Transparency Dictionaries with Succinct Proofs of Correct Operation

Ioanna Tzialla[◇] Abhiram Kothapalli[†] Bryan Parno[†] Srinath Setty[⋆]
[◇]New York University [†]Carnegie Mellon University [⋆]Microsoft Research

***Abstract*—This paper introduces Verdict, a transparency dictionary, where an untrusted service maintains a label-value map that clients can query and update (foundational infrastructure for end-to-end encryption and other applications). To prevent unauthorized modifications to the dictionary, for example, by a malicious or a compromised service provider, Verdict produces publicly-verifiable cryptographic proofs that it correctly executes both reads and authorized updates. A key advance over prior work is that Verdict produces efficiently-verifiable proofs while incurring modest proving overheads. Verdict accomplishes this by composing indexed Merkle trees (a new SNARK-friendly data structure) with Phalanx (a new SNARK that supports amortized constant-sized proofs and leverages particular workload characteristics to speed up the prover). Our experimental evaluation demonstrates that Verdict scales to dictionaries with millions of labels while imposing modest overheads on the service and clients.**

I. Introduction and Motivation

A *transparency dictionary* is a system in which an untrusted service maintains a list of label-value tuples that clients can query and update [22, 23, 33, 48, 63]. The dictionary supports efficient membership *and* non-membership queries, unlike transparency logs, which may only support membership queries. Transparency logs and dictionaries have been proposed as a foundation for PKIs, end-to-end encryption (e.g., for email or messaging), software updates, etc. For example, in key transparency [48], a dictionary maps unique identities to their public keys (e.g., RSA encryption keys).

Concretely, when used for key transparency, a transparency dictionary enables the following message exchange: To send a "top secret" document D to `bob@dom.org`, Alice picks a symmetric secret key k, encrypts D using k, and then encrypts k using the public key(s) associated with `bob@dom.org` in the transparency dictionary; she then sends both ciphertexts to `bob@dom.org` via an untrusted channel (e.g., the cloud). However, if the dictionary can return "rogue" public keys (i.e., public keys that Bob does not control), then even perfect cryptography cannot protect the secrecy of Alice's document. Non-membership queries are important for this scenario as well; otherwise, the service's dictionary might contain more than one tuple for the `bob@dom.org` label, allowing the service to show different tuples to different clients.[1]

Prior work [22, 23, 33, 48, 63] shows that an effective way to secure transparency dictionaries is to enforce that the dictionary maintained by the untrusted service remains *append-only*. Specifically, the domain of the dictionary (i.e., the set of labels) may only grow over time, and the associated values may only be updated according to an application-specific policy. In the context of key transparency, this means that the set of identities maintained by the dictionary never shrinks. Further, suppose that the first public key registered for `bob@dom.org` was legitimate. Then the application update policy might say that all subsequent updates must be authorized using one of the existing public keys.

A natural question here is: how can the untrusted service prove to its clients that its dictionary remains append-only? Ideally, we would like the service's proofs to be *publicly verifiable*, meaning that any client can verify that the entire service remains append-only. This ensures that any client of the system can obtain a strong security guarantee regardless of actions of other clients in the system. In the key transparency example, this means that when Alice verifies the service's proof, she knows that Bob's key has not been subverted, even if Bob is offline or fails to actively monitor updates to his own key. Hence, Alice can safely encrypt sensitive data using the public key the service returns for Bob. As a second criteria, the proofs must be *efficiently verifiable*: the cost of verifying proofs should be lower than reexecuting requests processed by the service, both asymptotically

Work began when Ioanna Tzialla and Abhriram Kothapalli were interns at Microsoft Research.

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23143
www.ndss-symposium.org

[1]There is an orthogonal but important requirement that the service should not be able to *equivocate* i.e., show different *versions* of its dictionary to different clients such that each version is well behaved. In general, this is impossible to prevent with a fully untrusted service [47]. However, in prior systems [22, 48, 63] and this work, if the service equivocates, it can be eventually detected by clients (§II, Appendix C).

and concretely (this also implies that proofs should be succinct). Satisfying the latter requirement makes it possible for any light client to download and verify proofs. As we discuss in §VII, prior work does not provide a dictionary abstraction [38], do not support public verifiability [33, 48], and/or impose significant proving/verification overhead [22, 23, 42, 63, 64].

In contrast, we describe Verdict, the first transparency dictionary with publicly and efficiently verifiable proofs of correct operation as well as modest proving overheads. Verdict achieves this by employing SNARKs [16, 30, 31], a cryptographic primitive that enables a *prover* to prove knowledge of a witness to an NP statement by producing a proof such that the size of the proof and the time to verify it are both *sub-linear* in the size of the statement.[2] Verdict employs SNARKs as follows. At the end of epoch i, the untrusted service publishes a succinct cryptographic commitment to C_i the current state of its dictionary.[3] It also produces a succinct SNARK proof demonstrating that the information in C_i is a superset of that in the dictionary committed to by C_{i-1}. Stated at this level, the use of SNARKs seems like an obvious solution.

Of course, the problem with such an "obvious" solution is well-known: the resource costs to produce SNARK proofs are, in general, excessive. However, we address this problem in our specific context, obtaining orders of magnitude speedups over a naïve application of SNARKs. First, we design a SNARK-friendly data structure (§III) that can be used to efficiently prove the necessary append-only invariants over the untrusted service's state. In more detail, Verdict uses this data structure to prove that the transparency dictionary's set of labels grows monotonically. Rather than directly prove (at significant expense) that each update to a label's value was applied correctly, Verdict also maintains, for each label, a provably append-only list of updates, which a client can apply locally to arrive at the current value.

Second, we design Phalanx, a new SNARK that leverages Verdict's particular workload characteristics to significantly drive down costs of the untrusted service and of the clients (§IV). Specifically, by leveraging the epoch-based nature of Verdict, Phalanx produces amortized constant-sized proofs and verification times. Furthermore, it leverages the data-parallel nature of the statement proven in each round to substantially reduce proof-generation costs. Phalanx may be of independent interest to other epoch-based services.

We implement Verdict in Rust as a generic library for constructing transparency dictionaries (§V). To demonstrate the concrete utility of Verdict, we apply it to our running example of key transparency, creating Keypal, a service for translating user identities to public keys without trusting the hosting service.

We evaluate Verdict (§VI) and compare it with three baselines that provide the similar security properties: AAD [63], and two variants of Merkle-Patricia trees. Unlike AAD, Verdict incurs low overheads even for large dictionaries with millions of labels, and unlike the Merkle-Patricia baseline, Verdict produces efficiently-verifiable proofs of correct operation. We also find that our workload-specific SNARK optimizations improve proving costs by an order of magnitude. Together, Verdict achieves about 4 updates/sec/CPU-core and about 2 inserts/sec/CPU-core, with a per-epoch (amortized) proof size of 651 bytes and a verification time of about 3 ms (for a dictionary with 2^{20} label-value tuples); Verdict can achieve about 18–22 updates/per/sec/CPU-core and 9–11 inserts/sec/CPU-core, with a per-epoch proof size of 290 bytes and a verification time of 161μs for the same dictionary size at the cost of deferred guarantees (§VI). This represents over an order of magnitude improvement over prior state-of-the-art, general proof systems for stateful services [40, 56, 57, 59] and over three orders of magnitude improvement over AAD [63].

Verdict's principal limitation is that the service still incurs significant CPU costs to produce proofs. Less fundamentally, although the proof-generation process is highly parallelizable, our current implementation of Verdict does not leverage multiple CPUs. Nevertheless, we believe, Verdict makes transparency dictionaries with efficiently-verifiable proofs affordable.

In summary, Verdict contributes:

1) A transparency dictionary that scales well asymptotically and concretely to large dictionaries.
2) A SNARK-friendly accumulator with $O(\log n)$ proofs of membership and non-membership.
3) A SNARK that provides amortized constant-sized proofs and verification times and that leverages particular workload characteristics (e.g., computations with repeated substructure) to speed up proof generation.

II. The Verdict Transparency Dictionary

A. Problem Statement and Definitions

Our goal is to build Verdict, a service that exposes a dictionary abstraction and can cryptographically prove to its clients the correctness of every request it executes. Specifically, the service's state is a label-value map that clients can query and update; i.e., clients can insert a new label-value pair, update the value associated with

[2] SNARKs provide an additional property called zero-knowledge, where the verifier learns nothing about the prover's witness beyond what is implied by the proven statement. Verdict's focus is on succinct verification property of SNARKs, but with modern SNARKs (including the one that Verdict employs), achieving zero-knowledge is "free" in terms of additional costs to the prover, the verifier, and proof sizes.

[3] Informally speaking, a cryptographic commitment scheme enables a *sender* to commit itself to a value by sending a short commitment and then later reveal the value such that the commitment scheme is binding (i.e., the sender cannot reveal a value different from what it originally committed), and hiding (i.e., a commitment does not reveal anything about the committed value).

an existing label, and look up the value associated with an existing label. By correct operation, we mean that the set of labels in the dictionary grows monotonically and that the value associated with a label is only updated according to a pre-specified application-specific policy. We refer to this primitive as a *transparency dictionary*. As a concrete application of Verdict, we construct Keypal (§V), a public key directory that enables clients to register their public keys under an identifier such that the keys can be retrieved by other clients who can be assured that the public keys they retrieve are legitimate.

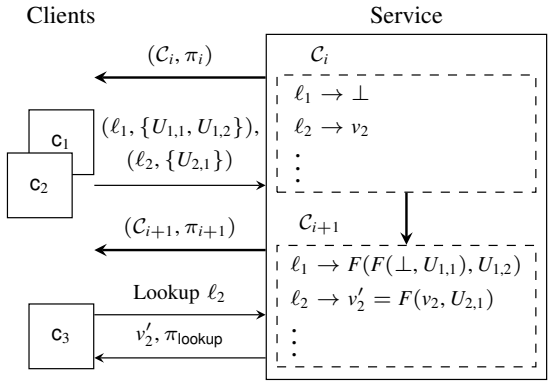

Figure 1: Example showing how Verdict's service updates its state in epoch i for some application-specific policy F. At the end of epoch i, the service broadcasts a new commitment $\mathcal{C}_{i+1}$ and a proof π_{i+1} after incorporating requests from clients c_1 and c_2. Later, client c_3 can query the server to retrieve the latest value (v_2') associated with label ℓ_2.

a) Threat Model: We assume that the service and any subset of clients can misbehave arbitrarily. For example, the service can misbehave when processing requests, or show different views of its state to different clients. Furthermore, we assume that the network can arbitrarily duplicate, drop, or reorder messages. However, we assume that the untrusted service and clients cannot break cryptographic hardness assumptions. We do not aim to protect against denial of service (an honest service can use standard techniques, e.g., rate-limiting or proof-of-work, to defend itself) nor to ensure liveness (although if the service is honest and the network is reliable, then requests from correct clients will be executed). Finally, we do not aim to *prevent* all misbehavior from the untrusted service. Indeed, there are fundamental limits: it is impossible to prevent an untrusted service from equivocating [47]. However, as in prior work [22, 48], Verdict's clients can eventually detect such equivocation through client-side mechanisms (e.g., gossip), or by requiring the service to disseminate its commitments and proofs through a public blockchain (Appendix C).

More formally, a transparency dictionary is a tuple of algorithms $(\mathsf{Setup}, \mathsf{ApplyUpdates}, \mathsf{VerifyUpdates}, \mathsf{Lookup}, \mathsf{VerifyLookup})$, with the following semantics. We assume that dictionary $D_0 = \perp$ and its commitment $\mathcal{C}_0$ is well-known. Below, let F denote an application-specific predicate that takes as input a label's old value, a request, and outputs a new value for the label. Let λ denote the security parameter.

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F)$: Returns public parameters pp used to produce and verify proofs.
- $(D_{t+1}, \mathcal{C}_{t+1}, \pi_{t+1}) \leftarrow \mathsf{ApplyUpdates}(\mathsf{pp}, D_t, \mathcal{C}_t, U)$: Takes as input a dictionary D_t, commitment $\mathcal{C}_t$, and a sequence of insert/update requests U. Outputs a dictionary D_{t+1}, a commitment $\mathcal{C}_{t+1}$ to D_{t+1}, and a proof of valid update π_{t+1}.
- $\{0,1\} \leftarrow \mathsf{VerifyUpdates}(\mathsf{pp}, \mathcal{C}_t, \mathcal{C}_{t+1}, \pi_{t+1})$: Takes as input a pair of commitments $(\mathcal{C}_t, \mathcal{C}_{t+1})$, and a proof of valid update π_{t+1}. Outputs 1 if: **(1)** the set of labels in the dictionary committed to by $\mathcal{C}_{t+1}$ is a superset of the set of labels in the dictionary committed to by $\mathcal{C}_t$; and **(2)** for each label in the dictionary committed to by $\mathcal{C}_{t+1}$, the associated value equals the value associated with the label in $\mathcal{C}_t$, *or* π_{t+1} proves the knowledge of a sequence of requests $(U_0, \ldots, U_{\ell-1})$ such that $F(v_i, U_i) = v_{i+1}$ for $0 \leq i < \ell$, and where v_0 is the value associated with the label in $\mathcal{C}_t$ (or $\perp$ if the label did not exist in $\mathcal{C}_t$) and v_ℓ is the value associated with the label in $\mathcal{C}_{t+1}$.
- $(v, \pi_{\mathsf{lookup}}) \leftarrow \mathsf{Lookup}(\mathsf{pp}, D_t, \mathcal{C}_t, \mathsf{label})$: Takes as input a dictionary D_t, commitment $\mathcal{C}_t$, and label label. Outputs a value v, and a proof of valid lookup π_{lookup}.
- $\{0,1\} \leftarrow \mathsf{VerifyLookup}(\mathsf{pp}, \mathcal{C}, \mathsf{label}, v, \pi_{\mathsf{lookup}})$: Takes as input a commitment $\mathcal{C}$, a label label, a value v, and a proof π_{lookup}. Outputs 1 if v is the value associated with label in $\mathcal{C}$.

