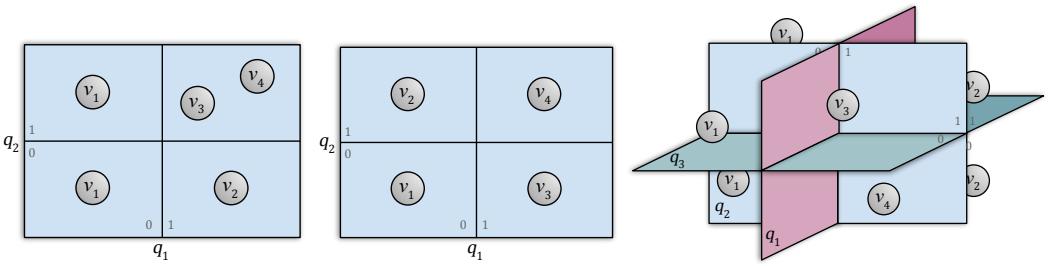
$$\ln\left(\frac{u-1}{u-2^i}\right)+\ln i \leq 1 + \ln i \leq 2^i - 1.$$

Putting these facts together, it holds that for $i \geq 1$,

$$(u-2^i)\left(\ln\left(\frac{u-1}{u-2^i}\right)+i\right) < u(2^i-1)$$

and thus the derivative is < 0 for $i \geq 1$. Using (1) and (2), we conclude that the expected number of queries is $> k$ (and approaches k as $k \to \infty$). □

Theorem 6 follows directly from Lemmas 3 and 4. □



	v_1	v_2	v_3	v_4
q_1	0	1	1	1
q_2		0	1	1

(a) 2D encoding.

	v_1	v_2	v_3	v_4
q_1	0	0	1	1
q_2	0	1	0	1

(b) Optimal 2D encoding.

	v_1	v_2	v_3	v_4
q_1	0	1	1	1
q_2		1	0	0
q_3			1	0

(c) 3D encoding.

Fig. 5: (Bottom) Three encodings of a value space $\mathbb{V} = \{v_1, v_2, v_3, v_4\}$ for some label ℓ and their (Top) hyperspace representation. In (a), we give an unsuccessful encoding. Although query q_1 can distinguish element v_1 from v_2, v_3, v_4, and query q_2 can distinguish v_2 from v_3, v_4, neither query can disambiguate v_3 from v_4. This fact is pictorially represented as v_3 and v_4 being assigned to the same quadrant in 2D space. In (b) we give a valid and optimal 2D encoding, with the expected number of queries to the AS for any value for ℓ is 2. In (c), we give a valid but non-optimal 3D encoding that can distinguish the four elements using 3 queries. Here, the expected number of queries for a given value of ℓ is 2.25.

4.3 Update Lower Bound

In this section, we turn our attention to an updatable AD (Def. 3) constructed out of an updatable AS (Def. 8). We ask how many times AS.Update must be invoked by AD.Update for such a construction.

Theorem 7. *For any valid encoding E, for any label ℓ, there exists v and v', such that to update $D[\ell] = v$ to $D[\ell] = v'$, we need $\frac{\log(n-1)}{\log M}$ many updates, or $\approx \frac{\log n}{\log M}$ for large n (where n is the size of the valueset).*

Note that if we only need to verify M set membership proofs for each AD lookup, then to change the value of the lookup, one of these M set memberships must return a different result to maintain lookup security Def. 5. By iterating this argument, we see that with u updates to the underlying AS, we can "reach" at most M^u different values.

We show some supporting lemmas in order to prove the theorem. First, we bound the number of states possible for the AD after a single update of the AS:

Lemma 5. *Let AD be any updatable, deterministically constructed AD built using black-box use of an updateable AS scheme, AS. Let D be the dictionary committed to by AD. Assume that, for some label ℓ, the encoding E_ℓ is valid (Def. 14). Let $m = |\mathcal{Q}_\ell|$ and*

$$M = \max_v |\{q | E_\ell(q, v) \neq \perp\}|.$$

If $D[l] = v$ for some value v before the update, then there are at most $M \leq m$ distinct values $v' \in \mathbb{V}$ such that $D[\ell] = v'$ after a single update of AS.

Proof. Let $\vec{q} = \{q_1, ..., q_m\} \in \{0,1\}^m$ denote the vector of query states relevant for some label ℓ in D, where $q_i = 0$ if the query is not present in the AS, and $q_i = 1$ if the query is present. $\vec{q}$ be such

that $D[\ell] = v$ for some value v; by assumption such a v must exist. Note that only M membership proofs from AS are needed to prove $D[\ell] = v$.

Any update of AS to a query $q \notin \mathcal{Q}_{\ell,v}$ (including queries not relevant for label ℓ) will not change $D[\ell]$ for a valid encoding, since the proof is still valid. Further note that for each $q_i \in \mathcal{Q}_{\ell,v}$, the only update that can be made is to remove it from AS if present or vice versa. Therefore, at most M updates invalidate the proof for $D[\ell] = v$, potentially changing the value of $D[\ell]$ to some $v' \neq v$. □

We can now bound the number of AD states after u updates of the AS:

Lemma 6. *Let* AD *be any updatable, deterministically constructed AD built using black-box use of an updateable AS scheme,* AS*. Let* D *be the dictionary committed to by* AD*. Assume that for some label* ℓ*, the encoding* E_ℓ *is valid (Def. 14), and let* $m = |\mathcal{Q}_\ell|$ *and*

$$M = \max_v |\{q | E_\ell(q, v) \neq \perp\}|.$$

If either (1) each state of AS *is a valid assignment for* ℓ *or (2)* $M = m$*, then the set of values* $\{v'_1, v'_2, ...\}$ *such that* $D[\ell] = v'_i$ *after* u *updates has size at most* M^u*.*

Note that the preconditions for this lemma hold for both constructions.

Proof. Let $D[\ell] = v$ denote the value mapped to by ℓ before any updates happen. We proceed by case distinction.

