ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Decentralized Access Control

Sandra Siby

School of Computer and Communication Sciences
Decentralized and Distributed Systems lab
EDIC Semester Project

January 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Eleftherios Kokoris-Kogias
EPFL / DEDIS

I. Introduction

Access control is the management of access to a resource. This can be in the physical world (for example, access to a room) or in the virtual world (for example, access to a file). When we consider a simple access control model where a public key is given access to a resource, we are faced with various questions on how we can extend this model and create a system that allows for easy, flexible management of access control and user identity management.

In this project, we design, implement and test a decentralized access control system. Our system is based on the concept of *policies* – a set of rules. We delve into the structure of these policies and how they can be used to provide not only access control, but also identity management. We investigate how access requests can be created and verified. Finally, we evaluate our system by performing some benchmark tests.

The report is organized as follows: In Section II, we discuss the motivation and the key goals of the project. We describe the design and implementation details in Section III. Section IV outlines our benchmark tests to evaluate the system performance. We discuss some related work in Section V. Finally, we discuss possible improvements in Section VI.

II. Motivation and Goals

We consider a simple access control model, as shown in Figure 1. In this model, there is a resource which provides some kind of access (for example, read access) to a public key.

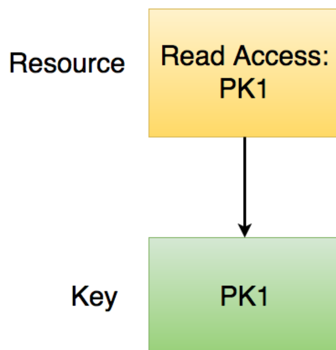

Figure 1: Simple access control model with static binding between resource and user key.

Several questions arise when we consider this model. For example, if the owner of the key decides to change her key, how does this impact her access to the resource? Also, if the owner of the resource want to provide access to multiple users, should she specify this access rule for each user separately or can she organize users into groups? We are also interested in investigating if we can expand this model to accommodate multiple rules or rules with conditions. Finally, we want to see how we can handle changes to access control rules.

The aim of this project is to answer the questions posed above. We want to design and implement a system that can achieve the following:

- Creation and management of access control rules.
- Management of user identities independent of the access control rules.
- Creation and management of groups of users for better organization.
- Evolution of user identities and access control with time.

III. Design and Implementation

Currently, there are various models to enforce access control [1]. Mandatory Access Control (MAC) is a model in which both resources and users have certain security labels. A global policy determines accesses based on the labels. MAC is typically used when security is of high priority (for example, in military applications). It enforces a strict checking mechanism. On the other hand, in the Discretionary Access Control (DAC) model, the owner of a resource determines who can access the resource. Role-based Access Control (RBAC) is based on the concept of roles and privileges. Users are assigned roles, which come with a set of access privileges. RBAC is a popular system and can be used to implement DAC and MAC.

Attribute-based Access Control (ABAC) [2] is a model that provides greater flexibility. In ABAC, users, resources and other variables are described using attributes. Access rights are controlled using *policies* which are a set of rules that can combine attributes. With ABAC, we can describe more complex relationships among the parties in the system using policies. In this project, we use an ABAC model and define our own policy structure.

There exist access control languages to express policies, such as XACML [3]. However, several of these languages tend to be verbose and more difficult to understand. Hence, we use a simple JSON based access control language to describe policies.

In the following sections, we describe our policy structure, access requests and request verification in detail.

A. Policy Structure

A policy consists of three components: ID, Version and Rules. ID is a random identifier generated at policy creation. A policy can be accessed by using its ID. Version indicates the version of the policy. Updates to the policy result in an increase in the version number. Rules refer to a list of access control rules.

A rule comprises three components: Action, Subjects and Expression. Action refers to the type of activity that can be performed on the resource. It is an application specific string indicating the activity. Subjects is the list of users that are allowed to perform an Action. We want to allow both individuals and groups of users to be Subjects and for users

to have control over their identity policies, independent of the access control policy. Hence, a Subject can either be a public key or an ID denoting another Policy object.

