Master Thesis

Prototype of the Accountable Key Infrastructure

Author[s]:
Baesso, Lorenzo

Publication Date:
2014

Permanent Link:
https://doi.org/10.3929/ethz-a-010144452

Rights / License:
In Copyright - Non-Commercial Use Permitted
Prototype of the Accountable Key Infrastructure

Master Thesis
Lorenzo Baesso
May 4, 2014

Advisors: Prof. Dr. A. Perrig, Dr. P. Szalachowski
Department of Computer Science, ETH Zürich
Abstract

Current Public Key Infrastructures (PKIs) rely on a very high level of trust in Certificate Authorities (CAs). Sometimes this can be dangerous as CAs can be compromised. The Accountable Key Infrastructure (AKI) aims at distributing trust among multiple parties in order to avoid single points of failure and at the same time give the possibility to domain owners to decide whom they shall trust. Moreover this infrastructure is able to handle common operations, like certificates registration and revocation, as well as detection of misuse cases (e.g. CAs or domain private keys get compromised).

My contribution consists in designing, implementing, and testing a prototype of AKI. In this paper I first introduce the parties involved and the certificates used in this architecture. I explain what these new certificates are used for and how to create them. Then the core part of my work is discussed: the actual infrastructure on which AKI runs. Meaning which requirements need to be considered, which are the possible workflows, and how the parties are actually implemented. I also give an elucidation on the possible use cases; that is, all possible functionalities domains are provided by this first implementation of AKI.

Finally I show the results from my tests and I make my considerations. The first part of the analysis focuses on how the time is fractioned for cryptographic operations, computational operations, etc. Then I examine the system’s performance. That is, I show the results of some stress-tests in which the aim is to find the number of processed requested per second. I also make a study of the introduced bandwidth overhead analyzing the size of the exchanged messages, for all use cases. The last part of the analysis is an informal security analysis in which I discuss the system’s resiliency to well known attacks (e.g. Denial of Service attack, replay attack, etc.) and I suggest which improvements could be done in the future.
# Contents

## 1 Introduction
1.1 Accountable Key Infrastructure ........................................... 1
1.2 AKI Policies and Certificates ............................................. 2
   1.2.1 AKI Policy ....................................................... 2
   1.2.2 AKI Certificate ................................................ 3
1.3 Parties Involved .......................................................... 3
   1.3.1 Domains .......................................................... 3
   1.3.2 Certificate Authorities ......................................... 4
   1.3.3 Integrity Log Servers ........................................... 4
   1.3.4 Validators ....................................................... 4

## 2 AKI Certificates and Policies Creation
2.1 AKI Policy Creation ....................................................... 5
2.2 AKI Certificate Creation ................................................ 8

## 3 AKI Network Architecture
3.1 System Configuration and Requirements ............................... 10
3.2 General Overview ......................................................... 11
3.3 Implementation .......................................................... 11
3.4 Structure of the messages .............................................. 13

## 4 Use Cases
4.1 Policy Registration ....................................................... 14
4.2 Certificate Registration ................................................ 16
4.3 Integrity Verification .................................................... 17
4.4 Policy Update ........................................................... 18
4.5 Certificate Revocation .................................................. 19

## 5 Performance Analysis
5.1 Detailed time analysis ................................................... 21
5.2 Efficiency analysis ...................................................... 27
5.3 Bandwidth overhead and latency time analysis ..................... 31

## 6 Security Analysis

## 7 Conclusion

## A Appendix
A.1 AKI Policy/Certificate Creation ...................................... 36
A.2 AKI Prototype Usage .................................................... 39

## Bibliography
This implementation of the Accountable Key Infrastructure is mostly based on the specification provided by P. Szalachowski, currently working as postdoctoral researcher in the Network Security Group of professor A. Perrig at ETH Zurich. Most of this specification is related to the AKI paper [KHP+13] and the SCP paper [SMP14]. The first describes the first version of AKI, while the second one contains all the latest improvements. Part of this work has also to be attributed to the master student L. Chen from EPFL.