Case (1): By Lemma 5, after a single update, there are at most M distinct values v' such that $D[\ell] = v'$. Let $D[l] = v'$ denote the new value after a single update. Applying Lemma 5 again yields that, from each value, we can reach at most M new values v'', i.e., at most M^2 values overall. Note that one of the possible updates is to revert v' to v. Repeating this argument yields M^u possible values after, at most, u updates.

Case (2): Any updates that might change $D[\ell]$ must modify one of the m relevant queries. After a single update, $\vec{q}$ has size m (one bit of $\vec{q}$ is flipped from 0 to 1 or vice versa). For each such vector, we can make $m - 1$ updates that do not reverse the previous change, yielding $m(m-1) + m = m^2$ different configurations after at most 2 updates. Repeated application of this argument yields at most $\leq m^u = M^u$ distinct configurations after u updates. Each configuration corresponds to at most 1 assignment for $D[\ell]$ for a valid encoding. □

Now we can prove Theorem 7:

Proof. By Lemma 6, we reach $\leq M^u$ distinct values, using at most u updates. Applying this yields

$$n - 1 \leq M^u \Leftrightarrow \log(n-1) \leq u \log M \Leftrightarrow \frac{\log(n-1)}{\log M} \leq u.$$

□

The preconditions hold for both constructions. Using the theorem, we get a lower bound of $\frac{\log n}{\log \log n}$ for Compiler 1 (Fig. 2 and 3), in which the update cost is at most $\log n$. The bound is almost tight as $\log \log n$ is small for any reasonable n. For Compiler 2 (Fig. 4), we have an actual update cost of 2 (assuming that each label maps only to a single value and not to sets of values), while the bound from Theorem 7 yields $\frac{\log n}{\log n} = 1$, so again the bound is almost tight.

5 From Updatable AD to Memory Checker

It may seem trivial to build a memory checker given an updatable AD as defined in Def. 3. We could treat the memory locations of the memory checker as AD keys and the data at that location as the AD's value for that key. Unfortunately, such a construction requires a subtler approach, as discussed below.

(In-)security of a naïve construction. We arrive at obstacles in proving the RoW security for a memory checker built trivially from an AD, as described above. Recall the definition of security (value-binding) for an updatable AD, given in Def. 6. This definition states, informally, that if two lookups of a label ℓ are conducted at the same time, i.e., $\mathsf{UpdAD.VerifyUpdate}$ is called concerning digest d, the two values verified for ℓ must match. Also, note that the $\mathsf{UpdAD.VerifyUpdate}$ function does not take as input the set of updates U. Thus, (1) from a purely syntactical point of view, the $\mathsf{UpdAD.VerifyUpdate}$ function cannot be used by itself to implement a $\mathsf{MemCheck.VerWrite}$, (2) the value-binding guarantee is semantically weaker than RoW security – even if a read value, r_ℓ, for a label ℓ differs from the most recent value written value w_ℓ for ℓ as long as the provided read cannot be equivocated, value-binding can be satisfied in the AD, but RoW security of the memory checker would not be satisfied.

5.1 From an AD with Write-Committing Security

We first introduce a new security notion for ADs, which we call *write-committing*. Write-committing guarantees that the proof output by Update commits to the number of changes made by a set of updates U. In particular, let d and d^* be digests from consecutive time stamps, and let $L \in \mathbb{L}$ be the set of labels to which updates were made. If $\mathsf{VerifyUpdate}$ passes, then we require that $|U| = |L|$ with high probability. However, note that $\mathsf{VerifyUpdate}$, by itself, does not take the set of updates U as input, so we introduce a function f that takes as input a digest d, its update d^* and the update proof π^{updt} and outputs an integer, symbolizing the number of updates purportedly made between d and d^*.

In the remainder of this section, we prove that this property and lookups for all expected changes are equivalent to RoW security.

Definition 15. *Let $\lambda \in \mathbb{N}$. UpdAD has **write-committing security** if there exists* counting function *f, computable in time polynomial in its input size, such that the following 2 statements hold:*

$$\Pr\left[\begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{sp}) \\ (\cdot, d^*, \pi^{updt}) \leftarrow \mathsf{Update}(\mathsf{pp}, d, \mathsf{st}, U), \\ 1 \leftarrow \mathsf{VerifyUpdate}(\mathsf{pp}, d, d^*, \pi^{updt}) : \\ f(d, d^*, \pi^{upd}) = |U| \end{array}\right] = 1.$$

and, for all probabilistic $\mathsf{poly}(\lambda)$-time adversaries $\mathcal{A}$ and any set of updates U:

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{sp}) \\ (V \neq V^*, \pi^{updt}, \pi, \pi^*, d, d^*) \leftarrow \mathcal{A}(\mathsf{pp}) : \\ \qquad f(d, d^*, \pi^{upd}) \neq |L| \\ \qquad 1 \leftarrow \mathsf{VerifyUpdate}(\mathsf{pp}, d, d^*, \pi^{updt}) \\ L = \left\{ \ell \in \mathbb{L} \;\middle|\; \begin{array}{r} 1 \leftarrow \mathsf{VerLookup}(\mathsf{pp}, d, \ell, V, \pi) \\ 1 \leftarrow \mathsf{VerLookup}(\mathsf{pp}, d^*, \ell, V^*, \pi^*) \end{array} \right\} \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Compiler Description The strong security notion of write-committing enables a relatively simple memory checker construction. $\mathsf{MemCheck}$ first initializes an updateable AD with write-committing

```
MemCheck.Setup(1^λ, sp) → pp
    len_lab, len_val ← sp // bit length of memory locations and values
    pp ← UpdAD.Setup(1^λ, sp)
    return pp
MemCheck.Init(pp, st) → T, U
    d, st′ ← UpdAD.Commit(pp, st)
    return ((pp, d), (pp, st′, d))
MemCheck.Read(i, U) → (v, π)
    (pp, st, d) ← U
    (v, π) ← UpdAD.Lookup(pp, st, i)
    return (v, π)
MemCheck.VerRead(π, i, v, T) → b
    (pp, d) ← T
    b ← UpdAD.VerLookup(pp, d, i, v, π)
    return b
MemCheck.Write(i, v, U) → (δ, π, U′)
    (pp, st, d) ← U
    st′, d′, π_1 ← UpdAD.Update(pp, d, st, ((i, v)))
    v*, π_2 ← UpdAD.Lookup(pp, st′, i)
    U′ ← (pp, st′, d′)
    π ← (v*, π_1, π_2)
    return (d′, π, U′)
MemCheck.VerWrite(i, v, T, δ, π) → (b, T′)
    (pp, d) ← T
    (v*, π_1, π_2) ← π
    if v ≠ v* then
        return (0, T)
    b_1 ← UpdAD.VerifyUpdate(pp, d, δ, π_1)
    b_2 ← UpdAD.VerLookup(pp, δ, i, v, π_2)
    b ← b_1 ∧ b_2
    if b ≠ 1 then
        return (0, T)
    T′ ← (pp, δ)
    return (b, T′)
```

