ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Firenet: a transparent and secure decentralized network management scheme

Jingyue ZHAO

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

February 2018

Supervisor
Prof. Katerina Argyraki
EPFL / NAL

Supervisor
Prof. Bryan Ford
EPFL / DEDIS

I. Background

The long-term goal of Firenet is to explore a transparent and secure decentralized network management scheme for large-scale networks managed by groups of administrators (maybe with differentiated roles and responsibilities in an hierarchic). Both the endpoint hosts and the administrators of the network can be geographically distributed. The administrators manage certain subnets separately or/and the entire network together. The network policies made by the administrators are applied to different ranges of the network (e.g., some subnets or the whole network). The decisions of each administrator may affect the entire network because these network policies may have direct or indirect impacts on other subnets. The administrators need a way to check and supervise the policy making process (maybe hierarchically) to avoid that a compromised administrator (may be bribed, threatened, or coerced) to make malicious policies which will have catastrophic effect on the network.

In this semester project, we assume that a single small group of administrators manage the network together. They are equal to each other and each policy they make need to be checked and approved by a threshold of administrators.

II. System and Threat Model

In a large network, the *network administrators* (i.e., admins) make network policies together to manage and secure the network. The policies are followed and adopted by the *follower routers* which can be seen as the SDN controllers. For now, we assume the network is managed by a single small group of admins who are equal to each other: anyone of them can propose a new policy, and the policy is valid only if a threshold of the admins approve it. In the long term, we may have more complex hierarchical network management model. For example, there are multiple groups of admins with more differentiated roles and responsibilities so that the network policies will also be hierarchical and followed by different ranges or parts of the network. A policy is a set of network rules (e.g., rejecting the packets from a certain IP address or blocking some incoming ports). We assume that up to a threshold number of admins may be compromised (e.g., malicious, bribed, or coerced by an adversary) to make or approve bad policies.

A number of *witness servers* inside the network form a *cothority* (collective authority) together. They are chosen by the admins and collectively validate and sign a newly made policy using CoSi [1]. For now, we assume that the witnesses are trusty. The policies are stored in a network-wide public hash chain on the witness servers.

Follower routers play the role of SDN controllers, which periodically request the witness servers in the cothority to get the latest policy block. They validate the policy by verifying only a single collective signature and then adopt the policy. We assume for now that the follower routers honestly adopt the policies. But in the future work, the follower routers may be malicious and we need to guarantee that if a router is compromised, it will not affect other part of the network (e.g., other follower routers and their endpoints). And for now, the follower routers do not need to provide any proof for their identities, which will be improved in future.

Endpoint hosts are connected to the follower routers with the expected network policies made together by the admins. We also assume for now that the endpoint hosts are honest. Maybe later we can assume that they may be malicious and we need to keep their malicious effect to themselves.

In summary, we may build more complex threat model in which the follower routers and endpoints can be malicious in the future. And the potential goal is that other normal routers and endpoints will not be affected by the compromised ones. In this semester project, however, we focus on the compromised admin threat model.

III. Design of Firenet

The design architecture of Firenet is shown in Figure 1.

Step 1: To protect against a single compromised admin, Firenet requires that admins have individual signing keys and that a threshold of the administrator sign each network policy. At the beginning, the admins collect all their public keys in a configuration file, together with a threshold value that specifies the minimal number of valid admin signatures required to make a policy proposal valid. To propose a new network policy, an admin who becomes the leader admin in this policy making process, creates the policy file and sends it to all the other admins. Each admin checks the new policy file and signs it respectively, if the admin believes the policy is valid. Otherwise, the admin will not sign the policy. The signatures of admins are added into an append-only list. Then the leader admin sends the policy, the configuration and the signatures to the cothority.

Step 2: Firenet uses a cothority to validate each new policy release of admins and collectively sign it. The cothority is a set of witness servers. These servers are inside the network, chosen by the admins. To validate a new policy release, each witness server of the cothority checks the admin signatures with the public keys and the threshold number defined in the configuration file. After the validation, the witness servers collectively sign the policy and the corresponding configuration using CoSi [1].

