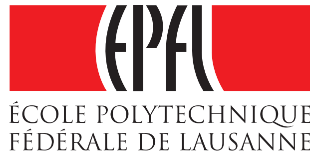

Implementation of a robust and scalable consensus protocol for blockchain

Raphaël Dunant

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

January 2018

Responsible	Supervisor	PhD Supervisor
Prof. Bryan Ford	Linus Gasser	Lefteris Kokoris Kogias
EPFL / DEDIS	EPFL / DEDIS	EPFL / DEDIS

Contents

1 Introduction **3**
1.1 CoSi protocol . . . 3
1.2 ONet . . . 4

2 Architecture **4**
2.1 Tree generation . . . 5
2.2 CoSi . . . 5
2.3 Failing nodes . . . 5

3 Implementation **6**
3.1 Cothority Template . . . 7
3.2 Tree generation . . . 7
3.3 Multiple sub-protocols . . . 7
3.4 CoSi with multiple sub-protocol . . . 7
3.5 Channels vs handlers . . . 8
3.6 Failing nodes . . . 9
3.6.1 Failing subleaders . . . 9
3.6.2 Failing leafs . . . 9
3.6.3 Consequences of failing behaviour . . . 9
3.7 Simulation . . . 10
3.8 Unit tests and documentation . . . 10

4 Results analysis **11**

5 Future work **13**
5.1 BFT-CoSi . . . 13
5.2 Real Blockchain . . . 14

6 Installation **14**

7 Conclusion **14**
7.1 Personal Experience . . . 15

8 Bibliography **15**

1 Introduction

The purpose of this project is to implement a decentralised witness cosigning protocol as described in the paper "Keeping Authorities "Honest or Bust" with Decentralised Witness Cosigning" [7]. This project aims to have a complete, functional, failure resilient, documented and tested code base to allow witness cosigning using the CoSi protocol, explained in the next section. This project uses knowledge from previous tests to create a scalable network of witness using a three-level tree. At the time of the project start, a CoSi code already existed, but was not documented, nor practical. It was assembled only for testing purpose and therefore, we decided to throw it away to start anew. The main reasons for this projects were:

1. The need of a reusable, integrated with existing frameworks, code base.
2. The need to test extensively the CoSi protocol, its behaviour and resilience.

The following subsections present the CoSi algorithm, the algorithm upon which the project is based as well as ONet and Kyber, the two main libraries used in this project.

1.1 CoSi protocol

CoSi is a signature protocol allowing a computer to sign a given proposal using other independent witnesses. The proposal can be a timestamp, a contract, a software update or any document that need to have a signature. This protocol therefore makes proposal signing not dependent on a single entity, the computer signing the proposal, but instead certify that a certain number of witnesses have validated the proposal. The protocol is scalable by using two main optimisation. First a communication tree between the witnesses, which is described in the section "Tree generation". And a Schnorr Signature [6], which is a method to aggregate signatures to have a final signature size independent of the number of witnesses. The simplified protocol takes as input a proposal to be signed and outputs an easily verifiable aggregate signature. To verify this signature aggregate, one only needs to know every witnesses' public keys.
More specifically, the protocol uses four messages in two round-trip, starting at the root.

1. **Announcement:** The leader announce the start of the protocol and sends the proposal down the tree.
2. **Commitment:** Each node accepting the proposal generates a Schnorr commitment and its corresponding secret, based on a random number. It then aggregate its commitment with their children's commitment (if any) and sends the aggregate to its parent.

3. **Challenge:** Once the root received the commitments, it generates a Schnorr challenge and send it down the tree.
4. **Response:** In this last phase, every committed node generates a response, based on the collective challenge, its commitment, its secret number and its private key. It then aggregates this response with its children responses (if any) and sends it up the tree, as the commitment in phase 2.