In certain cases, we may require the functionality to express conditions in access control rules. For example, access to a particular document might need approval from another party. To build more sophisticated rules, we introduce the concept of Expressions. An Expression is a string of the form: $\{operator : [operands]\}$. $operator$ can be a logical operator (such as AND/OR/NOT) and $[operands]$ can be a list of Subjects. An example would be: $\{AND : [ID_{Group1}, ID_{Bob}]\}$. In the context of signatures, this means that both ID_{Group1} and ID_{Bob}'s signatures are required by a particular rule. We can combine expressions to express more complex conditions for rules. For example, the expression: $\{`OR' : [\{`AND' : [S1, S2]\}, \{`AND' : [S3, S4]\}]\}$ evaluates to ((S1 AND S2) OR (S3 AND S4)).

Thus, a rule consists of an Action, Subject and Expression. In our model, we assume that a policy has a default Admin rule, which is created at policy creation time. The creator of the policy adds users to the Subjects parameter. The Subjects in the Admin rule indicate the list of users that are allowed to update the policy. A sample policy for a document, expressed in the JSON based language, is shown in Figure 2. The policy states that it has one Admin rule. The admins are S1 and S2 and any changes to the policy require both S1 and S2's signatures.

```
{
    "ID" :  2345
    "Version" : 1,
    "Rules" :
    [
        {
            "Action" : "Admin",
            "Subjects" : [S1, S2],
            "Expression" : "{'AND' : [S1, S2]}"
         }
    ]
}
```

Figure 2: Sample policy in JSON access control language.

Figure 3 shows an example of how we can use policies to achieve both identity management and access control. Report X has a Policy with three rules. One of the rules gives access to Group A and Bob. Group A's Policy has a rule that allows access for Amy and Jake. Amy's Policy contains a rule with her public keys PK1 and PK2. Note that an empty Expression field means that there are no special conditions for the Subjects. The policies can refer to each other using IDs. Not only this, groups of policies can be formed, as in the case of Group A. However, the three policies are maintained by three separate parties and evolve independently of one another. Changes to each policy are made by the admin group for that policy (as stated by the Admin rule in the policy). The arrows show how the Policy IDs can be used to link policies together. In this example, there is a link from Report X's Policy to Amy's key PK1, as indicated by the arrows.

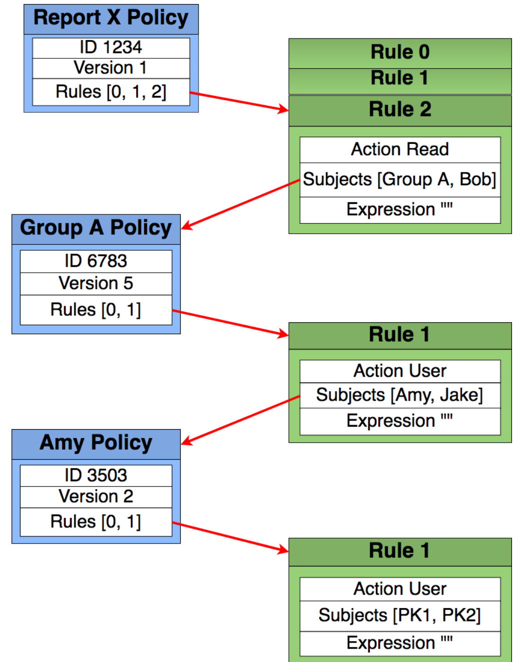

Figure 3: Policies and Rules.

B. Access Requests and Verification

We describe how requests to access a resource are created and verified. A request should have information about the access policy that allows the action the requester wants to perform. Hence, a request consists of the policy and the specific rule invoked. In addition to this, there might be extra information required to complete a request. For example, a set of documents might be governed by the same policy but the requester might require access to a specific document. Our request structure also consists of a message field where extra information can be provided. A request, Req, is of the form:

$$Req = [ID_{Policy}, Index_{Rule}, M],$$

where ID_{Policy} is the ID of the target policy outlining the access rules, $Index_{Rule}$ is the index of the rule specifying the access required by the requester and M is a message describing extra information relevant to the request.

To have accountability and verify that the requester is allowed to access the resource, we use signatures. The requester signs the request with his key and creates a signature consisting of the signed message and the public key used. A request signature Sig_{Req} is of the form:

$$Sig_{Req} = [R_{SK}, PK],$$

where R_{SK} is the Req signed with the requester's signing key, SK, and PK is the requester's corresponding public key.

An access request consists of the request and the signature, (Req, Sig_{Req}). Note that the public key corresponding to the key used to sign the request must be a key that is allowed to access the resource under the access control policy and must be present in Sig_{Req}.

