Compact Certificates of Collective Knowledge

Silvio Micali[*][†], Leonid Reyzin[*][‡], Georgios Vlachos[§], Riad S. Wahby[*][¶], Nickolai Zeldovich[*][†]
[*]Algorand
[†]MIT CSAIL
[‡]Boston University
[§]Axelar and University of Waterloo
[¶]Stanford
silvio@algorand.com, reyzin@cs.bu.edu, georgios@axelar.network, rsw@cs.stanford.edu, nickolai@csail.mit.edu

***Abstract*—We introduce *compact certificate schemes*, which allow any party to take a large number of signatures on a message M, by many signers of different weights, and compress them to a much shorter certificate. This certificate convinces the verifiers that signers with sufficient total weight signed M, even though the verifier will not see—let alone verify—all of the signatures. Thus, for example, a compact certificate can be used to prove that parties who jointly have a sufficient total account balance have attested to a given block in a blockchain.**

After defining compact certificates, we demonstrate an efficient compact certificate scheme. We then show how to implement such a scheme in a decentralized setting over an unreliable network and in the presence of adversarial parties who wish to disrupt certificate creation. Our evaluation shows that compact certificates are 50–280× smaller and 300–4000× cheaper to verify than a natural baseline approach.

I. Introduction

Suppose many people wish to attest to having witnessed an important event. They could each sign an attestation message M that has the relevant information about the event. The resulting collection of signatures will constitute a certificate of this event. This certificate, however, will be quite large and will take a long time to verify. Our goal is to reduce the size and verification time by combining multiple signatures into a single *compact certificate*. Moreover, we want to ensure that, even though each attestor has a signing public key, the verifier will need access only to a small subset of these keys.

We generalize the above problem in two ways. First, we wish to go beyond public keys and signatures to any NP statements and their witnesses (provided as attestations); signatures are then just a special case, with each NP statement comprising a public key and the message M. Second, our attestors are not all equal; rather, they have assigned weights. Our goal is to show that attestors with sufficient total weight have provided witnesses to their corresponding NP statements.

Our first contribution is to define compact certificate schemes in terms of their functionality and security in Section III. Crucially, our definition ensures that the verifiers never need to receive or store a linear amount of information: they need neither all the NP statements nor all the weights, but only a commitment to this information.

We then construct the first ever compact certificate scheme in Section IV. Using our scheme, anyone in possession of the signatures (more generally, NP witnesses) can reduce the size of the attestations and the verification time from linear to logarithmic in the total number of public keys (more generally, NP statements). We prove the security of our scheme in Section V-A and analyze its concrete security parameters in Section V-B.

Compact certificates do not assume that all attestors have provided their attestations. Some of the attestors may not have witnessed the event, or may be off-line or dishonest. A compact certificate can be formed as long as any set of attestors with sufficient weight have provided correct attestations.

After defining and constructing compact certificates, we consider their use in situations when some attestors are honest and some are adversarial. We consider two possible security goals in such situations, both assuming some bound on the fraction of adversarial attestors. The first, easier to achieve, goal, is to produce a compact certificate guaranteeing that at least one honest attestor testified to a given statement. However, even honest attestors may have different views of what actually happened. Thus, the second, more difficult, goal, is to produce an *incontrovertible* certificate, i.e., to produce a compact certificate guaranteeing that a *majority* of honest attestors testified to the statement (and thus, if honest attestors are assumed not to contradict themselves, it is computationally infeasible to produce an incontrovertible certificate for a contradictory statement, because any contradictory statement would get only a minority of attestors). We analyze these goals in Section VI.

A. Performance Evaluation

We implement our compact certificate scheme and evaluate its performance experimentally in Section VII. For one million attestors and 128-bit security, the cost of creating the certificate (excluding the cost of verifying the attestations themselves) is about 6 seconds. Certificate size ranges from 120 kBytes to 650 kBytes and verification cost ranges from 9 msec to 72 msec, depending on the fraction of cooperating attestors. Compared to the naïve approach of retrieving and verifying a large fraction of all attestors' signatures, compact certificates are roughly 50–280× smaller and take roughly 300–4000× less time to verify.

B. Blockchain Application

We also explore a specific application of compact certificates: certifying a recent state in a blockchain that uses stake-weighted voting (e.g., [13, 37, 38, 56]). The correctness of a typical blockchain is difficult to verify for someone who is not participating in real time: the verifier must start with the genesis block and proceed forward one block at a time, checking that each block was properly added according to the rules of the specific blockchain. Instead, using a compact certificate scheme, we can have blockchain participants sign a recent block after it has been decided upon, and collect the signatures to form a compact certificate for this block. The weight of a signer, in this case, could be the signer's account balance or stake in the blockchain. This process is meant to supplement, not supplant, existing block consensus mechanisms; thus, we can choose to have it performed only for every Nth block for some N.

Assuming the adversary controls less total stake than the weight of the signers proven by the compact certificate, the verifier can be assured that the block was signed by at least one honest participant and is thus correct. To determine the stake, the verifier could use some pre-specified earlier block—for example, the most recent block that has a compact certificate. This would allow much faster verification of the whole blockchain, starting from the genesis block, because one would need to verify only a $1/N$ fraction of blocks, and each block verification would use less CPU time and bandwidth, because of the compact certificate attesting to its correctness.

The blockchain application presents additional challenges beyond the construction of the compact certificate in a standalone, centralized setting. A design needs to address how frequently to certify blocks, how to determine who will collect the signatures, how to determine if enough signatures have been collected, how the signatures will be retransmitted in case of network errors, and how retransmission will stop without creating vulnerabilities to denial of service of attacks. We address these challenges in Section VIII.

C. Related Work

Signatures: Aggregate signatures (which compress signatures of multiple signers on different messages) [1, 22, 23, 54, 55, 72, 79], multisignatures (which compress signatures of multiple signers on the same message) [4–6, 18, 21, 25, 30, 32, 43, 44, 59, 62, 65, 71, 74, 80, 82], threshold signatures (which allow multiple signers to coordinate producing a single signature) [18, 21, 29, 39–42, 50–53, 64, 70, 85, 87] and designs that combine their aspects (e.g., [2, 66]) can help reduce signature size. However, all these approaches require considerably more coordination than compact certificates.

First, consider aggregate signatures and multisignatures. These schemes require special-purpose designs, in contrast to compact certificates, which work with any underlying signature scheme (and, more generally, with any NP statement). Moreover, the verifier of aggregate signatures and multisignatures needs to know all the public keys that participated in the signing process, making sublinear-size certificates and/or sublinear-time verification impossible.

Threshold signatures apply secure multi-party computation to key generation and signing, and thus in principle work with any signature scheme. In contrast to compact certificates, however, they require the signers to coordinate (exchange messages) during key generation and, depending on the scheme, also during computation. Moreover, a compact certificate scheme can be used regardless of the number of attestors who participated, while in a threshold signature scheme, the minimum required number of signers is set at key generation time and cannot be arbitrarily changed.

Finally, we emphasize that a compact certificate scheme is designed to handle attestors of varying weights—a feature generally not present in the aforementioned signature schemes. And, of course, compact certificates can handle any NP statement, not just a signature verification predicate.

Succinct arguments: Several lines of work on succinct non-interactive arguments of knowledge (SNARKs) [3, 8, 9, 11, 12, 16, 17, 24, 34–36, 47–49, 58, 73, 83, 90, 91] allow a prover to construct a short proof for any NP statement. In principle a SNARK could be used to generate a compact certificate, in essence by proving a statement that verifies some or all attestors' signatures. In practice, the cost of using this approach for a large number of attestors is prohibitive: for commonly used SNARKs, 10 ms is a conservative estimate for the cost of proving validity of one attestor's signature [81, §4.4 and Fig. 3], meaning that constructing a compact certificate for one million users would take well over an hour. Under the same conditions, our scheme's proving cost is about 6 seconds (§VII).

A better approach is to combine SNARKs and compact certificates: rather than proving validity of attestors' signatures directly, prove knowledge of a valid compact certificate. This approach would reduce the SNARK's proving costs by orders of magnitude (because the compact certificate verifier only checks a small number of attestors' signatures) while potentially reducing certificate size and verification cost. We leave evaluating this approach to future work.

II. Background

We assume familiarity with Merkle trees [75] and the cryptographic modeling of hash functions as random oracles [7]. All hash functions used in this paper—including those used to build a Merkle tree—will be modeled as random oracles (a reader interested in a detailed formalization of Merkle trees built with random oracles may see [10, §2.2 and §3.1]). We will denote the output length of a hash function by λ.

A. Vector Commitments

Vector commitments (introduced in [31, 68]) provide a way to commit to a list of values and then efficiently reveal only a subset of those values. These commitments are binding, but not hiding. Note that vector commitments with the properties listed below can be provided by Merkle trees [75] (see Appendix A), by algebraic techniques [19, 28, 31, 33, 57, 63, 67, 68, 88, 89], or by polynomial commitments (introduced in [60]; see, e.g., [20, 27] for an overview) adapted to vectors per [57, Appx. C].

Vector Commitment Functionality: A vector commitment consists of three algorithms: $\mathtt{Commit}(A)$ takes a list A of values and produces a short output C; $\mathtt{ComProve}(i, A)$ produces a proof π_i; and $\mathtt{ComVerify}(C, i, v, \pi_i)$ outputs T ("true" or "accept") if $A[i] = v$ and C and π_i were correctly produced via Commit and ComProve, respectively. Since there is no hiding property, we assume these algorithms are deterministic.

Vector Commitment Security: In our application, we need vector commitment security to hold only when the committer is trusted (which is a weaker security goal than when C can be computed adversarially). We thus assume that (under appropriate cryptographic assumptions) vector commitments provide the following security property: if C was produced correctly via $\mathtt{Commit}(A)$ for some A, then no adversary running in time t on input A has probability greater than $\mathbf{Insec}^{\mathrm{com}}(t)$ of outputting (i, v, π_i^*) such $A[i] \neq v$ but $\mathtt{ComVerify}(C, i, v, \pi_i^*) =$ T.

