Collective Signing

Dylan Visher

April 13, 2015

1 Abstract

In today’s age of increasingly powerful state-level hacking activities, we need to find a means for creating a trusted authority. As they exist today, authorities are centralized, resulting in "weakest-link" security. We propose a system of cothorities, collective authorities, to guarantee "strongest-link" security. We have built an infrastructure enabling thousands of participants to participate in the collective signing protocol: witnessing, validating, and co-signing an authority’s public actions with minimal overhead. We have implemented a timestamping application on top of the collective signing protocol to test its scalability over 4000 widely distributed participants.

2 Introduction

Currently we rely on authorities to provide security critical services. Authorities, conventionally, are limited to being one (or a few) centralized servers generally under the same administrative domain. The Tor directory service attempts to protect the client by demanding the consensus of a small group of servers. However, as recent large-scale attacks have shown, this is no longer a realistic prevention mechanism.

We propose to replace these centralized authorities into large-scale collective authorities, cothorities. We split trust across many independently-run servers and guarantee strongest-link security. Clients can validate a cothority’s output using simple cryptographic operations similar to conventional signature verification.

Similar work with cothorities exists already, however, they have been unable to be deployed to large groups, generally 10 nodes in practice. The system we propose is able to scale these same concepts across a much larger scale in practical arguments.

For optimal understanding of below please read: Decentralizing Authorities into Scalable Strongest-Link Cothorities
3 Personal Contributions

- Designed Network Architecture
- Developed Networking Stack for Testing and Deployment
- Tested on simple LAN network, PlanetLab and DeterLab
- Built infrastructure for logging and result aggregation
- Collaboratively implemented viewchanges and voting
- Contributed to paper: Decentralizing Authorities into Scalable Strongest-Link Cothorities

4 Overall Architecture

The collective-signing protocol needs for each collective-signing node to be organized in a tree overlay network. Each node needs to be able to communicate with its children and its parent. Thus there is a Host layer beneath the collective-signing protocol that abstracts away this communication layer.

5 Network Architecture

The Network Architecture is built around two simple concepts: a connection and a host. A connection is responsible for peer-to-peer message passing. In conjunction, these interfaces allows for the collective-signing protocol to send messages to their parents and children in the collective-signing tree.

These interfaces have two main implementations currently: in-memory CSP-style bidirectional channel-based connections for testing on a single machine, and TCP connections for testing both on a single machine and on a large scale distributed collection of servers.

In the future it would be good to implement our own higher-level network protocol built on top of UDP for "connectionless" message passing. This would allow our nodes to be tested with even greater fan-out and not be restricted by file-descriptor limits.

5.1 Connections

A connection is responsible for the bi-directional communication between two Hosts.

// Conn is an abstract bidirectional connection.
// Conn abstracts away the network connection and data-format.
type Conn interface {
    // Name returns the name of the "to" end of the connection.
    Name() string
// PubKey returns the public key associated with the peer.
PubKey() abstract.Point
// SetPubKey sets the public key associated with the peer
SetPubKey(abstract.Point)

// Put puts data to the connection.
Put(data BinaryMarshaler) error
// Get gets data from the connection.
// Blocks until successful or network error.
// Returns io.EOF if the channel has been closed.
Get(data BinaryUnmarshaler) error

// Connect establishes the connection.
// Must be called before Put/Get.
Connect() error

// Close closes the connection.
// Any pending Put or Get’s will return io.EOF.
Close()

// True if the connection has been closed
Closed() bool

}  

When using a "Conn" one must first use the Connect() method in order to fully establish the connection. A connection prior to calling Connect() merely represents the possibility of a connection between the two, and has not connected using the network layer, etc.

The Put and Get methods take in BinaryMarshalers and BinaryUnmarshalers respectively. The BinaryMarshaler interface specifies the single method MarshalBinary() which returns a slice of bytes ([]byte). The BinaryUnmarshaler is similar except it takes in a slice of bytes and initializes itself. By using this interface it allows for the uniform treatment of objects so as long as your object has a MarshalBinary() method, it can be marshaled by the connection and sent over, and the as long as your object has a UnmarshalBinary([]byte) method, it can be unmarshaled on the other side. This allows the calling code to be cleaner as they no longer have to explicitly marshal an object before sending it, or unmarshal bytes before receiving. It also allows the caller explicit control of the wire-format of their data.