Step 3: To build an admins-transparent policy making process protecting against policy release history tampering, we adopt a network-wide public log chain for policy releases in the form of collectively signed decentralized chain. In this way, even if a powerful attacker successfully compromises a threshold of admins and makes bad policy valid via cothority, his malicious behaviors will be under public exposure to the admins. This adminsâĂŹ supervision hinders the attackers from making malicious policies since the real-world attackers prefer stealthy behaviors. To build a policy chain maintained by a number of witness servers to record each valid policy release, everytime the admins send a new policy to the cothority, they also need to provide its previous policy block hash. And this previous policy block should be the latest block in the chain without any other existing children block. Each time the cothority receives a new policy request from the admins (which includes a policy, the corresponding configuration and its previous policy block hash), the cothority will first validate and collective sign the policy and configuration (see step 2), and then check if this policy request has the correct hash of the latest block in the chain and that there is no other block with the same parent in the chain. After all the above validation and check, the cothority will create a new policy block with the new policy, its configuration and co-signature as the data part, and append this new block to the existing policy chain. Here, we use Skipchain [2] as the structure of the policy chain. Also, each time a new policy is accepted to the policy chain, the cothority will return the new policy block and the block hash to the admins, so that the admins are supposed to always have the latest block hash.

Step 4: The follower routers periodically (e.g., several minutes) request the cothority for the latest policy chain and adopt the newest policy that have not been deployed on themselves. We assume for now that each policy block is self-sufficient so that the follower router does not need to refer to the history policy blocks once it has obtained the latest one in the policy chain. After a follower router has downloaded a new network policy, it verifies the policy by checking the single collective signature of the cothority. If the collective signature is valid, then the the policy is valid and will be deployed by the follow router.

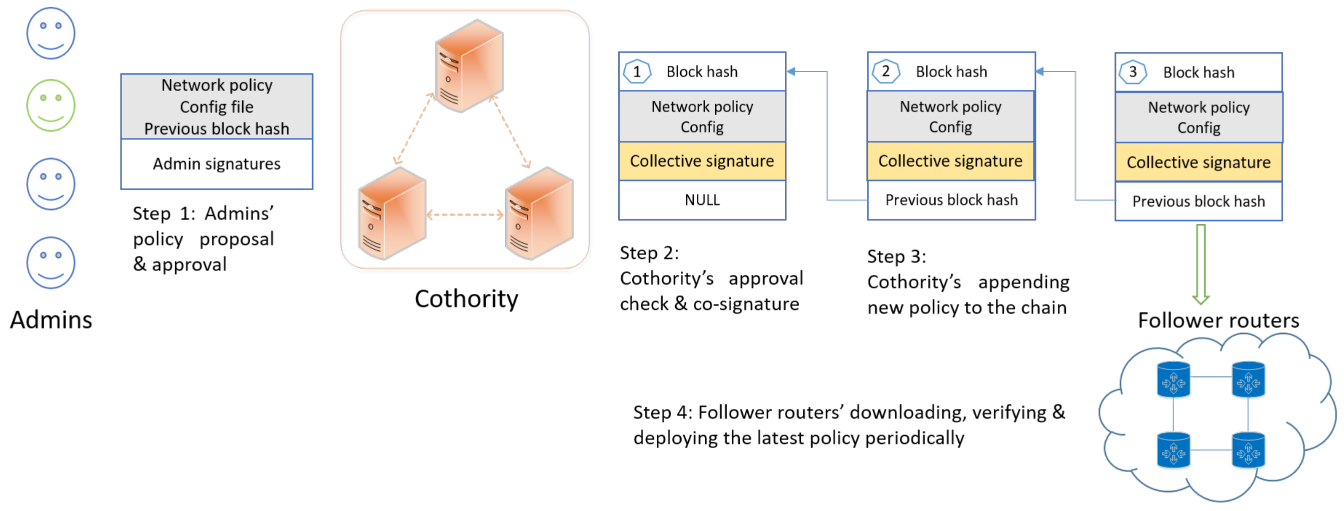

Figure 1: Architecture overview of Firenet

Another possible design is that we combine the admins and the cothority. Each admin controls a witness server, forming the collective administration (co-admin). Every Time an admin wants to release a policy, he (his witness server) becomes the leader of CoSi and tries to make all the admins reach the consensus to approve this policy. The other admins check the policy and decide whether join the collective signing process. Once the policy is successfully collectively signed, it will be added into the policy chain. The follower router pull the policy chain periodically to get the latest policy. This scheme is not ideal because if we have multiple groups of admins or a hierarchical management model, then it is not necessary to form a cothority in each group. Different groups of admins can request service from the single cothority inside the network to form one hierarchical policy chain.