Fig. 6: Algorithms for constructing a memory checker using an AD. Note that all locations in a memory checker are assumed to be initialized at Init, thus we omit op from UpdAD.Update's inputs and implicitly assume op = update.

security and counting function f. A one-to-one correspondence exists between the untrusted memory locations of MemCheck and the key-value pairs in UpdAD. In particular, writing a value v to location i of MemCheck requires updating the pair (i, v) via UpdAD.Update and then generating a lookup proof using UpdAD.Lookup for the same pair. The lookup proof ensures that the value written to location i is the same value read from location i. Correspondingly, verifying a write requires verifying that the proofs generated by UpdAD.Update and UpdAD.Lookup, respectively, as well as checking that f outputs 1, i.e., only one label's value changed in UpdAD; MemCheck.VerWrite only succeeds if all checks verify.

Reading a value from a location i requires running UpdAD.Lookup on the key i to obtain the value and lookup proof, (v, π), which is returned. Verifying a read then runs UpdAD.VerLookup on

the pair (i, v) with the pair π. MemCheck.VerRead succeeds if π is correct. The pseudocode can be found in Fig. 6.

Lemma 7. *If* UpdAD *is an updateable AD satisfying write-committing (Def. 15) with counting function f, then the scheme* MemCheck *(Fig. 6) satisfies RoW security (Def. 12).*

Proof. Suppose that MemCheck is not RoW secure and that there exists an adversary $\mathcal{A}$ that runs in time $\mathsf{poly}(\lambda)$ that can output a tuple

$$(\ell, v, v^*, \pi, \pi^*, (T_j, \ell_{j-1}, v_{j-1}, \delta_{j-1}, \pi_{j-1})_{j\in[1,n]}, k \in [0, n])$$

such that $v \neq v^*$ and

$$1 = \mathsf{MemCheck.VerRead}(\ell, v, T_0, \pi) = \mathsf{MemCheck.VerRead}(\ell, v^*, T_k, \pi^*)$$

and for all $j \in [1, n]$, (1) $\ell \neq \ell_j$, and, (2) with probability more than $\mathsf{negl}(\lambda)$, $\mathsf{MemCheck.VerWrite}(\ell_j, v_j, T_j, \delta_j, \pi_j) = (1, T_{j+1})$. By construction, MemCheck.VerWrite checks that $f(T_{j-1}, T_j, \pi_{j-1}) = 1$ for each $j \in [1, n]$. Then, consider the following two cases:
Case 1: $k = 0$, and thus a value binding adversary $\mathcal{A}^*$ can use $\mathcal{A}$ as a subroutine to find $d, \ell, v, v^*, \pi, \pi^*$, such that $v \neq v^*$ such that $(\mathsf{pp}, d) \leftarrow T_0$ and $\mathsf{UpdAD.VerLookup}(\mathsf{pp}, d, \ell, v, \pi) = 1$ and $\mathsf{UpdAD.VerLookup}(\mathsf{pp}, d, \ell, v^*, \pi^*) = 1$, thus breaking value-binding security for UpdAD.
Case 2: If $k > 0$, then, with less than negligible probability, there exists some timestep $i \in [1, k]$, such that $(\mathsf{pp}, d_{i-1}) \leftarrow T_i$, $f(d_{i-1}, d_i, \pi_i) = 1$, and both ℓ_{i-1} and ℓ are in the set

$$L_j = \left\{ \ell' \in \mathbb{L} \;\middle|\; \begin{matrix} 1 \leftarrow \mathsf{VerLookup}(\mathsf{pp}, d_i, \ell', v_i, \pi_i^{lkp}), \\ 1 \leftarrow \mathsf{VerLookup}(\mathsf{pp}, d_{i-1}, \ell', v_{i-1}, \pi_{i-1}^{lkp}) \end{matrix} \right\}.$$

Thus, a write-committing adversary $\mathcal{A}^*$ can use $\mathcal{A}$ to subvert write-committing security. □

5.2 From an AD with Value Binding Security

We now construct a memory checker using an AD that only satisfies value binding (Def. 6).

Compiler Description Given an AD with value-binding, we can build a memory checker as follows. The construction comprises a hierarchical sequence of ADs, where AD_ϵ contains only two labels: $0, 1$. The respective values for these labels are d_0 and d_1, the commitments to the next level of ADs, AD_0 and AD_1; in particular, d_0 and d_1 commit to all the values whose labels' first bit is 0 and 1, respectively. Each AD_ℓ contains only two labels: the labels for its left child $\ell \| 0$ and right child $\ell \| 1$ and whose corresponding values are commitments to $AD_{\ell\|0}$ and $AD_{\ell\|1}$, respectively. The leaf nodes of this hierarchical data structure are ADs containing one label (corresponding to the MemCheck memory location) and its value (the value contained at that memory location).