As in prior transparency logs [22, 33, 48, 63], the service in a transparency dictionary operates in a sequence of (fixed) time epochs. In each epoch, the service collects requests submitted by clients, uses the ApplyUpdates procedure to update its internal state and produce a new commitment and a proof of valid update, which it publishes to clients. Clients collect the sequence of commitments and the associated proofs published by the service at each time epoch and use the VerifyUpdates to check if the service correctly updated its state. Similarly, for lookup requests, the service uses the Lookup procedure to produce a response along with a proof, which the clients can verify by applying the VerifyLookup procedure to the proof and the latest service commitment they have received.

We formally define the properties of a transparency dictionary in Appendix A but we summarize them here.

- **Correctness.** If the service is honest, clients do not reject responses produced by the service.
- **Update soundness.** The set of labels in a dictionary grows monotonically and the values are updated only according to an application-specific policy.
- **Lookup soundness.** The service cannot return an incorrect value for any label included in a commitment.
- **Fork consistency.** If the service equivocates at some

point in time by presenting different sequences of commitments to different sets of clients, it cannot undetectably merge their views.

Additionally, we desire the following properties:

- **Public verifiability.** Any client can check proofs produced by the service to verify that that the set of labels in the dictionary maintained by the service grows monotonically and that values are updated only according to an application-specific policy.
- **Efficient verifiability.** Compared with re-executing every request processed by the service, the cost to verify a proof is more resource efficient (in terms of CPU, network bandwidth, storage, etc.), both asymptotically and concretely. This implies that proofs produced by the service are succinct.

B. Verdict: Architectural Overview

At its core, Verdict utilizes cryptographic SNARKs [15, 30, 31] to prove that the untrusted service correctly maintains its dictionary abstraction. Generically, a SNARK allows a prover to demonstrate that an arbitrary polynomial relation $\mathcal{R}(w,x)$ holds, where w is a (possibly secret) witness, and x is a public input. The prover produces a proof π such that the size of π and the time to verify it are both sub-linear in the size of the circuit that computes $\mathcal{R}$.

At the end of epoch $t+1$, Verdict uses SNARKs to prove the following invariants about the dictionaries committed to by $\mathcal{C}_t$ and $\mathcal{C}_{t+1}$ (the service's commitments at the end of epochs t and $t+1$). First, the set of labels in the dictionary committed to by $\mathcal{C}_{t+1}$ is a superset of the set of the labels in the dictionary committed to by $\mathcal{C}_t$. Second, for each (ℓ, v) in the dictionary committed to by $\mathcal{C}_t$, one of the following holds: either (ℓ, v) exists in the dictionary committed to by $\mathcal{C}_{t+1}$; or there is a sequence of operations (valid according to an application-specific policy) such that v' is the result of applying those operations to v and (ℓ, v') is in $\mathcal{C}_{t+1}$.

Implemented in a naïve manner, the approach above would be prohibitively expensive. First, for each value updated, the service must prove, using an expensive SNARK, that the operation applied on the value is legitimate and that it faithfully executed that operation. For example, in the context of a key directory, when updating a public key associated with an identity, the service must prove that it knows of a valid digitally-signed request in the sense that the signature can be verified using one of the non-revoked keys associated with the identity. Verifying such cryptographic operations via a SNARK is quite costly [26, 40, 51]. Even worse, as described above, to prove that its $O(N)$ dictionary is updated correctly, the size of the circuit that the service must prove using a SNARK is $\Omega(N)$. In other words, even when the number of values updated in a given epoch is far less than N, the service's cost for each epoch is still $\Omega(N)$. Finally, even non-cryptographic computations are costly to prove using SNARKs.

We now discuss how Verdict addresses these issues. Figure 2 compares Verdict's asymptotics with prior work.

a) Validating Value Updates via Hashchains: To reduce the service's overhead, instead of directly proving that the service only processes operations that are valid according to an application-specific policy or that the server processes those operations faithfully, Verdict employs a simpler and cheaper alternative. In Verdict's transparency dictionary, the value associated with a label is an append-only hashchain of operations, where nodes store raw operations requested on the label, as well as the cryptographic hash of the previous node in the chain. For example, in the context of key transparency, a hashchain records two types of operations: **(1)** adding a new key; and **(2)** revoking an existing key, and each operation is digitally signed by the client requesting the update.

A hashchain is valid if each node includes a correct hash of the previous node, and if the result of applying each operation complies with the application's-specific policy defined by F. For example, in key transparency, each computed value v_i would be a set of public keys. F would allow any key to be added if it is the first operation (i.e., $v_i = \perp$), and it would accept subsequent operations if they are digitally signed by an unrevoked key previously added in the hashchain.

When a client retrieves a hashchain associated with a label, it can quickly apply operations recorded on the hashchain to construct the current value associated with the label, checking the validity of the cryptographic hashes and compliance with F along the way. This design supports a richer class of application-specific policies without requiring the service to prove the validity of those policies using SNARKs.

b) Succinct Commitments and Proofs via Indexed Merkle Trees: We now discuss how Verdict commits to a dictionary that maps labels to hashchains such that it can efficiently produce: (i) lookup proofs, and (ii) update proofs. In a nutshell, Verdict's service stores its dictionary in a commodity storage service. In addition, it creates a "derived" dictionary that stores a map from from the cryptographic hash of a label to the cryptographic hash of the last node in the hashchain associated with the label. In particular, for each $(\mathsf{label}, \mathsf{c})$ tuple in the service's state, the derived dictionary contains $(H(\mathsf{label}), h)$, where H is a cryptographic hash function and h is the cryptographic hash of the last node in c.

Observe that to commit to its original dictionary the service only needs to commit to its derived dictionary. This is because each (label,value) tuple in the derived dictionary is cryptographically bound (via the collision-resistance of H) to a unique tuple in the original dictionary. More crucially, this choice ensures that the labels and values that Verdict needs to commit are constant-sized and in particular short (e.g., 32 bytes

	space (life-time)	updates (per-epoch) prover	proof size	verifier	lookups (per-op) prover/proof size/verifier	assumption
AAD [63]⋆	$\lambda \cdot N$	$\beta \cdot \log^3 n$	$\log n$	$\log n$	$\log^2 n$	q-type
SEEMless [22]	N	$\beta \cdot \log N$	$\beta \cdot \log N$	$\beta \cdot \log N$	$\log N$	CRHF
Verdict	N	$\beta \cdot \log n$	$\log(\beta \cdot \log n)/k$	$\log(\beta \cdot \log n)/k$	$\log n$	SXDH
Verdict (lazy)	N	$\beta \cdot \log n$	$\log(\beta \cdot \log n)/k$	$\beta \cdot \log n/k$	$\log n$	DLOG

⋆ Requires a trusted setup

Figure 2: Asymptotic costs for updates and lookups under Verdict and its baselines. β is the number of updates per epoch, n is the maximum number of labels in the dictionary, ℓ is the maximum number of operations associated with a label. We let $N = n \cdot \ell$. Lookup responses include an additive cost of $\Omega(\ell)$. Verdict variants use indexed Merkle trees (§III) with different variants of Phalanx (§IV). k denotes the number of epochs over which costs are amortized (see IV-C for details).

each when using SHA-256 as the hash function).

To commit to a derived dictionary with ℓ-sized labels and values, we design a cryptographic accumulator using "textbook" Merkle trees,[4] which we refer to as *indexed Merkle trees*. We provide details in Section III, but a distinguishing aspect of indexed Merkle trees is that they support *both* efficient membership proofs (i.e., proofs for statements of the form $(\ell, v) \in \mathcal{C}$, where (ℓ, v) is a label-value pair and $\mathcal{C}$ is a commitment to a dictionary) and non-membership proofs (i.e., proofs for statements of the form $(\ell, v) \notin \mathcal{C}$). In particular, for a dictionary of size N, it produces $O_\lambda(1)$-sized commitments (e.g., 32 bytes when using SHA-256), and $O_\lambda(\log N)$-sized proofs of membership and non-membership.

Thus, at the end of each epoch, Verdict's commitment is of size $O_\lambda(1)$. Furthermore, for a lookup request issued by a client, π_{lookup} is either a proof of membership (if the hash of the requested label exists in the derived dictionary) or a proof of non-membership (otherwise). Note that the indexed Merkle tree itself is stored in a commodity storage service, so producing a lookup proof does not require the use of SNARKs, so the throughput of the service for lookup operations is bound purely by the throughput of the underlying storage service to retrieve proofs of membership (or non-membership).

Another crucial aspect of indexed Merkle trees is that, by leveraging proofs of non-membership, they provide efficient, $O_\lambda(\log N)$, proofs of insertion and update.

c) Condensing Merkle Proofs via SNARKs: As described above, Verdict uses Merkle proofs to reduce the cost of proving/verifying dictionary updates from $\Omega(N)$ to $\log(N)$. However, for an epoch that consists of β insert/update operations, the proof length and verification time are both $O_\lambda(\beta \cdot \log N)$. Cost wise, this is equivalent to requiring the service to broadcast all updates it processes in order to prove that it maintains the desired append-only properties. Verdict reduces both proof sizes and verification times by employing SNARKs.

Specifically, for $\beta > 0$ insert/update operations, the service processes each operation sequentially, and for each operation, the service produces a new commitment and an $O_\lambda(\log N)$-sized proof. The untrusted service however does not publish these intermediate commitments nor the $O_\lambda(\log N)$-sized proofs. Instead, it employs SNARKs to prove that it *knows* indexed Merkle tree proofs that a verifier would accept, thereby achieving a proof of valid updates whose size and verification time are both sub-linear in $\beta \cdot \log N$ (Figure 2).

As discussed in Section III, our indexed Merkle trees are carefully designed to be SNARK-friendly.

d) Accelerating SNARKs in Verdict's Context: Thus far, Verdict could be instantiated with any generic SNARK system. However, despite the proof-friendly nature of our indexed Merkle trees, the costs would still remain high. Hence, we introduce Phalanx, a new SNARK that leverages particular workload characteristics to provide amortized constant-sized proofs and to reduce proof generation costs (§IV provides details).

III. Indexed Merkle Trees

This section describes the SNARK-friendly accumulator that Verdict employs to maintain state.

A. Requirements and Possible Instantiations

Let λ denote the security parameter, and let $\mathtt{negl}(\lambda)$ denote a negligible function in λ. Let "PPT algorithms" refer to probabilistic polynomial time algorithms.

Recall that a cryptographic accumulator [14] enables a *prover* to commit to a collection D (e.g., a dictionary with a set of label-value pairs) by sending a succinct commitment $\mathcal{C}$ to a *verifier*. Such digests are *binding*, meaning that it is computationally infeasible to identify a different collection of items with the same digest. In addition, cryptographic accumulators support succinct *proofs of membership*: for any item (i.e., a label-value tuple) $x \in D$, the prover can produce a succinct proof π such that an honest verifier accepts π, and for any $x \notin D$ and any purported proof π produced by a PPT algorithm, $\Pr[\text{the verifier accepts } \pi] \leq \mathtt{negl}(\lambda)$.

A classic example of an accumulator is a Merkle tree, where the root of the Merkle tree commits to items stored at leaf nodes and provides $O_\lambda(\log n)$-sized proofs of membership, where n is the size of the collection.

As we discuss in Section II-B, for Verdict, we require accumulators with succinct proofs for membership, inserts, and updates. Specifically, suppose that the prover commits to its collection D with a commitment $\mathcal{C}$:

[4] A cryptographic accumulator is a commitment scheme with support for proving membership of an element against a commitment (§III).

- *Proof of insertion.* For any $x \notin D$, the prover can produce a new commitment $\mathcal{C}'$ and prove that $x \notin D$ and $\mathcal{C}'$ commits to $D \cup \{x\}$ with a succinct proof π such that a verifier accepts π. If $x \in D$ or if $\mathcal{C}'$ does not commit to $D \cup \{x\}$, then for any purported proof π produced by a PPT algorithm, $\Pr[\text{the verifier accepts } \pi] \leq \texttt{negl}(\lambda)$.
- *Proof of update.* For any $x \in D$, the prover can produce a new commitment $\mathcal{C}'$ and prove that $\mathcal{C}'$ commits to $D \cup \{F(x,x')\} - \{x\}$ with a succinct proof π such that a verifier accepts π, where F is any function. If $\mathcal{C}'$ does not commit to $D \cup \{F(x,x')\} - \{x\}$, then for any purported proof π produced by a PPT algorithm, $\Pr[\text{the verifier accepts } \pi] \leq \texttt{negl}(\lambda)$.