IV. Network Policy Description Language

Here, we start from a simple example to describe the network policy of blocking certain ports. We define the network policy description language based on iptables [3]. A network policy consists of multiple network rules.

We use JSON objects to describe network rules in the default table, filter, containing three chains: INPUT, OUTPUT and FORWARD (defined in iptables). As is shown in Table I, a network rule is a JSON object specified by matches (i.e., conditions the packet must satisfy so that the policy can be applied), and one target (i.e., action taken when the packet matches all conditions).

Property		**Value**
Matches	Chain	INPUT, OUTPUT, FORWARD
	Protocol	TCP, UDP, ICMP, ALL
	Source IP/network	x.x.x.x, x.x.x.x/x, ALL
	Source ports	Port number(s)
	Destination IP/network	x.x.x.x, x.x.x.x/x, ALL
	Destination ports	Port number(s)
Action		ACCEPT, DROP, REJECT

Table I: Network rule format

The following is an example rule of blocking the input TCP ports 443 and 444.

```
{
"Match":{"Chain":"INPUT","Protocol":"TCP","Src":"ALL","Sports":"ALL","Dest":"ALL","Dports":"4
    43,444"},
"Action":"DROP"
}
```

At this stage, we assume that one network policy is self-sufficient (i.e., the follower router only needs the latest policy for the deployment without referring to the previous ones). A network policy JSON object consists of a short policy description, the number of rules it contains and a JSON object array of network rules (as is shown in Table II).

Property	**Value**
Policy description	String
Number of network rules	Int
An array of network rules	Network rule objects

Table II: Network policy format

We follow the âĂIJfirst matchâĂİ rule: the scanning of the network rules in one policy is in order from the first to the last one, and will stop as soon as a match is found. We put the default rules at the end of the policy array. If no match is found for a packet, then it will follow the default rule. The default rules for INPUT, OUTPUT, FORWARD are all ACCEPT. The following is an example of one network policy with 4 network rules to block the input TCP ports 443 and 444.

```
{
"Description":"block 2 input ports",
"Num":4,
"Rules":[
{"Match":{"Chain":"INPUT","Protocol":"TCP","Src":"ALL","Sports":"ALL","Dest":"ALL","Dports":"
    443,444"}, "Action":"DROP"},
{"Match":{"Chain":"INPUT","Protocol":"ALL","Src":"ALL","Sports":"ALL","Dest":"ALL","Dports":"
    ALL"}, "Action":"ACCEPT"},
{"Match":{"Chain":"OUTPUT","Protocol":"ALL","Src":"ALL","Sports":"ALL","Dest":"ALL","Dports":
    "ALL"}, "Action":"ACCEPT"},
{"Match":{"Chain":"FORWARD","Protocol":"ALL","Src":"ALL","Sports":"ALL","Dest":"ALL","Dports"
    :"ALL"}, "Action":"ACCEPT"}
]
}
```

V. Implementation

We implemented the first version of Firenet prototype in Go [4] based on the cothority [5] framework. We built on existing open-source code implementing CoSi and Skipchain. The new code implementing the Firenet prototype was about 1.3kLOC.

The overall implementation structure is shown in Figure 2. Each admin installs and runs a network management application (NM APP) on his server, which talks to the network management service (NM service) running on the cothority nodes (i.e., conodes). NM APP releases new policy and requests the cothority to validate the policy and append it to the policy chain. To respond to the request from NM APP, NM services on conodes validate new policy and collectively sign it using CoSi API, and calls Skipchain API to create and store a new policy block in the chain. For the follower fouters, each of them runs a follower router application (FR APP) which requests the NM service running on the conodes for the latest policy and validate the collective signature. To respond to the request from FR APP, NM service calls the Skipchain API to return the latest policy block to the FR APP, and calls CoSi API to verify the collective signature.

