Decentralizing Authorities into Scalable Strongest-Link Cothorities

Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, and Bryan Ford
Yale University

ABSTRACT

Online infrastructure often depends on security-critical *authorities* such as logging, time, and certificate services. Authorities, however, are vulnerable to the compromise of one or a few centralized hosts yielding "weakest-link" security. We propose collective authorities or *cothorities*, an architecture enabling thousands of participants to witness, validate, and co-sign an authority's public actions, with moderate delays and costs. Hosts comprising a cothority form an efficient communication tree, in which each host validates log entries proposed by the root, and contributes to collective log-entry signatures. These collective signatures are small and efficient to verify, while embodying "strongest-link" trust aggregated over the collective. We present and evaluate a prototype cothority implementation supporting logging, timestamping, and public randomness (lottery) functions. We find that cothorities can scale to support over 4000 widely-distributed participants while keeping collective signing latencies to within a few seconds.

1. INTRODUCTION

Our online (and offline) infrastructure often depends on *authorities* that provide conceptually simple but security-critical services that many higher-level services and applications depend on. For example, timestamp servers attest that a document existed at a particular time [1]; notaries attest that a document was signed by particular parties [2]; certificate authorities attest that a human-readable name is owned by the holder of a given public key [18]; directory authorities maintain lists of servers available to support particular applications [26,69]; randomness authorities produce public random numbers useful in games or lotteries [58,61].

Authorities are conventionally implemented via one or a few centralized servers, creating attractive targets for powerful adversaries. In the Certificate Authority (CA) system underlying SSL/TLS [24, 34], for example, attackers who steal secret keys from any of hundreds of CAs [27] can and have misused CA authority to impersonate popular web sites and spy on users [5,12,13].

Proposed logging and monitoring solutions such as Certificate Transparency [43,44] unfortunately offer only retroactive defense, leaving clients vulnerable to fake certificates for a time unless they first verify all certificates with multiple independent logs and monitors. The alternative of splitting an authority into a small *consensus group*, as in the Tor directory service [74,75], protects clients from any one compromised server. It is questionable, however, whether compromising or exfiltrating keys from *a few* servers in such a group remains beyond the capabilities of today's adversaries, particularly from increasingly powerful and often purportedly legal state-level hacking activities [35,66].

We propose to replace high-value authorities with large-scale *collective authorities*, which we call *cothorities*. The basic goal is to split trust across a large and diverse body of independently-run servers. A cothority guarantees *strongest-link* security whose strength increases as the collective grows, instead of decreasing to weakest-link security as in today's CA system. Each of potentially thousands of hosts comprising a cothority independently validates each public output, contributing a share of a collective digital signature to each validated output, or withholding its signature and raising an alarm if misbehavior is detected. Clients can validate a cothority's output – such as a log record, timestamp, certificate, or random number – using a single inexpensive cryptographic operation comparable to conventional signature verification. This collective signature attests to the client that not just one but *many* well-known servers (ideally thousands) independently checked and signed off on that output.

The formal foundation for cothorities already exists in the form of cryptographic threshold signatures [70], aggregate signatures [11], and multisignatures [8,54], but to our knowledge these primitives have been deployed only in small groups (*e.g.*, ≈ 10 nodes) in practice. Our main contribution is to demonstrate how to scale these techniques across thousands of servers in practical environments. A first-order technical challenge is limiting the computation and network bandwidth costs imposed on each participating server; we solve this challenge using tree-based communication structures comparable to those long used in multicast protocols [15,76].

A second-order challenge is handling server failures, which we expect to be rare but non-negligible, and if not addressed would make a cothority vulnerable to crashes or denial-of-service attacks by any server. We explore two solutions to this availability challenge. First, we allow each log entry's signature to contain a few *exceptions* explicitly listing servers whose contributions to the collective signature could not be obtained promptly. Second, we can require cothority servers to secret-share their temporary signing keys with a group of independent *insurers*, who can reconstruct the server's key if the server fails. These two approaches to guaranteeing availability represent tradeoffs and may be employed independently or together.

We have built a working cothority server prototype implementing collectively signed logging, timestamping, and vote-counting services. Experimental evaluations on DeterLab [23] demonstrate that the prototype scales easily to over 4000 participant servers, handling hundreds of thousands of client requests per second in aggregate. These large experimental cothorities can produce, validate, and sign new log entries with typical latencies of only 1–5 seconds – delays easily tolerable by typical authority services.

This paper makes the following contributions: (a) it proposes cothorities, a scalable approach to strongest-link trust for security-sensitive services; (b) it designs CoSi, the first protocol to make multisignatures scale to thousands of nodes; and (c) it presents and evaluates a prototype cothority implementation demonstrating that large-scale cothorities are a viable alternative to current centralized-trust authorities.

Section 2 of this paper explores the background and motivation for cothorities. Section 3 then presents CoSi, the collective signing protocol we use for the cothority architecture. Section 4 details the challenges and approaches to dealing with availability issues and failures. Section 5 describes the details of our implementation and Section 6 experimentally evaluates it. Section 7 further discusses the applicability of cothorities to real-world applications as well as outlines future work. Section 8 summarizes related work and Section 9 concludes.

2. BACKGROUND AND MOTIVATION

This section briefly reviews several types of conventional authorities, their weaknesses, and how a cothority architecture might strengthen them. We revisit prototype implementations of some of these applications later in Section 5.

Tamper-Evident Logging Authorities.

Many storage systems and other services rely on tamper-evident logging [19,46]. Logging services, however, are vulnerable to *equivocation* where a malicious log server rewrites history or presents different "views of history" to different clients. Solutions include weakening consistency guarantees as in SUNDR [46], or adding trusted hardware as in TrInc [45]. Equivocation is the fundamental reason Byzantine agreement in general requires $N = 3f + 1$ total nodes to tolerate f arbitrary failures [16]. A cothority architecture does not change this basic ratio, but enables both N and f to be large: *e.g.*, with $N > 3000$ participants independently checking and co-signing each new log entry, arbitrarily colluding groups up to 1000 participants cannot successfully equivocate or rewrite history.

Timestamping Authorities.

A timestamping authority [1, 37] typically enables clients to incorporate a cryptographic hash of some document (*e.g.*, a design to be patented) into a timestamped public log. The client can later prove to a third-party that the document existed at a historical date by pointing to the authority's log entry containing the document's hash. A corrupt timestamp authority, however, might mount the history-rewriting and equivocation attacks above, or might incorrectly timestamp (*e.g.*, pre-date) otherwise properly-sequenced log entries. In a cothority, many participants can not only protect the log's structure but independently verify that each entry's timestamp is reasonably close to the present time.

Certificate Authorities for Public-Key Infrastructure.

Certificate Authorities (CAs) sign certifcates attesting that the holder of a public key legitimately represents a name such as `google.com`, to authenticate SSL/TLS connections [24, 34]. Current web browsers directly trust dozens of root CAs and indirectly trust hundreds of intermediate CAs [27], any one of which can issue fake certificates for any domain if compromised. Due to this "weakest-link" security, hackers have stolen the "master keys" of CAs such as DigiNotar [5, 13] and Comodo [12] and abused certificate-issuance mechanisms [41, 42] to impersonate popular websites and attack their users.

As a stopgap, browsers such as Chrome hard-code *pinned* certificates for particular sites such as `google.com` [29] – but browsers cannot ship with hard-coded certificates for the whole Web. Generalizations of this approach pin the first certificate a client sees [20, 50, 71], protecting a site's regular users but not new users. An alternative is for browsers to check server certificates against public logs [6, 39, 43, 44, 64, 73], which independent monitors may check for invalid certificates. Monitoring can unfortunately detect misbehavior only *retroactively*, placing victims in a race with the attacker. Web browsers could check all certificates they receive against such logs and/or via multiple Internet paths [3, 7, 49, 77], but such checks add delays to the critical page-loading path. Further, these approaches assume Web users can connect to independent logging, monitoring, or relaying services without interference, but this assumption fails when the user's own ISP is the adversary – a scenario that has unfortunately become all too realistic whether motivated by state-level repression [5, 13] or profit [38].