Lastly, the aggregated response and the aggregated commitment are used to sign the proposal. Note that the behaviour if a node is failing is discussed in the next section.
This protocol therefore returns a global aggregated signature, insuring that a certain number of nodes have validated a proposal.
The cryptographic part of the CoSi protocol is fully implemented in the Kyber library [4]. The Kyber library is a library created by DEDIS to be a toolbox for basic cryptographic operations. It contains all the cryptographic part of the CoSi algorithm (Schnorr signature, mask handling, etc.) and therefore allowed this project to focus on the actual protocol instead of technical cryptography. Kyber can be found on Github. [4]

1.2 ONet

ONet [5] is the main library used in the project. It has been developed by DEDIS, the Decentralized and Distributed Systems laboratory of EPFL. This project uses the version 1 of the library, because it was stable and compatible with the CoSi implementation at the start of the project. The official definition of ONet is:
The Overlay-network (ONet) is a library for simulation and deployment of decentralised, distributed protocols. It offers an abstraction for tree-based communications among thousands of nodes and is used both in research for testing out new protocols and running simulations, as well as in production to deploy those protocols as a service in a distributed manner.
As stated in this definition, ONet handles the servers, the nodes and the communication, allowing this project to focus only on the actual protocol. The choice to use this library was logical, since we wanted to develop a product compatible with other DEDIS code base, allowing a simple integration in existing DEDIS code base.

2 Architecture

This section describes in details the Project architecture, how each part of the Protocol is working, as well as why such design choices have been made. It follows the development order of the project, that is, tree generation, CoSi protocol and failing nodes.

2.1 Tree generation

The CoSi protocol uses a communication tree to have better performance and scalability. Each computer in the protocol is a node on the tree. The tree represents the path the messages take in the protocol. For optimal results, the tree ideally should be a balanced tree, with all branches of the tree having the same height. Having a balanced tree evens out the time a message takes to do a round-trip from the root to the leafs.
For this project, it has been decided to use a tree of height two. This is a compromise between vulnerability to denial of service and long round-trip time. In a tree of height one, it is possible that the root experiences too many connexions at the same time, since every single node is connected to it. Having a tree with many levels slows down the protocol, because the network delay is often the bottleneck and the time to go from the root to the leafs is the limiting factor. Therefore, a tree of height two has been decided as a basis for this project.
This project uses the following naming convention. The two-level tree is defined as a *root* with a certain number of children. Those children, the nodes at level one, are called *subleaders* and their respective branches, composed of the root, them and their children, are called *subtrees*. Therefore, the tree consists in the composition of a number of subtrees with the same root. This means that the tree creation is defined by two parameters:

1. The list of nodes (or servers) of the tree
2. The number of nodes per subtree

2.2 CoSi

The plan to use the CoSi protocol was to have a defined behaviour for each node upon reception of a given message. Originally, if a node receives a message of type announcement, it validates the proposal and forwards it to its children, if any. If the node receives all its children commitment, it generates his commitment and aggregates it with others, then send it to its parent. And so on for every four type of message presented previously.
This behaviour was the original planned one, but, halfway through the project, we added the constraint that those message have to be ordered, insuring that the steps are in order. This decision is discussed in the section "channels vs handlers".

2.3 Failing nodes

For this project, we decided to handle finely only nodes failing at the start of the protocol, causing it to time out at the commitment phase. That is a node never answering to any message. If the protocol fails in any other way, for example failing at challenge time, going into an incorrect state, receiving

forged incorrect messages or others, we let the protocol timeout and restart it. We do not try to correct the protocol at runtime in those situations as this would complicate the project and bring new challenges. However this can be done in the future (see section "future works").
During the commitment phase, a participation mask is created. This mask contains a flag for every node in the protocol indicating whether the node participates in the protocol or not. This participation mask is then used at challenge generation and final signature. A node not participating in the commitment phase is ignored in the following phases. The mask also allows to see easily the number of participating nodes and for verifiers to decide whether this number is enough for them or not.

It is important to note that the tree is generated by the root at the start of the protocol. The root arranges the tree and decides which node fulfills which role, subleader or leaf.
We assume for this project that the root is behaving correctly. In the Byzcoin paper [2], the nodes monitor the leader and can vote to dismiss it if they found it to be faulty or unresponsive. In that case, the next leader in a well-known schedule takes over as the new root. In this project, this behaviour is not handled for simplicity purpose, but can be added if needed (see future works).
The expected behaviour if a non-root node failing is to ignore it in the protocol. This is easy for leafs, since they can be excluded from the protocol just by ignoring them. However, if the failing node is a subleader (node of the first-level), the tree needs to be rearranged to avoid losing the validation of all leafs in this subtree. This is done by the leader (root), as it is the root's trusted authority to create the tree. If a subleader times out, the root chooses another subleader amongst the leafs of this subtree. Note that all other subtrees can stay the same, only the subtrees with faulty subleaders need to be rearranged. Then the root can safely restart the protocol. The failure behaviour can be optimised by sending a new announcement only to the failing subtrees, allowing the other subtrees to continue the protocol as if nothing happened.