Note that an important building block for a proof of insertion is a a *proof of non-membership*: for any $x \notin D$, the prover can prove that $x \notin D$ by producing a succinct proof π such that the verifier accepts π, and for any $x \in D$ and any purported proof π produced by a PPT algorithm, $\Pr[\text{the verifier accepts } \pi] \leq \texttt{negl}(\lambda)$.

In addition to the above properties, we desire a cryptographic accumulator where these proofs are *SNARK-friendly*; i.e., where the proof verification algorithm can be efficiently encoded in the input language of a modern SNARK. This can be thought of as an arithmetic circuit over a large field, although in practice most modern SNARKs use a generalization known as R1CS. The efficiency of this encoding matters, since, as described in Section II-B, the Verdict service relies on SNARKs to condense many insert/update proofs into a single succinct proof of correct operation. To accomplish this, the service first updates the cryptographic accumulator to produce a sequence of insert/update proofs, one for each operation. The service then proves that it *knows* a sequence of valid proof of insert/updates using a SNARK.

We examine existing accumulators in the literature and folklore. Figure 8 summarizes our findings, where n is the number of items in a collection. Briefly:

- Merkle trees [17, 49] do not support succinct proofs of non-membership nor insertion.
- RSA accumulators [20] support $O_\lambda(1)$-sized membership and insert/update proofs, but they impose $O_\lambda(n)$ costs on the prover to produce them; this is expensive, both asymptotically and concretely. Also, they require a trusted setup as well as big number arithmetic that is inefficient to encode as an arithmetic circuit.
- Merkle-Patricia trees (or more generally, Sparse Merkle trees) (e.g., [22, 25, 37, 50]) support $\Theta(\log n)$-sized proofs of membership, non-membership, insertion, and updates. However, by design, paths in these trees are of variable length, so devising arithmetic circuits to verify Merkle proofs introduces significant complexity. Of course, one can use fixed-depth trees (e.g., depth-256) to make them SNARK-friendly, but this incurs over an order of magnitude higher costs at our collection sizes (2^{20}–2^{30} items).
- Like Merkle-Patricia trees, Merkle-AVL trees are not SNARK-friendly as they require rebalancing upon insertion of new nodes.

B. Indexed Merkle Trees

We now describe *indexed Merkle trees*, our SNARK-friendly variant of "textbook" Merkle trees, with support for efficient non-membership and insertion proofs.

An indexed Merkle tree is a standard Merkle tree in the following way: each item in a collection is stored at a leaf node in a Merkle tree. This ensures that indexed Merkle trees support $O_\lambda(1)$-sized commitments, $O_\lambda(\log n)$-sized proofs of membership, and $O_\lambda(\log n)$-sized proofs of update. To support $O_\lambda(\log n)$-sized proofs of non-membership and proofs of insertion, we encode additional metadata at each leaf node and maintain an invariant upon insertions and updates. We now elaborate.

Suppose that a collection is a set of label-value tuples, where each label and each value is of a fixed size w. This is the case for Verdict's derived dictionary (§II-B).[5] WLOG, we assume that labels can be sorted (e.g., with a bitwise ordering of labels). Unlike a textbook Merkle tree, a leaf node in an indexed Merkle tree is of the form: $\langle \mathsf{active}, \mathsf{label}, \mathsf{value}, \mathsf{next} \rangle$, where active is a bit indicating whether the leaf node holds a valid tuple. If active is 1, then label and value are respectively the label and its associated value stored at the leaf node, and next is a label in the tree that is larger than label.

a) Verifiable Initialization: We now discuss how the prover can initialize an empty indexed Merkle tree and how the verifier can efficiently compute a commitment to such an empty tree. Suppose that the indexed Merkle tree maintained by the prover has a capacity of $n \geq 2$ (i.e., it has n leaf nodes), where each leaf node stores the same tuple $\langle 0, 0^\mathsf{w}, 0^\mathsf{w}, 0^\mathsf{w} \rangle$ (we later discuss how the prover can double the capacity of a Merkle tree and how the verifier can efficiently verify that). Any verifier—without help from the prover—can compute the root of such a tree with $O(\log n)$ hash computations. WLOG, Verdict designates two labels as reserved: 0^w and 1^w. These denote respectively the lowest and the highest values of labels in the system. Furthermore, the prover picks a designated leaf node (WLOG, the left-most node in the initial Merkle tree) in the initial indexed Merkle tree and updates it to hold the following tuple: $\langle 1, 0^\mathsf{w}, 0^\mathsf{w}, 1^\mathsf{w} \rangle$. The verifier—without help from the prover—can compute the root of the updated Merkle tree in $O_\lambda(\log n)$ time.

[5] For other contexts, one can employ the same technique as in Verdict to derive a new collection with fixed-sized labels and values.

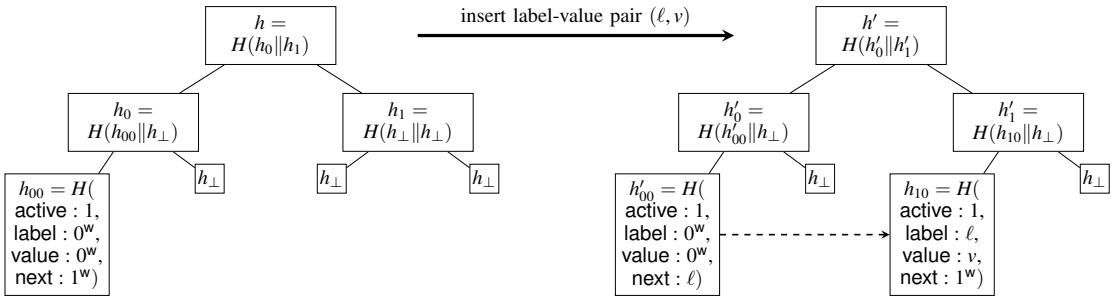

Figure 3: Indexed Merkle trees in action. We depict state changes from inserting a new label-value pair (ℓ, v) to an empty indexed Merkle tree with capacity 4. The choice of which non-active leaf node holds (ℓ, v) is arbitrary (see the text for details). $h_\perp = H(\mathsf{active} : 0, \mathsf{label} : 0^w, \mathsf{value} : 0^w, \mathsf{next} : 1^w)$. Dotted arrow indicates a leaf's "pointer" to the next active label. In the updated tree, to prove non-membership of ℓ' where $\ell' < \ell$, the prover presents the leaf node h'_{00} and its proof of membership.

b) Invariant: A core invariant that an indexed Merkle tree maintains is that for any "active" leaf node $N = \langle 1, \ell, v, n \rangle$, $N.n$ is either: (i) 1^w, or (ii) there exists some leaf node in the tree with label $N.n$ and there are no active leaf nodes in the tree with labels in the range $(N.\ell, N.n)$. Observe that this invariant holds for the indexed Merkle tree computed at the end of the initialization step. Below, we discuss how the invariant holds despite inserting new label-value pairs or updates to existing label-value pairs.

c) Proof of Non-Membership: A core building block for proof of insertion is a proof of non-membership. To prove the absence of a particular label ℓ in an indexed Merkle tree with commitment $\mathcal{C}$, the prover furnishes a proof of membership for a unique leaf node with contents $\langle 1, \mathsf{low}, v, \mathsf{high} \rangle$ such that $\mathsf{low} < \ell < \mathsf{high}$. This proof can be verified using commitment $\mathcal{C}$. It is easy to see that this constitutes a proof of non-membership given the invariant stated above.

d) Maintaining the Invariant: In Verdict, there are only two types of operations.

(1) Updates. For a leaf node $N = \langle 1, \mathsf{label}, \mathsf{value}, \mathsf{next} \rangle$, updating $N.\mathsf{value}$. This trivially upholds the desired invariant. The proof of update is a proof of membership of the old leaf node against the old commitment. Verification involves verifying the proof of membership and then locally computing an updated commitment using the updated leaf node.

(2) Inserts. As shown in Figure 3, to insert a new label-value pair (ℓ, v) into an indexed Merkle tree, the proof of insertion is produced as follows.

- The prover identifies an "inactive" leaf node;[6] i.e., a leaf node of the form $N = \langle 0, 0^w, 0^w, 0^w \rangle$. If this fails, invoke the capacity doubling procedure described below and then retry this step.
- The prover produces a proof of non-membership of ℓ in the indexed Merkle tree, i.e., a unique leaf node $N^\star = \langle 1, \mathsf{low}, val, \mathsf{high} \rangle$ such that $\mathsf{low} < \ell < \mathsf{high}$.
- The prover updates $N^\star$ to hold the value $\langle 1, \mathsf{low}, val, \ell \rangle$, and produces a proof of update.
- The prover updates the previously inactive node N to hold $\langle 1, \ell, v, \mathsf{high} \rangle$, and produces a proof of update.

e) Supporting Dynamic Capacity: The number of leaf nodes of an indexed Merkle tree can be doubled at any time such that a verifier can verify that the new Merkle tree contains all of the data from the original tree with $O_\lambda(\log n)$ computation. Specifically, given the root r of an existing 2^i-sized indexed Merkle tree, the prover and the verifier can compute the (unique) root of the 2^{i+1}-sized Merkle tree as: hash(r, r'), where r' is the root of the Merkle tree where all leaf nodes have the same default value of $\langle 0, 0^w, 0^w, 0^w \rangle$. Note that r' can be computed by the verifier using $O(i)$ hashes. This does not require any precomputation or amortization as the cost is comparable to the cost of verifying a membership proof, which is logarithmic in the size of the collection.

IV. Reducing Costs of SNARKs with Phalanx

In a Verdict epoch with β update operations, the untrusted service produces β indexed Merkle update proofs. A straightforward approach to prove the correctness of these indexed Merkle proofs with a SNARK is to use an arithmetic circuit (say of size C) that verifies one update proof, replicate it β times, and then employ a generic SNARK (e.g., Spartan [56], Xiphos [59]) on the combined circuit. The result would be a prover that runs in time $O_\lambda(C \cdot \beta)$. The per-epoch proof sizes and verification times are $O_\lambda(\sqrt{C \cdot \beta})$ when using Spartan and $O_\lambda(\log(C \cdot \beta))$ when using Xiphos.

To make Verdict concretely efficient, we develop Phalanx, a SNARK that substantially reduces resource costs imposed by prior SNARKs. Specifically, Phalanx produces (amortized) constant proof sizes and verification times. Additionally, Phalanx does not require a

[6] Verdict's service maintains a separate index of inactive leaf nodes; the choice of which inactive leaf node to use is arbitrary. In particular, note that leaf nodes in an indexed Merkle tree are *not* sorted, so if a new label falls between two existing labels in the dictionary, the new label can be inserted at any "inactive" leaf node in the tree.

trusted setup. Given these, Phalanx is of independent interest (e.g., to other epoch-based services [40, 51]).

A. Overview of Phalanx

Phalanx targets epoch-based services where the prover in each epoch proves the satisfiability of the same N-sized circuit (with different witness values). In Verdict, the circuit verifies β indexed Merkle proofs.

In more detail, Phalanx's prover and verifier maintain a $O_\lambda(1)$-sized *running instance*, which collectively represents the circuit satisfiability instances of *all* prior epochs (the verifier and the prover initialize the running instance to the first epoch's instance). At the end of each epoch, the prover produces an $O_\lambda(1)$-sized proof that enables the verifier to combine the circuit satisfiability instance of that epoch with the running instance; the verifier incurs $O_\lambda(1)$ computation to update its running instance. At the end of each epoch, the prover also produces an $O_\lambda(\log N)$-sized proof to prove the satisfiability of the running instance. Verifying the logarithmic-sized proof (in addition to verifying constant-sized proofs for all prior epochs) verifies that the circuit satisfiability instances from all prior epochs are satisfiable.

In Verdict's context, a client must verify constant-sized proofs every epoch, but it need not verify the logarithmic-sized proof in every epoch. Specifically, a client only needs to verify the logarithmic-sized proof of the most recent epoch before issuing lookup operations. Furthermore, in Verdict, we observe that the running instance is data-parallel (as the circuit verifies β indexed Merkle proofs); Phalanx leverages this to substantially speedup proof-generation costs.

B. Preliminaries

a) SNARKs: Recall that a SNARK is a cryptographic primitive that enables a prover to demonstrate its knowledge of a witness to an NP statement (e.g., a circuit satisfiability instance) with a proof that can be verified in time sub-linear (ideally, polylogarithmic) in the time to check the NP witness; this also implies that the proof is sub-linear in the size of the NP witness. For our purpose, SNARKs enable proving the correct execution of (stateful) computations, since those executions can be represented with NP statements. See Appendix B for a formal definition of SNARKs.

b) R1CS: Rank-1 constraint satisfiability (R1CS) is an NP-complete language that generalizes arithmetic circuits [13, 30, 58]. R1CS is a popular target for toolchains that compile programs in high-level languages [12, 18, 35, 40, 52, 57, 60, 65]. In more detail, let $\mathbb{F}$ denote a finite field (e.g., the set $\{0, 1, \ldots, p-1\}$ for a large prime p, with addition and multiplication operations). An R1CS instance is a tuple $((\mathbb{F}, A, B, C, m, n, \ell), \mathsf{x})$, where $\mathsf{x} \in \mathbb{F}^\ell$ is the public input and output of the instance, $A, B, C \in \mathbb{F}^{m \times m}$, $m \geq |\mathsf{x}| + 1$, and there are at most $n = \Omega(m)$ non-zero entries in each matrix. A witness $W \in \mathbb{F}^{m-\ell-1}$ satisfies an R1CS instance $((\mathbb{F}, A, B, C, m, n, \ell), \mathsf{x})$ if $(A \cdot Z) \circ (B \cdot Z) = C \cdot Z$, where $Z = (W, \mathsf{x}, 1)$. In a nutshell, the matrices encode the structure of an arithmetic circuit, where gates can compute an arbitrary bilinear operation over Z. We refer to n as the size of the R1CS instance.

c) SIMD R1CS: To capture data-parallel (or SIMD) computations, we introduce a natural extension of R1CS that we refer to as SIMD R1CS. Informally, SIMD R1CS considers the same circuit (represented with matrices A, B, C) over β witness vectors $\{w_1, \ldots, w_\beta\}$ and the corresponding input/output vectors $\{x_1, \ldots, x_\beta\}$, representing β data-parallel units. WLOG, we assume that β, m, and n are powers of 2, and ℓ is even.