By federating today's hundreds of CAs into a single *certificate cothority*, each CA could in principle validate certificates proposed by all other CAs *before* they are collectively signed. For example, the CA currently responsible for a given domain such as `google.com` could verify that no other CA proposes a `google.com` certificate, raising an alarm and proactively preventing the signing of fake certificates in the first place. Each certificate would still be validated by a single digital signature, but that signature would embody much stronger and broader-based *trust*.

Public Randomness Authorities.
Randomness authorities [58, 61] generate non-secret random numbers or coin-flips, which are useful for many purposes such as lotteries, sampling, or choosing elliptic curve parameters. NIST's Randomness Beacon [58], for example, produces a log of signed, timestamped values from a hardware source. If compromised, however, a randomness authority could deliberately choose its "random" values as to win a lottery, or could look into the future to predict a lottery's outcome. In the wake of the DUAL-EC-DRBG debacle [17], the NIST beacon has been skeptically called "the ~~NSA~~NIST Randomness Beacon" [72] and "Project 'Not a backdoor'" [62]. A randomness cothority, in contrast, could combine many independent randomness sources into a single log of collectively-signed random values, preventing even a substantial set of compromised servers from controlling or predicting its output.

Directory Authorities.
Clients of the Tor anonymity system [74] rely on a directory authority [75] to obtain a list of available anonymizing relays. A compromised directory authority could give clients a list containing only attacker-controlled relays, however, thereby de-anonymizing all clients. To mitigate this risk, Tor clients accept a list only if it is signed by a majority of a small *consensus group*, currently about 8 servers. Because these directory servers and their private directory-signing keys represent high-value targets for increasingly powerful state-level adversaries it is increasingly questionable whether splitting trust over such a small, relatively centralized group offers adequate security. A cothority architecture, in contrast, could spread this trust across a much wider base of hundreds or even thousands of directory servers.

3. COLLECTIVE SIGNING

This section presents CoSi, the first practical *collective signing* protocol we know of to build large-scale cothorities.

3.1 Principles of Operation

We consider a given cothority to be implemented by a single instance of the CoSi protocol, whose conceptual architecture is illustrated in Figure 1. For simplicity we first consider an architecture whose participants play two asymmetric roles: there is a single distinguished *leader* and any number N of *signers*. Later in Section 4.3 we will address leader-rotation mechanisms to support homogeneous groups.

The leader implements the cothority's authoritative logic: *e.g.*, assigning timestamps or certificates, generating random

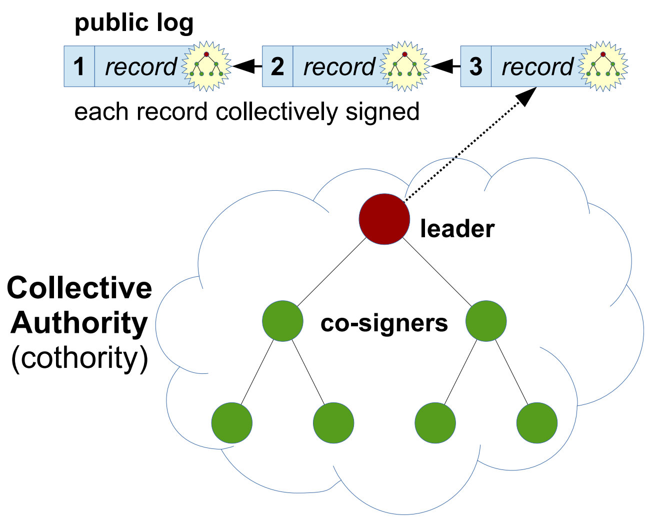
public log
1 record
2 record
3 record
each record collectively signed
leader
co-signers
Collective Authority (cothority)

Figure 1: CoSi protocol architecture

numbers, etc. The signers are additional reliable, independently-run servers maintained by individuals or organizations who may have volunteered – or contracted with the leader – to monitor, validate, and co-sign the leader's authoritative actions. Of course most realistic authorities provide a service to clients (*e.g.*, users requesting timestamps or certificates), but these clients do not directly participate in the CoSi protocol so we will ignore them for now.

We assume for now that the leader and its set of signers is fixed, and that all participants know and agree on each others' public keys. For example, the leader might administratively create and publish a roster listing its public key and those of all N signers. We can preclude disagreement by taking a cryptographic hash of this roster as as a unique identifier for the cothority and its instance of the CoSi protocol, so that different rosters represent different cothorities. We consider extensions for dynamic leader selection and roster evolution later in Section 4.4.

We assume that the leader's authoritative actions – such as assigning timestamps or certificates – are to be recorded or summarized publicly in a tamper-evident log, and that the information in this log suffices to validate the leader's actions. To append to this log, the leader periodically initiates a new *round* of the CoSi protocol. In each round, the leader announces a record to be added to its log, assigning each record a consecutive sequence number starting from 1. After announcing the next record, the leader coordinates with all of its signers to validate the log entry and generate a collective signature for it. The result of a successful round is thus a collectively signed log entry, which anyone may subsequently check against the roster, to verify that not only the leader but *all* signers in the roster saw and attested to the validity of each log entry.

In general the signers will perform application-specific semantic checks on each of the leader's proposed log entries before "signing off" on them. For simplicity of exposition, however, we assume for now that the signers treat each record's content as opaque, and merely ensure that these records form a properly sequenced log. Thus, for now signers verify only that the leader indeed assigns sequence numbers consecutively, and never attempts to roll back or rewrite history, or to equivocate by signing multiple records with the same sequence number. This basic, "semantic-free" instantiation of CoSi thus implements an anti-equivocation mechanism: a decentralized analog of TrInc [45]. Section 3.5 will later cover ways to incorporate application-specific data and validation rules.

We assume for now that participants never fail or become disconnected, relaxing this admittedly unrealistic assumption later in Section 4. However, we are not at the moment highly concerned about minimizing the latency of collective signing operations.

3.2 Schnorr (Multi-)Signatures

While CoSi could in principle build on many digital signature schemes that support efficient public key and signature aggregation, we focus here on the simplest and most well-understood such scheme we are aware of: Schnorr signatures [68] and multisignatures [8,54].

Schnorr signatures rely on a group $\mathcal{G}$ of prime order q in which the discrete logarithm problem is believed to be hard; in practice we now use elliptic curves for $\mathcal{G}$. Given a well-known generator G of $\mathcal{G}$, each user chooses a random private key $x < q$, and computes her corresponding public key $X = G^x$. (We use multiplicative-group notation for consistency with the historical literature on Schnorr signatures, although additive-group notation may be more natural with elliptic curves.)

Schnorr signing is conceptually a *prover-verifier* or Σ-protocol [21], which we make non-interactive using the Fiat-Shamir heuristic [31]. To sign a message M, the prover picks a random secret $v < q$, computes a *commit*, $V = G^v$, and sends V to the verifier. The verifier responds with a random *challenge* $c < q$, which in non-interactive operation is simply a cryptographic hash $c = \mathrm{H}(V||M)$. The prover finally produces a *response*, $r = v - cx$, where x is the prover's private key. The challenge-response pair (c, r) is the Schnorr signature, which anyone may verify using the signer's public key $X = G^x$, by recomputing $V' = G^r X^c$ and checking that $c \stackrel{?}{=} \mathrm{H}(V'||M)$.