On receiving an access request, the verifier checks that the R_{SK} is present and correct. The verifier then checks the *access proof*: that there is a valid path from the target policy, ID_{Policy}, to the requester's public key, PK. This may involve multiple levels of checks, if the requesters key is not present directly in the list of subjects but is included transitively in a policy that is a subject. The verifier searches along all paths till PK is found. This check also takes the policy versions into account – the path should consist of the latest versions of the policies for the access proof to be valid.

Multi-signature requests: Sometimes, an access request may require multiple parties to sign off. Conditions for multi-signature approval can be described using the Expression field in the rules. For example, a rule might have an Expression as follows: $\{AND : [Sig_{Group1}, Sig_{Bob}]\}$. This means that for a request to be valid, it should have Group1 and Bob's signatures. Hence, we need to introduce multi-signature access requests.

An access request in this case would be of the form $(Req, [Sig_{Req}])$ where $[Sig_{Req}]$ is a list of signatures from the parties that are required for the access.

The verification process is similar to the single signature case. The verifier checks all the signatures in the list and obtains a link from ID_{Policy} to PK. In addition to the signature check, the verifier also checks that the signatures adhere to the expression stated in the rule.

Figure 4 shows an example of the path verification performed by the verifier. Report X has a policy with a Rule granting read access to Bob and Amy. There is an expression stating that both Bob's an Amy's signatures are required to obtain access. Hence, if Bob wants to access, he sends a request $(Req, [Sig_{Req,Bob}, Sig_{Req,Amy}])$, where $Req = [1234, 2, \text{“}ReportX''']$, $Sig_{Req,Bob} = [R_{SK4}, PK4]$ and $Sig_{Req,Amy} = [R_{SK1}, PK1]$. The verifier checks the paths from the policy to Bob's $PK4$ and Amy's $PK1$. Paths are shown in red and blue respectively. Then the expression AND : [0,1] is checked against the signatures. If all checks pass, the request is considered to be verified.

Multipath Requesters: In cases where there are multiple paths from the target access policy to the requester, evaluation of expressions can be confusing. An example is shown in Figure 5. There are three paths from EPFL's policy to Sandra's policy. If there is a rule in the EPFL policy with an expression stating that a member of EDIC should sign a particular request, the path $EPFL- > EDIC- > Sandra$ has to be validated. If one of the other paths is chosen, there is no proof that Sandra is a part of the required EDIC.

Multipaths indicate that there should be some mechanism to choose the correct path. One option to resolve this issue is to have the verifier obtain all the paths and check them against the expression. The other option is to push path selection to the requester. We describe the latter option in more detail.

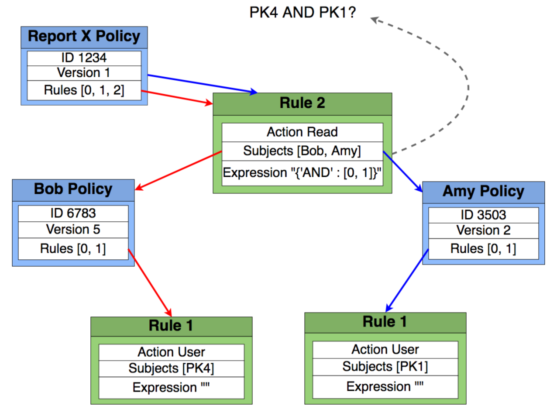

Figure 4: Verifier's path checking for multi-signature requests.

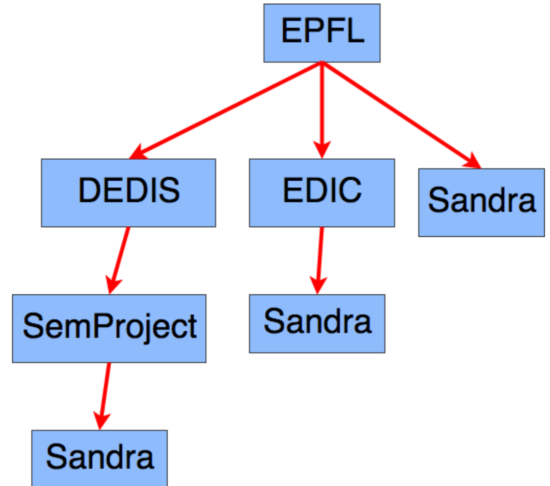

Figure 5: Multiple paths from target policy to requester.