More formally, a SIMD R1CS instance is a tuple $\phi = ((\mathbb{F}, A, B, C, m, n, \ell, \beta), x)$, where $A, B, C \in \mathbb{F}^{m \times m}$ each of which has at most $n = \Omega(m)$ non-zero values, public input/output $x \in \mathbb{F}^\ell$. A witness (W, x), where $W \in \mathbb{F}^{(m-\ell-1) \times \beta}$ (each column of W is a purported witness for a separate data-parallel unit) and $\mathsf{x} \in \mathbb{F}^\ell \times \beta$ (each column of x is a purported input/output for a separate data-parallel unit), satisfies ϕ if $(A \cdot Z) \circ (B \cdot Z) = C \cdot Z$, where $Z = (W, \mathsf{x}, \vec{1})^\top$. In Verdict, we additionally require *IO consistency*, i.e., that the input of each data-parallel unit is the output of the previous unit. WLOG, we assume that for each i, $|x_i|$ is even and that the first half of x_i and the second half of x_i are respectively the input and output of the ith data-parallel unit. Formally, we require that

$$\begin{aligned} &\mathsf{x}_i[\ell/2:] = \mathsf{x}_{i+1}[:\ell/2] \text{ for all } 1 \leq i \leq \beta - 1 \\ &\mathsf{x}_1[:\ell/2] = x[:\ell/2] \\ &\mathsf{x}_\beta[\ell/2:] = x[\ell/2:]. \end{aligned} \tag{1}$$

C. Constant Proof Sizes and Verification Times

Phalanx builds on recent work [36] that enables the prover and the verifier to combine two N-sized R1CS instances into a single N-sized instance with an $O_\lambda(1)$-sized proof, such that the prover only needs to prove the validity of the combined instance. Such a protocol is referred to as a *folding scheme* and is used to construct recursive SNARKs. We extend prior work [36] to design a folding scheme for SIMD R1CS. Specifically, in each Verdict epoch, Phalanx's prover and verifier fold a SIMD R1CS instance (encoding the statement proven in the epoch) into a running instance. Below, we describe details of Phalanx's folding procedure.

a) A Folding Scheme for SIMD R1CS: Consider a SIMD R1CS instance $\phi = ((\mathbb{F}, A, B, C, m, n, \ell, \beta), x)$. While it is unclear how to fold two such instances in general, it is possible to fold two SIMD R1CS instances with the same structure (i.e., the same A, B, C matrices) by first "relaxing" the instance. Specifically, we define a "slack" matrix $E = \vec{0}^{m \times \beta}$ and scalar $u = 1$ and transform

the satisfiability check into checking

$$AZ \circ BZ = u \cdot CZ + E, \tag{2}$$

where $Z = (W, \mathsf{x}, u)^\top$.

Now, given two instances $\phi_1 = (x_1, E_1, u_1)$, and $\phi_2 = (x_2, E_2, u_2)$ defined over the same matrices A, B, C with corresponding witnesses (W_1, x_1) and (W_2, x_2), the verifier can fold them into a new instance ϕ by randomly sampling $r \in \mathbb{F}$ and taking a random linear combination:

$$\phi \leftarrow (x_1 + r \cdot x_2, E_1 + r \cdot T + r^2 \cdot E_2, u_1 + r \cdot u_2)$$
$$(W, \mathsf{x}) \leftarrow (W_1 + r \cdot W_2, \mathsf{x}_1 + r \cdot \mathsf{x}_2)$$

where $T \leftarrow AZ_1 \circ BZ_2 + AZ_2 \circ BZ_1 - u_1 CZ_2 - u_2 CZ_1$ for $Z_1 \leftarrow (W_1, \mathsf{x}_1, u_1)$ and $Z_2 \leftarrow (W_2, \mathsf{x}_2, u_2)$.

With textbook algebra, it is easy to show that (W, x) satisfies check (2) with respect to ϕ if (W_1, x_1) and (W_2, x_2) satisfy check (2) with respect to ϕ_1 and ϕ_2 respectively. Conversely, as with most batching techniques, soundness holds due to the randomness of the linear combination, which ensures with high probability that if (W, x) satisfies check (2) with respect to ϕ then (W_1, x_1) and (W_2, x_2) also satisfy check (2). The same reasoning holds for IO consistency i.e., if matrices x_1 and x_2 both satisfy their respective IO consistency checks with respect to x_1 and x_2, then a folded input

$$\mathsf{x} \leftarrow \mathsf{x}_1 + r \cdot \mathsf{x}_2$$

for some randomly sampled $r \leftarrow_R \mathbb{F}$, also satisfies the IO consistency check with respect to x. Conversely, if x satisfies the IO consistency check, then it must hold with high probability that both x_1 and x_2 also satisfy the input consistency check with respect to x_1 and x_2.

For efficiency, the prover treats $(E_1, E_2, W_1, W_2, \mathsf{x}_1, \mathsf{x}_2)$ as the witness and provides additively homomorphic commitments to these values as part of the instance. The prover also provides a commitment to T, instead of sending it directly. Then, instead of computing (linearly sized) E, W, and x, the verifier homomorphically computes commitments to E, W, and x as part of the new instance. Because the folding scheme is public coin,[7] we make it non-interactive via the Fiat-Shamir transform [28]. If the commitments are constant-sized (as is the case with our choice, see §IV-E), then Phalanx achieves (per epoch) constant-sized proofs and verification times.

b) Bootstrapping and Inter-Epoch IO Consistency.: At initialization, the running instance in Phalanx is the SIMD R1CS of the first Verdict epoch, relaxed by using default values of u and E (i.e., $u = 1$ and $E = \vec{0}^{m \times \beta}$). For epoch i ($i \geq 2$), the prover and the verifier use the folding scheme described above to fold the SIMD R1CS instance of epoch i with the running instance. Note that the the instance that is folded into the running instance is a SIMD R1CS instance, so it is first relaxed by using the default values of u and E.

In addition to the running instance, the verifier maintains $y_{last} \in \mathbb{F}^{\ell/2}$, which represents the output of the last SIMD R1CS instance that was folded. At initialization, $y_{last} = x[\ell/2:]$, where x is the public input/output of the SIMD R1CS of the first Verdict epoch (the verifier additionally checks that $x[:\ell/2]$ holds the well-known initial input e.g., in Verdict, $x[:\ell/2]$ must hold the Merkle root of an empty indexed Merkle tree of a pre-defined size). In epoch i (where $i \geq 2$), the verifier checks that $y_{last} = \phi.x[:\ell/2]$, where ϕ is the SIMD R1CS instance of epoch i, and then after running the folding procedure, it updates y_{last} to $\phi.x[\ell/2:]$.

D. Proving the Satisfiability of Running Instance

The previous subsection describes how Phalanx's prover and verifier fold a SIMD R1CS instance (of each epoch) into a running instance. We now describe how the prover produces a succinct proof of the satisfiability of the running instance in each epoch. To accomplish this, Phalanx relies on techniques from Spartan [56].

As background, Spartan [56] combines the sum-check protocol [44] with polynomial commitments [34] to obtain a SNARK. Alternatively, Spartan [56] can be viewed as combining a public-coin *polynomial interactive oracle proof (IOP)* [19] for R1CS with polynomial commitments [41]. A polynomial IOP is an interactive proof [32] where in each round the prover sends a polynomial as an oracle and the verifier may request an evaluation of the polynomial at a point in its domain. A public-coin polynomial IOP can be compiled into a public-coin interactive argument of knowledge using a polynomial commitment scheme.[8] The resulting interactive argument can then be turned into a SNARK in the random oracle model [28]. We refer the reader to prior work [41, 56, 59, 62] for details. Thaler [62] in particular provides detailed background as well as descriptions of several SNARKs, including Spartan.

Our main contribution is to provide a polynomial IOP for (relaxed) SIMD R1CS, adapting the polynomial IOP for R1CS from Spartan. To create a new polynomial IOP for relaxed SIMD R1CS, we first encode a relaxed SIMD R1CS instance as a set of polynomials. For this, we first interpret matrices and vectors as functions that map bits to elements of $\mathbb{F}$. For example, a vector V of length m over $\mathbb{F}$ can be viewed as function with signature: $\{0,1\}^{\log m} \rightarrow \mathbb{F}$. We then take *multilinear extensions* of these functions. A multilinear extension of a function is the unique multilinear polynomial whose evaluations match that of the function over the domain of

[7] An interactive protocol is public coin if the verifier's challenges are chosen uniformly at random.

[8] Instead of sending a polynomial, the prover sends a commitment to its polynomial, and when the verifier requests an evaluation of the polynomial, the prover sends an evaluation along with a proof that the evaluation is consistent with the prior commitment.

the function. For example, a multilinear extension of the aforementioned V viewed as a function is a polynomial $\widetilde{V} : \mathbb{F}^{\log m} \to \mathbb{F}$, where $\widetilde{V}(i) = V(i)$ for all $i \in \{0,1\}^{\log m}$.

Let $\widetilde{A}$, $\widetilde{B}$, $\widetilde{C}$, represent the multilinear extensions of A, B, C respectively. Consider a purported witness $(W, E, \mathsf{x}) \in (\mathbb{F}^{m-\ell-1 \times \beta}, \mathbb{F}^{m \times \beta}, \mathbb{F}^{\ell \times \beta})$. Let $\widetilde{W}$, $\widetilde{E}$, $\widetilde{\mathsf{x}}$ denote the corresponding multilinear extensions, and let $\widetilde{Z}$ denote the multilinear extension of matrix $Z = (W, \mathsf{x}, \vec{1})^\top$. As part of the running instance, Phalanx's prover and verifier hold x, u, and commitments $\overline{W}$, $\overline{E}$, and $\overline{\mathsf{x}}$ to the prover's witness $\widetilde{W}$, $\widetilde{E}$, and $\widetilde{\mathsf{x}}$ respectively.

Given these polynomials, we define a polynomial F that evaluates to zero iff a given SIMD R1CS instance is satisfiable. Let $s = \log m$ and $c = \log \beta$.

$$\begin{aligned} F(k,i) = &\left(\sum_{j \in \{0,1\}^s} \widetilde{A}(i,j) \cdot \widetilde{Z}(k,j) \right) \cdot \\ &\left(\sum_{j \in \{0,1\}^s} \widetilde{B}(i,j) \cdot \widetilde{Z}(k,j) \right) - \\ &u \cdot \left(\sum_{j \in \{0,1\}^s} \widetilde{C}(i,j) \cdot \widetilde{Z}(k,j) \right) + \widetilde{E}(k,i) \end{aligned} \tag{3}$$

Lemma 1. *If (W, x, E) is a satisfying assignment to a relaxed SIMD R1CS instance $((\mathbb{F}, A, B, C, m, n, \ell, \beta), (x, \overline{E}, u, \overline{W}))$, then $F(k,i) = 0$ for all $k \in \{0,1\}^c$ and $i \in \{0,1\}^s$.*

It is unclear how the prover can prove that F evaluates to zero over the Boolean hypercube defined by k and i, so we instead define a *multilinear* polynomial Q:

$$Q(t_1, t_2) = \sum_{k \in \{0,1\}^c} \sum_{i \in \{0,1\}^s} F(k,i) \cdot \widetilde{\mathsf{eq}}((k,i),(t_1,t_2))$$

where $\widetilde{\mathsf{eq}}$ is the multilinear extension of the function eq defined as follows: $\forall x, y$ over the domain of eq, $\mathsf{eq}(i,j) = 1$ if $i = j$ and 0 otherwise.

Due to the multilinearity of Q and the observation that if Z is a satisfying witness, then $Q(k,i) = 0$ for all $k \in \{0,1\}^c$ and $i \in \{0,1\}^s$, we have that Q is the zero polynomial iff Z is a satisfying witness. Therefore, it is sufficient for the prover to prove $Q(\tau_1, \tau_2) = 0$, where $\tau_1, \tau_2 \leftarrow_R \mathbb{F}$ are chosen by the verifier. Furthermore, the instance to be proven is in a form suitable for the application of the sum-check protocol [44]: an interactive proof system for proving $T = \sum_{i \in \{0,1\}^s} G(i)$, where G is an s-variate multivariate polynomial over $\mathbb{F}$ and $T \in \mathbb{F}$.