With Schnorr multisignatures [59], there are N signers with individual private keys $x_1, \ldots, x_N$ and corresponding public keys $X_1 = G^{x_1}, \ldots, X_N = G^{x_N}$. We compute an *aggregate* public key X from the individual public keys as $X = \prod_i X_i = G^{\sum_i x_i}$. The N signers can collectively sign a message M as follows. Each signer i picks a random secret $v_i < q$, and computes a commit $V_i = G^{v_i}$. One participant (*e.g.*, a leader) collects all N commits, aggregates them into a collective commit $V = \prod_i V_i$, and uses a hash function to compute a collective challenge $c = \mathrm{H}(V||M)$. The leader distributes c to the N signers, each of whom computes and returns its response share $r_i = v_i - cx_i$. Finally, the leader aggregates the response shares into $r = \sum_i r_i$, to produce a collective signature (c, r). Anyone can verify this constant-size signature against the message M and the aggregate public key X using the normal Schnorr signature verification scheme above.

When forming an aggregate public key X from a roster of individual public keys $X_1, \ldots, X_N$, all participants must validate each individual public key X_i by requiring its owner i to prove knowledge of the corresponding private key x_i, *e.g.*, with a zero-knowledge proof or self-signed certificate. Otherwise, a dishonest node i can perform a *related-key attack* [55] against a victim node j by choosing $X_i = G^{x_i} X_j^{-1}$, and thereafter produce collective signatures apparently signed by j without j's actual participation.

While multisignatures are well-understood and formally analyzed, to our knowledge they have so far been considered practical only in small groups (*e.g.*, $N \approx 10$). The next sections describe how we can make multisignatures scale to thousands of participants, and address the availability challenges that naturally arise in such contexts.

3.3 Forming Trees for Collective Signing

To make multisignatures scale to many participants, CoSi distributes the communication and computation costs of multisignatures across a communication tree analogous to those long utilized in multicast protocols [15].

We assume that all N CoSi servers have been organized into a communication tree rooted at the distinguished leader, whose structure is defined by a *tree roster* known to and agreed upon by all participants. We assume for now that this tree roster is static, deferring liveness and evolution issues to Section 4. We represent this tree roster efficiently as a Merkle tree [52] whose structure mirrors the communication topology, with one Merkle node per CoSi server.

The Merkle node for each host i includes: (a) i's public key $X_i = G^{x_i}$ and self-signed certificate, (b) cryptographic hash-links to the Merkle nodes representing i's immediate children, and (c) a partial aggregate public key $\hat{X}_i$ combining i's public key with those of its descendants. If D_i is the set of i's descendants including i itself, then $\hat{X}_i = \prod_{j \in D_i} X_j = G^{\sum_{j \in D_i} x_j}$. The leader's aggregate public key, $\hat{X}_0$, combines the public keys of all participants, and may be verified by anyone via a bottom-up traversal of the tree roster. While verifying the full tree roster takes $O(N)$ time, we assume this is done rarely – *e.g.*, on administrative time scales – and not during each CoSi protocol round.

3.4 Tree-based Signing

Each CoSi protocol round consists of four phases, As mentioned before, each node i has a public key $X_i = G^{x_i}$, where x_i is i's private key and G is a generator of a suitable integer or elliptic curve group. For each node i, we define the *composite* public key $\hat{X}_i$ as the combination of the public keys of node i and all its descendants using the group operation: $\hat{X}_i = \prod_{j \in D_i} X_j = G^{\sum_{j \in D_i} x_j}$, where D_i is the set of transitive descendants of i including i itself. Taking node 0 to be the root, each log entry produced by CoSi will be verifiable using the network-wide composite public key $\hat{X}_0$, hence producing these log entry signatures will require the participation of every node i with its "share" x_i of the composite private key.

A single round of the CoSi protocol consists of four phases, representing two communication "round-trips" through the spanning tree:

1. **Announcement:** The leader announces a message M to be signed, which signers forward multicast-style down through the tree. Signers may check M against application-specific validity rules – *e.g.*, verifying that the sequence numbers of log entries increase consecutively – but CoSi's basic signing mechanism is agnostic to the particular set of validation rules applied.
2. **Commitment:** Each node i picks a random secret v_i and computes its individual commit $V_i = G^{v_i}$. In a bottom-up process, each node i waits for a message $(V_j, \hat{V}_j)$ from each node j among its set of immediate children C_i. Node i then computes partial aggregate commit $\hat{V}_i = V_i \prod_{j \in C_i} \hat{V}_j$, then passes $(V_i, \hat{V}_i)$ up to its parent, unless i is the leader.
3. **Challenge:** The leader computes a collective challenge $c = \mathrm{H}(\hat{V}_0 || M)$, and multicasts it down through the tree.
4. **Response:** In a final bottom-up phase, each node i waits to receive a partial aggregate response $\hat{r}_j$ from each of its immediate children $j \in C_i$. Node i now computes its individual response $r_i = v_i - c x_i$, and its partial aggregate response $\hat{r}_i = r_i + \sum_{j \in C_j} \hat{r}_j$. Node i finally passes $\hat{r}_i$ up to its parent, unless i is the root.

Notice that during phase 4, each node i's partial aggregate response $\hat{r}_i$, together with the collective challenge c, forms a valid Schnorr multisignature on message M, verifiable against i's partial aggregate commit $\hat{V}_i$ and the corresponding partial aggregate public key $\hat{X}_i$ from the tree roster. Thus, each node in the tree can immediately check its childrens' responses for correctness, and expose any node producing an incorrect response. While nothing prevents a malicious node i from computing V_i or $\hat{V}_i$ dishonestly in phase 2, i will be unable to produce a correct response in phase 4 unless it knows the discrete logarithm v_i such that $V_i = G^{v_i}$.

The final collective signature is $(c, \hat{r}_0)$, which any third-party may then verify as a standard Schnorr signature by re-computing $\hat{V}_0' = G^{\hat{r}_0} \hat{X}_0^c$ and checking that $c \stackrel{?}{=} \mathrm{H}(\hat{V}_0' || M)$. The scheme's correctness stems from the fact that $\hat{V}_0 = G^{\sum_i v_i}$, $\hat{r}_0 = \sum_i v_i - c \sum_i x_i$, and $\hat{X}_0 = G^{\sum_i x_i}$. The scheme's unforgeability stems from the fact that the hash function makes c unpredictable with respect to $\hat{V}_0$, and the collective cannot produce the corresponding response $\hat{r}_0$ without the (collective) knowledge of the secret key x_i of *every node* i whose public key is aggregated into $\hat{X}_0$. These properties are direct implications of the structure of Schnorr signatures, and are neither novel nor surprising theoretically and in fact proven secure in [8, 54], although we are not aware of prior practical systems that have actually implemented efficient signing trees of this kind.

3.5 Contributory Signing Protocol

We now relax the simplifying assumption that only the leader proposes values to sign. Signers can use CoSi in an *contributory* fashion such that the resulting signature is on a cryptographic summary of items proposed by all signers.

The CoSi protocol works like before, but the message M now corresponds to the top of a Merkle tree built on potentially application-specific contributions from all signers. The signers build the Merkle tree of contributions in a bottom-up process following a leader announcement of a new round. Each signer proposes its contribution, hashes it with the contributions received from its children and passes it upward to its parent who repeats the process until it reaches the leader (the top of the Merkle tree). In response, the leader adds its own contribution which results in a top hash — a cryptographic summary of all items propagated throughout the tree. Finally, the leader passes down the top hash along with a *proof* that he properly incorporated the children's contributions. In turn, each signer repeats the process until each node in the tree receives the top hash and its own proof that its contribution was included and therefore witnessed by other signers.