B. Non-interactive random oracle proofs of knowledge

As defined in [10, §2.3], a non-interactive random oracle proof of knowledge (NIROPK) consists of two algorithms, a prover $\mathbb{P}$ and a verifier $\mathbb{V}$, which both have access to the same oracle $\rho : \{0,1\}^* \to \{0,1\}^\lambda$, chosen uniformly at random. Let $\mathcal{R}$ be an NP relation with inputs $\mathbb{x}$ and witnesses $\mathbb{w}$. The functionality of a NIROPK is as follows: $\mathbb{P}(\mathbb{x}, \mathbb{w})$ outputs a proof π and $\mathbb{V}(\mathbb{x}, \pi)$ outputs T ("true" or "accept") or F ("false" or "reject").

The security notion is that of a proof of knowledge. It is defined by introducing a probabilistic polynomial time *knowledge extractor* algorithm $\mathbb{E}$ who extracts witnesses from an adversarial prover $\tilde{\mathbb{P}}$. Extractor $\mathbb{E}$ is allowed to run $\tilde{\mathbb{P}}$ only as a black box (denoted $\mathbb{E}^{\tilde{\mathbb{P}}}$), but may respond to random oracle queries of $\tilde{\mathbb{P}}$ however $\mathbb{E}$ chooses (i.e., to "program" the random oracle).

Definition 1 ([10])**.** A pair $(\mathbb{P}, \mathbb{V})$ is a NIROPK with knowledge error e for $\mathcal{R}$ if it satisfies the following:

- *Completeness*: if $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, then $\mathbb{V}(\mathbb{x}, \mathbb{P}(\mathbb{x}, \mathbb{w})) = \mathrm{T}$.
- *Proof of knowledge*: there exists a knowledge extractor $\mathbb{E}$ such that, for any $\mathbb{x}$ and adversary $\tilde{\mathbb{P}}$ who with probability δ (computed over the random choice of ρ) outputs π acceptable to $\mathbb{V}(\mathbb{x}, \pi)$, $\mathbb{E}^{\tilde{\mathbb{P}}}$ produces $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ with probability at least $\delta - e$.

III. Defining Compact Certificate Schemes

In this section, we define the syntax and security of compact certificates schemes. Our definition is inspired by the definition of a NIROPK system (Section II-B). We will, however, change how the verifier obtains inputs (in contrast to NIROPK, some inputs will be provided committed, and some will be provided by the prover).

Let $\mathcal{R}^{\mathrm{compcert}}$ be an NP relation with two-part inputs (input, globalinput) and witnesses witness (for example, for the signatures application describe in §I, input is an attestor's public key, globalinput is the message all the attestors sign, and witness is the attestor's signature). By definition of NP relations, there is a polynomial-time algorithm that checks if ((input, globalinput), witness) $\in \mathcal{R}^{\mathrm{compcert}}$ (for example, verifies the signatures).

An attestor is a pair (input, weight); let attestors be the list of all potential attestors.

A compact certificate scheme for $\mathcal{R}^{\mathrm{compcert}}$ has two participants, a prover $\mathbb{P}$ and a verifier $\mathbb{V}$, who both have access to the same oracle $\rho : \{0,1\}^* \to \{0,1\}^\lambda$, chosen uniformly at random. We assume $\mathbb{P}$ knows the list attestors, and $\mathbb{V}$ knows the vector commitment $C_{\mathtt{attestors}} = \mathtt{Commit}(\mathtt{attestors})$.

Let witnesses be a list of the same length as attestors; for some (but possibly not all) i, $((\mathtt{attestors}[i].\mathtt{input}, \mathtt{globalinput}), \mathtt{witnesses}[i]) \in \mathcal{R}^{\mathrm{compcert}}$. Such i are called *valid*. Let provenWeight be a number.

We will say that the tuple

(attestors, globalinput, witnesses, provenWeight)

is *sufficiently weighty* if

$$\sum_{i \text{ is valid}} \mathtt{attestors}[i].\mathtt{weight} > \mathtt{provenWeight}\,.$$

Given such a sufficiently weighty tuple, $\mathbb{P}$ produces a certificate cert. On input $(C_{\mathtt{attestors}}, \mathtt{globalinput}, \mathtt{provenWeight}, \mathtt{cert})$, $\mathbb{V}$ outputs T ("true" or "accept") or F ("false" or "reject"), to indicate whether it can confirm that the tuple is indeed sufficiently weighty. Note that $\mathbb{V}$ assumes that $C_{\mathtt{attestors}}$ was correctly generated as a vector commitment to attestors; its remaining inputs may be adversarial.

Definition 2. A pair $(\mathbb{P}, \mathbb{V})$ constitutes a compact certificate scheme with knowledge error e if it satisfies the following:

- *Compact Completeness.* If $\mathbb{x} = (\mathtt{attestors}, \mathtt{globalinput}, \mathtt{witnesses}, \mathtt{provenWeight})$ is sufficiently weighty, then for $\mathtt{cert} = \mathbb{P}(\mathbb{x})$ and $C_{\mathtt{attestors}} = \mathtt{Commit}(\mathtt{attestors})$,

  $$\mathbb{V}(C_{\mathtt{attestors}}, \mathtt{globalinput}, \mathtt{provenWeight}, \mathtt{cert}) = \mathrm{T}\,.$$

  Moreover, the length of cert depends at most polylogarithmically on the length of the attestors list.

- *Proof of Knowledge.* There exists a knowledge extractor $\mathbb{E}$ (as defined in Section II-B) such that, for any (attestors, globalinput, provenWeight), for $C_{\mathtt{attestors}} = \mathtt{Commit}(\mathtt{attestors})$, and for any adversary $\tilde{\mathbb{P}}$ who with probability δ (computed over the random choice of ρ) outputs cert such that

  $$\mathbb{V}(C_{\mathtt{attestors}}, \mathtt{globalinput}, \mathtt{provenWeight}, \mathtt{cert}) = \mathrm{T}\,,$$

  $\mathbb{E}^{\tilde{\mathbb{P}}}$ produces witnesses such that

  (attestors, globalinput, witnesses, provenWeight)

  is sufficiently weighty, with probability at least $\delta - e$.

The knowledge error may be a function of the hash function output length λ and the adversarial running time and number of random oracle queries.

Note that multiple witnesses for a single entry in the attestors list (e.g., multiple signatures by the same signer) will not count multiple times, because the definition of sufficiently weighty given above counts the weight of each attestor at most once.

IV. Our compact certificate scheme P_{compcert}

We now give a concrete instantiation of a compact certificate scheme (Section III), which we call P_{compcert}. For concreteness and ease of exposition, we will describe our scheme for the language of digital signatures. That is, `input` is a public key, `attestors` is a list of pairs (`pk`, `weight`), `globalinput` is a message M, `witness` is a signature of `pk` on M, and the compact certificate establishes that the prover knows a sufficiently weighty set of signatures on M. The case of other NP languages is the same, mutatis mutandis.

The first idea of our scheme is to use techniques due to Kilian [61] and Micali [76, 77]. In contrast to the CS Proofs approach, which puts elements of a probabilistically checkable proof in the leaves of a Merkle tree, in our scheme the prover will associate each element of `attestors` (and the corresponding signature, if known) with a leaf in a Merkle tree. Applying a hash function (modeled as a random oracle) to the root of this tree, the prover will determine which leaves to reveal. The certificate `cert` will consist of the Merkle tree root, the revealed leaves with their authenticating paths in the Merkle tree (to convey the relevant signatures to the verifier), and vector commitment proofs produced by `ComProve` to convey the relevant public keys and weights.

This idea is insufficient by itself, however: we have not described *how* the hash function picks which leaves to reveal. The problem with picking leaves at random is that there could be many low-weight leaves, and revealing those will do little to convince the verifier; revealing leaves without signatures is also unhelpful. The key ingredient of our scheme is a mechanism for choosing which leaves to reveal that chooses *among only the attestors that produced signatures* and *in proportion to their relative weight*. Importantly, this mechanism has very low cost and cannot be gamed by the adversary.

At a high level, this mechanism works as follows. Let `signedWeight` represent the total weight of all attestors who contribute an attestation. We will partition the range from to 0 to `signedWeight` into subranges; there will be one subrange for each contributing attestor, with the length corresponding to the attestor's weight. The endpoints of each participating attestor's subrange will be committed in the corresponding Merkle leaf; subranges for attestors who contribute no signature will be empty. The hash function, when applied to the Merkle root, will determine a point in the range from 0 to `signedWeight`, and the prover will have to reveal the leaf whose subrange contains that point. Given sufficiently many such reveals, the verifier will be convinced, with high certainty, that a large fraction of the range is covered by valid leaves, because each random choice made by the hash function falls into a covered subrange. This implies (by the security of the Merkle tree) that the prover must know signatures for attestors corresponding to a large fraction of `signedWeight`.

A surprising feature of this approach is that the verifier does not need to check the correctness of the subranges claimed by the prover—only that each individual revealed subrange is of the correct length and equal to the weight of its attestor (and, of course, that the attestor's signature is valid). An adversarial prover can arrange subranges however it pleases; in particular, making subranges overlap only makes the adversary's life harder, because it becomes more difficult to cover the entire range given the valid signatures in the adversary's possession.

We are now ready to proceed with the details of the protocol. We will assume $\texttt{Hash}_{\texttt{range}}$ outputs (nearly) uniform values between 0 and `range`, excluding `range` itself (formally, we need to have a fresh random oracle for each value of `range`, which can be accomplished by encoding `range` unambiguously into the hash's input). We will assume that $\mathbb{V}$ wants to achieve knowledge error approximately 2^{-k} for some k, and that the adversary runs in time at most t and makes at most $Q = 2^q$ random oracle queries. These parameters determine how many Merkle leaves `cert` will contain (see Section V-B).