Phalanx's interactive argument proceeds as follows:

1) The verifier sends $(\tau_1, \tau_2) \leftarrow_R \mathbb{F}^c \times \mathbb{F}^s$.
2) The prover and the verifier use the sum-check protocol to reduce the task of checking $Q(\tau_1, \tau_2) = 0$ to checking $F(r_k, r_i) = e$, where $(r_k, r_i) \in \mathbb{F}^{c+s}$ are chosen by the verifier over the course of the sum-check protocol and $e \in \mathbb{F}$.
3) The prover and the verifier use the sum-check protocol to reduce the task of checking $F(r_k, r_i) = e$ to checking claimed evaluations of $\widetilde{A}, \widetilde{B}, \widetilde{C}$ at (r_i, r_j), $\widetilde{E}(r_k, r_i)$, and $\widetilde{Z}(r_k, r_j)$, where $r_j \in \mathbb{F}^s$ is once again sampled by the verifier during the sum-check protocol. The first three are evaluated by the verifier locally (or by using a (sparse) polynomial commitment scheme [56]).[9] The prover proves evaluations of $\widetilde{E}$, $\widetilde{W}$, and $\widetilde{\mathsf{x}}$. Below, we show how the verifier can efficiently evaluate $\widetilde{Z}(r_k, r_j)$ using an evaluation of $\widetilde{W}$ and $\widetilde{\mathsf{x}}$.

a) Computing $\widetilde{Z}(r_k, r_j)$: WLOG, assume that for each k, $|W[k]| = |\mathsf{x}[k]| + 1 = m/2$. For all $0 \le k < \beta$ and $0 \le j < m$, we have

$$Z_{k \cdot m + j} = \begin{cases} \mathsf{x}[k \cdot m/2 + j/2], & \text{if } j < m/2 \\ W[k \cdot m/2 + j/2], & \text{otherwise} \end{cases} \tag{4}$$

Let b be the binary representation of $(k \cdot m + j)$. We use $b[i]$ to refer to the bit i of b with $b[0]$ corresponding to the MSB. Then by equation (4), we have that $\forall b \in \{0,1\}^{c+s}$

$$Z[b] = \begin{cases} \mathsf{x}[b[0,1,\ldots,(c-1),(c+1),\ldots,(c+s)]], \\ \qquad\qquad \text{if } b[c] = 0. \\ W[b[0,1,\ldots,(c-1),(c+1),\ldots,(c+s)]], \\ \qquad\qquad \text{otherwise.} \end{cases}$$

Thus, for the multilinear extensions of Z, W and x:

$$\begin{aligned} \widetilde{Z}(r) = &(1 - r[c]) \cdot \widetilde{\mathsf{x}}(r[0,\ldots,(c-1),(c+1)\ldots]) + \\ &r[c] \cdot \widetilde{W}(r[0\ldots(c-1),(c+1)\ldots]) \end{aligned}$$

Thus, the prover sends an evaluation of $\widetilde{W}$ and $\widetilde{\mathsf{x}}$ at point $r' = r[0,\ldots,(c-1),(c+1),\ldots]$ along with a proof of correct evaluation. This aids the verifier with completing the final step of the interactive argument depicted above.

b) Proving IO Consistency Checks: The prover can send x and the verifier can check: (1) x is consistent with the commitment $\overline{\mathsf{x}}$ it holds as part of the running instance; (2) x satisfies the desired IO consistency; and (3) x is consistent with the public input/output x in the running instance. However, this incurs $O(\beta)$ proof sizes and verifier times. Instead, Phalanx does the following: At the time of folding, instead of committing to x, the prover commits to a "deduplicated version" of x (which obviates the need to prove check (2)). The prover then uses a simplified Spartan to prove the knowledge of x such that checks (1) and (3) hold; the prover also proves the evaluation of the multilinear extension of x necessary to compute $\widetilde{Z}(r_k, r_j)$ described above. The proof sizes are $O(\log(\beta))$ and verification times requires $O(\beta)$ finite field operations. Using sparse polynomial commitments from Spartan [56], the verification times

[9] Even in the case the verifier evaluates these polynomials locally, the cost is $O(n)$ and is independent of β, whereas without a sparse polynomial commitment scheme, Spartan would require $O(n \cdot \beta)$ time.

can also be made $O(\log \beta)$ cryptographic operations; however, doing so provides benefits only when β is large (e.g., $\beta > 2^{15}$).

E. Phalanx's Commitment Scheme and Phalanx (Lazy)

Phalanx needs a polynomial commitment scheme. For this, Phalanx uses Dory [39], which results in the following asymptotics. Phalanx produces $O_\lambda(1)$-sized proofs and verification times for each Verdict epoch (§IV-C). Furthermore, for proving the running instance (§IV-D) with β data-parallel units each of size c, Phalanx produces $O_\lambda(\log(c \cdot \beta))$-sized proofs that can be verified in $O_\lambda(\log(c \cdot \beta))$ time.

Another commitment scheme choice is Hyrax-PC [66]. With this choice, Phalanx provides $O_\lambda(1)$-sized proofs and verification times for each Verdict epoch (§IV-C). It also produces $O_\lambda(\log(c \cdot \beta))$-sized proofs for proving a running instance. However, with this choice, the cost of verifying the proof for the running instance is $O_\lambda(c \cdot \beta)$, which is high. So, they can be verified only infrequently. Given this, we refer to this variant of Phalanx as "Phalanx (lazy)" since it provides *deferred* guarantees i.e., the guarantees hold only when clients, in some future epoch, check a succinct proof produced by the prover.

Finally, one might wonder if one needs Phalanx (lazy) since Phalanx provides better flexibility and asymptotics. We find that Phalanx (lazy)'s prover is $\approx 5\times$ faster than Phalanx's at the cost of providing deferred guarantees.

V. Implementation and Applications

We implement Verdict in about 7,400 lines of Rust. This consists of an implementation of the Verdict service, which uses Redis [7] to store its state (a map from labels to append-only hashchains and an indexed Merkle tree that in turn stores a derived dictionary), and a client library that exposes VerifyUpdates and VerifyLookup procedures to verify proofs produced by the service. This is about 3,600 lines of Rust. We implement Phalanx as a library by extending libSpartan [8] and Xiphos [59] with about 6,000 lines of Rust.

Implementing Verdict requires constructing a SIMD R1CS instance that verifies a batch of indexed Merkle proofs. However, libSpartan provides only a low-level API that accepts R1CS matrices, which is unusable for more complex applications such as Verdict. We address this by leveraging bellman [1, 2], which provides R1CS "gadgets" for hash functions and other primitives that can be composed and extended to build higher-level apps. Specifically, we implement an adapter that lets a programmer compose and use existing bellman gadgets with libSpartan; this is about 1,000 lines of Rust. For the hash function in Verdict's SIMD R1CS, we use MiMC [9], a SNARK-friendly hash function.

a) Application: Key Transparency: As a concrete application of Verdict, we design Keypal, a public-key directory, where the service's state is a label-value map in which labels are client identifiers (e.g., email addresses) and values are the set of public keys associated with the identifier. Keypal supports four types of requests from clients: **(1)** register a new id and associate an initial key, **(2)** add a new key to an existing id, **(3)** revoke a key associated with an existing id, and **(4)** look up the set of keys associated with an existing key. For adding or revoking keys, Keypal uses a simple policy that such requests must be digitally signed by one of the existing, unrevoked keys associated with the identity. Keypal's implementation uses Ed25519 signatures [6].

VI. Experimental Evaluation

Our principal experimental questions are:

- How does Verdict compare with prior work?
- How do Verdict's techniques improve its costs?

We run our experiments on Azure Standard F64s_v2 (64 vCPUs, 2.70 GHz Intel(R) Xeon(R) Platinum 8168, 128 GB RAM) running Ubuntu 18.04. However, we only one utilize one CPU core in our experiments.

A. Comparison with Prior Work

We compare Verdict and Verdict (lazy) with three baselines and use Keypal as the concrete application.

First, AAD [63], a prior transparency dictionary, with asymptotic and security properties similar to Verdict. Unlike Verdict, it requires a trusted setup. Second, a naive baseline in which the service organizes its dictionary in a Merkle-Patricia tree. In epoch t, the prover sends a commitment $\mathcal{C}_t$ to its updated state and the following: for each update, the label, the new value, the old value (if it exists), and a membership proof. The verifier uses the membership proof to verify that the old value (or its absence) is consistent with $\mathcal{C}_{t-1}$ and the new value is consistent with $\mathcal{C}_t$. Third, naive++, an optimized version of the naive baseline in which the service applies a batch of updates at once, and instead of producing a separate proof for each update, it merges the individual proofs and deduplicates them when possible.

Note that naive and naive++ are more efficient than prior work such as SEEMless [22], which additionally protects privacy, so the service cannot directly send Merkle proofs to clients and hence incurs additional expense. Similarly, other privately-verifiable works [33, 48] when transformed to produce publicly-verifiable proofs would incur more expense than naive and naive++.

We are interested in the performance of two operations: **(1)** lookups, and **(2)** a batch of inserts/updates. For lookups, our evaluation metrics are: **(a)** the size of a lookup proof; and **(b)** the verifier's cost of verifying a lookup proof. For updates, our metrics are: **(a)** the

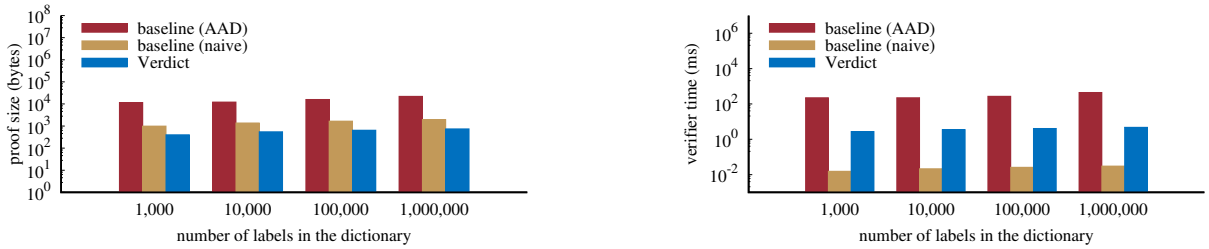

Figure 4: Proof sizes and verification times for lookups, with varying number of labels in a dictionary. Results for naive++ and Verdict (lazy) are the same as the results for naive and Verdict respectively, so we do not depict them.

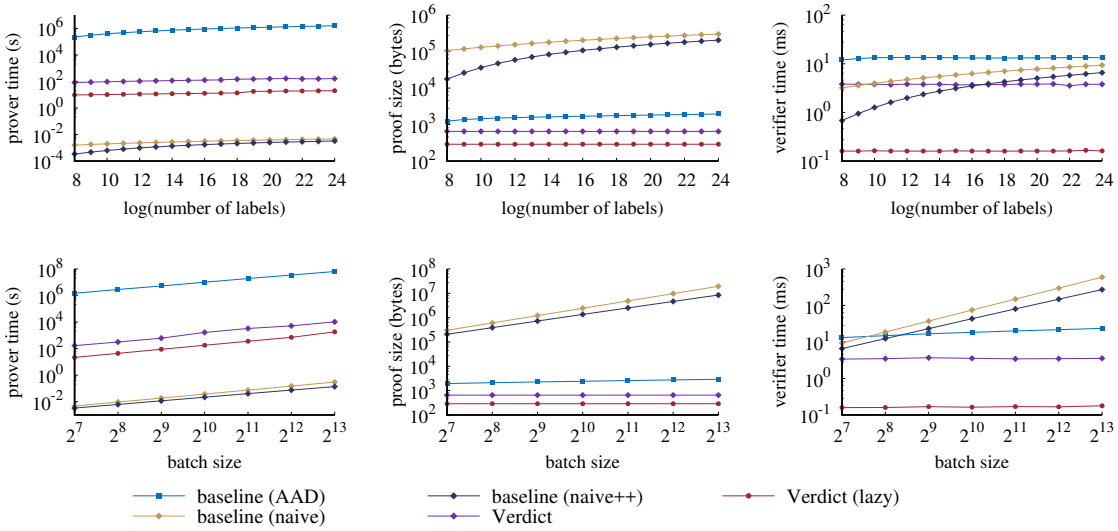

Figure 5: Per-epoch prover time, proof sizes, and verifier time for updates/inserts. The first row depicts costs for a batch size of $\beta = 128$ with varying number of labels in a dictionary (n), and the second row depicts costs for $n = 2^{24}$ and varying β. For Verdict, the depicted prover time includes the cost of producing both a constant-sized proof and a logarithmic-sized proof, while the depicted proof sizes and verification times are for the constant sized proofs (since clients can amortize the the cost of verifying logarithmic-sized proofs); similarly, for Verdict (lazy), we depict per-epoch costs; §VI-A provides details of amortization.

prover's cost of processing a batch of updates and producing an update proof; **(b)** the size of an update proof; and **(c)** the verifier's cost to verify an update proof. We do not focus on the cost of producing a lookup proof since in Verdict and the naive baselines, the cost is based purely on the performance of the underlying storage system used to maintain state (recall that in Verdict, a lookup proof is produced by retrieving appropriate nodes in an indexed Merkle tree stored in a commodity storage service). Verdict supports thousands of lookup requests per second on a commodity VM, and it can be scaled up with standard systems techniques.