Application semantics determine how signers combine their contributions during this process. In an *aggregation mode*, each signer proposes a unique value, which is aggregated into a single, collective value that is a function of each signer's input. This mode is useful for generating cryptographic random numbers for example. In a *logging mode*, each signer's proposed contribution is opaque to others, consisting of a hash of a single or many values (*e.g.*, consisting of items its clients requested to log or time-stamp). Each signer's signature represents an acknowledgement that a particular contribution was indeed seen and merged together. This mode, in fact, becomes a mechanism for tamper-evident logging. A *validation mode* gives each signer an opportunity to view and validate other signer's contributions and withhold its signature, raising an alarm and alerting others to a potential wrongdoing, if some contribution is invalid according to application-specific checks. Therefore, the collective signature not only conveys that each contribution was seen at a particular time but also that it was validated by each signer. While this is the most expensive mode with $O(N)$ as opposed to $O(\log n)$ cost, it gives signers a unique ability to enforce even fine-grain correctness checks. For example, if applied to Certificate Transparency, each signer could verify the newly proposed batch of certificates to ensure that they are valid and also endorsed by authorized entities.

Regardless of the contributory mode the signers use for CoSi, each signer only contributes its signature share if the resulting top hash is valid with respect to a specific requirements of each mode.

4. AVAILABILITY AND DYNAMICS

This section addresses the challenge of handling both temporary server failures and long-term changes in a cothority's roster. We first explore two alternative approaches to handling signer failures, which may be used either separately or in combination; we then address leader failures and roster evolution.

4.1 Signing Exceptions

In the first approach, we assume that failures of CoSi signers are rare and brief, such that at most a few nodes, *e.g.*, $O(\log N)$, are expected to be offline at any given time. This stability might be satisfied by administrative fiat, for example: *e.g.*, the existing members of a CoSi community might admit new members only after the candidate has demonstrated adequate provisioning and high availability. Individual participants might ensure this high availability, for example, through conventional state machine replication such as Paxos [40, 60] or BFT [16] across a small number of centrally-managed physical machines. Such replication would operate below and be invisible to the CoSi protocol, so we do not delve into the details here.

If server failures are rare but do occasionally happen, we can account for a few failures in any given round by relaxing the demand that each collective log entry be signed with the combination of *all* nodes' public keys. Instead, if in any phase of the basic protocol some node i discovers that one of its immediate children $j \in C_i$ is temporarily offline or unresponsive, i leaves j and any descendants of j out of phase 4 of the CoSi protocol. The aggregate response that node i passes upward to its parent will therefore be a valid signature not for i's "ideal" aggregate public key $\hat{X}_i = \prod_{j \in D_i} X_j$, but rather for a modified aggregate public key $\hat{X}'_i = \hat{X}_i \hat{X}_j^{-1}$.

Therefore, when a failure occurs, signer i indicates the j's missing share of the collective response by including a list of *exceptions* in the message it passes upward, one for each node j whose contribution is missing from i's aggregate response $\hat{r}_i$ due to the failure of node j. Each such exception indicates the aggregate public key $\hat{K}_j$ of the missing node j, the total number of descendants of j whose contributions are missing, j's aggregate contribution $\hat{V}_j$ to the collective commitment $\hat{V}_0$, and any Merkle path information required to verify this information in the cothority's tree roster.

For example, suppose the spanning tree on which the CoSi protocol operates is defined during setup or incremental group evolution via a Merkle tree roster listing the public key X_i, aggregate subtree key $\hat{X}_i$, and number of descendants $|D_i|$ of each node i. Then if a child j of node i fails in a given round, node i produces an exception record for j that includes the appropriate records in the Merkle tree roster from the root node down through node j, enabling anyone to verify the validity of the partial key under which the signature was issued with respect to a well-known, top-level group configuration.

To ensure that a collective signature containing exceptions remains unforgeable, the hash used to compute the challenge in phase 3 above must depend on not just the "complete" aggregate commit $\hat{V}_0$, but also all the *individual* commits V_i for nodes i whose commits might need to be removed from the aggregate. For this reason, we now compute the collective challenge as $c = \mathrm{H}(T||M)$, where T is the root of a Merkle tree whose structure exactly mirrors that of the static tree roster, and records the individual and aggregate commits $(V_i, \hat{V}_i)$ for each node i in this round. To verify a collective signature containing exceptions, the verifier removes both the public key contributions K_i and the corresponding commit contributions V_i for each missing node i, by multiplying the aggregate public key with K_i^{-1} and multiplying the aggregate commit with V_i^{-1} for each missing i.

In addition to validating any exception records to verify that the modified commitment and modified aggregate public key indeed reflects a correct subset of signers, the configuration policy of a specific cothority must define a quorum, or lower bound on the number of signers whose public keys must be included in (or maximum number of nodes that may be missing from) the modified aggregate key, against which the collective signature will be verified.

4.2 Life Insurance Policies

While signing exceptions work, they make collective signatures larger and no longer constant-size, and verifying those signatures is slightly more complex for clients.

An alternative approach ensures that if a given node j fails, then some set of other nodes can collectively take over for j's role in the collective signature generation process that requires j's private ephemeral signing key x_j. This approach relies on verifiable secret sharing (VSS) [30].

Each signer j splits its private signing key into k verifiable shares using a degree t polynomial, so any t-of-k share holders can reconstruct the secret but fewer than t receivers learn nothing about it. The k other signers holding shares of j's signing key serve as *insurers* for j. The number of insurers k need not be large: for example, $k = O(\log N)$ suffices provided these k are chosen randomly from the N total servers and a constant fraction of the N servers are honest.

Upon receiving their shares of j's private key, the insurers issue a confirmation of this fact, which collectively serve as a publicly-verfiable *proof-of-insurance* for j. During a CoSi round, if a quorum of j's insurers agrees that j has failed, they use largely standard VSS techniques to reproduce j's missing component of the collective signature.

If j fails after phase 2 (commitment) but before phase 4 (response), j's insurers must be able to reconstruct not only j's private signing key x_j but also j's ephemeral secret

v_j. To address this challenge, in phase 2 signer j generates shares of its ephemeral secret v_j and encrypts them for the same nodes holding shares of its signing key, so that the insurers can reconstruct v_j if needed. A malicious signer j could produce incorrect shares of v_j in phase 2, but we treat this readily detectable condition as a more serious "hard failure" demanding administrative action. We can revert to the exception mechanism above to preserve liveness in the face of such, hopefully rare, hard failures.

One important issue is how each signer j chooses its insurers to hold shares of its private signing key x_j. On one hand, it might be reasonable for each node j itself to have sole choice of its insurers, since it is ultimately j's secret they are supposed to protect. This freedom could create a DoS attack vector, however, in which a set of malicious nodes deliberately choose colluding insurers that will all go offline together at a time of the attacker's choosing, ensuring that these nodes' secrets cannot be reconstructed.

An alternative is for j's choice of insurers to be random but verifiable by others, so that j cannot control the choice of its insurers but nevertheless receives a strong probabilistic guarantee that no colluding group limited to a given size can reconstruct j's secret signing key unless the insurers agree that j has failed. A potential solution is to choose the insurers through a *lottery*, where each signer receives a deterministic *lottery ticket* created using a hash function applied to some public previous-round output and the singer's identity. Choosing these insurers in a truly bias-resistant fashion is another important challenge that we largely leave to future work, but discuss briefly later in Section 7.

4.3 Leader Failures and View Changes

So far we have treated the leader as a distinguished role, and have addressed only signer failures: if the leader fails the collective will cease operation. In some applications this may be appropriate, *e.g.*, if by design messages to be signed originate only from one source and are to be merely witnessed and validated by the signers. In other environments, however, we would prefer the members of a cothority to play symmetric roles, such that *all* servers can contribute to the messages being signed and a failed leader can be replaced.

We address leader features using a *view change* protocol, a standard technique used in Byzantine agreement protocols [16]. For a given cothority membership roster, we use a well-known, deterministic algorithm to define a schedule of *views*, each numbered view selecting one server in the roster as the view's leader. Given this choice of leader, the rest of the communication tree for that view is similarly defined by a well-known, deterministic function of the cothority roster and the view number.