To verify an update, we must check that no values other than the requested leaf are modified. This can be done by verifying that only one label-value pair has been modified in each AD along the path from leaf to root. This construction makes $\mathcal{O}(\log(n))$ queries to updateable ADs for each read/write operation. The pseudocode can be found in Fig. 7 and 8.

```
MemCheck.Setup(1^λ, sp) → pp
    n, len_val ← sp // Length of memory locs/vals
    len_lab ← log n
    pp ← UpdAD.Setup(1^λ, sp)
    return pp

MemCheck.Init(pp, st) → T, U
    [v_1, ..., v_n] ← st // v_i is the value at memory location i
    st' ← [][] // An arr of log n arrs
    for α ∈ [log n − 1] do
        ℓ = log n − α // Iterate starting at deeper tree levels
        st'[ℓ] = [] // Stores the ℓth level of the tree
        st'' ← [] // Stores the digests of level ℓ
        for j ∈ [2^ℓ − 1] do
            e ← [(0, st[2j]), (1, st[2j + 1])]
            d_j, st_j ← UpdAD.Commit(pp, e)
            st'[ℓ][j] ← (d_j, st_j)
            st''.append(d_j)
        st ← st'' // Treat as state for the next level
    (d_root, st_root) ← st'[0][0] // Digest of the root node
    return ((pp, d_root), (pp, st'))

MemCheck.Read(i, U) → (V, π)
    V ← ∅, Π ← ∅
    (pp, st) ← U
    ℓ ← log n
    b_1 b_2 ... b_ℓ ← [[i, ℓ]]
    curr ← st[0][0] // Start at the root AD
    for j ∈ [ℓ] do
        (v_j, π_j) ← UpdAD.Lookup(pp, curr, b_j)
        π.append((v_j, π_j))
        V ← v_j
        if j < ℓ then
            k ← from_bits(b_1...b_j)
            (d_{j,k}, st_{j,k}) ← st[j][k]
            curr ← st_{j,k}
    return (V, π)

MemCheck.VerRead(pp, i, T, V, π) → b
    (pp, d) ← T
    ℓ ← log n
    ((v_j, π_j))_{j∈[ℓ]} ← π
    b_1 b_2 ... b_ℓ ← [[i, ℓ]]
    b ← 1, V* ← 0
    for j ∈ [ℓ] do
        b ← b ∧ UpdAD.VerLookup(pp, d, b_j, v_j, π_j)
        V* ← v_j
        if j < ℓ then
            d ← v_j
    return b ∧ (V == V*)
```

Fig. 7: Algorithms for constructing a memory checker using an authenticated dictionary that is not write-committing but is value-binding (Part 1). We assume that all locations in the memory checker are initialized and that op = update.

$\mathsf{MemCheck.Write}(i, v, U) \to (\delta, \pi, U')$
  $(\mathsf{pp}, \mathsf{st}) \leftarrow U$
  $\pi \leftarrow []$
  $\mathsf{st}' \leftarrow \mathsf{st}$
  $\ell \leftarrow \log n$
  $b_1 b_2 \dots b_\ell \leftarrow [\![i, \ell]\!]$
  $d_{\mathsf{curr}} \leftarrow v$
  for $j \in [\ell]$ **do**
    $\alpha \leftarrow \log n - j$
    if $\alpha > 0$ **then** $k \leftarrow \mathsf{from\_bits}(b_1 b_2 \dots b_\alpha)$ **else** $k \leftarrow 0$
    $(d_\alpha, \mathsf{st}_\alpha) \leftarrow \mathsf{st}[\alpha][k]$
    $(d_j^c, \pi_j^c) \leftarrow \mathsf{UpdAD.Lookup}(\mathsf{pp}, \mathsf{st}_\alpha, 1 - b_{\alpha+1})$
    $(d_j, \pi_j) \leftarrow \mathsf{UpdAD.Lookup}(\mathsf{pp}, \mathsf{st}_\alpha, b_{\alpha+1})$
    $(\mathsf{st}_j^{updt}, d_j^{updt}, \pi_\alpha^{updt}) \leftarrow \mathsf{UpdAD.Update}(\mathsf{pp}, d_\alpha, \mathsf{st}_\alpha, (b_{\alpha+1}, d_{\mathsf{curr}}))$
    $\mathsf{st}'[j][k] \leftarrow (d_j^{updt}, \mathsf{st}_j^{updt})$
    $(d_j', \pi_j') \leftarrow \mathsf{UpdAD.Lookup}(\mathsf{pp}, \mathsf{st}_j^{updt}, b_{\alpha+1})$
    $(\cdot, \pi_j^{c\prime}) \leftarrow \mathsf{UpdAD.Lookup}(\mathsf{pp}, \mathsf{st}_j^{updt}, 1 - b_{\alpha+1})$
    $\pi.\mathsf{append}((d_j^c, \pi_j^c, \pi_j^{c\prime}, d_j, \pi_j, d_j', \pi_j{}', d_j^{updt}, \pi_j^{updt}))$
    $d_{\mathsf{curr}} \leftarrow d_j^{updt}$
  $U' \leftarrow (\mathsf{pp}, \mathsf{st}')$
  return $(d_{\mathsf{curr}}, \pi, U')$