1.1 Accountable Key Infrastructure

The Accountable Key Infrastructure is an innovative proposal for a new type of PKI, which is part of the more general future Internet architecture called SCION [ZHH+11]. SCION is a promising project that aims to overcome the lacks of the currently used authentication infrastructures (i.e. DNSSEC, BGPSEC and RPKI, Certificate Transparency, etc.).

The main characteristic of AKI is that it provides public visibility of all certificate operations (e.g. registration, revocation, etc.) so that if something looks suspicious, it will be detected very quickly (in the order of hours and not days like in the current solutions). This also aims to ensure a certain level of accountability within the architecture. The parties that are meant to log the certificate operations are called Integrity Log Servers (ILSs). ILSs make use of Merkle Hash Trees (MHTs) to efficiently produce proof of presence or absence of a certificate [Mer88]. Such a proof is essential for the verification of a certificate.

Although these parties have no motivation to make any profit by going rouge, it is still possible that some skilled attackers with plenty of resources manage to compromise them. The same holds for the Certificate Authorities, which currently are blindly and completely trusted by domain owners. In AKI there is a specific mechanism adapted to prevent this risk. That is, the usage of the so called Validators. Validators are trusted parties intended to monitor the ILSs’ activity, raising an alarm in case of detected misbehavior.

Another problem that AKI tries to address is the risky situation in which the browser cannot verify a certificate and the user is left to decide whether to proceed or not with the unsafe communication. Users usually do not have the knowledge to make IT security related decisions, so one goal of AKI is to move such a responsibility to the domain’s side. In AKI domain owners are supposed to define a single policy and “link” multiple certificates to it; precisely, one certificate per public key. In this policy domain owners can specify the rules to follow in order to update the policy itself or register and revoke certificates. These rules are expressed in terms of parameters that can be set by the domain owners.

Domains have also their way to protect themselves against one or more compromised CAs. That is, instead of having each one of their certificates signed by only one CA, in AKI a certificate can be signed by multiple CAs and the certificate, to be considered as valid, should have a certain threshold of valid signatures. That is why the certificates in AKI are also called Multi Signature Certificates (MSCs).
1.2 AKI Policies and Certificates

AKI policies and certificates play an essential role since the information stored within them is used for many security checks in AKI. Although in AKI a policy and a certificate have some similarities (e.g. they both contain multiple signatures), there are some essential differences. For instance, a domain has one and one only policy, while it can have multiple certificates (according to the number of public keys owned). Supposedly a policy certifies a long term key pair, while this is not the case for the certificates. Moreover, the purposes they serve to are very different, as explained in this section.

1.2.1 AKI Policy

A policy, also called Subject Certificate Policy (SCP), is a special MSC binding a policy key pair to parameters governing and protecting the usage of a domain’s certificate. Thus, by defining a policy a domain can establish the level of security of its certificates. This is achieved by setting some parameters that the newly generated certificates must be compliant with. For instance, a domain owner can specify which CAs are allowed to sign the certificates, as well as more technical things (e.g. TLS connection settings). These parameters are bound to the subject’s identity as well as to the policy public key. Except for the policy version, all these parameters are optional and the full list of them can be found in [SMP14]. Following are the parameters that I consider more relevant for the implementation of this prototype (part of the definitions are directly taken from the paper just cited).

- **CA_LIST**: list of domain’s trusted CAs, which can sign policies and certificates.
- **LOG_LIST**: list of domain’s trusted logs (i.e. ILSs), which can register a new policy or certificate. Moreover, they can also update current policies and revoke current certificates.
- **LOG_TIMEOUT**: how long proofs from the logs should be valid for a browser. This parameter typically varies from 1 to 24 hours.
- **UP_CA_MIN**: threshold number of signatures required to update a policy. This parameter is typically set within the range from 2 to 4.
- **UP_CA_TH**: threshold number of signatures required to update a policy when the previous policy’s key is lost. Usually \( \text{CA\_TH} = \text{CA\_MIN} + 1 \).
- **UP_COP_UNLINKED**: minimum cool-off period applied when a new policy is not linked with a previous one (i.e. the update request was not signed by the previous policy’s key). An upper bound exists for COP_UNLINKED (i.e. 7 days).
- **UP_COP_UNTRUSTED**: minimum cool-off period applied when a new policy is signed by at least one CA not included in CA_LIST. As for COP_UNLINKED, COP_UNTRUSTED also comes with an upper bound (i.e. 10 days).
- **POL_VER_IN_POL**: number assigned in ascending order starting from 1 to denote the version of the currently valid policy.
1.2.2 AKI Certificate