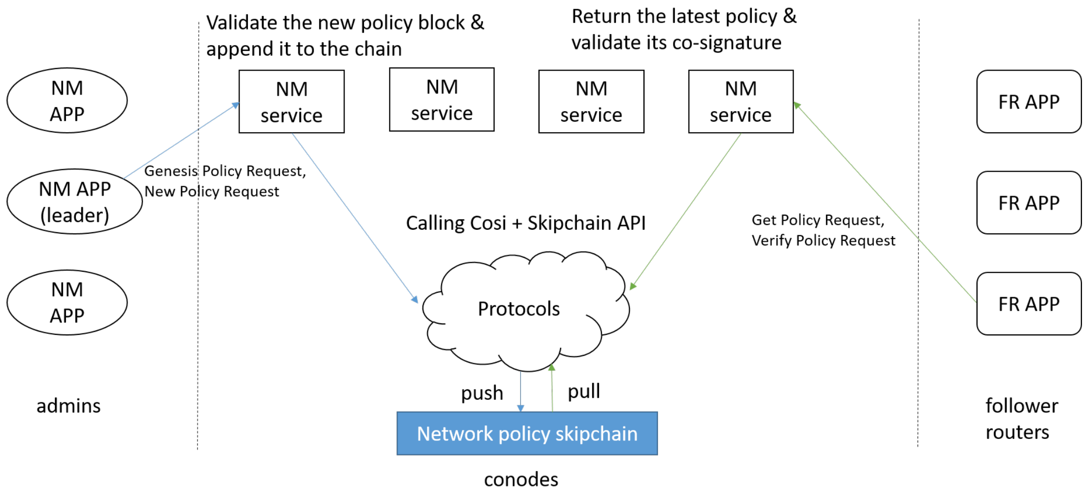
Validate the new policy block & append it to the chain
Return the latest policy & validate its co-signature
NM APP
NM APP (leader)
NM APP
admins
Genesis Policy Request, New Policy Request
NM service
NM service
NM service
NM service
Calling Cosi + Skipchain API
Protocols
Get Policy Request, Verify Policy Request
push
pull
Network policy skipchain
conodes
FR APP
FR APP
FR APP
follower routers

Figure 2: Implementation framework based on Cothority

A. Admin network management application

As is shown in Figure 2, the network management applications (NM APP) running on the adminsâĂŹ servers have two functions: Genesis Policy Request and New Policy Request. With this two functions, the admin send policy release request to the network management service running on the conodes to append a new policy block into the policy chain. Both of this two kinds of requests need to include the policy, the configuration, and signatures of admins on the policy. The difference is that Genesis Policy Request is used to create the genesis policy skipblock, so that it also needs to provide the arguments related to the skipchain structure.

And for the New Policy Request, instead of the skipchain parameters, it needs to include the previous block hash so that the cothority can append this policy to its previous one.

B. *Follower router application*

The follower router application (FR APP) talks to the network management service on the conodes with Get Policy Request and Verify Policy Request. It periodically sends Get Policy Request to the network management service running on the conodes to obtain the latest policy. And then sends Verify Policy Request to the service to validate the collective signature of the policy block. If the policy is valid, then the follower router application will deploy it.

C. *Network management service*

To respond to the Genesis Policy Request and New Policy Request from the admins, network management service (NM service) first check the signatures of the admins in the request. If the number of signatures reaches the threshold, then the NM service calls CoSi API to collectively sign the valid policy. In order to store the collective signed policy to the policy chain, NM service calls the skipchain API to form a new policy block and append it to the policy chain.

To respond to the Get Policy Request from the follow routers, NM service stores the latest policy block on each conode, and returns it to the routers efficiently when requested. As for the Verify Policy Request from the follow routers, NM service uses calls CoSi API for the policy validation.