$\mathsf{MemCheck.VerWrite}(i, v, T, \delta, \pi) \to (b, T')$
  $((d_j^c, \pi_j^c, \pi_j^{c\prime}, d_j, \pi_j, d_j', \pi_j', d_j^{updt}, \pi_j^{updt}))_{j \in [\log n]} \leftarrow \pi$
  $\ell \leftarrow \log n$
  $d_{\mathsf{curr}} \leftarrow d$
  $d_{\mathsf{curr}}^{updt} \leftarrow \delta$
  $b_1 b_2 \dots b_\ell \leftarrow [\![i, \ell]\!]$
  $b \leftarrow 1$
  for $j \in [\ell]$ **do**
    $b \leftarrow b \wedge \mathsf{UpdAD.Update}(\mathsf{pp}, d_{\mathsf{curr}}, d_{\mathsf{curr}}^{updt}, \pi_j^{updt})$
    // Check that off branch did not change
    $b_1 \leftarrow \mathsf{UpdAD.VerLookup}(\mathsf{pp}, d_{\mathsf{curr}}, 1 - b_j, d_j^c, \pi_j^c)$
    $b_2 \leftarrow \mathsf{UpdAD.VerLookup}(\mathsf{pp}, d_{\mathsf{curr}}^{updt}, 1 - b_j, d_j^c, \pi_j^{c\prime})$
    // Check previous and updated digests for changed branch
    $b_3 \leftarrow \mathsf{UpdAD.VerLookup}(\mathsf{pp}, d_{\mathsf{curr}}, b_j, d_j, \pi_j)$
    $b_4 \leftarrow \mathsf{UpdAD.VerLookup}(\mathsf{pp}, d_{\mathsf{curr}}^{updt}, b_j, d_j', \pi_j')$
    $d_{\mathsf{curr}} \leftarrow d_j, d_{\mathsf{curr}}^{updt} \leftarrow d_j'$
    $b \leftarrow b \wedge \bigwedge_{i=1}^{4} b_i$
  $b \leftarrow b \wedge (d_{\mathsf{curr}} == v)$
  if $b \neq 1$ **then return** (b, T)
  return (b, δ)

Fig. 8: Algorithms for constructing a memory checker using an authenticated dictionary that is not write-committing but is value-binding (Part 2).

Lemma 8. *If* UpdAD *is an updateable AD satisfying value binding (Def. 6), then the scheme* MemCheck *(Fig. 7 and 8) satisfies RoW security (Def. 12).*

Proof. Suppose that MemCheck is not RoW secure and that there exists an adversary $\mathcal{A}$ that runs in time $\mathsf{poly}(\lambda)$ that can output a tuple $(\ell, v, v^*, \pi, \pi^*, (T_j, \ell_{j-1}, v_{j-1}, \delta_{j-1}, \pi_{j-1})_{j\in[1,n]})$ such that $v \neq v^*$ and

$$1 = \mathsf{MemCheck.VerRead}(\ell, v, T_0, \pi),$$
$$1 = \mathsf{MemCheck.VerRead}(\ell, v^*, T_k, \pi^*),$$

and all $j \in [1, n]$, $\ell \neq \ell_{j-1}$ and $\mathsf{MemCheck.VerWrite}(\ell_j, v_j, T_j, \delta_j, \pi_j) = (1, T_{j+1})$, with probability $> \mathsf{negl}(\lambda)$. Let $b_1, ..., b_{m+1}$ be the bitwise representation of ℓ. Note that our construction verifies, for each AD along the path from AD_ϵ through $\mathsf{AD}_{b_1,...,b_m}$ by verifying lookups for $1 - b_i$ yield the same value before and after the update for all $i \in [m + 1]$ and, since each $\mathsf{AD}_{b_1,...,b_i}$ has exactly two keys, only one key must have been modified, with high probability. If an adversary subverts this property, there must be some $j \in [k - 1]$ and some $\mathsf{AD}_{b_1,...,b_p}$ such that the lookup for b_{p+1} does not change when doing AD.VerLookup before and after the update but AD.VerLookup verifies a different value as part of the verification for ℓ, v^*, breaking value binding. □

6 From Memory Checkers to Authenticated Dictionaries

Memory checkers have a more limited functionality as compared to ADs, since they only provide locations instead of more general keys. Below, we discuss a simple hash-table-based solution, followed by a cuckoo hashing solution that shows a very efficient solution (amortized $\mathcal{O}(1)$ overhead for each operation) for building an AD, given an RoW memory checker.

We present these constructions at a high level and without proof.

A simple construction. A technique similar to a standard hash table [51] can be used to instantiate an AD using a memory checker. The location to store the data related to a given key is chosen using a hash function. Since the hash function may have collisions, a memory location m contains the commitment to an authenticated linked list, i.e., a hash chain. The linked list at location m commits to all the changes made to values for the labels that hash to m, in chronological order. This construction satisfies the value-binding property.

$\mathcal{O}(1)$ **solution.** A well-known solution for storing a general key-value store, given one or more arrays, is Cuckoo-hashing [73]. In the average case, cuckoo hashing provides $\mathcal{O}(1)$-time insertions and amortized $\mathcal{O}(1)$-time resizing, if needed. In order to implement an AD's update operation, a server could show that the locations updated in the memory checker correspond to the correct changes for a cuckoo-hashing-based insertion, and a lookup would correspond to providing proofs of memory state at the expected memory locations for a given key using the hash functions for the cuckoo hashing scheme.

More generally, constructing an AD out of a memory-checker is equivalent to several existing lines of work, e.g., hash-based data structures [29,51] and memory-based key-value stores [40,41]. Modifications similar to those provided above can be made to general solutions to build ADs from memory checking.

7 Memory Checking Lower Bounds

We now discuss lower bounds for constructions of memory checkers built from ADs. Due to the stronger security guarantees of memory checkers and a slew of existing work in data structures, there exist $\mathcal{O}(1)$ solutions for building ADs from memory checkers, e.g., using Cuckoo hashing, as detailed in Sec. 6.

In Sec. 5, we presented two constructions for a memory checker based on an AD. In particular, we detailed the fact that the generally accepted memory checking security definition of read-overwrite security (RoW, defined in Def. 12), is stronger than the definition most commonly satisfied by authenticated dictionaries – value binding (Def. 6). In Sec. 5.1, we presented a definition for write-committing security for an AD, which can then be used to construct a RoW-secure memory checker with only $\mathcal{O}(1)$ overhead for each operation. However, our black-box construction in Sec. 5.2, for a memory checker using an AD with only value-binding security requires $\mathcal{O}(\log(n))$-overhead for write operations. Based on our attempts to improve this construction, we arrive at the following conjecture, whose proof we leave for future work.

Conjecture 1. Let UpdAD be an updatable AD with value-binding security. If $\mathsf{MemCheck}$ is an RoW-secure memory checker with n memory locations, built with a deterministic construction that uses UpdAD as a black box, then either (1) UpdAD is write-committing or (2) if UpdAD is not write-committing, but only value-binding (Def. 6), then each $\mathsf{MemCheck.Write}$ must make $\Omega(\log n)$ calls to $\mathsf{UpdAD.Lookup}$ for $\mathsf{MemCheck}$ to be RoW-secure.