If the requester performs path selection, the signing process can be changed to do a path search. When a request has to be signed, the requester searches for all the paths from the access policy to the requester. If there are multiple paths, the requester selects the specific path he/she wants to use and sends this information along with the signature. Thus, a request signature, Sig_{Req}, is now of the form:

$Sig_{Req} = [R_{SK}, PK, Path]$,

where $Path$ is the path from the target policy to the requester.

On the verification side, the verifier first checks the signature. Then, the verifier confirms that $Path$ is correct and that PK is present at the end of $Path$.

C. Evolving Policies

We use access requests to update policies. An update to a policy refers to a change in its rule set. An update to a policy

results in an increment in its version number. One method of ensuring that we have a verified record of all policy updates is to use the skipchain architecture [4] [5]. Each policy creates its own policy skipchain and updates to a policy results in the creation of a new block.

Figure 6 shows an example of how this skipchain based system would work. Report X has a policy skipchain. The latest version of the policy grants read access to Group A, of which Bob is a member. Report X, Group A and Bob have their own policy skipchains which they independently manage. Bob can access Report X since he is a member of the group that has access. If a new version of Report X's policy (v3) is created where Group A does not have access, its members will no longer be able to read the document.

In case of an access request from Bob's Key1 to Report X's policy, the path is shown in red. The rules in the latest policy skipblock indicate which skipchain to access. The skiplinks allow for fast traversal through the versions of the policy. All access proofs would check the latest versions of the policies, which would be the last skipblock in each policy skipchain.

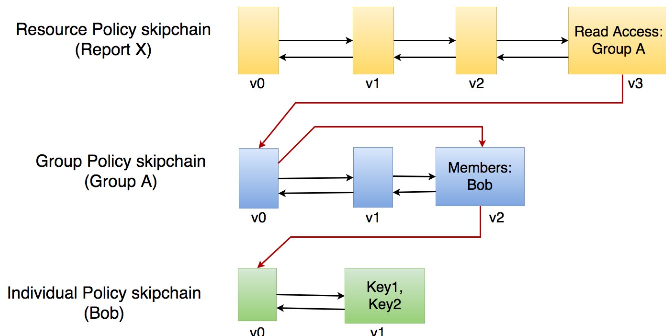

Figure 6: Evolution of policies using skipchains.

D. Implementation

We implemented the policy structure and the access requests signing and verification in Go. The current version of the code supports policy creation and updates, rule creation and updates. The code also has functionality to sign and verify single signature and multi-signature requests. We developed a parser to evaluate expressions that use the logical operators AND/OR/NOT. Finally, the implementation also supports signing with path selection by the requester. The code is available on Github [6]. Note that we did not perform the integration of the access control system with the skipchain architecture in this project.

IV. Evaluation

We performed two types of tests after implementing the system, using the Go testing and benchmarking modules. First, we completed unit tests to ensure that all the implemented functionality was correct and worked as expected. Then, we conducted benchmark tests to evaluate the performance of the system. We describe the benchmark tests in more detail in this section.

We undertook benchmarking only for access request verification. This is because, out of all the operations, the verification takes the longest time to complete. Other functions, such as creation or updates to policies, are less time consuming. We considered three cases for benchmarking: verification of single signature requests, verification of multi-signature requests and signing of single signature request with path selection. Note that, in the last case, we looked at the signing function rather than the verification function since the path selection happens during signing.

For our experimental setup, we created functions that could take in inputs such as number of subjects, expression parameters, depth of a policy etc., and create a policy environment that we then used for benchmarking.

A. Single signature Request Verification

We benchmark the request verification time for single signature requests. The request verification time can be considered to be the sum of the signature verification time and the time taken to find the path from the target policy to the requester. We vary the *depth* of the requester. The depth refers to the distance between the target policy in the request and the parent policy of the requester. The depth expresses the levels of checks that a verifier has to undertake to find the path. Our reasoning behind varying the depth is to investigate whether the checking for access proof has a significant impact on the request verification time.

Figure 7 shows the variation in request verification time with depth of the requester. Note that since we use a log scale, the lowest value is set to a value close to zero (0.01). Interestingly, we see that most of the request verification time goes into signature verification. The signature verification takes ≈ 385 us and accounts for $92.04 - 99.94\%$ of the total time. We observe that even at a depth of 200, a relatively high scenario for a real-life access control system, path finding takes only about 35 us.