In each view, all signers expect that view's leader to initiate and complete collective signing rounds at an established rate. Any server who decides that the protocol is not making adequate progress can broadcast a *view change* message proposing a larger view number. As soon as a configured threshold of servers have broadcast view changes for a new view v, the new leader considers v to be active and commences initiation of CoSi rounds with itself as the root.

To guarantee the agreement property, that no two views can make progress simultaneously even in the presence of up to f Byzantine nodes, the view change threshold needs to be $2f+1$ out of $3f+1$ total nodes. However, not all cothorities may require this agreement property, and the protocol can function without it. For example, if a major network-split divides a timestamp cothority into two or more isolated subsets, it may be preferable for each subset to be able to survive and continue making progress, each under a different leader and producing a temporarily separate timeline, until the split is healed. Whether the agreement property is essential or not depends on the semantics of the cothority.

In our current protocol, a view change message is individually signed by the server sending it, so verifying all relevant view change messages can unfortunately take $O(N)$ time and network bandwidth per node. This is not a critical issue provided view changes needed relatively seldom, as we expect, but this could represent a DoS attack vulnerability in the worst case. We expect that collective signing could be extended to cover the *initiation* of a new view as well as rounds within it, so a new leader can produce and exhibit a collective signature as a more compact and efficiently-checkable proof of its leadership, but we leave this enhancement to future work.

4.4 Cothority Evolution

So far we have assumed that a cothority's tree roster – the Merkle tree containing the public keys of all servers and the aggregate public keys of their respective sets of descendants – is static and unchanging. In practice this roster will of course need to change, although we expect it to change much less frequently than signing operations, *e.g.*, on administrative time-scales.

To address this challenge, a cothority can authorize and collectively sign changes to its own roster. When current members wish to add or remove a cothority member, we assume they first administratively discuss and agree on the change using out-of-band communication mechanisms: *e.g.*, meetings, online discussion on E-mail lists, etc. Once a change has been administratively agreed, by adding and/or removing members, the cothority's current leader proposes the new roster, and a policy-defined threshold of current members must collectively sign it. Once validated and collectively signed in this way, the "old cothority" represented by the old roster essentially ceases to function, and a "new cothority" represented by the new roster commences operation. This same mechanism similarly enables existing members to refresh or upgrade their signing keys periodically.

One issue is how clients and other third-parties track and validate cothority roster changes. For example, a client application may have a cothority roster hard-wired into its software, just as web browsers currently contain hard-wired list of root CAs, but the cothority's roster may evolve faster than client software gets updated.

To enable old clients to learn about and validate new cothority rosters, the client can follow a chain of collectively-signed roster-change records forward from the last version it knows about to the current version. For example, if the client has version V_0 and the current roster version is V_2, the client first obtains the change-record from V_0 to V_1 and validates its signature against the V_0 roster, then obtains the change-record from V_1 to V_2 and validates its signature against the V_1 roster. So that clients need not pick through a cothority's entire log to catch up on the latest roster, a cothority can maintain a separate log containing only roster-change events.

A related issue is how long these forward roster validation chains might become, but we do not expect this to be a major problem in practice if roster updates are administratively limited to a reasonable rate, *e.g.*, by batching all roster changes together at most once per month.

5. PROTOTYPE IMPLEMENTATION

We have built and evaluated a working prototype of a cothority server supporting not only the basic CoSi protocol for collective signing, but also demonstrating cothority application functionality for tamper-evident logging, timestamping, public randomness, and voting/agreement functions.

The cothority server prototype is written in Go [36]; its primary implementation consists of 7600 lines of server code as measured by CLOC [22]. The server also depends on a custom 21,000-line Go library of advanced crypto primitives such as pluggable elliptic curves, zero-knowledge proofs, and verifiable secret sharing; our cothority prototype relies heavily on this library but does not use all its facilities. Both the CoSi prototype and the crypto library are open source and available at `https://github.com/DeDiS`.

The cothority prototype currently implements tree-based collective signing as described above, the signing exception protocol for handling server failures, the view change mechanism to handle leader failures, and a voting and vote-tallying mechanism that can be used to validate roster changes as well as for general collective-voting purposes. The alternate, life insurance approach to handling signer failures (Section 4.2) is partly implemented but not yet fully integrated into the prototype.

We evaluated the cothority implementation with Schnorr signatures implemented on the NIST P-256 elliptic curve [4], although the implementation also works and has been tested with other NIST curves as well as the Ed25519 curve [9].

The cothority server is structured in three main layers: the Host layer handling networking, the Signer layer implementing the collective signing protocol, and the Stamper layer implementing Merkle tree logging and timestamping.

The Host layer implements a TCP-based overlay network. Hosts connect to each other forming a communication tree, whose structure is currently defined statically by the roster. The Host layer handles networking functions, detects host failures to determine when to trigger signing exceptions, and attempts to reconnect when TCP connections fail.

The Signer layer builds on the host abstraction to coordinate with other signers and implement collective signing. This layer can also incorporates application-specific messages such as vote records into its processing. When a new node joins, the Signer layer is responsible for dynamically establishing key-pairs with the other node. The Signer layer drives the Announce, Commit, Challenge, Challenge Response phases as described above, as well as a simple voting protocol to support roster evolution.

5.1 Signing Modes and Applications

We implemented three modes of execution for the signing nodes: simple collective signatures as described in Section 3.4, collective signature based on Merkle trees as required for exception-handling as described in Section 4.1, and simple collective signatures paired with individual-host signatures currently required for voting and roster changes.

In the simple key mode, the root simply announces a message and the signers collectively generate a signature.

The Merkle key mode is used by the timestamper application, to consolidate into a single Merkle tree all timestamps that clients submit to each signer for timestamping since the last round. Client may connect to any server and send a StampRequest; after the round completes the client receives in response a collectively signed timestamp log entry and a Merkle path proving to any third party that the client's hash was included.

Finally, to test voting and group evolution we extended signing nodes as to be able to vote on any matter by signing individual votes in addition to collectively signing all votes. The combination of signatures allowed a node to verify all other votes and the correctness of the tally. While this verification process currently incurs $O(N)$ costs, we expect it could be reduced with improved, verifiable tallying methods in the future.

6. EVALUATION

The primary questions we wish to evaluate are whether the cothority architecture is practical and scalable to large numbers, *e.g.*, thousands of participating servers, in realistic scenarios. Important secondary questions are what the important costs are such as collective signing latency and computation costs.

While this paper's primary focus is on the basic CoSi protocol and not on particular applications or types of cothorities, we evaluated the CoSi prototype in the context of the Timestamping application.

6.1 Experimental Setup

We evaluated the prototype on DeterLab [23], using 32 physical machines configured in a star-shaped virtual topol-

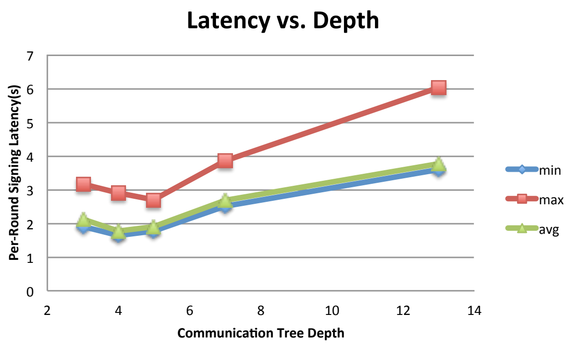

Figure 2: Collective signing latency versus branching factor

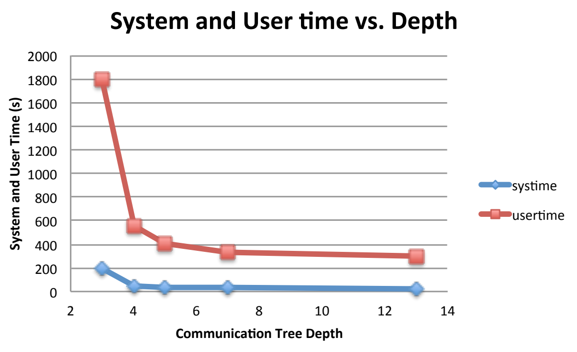

Figure 3: Collective signing latency versus branching factor

ogy. To simulate larger numbers of CoSi participants than available testbed machines, we run up to 128 separate CoSi server processes on each machine. A corresponding set of CoSi client processes on each machine generate load on each server by issuing regular Timestamp requests to the server processes.