A. $\mathbb{P}$: *Creating the certificate*

A prover $\mathbb{P}$ who wishes to prove that elements of `attestors` with total weight at least `provenWeight` have signed a message M runs the following algorithm:

1) Set `signersList` to empty and `signedWeight` to 0.

2) Obtain signatures of attestors until `signedWeight` > `provenWeight`, where `signedWeight` is computed as described immediately below.
For each signature obtained,
 - Find the location i of the attestor who created it in the `attestors` list and verify that $i \notin$ `signersList` (otherwise reject this signature as a duplicate and continue).
 - Verify the signature under `attestors`$[i]$`.pk` (this is done to prevent a denial of service attack, in which a bad signature could cause the prover to create an invalid certificate—see §VIII-A). If verification succeeds, set

 $$\texttt{signedWeight} = \texttt{signedWeight} + \texttt{attestors}[i].\texttt{weight}$$

 and add i to `signersList`. Otherwise, reject this signature.

 For reasons discussed below, higher `signedWeight` will result in a smaller compact certificate, so it's good to obtain more. In fact, as discussed in Section IV-B, some verifiers may choose to reject certificates that are too long, in which case the prover will need to increase `signedWeight` (by obtaining more signatures).

3) Initialize a list `sigs` having the same length as `attestors`. Each entry in `sigs` consists of a triple (`sig`, `L`, `R`), which is computed as follows. For each i starting with 0, first set

 $$\texttt{sigs}[i].\texttt{L} = \texttt{sigs}[i-1].\texttt{R}$$

 (with the base case $\texttt{sigs}[0].\texttt{L} = 0$). Next, if i is in `signersList`, set

 $$\texttt{sigs}[i].\texttt{R} = \texttt{sigs}[i].\texttt{L} + \texttt{attestors}[i].\texttt{weight}$$

 and let `sigs`$[i]$`.sig` be the signature on M under `attestors`$[i]$`.pk` that the prover obtained in the previous step. Otherwise, set $\texttt{sigs}[i].\texttt{R} = \texttt{sigs}[i].\texttt{L}$ and leave `sigs`$[i]$`.sig` empty.

In addition, we define (but do not store)

$$\texttt{sigs}[i].\texttt{weight} \overset{\text{def}}{=} \texttt{sigs}[i].\texttt{R} - \texttt{sigs}[i].\texttt{L}\,.$$

Notice that the R value of the last entry in sigs will be equal to signedWeight.

4) Compute $\texttt{Root}_{\texttt{sigs}}$ as the Merkle root of a Merkle tree whose leaves are the entries of sigs.

5) Create a function IntToInd that allows efficient lookups from a value coin, such that $0 \le \texttt{coin} < \texttt{signedWeight}$, to the unique index i such that $\texttt{sigs}[i].\texttt{L} \le \texttt{coin} < \texttt{sigs}[i].\texttt{R}$. (Note that this function can be easily implemented via a binary search on the L values of the sigs array.) We will denote such i via IntToInd(coin).

6) Create a map T as follows. First, define

$$\texttt{numReveals} = \left\lceil \frac{k+q}{\log_2 \left(\texttt{signedWeight}/\texttt{provenWeight}\right)} \right\rceil. \quad (1)$$

Then, for $j \in \{0, 1, \ldots, \texttt{numReveals} - 1\}$, let

$$\texttt{Hin}_j = (j, \texttt{Root}_{\texttt{sigs}}, \texttt{provenWeight}, M, C_{\texttt{attestors}}),$$
$$\texttt{coin}_j = \texttt{Hash}_{\texttt{signedWeight}}(\texttt{Hin}_j), \qquad \text{and}$$
$$i_j = \texttt{IntToInd}(\texttt{coin}_j)\,.$$

If $\texttt{T}[i_j]$ is not yet defined, define $\texttt{T}[i_j]$ to consist of the four-tuple (s, π_s, p, π_p) containing:

- the tuple $s = \texttt{sigs}[i_j]$ (without the R value),
- the Merkle authenticating path π_s to the i_j^{th} leaf,
- $p = \texttt{attestors}[i_j]$, and
- $\pi_p = \texttt{ComProve}(i_j, \texttt{attestors})$.

The resulting compact certificate cert consists of $\texttt{Root}_{\texttt{sigs}}$, signedWeight, and the map T, which has at most numReveals entries, but will have fewer if different iterations of Step 6 select the same i value (see Figure 7, Section VII).

B. $\mathbb{V}$: Verifying the certificate

The verifier $\mathbb{V}$ knows $C_{\texttt{attestors}} = \texttt{Commit}(\texttt{attestors})$, and receives the message M, the value provenWeight, and the compact certificate cert consisting of $\texttt{Root}_{\texttt{sigs}}$, signedWeight, and a map T with up to numReveals entries, each containing the four-tuple (s, π_s, p, π_p), where numReveals is defined in Equation (1) (Section IV-A).

If $\texttt{signedWeight} \le \texttt{provenWeight}$, then $\mathbb{V}$ outputs False. ($\mathbb{V}$ may choose to require a higher signedWeight in order to avoid having to verify certificates that are too long, for example, to protect itself against having to do too much work; this may also be accomplished simply by limiting the maximum size of the map T that $\mathbb{V}$ will accept.) Otherwise, for each entry i such that $\texttt{T}[i]$ is defined (as (s, π_s, p, π_p)), $\mathbb{V}$ performs the following steps to validate it:

- check that π_s is the correct authenticating path for the i^{th} leaf value s with respect to $\texttt{Root}_{\texttt{sigs}}$;
- check that $\texttt{ComVerify}(C_{\texttt{attestors}}, i, p, \pi_p) = \text{T}$; and
- check that $s.\texttt{sig}$ is a valid signature on M under $p.\texttt{pk}$.

If any of the above checks fails, $\mathbb{V}$ outputs False. Otherwise, for $j \in \{0, 1, \ldots, \texttt{numReveals} - 1\}$, $\mathbb{V}$ computes

$$\texttt{Hin}_j = (j, \texttt{Root}_{\texttt{sigs}}, \texttt{provenWeight}, M, C_{\texttt{attestors}}) \quad \text{and}$$
$$\texttt{coin}_j = \texttt{Hash}_{\texttt{signedWeight}}(\texttt{Hin}_j),$$

then checks that there exists i such that $\texttt{T}[i]$ is defined and is equal to (s, π_s, p, π_p) with $s.\texttt{L} \le \texttt{coin}_j < s.\texttt{L} + p.\texttt{weight}$. If no such i exists, then $\mathbb{V}$ outputs False.

If all of the above checks pass, then $\mathbb{V}$ outputs True. Otherwise, $\mathbb{V}$ outputs False.

C. Optimizations

- To save space and reduce the cost of computing $\texttt{Root}_{\texttt{sigs}}$, the entry $\texttt{sigs}[i]$ may be left entirely empty for $i \notin \texttt{signersList}$, and the R value of each entry in sigs need not be stored (since it equals the L value of the next entry).
- Computing numReveals precisely in the prover and verifier algorithms requires high-precision arithmetic, which may be slow and difficult to implement. Instead, we propose (in Appendix B) and implement (in Section VII) an approximate calculation of numReveals.
- Combining multiple Merkle paths into a single subtree will save bandwidth, because of overlapping entries. Moreover, because higher-weight entries in the sigs and attestors lists are more likely to be revealed, sorting attestors by weight before committing to it will likely provide more overlap in Merkle paths and thus will reduce the total proof size. We implement this optimization in Section VII.
- Aggregatable vector commitments (see [57] and references therein) allow one to combine multiple proofs π_p into one, reducing the size of the certificate (we do not implement this optimization, because it comes at a considerable computational cost; instead, we use a Merkle tree for $C_{\texttt{attestors}}$).

V. Security

In this section, we first prove security of the P_{compcert} scheme given in Section IV, and then discuss concrete parameter choices.

A. Security Proof

The noninteractive protocol P_{compcert} defined in Section IV is essentially the result of applying Merkle Trees [75] and the Fiat-Shamir [45] transform (similarly to Micali's CS Proofs [77]) to the interactive protocol $P_{\text{interactive}}$ described in Figure 1. Security intuition is provided by Lemma 1. The rest is technicalities.

Theorem 1. *The protocol P_{compcert} is a compact certificate system with knowledge error*

$$e < Q \cdot \left(\frac{\texttt{provenWeight}}{\texttt{signedWeight}}\right)^{\texttt{numReveals}} + \frac{1}{2} \cdot \frac{Q^2}{2^\lambda} + \mathbf{Insec}^{\text{com}}(t),$$