To measure the performance of AAD [63], we use its C++ implementation [4]. Unfortunately, it only supports small dictionaries (e.g., 8,192 label-value tuples); for larger dictionaries, the prover time is several hours or more, so we use results from smaller experiments and the authors' cost models to extrapolate its costs for larger dictionary sizes. Furthermore, we assume that there is only one value associated with the requested label as the lookup proof sizes and verification times grow with the number of operations associated with the requested label. For updates/inserts, AAD's performance depends on the number of trees in the forest that comprises the directory, so for a desired dictionary size, we measure the cost of doubling the dictionary and compute the amortized per-operation cost (this is optimistic for AAD).

For the naive baselines, we report costs based on microbenchmarks and cost models. We assume that the Merkle-Patricia trees use SHA-256 for the hash function.

a) Results for Lookups: Figure 4 depicts our results. First, Verdict provides the shortest lookup proofs. Verdict has 28–29× smaller proofs than AAD. Proofs in the naive baseline are ≈2.5× larger than Verdict's proofs because each membership proof in a Merkle-Patricia tree contains more data in intermediate nodes.

Second, Verdict's lookup proof verification times are not the fastest, but they are fast: 2.5–5 ms for Verdict and tens of microseconds for the naive baselines. This is because the naive baselines use SHA-256, whereas Verdict uses a SNARK-friendly hash function, which is more expensive on x86. However, Verdict's verification times are more than 80× cheaper than AAD's.

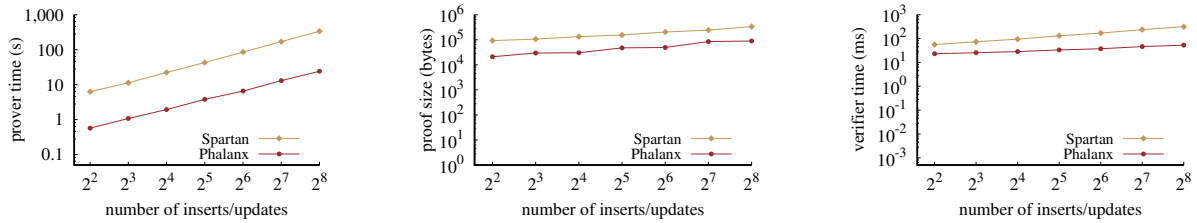

Figure 6: Prover time, proof sizes, and verifier time under Phalanx and its baseline Spartan [56] for a SIMD R1CS instance with varying number of insert/update operations on a dictionary with 2^{20} labels (see the text for details).

b) Results for Inserts/Updates: We experiment with Verdict and its baselines using $\beta = 128$ with varying number of labels in the dictionary. To demonstrate scalability, we also experiment with varying batch sizes for a dictionary with 2^{24} labels. For the latter large-scale experiments, we use a different VM with larger RAM, which is an Azure Standard E64-16ds_v4 with 504 GB RAM (we use only one CPU core and its performance is identical to that of the VM listed earlier); also, Verdict uses $< 50\%$ of the available memory in all experiments

Figure 5 depicts our results. We find the following.

Verdict's per-epoch constant-sized proofs are 651 bytes and take about 3 ms to verify. Verdict's logarithmic-sized proofs for the running instance are 26–32 KB for a dictionary with 2^{24} labels, when the batch size varies from $\beta = 2^7$ to $\beta = 2^{13}$; verifying these proofs takes 44 ms for $\beta = 128$ and 77 ms for $\beta = 2^{13}$.

Recall that Verdict's clients only need to verify the logarithmic-sized proof of the most recent epoch (before issuing lookup operations). However, we find that, for a dictionary of size 2^{24}, even if clients check these proofs in *every* epoch, Verdict's clients are more efficient than the ones of naive and naive++ (which employ a fast hash function) as long as $\beta \geq 2^{10}$. Furthermore, at this configuration, Verdict's proof size is over an order of magnitude shorter. Specifically, it is ≈30 KB whereas naive's proof size is ≈2.4 MB and naive++'s proof size is ≈1.4 MB. Verdict's logarithmic-sized proofs are concretely larger than AAD's, but Verdict's clients can incur lower amortized proof sizes and verification times (e.g., if clients verify these logarithmic-sized proofs after 14 epochs when the batch size is 2^{13}). For prover times, Verdict is slower than both naive and naive++, but Verdict is faster than AAD by 2–3 orders of magnitude.

Verdict (lazy) produces shorter per-epoch proofs than Verdict: proofs are 290 bytes and verification times are 161μs. However, generating proofs for running instances is over 3× slower than in Verdict (e.g., for dictionaries with 2^{24} labels and $\beta = 128$ Verdict (lazy) takes takes 303 s whereas Verdict takes 69 s). Finally, verifying these proofs is slower than in Verdict by over an order of magnitude: for the aforementioned workload, the verifier under Verdict (lazy) takes 29s. So, unlike Verdict, Verdict (lazy) needs hundreds of epochs to achieve faster verification times than the naive baselines.

	Phalanx (eager)	Phalanx
Prover time	72 s	168 s
Proof size (per-epoch)	26 KB	651 bytes
Verifier time (per-epoch)	44 ms	3.4 ms
Proof size (fixed)	N/A	26 KB
Verifier time (fixed)	N/A	44 ms

Figure 7: Performance of Phalanx (eager) and Phalanx for proving a batch of 128 insert/update operations on a dictionary with 2^{24} labels. See the text for details.

c) Storage Costs: Compared to a baseline that stores only a dictionary, Verdict incurs ≈2× overhead from maintaining an indexed Merkle tree. This overhead is similar to those of naive and naive++, but much smaller than AAD, Merkle^2, CONIKS, or SEEMless. Concretely, for a dictionary with 2^{20} labels, Verdict uses 128 MB to store the indexed Merkle tree. Moreover, Verdict maintains a hashchain for each label. Each node in the chain is 167 bytes and contains an operation (49 bytes), a signature (78 bytes), and the hash of the previous node (40 bytes). Finally, for the service to produce proofs, it stores SNARK parameters. Phalanx's parameters are smaller than Phalanx (lazy)'s parameters, both asymptotically and concretely. Concretely, for a dictionary with 2^{24} labels and $\beta = 1024$, the parameters are about 103 MB under Phalanx and 2 GB under Phalanx (lazy).

B. Improvements From Verdict's Techniques

a) Benefits of Phalanx's polynomial IOP for SIMD R1CS: We consider a SIMD R1CS instance where each sub-circuit performs one update/insert operation on a dictionary with 2^{20} labels. We measure the performance of Phalanx with a varying number of update/insert operations, and our performance metrics are: (1) the prover time, (2) the proof size, and (3) the verifier time. Our baseline is Spartan [56]. To focus performance on the polynomial IOP, we configure Phalanx to use the same polynomial commitment scheme as Spartan, which is Hyrax-PC configured to produce $O_\lambda(\sqrt{m})$-sized commitments for m-sized multilinear polynomials.

Figure 6 depicts our results. We find that Phalanx is more efficient than Spartan, with Phalanx's prover faster by over an order of magnitude compared to Spartan. Prior to this work, Spartan offers the fastest zkSNARK [56], so for SIMD computations, Phalanx has the fastest prover.

b) Cost and benefits of Phalanx's folding scheme: To evaluate the benefits of Phalanx's folding scheme, we consider a variant of Phalanx that proves SIMD R1CS instances but does not employ Verdict's folding scheme. We refer to this variant as "Phalanx (eager)". Asymptotically, for an N-sized SIMD R1CS, the prover times are $O_\lambda(N)$ under both Phalanx and Phalanx (eager); the proof sizes and verifier times are $O_\lambda(\log N)$ under Phalanx (eager) whereas they are $O_\lambda(1)$ under Phalanx. For proving running instances, Phalanx however produces $O_\lambda(\log N)$-sized proofs that take $O_\lambda(\log N)$ time to verify, but this cost can be amortized across epochs at each client's discretion (§IV-E).

Figure 7 depicts our results for applying a batch of 128 inserts/updates on a dictionary with 2^{24} labels (the relative results for other dictionary sizes and batch sizes are similar). Since Phalanx must commit to an additional vector T (§IV-C), its prover incurs higher costs than Phalanx (eager) by about 2.3×. In exchange, the per-epoch verifier time and proof sizes are both lower by over an order of magnitude under Phalanx compared with Phalanx (eager). Even when accounting for fixed costs incurred by Phalanx, Phalanx still incurs lower verifier costs and proof sizes than Phalanx (eager) as long as the fixed costs are amortized over ≥ 2 epochs.

VII. Related Work

Many prior works provide a transparency dictionary. However, they do not satisfy all of our requirements (§I). For example, CONIKS [48], does not provide public verifiability. Specifically, whenever the untrusted CONIKS service processes a batch of updates, it produces a separate privately-verifiable proof for each client. Very recently, Merkle^2 [33] improves on CONIKS by partially proving the append-only property with a publicly-verifiable proof. However, it still produces a privately-verifiable proof for each update submitted by a client. It is possible to combine all the privately-verifiable proofs produced by the untrusted service to construct a publicly-verifiable proof. However, such a proof is neither succinct nor efficiently-verifiable. SEEMless [22] provides publicly verifiable proofs, but it does not satisfy efficient verifiability. Similarly, keybase [3] maintains a key directory and periodically publishes a commitment to its state on a public blockchain. However, it does not provide efficient verifiability. AAD [63] produces publicly-verifiable proofs with efficient verifiability. However, it requires a trusted setup[10] and imposes high concrete costs on the service to produce proofs (§VI). Finally, recent, concurrent works [23, 64] explore the use of SNARKs to reduce costs for clients of transparency dictionaries. However, these solutions incur high prover costs to produce proofs, and they require a trusted setup.

In *verifiable state machines (VSMs)*, an untrusted service proves the correct execution of state machine transitions using SNARKs [57]. Due to their generality, VSMs imply transparency dictionaries. However, existing work does not optimize core ingredients to efficiently realize a transparency dictionary. Specifically, several works [18, 29] compose Merkle trees with SNARKs, but they target general computation. Spice [40, 57] composes a multiset-based data structure with SNARKs. However, it must treat both reads and writes in a uniform manner, so the high cost of SNARKs is incurred for both reads and writes. Whereas, in Verdict, reads do not require the use of SNARKs. Similarly, Ozdemir et al. [51] compose RSA accumulators with SNARKs. However, as discussed in Section III, RSA accumulators require $O_\lambda(n)$ computation for the service to respond to a lookup operation over an n-sized dictionary. Also, RSA accumulators require a trusted setup.

A large body of work offers untrusted storage systems [27, 43, 45, 47, 53]. However, they target scenarios where a small number of clients collectively read and write data stored at the service. Furthermore, they require clients to effectively execute all updates processed by the service. In Orochi [61], a service executes requests and produces a log such that anyone can verify whether the service behaved correctly. Unfortunately, this amounts to reexecuting all the requests. Concerto [11] is a verifiable key-value store. However, Concerto's verifier must replay all operations executed by an untrusted service. Hence, it does not provide efficient verifiability. Furthermore, verification requires periodically scanning the entire state maintained by the service. Byzantine fault-tolerant systems [21, 46, 67] can provide a transparency dictionary. However, they do not provide cryptographic proofs and require threshold assumptions.

While Verdict shares some of its tools with blockchains, it differs from them in several respects. First, blockchains employ a single global hashchain to order blocks of transactions, whereas Verdict uses a hashchain for each user to order updates. Second, while blockchains can be used to build a transparency dictionary [5, 55] (e.g., Microsoft's ION [5] records operations submitted by users in Bitcoin's blockchain), to retrieve the latest public key associated with a particular identity, a client must effectively reexecute all operations in the order they are recorded on the blockchain. In contrast, with Verdict, an untrusted service provides a response to a lookup along with a proof. Verdict's service can equivocate and expose different sequences of commitments to different clients, but this can be prevented by having the service use a blockchain to disseminate a single sequence of commitments to clients (note that the use of blockchain still does not require Verdict's clients to reexecute all operations processed by the system).

[10]Trusted setup means that a party (or a group of parties of which at least one is honest) is trusted by all clients. That party produces public parameters for a proof system using a secret trapdoor. However, anyone with the knowledge of the trapdoor can forge false proofs.

Persistent authenticated dictionaries (PADS) [10, 24, 54] provide specialized data-structures such as skip-lists and red-black trees. PADs provide succinct proofs for each operation (logarithmic in the size of the state). However, to prove that a batch of updates were executed correctly, the proof size grows linearly with the number of updates, similar to the two naive baselines that we consider in Section VI. In contrast, Verdict uses SNARKs to produce a succinct proof for a batch of updates.

VIII. Summary and Conclusion

This paper asks: can we build a service that provides a dictionary abstraction and produces efficiently-verifiable cryptographic proofs of its correct operation? Our system, Verdict, meets these requirements and scales well to large dictionaries. To achieve this, Verdict incorporates a novel synthesis of existing and new techniques in SNARKs and cryptographic accumulators. Finally, we believe Verdict represents a novel application of SNARKs to build real-world services, such as public key directories, that can prove their own correctness.