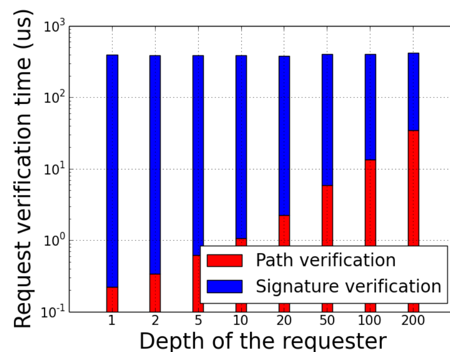

Figure 7: Benchmarking result for single signature request verification.

B. Multi-signature Request Verification

Since signature verification is the major factor contributing to request verification time, it will have an impact on multi-signature requests. We measure the verification rate for multi-signature requests. The verification rate is the number of requests verified per second. We create requests with varying number of signatures. Figure 8 shows the verification rate for requests with different number of signatures. We show the results for a requester depth of 2 and 10.

We notice that there is a significant reduction in number of requests that can be verified when the number of signatures increases. This is expected since the signature verification step has to be performed for each signature. We also notice that the depth of the requester does not play a notable role in the verification rate.

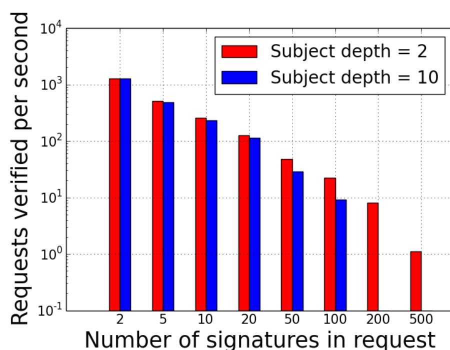

Figure 8: Benchmarking result for multi-signature request verification. Verification rate vs Number of signatures in request.

C. Signing with Path Selection

In our final benchmark test, we consider the case of a multi-path requester that signs with path selection. Since the path selection process has to find all possible paths and return them to the requester, we are interested in determining the time taken for this process. We create a requester that has multiple paths from the target policy and performs a path check during signing. We vary the number of paths and look at its impact on the signing rate - the number of requests signed per second.

Figure 9 shows the results for a requester at depths 2 and 10. The depth value is the same at all paths in this experiment.

We observe that the signing rate reduces rapidly with the increase in number of paths, from $\approx$ 320000 at 2 paths to $\approx$ 530 at 500 paths (depth = 2). This is probably because all possible paths now have to be discovered and stored. The depth also has an impact on the signing rate, most likely due to the searching and storage overheads.

We do not display the verification time here since it is dominated by the signature verification and looks similar to the single signature verification case (one signature is verified and one path is searched).

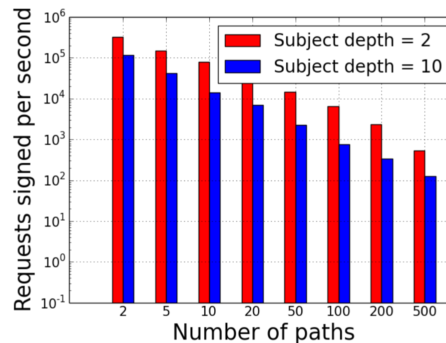

Figure 9: Benchmarking result in signing with path selection. Signing rate vs Number of paths of the requester.

V. Related Work

Kokoris-Kogias et al. [4] describe a system where a set of trusted servers act as an authority (termed *cothority*) and manage blockchains that can store identity information for users (such as SSH keys). These blockchains, called *Identity-Blockchains*, have forward and backward links. While backward links are used for ordering between blocks, Forward links provide a method for easy verification of changes to the user's identification data. CISC (CISC Identity Skipchains) [7] is an implementation of the concept described in [4]. Chaniac [5] allows for transparent software updates and introduces the concept of *skipchains*, a combination of skiplists and blockchains. The skiplists enable efficient traversal. We use the concepts in these works in designing how our policies can evolve with time.