If Conj. 1 is true, then the existing lower bounds for memory checking can be translated to lower bounds for ADs (e.g., [18,90]).

References

1. Rfc 6962: Certificate transparency (2013), https://datatracker.ietf.org/doc/html/rfc6962#section-2.1.2
2. Ajtai, M.: The invasiveness of off-line memory checking. In: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. pp. 504–513 (2002)
3. Algorand Developer Documentation: Algorand state proofs overview (2025), https://developer.algorand.org/docs/get-details/stateproofs/
4. Amazon Web Services, Inc.: Amazon Quantum Ledger Database (Amazon QLDB) Developer Guide. Amazon Web Services, Inc. (2025), https://docs.aws.amazon.com/pdfs/qldb/latest/developerguide/qldb-dg.pdf, accessed: 2025-05-12
5. Anagnostopoulos, A., Goodrich, M.T., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: Information Security (2001)
6. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Khan, O., Kissner, L., Peterson, Z., Song, D.: Remote data checking using provable data possession. ACM Trans. Inf. Syst. Secur. **14**(1) (jun 2011). https://doi.org/10.1145/1952982.1952994, https://doi.org/10.1145/1952982.1952994
7. Ateniese, G., Burns, R.C., Curtmola, R., Herring, J., Kissner, L., Peterson, Z.N.J., Song, D.X.: Provable data possession at untrusted stores. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007. pp. 598–609. ACM (2007). https://doi.org/10.1145/1315245.1315318, https://doi.org/10.1145/1315245.1315318
8. Ateniese, G., Di Pietro, R., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks. SecureComm '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1460877.1460889, https://doi.org/10.1145/1460877.1460889
9. Au, M.H., Tsang, P.P., Susilo, W., Mu, Y.: Dynamic universal accumulators for ddh groups and their application to attribute-based anonymous credential systems. In: Cryptographers' track at the RSA conference. pp. 295–308. Springer (2009)
10. Au, M.H., Wu, Q., Susilo, W., Mu, Y.: Compact e-cash from bounded accumulator. In: Cryptographers' Track at the RSA Conference. pp. 178–195. Springer (2007)
11. Baldimtsi, F., Camenisch, J., Dubovitskaya, M., Lysyanskaya, A., Reyzin, L., Samelin, K., Yakoubov, S.: Accumulators with applications to anonymity-preserving revocation. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 301–315. IEEE (2017)
12. Baldimtsi, F., Karantaidou, I., Raghuraman, S.: Oblivious accumulators. In: IACR International Conference on Public-Key Cryptography. pp. 99–131. Springer (2024)

13. Benaloh, J., De Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. In: Workshop on the Theory and Application of of Cryptographic Techniques. pp. 274–285. Springer (1993)
14. Benarroch, D., Campanelli, M., Fiore, D., Gurkan, K., Kolonelos, D.: Zero-knowledge proofs for set membership: Efficient, succinct, modular. Designs, Codes and Cryptography **91**(11), 3457–3525 (2023)
15. Bitcoin Developer Documentation: Operating Modes. Bitcoin.org (2025), `https://developer.bitcoin.org/devguide/operating_modes.html`, bitcoin Developer Guide; Accessed: 2025-05-12
16. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: [1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science. pp. 90–99 (1991). `https://doi.org/10.1109/SFCS.1991.185352`
17. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to iops and stateless blockchains. In: Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I 39. pp. 561–586. Springer (2019)
18. Boyle, E., Komargodski, I., Vafa, N.: Memory checking requires logarithmic overhead. In: Proceedings of the 56th Annual ACM Symposium on Theory of Computing. pp. 1712–1723 (2024)
19. Buldas, A., Laud, P., Lipmaa, H.: Accountable certificate management using undeniable attestations. In: Proceedings of the 7th ACM Conference on Computer and Communications Security. pp. 9–17 (2000)
20. Camacho, P., Hevia, A., Kiwi, M., Opazo, R.: Strong accumulators from collision-resistant hashing. In: Information Security: 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings 11. pp. 471–486. Springer (2008)
21. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18–22, 2002 Proceedings 22. pp. 61–76. Springer (2002)
22. Campanelli, M., Fiore, D., Greco, N., Kolonelos, D., Nizzardo, L.: Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In: Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part II 26. pp. 3–35. Springer (2020)
23. Campanelli, M., Fiore, D., Han, S., Kim, J., Kolonelos, D., Oh, H.: Succinct zero-knowledge batch proofs for set accumulators. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 455–469 (2022)
24. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16. pp. 55–72. Springer (2013)
25. Chase, M., Deshpande, A., Ghosh, E., Malvai, H.: Seemless: Secure end-to-end encrypted messaging with less trust. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (2019)
26. Chase, M., Healy, A., Lysyanskaya, A., Malkin, T., Reyzin, L.: Mercurial commitments with applications to zero-knowledge sets. In: Advances in Cryptology–EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings 24. pp. 422–439. Springer (2005)
27. Chen, B., Curtmola, R.: Robust dynamic provable data possession. In: 2012 32nd International Conference on Distributed Computing Systems Workshops. pp. 515–525 (2012). `https://doi.org/10.1109/ICDCSW.2012.57`
28. Clarke, D., Gassend, B., Suh, G.E., van Dijk, M., Devadas, S.: Offline authentication of untrusted storage (2002)
29. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2022)
30. Crosby, S.A., Wallach, D.S.: Super-efficient aggregating history-independent persistent authenticated dictionaries. In: Computer Security–ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings 14. pp. 671–688. Springer (2009)
31. Crosby, S.A., Wallach, D.S.: Authenticated Dictionaries: Real-World Costs and Trade-Offs. ACM Trans. Inf. Syst. Secur. (2011)