To mimic a realistic wide-area environment in which the cothority's servers might be distributed around the world, the virtual network topology imposes a round-trip latency of 100 milliseconds between any two machines. The TimeStampers aggregate messages from their clients and every 10 seconds request the batch of messages to be signed collectively as part of a single aggregate Merkle tree per round.

6.2 Computation Costs

The first experiment focuses on the protocol's per-node computation costs. We expect total signing latency to depend on both network communication latencies and computation time within each node.

In our test framework, on each machine, we run multiple TimeStamper processes, each with an embedded Signer, and an associated client. The timestampers respond to requests and collectively sign timestamps, as described above. One client located on the same physical machine sends one request about every 35 miliseconds to keep the corresponding server loaded; we empirically found this load to be close to the maximum request rate the timestamp servers could keep up with in this environment.

Figure 2 shows how measured System and User time on the most heavily-loaded signing node (typically the root) varies depending on the depth of the configured communication tree. We observed that CPU utilization was higher when the depth was lower, which makes sense as lower depth implies higher branching factor (degree) and hence more computation per node at each level. The number of responses to be processed per Commit and Response round is linear in the number of children a signing node has, and thus linear in the branching factor.

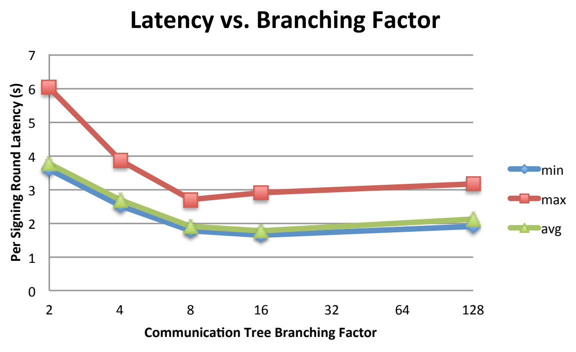

Figure 4: Collective signing latency versus branching factor

With higher branching factors, a node spends more CPU time to process all its children commits and responses. At small depths, we found that reduction in network latency due to the shallow tree was not enough to counterbalance the added computation time required by the signers at each level.

6.3 Collective Signing Latency

Figure 3 shows how total per-round collective signing latency varies with communication tree depth. By increasing depth, we increase the root to leaf round-trip latency, by about 100 milliseconds per unit of depth added. Increasing depth, however, implies decreasing the branching factor, decreasing the CPU time spent per node. In this environment we find the best results for depths 4 and 5. For depth 3, computation time dominates, while for depths greater than 5 network latencies dominate. The current CoSi prototype makes no attempt to parallelize its computations, however, so optimization and parallelization of the computations might make small depths more attractive.

For comparison, Figure 4 shows the relationship between per-round latency and branching factor (maximum number of children per CoSi server), confirming that overall latency is minimized at intermediate depth and branching factors.

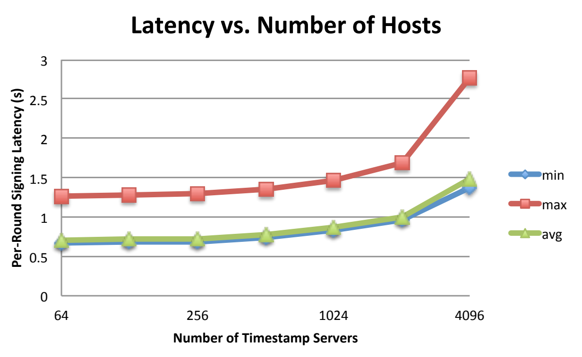

Figure 5: Collective signing latency versus number of participating servers

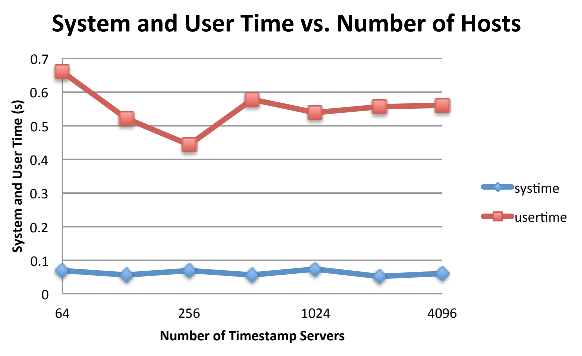

Figure 6: Per-node, per-round computation cost versus number of participating servers

6.4 Scalability

The next experiment evaluates the scalability of the CoSi protocol to large numbers of hosts, varying from 64 to 4096 hosts timestamp servers, while maintaining a constant communication tree depth of 4.

Latency increases with the number of hosts as we would expect, but total latency scales gradually. Per-round collective signing latencies average under 1.5 seconds for 4096 participants, and the maximum latency we observed over hundreds of runs was under 3 seconds. Given that authority protocols are often moderately latency-tolerant, often operating at timescales of minutes or hours, these results suggest that collective signing should not create a serious performance bottleneck for building cothorities.

Figure 6 similarly shows how computation costs scale with total number of hosts. We find that there is not much variation between running on 64 up to 4096 hosts with a constant depth.

6.5 Client Load

As discussed above, for each CoSi server a separate process on the same physical machine acted as a client to create TimeStamp requests at a constant rate. We tested the system under a variety of client load rates, from one request every 5 seconds to one request every 35ms – the last case amounting to 30 requests per second on each timestamp server. Client loads within this range did not significantly affect the collective signing latencies we observed, however, so we omit these graphs.

At the largest-scale experiments with 4096 timestamp servers spread across 32 physical testbed machines (128 servers per machine), each physical machine effectively handled an aggregate client load of about 3,700 timestamp requests per second, or 120,000 timestamp requests per second across the 4096-server collective. Further, the current CoSi implementation and timestamp server code is largely unoptimized and completely unparalllelized within each server and with more powerful, unshared machines, we expect that each server could readily handle much larger loads.

6.6 Leader Failures Cost

We have also evaluated the cost of handling leader failures using view changes as described in Section 4.3. Figure 7 shows the latencies of a series of rounds involving several view changes, caused by randomly-injected leader failures, in a tree of 4096 host servers. Figure 8 shows zooms on the first view change in this trace.

Once a root node fails, another leader is randomly chosen from the remaining nodes. Our view change implementation currently does not rebalance the tree after the change, which results in a heavily unbalanced tree if the chosen node is located far away from the root in the tree structure. Therefore, the baseline latency increases significantly after the first view change because of the increased depth of the tree, but stays consistent thereafter. We expect that proper tree balancing will mitigate this latency increase.

Additionally, each view change constitutes an idle round. Hence, the servers must process all outstanding timestamp requests during the first successful round after the view change, which results in an increased latency for that round. The overall results, however, indicate that leader failures can be handled efficiently.

7. DISCUSSION AND FUTURE WORK

This paper's primary technical focus has been on the CoSi protocol for collective signing, and we make no pretense to have addressed all the important issues relevant to applying CoSi in any particular cothority application context. However, we briefly revisit some of the motivating applications introduced in Section 2 in light of the above implementation and evaluation results.

Tamper-Evident Logging and Timestamping.

While the current CoSi prototype is basic, it nevertheless already implements the essential functionality of a classic tamper-evident logging and timestamping authority [1, 37].