The BinaryMarshaler and BinaryUnmarshaler interfaces are both parts of the standard library so it is relatively idiomatic code that other contributors can easily interface with.

In addition, we don’t let connections return return newly created objects. Instead, by taking in an object for Get, we allows the caller explicit control over program garbage as they can reuse objects.
5.1.1 Go Routines: In-Memory Connections

In order to test the connection layer, a similar connection layer is built on Go’s in-memory channels. These channels are CSP style channels and allow bi-directional communication. However, in order to make each Host know which messages were for it, the GoConn is built as two bidirectional channels, one to read from and one to write to.

Ultimately, when making distributed systems it is essential to have a way to test it at a small scale in as close to a deterministic fashion as possible. Though channels are not quite deterministic, by making the channels require a synchronization on each read or write (i.e. both goroutines must be reading and writing simultaneously) the program becomes easier to reason about. If there are problems with the TCP connections that are not with the channel based connections, then, before looking elsewhere, you can make the channels be buffered to simulate the asynchronous nature of TCP, and you can also add in a “random” delay to simulate latency between the two communicators. Ultimately this makes as much of the program inspectable as possible. All goroutines can be monitored and their stack traces can be printed and analyzed, deadlocks can be detected and reported, etc.

5.1.2 TCP Connections

The TCPConn implementation parallels GoConn’s except they use TCP connections as the primary transport mechanism. It is an asynchronous transport that can be used across multiple computers. TCPConns are the connections that are used for our tests on larger-scale setups.

5.2 Hosts

The Host layer is responsible for abstracting away the communication tree that is necessary for collective-signing. Given that the communication tree changes every view, it intuitively makes sense for the views to be contained by the Host.

Below is the core subsection of the Host interface:

```go
// Host is an abstract node on the Host tree.
// Representing this tree, it has the ability to:
// PutUp to its parent
// PutDown to its children
// Get messages from its peers
//
// It is up to the caller to de-multiplex the messages sent back from Get.
//
// A Host also implements multiple views of this tree.
// Each view is a unique tree (parent, children) configuration.
// Applications using a Host can specify which view to use for operations.

type Host interface {
    // Name returns the name of this host.
}
```
Name() string

// Peers returns a mapping from peer names to Connections.
Peers() map[string]Conn

// NewView creates a NewView to operate on.
// It creates a view with the given view number, which corresponds to the tree
// with the specified parent and children.
NewView(view int, parent string, children []string, hostlist []string)

// Returns map of this host's views
Views() *Views

// PutUp puts the given data up to the parent in the specified view.
// The context is used to timeout the request.
PutUp(ctx context.Context, view int, data BinaryMarshaler) error

// PutDown puts the given data down to the children in the specified view.
// The context is used to timeout the request.
PutDown(ctx context.Context, view int, data []BinaryMarshaler) error

PutTo(ctx context.Context, host string, data BinaryMarshaler) error

// Get returns a channel on which all received messages will be put.
// It always returns a reference to the same channel.
// Multiple listeners will receive disjoint sets of messages.
// When receiving from the channels always receive from both the network
// messages channel as well as the error channel.
Get() (chan NetworkMessg, chan error)

// Connect connects to the parent in the given view.
// Must be called before calling Put and Get.
Connect(view int) error
ConnectTo(host string) error

// Listen listens for incoming connections.
Listen() error

// Close closes all the connections in the Host.
Close()

// SetSuite sets the suite to use for the Host.
SetSuite(abstract.Suite)

// PubKey returns the public key of the Host.
PubKey() abstract.Point
The core interface of a host is the ability to PutUp (send to the parent) for a given view, and PutDown (send to all children), for a given view, and Get which return a stream of all messages that it receives. The collective signing protocol uses these functions to communicate easily in the network tree structure.