3 Implementation

Now that the theoretical basis has been discussed, this section discusses the actual implementation. We discuss the tradeoffs that had to be made, the design decisions that shaped the implementation and the problems encountered during the development. This section follows the logical order of the implementation, starting from the template adaptation, continuing by the tree generation, protocol implementation, Failing nodes handling, Simulation.

3.1 Cothority Template

The practical implementation started with the "cothority template" model [1]. This template is a model of the implementation of a protocol on a tree and has been a real help to dive into libraries code and best practises. However, the project quickly deviated from this template. Indeed, the project has to generate the tree explicitly instead of using the library functions, as it needed a finer precision and the library only allowed n-ary trees. Another example is the fact that the template did not allow easy use of timeouts. In summary, using the template was a great help as a start, but soon showed its limits for this project.

3.2 Tree generation

The first implementation of the tree generation function was a simple two-layer tree, as in the original plan. This proved a problem because, when a subleader is failing, the tree needs to be rearranged. ONet does not allow rearranging of trees during a protocol run and therefore the whole protocol needs to be restarted if there is one failing subleader. Therefore, it has been decided that the tree generation function instead returns multiple trees, each having the same root, one subleader and all of this subleader's children. This allows an easy restart of the protocol on a failing subleader, but involves a synchronisation overhead for the root. We decided that the gains in robustness outweighs the losses in simplicity.

3.3 Multiple sub-protocols

The flow of information had to be modified to handle multiple subtrees. The protocol was therefore separated in a super-protocol for the root and sub-protocols for every tree. The super-protocol handle tree generation (and tree rearranging if a subleader fails) and all common generation (challenge, final signature). His most important task is to handle the correct behaviour of every subprotocols, aggregating their answers and giving them messages to send to everyone. Subprotocols define the behaviour of every non-root node. In subprotocols, each node follows the CoSi protocol.
All the protocols are created, started, handled and stopped by the ONet library. The communication between those protocols is done by channels at the root level.

3.4 CoSi with multiple sub-protocol

We describe here in more details in the path the CoSi protocol takes in our multi-protocol environment. For simplicity purpose, the overall picture has been simplified, omitting some information.

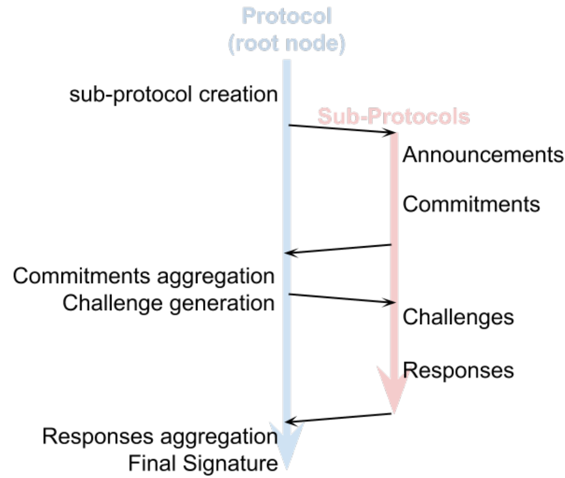

Figure 1: CoSi flow of information

As we can see in figure 1, the simplified main protocol starts by creating all subprotocols and sending the proposal to their root. The root of every started subprotocols creates and send an announcement containing the proposal to the subleader who in turn sends it to the leafs. Every node generate its commitment (if it accepts the proposal) and send it back to its parent. Once the subprotocol root receives the subleader commitment, it sends it to its super-protocol instance.
In the super-protocol, the root receives every subprotocol commitments. Note that it restarts unresponsive subprotocols after having changed their subleader. Once every commitment is received, the root generates its own commitment, aggregate the commitments and generate the challenge. Then it sends the challenge to every subprotocol root.
As in the first phase, the challenges are propagated in a similar way as the announcement. The responses are created, propagated and send back to the protocol in a similar manner as the commitments. Once a node has sent its final response aggregate, it terminates.
Finally, the super-protocol root generates its own response and aggregate all the responses. Using the aggregated commitment, the aggregated responses and the participation mask, the root signs the proposal. The final signature is returned as the result of the protocol.