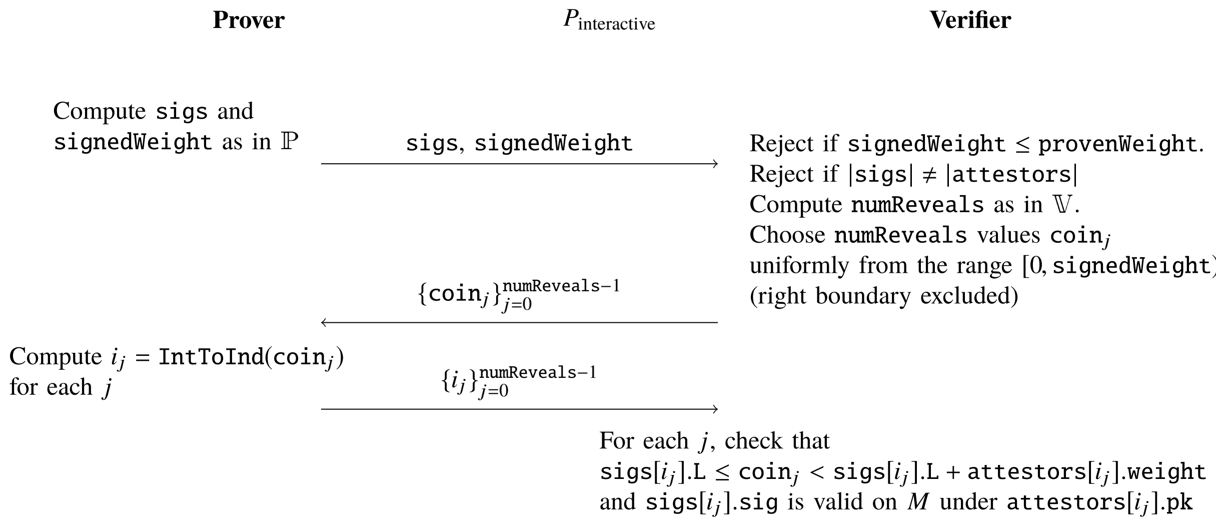
Prover
$P_{\text{interactive}}$
Verifier
Compute `sigs` and `signedWeight` as in $\mathbb{P}$
`sigs`, `signedWeight`
Reject if `signedWeight` $\leq$ `provenWeight`.
Reject if $|$`sigs`$| \neq |$`attestors`$|$
Compute `numReveals` as in $\mathbb{V}$.
Choose `numReveals` values $\texttt{coin}_j$ uniformly from the range $[0, \texttt{signedWeight})$ (right boundary excluded)
$\{\texttt{coin}_j\}_{j=0}^{\texttt{numReveals}-1}$
Compute $i_j = \texttt{IntToInd}(\texttt{coin}_j)$ for each j
$\{i_j\}_{j=0}^{\texttt{numReveals}-1}$
For each j, check that $\texttt{sigs}[i_j].\texttt{L} \leq \texttt{coin}_j < \texttt{sigs}[i_j].\texttt{L} + \texttt{attestors}[i_j].\texttt{weight}$ and $\texttt{sigs}[i_j].\texttt{sig}$ is valid on M under $\texttt{attestors}[i_j].\texttt{pk}$

Fig. 1. Protocol $P_{\text{interactive}}$ (§V-A). Protocol P_{compcert} (§IV) is essentially the result of applying Merkle Trees and the Fiat-Shamir transform to $P_{\text{interactive}}$.

where λ is the output length of the hash function used in the Merkle tree, $Q = 2^q$ is the number of random oracle queries[1] *made by the adversary, t is the running time of the adversary, and* $\mathbf{Insec}^{\text{com}}(t)$ *is the insecurity of the vector commitment used to produce* $C_{\texttt{attestors}}$ *(Section II-A).*

Proof. First, consider the following relation $\mathcal{R}$ of pairs. Let an instance $\mathbb{x} = (\texttt{attestors}, M, \texttt{provenWeight})$. For a list of signatures $\mathbb{w}$, the pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ if and only if $(\texttt{attestors}, M, \mathbb{w}, \texttt{provenWeight})$ is sufficiently weighty (Section III).

We first study a (rather inefficient) Interactive Oracle Proof (IOP) $P_{\text{interactive}}$ for this relation $\mathcal{R}$, in which the on common input $\mathbb{x}$, the prover will prove knowledge of $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in R$. An IOP [10, §4] is an interactive proof that is best viewed as a multi-round analogue of a Probabilistically Checkable Proof (PCP): a prover and a verifier exchange messages, but the verifier reads only a small (randomly chosen) portion of each verifier message. In our case, the prover messages are simply uniformly random coins, and thus the protocol is *public-coin.*

Lemma 1. *The protocol $P_{\text{interactive}}$ (Figure 1) is a public-coin interactive oracle proof of knowledge (as defined in [10, §4.2]) for $\mathcal{R}$ with knowledge error* $e = (\texttt{provenWeight}/\texttt{signedWeight})^{\texttt{numReveals}}$.

Proof. Completeness is self-evident, and all that we need to show is the proof of knowledge property. Indeed, to extract $\mathbb{w}$, simply remove the `L` field from every entry of `sigs` and output the result. It remains to show that if the verifier accepts with probability ϵ, then the resulting $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ (i.e., the total weight of valid signatures is at least `provenWeight`) with probability at least $\epsilon - e$.

Consider `sigs` sent by the prover in the first message. The prover can lie about the `L` values of some (or all) elements of `sigs`, but not about their weight or the correctness of their signatures. The important feature of the `L` values for security is not their correctness, but rather their fixity once the first message is sent by the prover. Fixing $\texttt{sigs}[i].\texttt{L}$ for a given i ensures that the prover can use the validity of $\texttt{sigs}[i].\texttt{sig}$ in response to some $\texttt{coin}_j$ only if $\texttt{sigs}[i].\texttt{L} \leq \texttt{coin}_j < \texttt{sigs}[i].\texttt{L} + \texttt{attestors}[i].\texttt{weight}$. Thus, no matter what $\texttt{sigs}[i].\texttt{L}$ is set to, the total amount of the range $[0, \texttt{signedWeight})$ that $\texttt{sigs}[i]$ can cover is limited to $\texttt{attestors}[i].\texttt{weight}$. Therefore, after the first message is sent, if the total weight of attestors whose signatures are valid in `sigs` is less than `provenWeight`, then the probability, for each j, that there exists an i_j for that will satisfy the verifier is less than `provenWeight/signedWeight`. Thus, the probability that the prover will convince the verifier for all `numReveals` values of $\texttt{coin}_j$ is less than $(\texttt{provenWeight}/\texttt{signedWeight})^{\texttt{numReveals}} = e$.

Therefore, either the prover's first message makes the knowledge extractor succeed, or the prover has to get very lucky (probability less than e) with the verifier's coins. By the union bound, the prover's success probability is less than that of the knowledge extractor plus e. □

We can modify $P_{\text{interactive}}$ to send the second prover message as a map rather than a list (i.e., in arbitrary order and with duplicates removed, with the verifier figuring out which set element to use for which $\texttt{coin}_j$). The analysis remains the same. Let us call this modified protocol $P'_{\text{interactive}}$.

As shown by [10, Theorem 7.1], the process of applying Merkle Trees (which shorten the first message) together with the Fiat-Shamir transformation (which replaces interaction with random oracles) to a public-coin interactive oracle proof of knowledge results in a NIROPK, and thus the required knowledge extractor per Definition 1 exists (we will analyze its knowledge error shortly).

[1]For easier analysis in terms of Q, we assume, without loss of generality, that the adversary always runs the verification algorithm on the proof it outputs, making the necessary random oracle queries in the process.

To show that P_{compcert} is a compact certificate scheme, we need to show the existence of a knowledge extractor according to Definition 2. We can essentially use the knowledge extractor given by [10, Theorem 7.1]. The most salient difference between P_{compcert} and the transformation of [10] applied to $P'_{\text{interactive}}$ is that in P_{compcert}, the verifier does not have all of $\mathbb{x}$. Instead, `attestors` is replaced by $C_{\texttt{attestors}}$, and the prover provides only those elements of `attestors` that the verifier needs. Since $C_{\texttt{attestors}}$ is trusted (per Section III; from a security point of view, this is equivalent to having $C_{\texttt{attestors}}$ computed by the verifier), any prover who convinces the verifier to accept an incorrect element of `attestors` would break the security of the vector commitment; this accounts for the $\textbf{Insec}^{\text{com}}$ term in the knowledge error given in Theorem 1.

There are other, minor differences between our transformation from $P'_{\text{interactive}}$ to P_{compcert} and the IOP-to-NIROPK transformation of [10, §6]. The differences arise because of the specifics of $P_{\text{interactive}}$, which enable us to simplify the transformation slightly; these simplifications do not affect the extractor construction in any significant way. A reader unfamiliar with the details of [10, §6] may safely skip this list.

- As a consequence of replacing `attestors` with $C_{\texttt{attestors}}$ in the verifier input, the verifier does not hash all of $\mathbb{x}$, but also replaces `attestors` by $C_{\texttt{attestors}}$. This has no effect on the security proof.
- The prover's second message, which is short, is sent in the clear rather than Merkle-hashed. This does not decrease security.
- We do not compute hash values that tie together multiple rounds of Merkle roots and verifier randomness, because Merkle hashing is applied only once (so chaining of roots is not needed) and the verifier needs randomness only once. This only simplifies the hash chains that are needed to analyze security.
- We do not have the prover compute the final hash value σ_k of the alleged verifier queries (which any prover can do correctly anyway by expending one more random oracle query); instead, we assume (Footnote 1) that the prover makes at least the same random oracle queries as the verifier.

To analyze the knowledge error of P_{compcert}, we will apply the same analysis as in [10, Thm. 7.1]. The analysis shows that if the `attestors` vector is used instead of $C_{\texttt{attestors}}$, the knowledge extractor will succeed unless the adversary gets a lucky choice of the sequence $\{\texttt{coin}_j\}_{j=0}^{\texttt{numReveals}-1}$ in one of its Q queries to `Hash` (which happens with probability at most

$$Q \cdot \left(\frac{\texttt{provenWeight}}{\texttt{signedWeight}}\right)^{\texttt{numReveals}}$$

by Lemma 1 and the union bound), or gets a collision of random oracle outputs (which happens with probability at most $\frac{1}{2}Q^2/2^\lambda$).[2] Because we replace `attestors` by $C_{\texttt{attestors}}$, we have to also add the probability $\textbf{Insec}^{\text{com}}$ that the adversary breaks the vector commitment.

We thus get that the protocol P_{compcert} is a compact certificate scheme with soundness error

$$e < Q \cdot \left(\frac{\texttt{provenWeight}}{\texttt{signedWeight}}\right)^{\texttt{numReveals}} + \frac{1}{2} \cdot \frac{Q^2}{2^\lambda} + \textbf{Insec}^{\text{com}},$$

as claimed.

□

B. *Choosing Parameters for Desired Security*

The knowledge error e of Theorem 1 has three terms. The $\textbf{Insec}^{\text{com}}$ term depends only on the commitment used for $C_{\texttt{attestors}}$, so as long as this commitment is sufficiently secure, there is nothing to analyze. The $1/2 \cdot Q^2 \cdot 2^{-\lambda}$ term is small enough as long as λ is long enough; for practical purposes, 256-bit λ suffices for 128-bit security, as is usual for collision-resistant hashing.