32. Curtmola, R., Khan, O., Burns, R., Ateniese, G.: Mr-pdp: Multiple-replica provable data possession. In: 2008 The 28th International Conference on Distributed Computing Systems. pp. 411–420 (2008). https://doi.org/10.1109/ICDCS.2008.68
33. Dodis, Y., Kiayias, A., Nicolosi, A., Shoup, V.: Anonymous identification in ad hoc groups. In: Advances in Cryptology-EUROCRYPT 2004: International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004. Proceedings 23. pp. 609–626. Springer (2004)
34. Dryja, T.: Utreexo: A dynamic hash-based accumulator optimized for the bitcoin utxo set. Cryptology ePrint Archive (2019)
35. Dwork, C., Naor, M., Rothblum, G.N., Vaikuntanathan, V.: How efficient can memory checking be? In: Theory of Cryptography: 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings 6. pp. 503–520. Springer (2009)
36. Erway, C.C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. ACM Trans. Inf. Syst. Secur. **17**(4) (apr 2015). https://doi.org/10.1145/2699909, https://doi.org/10.1145/2699909
37. Esiyok, I., Berrang, P., Cohn-Gordon, K., Künnemann, R.: Accountable javascript code delivery. arXiv preprint arXiv:2202.09795 (2022)
38. Etemad, M., Küpçü, A.: Database outsourcing with hierarchical authenticated data structures. In: International Conference on Information Security and Cryptology. pp. 381–399. Springer (2013)
39. Etemad, M., Küpçü, A.: Transparent, distributed, and replicated dynamic provable data possession. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) Applied Cryptography and Network Security. pp. 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
40. Fan, B., Andersen, D.G., Kaminsky, M.: {MemC3}: Compact and concurrent {MemCache} with dumber caching and smarter hashing. In: 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). pp. 371–384 (2013)
41. Fitzpatrick, B.: Distributed caching with memcached. Linux journal **2004**(124), 5 (2004)
42. Goodrich, M.T., Papamanthou, C., Tamassia, R., Triandopoulos, N.: Athos: Efficient authentication of outsourced file systems. In: International Conference on Information Security. pp. 80–96. Springer (2008)
43. Goodrich, M.T., Tamassia, R., Hasić, J.: An efficient dynamic and distributed cryptographic accumulator. In: International Conference on Information Security. pp. 372–388. Springer (2002)
44. Goodrich, M.T., Tamassia, R., Hasic, J.: An efficient dynamic and distributed rsa accumulator. arXiv preprint arXiv:0905.1307 (2009)
45. Goodrich, M.T., Tamassia, R., Triandopoulos, N.: Efficient authenticated data structures for graph connectivity and geometric search problems. Algorithmica **60**(3), 505–552 (2011). https://doi.org/10.1007/S00453-009-9355-7, https://doi.org/10.1007/s00453-009-9355-7
46. Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: Aggregating proofs for multiple vector commitments. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 2007–2023 (2020)
47. Heitzmann, A., Palazzi, B., Papamanthou, C., Tamassia, R.: Efficient integrity checking of untrusted network storage. In: Proceedings of the 4th ACM international workshop on Storage security and survivability. pp. 43–54 (2008)
48. Hu, Y., Hooshmand, K., Kalidhindi, H., Yang, S.J., Popa, R.A.: Merkle2: A low-latency transparency log system. In: 2021 IEEE Symposium on Security and Privacy (SP) (2021)
49. Jhanwar, M.P., Tiwari, P.R.: Trading accumulation size for witness size: a merkle tree based universal accumulator via subset differences. Cryptology ePrint Archive (2019)
50. Juels, A., Jr., B.S.K.: Pors: proofs of retrievability for large files. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007. pp. 584–597. ACM (2007). https://doi.org/10.1145/1315245.1315317, https://doi.org/10.1145/1315245.1315317
51. Knuth, D.E.: Sorting and searching. The Art of Computer Programmimg **422**, 559–563 (1973)
52. Laurie, B.: Certificate transparency: Public, verifiable, append-only logs. Queue **12**(8), 10–19 (Aug 2014). https://doi.org/10.1145/2668152.2668154, https://doi.org/10.1145/2668152.2668154
53. Lawlor, S., Lewi, K.: Deploying key transparency at whatsapp (2023), https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/