3.5 Channels vs handlers

The basic template discussed previously used handlers for every message reception. This means that a node have a defined behaviour upon reception of a certain type of message. This behaviour is obviously different if the node is a root, subleader or leaf. For example, a subleader receiving an announcement forwards it to all of its children and a root receiving a commitment

in the sub-protocol sends it to the super-protocol. However, using handlers have two major drawbacks:

1. The handlers used by ONet do not support timeouts. As timeouts are one of the main requirement of this project, we cannot use handlers to deal with failing nodes.

2. Handlers are unordered. For example a node can receive a challenge before receiving an announcement, meaning it did not generate a secret and has not been taken into account in the mask. Another example is a node receiving multiple announcements and then a challenge. This would mean arbitrarily choosing a secret. It is a lot simpler to have order operations, as allowed by channels.

Therefore, we made the choice to use channels. All the behaviour of the node is contained in one *dispatch* method instead of different handlers functions.

3.6 Failing nodes

Once the whole protocol was implemented, tested and working, the next step was to implement failing nodes. As stated in the "Architecture" part, there is two different possibilities of failure: subleaders and leafs.

3.6.1 Failing subleaders

In any given subprotocol, the root is waiting on the subleader for the aggregated commitment. If the subleader times out, the root immediately sends to the super-protocol a signal indicating the timeout of the subleader. The root, once every subprotocol has been started, iterates through every protocol, get the valid aggregate of commitment and rearrange the failing subtrees. Once a failing subtree is rearranged, it restarts the subprotocol. Every time a failing subtree is detected, the subleader of this subtree is changed to another node of the tree. If all nodes have been tried as subleader and none works, the whole tree is then ignored in the participation mask.

3.6.2 Failing leafs

If a leaf times out, it is simply ignored in the participation mask and in the protocol.

3.6.3 Consequences of failing behaviour

The implementation of timeouts imply that the subprotocol could be stopped while some nodes were still running. For example, a running leaf with a failing subleader should be stopped to be restarted again. Therefore a stopping

mechanism was implemented. When a subprotocol is stopped, a stopping signal is broadcasted to this subtree. Nodes waiting on any channel see the channel closing and automatically stop.
The timeout is very critical and has been problematic in a big number of tests. There exists three different timeouts, in decreasing order of length:

1. The global protocol timeout, defining how much the root is willing to wait for the algorithm to complete.

2. The subleader timeout

3. The leaf timeout

Note that it is important to have those last two separate as the subleader should be able to send a message to the root before its own timeout has elapsed. Knowing the timeout allows nodes to adapt the time they use to verify the proposal. Therefore, those timeouts are defined at the start of the protocol and sent in the announcement message to every node.

3.7 Simulation

A part of the project is dedicated to running a real simulation, outside of local tests. A simulation has some differences with a local test, for example, running the code on different processes forbids the use of global variables or other shared values. But this proved to not be a problem as it was coded with this constraint from the start.
A simulation run is defined by four varying parameters: The number of nodes, the number of subleaders, the number of failing subleaders and the number of failing leafs.
The main problem encountered while running the simulation was to setup the timeouts to the correct values. The results of the simulation are explored and commented in the next section.

3.8 Unit tests and documentation

One of the main purpose of the project is to have a reliable and reusable code that could easily be used by future projects. The project coding was therefore done with a strong emphasis on having a documented and tested code.
We should note that a code is never bug free and that new unit tests can always be added. The unit tests are never the same as a real simulation, with races, real compartmentalisation and network latencies. But the code is more resilient and have less bugs thanks to this policy.