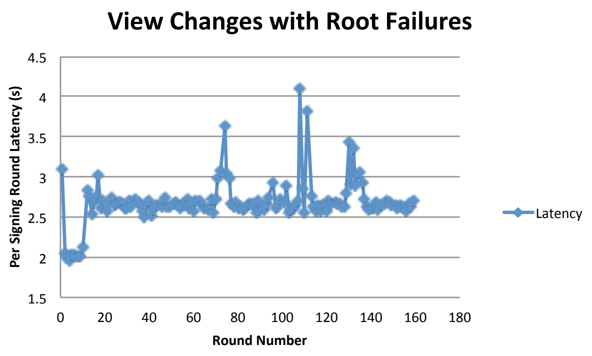

Figure 7: Latency of randomly imposed view changes over number of rounds

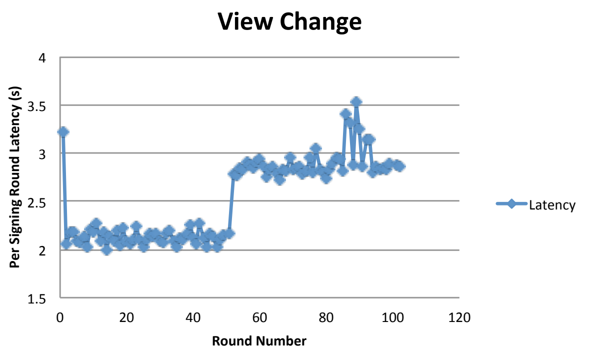

Figure 8: Latency of a single view change

As neither the leader nor any signer can produce a collective signature without the participation of a quorum of the potentially large collective, such a timestamp cothority can offer much stronger protection against the equivocation, history-rewriting, or log-entry back-dating attacks that a centralized timestamp service can mount if compromised.

Public Randomness Cothorities.

While not our present focus, the current CoSi prototype also effectively implements a collective public randomness service that could improve the trustworthiness of public randomness authorities [58, 61]. Notice that in phase 2 of the signing protocol (Section 3.4) each server i commits to a fresh random secret v_i, contributing to a collective random secret $\sum_i v_i$ that no participant will know unless *all* signers are compromised or the discrete-log hardness assumption fails. The final response produced in phase 4 depends unpredictably and 1-to-1 on this random secret and the collective challenge c. Thus, we can use the final aggregate response $\hat{r}_0$ as a per-round public random value that was collectively committed in phase 2 but will be unpredictable and uncontrollable by any participant unless all signers are colluding.

While these random outputs will be unpredictable and uncontrollable, our current prototype cannot fully guarantee that they are *unbiased*, due to its reliance on the signing exception mechanism for availability. In particular, if a malicious leader colludes with f other signers, then the leader can control whether these colluders appear online or offline to produce up to 2^f different possible final aggregate responses with different exception-sets, and choose the one whose response is "most advantageous" to the leader, just before completing phase 4 of the protocol. The life insurance policy approach to handling signer failures (Section 4.2), once implemented, addresses this bias issue by ensuring that *every* node's secret is unconditionally incorporated in the final response, unless a catastrophic failure makes some server's secret unrecoverable even via secret-sharing.

Certificate Cothorities.

Replacing the current "weakest-link" CA system with a "strongest-link" cothority may be the most potentially compelling and immediately urgent use-case for CoSi. In a cothority architecture in which not just one CA but many or all of them inspect and collectively sign each certificate, stolen CA keys such as those of DigiNotar [5, 13] and Comodo [12] would not by themselves be usable to sign certificates that a standard web browser would accept. Not just CAs but browser vendors and security companies could incorporate monitoring servers into the certificate cothority as signers, to watch for and proactively block the signing of unauthorized certificates, such as certificates proposed by a CA that is not recorded as having contractual authority over a given domain. Unlike Certificate Transparency [43, 44] and other logging approaches [6, 39, 64, 73], web clients need not delay web page loading to check separate logging or monitoring servers, since CoSi binds these decentralized checks proactively into each certificate's collective signature.

Deploying a certificate cothority would of course require addressing many additional issues beyond the basic collective signing mechanism covered here, not just technical but also organizational and political. One important technical challenge is backward compatibility and incremental deployment. We anticipate that current root CAs might gradually transition their root signing keys into cothority keys, with their current sets of delegated CAs (and any other cooperative root CAs) serving as signers. This way, each root CA could transition independently at its own pace, driven by pressure from users and browser vendors to increase security. Web browsers would of course need to be gradually upgraded to support Schnorr signatures in addition to the common RSA and DSA schemes. During their transition period root CAs could retain traditional root CA keys for use in older web browsers while embedding a cothority key instead into suitably upgraded browsers.

Other Types of Cothorities.

It should be possible to apply the tree-based scaling techniques explored here not only to collective signing but many other cryptographic primitives amenable to tree-structured aggregation of operations. A large-scale cothority might collectively decrypt ElGamal [28] ciphertexts at particular future dates or on other checkable conditions, to implement time-lock vaults [56, 63] or fair-exchange protocols [32] for example. More generally, the cothority architecture might present an interesting alternative to the currently-popular BitCoin blockchain mechanism [57] as a foundation on which to build decentralized ledgers and cryptocurrencies.

8. RELATED WORK

The theoretical foundations for cothorities already exist in the form of threshold signatures [10, 70], aggregate signatures [11, 47, 48], and multisignatures [8, 54]. Threshold signatures allow some subset of authorized signers to produce a signature, however, often making it impossible for the verifier to find out which signers were actually involved. In aggregate signatures, a generalization of multisignatures, signers produce a short signature by combining their signatures on individual messages through an often serial process. On the other hand, multisignatures closely fit the requirements of CoSi for security, efficiency and the simplicity of generation across many signers. However, to our knowledge these primitives have been deployed only in small groups (e.g., ≈ 10 nodes) in practice.

Merkle signatures [14,51,53], which also use Merkle trees in the signing process, allow a single singer to efficiently produce a large number of one-time signatures verifiable under the same public key, as opposed to allowing a number of singers to sign a message under their own public keys.

Timestamping services [1, 37] enable clients to prove the existence of some data (*e.g.*, contracts, research results, copyrightable work) before a certain point in time by including it in a timestamped log entry. Typically, a trusted third party acts as a timestamping authority [25, 33, 65] and has a unilateral power to include, exclude or change the log of timestamped data.

Even tamper-evident logging services are vulnerable to *equivocation*, however, where a malicious log server rewrites history or presents different "views of history" to different clients. Solutions include weakening consistency guarantees as in SUNDR [46], adding trusted hardware as in TrInc [45] or utilizing a trusted party [67] in some fashion. Certificate Transparency [43, 44] and NIST Randomness Beacon [58] are examples or application-specific logging services that exemplify issues related to a trusted-party design paradigm.

Certificate Transparency [43, 44] requires CAs to insert newly-signed certificates into public logs, which independent auditors and monitors may check for consistency and invalid certificates. Unfortunately, a single log server still has an unilateral power to sign off on certificates, and passive monitoring and auditing can detect misbehavior only retroactively *after* it has occurred, placing victims in a race with the attacker.

NIST Randomness Beacon [58] logs the random values it produces by signing them using its own private key and chaining them with the previously produced values. While a dishonest beacon cannot selectively change individual entries, it can rewrite the entire history from a chosen point and present different views of the history to different clients. Additionally, there is no guarantee of freshness of the published randomness. While the quality of the output is likely not affected if the beacon precomputes the randomness, the beacon gets to see these values beforehand effectively becoming vulnerable to insider attacks.

9. CONCLUSION

This paper has demonstrated how using theoretically established and well-understood cryptographic techniques, we can build strongest-link collective authorities whose trust may be distributed across not just a few but hundreds or thousands of servers. The encouraging scalability and performance results we have observed with our CoSi prototype lead us to believe that large-scale cothorities are practical. If this is the case, we feel that there may be no technical reason to settle for the centralized, weakest-link security offered by current designs for today's common types of critical authorities. We can and should demand better security from our authorities.

10. REFERENCES

[1] C. Adams and D. Pinkas. Internet X.509 public key infrastructure time stamp protocol (TSP). 2001.