54. Len, J., Chase, M., Ghosh, E., Laine, K., Moreno, R.C.: {OPTIKS}: An optimized key transparency system. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 4355–4372 (2024)
55. Leung, D., Gilad, Y., Gorbunov, S., Reyzin, L., Zeldovich, N.: Aardvark: An asynchronous authenticated dictionary with applications to account-based cryptocurrencies. In: 31st USENIX Security Symposium (USENIX Security 22) (2022)
56. Li, J., Li, N., Xue, R.: Universal accumulators with efficient nonmembership proofs. In: International Conference on Applied Cryptography and Network Security. pp. 253–269. Springer (2007)
57. Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: Theory of Cryptography Conference. pp. 499–517. Springer (2010)
58. Malvai, H., Kokoris-Kogias, L., Sonnino, A., Ghosh, E., Oztürk, E., Lewi, K., Lawlor, S.F.: Parakeet: Practical Key Transparency for End-to-End Encrypted Messaging. IACR Cryptol. ePrint Arch. (2023)
59. Mathialagan, S.: Memory checking for parallel rams. In: Theory of Cryptography Conference. pp. 436–464. Springer (2023)
60. McMillion, B.: Key transparency (2023), https://www.ietf.org/archive/id/draft-mcmillion-key-transparency-00.html
61. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: {CONIKS}: Bringing key transparency to end users. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 383–398 (2015)
62. Merrill, K., Newman, Z., Torres-Arias, S., Sollins, K.R.: Speranza: Usable, privacy-friendly software signing. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 3388–3402 (2023)
63. Micali, S., Rabin, M., Kilian, J.: Zero-knowledge sets. In: 44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings. pp. 80–91. IEEE (2003)
64. Microsoft Corporation: Azure Confidential Ledger Overview: Enabling data integrity for data sources. Microsoft Corporation (2025), https://learn.microsoft.com/en-us/azure/confidential-ledger/overview#enabling-data-integrity-for-data-sources, accessed: 2025-05-12
65. Miller, A., Hicks, M., Katz, J., Shi, E.: Authenticated data structures, generically. SIGPLAN Not. (2014)
66. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system, https://bitcoin.org/bitcoin.pdf
67. Naor, M., Rothblum, G.N.: The complexity of online memory checking. Journal of the ACM (JACM) **56**(1), 1–46 (2009)
68. Nguyen, L.: Accumulators from bilinear pairings and applications. In: Topics in Cryptology–CT-RSA 2005: The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005. Proceedings. pp. 275–292. Springer (2005)
69. Nikitin, K., Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Gasser, L., Khoffi, I., Cappos, J., Ford, B.: {CHAINIAC}: Proactive {Software-Update} transparency via collectively signed skipchains and verified builds. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1271–1287 (2017)
70. Oprea, A., Reiter, M.K.: Integrity checking in cryptographic file systems with constant trusted storage. In: USENIX Security Symposium. pp. 183–198. Boston, MA; (2007)
71. Ozdemir, A., Laufer, E., Boneh, D.: Volatile and persistent memory for zksnarks via algebraic interactive proofs. Cryptology ePrint Archive (2024)
72. Ozdemir, A., Wahby, R., Whitehat, B., Boneh, D.: Scaling verifiable computation using efficient set accumulators. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2075–2092 (2020)
73. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: European Symposium on Algorithms. pp. 121–133. Springer (2001)
74. Papadopoulos, D., Papamanthou, C., Tamassia, R., Triandopoulos, N.: Practical authenticated pattern matching with optimal proof size. Proc. VLDB Endow. **8**(7), 750–761 (2015). https://doi.org/10.14778/2752939.2752944, http://www.vldb.org/pvldb/vol8/p750-papadopoulos.pdf
75. Papamanthou, C., Tamassia, R.: Optimal and parallel online memory checking. Cryptology ePrint Archive (2011)
76. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. p. 437–448. CCS '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1455770.1455826, https://doi.org/10.1145/1455770.1455826
77. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Cryptographic accumulators for authenticated hash tables. Cryptology ePrint Archive (2009)

78. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables based on cryptographic accumulators. Algorithmica **74**(2), 664–712 (2016). https://doi.org/10.1007/S00453-014-9968-3, https://doi.org/10.1007/s00453-014-9968-3
79. Reijsbergen, D., Maw, A., Yang, Z., Dinh, T.T.A., Zhou, J.: {TAP}: Transparent and {Privacy-Preserving} data services. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6489–6506 (2023)
80. Srinivasan, S., Chepurnoy, A., Papamanthou, C., Tomescu, A., Zhang, Y.: Hyperproofs: Aggregating and maintaining proofs in vector commitments. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3001–3018 (2022)
81. Srinivasan, S., Karantaidou, I., Baldimtsi, F., Papamanthou, C.: Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2719–2733 (2022)
82. Tamassia, R., Triandopoulos, N.: Efficient content authentication in peer-to-peer networks. In: Katz, J., Yung, M. (eds.) Applied Cryptography and Network Security, 5th International Conference, ACNS 2007, Zhuhai, China, June 5-8, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4521, pp. 354–372. Springer (2007). https://doi.org/10.1007/978-3-540-72738-5_23, https://doi.org/10.1007/978-3-540-72738-5_23
83. Tamassia, R., Triandopoulos, N.: Certification and authentication of data structures. AMW **619** (2010)
84. Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable subvector commitments for stateless cryptocurrencies. In: International Conference on Security and Cryptography for Networks. pp. 45–64. Springer (2020)
85. Tomescu, A., Bhupatiraju, V., Papadopoulos, D., Papamanthou, C., Triandopoulos, N., Devadas, S.: Transparency logs via append-only authenticated dictionaries. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (2019)
86. Tyagi, N., Fisch, B., Zitek, A., Bonneau, J., Tessaro, S.: Versa: Verifiable registries with efficient client audits from rsa authenticated dictionaries. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (2022)
87. Tzialla, I., Kothapalli, A., Parno, B., Setty, S.: Transparency dictionaries with succinct proofs of correct operation. Cryptology ePrint Archive (2021)
88. Van Dijk, M., Rhodes, J., Sarmenta, L.F., Devadas, S.: Offline untrusted storage with immediate detection of forking and replay attacks. In: Proceedings of the 2007 ACM workshop on Scalable Trusted Computing. pp. 41–48 (2007)
89. Wang, H., He, D., Fu, A., Li, Q., Wang, Q.: Provable data possession with outsourced data transfer. IEEE Transactions on Services Computing **14**(6), 1929–1939 (2021). https://doi.org/10.1109/TSC.2019.2892095
90. Wang, W., Lu, Y., Papamanthou, C., Zhang, F.: The locality of memory checking. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 1820–1834 (2023)
91. Wang, W., Ulichney, A., Papamanthou, C.: {BalanceProofs}: Maintainable vector commitments with fast aggregation. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 4409–4426 (2023)
92. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger (2014)
93. Wuille, P.: How do spv (simple payment verification) wallets learn about incoming transactions? Bitcoin Stack Exchange Q&A (2013), https://bitcoin.stackexchange.com/questions/11054/how-do-spv-simple-payment-verification-wallets-learn-about-incoming-transactio, answered Jun. 19, 2013; Accessed May 12, 2025
94. Yiu, M.L., Lin, Y., Mouratidis, K.: Efficient verification of shortest path search via authenticated hints. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). pp. 237–248 (2010). https://doi.org/10.1109/ICDE.2010.5447914
95. Yum, D.H., Seo, J.W., Lee, P.J.: Generalized combinatoric accumulator. IEICE transactions on information and systems **91**(5), 1489–1491 (2008)
96. Zhang, Y., Katz, J., Papamanthou, C.: An expressive (zero-knowledge) set accumulator. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 158–173. IEEE (2017)