The interesting term to analyze is thus $Q \cdot (\texttt{provenWeight}/\texttt{signedWeight})^{\texttt{numReveals}}$. If we want this term to be smaller than 2^{-k}, then, recalling that $Q = 2^q$ and solving

$$2^{-k} = 2^q \cdot \left(\frac{\texttt{provenWeight}}{\texttt{signedWeight}}\right)^{\texttt{numReveals}}$$

for `numReveals` gives

$$\texttt{numReveals} = \frac{k+q}{\log_2(\texttt{signedWeight}/\texttt{provenWeight})}, \qquad (2)$$

which is at most the value we use in the prover and verifier algorithms (Equation (1), Section IV-A).

Note that the closer `signedWeight` is to `provenWeight`, the larger `numReveals` will be, and thus the larger the compact certificate. Thus, as discussed in Section IV-B, verifiers may choose to require a value for `signedWeight` that limits `numReveals`, resulting in a shorter certificate and therefore lower verification cost.

VI. Using Compact Certificates When Some Attestors are Adversarial

The statement of Theorem 1 and the analysis of Section V-B give us concrete bounds on the insecurity of our compact certificate scheme. We now wish to understand how these bounds apply when some of the attestors are adversarial.

In the rest of this section, we compute what `provenWeight` should be in two possible scenarios. We then demonstrate examples of `numReveals` computed according to Equation (1) (Section IV-A) for the given `provenWeight` for 128-bit security (i.e., an adversary making $Q = 2^q$ queries to the random oracle should succeed with probability at most $Q \cdot 2^{-128}$; thus, we set $2^{-k} = Q \cdot 2^{-128}$ — equivalently, $k + q = 128$).

[2]To see why our security loss is slightly better than in [10, Thm. 7.1], note that the reduction fails only in case of hash collision or a hash output guess; guessing is prevented by the assumption in Footnote 1, and hash collisions are overcounted in [10, Claim 7.3], because ρ_1 and ρ_2 collisions can be counted separately.

Maximum adversarial fraction f_A	Fraction $1 - \frac{\texttt{signedWeight}}{\texttt{totalWeight}}$ of attestations missing: 0	$f_A/2$	f_A	$1.5f_A$	$2f_A$
5%	30	30	31	31	31
10%	39	40	41	42	43
15%	47	49	52	55	58
20%	56	59	64	71	81
25%	64	71	81	97	128
30%	74	86	105	147	309
35%	85	104	144	291	—
40%	97	128	219	—	—
45%	112	164	442	—	—

Fig. 2. `numReveals` values for proving at least one honest attestation (Section VI-A). '—' means that no such value is possible.

Maximum adversarial fraction f_A	Fraction $1 - \frac{\texttt{signedWeight}}{\texttt{totalWeight}}$ of attestations missing or not agreeing with the majority: 0	$f_A/2$	f_A	$1.5f_A$	$2f_A$
5%	138	144	150	157	165
10%	149	163	181	204	237
15%	161	187	227	298	452
20%	174	219	309	576	—
25%	189	264	487	—	—
30%	206	331	1198	—	—
35%	226	443	—	—	—
40%	249	665	—	—	—
45%	276	1331	—	—	—

Fig. 3. `numReveals` values for proving that a majority of honest weight signed M (Section VI-B). '—' means that no such value is possible.

The two scenarios we consider are:

- Proving that at least one honest attestor provided an attestation (this is useful when we can safely assume that there can be no disagreement among honest attestors): parameters worked out in Section VI-A.
- Providing an *incontrovertible* certificate—that is, proving that a majority of honest attestors provided an attestation (this is useful to establish consensus): parameters worked out in Section VI-B.

A. Parameters for Proving At Least One Honest Attestation

Suppose we can be assured that there is no disagreement among honest attestors. For example, in a blockchain that guarantees no forks (e.g., [38, 56]), honest participants will always agree on the block at a particular height. In that case, the truth can be established by any single honest attestor.

If we let f_A denote an upper bound on the fraction of the weight controlled by the adversary, then it suffices to prove that the total weight of signers is at least $\texttt{provenWeight} = f_A \cdot \texttt{totalWeight}$, where

$$\texttt{totalWeight} = \sum_i \texttt{attestors}[i].\texttt{weight},$$

since this guarantees that at least one of the signers is honest.

The actual `signedWeight` value can vary, depending on the number of attestors who do not submit attestations due to adversarial corruption, lost network connectivity, or other faults. Figure 2 shows `numReveals` computed according to Equation (1) (Section IV-A) for a number of scenarios ranging from more optimistic to more pessimistic.

B. Incontrovertible Certificates: Parameters for Proving Majority Agreement

In contrast to the previous section, suppose now that there is no guarantee of agreement even among honest attestors. If we wish to ensure that no two compact certificates attesting to the same event can contradict each other, then we need to verify that a majority of honest attestors attest to the same version of an event.

If the corrupted fraction is x, then half of honest weight is $(1 - x)/2$, and thus it suffices to prove that the total weight of valid attestations is more than $(1 - x)/2 + x = (1 + x)/2$. Thus, if $x < f_A$, then it suffices to prove that total weight of valid attestations is at least $\texttt{provenWeight} = (1 + f_A)/2$. The value `numReveals`, as explained above, depends not only on `provenWeight`, but also on the actual weight `signedWeight` of attestations that certificate creator collected. As in the previous section, the actual `signedWeight` value can vary. Figure 3 gives examples of `numReveals` computed according to Equation (1) (Section IV-A) for different values of f_A and `signedWeight`.

VII. Performance evaluation

This section empirically answers the following questions about the compact certificate scheme P_{compcert} from Section IV.

- How much CPU time is required to create a compact certificate?
- How much CPU time is required to verify a compact certificate?
- What is the size of a compact certificate?

A. Implementation

To evaluate the performance of compact certificates, we implemented a prototype of the compact certificate scheme described Section IV. The implementation consists of about 1,200 lines of Go code, including 400 lines of code for a Merkle commitment library and 200 lines of code for a deterministic floating-point library that efficiently computes `numReveals` (see Section IV-C and Appendix B). The Merkle tree library aggregates proofs for multiple elements together, by eliding common paths to the root (see Section IV-C). To make this optimization more effective, we sort the elements of the `attestors` array by weight. This ensures that high-weight elements, which appear more often, are clustered together and share more common elements in their path to the root of the tree. We used ed25519 [14, 15] signatures and SHA-512/256 [86] hash implementations from `libsodium` [69], and we used msgpack [46] to encode compact certificates into a byte sequence.

B. Experimental results

We ran our evaluation on an Intel Xeon Silver 4215R CPU (3.2 GHz) running Linux 5.11 and Go 1.16. Unless otherwise mentioned, we simulated 1 million attestors, each with equal weight, and set the target `provenWeight` to half of the total weight of all attestors. We set the security parameter $k + q = 128$ (Section VI). We ran each experiment 3 times, reporting the median outcome; there was little variance in the results.

To provide a baseline comparison, we also evaluated a naïve scheme where the certificate consists of a set of signatures that add up to `provenWeight`. Verifying this certificate requires verifying each of the signatures in the certificate.

To measure the time required to create and verify certificates, we performed the following steps:

Signing messages: We signed messages on behalf of all attestors. This involves a standard ed25519 signature. This took 21.6 seconds (21.6 μsec per signature); we expect that in a real deployment, this cost would be distributed across many nodes. This time was the same for both compact certificates and naïve certificates.

Building the certificate: We built a compact certificate using the resulting signatures, which took 56 seconds. This was dominated by the time to verify all of the signatures (55.4 seconds, or 55.4 μsec per signature, for `signedWeight` = `totalWeight`).[3] The rest of the time (400 msec) was mostly spent constructing the Merkle tree over the signatures. Notably, the time to generate the certificate is largely independent of `signedWeight`.

We also measured the time to generate a naïve certificate, which entails checking signatures until the total weight of the checked signatures is at least `provenWeight`. For `provenWeight` = `totalWeight`/2, this took 28 seconds (twice as fast as building the compact certificate, since it involves verifying half the number of signatures).

Figure 4 shows the size of the resulting compact certificate, as a function of `signedWeight`, for varying numbers of attestors. The size is dominated by the size of the Merkle proof for each reveal; a larger number of attestors translates into larger Merkle proofs. The certificate size shrinks with higher `signedWeight` because the certificate includes fewer reveals. The number of reveals ranges from 930 (for `signedWeight` at 55% of `totalWeight`) down to 129 (`signedWeight` = `totalWeight`) with 1 million attestors, yielding compact certificates ranging from 650 kBytes to 120 kBytes. In contrast, the size of the naïve certificate for 1 million attestors was 54.9 MBytes (corresponding to 500,000 signatures and public keys), which is roughly 80–450× larger than the compact certificates.

For 10K attestors, the naïve certificate is 540 KBytes (100× fewer signatures), and the compact certificate is 2-3× smaller (shorter Merkle proofs), making the naïve certificate 2–8× larger than the compact certificate. With 100 attestors, the naïve certificate is 5 KBytes, and the compact certificate is 7.5–12 KBytes (1.5–2.2× larger); compact certificates do not provide any savings when the number of attestors is comparable to the number of reveals.

[3]subsection VIII-A discusses the trade-off involved in lazy verification of signatures when building a compact certificate.

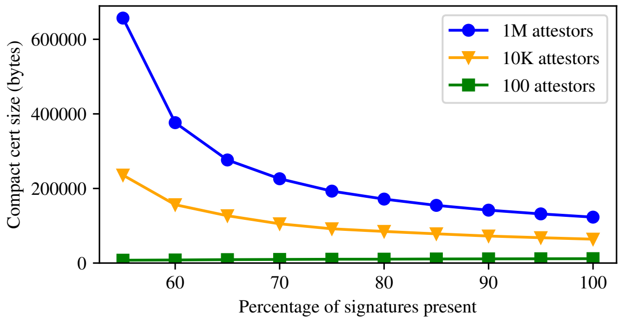

Fig. 4. Size of compact certificate (bytes), as a function of the percentage of signatures present (`signedWeight`), for `provenWeight` = `totalWeight`/2.