[2] L. M. Adleman. Implementing an electronic notary public. In *Advances in Cryptology*, 1983.

[3] M. Alicherry and A. D. Keromytis. DoubleCheck: Multi-path verification against man-in-the-middle attacks. In *14th IEEE Symposium on Computers and Communications (ISCC)*, July 2009.

[4] American National Standards Institute. Elliptic curve digital signature algorithm (ECDSA), 2005. ANSI X9.62:2005.

[5] C. Arthur. DigiNotar SSL certificate hack amounts to cyberwar, says expert. *The Guardian*, Sept. 2011.

[6] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski. ARPKI: Attack resilient public-key infrastructure. In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2014.

[7] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, and K. R. B. Butler. Forced perspectives: Evaluating an SSL trust enhancement at scale. In *Internet Measurement Conference (IMC)*, Nov. 2014.

[8] M. Bellare and G. Neven. Multi-signatures in the plain public-key model and a general forking lemma. In

ACM conference on Computer and communications security, 2006.

[9] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[10] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the Gap-Diffie-Hellman-Group signature scheme. In *Public key cryptography - PKC 2003*. 2002.

[11] D. Boneh, C. Gentry, B. Lynn, H. Shacham, et al. A survey of two signature aggregation techniques. *RSA cryptobytes*, 2003.

[12] P. Bright. How the Comodo certificate fraud calls CA trust into questions. *arstechnica*, Mar. 2011.

[13] P. Bright. Another fraudulent certificate raises the same old questions about certificate authorities. *arstechnica*, Aug. 2011.

[14] J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume. Merkle signatures with virtually unlimited signature capacity. In *Applied Cryptography and Network Security*, 2007.

[15] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[16] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[17] S. Checkoway, M. Fredrikson, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham. On the practical exploitability of Dual EC in TLS implementations. In *USENIX Security Symposium*, 2014.

[18] S. Chokhani and W. Ford. Internet X.509 public key infrastructure certificate policy and certification practices framework. 1999. RFC 2527.

[19] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, Aug. 2009.

[20] I. Dacosta, M. Ahamad, and P. Traynor. Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties. In *17th European Symposium on Research in Computer Security (ESORICS)*, Sept. 2012.

[21] I. Damgård. On Σ-protocols, 2010.

[22] A. Danial. Counting Lines of Code. `http://cloc.sourceforge.net/`.

[23] DeterLab network security testbed, September 2012. `http://isi.deterlab.net/`.

[24] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.

[25] DigiStamp - Trusted TimeStamp Authority. `https://www.digistamp.com/`.

[26] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *12th USENIX Security Symposium*, Aug. 2004.

[27] Electronic Frontier Foundation. The EFF SSL Observatory, 2011.

[28] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. Blakley and D. Chaum, editors, *Advances in Cryptology*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1985.

[29] C. Evans and C. Palmer. Certificate pinning extension for HSTS, Sept. 2011. draft-wing-v6ops-happy-eyeballs-ipv6-01.

[30] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Foundations of Computer Science*, 1987.

[31] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *IACR International Cryptology Conference (CRYPTO)*, pages 186–194, 1987.

[32] M. K. Franklin and M. K. Reiter. Fair exchange with a semi-trusted third party. In *ACM Conference on Computer and Communications Security*, Apr. 1997.

[33] Free Timestamping Authority. `http://www.freetsa.org/`.

[34] A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (SSL) protocol version 3.0, Aug. 2011. RFC 6101.

[35] B. Gellman and E. Nakashima. U.S. spy agencies mounted 231 offensive cyber-operations in 2011, documents show. *The Washington Post*, Aug. 2013.

[36] The Go programming language, Jan. 2015. `http://golang.org/`.

[37] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 1991.

[38] J. Hoffman-Andrews. Verizon injecting perma-cookies to track mobile customers, bypassing privacy controls, Nov. 2014.

[39] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure. In *22nd International Word Wide Web Conference (WWW)*, Apr. 2014.

[40] L. Lamport. The part-time parliament. Technical Report SRC-049, Systems Research Center, Sept. 1989.

[41] A. Langley. Further improving digital certificate security. *Google Online Security Blog*, Dec. 2013.

[42] A. Langley. Maintaining digital certificate security . *Google Online Security Blog*, Mar. 2015.

[43] B. Laurie. Certificate transparency. *ACM Queue*, 2014.

[44] B. Laurie, A. Langley, and E. Kasper. Certificate transparency, June 2013. RFC 6962.

[45] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, 2009.

[46] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[47] S. Lu, R. Ostrovsky, A. Sahai, H. Shacham, and B. Waters. Sequential aggregate signatures and multisignatures without random oracles. In *Conference on Theory and application of cryptographic techniques (EUROCRYPT)*. 2006.

[48] D. Ma and G. Tsudik. A new approach to secure logging. *ACM Transactions on Storage*, 5(1), Mar. 2009.

[49] M. Marlinspike. SSL and the future of authenticity. In *BlackHat USA*, Aug. 2001.

[50] M. Marlinspike and T. Perrin, Ed. Trust assertions for certificate keys, Jan. 2013. Internet-Draft draft-perrin-tls-tack-02.txt (Work in Progress).

[51] R. C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford University, 1979.

[52] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology (CRYPTO)*, 1988.

[53] R. C. Merkle. A certified digital signature. In *Advances in Cryptology (CRYPTO)*, 1989.

[54] S. Micali, K. Ohta, and L. Reyzin. Accountable-subgroup multisignatures. In *ACM conference on Computer and Communications Security*, 2001.

[55] M. Michels and P. Horster. On the risk of disruption in several multiparty signature schemes. In *Advances in Cryptology (ASIACRYPT)*, 1996.

[56] M. C. Mont, K. Harrison, and M. Sadler. The HP time vault service: Exploiting IBE for timed release of confidential information. In *12th International World Wide Web Conference (WWW)*, May 2003.

[57] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Oct. 2008.

[58] NIST Randomness Beacon. `http://www.nist.gov/itl/csd/ct/nist_beacon.cfm`.

[59] K. Ohta and T. Okamoto. Multi-signature schemes secure against active insider attacks. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Jan. 1999.

[60] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, June 2014.

[61] Randomness and Integrity Services Ltd. `random.org`, 1998.

[62] Reddit: NIST Randomness Beacon. `http://www.reddit.com/r/crypto/comments/21apkx/nist_randomness_beacon/`.

[63] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, Mar. 1996.

[64] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.

[65] Safe Creative Timestamping Authority (TSA) server. `http://tsa.safecreative.org/`.

[66] B. Schneier. US Offensive Cyberwar Policy. *Schneier on Security*, June 2013.

[67] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. volume 2, pages 159–176, May 1999.

[68] C.-P. Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology (CRYPTO)*, 1990.

[69] J. Sermersheim. Lightweight directory access protocol (ldap): The protocol. 2006.

[70] V. Shoup. Practical threshold signatures. In *EUROCRYPT*, 2000.

[71] C. Soghoian and S. Stamm. Certified lies: detecting and defeating government interception attacks against ssl. In *Financial Cryptography and Data Security*, Feb. 2011.

[72] How useful is NIST's Randomness Beacon for cryptographic use? `http://crypto.stackexchange.com/questions/15225/how-useful-is-nists-randomness-beacon-for-cryptographic-use`.

[73] P. Szalachowski, S. Matsumoto, and A. Perrig. PoliCert: Secure and flexible TLS certificate management. In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2014.

[74] Tor: Anonymity Online. `https://www.torproject.org`.

[75] Tor directory protocol, version 3. `https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt`, 2014.

[76] V. Venkataraman, K. Yoshida, and P. Francis. Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast. In *14th International Conference on Network Protocols (ICNP)*, Nov. 2006.

[77] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference (USENIX ATC)*, June 2008.