In order to establish the connections for the host, Host first "Listens" to incoming connection requests, and then once a request is received, they trade public keys to reveal and prove their identities. In order to ease the setup of connections the child for a view is always responsible for connecting to the parent for the given view. If this was not the case, there could be a data race where both child and parent initiate connections with the other, both accept, then they now have two TCP connections when in fact there should be one.

Each Host has a plugable crypto suite. The default is the crypto nist package, however the application can opt in to use a more optimized crypto suite like the optimized Edwards curve library at [github.com/dedis/crypto](https://github.com/dedis/crypto).

5.2.1 Views

In order for the collective signing to be reliable even with malicious nodes, it is necessary that there is a means for changing the tree topology. That way if there is a malicious root node, the configuration can change to have a different root and that node can be removed.

Since the host’s job is to abstract away the tree structure, it is necessary for the host to know about the views. Each Host has multiple views each view associated with a view number, and within each view there are children and a parent. This way a host can act on multiple views at the same time, getting from all of them, putting to a parent for a specific view, etc.

5.2.2 TCP Hosts

TCP Hosts must call the Listen function to start listening for inbound connections. When another host attempts to connect they undergo a public key pair swapping process and verification of the other node.

5.2.3 Go Host: Mocking TCP Hosts

GoHosts attempt to mock TCP Hosts to be able to be used for single-host testing. This means adding a listen interface where it doesn’t exactly make sense, but the key pairs need to be established. In order for GoHosts to not send messages to other peers before the connection is fully established, we keep track of which GoConn’s have successfully undergone the key pair swap process and are ready to use.

5.2.4 Design Quirks

Initially thought GetUp, GetDown, PutUp, PutDown, but then we realized we would get an off by one error once errors were allowed, it would be possible to
be Getting from Below messages meant for different rounds etc. To solve this problem we moved closer towards an event-driven architecture where the calling code receives a stream of messages and is responsible for demultiplexing it, thus GetUp, GetDown becomes just Get. Since we were sending from and receiving into Binary(Un)Marshalers it was impossible to tell which view it was destined for. This restriction also meant that GetUp, GetDown, would not be possible to implement with viewchanges given that application writers would want to define which views are needed.

Also the Host interface ultimately exemplified one of the shortcomings of the Conn interface earlier. To keep the Host consistent with the Conn interface, Put* and Get both took Binary(Un)Marshalers. However this proved to be awkward in the case of PutDown. PutDown had the interface PutDown([]BinaryMarshaler), but since BinaryMarshaler is an interface PutDown would not accept []Msg as its argument, even though a Msg satisfies the BinaryMarshaler interface. The reason has to do with how go stores interfaces in memory: a type id and a pointer to a value, which is clearly not the same as just a value. This means that every time calling code needs to call PutDown, it needs to repackage the []Msg into []BinaryMarshaler. This could potentially be cleaned up by using [][]byte instead of [][]BinaryMarshaler and making the corresponding changes to channels. This would put the work on the calling code to marshal and unmarshal the binary data from networking, however, it would not require packaging []Msg into []BinaryMarshaler, which are then each transformed into []byte. It is a tradeoff where there is no clear winner.

The other option would have been to use the reflect package to check if the argument was a slice, and then iterate through it, type casting along the way. This, however, would lose all type safety for the callers to PutDown.

6 Configuration Files

In order to test the collective signing protocol we needed a format for storing these collective-signing trees, and loading them in, either starting up the whole tree locally, or just one node if it is distributed over multiple machines.

To do this I created a simple JSON format, which essentially is a tree of filenames each with public and private keys. This allowed us to load this well defined tree and perform our tests on it, changing simple parameters (connection type, number of physical machines to use, what application to run on top of the signers, ...) to change whether it was loaded in using TCPHosts or GoHosts for local testing.

7 Testing

The collective-signing was tested on three different architectures. The Zoo over LAN, PlanetLab over WAN, and DeterLab over a simulated WAN.
7.1 Testing on the Zoo

The Zoo is a computer lab setup with a shared file system. To deploy and test our collective signing test on the Zoo we simply had to copy over the configuration files and the executables to a single node, then run the executable from each node. Each node would run its corresponding host in the configuration file, connecting to its parent, and accepting connections from its children. This was used initially for testing as it provided a reliable network that we could use, however, since it was not a Wide Area Network with high round trip times between nodes, it was not representative of the global scale which we hope this protocol will be deployed on.