4 Results analysis

The next step of the project was to test the protocol on different configurations. The simulation tests were run on the EPFL IC cluster on four different machines. Each test result is the average of 10 full runs, with a network delay of 50ms for each internode communication. We began by varying the number of nodes from 10 to 1000. Due to testbed limitation, we were not able to have more than 200-300 nodes per machine, due to limitations in number of address ports by the library and channel size. With 4 machines at our disposal, we capped the tests at 1000 nodes. See the CoSi paper for results with more than 1000 nodes [7]. The branching factor is defined as the square root of the number of nodes, allowing a load distribution as regular as possible. Here is the result, in simulation time:

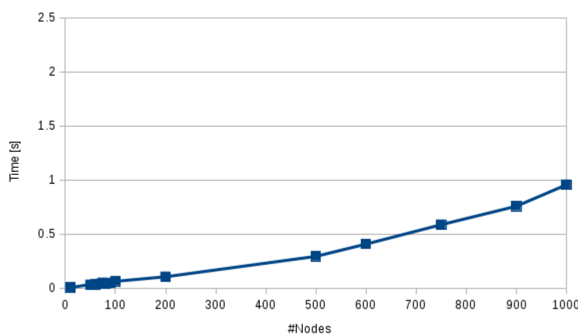

Figure 2: Protocol time in function of number of nodes

Those results are consistent with those of the CoSi paper [7]. The protocol latency increases with the number of hosts. It scales gracefully, always staying under 1 second even with 1000 nodes.
Then we tried to vary the number of failing nodes. First we vary the number of failing subleader, with 500 nodes and $\sqrt{500} \approx 22$ subleaders.

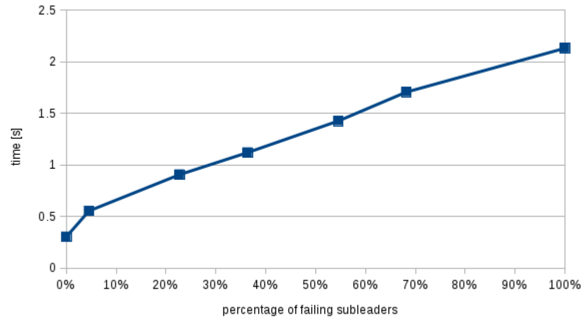

Figure 3: Protocol time in function of percentage of failing subleaders with 500 nodes

We observe that the protocol depends linearly on the number of failing subleaders. This is the expected behaviour as each failing subleader causes the root to rearrange the subtree, restart and reconfigure all subprotocol nodes (handled by the ONet library) and restart the subprotocol on that subtree. This complete recreation of subprotocol nodes takes time and slows the whole process, as the protocol is waiting on the full completion of the commitment phase to generate the challenge.
Finally, we tested the variation of failing leafs. This test still uses 500 nodes and $\sqrt{500} \approx 22$ subleaders. Note that we stopped the tests at 300 failing leafs. Observing the dependency between number of failing leafs and time, it is not interesting to continue the tests with more than this. Note that if all leafs are failing, the time is greatly reduced, because the second phase (challenge/response) only uses the first layer of the tree, gaining some time on the protocol. This behaviour has not been tested on the simulation and could be interesting to try.

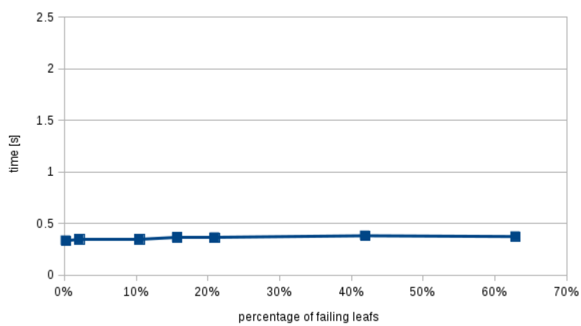

Figure 4: Protocol time in function of percentage of failing leafs with 500 nodes

As we can see, the variation of the number of failing leafs does not affect the overall protocol run time. This is explained by the fact that if a leaf is

failing, it does not causes any event that takes time. The node is simply ignored in the participation mask and in the next phases of the protocol. All failing leafs timeout approximately at the same time, as they are contacted approximately at the same time in the communication tree.

5 Future work

There is still a lot that could be added to the project.