Acknowledgments

We thank Justin Thaler and Michael Walfish for comments on an earlier version of this paper. This work was supported in part by Bosch, the Alfred P. Sloan Foundation, the NSF under Grant No. 1801369 and 1514422, the Air Force Office of Scientific Research under Grant No. FA9550-18-1-0421, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] "Bellman," https://github.com/zkcrypto/bellman.
[2] "Bellman-bignat," https://github.com/alex-ozdemir/bellman-bignat.
[3] "keybase," https://keybase.io/.
[4] "libaad," https://github.com/alinush/libaad-ccs2019.
[5] "Microsoft ION," https://github.com/decentralized-identity/ion.
[6] "A pure-Rust implementation of group operations on Ristretto and Curve25519," https://github.com/dalek-cryptography/curve25519-dalek.
[7] "Redis," https://redis.io.
[8] "Spartan: High-speed zkSNARKs without trusted setup," https://github.com/Microsoft/Spartan.
[9] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity," in *ASIACRYPT*, 2016.
[10] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, "Persistent authenticated dictionaries and their applications," in *International Conference on Information Security*, 2001, pp. 379–393.
[11] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy, "Concerto: A high concurrency key-value store with integrity," in *SIGMOD*, 2017.
[12] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer, "Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: Extended abstract," in *ITCS*, 2013.
[13] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, "Aurora: Transparent succinct arguments for R1CS," in *EUROCRYPT*, 2019.
[14] J. C. Benaloh and M. de Mare, "One-way accumulators: A decentralized alternative to digital sinatures (extended abstract)," in *EUROCRYPT*, 1994.
[15] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *ITCS*, 2012.
[16] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky, "Succinct non-interactive arguments via linear interactive proofs," in *TCC*, 2013.
[17] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," in *FOCS*, 1991.
[18] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *SOSP*, 2013.
[19] B. Bunz, B. Fisch, and A. Szepieniec, "Transparent SNARKs from DARK compilers," ePrint Report 2019/1229, 2019.
[20] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *CRYPTO*, 2002.
[21] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," vol. 20, no. 4, pp. 398–461, Nov. 2002.
[22] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai, "Seemless: Secure end-to-end encrypted messaging with less trust," in *CCS*, 2019.
[23] W. Chen, A. Chiesa, E. Dauterman, and N. P. Ward, "Reducing participation costs via incremental verification for ledger systems," Cryptology ePrint Archive, Report 2020/1522, 2020.
[24] S. A. Crosby and D. S. Wallach, "Super-efficient aggregating history-independent persistent authenticated dictionaries," in *European Symposium on Research in Computer Security*, 2009, pp. 671–688.
[25] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse Merkle trees: Caching strategies and secure (non-)membership proofs," Cryptology ePrint Archive, Report 2016/683, 2016.
[26] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno, "Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation," in *S&P*, 2016.
[27] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: group collaboration using untrusted cloud resources," in *OSDI*, 2010.
[28] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO*, 1986, pp. 186–194.
[29] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno, "Hash first, argue later: Adaptive verifiable computations on outsourced data," in *CCS*, 2016.
[30] R. Gennaro, C. Gentry, B. Parno, and M. Raykova,

"Quadratic span programs and succinct NIZKs without PCPs," in *EUROCRYPT*, 2013.
[31] C. Gentry and D. Wichs, "Separating succinct non-interactive arguments from all falsifiable assumptions," in *STOC*, 2011, pp. 99–108.
[32] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *STOC*, 1985.
[33] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa, "Merkle^2: A low-latency transparency log system," Cryptology ePrint Archive, Report 2021/453, 2021.
[34] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT*, 2010, pp. 177–194.
[35] A. Kosba, C. Papamanthou, and E. Shi, "xJsnark: A framework for efficient verifiable computation," in *S&P*, 2018.
[36] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes," Cryptology ePrint Archive, Report 2021/370, 2021.
[37] B. Laurie and E. Kasper, "Revocation transparency," 2013, www.links.org/files/RevocationTransparency.pdf.
[38] B. Laurie, "Certificate transparency," p. 4046, Sep. 2014.
[39] J. Lee, "Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments," Cryptology ePrint Archive, Report 2020/1274, 2020.
[40] J. Lee, K. Nikitin, and S. Setty, "Replicated state machines without replicated execution," in *S&P*, 2020.
[41] J. Lee, S. Setty, J. Thaler, and R. Wahby, "Linear-time and post-quantum zero-knowledge snarks for r1cs," Cryptology ePrint Archive, Report 2021/030, 2021.
[42] D. Leung, Y. Gilad, S. Gorbunov, L. Reyzin, and N. Zeldovich, "Aardvark: A concurrent authenticated dictionary with short proofs," Cryptology ePrint Archive, Report 2020/975, 2020.
[43] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *OSDI*, 2004.
[44] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems," in *FOCS*, Oct. 1990.
[45] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," vol. 29, no. 4, Dec. 2011.
[46] D. Mazières, "The Stellar consensus protocol: A federated model of Internet-level consensus," Stellar Development Foundation, Tech. Rep., 2016.
[47] D. Mazières and D. Shasha, "Building secure file systems out of Byzantine storage," in *PODC*, 2002, p. 108117.
[48] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: bringing key transparency to end users," in *USENIX Security*, 2015.
[49] R. C. Merkle, "A digital signature based on a conventional encryption function," in *CRYPTO*, 1988.
[50] A. Oprea and K. D. Bowers, "Authentic time-stamps for archival storage," 2009.
[51] A. Ozdemir, R. S. Wahby, and D. Boneh, "Scaling verifiable computation using efficient set accumulators," in *USENIX Security*, 2020.
[52] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *S&P*, May 2013.
[53] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling security in cloud storage SLAs with CloudProof," 2011.
[54] T. Pulls and R. Peeters, "Balloon: A forward-secure append-only persistent authenticated data structure," in *European Symposium on Research in Computer Security*, 2015, pp. 622–641.
[55] D. Reed, J. Law, and D. Hardman, "The technical foundations of Sovrin," The Sovrin Foundation, Tech. Rep., 2016.
[56] S. Setty, "Spartan: Efficient and general-purpose zkSNARKs without trusted setup," in *CRYPTO*, 2020.
[57] S. Setty, S. Angel, T. Gupta, and J. Lee, "Proving the correct execution of concurrent services in zero-knowledge," in *OSDI*, Oct. 2018.
[58] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish, "Resolving the conflict between generality and plausibility in verified computation," in *EuroSys*, Apr. 2013.
[59] S. Setty and J. Lee, "Quarks: Quadruple-efficient transparent zkSNARKs," Cryptology ePrint Archive, Report 2020/1275, 2020.
[60] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," in *USENIX Security*, Aug. 2012.
[61] C. Tan, L. Yu, J. B. Leners, and M. Walfish, "The efficient server audit problem, deduplicated re-execution, and the Web," in *SOSP*, 2017.
[62] J. Thaler, "Proofs, arguments, and zero-knowledge," http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2020.
[63] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas, "Transparency logs via append-only authenticated dictionaries," in *CCS*, 2019.
[64] N. Tyagi, B. Fisch, J. Bonneau, and S. Tessaro, "Client-auditable verifiable registries," Cryptology ePrint Archive, Report 2021/627, 2021.
[65] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," in *NDSS*, 2015.
[66] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zkSNARKs without trusted setup," in *S&P*, 2018.
[67] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *PODC*, 2019.

APPENDIX

A. Formal Properties of a Transparency Dictionary

We formally define the correctness and soundness properties of a transparency dictionary. In the definitions, we assume that both the initially empty dictionary $D_0 = \perp$ and its commitment $\mathcal{C}_0$ are well-known. A transparency dictionary satisfies the following properties.

- **Update Completeness.** Informally, clients do not reject update proofs produced by an honest service.

Formally, for any epoch t, and sequence of update operations $U_1, \ldots, U_t$, and application-specific update policy F, the following probability is 1:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ \forall i < t: \\ (D_{i+1}, \mathcal{C}_{i+1}, \pi_{i+1}) \leftarrow \mathsf{ApplyUpdates}(\mathsf{pp}, D_i, \mathcal{C}_i, U_i) \\ \mathsf{VerifyUpdates}(\mathsf{pp}, \mathcal{C}_i, \mathcal{C}_{i+1}, \pi_{i+1}) = 1 \end{array}\right]$$

- **Update Knowledge Soundness.** Informally, the set of labels in a dictionary grows monotonically, and the values are updated only according to an application-specific policy. Formally, for any application-specific update policy F and any PPT adversary $\mathcal{A}$, there exists an extractor $\mathcal{E}$, such that for any epoch t the following probability is $\mathtt{negl}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ (\{\mathcal{C}_i, \pi_i\}_{1 \le i \le t}, v, \pi_{\mathsf{lookup}}, \mathsf{label}) \leftarrow \mathcal{A}(\mathsf{pp}, \rho) \\ (\{U_i\}_{1 \le i \le k-1}) \leftarrow \mathcal{E}(\mathsf{pp}, \rho) \text{ for some } k \le t \\ \forall i < k: v_{i+1} \leftarrow F(v_i, U_i) \text{ where } v_1 = \perp \\ \forall i < t: \mathsf{VerifyUpdates}(\mathsf{pp}, \mathcal{C}_i, \mathcal{C}_{i+1}, \pi_{i+1}) = 1 \\ \mathsf{VerifyLookup}(\mathsf{pp}, \mathcal{C}_t, \mathsf{label}, v, \pi_{\mathsf{lookup}}) = 1 \\ v \neq v_k \end{array}\right]$$

where ρ denotes the input randomness for adversary $\mathcal{A}$.

- **Lookup Completeness.** Informally, if the service is honest, then clients accept lookup proofs. Formally, for any application-specific update policy F, epoch t, label label, and sequence of requests $U_1, \ldots, U_t$, the following probability is 1:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ \forall i < t: \\ (D_{i+1}, \pi_{i+1}) \leftarrow \mathsf{ApplyUpdates}(\mathsf{pp}, D_i, \mathcal{C}_i, U_i) \\ (v, \pi_{\mathsf{lookup}}) \leftarrow \mathsf{Lookup}(\mathsf{pp}, D_t, \mathcal{C}_t, \mathsf{label}) \\ \mathsf{VerifyLookup}(\mathsf{pp}, \mathcal{C}_t, \mathsf{label}, v, \pi_{\mathsf{lookup}}) = 1 \end{array}\right]$$

- **Lookup Soundness.** Informally, the service cannot return an incorrect value for any label included in a given commitment. Formally, for any application-specific update policy F, any PPT adversary $\mathcal{A}$, and epoch t, the following probability is negligible in λ:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ (\{\mathcal{C}_i, \pi_i\}_{1 \le i \le t}, v, v', \pi_{\mathsf{lookup}}, \pi'_{\mathsf{lookup}}, \mathsf{label}) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \forall i < t: \\ \mathsf{VerifyUpdates}(\mathsf{pp}, \mathcal{C}_i, \mathcal{C}_{i+1}, \pi_{i+1}) = 1 \\ \mathsf{VerifyLookup}(\mathsf{pp}, \mathcal{C}_t, \mathsf{label}, v, \pi_{\mathsf{lookup}}) = 1 \\ \mathsf{VerifyLookup}(\mathsf{pp}, \mathcal{C}_t, \mathsf{label}, v', \pi'_{\mathsf{lookup}}) = 1 \\ v' \neq v \end{array}\right]$$

- **Fork consistency.** Informally, if the service equivocates at some time t by presenting two different commitments to different sets of clients, it cannot forge a proof that merges the two different commitments. Formally, for any application-specific update policy F, any PPT adversary $\mathcal{A}$ and epoch t the following probability is negligible in λ:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F) \\ (\{\mathcal{C}_i, \pi_i, \mathcal{C}'_i, \pi'_i\}_{1 \le i < t},) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \forall i < t: \mathsf{VerifyUpdates}(\mathsf{pp}, \mathcal{C}_i, \mathcal{C}_{i+1}, \pi_{i+1}) = 1 \\ \forall i < t: \mathsf{VerifyUpdates}(\mathsf{pp}, \mathcal{C}'_i, \mathcal{C}'_{i+1}, \pi'_{i+1}) = 1 \\ \exists i < t: \mathcal{C}_i \neq \mathcal{C}'_i \\ \mathcal{C}_t = \mathcal{C}'_t \end{array}\right]$$

B. Formal Properties of a zkSNARK

To make our proofs self contained, we briefly recall SNARK-related definitions.