7.2 Testing on Planet Lab

PlanetLab is a global network for researching. Currently there are 668 sites and a total of 1342 nodes. We initially tested on the large scale network.

We built a similar script for testing on PlanetLab as on the Zoo. The testing program built the executables for each of the architectures of the planet lab nodes. The test program then sent the executables to each of the planet lab nodes with a config file that specified a minimum spanning tree over the network.

7.2.1 Creating Minimum Spanning Trees from Ping Times

In order to deploy onto planet lab we needed to create minimum spanning trees for the collective-signing overlay network. To do this, I built a simple program which I deployed to each node that collected ping times to other planet lab nodes. Given connectivity graph, we simply run a version of Dijkstra's algorithm where we can also specify our branching factor for the overlay-tree. This Minimum Spanning Tree is then encoded into the config file with the given hostnames.

7.2.2 Problems with Planet Lab

Initially when testing at a small scale Planet Lab was reasonable to work with, however PlanetLab nodes are notoriously overloaded and unreliable and our protocol assumes a somewhat reliable network (not too much churn). Some of the problems that we encountered were that it was hard to reserve that many node, that nodes often times don’t allocate enough space to contain the executables as well as configuration file (it grows linearly with the number of nodes). As we added more nodes it soon became infeasable to filter through all the nodes and find the ones that were reliable enough, gave us enough space, allowed us to open enough file descriptors. Furthermore some nodes were extremely restrictive for security reasons prohibiting us from running our executables entirely.
7.3 Testing on Deter Lab

Ultimately we settled on testing on DeterLab. DeterLab is another shared facility for researchers. However DeterLab, instead of exposing a true WAN, simulates any network topology desired on its local machine clusters. This proved to be much more reliable for testing. As this was our final testing environment I will describe in more detail the precise layout.

DeterLab has a gateway server which any one with an account can ssh into. This gateway server then exposes the nodes that you have created as existing in a simulated LAN. These nodes can only be accessed from the gateway server.

For us we decided to simulate a star topology on the network, meaning every node was directly connected with every other. They were connected with 100 ms round trip latencies through a series of central hubs responsible for simulating these latencies.

In order to work with this architecture, I built a series of scripts called "deploy2deter", "deter", "forkexec", and "exec". deploy2deter builds the executables for the target environment, constructs a configuration tree from a list of hostnames and a number of hosts to have on each machine. The configuration tree then represents a secondary virtual topology on top of the virtual topology of deterlab, in which multiple collective signing processes can be running on one machine, but no two nodes on a single machine are adjacent to each other on the configuration tree (graph coloring). This way we could extend our allotted machines even further in order to simulate networks of much greater sizes. These executables and configuration files are then all sent to the gateway server, where it runs the "deter" executable.

deter is responsible for redistributing the code and executables onto each of the machines. It then starts up timeclients, which are timestamp clients who send timestamp requests to the timestamp servers at a given rate, then deter runs a forkexec process on each of the machines.

forkexec runs exec with the unix time command in order to record and report system and user time.

exec is responsible for running the timestamps themselves, which are applications built on top of the collective signing layer.

Given the slow rate of data transfer onto the gateway node many things had to be done to speed up the process. All the executables and files necessary for the tests were all first put locally into a folder which was then ’rsync’ed with the gateway server in order to minimize datatransfer between tests where only flags changed or only configuration files. Given that most of the data was in executables, only the most minimal encryption was used (rsync requires always that you use something), and our tests were run in orders in order to decrease the amount of data transferred. For a completely clean run, it could easily take 3-4 minutes just to start up the timestamper processes on all the machines due to the poor performance of the gateway and the large amount of data (when multiplexed to each virtual machine).
7.3.1 Problems with DeterLab

DeterLab was very slow to start up (send all of the data over) so it was hard to rapidly test on large configurations. DeterLab places very low limits on the number of open file descriptors allowed. This meant that we rapidly got to the stage where we had too many TCP Connections and could no longer test on a depth 2 tree in a large configuration.