Finally, Figure 5 shows the size of the compact certificate for different values of `provenWeight`. As `provenWeight` goes up, the number of reveals in a certificate increases. For example, with `signedWeight` at 80% of `totalWeight`, compact certificates have 43, 91, 189, and 457 reveals for `provenWeight` at 10%, 30%, 50%, and 70% of `totalWeight` respectively.

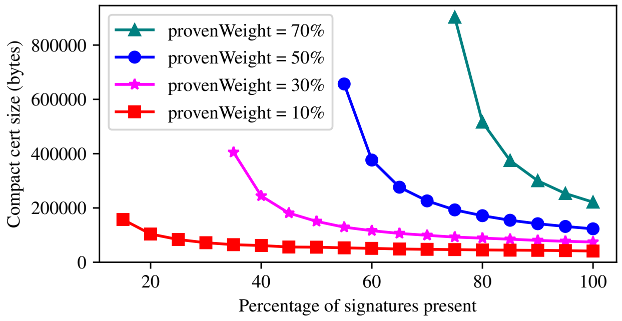

Fig. 5. Size of compact certificate (bytes), as a function of the percentage of signatures present (`signedWeight`), for a range of values of `provenWeight`.

Verifying the certificate: We verified the resulting compact certificate. Figure 6 shows the time required for verification, ranging from 67 msec (for `signedWeight` at 55% of `totalWeight`, with 931 reveals) down to 8.6 msec (for `signedWeight` = `totalWeight`, with 129 reveals). The time is dominated by the cost of checking the revealed ed25519 signatures, as seen from the fact that the verification time for 1M attestors is about the same as that for 10K attestors (which has nearly the same number of reveals but shorter Merkle proofs). The verification time for 100 attestors is much lower, because of the lower total number of reveals in the certificate.

In the naïve certificate scheme, the verification time was far higher—28 seconds, the same as the time to build the naïve certificate—corresponding to the time required to verify

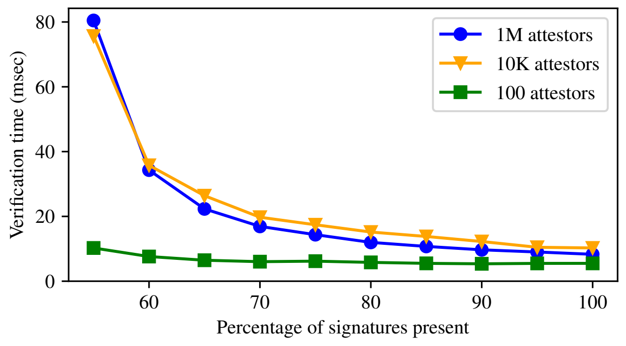

Fig. 6. Time taken to verify a compact certificate, as a function of the percentage of signatures present (`signedWeight`), for `provenWeight` = `totalWeight`/2.

500,000 signatures. This is roughly 400–3000× slower than checking a compact certificate.

Weight distribution: To evaluate the size of the compact certificate when the distribution of weights is not uniform, we generated several skewed distributions, based on a skew parameter s. The skew distribution was chosen to be easy to model and understand, to illustrate the effect of skew on certificate size, rather than modeling any real-world distribution of weights. Each skewed distribution consisted of 1 million attestors. The first attestor had a weight of 2^{44}, and the weight of each subsequent attestor was multiplied by $1 - 10^{-s}$ (rounded up to 1 unit of weight to ensure that all 1 million attestors have non-zero weight).

Figure 7 shows the results. With extremely skewed distributions, the number of distinct reveals (i.e., $|\mathtt{T}|$) is low because the same attestor is chosen to be revealed multiple times, but appears only once in the resulting certificate (Section IV-A, Step 6). For example, at $s = 1$, there are only 29 distinct reveals even though `numReveals` = 129. As seen from the graph, at extreme skew levels ($s \leq 2$), the naïve approach of sending the highest-weight signatures is more efficient than a compact certificate. For instance, with $s = 2$, signatures from the top 69 attestors account for half of the total weight.

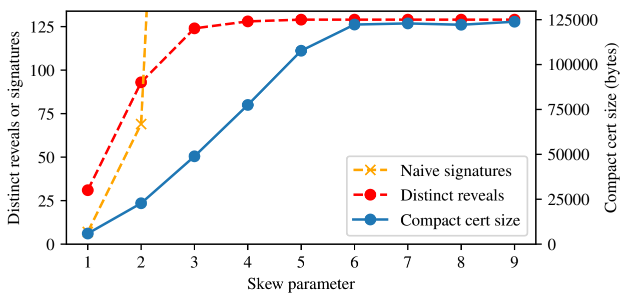

Fig. 7. Size of compact certificate (bytes) and number of distinct reveals, as a function of the skew of the weight distribution, for `signedWeight` = `totalWeight` = 2 · `provenWeight`.

At moderate skew levels ($s \geq 3$), attestors are no longer chosen to be revealed multiple times, but the certificate size is significantly smaller than the unskewed case because the Merkle proofs elide common paths to the root for high-weight attestors, which are clustered together (as a result of sorting by weight). For instance, with $s = 4$, there are 129 distinct reveals, but the certificate size is 76 kBytes, versus 124 kBytes for an unskewed distribution. A naïve approach would not fare well at this level of skew: $s = 4$ would require signatures from the top 6932 attestors to account for `provenWeight` = `totalWeight`/2.

VIII. Implementing Certificate Formation In A Decentralized Setting

In this section, we address the problem of constructing a compact certificate in a setting without a single trusted prover and with a somewhat unreliable network. This setting arises naturally in a permissionless blockchain system (e.g., [26, 38, 56, 78, 84]). Specifically, we use the Algorand blockchain[4] as an example. We added compact certificates to the Algorand system in order to provide succinct proofs of block validity for parties that are not verifying the blocks in real time. As described in Section I, this approach saves these parties from having to verify the entire blockchain in order to verify a particular block.

In Algorand's protocol, every block header contains a hash of the previous block header (much like every block chain, so that a block transitively commits to all prior blocks), and a "real-time" consensus protocol is used to agree on each next block. Compact certificates in Algorand certify every Nth block header, providing an efficient way of authenticating future blocks by jumping forward N blocks at a time. The N parameter is on the order of 128 or more, to amortize the cost of collecting signatures for the compact certificate. Validating a compact certificate for a block header requires a commitment to the participants that are expected to sign that block header. To provide this commitment, every Nth block header contains a Merkle commitment to the public keys and weights of participants eligible to form the next compact certificate.[5]

Algorand's use of compact certificates allows anyone to start from a known valid block header and efficiently validate a future block. To do so requires using previous-block hashes to validate the most recent block header that's a multiple of N (which will contain a Merkle commitment to the participants), then use compact certificates to jump forward by N block headers at a time, and then use previous-block hashes again to step back to the exact block that needs to be validated. One limitation of this approach is that compact certificates alone cannot be used to validate the most recent N blocks or so. Validating the most recent N blocks would require verifying messages from the "real-time" consensus protocol, which in Algorand's case requires having access to the account state.

We wish for the construction of a compact certificate to be both reliable and efficient, even if no single certificate creator or attestor can be relied upon. The honest and

[4]https://algorand.foundation/

[5]Participants are weighted by their account balance, which provides a similar stake-weighted guarantee as the "real-time" Algorand consensus protocol.

connected attestors will wish to make sure that the certificate is constructed without knowing who exactly is constructing it. We thus have to address several challenges that arise due to resource constraints, adversarial nodes, and fault tolerance.

We will use the term *node* to refer to a computer that participates in the decentralized protocol. In Algorand's deployment setting, there is a special type of node called a *relay*. The job of relay nodes is to support the decentralized protocol by relaying messages among all of the participants. A public directory of known relay nodes enables any participant to join the Algorand network protocol. Non-relay nodes typically connect to several relay nodes, but do not connect to one another. We assume that the underlying blockchain system (Algorand) provides a consensus mechanism; we will rely on this mechanism to ensure reliability.

We do not give evaluation results for the design in this section, because those results would depend almost exclusively on the details of the underlying blockchain and network. Algorand's deployment of compact certificates is in the early stages, and we do not yet have significant data from a real-world deployment.

A. Resource constraints: Collecting signatures

The first challenge in our decentralized setting lies in deciding what nodes will form compact certificates. Constructing a compact certificate requires access to all of the signatures from attestors (the `sigs` array from Section IV-A), even though the resulting compact certificate is far smaller. This means that any node that forms a compact certificate must receive and store many messages (linear in the number of attestors in the system). Requiring all nodes in a decentralized protocol to play this role would require bandwidth quadratic in the number of attestors, and can be costly if the number of attestors is high.

To avoid this cost, we treat relay nodes specially. Relay nodes are responsible for collecting all signatures that will be used to build the compact certificate, and relaying any signatures they receive to other relay nodes in the system, so that all relay nodes have all of the signatures. Non-relay nodes send their signatures to relay nodes, but do not receive signatures from other nodes. Each node in the system (both relay and non-relay) chooses several relay nodes to which it will send its messages. For relay nodes, this forms a network of relays so that signatures propagate between them in relatively few hops. For non-relay nodes, this ensures that their signatures will be quickly propagated across relay nodes, even if some relay nodes might be faulty.

It is important to choose carefully when to send a signature. All attestors in the system are likely to produce signatures at approximately the same time (e.g., when the next candidate block in a blockchain becomes available); if all nodes immediately send those signatures, the system will be overwhelmed by a spike of messages, many of which will have to be dropped.

To avoid such a bandwidth spike, we de-synchronize the transmission of signatures, by randomizing the time at which signatures are sent. In Algorand's protocol, we use the round number of the latest block as a proxy for time in this context. Specifically, we choose some window (e.g., the $N/2$ rounds after a multiple-of-N block is decided) to be designated for transmitting signatures. When an attestor signs a message, that attestor's node chooses a pseudorandom offset within the window at which the signature will be sent.[6] A relay node that receives a new, previously unseen signature, will immediately send that signature to other relays; this ensures rapid propagation of signatures. A relay nodes that receives a duplicate signature does not relay it.