A zero-knowledge succinct non-interactive argument of knowledge (zkSNARK [15, 31]) for a circuit $\mathcal{R}$ has the following semantics

- $(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R})$: Returns prover parameters pp and verifier parameters vp used to produce and verify proofs respectively for circuit $\mathcal{R}$, where λ is the security parameter.
- $\pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathsf{x}, w)$: Takes as input IO x and secret witness w and returns a proof π that $\mathcal{R}(\mathsf{x}, w) = 1$.
- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{vp}, \mathsf{x}, \pi)$: Takes as input IO x and proof π and returns 1 if π attests that the prover knows secret w such that $\mathcal{R}(\mathsf{x}, w) = 1$.

and satisfies the following properties:

- **Completeness.** Informally, a verifier does not reject an honest proof. Formally, for IO x, and witness w such that $\mathcal{R}(\mathsf{x}, w) = 1$ the following probability is 1:

$$\Pr\left[\begin{array}{c} (\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R}) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathsf{x}, w) \\ \mathsf{Verify}(\mathsf{vp}, \mathsf{x}, \pi) = 1 \end{array}\right]$$

- **Soundness.** Informally, if there exists no valid witness, then the prover cannot produce an accepting proof. Formally, for circuit $\mathcal{R}$, and IO x, if there exists no w such that $\mathcal{R}(\mathsf{x}, w) = 1$, then for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, the following probability is $\mathtt{negl}(\lambda)$:

$$\Pr\left[\begin{array}{c} (\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R}) \\ \pi' \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{vp}, \mathsf{x}) \\ \mathsf{Verify}(\mathsf{vp}, \mathsf{x}, \pi') = 1 \end{array}\right]$$

- **Knowledge Soundness.** Informally, if a prover produces an accepting proof, then it must know a valid witness. Formally, for any circuit $\mathcal{R}$, and PPT adversary $\mathcal{A}$, there exists PPT $\mathcal{E}$ such that the following probability is $\mathtt{negl}(\lambda)$:

$$\Pr\left[\begin{array}{c} (\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R}) \\ (\pi', \mathsf{x}') \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{vp}, \rho) \\ w \leftarrow \mathcal{E}(\mathsf{pp}, \mathsf{vp}, \pi', \mathsf{x}', \rho) \\ \mathsf{Verify}(\mathsf{vp}, \mathsf{x}', \pi') = 1 \text{ and } \mathcal{R}(\mathsf{x}', w) \neq 1 \end{array}\right]$$

where ρ is the input randomness for $\mathcal{A}$.

- **Zero Knowledge.** Informally, the prover does not reveal any information about its witness in the proof.

instantiation	membership prover	membership proof size	membership verifier	non-membership/insert/update prover	non-membership/insert/update proof size	non-membership/insert/update verifier	SNARK friendly?
Merkle trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	✓
RSA accumulators	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	✗
Merkle-Patricia trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	✗
Merkle-AVL trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	✗
Indexed Merkle trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	✓

Figure 8: Comparison of asymptotic costs of Merkle proofs of membership, non-membership, inserts, and updates under different instantiations of cryptographic accumulators, where n denotes the number of items in the committed collection.

Formally, for circuit $\mathcal{R}$ and $(\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R})$, there exists a PPT simulator $\mathcal{S}$ such that for all PPT $\mathcal{V}^*$ with input randomness ρ, and IO x and witness w such that $\mathcal{R}(\mathsf{x}, w) = 1$, $\mathsf{Prove}(\mathsf{pp}, \mathsf{x}, w)$ is computationally indistinguishable from $\mathcal{S}(\mathsf{pp}, \mathsf{vp}, \mathsf{x}, \rho)$.

- **Succinctness.** For circuit $\mathcal{R}$ and for any proof π output by Prove, $|\pi|$ is sublinear in $|\mathcal{R}|$.

C. Proofs of Verdict's Properties

Verdict's update and lookup completeness follows in a straightforward manner from its construction. Below, we prove the soundness properties.

Lemma 2. *Verdict is a transparency dictionary that satisfies update knowledge soundness.*

Proof: For security parameter λ and for some application-specific update policy F, let $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F)$. Suppose that PPT adversary $\mathcal{A}$ on input pp and some randomness ρ outputs

$$\{C_i, \pi_i\}_{1 \le i \le t}, v, \pi_{\mathsf{lookup}}, \mathsf{label}$$

for some epoch t such that

$$\mathsf{VerifyUpdates}(\mathsf{pp}, \mathcal{C}_i, \mathcal{C}_{i+1}, \pi_{i+1}) = 1 \quad \text{for all } i < t \quad (5)$$

and

$$\mathsf{VerifyLookup}(\mathsf{pp}, \mathcal{C}_t, \mathsf{label}, v, \pi_{\mathsf{lookup}}) = 1. \quad (6)$$

We must show that there exists PPT extractor $\mathcal{E}$ that on input pp and ρ, outputs for some $k \le t$

$$U_1, \ldots, U_{k-1}$$

such that for $v_{i+1} \leftarrow F(v_i, U_i)$, where $v_1 \leftarrow \perp$, we have that $v = v_k$ with probability $1 - \mathtt{negl}(\lambda)$. Intuitively, this would mean that the extractor has been able to extract a valid sequence of updates that, starting from the empty dictionary, result in value v (thus proving the adversary's knowledge of such a sequence). To show this, we will first show that $\mathcal{E}$ can extract *some* sequence of requests by the knowledge soundness of the underlying SNARK. Next, we will show that the sequence of requests provided in π_{lookup} (and the final resulting value), is valid with respect to the application specific update policy. Finally, we will show that the requests extracted by $\mathcal{E}$ must be equal to the requests provided in π_{lookup} by the binding property of the indexed Merkle tree, thus showing that $\mathcal{E}$ has indeed extracted a valid sequence of requests.

We start by showing that $\mathcal{E}$ can extract some sequence of requests: Because Equation 5 holds by the knowledge soundness of the underlying SNARK, for label $H(\mathsf{label})$, the adversary knows a sequence of values $U'_1, \ldots, U'_{k'-1}$ for some $k' < t$ such that

$$h'_{i+1} \leftarrow H(U'_i, h'_i)$$

for all $i < k' - 1$ (where $h_1 = \perp$) such that the indexed Merkle tree with root $\mathcal{C}_t$ contains $h'_{k'}$ under $H(\mathsf{label})$. Thus, $\mathcal{E}$ can extract these U'_i for all $i < t$.

Next, we show that the sequence of requests provided in π_{lookup} must be valid: Because Equation 6 holds, the included hashchain must be well-formed with probability $1 - \mathtt{negl}(\lambda)$. In particular, the hashchain contains values $U_1, \ldots, U_{k-1}$ for some $k < t$ such that

$$h_{i+1} \leftarrow H(U_i, h_i)$$

for all $i < k-1$ such that $h_1 = \perp$ and the indexed Merkle tree with root $\mathcal{C}_t$ contains h_k under label $H(\mathsf{label})$. Also, because Equation 6 holds, we must have that

$$v_{i+1} \leftarrow F(U_i, v_i)$$

for all $i < k$, where $v_1 = \perp$, and that

$$v_k = v. \quad (7)$$

Now, we show that the requests extracted by $\mathcal{E}$ must be equal to the requests provided in π_{lookup}: Because Equation 5 holds, the insert invariant of the indexed Merkle tree must hold with probability $1 - \mathtt{negl}(\lambda)$. In particular, the label $H(\mathsf{label})$ can only be inserted once under a single leaf node. But from the above reasoning we know that the indexed Merkle tree with root $\mathcal{C}_t$ must contain h_k under label $H(\mathsf{label})$, and must contain $h'_{k'}$ under label $H(\mathsf{label})$. By the binding property of indexed Merkle trees, this implies that $h_k = h'_{k'}$. Therefore, by the binding property of H, we must have that $k = k'$ and moreover that $U'_i = U_i$ for all $i < k$ with probability $1 - \mathtt{negl}(\lambda)$. Thus we must have

$$v_{i+1} = F(U'_i, v_i)$$

for all $i < k$. From Equation 7, we have that $v_k = v$. Thus, the material extracted by $\mathcal{E}$ is valid with probability $1 - \mathtt{negl}(\lambda)$. ■

Lemma 3. *Verdict is a transparency dictionary that satisfies lookup soundness.*

Proof: For security parameter λ and for some application-specific update policy F, let $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, F)$. Suppose that PPT adversary $\mathcal{A}$ on input pp outputs

$$\{\mathcal{C}_i, \pi_i\}_{1 \le i \le t}, v, v', \pi_{\mathsf{lookup}}, \pi'_{\mathsf{lookup}}, \mathsf{label} \tag{8}$$

such that

$$\mathsf{VerifyUpdates}(\mathsf{pp}, \mathcal{C}_i, \mathcal{C}_{i+1}, \pi_{i+1}) = 1 \quad \text{for all } i < t \tag{9}$$

and

$$\mathsf{VerifyLookup}(\mathsf{pp}, \mathcal{C}_t, v, \mathsf{label}, \pi_{\mathsf{lookup}}) = 1 \tag{10}$$
$$\mathsf{VerifyLookup}(\mathsf{pp}, \mathcal{C}_t, v', \mathsf{label}, \pi'_{\mathsf{lookup}}) = 1. \tag{11}$$

We must show that $v' = v$ with probability $1 - \mathtt{negl}(\lambda)$. To do so, we will show that the sequence of requests provided in π_{lookup} must be equal to the sequence of requests provided in π'_{lookup} due to the binding property of indexed Merkle trees.

We first consider the sequence of requests provided in π_{lookup}: Because Equation 10 holds, the hashchain included in π_{lookup} contains requests $U_1, \ldots, U_{k-1}$ for some $k < t$ such that

$$h_{i+1} \leftarrow H(U_i, h_i)$$
$$v_{i+1} \leftarrow F(U_i, v_i)$$

for all $i < k - 1$ such that

$$h_1 = \perp \tag{12}$$
$$v_1 = \perp \tag{13}$$
$$v_k = v \tag{14}$$

and that the indexed Merkle tree with root $\mathcal{C}_t$ contains h_k under label $H(\mathsf{label})$.

Symmetrically, we consider the sequence of requests provided in π'_{lookup}: Because Equation 11 holds, the hashchain included in π'_{lookup} contains requests $U'_1, \ldots, U'_{k-1}$ for some $k' < t$ such that

$$h'_{i+1} \leftarrow H(U'_i, h'_i)$$
$$v'_{i+1} \leftarrow F(U'_i, v'_i)$$

for all $i < k' - 1$ such that

$$h'_1 = \perp \tag{15}$$
$$v'_1 = \perp \tag{16}$$
$$v'_{k'} = v' \tag{17}$$

and that the indexed Merkle tree with root $\mathcal{C}_t$ contains $h'_{k'}$ under label $H(\mathsf{label})$.

Now, we show that the requests provided in π_{lookup} must be equal to the requests provided in π'_{lookup}: Because Equation 9 holds, the insert invariant of the indexed Merkle tree must hold with probability $1 - \mathtt{negl}(\lambda)$. In particular, the label $H(\mathsf{label})$ can only be inserted once under a single leaf node. But from the above reasoning, we know that the indexed Merkle tree with root $\mathcal{C}_t$ must contain h_k under label $H(\mathsf{label})$, and must contain $h'_{k'}$ under label $H(\mathsf{label})$. By the binding property of indexed Merkle trees, this implies that $h_k = h'_{k'}$ with probability $1 - \mathtt{negl}(\lambda)$. Therefore, by the binding property of H, we must have that $k = k'$ and moreover that $U'_i = U_i$ for all $i < k$ with probability $1 - \mathtt{negl}(\lambda)$. By Equations 14 and 17, this implies that $v = v'$. ∎

a) Achieving Fork Consistency: We have thus far shown that Verdict satisfies lookup soundness and update soundness. These do not prevent the service from presenting a stale view of its current state (e.g., an older state that does not include recent updates).

To mitigate this issue, the service can publish its commitments (and proofs) to a public bulletin board (e.g., a blockchain). Alternatively, clients can detect inconsistent views via a peer-to-peer gossip protocol. In the latter case, the service cannot fork the clients' views indefinitely. But, it does not prevent the service from quietly forking the clients views for a brief period and later merging the views by replaying updates from both forks. We prevent this as follows: We require the service to include in its published commitment $\mathcal{C}_i$ not only the root of the indexed Merkle tree but also a hash h_i that is $\perp$ for $i = 0$ and $H(\mathcal{C}_{i-1})$ otherwise where H is a collision resistant hash function. We also extend $\mathsf{VerifyUpdates}$ to check that $\mathcal{C}_i.h_i = H(\mathcal{C}_{i-1})$;

Lemma 4. *Verdict is a transparency dictionary that satisfies fork consistency.*

Proof: Denote $\mathcal{C}_i.h_i$ as h_i and $\mathcal{C}'_i.h_i$ as h'_i Since for all $j < t$,

$$\mathsf{VerifyUpdates}(pp, \mathcal{C}_j, \mathcal{C}_{j+1}, \pi_j) = 1 \tag{18}$$

and

$$\mathsf{VerifyUpdates}(pp, \mathcal{C}'_j, \mathcal{C}_{j+1}, \pi_{j+1}) = 1 \tag{19}$$

it is sufficient to show that in order for the adversary to produce $\mathcal{C}_t = \mathcal{C}'_t$ and $\mathcal{C}_i \neq \mathcal{C}'_i$ for some $i < t$, it needs to find a collision for the hash function H.

Equations 18 and 19 imply that for all $j \le t$,

$$h_j = H(\mathcal{C}_{j-1})$$

and

$$h'_j = H(\mathcal{C}'_{j-1})$$

. If there exists some $i < t$ for which $\mathcal{C}_i \neq \mathcal{C}'_i$ but $\mathcal{C}_t = \mathcal{C}'_t$ there exists some $k \in (i, t)$ s.t. $\mathcal{C}_k \neq \mathcal{C}'_k$ and $\mathcal{C}_{k+1} = \mathcal{C}'_{k+1}$. The latter implies that $h_{k+1} = h'_{k+1}$ and, thus, $H(\mathcal{C}_k) = H(\mathcal{C}'_k)$ as requested.

∎