The testing infrastructure itself is somewhat unreliable there were many days where the infrastructure was unavailable due to outages, but it was still much better than PlanetLab.

Other times DeterLab would have mysterious problems where the virtual topology would not be properly established. Sometimes the connections would not be established between two nodes which were supposed to be connected, and sometimes latencies would mysteriously change.

8 Logging Architecture

In order to facilitate the aggregation and visualization of statistics and logging information, I created a server that aggregates the logs from the timestampers.

8.1 Logging Messages

In order to facilitate the remote logging of messages we use a package called logrus. Every message that is logged is, in addition to being put to stdout, formatted into JSON with keys indicating time sent, the level (info, warn, error), who sent it, etc. the JSON message is then sent via websocket to the logserver.

8.2 Logging Server

The logserver listens to incoming log messages from timestampers. It runs a websocket server which clients can connect to and send their log messages. These log messages are then aggregated into growing array of log messages.

Clients can also connect to the log server’s log endpoint in order to request the stream of logs. Once a client connects, they receive the entire history of log entries in their original JSON format. For each client I keep track of their index into this ever growing log as they advance through it. If they ever reach the end of the stream, the simply wait for a new message to come in and then they continue to stream.

In order to handle the large amount of nodes running in our virtual network, without running out of file descriptors, the log server architecture can be deployed such that there are three log servers, with a single master node, to which the subsidiary log servers forward their logs. This allows us to linearly decrease the number of open TCP connections in our log servers by the number of log servers that we run in this tree.
8.3 Logging Frontend

The logging frontend is served by the logging server. It gives us a runtime view of the logging process as well as current runtime statistic. It parses the log entries as needed, writing errors as red, warnings as orange, and info as black. It then uses flags like "root_round" to specific data like how long it took to run a round. It then gives us some summary statistics like Min, Max, Average and Standard Deviation. Due to possibility of numerical errors and to give us a fast stream based standard deviation, Welford’s method is used. It also outputs the timing information to the a runtime chart which labels the x-axis with the collective signing round number and the y-axis with round latency in number of seconds. This graph able to give us good summary data and locate the spikes in our performance while we are running it.

The logging server extends another endpoint which enables us to look at each of the specific timestampers for debugging. Each timestamper runs an http service that exports its cpu profiles, memory profiles, and key statistics like garbage collection times, stack traces, etc. The log server then acts as a reverse proxy forwarding these endpoints at special debugging endpoints of their own. This was crucial early on to find small bugs in logic that developed when we were rapidly iterating.

8.4 Testing Log Receiver

In order to aggregate testing results across a large number of automated tests, I simply built another consumer to the logging server endpoint which consumed the logs until the time stampers indicated that they were closing. Like the
logging frontend, it also aggregated min, max, average, and standard deviations for all of the runs, storing it in csv files for later processing.

9 Viewchanges And Voting

I also collaboratively implemented view changes and Voting. View changes are necessary for removing malicious nodes, ensuring that malicious nodes who are quietly disrupting progress do not stay as root, etc. The view change mechanism currently is quite simple: there is a deterministic algorithm for deciding the tree structure given a list of hostnames and a round number. The view change is then initiated by the root of the next tree and other nodes either accept or reject the view change. If the view change has received enough accepts (2f+1), then the new view is accepted, otherwise this view change is ignored.

Voting is a more general process, similar to that of view changes. The root announces a vote (like adding or removing a peer), and sends this down the tree, everyone aggregates their vote and the vote of their children, signing their vote with their public key for verification purposes. Then the root aggregates these votes and sends out whether the vote was passed or rejected along with the proofs. These proofs can then be verified by any node in the tree, so if a node is malicious and attempts to change votes, it will be detected.

10 Conclusion

For more details of the overall algorithm and our achieved results please look at the paper: http://arxiv.org/pdf/1503.08768v1.pdf, for a more detailed look at the implementation look at github.com/dedis/prifi/coco. The next steps for the project will be refactoring the code with all the lessons that we have learned along the way in mind and implementing a certificate-authority on top of it.

11 References