A significant cost of the compact certificate protocol lies in the verification of the signatures from all attestors prior to the construction of the compact certificate. A natural optimization to consider is to avoid eager verification of these signatures: relay nodes would not bother verifying the signature, and would relay the signature to others as-is. Unfortunately, this optimization is problematic, for the following reason. To prevent an adversary from flooding the network with invalid signatures, relays would need to force verification of signatures once they receive two or more signatures from the same attestor. As a result, while the optimistic case avoids the CPU cost of verifying signatures, it would be relatively easy for an adversary to force relays to verify all signatures anyway, and to increase the bandwidth cost of signatures by relaying fake signatures first. Furthermore, this optimization makes it difficult for a relay node to know when it's ready to build a compact certificate: the node might start building a compact certificate, only to discover that one of the signatures that it needs to reveal is not valid, requiring it to drop that signature and restart the build. A non-trivial fraction of bad signatures can lead to many such restarts. To avoid the complexity and to achieve more predictable resource consumption, we chose to avoid this optimization.

It is also important to limit the number of attestors that are allowed to contribute signatures to the compact certificate, because otherwise an adversary who can form a large number of attestor identities can force relay nodes to handle and maintain in memory a large number of signatures. This can be addressed by capping the number of attestors for the purposes of compact certificates to some moderate number—say, the top 1 million accounts by weight—as long as all nodes in the system agree on which precise subset of accounts constitutes the set of attestors.

B. Adversarial nodes: When to create a certificate?

In a decentralized setting, there is a tension regarding when to form a compact certificate. On the one hand, it is desirable to form a compact certificate quickly, so that relay nodes can stop maintaining the set of all signatures in memory, attestors can stop re-transmitting their signatures (as we describe in the next subsection), and the compact certificate can be used sooner to convince verifiers. On the other hand, the total weight of all signatures at a relay node grows over time, as more signatures arrive from different attestors. Thus, waiting for more signatures to arrive enables a smaller compact certificate (because a higher `signedWeight` implies a lower `numReveals`, per Section V-B).

[6]A convenient scheme is to choose the offset pseudorandomly based on the public key of the signer.

This tension is exacerbated by the presence of adversarial nodes. In a decentralized setting, any relay node should be able to create a compact certificate; however, if one of those nodes was adversarial, it could create a compact certificate with the lowest acceptable signedWeight (and thus the largest possible numReveals), leading to a larger-than-necessary certificate.

To address this tension, we implement a decaying threshold for signedWeight of an acceptable compact certificate. The threshold decays with time, where time is measured by the round number of the latest block. The creation of the compact certificate can start at the time that corresponds to the end of the window for sending signatures (from the previous section). At that time, the threshold is initialized to totalWeight (i.e., requiring signatures from every attestor); the threshold decays linearly towards provenWeight (i.e., the lowest acceptable value for signedWeight) time goes on. At any given time, nodes in the system will accept a compact certificate only if its signedWeight is at least the current threshold value. The decay rate should be gradual enough to allow honest nodes to propose the best compact certificate they can construct, while still allowing the system to make progress (by forming at least *some* compact certificate) in a reasonable time frame.

In order for the nodes to have a consistent view of the current threshold (and thus have an agreement on whether a compact certificate is acceptable), they must agree on the time (round number) at which a compact certificate appears. In Algorand, a convenient way to ensure that each proposed compact certificate appears at an unambiguous round number is to include the compact certificate (attesting to an earlier block) as part of a (later) block; the compact certificate is then said to appear at the round number of the block that contains it.

C. Fault tolerance: Retransmitting signatures

It is important that compact certificates can be formed even if network or node failures occur during signature collection, as long as a sufficient set of nodes comes back online afterwards. To this end, we follow a simple rule: nodes must durably store their attestors' signatures until they see that a corresponding compact certificate is durably stored by the system. In a blockchain setting, storing the compact certificate on the blockchain itself provides a convenient way of ensuring the durability of the compact certificate, and thus making it safe to delete the input signatures that would have been necessary for creating the certificate.

Nodes periodically retransmit their stored signatures, so that a compact certificate can be formed even if the signatures were lost when they were first sent over the network. In particular, each node periodically (de-synchronized, just like during the initial sending) sends out all of its stored signatures to the relays to which it is connected. When a relay receives a signature that it already knows, it does not relay this signature immediately; the relay will resend that signature on its own retransmission schedule. On the other hand, when a relay receives a new signature for the first time, it will immediately relay it to other nodes, to ensure timely propagation.

In the common case, we expect that network-level retransmission (e.g., TCP) should ensure reliable message propagation, so the above retransmission plan should not start until after the initial transmission time window plus the decay time. This ensures that, in the common case, signatures for a compact certificate are sent at most once by each node to each of its connected relays.

Acknowledgments

We thank Alessandro Chiesa for clarifying the results of [10], Chris Peikert for a careful reading of our proof, Fraser Brown for editorial feedback, and Max Justicz for suggestions on and review of our implementation. We thank Sebastian Faust and the anonymous referees of both the real and the shadow program committees of IEEE S&P for help improving our presentation. The work of G.V. was performed while the author was at Algorand.

References

[1] Jae Hyun Ahn, Matthew Green, and Susan Hohenberger. Synchronized aggregate signatures: new definitions, constructions and applications. In *ACM CCS 2010*, October 2010.

[2] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. SANA: Secure and scalable aggregate network attestation. In *ACM CCS 2016*, October 2016.

[3] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM CCS 2017*, October / November 2017.

[4] Ali Bagherzandi, Jung Hee Cheon, and Stanislaw Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In *ACM CCS 2008*, October 2008.

[5] Ali Bagherzandi and Stanislaw Jarecki. Multisignatures using proofs of secret key possession, as secure as the Diffie-Hellman problem. In *SCN 08*, September 2008.

[6] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *ACM CCS 2006*, October / November 2006.

[7] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93*, November 1993.

[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO 2019, Part III*, August 2019.

[9] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO 2013, Part II*, August 2013.

[10] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *TCC 2016-B, Part II*, October / November 2016.

[11] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO 2014, Part II*, August 2014.

[12] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security 2014*, August 2014.

[13] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *FC 2016 Workshops*, February 2016.

[14] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *CHES 2011*, September / October 2011.

[15] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[16] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS 2012*, January 2012.

[17] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC 2013*, March 2013.

[18] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *PKC 2003*, January 2003.

[19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In *CRYPTO 2019, Part I*, August 2019.

[20] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Efficient polynomial commitment schemes for multiple points and polynomials. Cryptology ePrint Archive, Report 2020/081, 2020. https://eprint.iacr.org/2020/081.

[21] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *ASIACRYPT 2018, Part II*, December 2018.

[22] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT 2003*, May 2003.

[23] Kyle Brogle, Sharon Goldberg, and Leonid Reyzin. Sequential aggregate signatures with lazy verification from trapdoor permutations - (extended abstract). In *ASIACRYPT 2012*, December 2012.

[24] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *EUROCRYPT 2020, Part I*, May 2020.

[25] Mike Burmester, Yvo Desmedt, Hiroshi Doi, Masahiro Mambo, Eiji Okamoto, Mitsuru Tada, and Yuko Yoshifuji. A structured ElGamal-type multisignature scheme. In *PKC 2000*, January 2000.

[26] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014.

[27] Benedikt Bünz, Mary Maller, Pratyush Mishra, and Noah Vesely. Proofs for inner pairing products and applications. Cryptology ePrint Archive, Report 2019/1177, 2019. https://eprint.iacr.org/2019/1177.

[28] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In *ASIACRYPT*, 2020.

[29] Ran Canetti, Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In *CRYPTO'99*, August 1999.

[30] Claude Castelluccia, Stanislaw Jarecki, Jihye Kim, and Gene Tsudik. A robust multisignatures scheme with applications to acknowledgment aggregation. In *SCN 04*, September 2005.

[31] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *PKC 2013*, February / March 2013.

[32] Chin-Chen Chang, Jyh-Jong Leu, Pai-Cheng Huang, and Wei-Bin Lee. A scheme for obtaining a message from the digital multisignature. In *PKC'98*, February 1998.

[33] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968, 2018. https://eprint.iacr.org/2018/968.

[34] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT 2020, Part I*, May 2020.

[35] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *EUROCRYPT 2020, Part I*, May 2020.

[36] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, May 2015.

[37] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *FC 2019*, February 2019.

[38] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT 2018, Part II*, April / May 2018.

[39] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *CRYPTO'87*, August 1988.

[40] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO'89*, August 1990.

[41] Yvo Desmedt and Yair Frankel. Shared generation of authenticators and signatures (extended abstract). In *CRYPTO'91*, August 1992.

[42] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy*, May 2018.

[43] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy*, May 2019.

[44] Rachid El Bansarkhani and Jan Sturm. An efficient lattice-based multisignature scheme with applications to bitcoins. In *CANS 16*, November 2016.

[45] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO'86*, August 1987.

[46] Sadayuki Furuhashi et al. Messagepack: it's like JSON, but fast and small. https://msgpack.org/.

[47] Ariel Gabizon. AuroraLight: Improved prover efficiency and SRS size in a sonic-like system. Cryptology ePrint Archive, Report 2019/601, 2019. https://eprint.iacr.org/2019/601.

[48] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[49] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT 2013*, May 2013.

[50] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *ACM CCS 2018*, October 2018.

[51] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *ACNS 16*, June 2016.

[52] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust and efficient sharing of RSA functions. In *CRYPTO'96*, August 1996.

[53] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure applications of Pedersen's distributed key generation protocol. In *CT-RSA 2003*, April 2003.