VI. Evaluation

In the experiments, we used 32-core Intel Xeon CPU E5-2650 at 2.6 GHz with 66GB of RAM and, where applicable, ran up to 128 nodes on one server. And we evaluated the CPU time of the four requests, Genesis Policy Request, New Policy Request, Get Policy Request and Verify Policy Request, with the various number of admins and conodes. We used two example policy files and simulate the process in which the admins first create a genesis policy block and then add a new policy block to the policy chain, and then the follower routers download the latest policy and verify it.

As is shown in Figure 3, the CPU time spent on Genesis Policy Request and New Policy Request increases with more admins because the cothority needs to check more signatures in the first step. The reason that New Policy Request costs more time than Genesis Policy Request is that the cothority needs to check the previous block hash and then accept the new block to the existing policy chain. With 100 admins involved, about 0.18 second is spent on one New Policy Request. The time cost of Get Policy Request and Verify Policy Request stays stable since the two requests are not related to the number of admins.

From Figure 4, we can see how the time cost of Genesis Policy Request and New Policy Request increases with the scale-out of cothority. With a cothority consisting of 128 conodes, one New Policy Request costs around 20 seconds, which we believe mainly results from the time cost of adding a new block to the Skipchain. CPU time of Get Policy Request stays stable since each conode stores the latest policy block and can return it directly to the follower router. The stable cost of Verify Policy Request reflects the good performance of CoSi to verify the single collective signature.

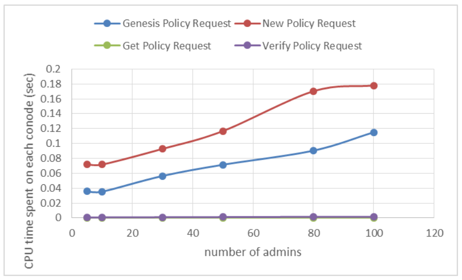

Figure 3: Time cost with different number of admins (4 conodes)

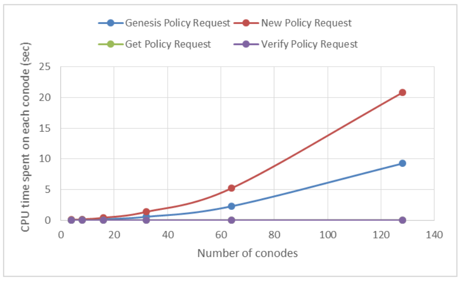

Figure 4: Time cost with different number of conodes (10 admins)

VII. Future work

We vision the future work in two general directions: performance analysis and protocol improvement.

On the performance part, firstly we can break down the time cost of adding a new network policy into the first 3 steps described in Section III and analyze the time cost of these processes with different number of administrators and cothority. We can perform regression based on the testing data generated from the simulation and analyze the performance curve tendency and limit with the scale-out of the system. Moreover, we can evaluate the bandwidth cost of different operations in the next step and analyze the performance tendency in a similar way to that of time cost.

On the protocol part, the first direction is to design the policy hierarchic. Since the the network management may be hierarchical in the future in large-scale networks, we should design the corresponding network policy description framework hierarchically (e.g., to build reference between the master policy and its sub policies). The second potential research work is to scale up the network management scheme to multiple groups of administrators in a hierarchic.

References

[1] D. V. D. I. W. P. J. L. G. N. G. I. K. B. F. Ewa Syta, Iulia Tamas, “Keeping authorities “ honest or bust” with decentralized witness cosigning,” in *Security and Privacy (SP), 2016 IEEE Symposium on. Ieee*, California, USA, May 2016, pp. 526–545.

[2] P. J. N. G. Kirill Nikitin, Eleftherios Kokoris-Kogias and L. Gasser, “Chainiac: Proactive software-update transparency via collectively signed skipchains and verified builds,” in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, Canada, 2017, pp. 1271–1287.

[3] Linux iptables. [Online]. Available: https://wiki.archlinux.org/index.php/Iptables

[4] The go programming language. [Online]. Available: https://golang.org

[5] Cothority. [Online]. Available: https://github.com/dedis/cothority/wiki