Maesa et al. [8] outline a blockchain based access control system that uses the Bitcoin blockchain and XACML policies. In this system, policies are stored on the blockchain in a shorter XACML format and access rights are transferred by means of transactions. Enforcement of policies is done using the process defined by the XACML standard. FairAccess [9] is an access control framework that aims to provide a blockchain based access control system for IoT (Internet of Things) devices. It uses authorization tokens, a scripting language and the Bitcoin blockchain. Zyskind et al. [10] discuss the possibility of using a blockchain as an access control manager while using off-blockhain storage systems for data. This is further expanded in Enigma [11], a decentralized computation platform. Enigma provides functionality for users to store and run computations on data with privacy guarantees. It uses a blockchain for identity management, access control and logging.

VI. Conclusion

In this project, we designed, implemented and performed basic testing of a policy based access control model. We introduced

functionality for policy creation and updates, access request creation, signing and verification. Our system allows for easy management of policies, organization of users into groups and both access control and identity management. The code and API are available on Github [6].

Future Work

We highlight a few avenues for possible exploration and improvement:

- *THR operator:* Currently, we consider only AND/OR/NOT operators in Expressions. A consideration for a third operator would be THR. THR is a threshold operator of the form $\{THR : [thr_val, S1, S2, S3,]\}$. thr_val is a threshold value - a positive integer. The operation means that a threshold of signatures $>= thr_val$ needs to be obtained for the expression to be valid. This is especially useful for voting on policy updates, where a threshold of admin subjects have to agree on a change to the policy. A more complex expression can be created by introducing the concept of weights. It is possible that some subjects votes carry a higher weightage. Keeping weights in mind, the expression is $\{THR : [thr_val, S1, weight1, S2, weight2, S3, weight3]\}$, where $weight1$ is weight of subject $S1$ and so on. Votes for a change would be the sum of the weights of the subjects that agreed to the change.
- *Extensions of attributes:* Currently, the policy model is kept very simple. The reasoning behind this is to have a general model that can then be tailored based on the application. However, it might be interesting to explore the possibility of adding more features to this general model. For example, perhaps an Environment variable in a Rule to have a richer set of conditions than can be provided by Expressions.
- *Sub-policies and linking to other policies:* In this version of the architecture, policies are relatively simple and consist of a list of rules. We can explore a more advanced architecture where policies can have sub-policies which then consist of rules. In addition to this, it would be interesting to have the ability to link policies/sub-policies from other policies/sub-policies. Currently, we link policies through the Subject field in a Rule, but it would be feasible to have a policy that contains a list of other policies rather than rules.
- *Versioning and Consistency:* The current implementation of the policies has not been integrated with the skipchain architecture. This is planned for the On-Chain Secrets project. While integration would solve the issue of versioning, we still have to consider the problem of consistency – all the parties in the system would have to work on the same version of a particular policy. Exploring how consistency can be achieved with different policy versions and access requests/verification is an interesting option.
- *Alternatives to the 'one skipchain per policy' architecture:* Currently, we consider the one skipchain per policy design. This might not be the most appropriate, especially when the number of policies is large since maintaining skipchains for so many policy objects can prove to be cumbersome. Hence, we should also explore alternate designs - perhaps using a skipchain for groups of related policies rather than individual policies. We could also investigate using collections to group policies.

References

[1] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. In *International School on Foundations of Security Analysis and Design*, pages 137–196. Springer, 2000.

[2] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *Computer*, 48(2):85–88, 2015.

[3] Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sushil Jajodia. Policies, models, and languages for access control. In *International Workshop on Databases in Networked Information Systems*, pages 225–237. Springer, 2005.

[4] Lefteris Kokoris-Kogias, Linus Gasser, Ismail Khoffi, Philipp Jovanovic, Nicolas Gailly, and Bryan Ford. Managing identities using blockchains and cosi. In *9th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2016)*, 2016.

[5] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. Chainiac: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1271–1287, 2017.

[6] Sandra Siby. Decentralized access control. https://github.com/sandrasiby/cothority_template, 2017.

[7] DEDIS. CISC Identity Skipchain. https://github.com/dedis/cothority/tree/master/cisc, 2017.

[8] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. Blockchain based access control. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 206–220. Springer, 2017.

[9] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. Fairaccess: a new blockchain-based access control framework for the internet of things. *Security and Communication Networks*, 9(18):5943–5964, 2016.

[10] Guy Zyskind, Oz Nathan, et al. Decentralizing privacy: Using blockchain to protect personal data. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 180–184. IEEE, 2015.

[11] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471*, 2015.