[54] Craig Gentry, Adam O'Neill, and Leonid Reyzin. A unified framework for trapdoor-permutation-based sequential aggregate signatures. In *PKC 2018, Part II*, March 2018.

[55] Craig Gentry and Zulfikar Ramzan. Identity-based aggregate signatures. In *PKC 2006*, April 2006.

[56] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[57] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In *ACM CCS 20*, November 2020.

[58] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT 2016, Part II*, May 2016.

[59] Thomas Hardjono and Yuliang Zheng. A practical digital multisignature scheme based on discrete logarithms. In *AUSCRYPT'92*, December 1993.

[60] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT 2010*, December 2010.

[61] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *24th ACM STOC*, May 1992.

[62] Yuichi Komano, Kazuo Ohta, Atsushi Shimbo, and Shinichi Kawamura. Formal security model of multisignatures. In *ISC 2006*, August / September 2006.

[63] Russell W. F. Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *CRYPTO 2019, Part I*, August 2019.

[64] Susan K. Langford. Threshold DSS signatures without a trusted party. In *CRYPTO'95*, August 1995.

[65] Duc-Phong Le, Alexis Bonnecaze, and Alban Gabillon. Multisignatures as secure as the Diffie-Hellman problem in the plain public-key model. In *PAIRING 2009*, August 2009.

[66] Chuan-Ming Li, Tzonelih Hwang, and Narn-Yih Lee. Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders. In *EUROCRYPT'94*, May 1995.

[67] Benoît Libert, Somindu C. Ramanna, and Moti Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In *ICALP 2016*, July 2016.

[68] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *TCC 2010*, February 2010.

[69] libsodium: A modern, portable, easy to use crypto library. https://github.com/jedisct1/libsodium.

[70] Yehuda Lindell. Fast secure two-party ECDSA signing. In *CRYPTO 2017, Part II*, August 2017.

[71] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In *EUROCRYPT 2006*, May / June 2006.

[72] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential aggregate signatures from trapdoor permutations. In *EUROCRYPT 2004*, May 2004.

[73] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *ACM CCS 2019*, November 2019.

[74] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr multi-signatures with applications to Bitcoin. Cryptology ePrint Archive, Report 2018/068, 2018. https://eprint.iacr.org/2018/068.

[75] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO'87*, August 1988.

[76] Silvio Micali. CS proofs (extended abstracts). In *35th FOCS*, November 1994.

[77] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

[78] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[79] Gregory Neven. Efficient sequential aggregate signed data. In *EUROCRYPT 2008*, April 2008.

[80] Kazuo Ohta and Tatsuaki Okamoto. A digital multisignature scheme based on the Fiat-Shamir scheme. In *ASIACRYPT'91*, November 1993.

[81] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX Security 2020*, August 2020.

[82] Sangjoon Park, Sangwoo Park, Kwangjo Kim, and Dongho Won. Two efficient RSA multisignature schemes. In *ICICS 97*, November 1997.

[83] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, May 2013.

[84] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. 2018.

[85] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract) (rump session). In *EUROCRYPT'91*, April 1991.

[86] Secure hash standard. National Institute of Standards and Technology, NIST FIPS PUB 180-4, U.S. Department of Commerce, August 2015.

[87] Douglas R. Stinson and Reto Strobl. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *ACISP 01*, July 2001.

[88] Steve Thakur. Batching non-membership proofs with bilinear accumulators. Cryptology ePrint Archive, Report 2019/1147, 2019. https://eprint.iacr.org/2019/1147.

[89] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *SCN 20*, September 2020.

[90] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, May 2018.

[91] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO 2019, Part III*, August 2019.

Appendix A
Using Merkle Trees Unambiguously

Merkle trees ensure that there is a unique decommitment for every leaf position. However, the security goal of the vector commitment $C_{\texttt{attestors}}$ as well as of the Merkle tree with root $\texttt{Root}_{\texttt{sigs}}$, is to ensure that there is a unique decommitment for every index i. This goal can be achieved by ensuring that there is an unambiguous mapping, enforced by the verifier, between indices and leaf positions. How to construct this mapping depends on the tree structure and the verifier's knowledge. We suggest the following options.

- The size of the vector should be included as part of the commitment, and the tree structure should be fixed for any given size.
- Alternatively, if the tree structure is variable (which is helpful when the tree is constructed dynamically), but we can be sure that the commitment is computed by a trusted party (we make that assumption on $C_{\texttt{attestors}}$), then the data at each leaf can include its index, and the verifier will check that this index is equal to i.
- Finally, if the tree structure is variable and the commitment is not trusted, then the mapping from indices to leaf positions can be provided by another, trusted, commitment, as long as the verifier checks that this mapping is followed. Thus, the tree structure for $\texttt{Root}_{\texttt{sigs}}$ can simply parallel the structure of the tree that computes $C_{\texttt{attestors}}$, and the verifier will check that the paths in the two trees are the same for a given index (in addition to verifying that the index matches in $C_{\texttt{attestors}}$, as per the previous item).

Appendix B
Computing `numReveals` efficiently

In order to compute the value

$$\texttt{numReveals} = \left\lceil \frac{k+q}{\log_2\left(\texttt{signedWeight}/\texttt{provenWeight}\right)} \right\rceil$$

(per analysis in Section V-B) while avoiding expensive precise integer arithmetic or imprecise (and not always cross-platform compatible) floating-point arithmetic, we may wish to use approximate multiplication and exponentiation. Approximate multiplication, described below, stores only the most significant bits of intermediate values ("mantissa") and a second value ("exponent") representing the number of remaining, not stored, bits. In this section we describe this method of computing `numReveals` and analyze the error it produces.

Definition and Analysis of Approximate Multiplication and Exponentiation: Suppose we are limited to multiplication of integers less than $2w$, where w is a power of 2 (e.g., $w = 2^{31}$ and thus multiplication is limited to 32-bit integers and never produces an answer longer than 64 bits). For a positive integer x, define $[x]_w$ (respectively, $[x]^w$) as follows: if $x < 2w$, $[x]_w = [x]^w = x$; else $[x]_w$ (respectively, $[x]^w$) is equal to x rounded down (respectively, up) to the nearest multiple of 2^p, where p the unique integer such that $w \le x/2^p < 2w$.

For any x, $[x]_w$ (respectively, $[x]^w$) can be represented as $x_m \cdot 2^{x_e}$, where the mantissa $x_m = \lfloor x/2^p \rfloor$ (respectively, $\lceil x/2^p \rceil$) and the exponent $x_e = p$ (except when $x < 2w$, in which case $x_m = x$ and $x_e = 0$, or when rounding up produces $2w$, in which case $x_m = w$ and $x_e = p + 1$). Thus, multiplying values after $[\cdot]_w$ or $[\cdot]^w$ has been applied involves a multiplication of mantissas (which are less than $2w$) and an addition of exponents.

Note that if $x \ge w$, then by definition of p, $x/w \ge 2^p$ and thus $(1 - 1/w)x \le x - 2^p < [x]_w \le x \le [x]^w < x + 2^p \le (1 + 1/w)x$. We thus have, regardless of whether $x \ge w$ or not,

$$(1 - 1/w)x < [x]_w \le x \le [x]^w < (1 + 1/w)x\,.$$

For any integer a, let $a^{\lessapprox n}$ (respectively, $a^{\gtrapprox n}$) denote the result of starting at 1 and applying n repetitions of multiplying the result by a and applying $[\cdot]_w$ (respectively, $[\cdot]^w$) to the result. Formally,

$$a^{\lesssim n} \overset{\text{def}}{=} \underbrace{[\ldots[[a \cdot a]_w \cdot a]_w \cdot \cdots \cdot a]_w}_{n \text{ times}}$$

and

$$a^{\gtrsim n} \overset{\text{def}}{=} \underbrace{[\ldots[[a \cdot a]^w \cdot a]^w \cdot \cdots \cdot a]^w}_{n \text{ times}}$$

To avoid large-number arithmetic when exponentiating, we will first apply $[\cdot]_w$ or $[\cdot]^w$ to a value, and then perform one of the two forms of approximate exponentiation we just defined. To bound the error this method produces, observe that from the above inequality, we have

$$(1 - 1/w)^n a^n < a^{\lesssim n} \leq a^n \leq a^{\gtrsim n} < (1 + 1/w)^n a^n$$

and therefore

$$(1 - 1/w)^{2n} a^n < ([a]_w)^{\lesssim n} \leq a^n \leq ([a]^w)^{\gtrsim n} < (1 + 1/w)^{2n} a^n.$$

Using Approximate Exponentiation to Compute `numReveals`: To find `numReveals`, find the smallest positive integer n (using a simple loop incrementing n by one, multiplying, and rounding) such that

$$2^{k+q} \cdot ([\texttt{provenWeight}]^w)^{\gtrsim n} \leq ([\texttt{signedWeight}]_w)^{\lesssim n}.$$

Set `numReveals` $= n$.

This method never underestimates `numReveals` and thus guarantees security at least k, because $2^{k+q} \cdot \texttt{provenWeight}^n \leq 2^{k+q} \cdot ([\texttt{provenWeight}]^w)^{\gtrsim n} \leq ([\texttt{signedWeight}]_w)^{\lesssim n} \leq \texttt{signedWeight}^n$ and thus conditions in Section V-B for achieving security parameter k are satisfied.

This method may overestimate `numReveals`. However,

$$\frac{([\texttt{signedWeight}]_w)^{\lesssim n}}{([\texttt{provenWeight}]^w)^{\gtrsim n}} > \frac{\texttt{signedWeight}^n}{\texttt{provenWeight}^n} \cdot \frac{(1 - 1/w)^{2n}}{(1 + 1/w)^{2n}}$$

$$\approx \frac{\texttt{signedWeight}^n}{(\texttt{provenWeight}(1 + 4/w))^n}.$$

Therefore, the cost of this method is equivalent to the cost of increasing `provenWeight` by a factor of approximately $(1 + 4/w)$. If we are using 32-bit integers, the cost of this method is less than the cost of increasing `provenWeight` by two parts per billion.