1. Implement a full Byzantine fault tolerance-CoSi (BFT-CoSi) [2] as described in next section.
2. Handle root node failure. That is to have witnesses voting on deposing the leader if it behaves incorrectly. With it, a failure of every type of node would be handled.
3. Handle nodes failure at later times than announcement. A node (or the connexion connecting two nodes) can also fail anytime during the protocol. In that case, we could handle more finely the timeout, or even adapt the aggregates during run to exclude those nodes.
4. Expend the unit tests and unit tests coverage. This insures a more stable and resilient code and can always be done.
5. Implement the code on a real bockchain.
6. Make the project code support the new version of ONet. ONet has, during the project, shifted from version 1 to version 2 and a lot of its public interface has changed. Making the code use the last version of the library would be a clear advantage for future-proofing the project.
7. Use Omniledger's Sharding Via Bias-Resistant Distributed Randomness as described in the Omniledger paper [3]. This would insure that the leader cannot create the subtrees to its own interest, but instead is entitled to follow a random process, effectively decentralising the system even more.

5.1 BFT-CoSi

The next logical step is to implement a full Byzantine fault tolerance-CoSi (BFT-CoSi) [2] instead of a simple CoSi. The BFT-CoSi protocol uses two rounds of CoSi, the first one for nodes to prepare and affirm that they have settled on the proposal, the second to really commit and validate the proposal. This BFT-CoSi is a more robust way to sign a proposal that should be unique, for example a block in a blockchain.

5.2 Real Blockchain

The code could be implemented easily on a real blockchain, as Byzcoin is doing. Since the code has been built from the start to that purpose, it should be fairly easy to do.

6 Installation

The repository and the code are publicly available at *github.com/dedis/-student_17_bftcosi*. To get the code, the simplest way is to use *go get* on a UNIX machine. The library onet.v1 is also necessary for the project to work, you can get it with *go get* as well.
The project is separated in different folders.

- The main code is in the folder *protocol*. It contains the full implementation of the protocol with helper functions. The complete explanation of the different files of this folder is present in the file *doc.go*.

- The folder *protocol_test* contains unit tests testing different behaviour of the protocol. Those tests can also serve as examples of how to use the code base.

- Finally, the folder *simulation* allows to easily run a simulation. The parameters of the simulation are in the file *protocol.toml*. Here can easily be defined the number of hosts, subtrees, failing subleaders and failing leafs as well as a few others parameters.

To start a simulation, simply run the simulation executable with the *protocol.toml* as input (on UNIX machines: *./simulation protocol.toml*). This returns a file *test_data/protocol.csv* containing all the interesting information about the run: the time it took to complete, the bandwidth used, the time it took in the protocol, the time it took in the system, and so on.

7 Conclusion

The purpose of this project was to create a robust, scalable and reusable witness Cosigning protocol. I feel like this project has been completed with success. Indeed, a lot can be added, but the main goals of this projects have been fulfilled.
The tests are showing that the protocol is resilient to failing nodes, scales gracefully and returns valid signature under the threat model. This means that we now have a solid code to use for anyone interested in witness cosigning. That code will hopefully be included in many projects in the future.

7.1 Personal Experience

Even though I feel like there is a lot more that could be done, I am very proud of what has been accomplished. I learned a lot about project management and decentralised systems. I have now far more confidence in my project organisation skills and that I can bring projects to a satisfying end. I know for sure that this will not be the last project I have in this field. In conclusion, I am overall satisfied about this project.

8 Bibliography

References

[1] *Cothority Template: Implement new cothority protocols, services and (client) applications.* `https://github.com/dedis/cothority_template`. Accessed: 2017-11-01.

[2] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. *Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing.* Technical Report ISBN 978-1-931971-32-4, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, August 2016.

[3] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. *OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding.* Technical Report 2017:406, École Polytechnique Fédérale de Lausanne and Trinity College, Lausanne, Switzerland and USA, October 2017.

[4] *Kyber: Advanced crypto library for the Go language.* `https://github.com/dedis/kyber`. Accessed: 2017-11-01.

[5] *ONet: Cothority network library.* `https://github.com/dedis/onet`. Accessed: 2017-11-01.

[6] C. P. Schnorr. *Efficient Identification and Signatures for Smart Cards*, pages 239–252. Springer New York, New York, NY, 1990.

[7] E. Syta, I. Tamas, D. Visher, D. I-Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. *Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning.* Technical Report arXiv:1503.08768, Yale University and École Polytechnique Fédérale de Lausanne, New Haven, CT, USA and Lausanne, Switzerland, May 2016.