As already mentioned, a certificate is signed by multiple CAs. This is done in order to avoid that a single compromised CA sign a malicious party that is trying to impersonate someone else’s identity (i.e. by faking the legitimate website). Certificates are created and revoked more often than policies, as they are not supposed to certify long term keys. For instance they might get replaced every year, while this happens every 5-10 years for the policies (in case the policy key pair does not get compromised beforehand).

The certificate validation is a bit more elaborated than the policy’s. Indeed, certificates must first be compliant with the parameters set in the associated policy and secondly they must contain a certain threshold of valid signatures. Further more, browsers need to be modified so that they can verify AKI certificates, in case the domain they are trying to connect to supports AKI. In order to verify an AKI certificate, a browser needs to receive from the domain the proof of presence for that certificate (together with the certificate itself) and from the ILS that registered such a certificate the root hash of its Merkle Hash Tree.

The only extra parameter introduced in the certificates is called POL_VER_IN_CERT and it is used to link a domain’s certificate to the current policy. If the policy version and certificate’s policy version do not match, then the certificate is considered invalid. In this way we can always make sure that a certificate respects the security level established by the latest policy.

1.3 Parties Involved

AKI can run on a network with a few hundred machines but they all basically break down to four different types: domains, Certificate Authorities, Integrity Log Servers, and Validators.

1.3.1 Domains

Domains are the real beneficiaries of AKI. Supposedly a domain has multiple key pairs to certify but first it needs to get its policy key pair certified by multiple CAs. Then it can have any arbitrary number of AKI certificates created by the trusted CAs for the rest of the key pairs (the policy and certificate creation is described in Chapter 2). Once this first (out of the protocol) step is completed, domains can finally start to interact with the AKI servers. There are five different types of requests (Chapter 4) that domains may send to the AKI servers: policy registration request, certificate registration request, integrity verification request (basically a check on registered certificates), policy update request and certificate revocation request.

All a domain needs to do, once it has sent a request, is waiting for a reply from the contacted CA/ILS. If for any reason something goes wrong in the process and the domain does not get either a success or failure reply within a prefixed time limit (in the order of seconds), it can eventually try again after such a window of time is expired.
1.3. Parties Involved

1.3.2 Certificate Authorities

Certificate Authorities sign domain policies and certificates but not only. They are also used to perform some checks on domain requests and in case of success they forward them to the Integrity Log Servers. The ILS replies that pass through the CAs are checked as well and then forwarded to the domains. Because of this ability to check domain requests and validate ILS replies, CAs are said to act as Validators. In this work CAs are always considered to be Validators too but in reality Validators can actually be distinct parties in the AKI architecture.

1.3.3 Integrity Log Servers

Integrity Log Servers are probably the most important components of AKI because they are the parties that can actually change the state of the infrastructure. Meaning that they are the only parties in charged of registering/updating/revoking policies and certificates. Each ILS keeps track of all valid domain requests received and periodically (in the order of minutes) updates its status. That is, if a request is valid (i.e. it passes all security checks) it gets logged by the ILS, instead if a request is not valid it will not be logged and a failure message will be sent back. To maintain this information, ILSs keep an append-only database (at the moment in the form of separate files) of issued and revoked certificates as well as the current policies. This database is stored using a Merkle Hash Tree, which allows the log server to efficiently prove that a certificate has (not) been registered and that the database has not been retroactively altered. Additionally, an ILS uses a hierarchically-structured MHT that allows to efficiently retrieve the policy linked to a certain certificate as well as the the proof that such policy has been registered, or not.

1.3.4 Validators

A Validator is a trusted auditor that keeps local copies of all ILS databases. It periodically downloads the latest version with all the policy registration, certificate registration, and certificate revocation requests. Having these local copies enables the Validators to perform a series of security checks intended to monitor the behavior of the ILSs as well as guaranteeing a certain level of freshness and consistency of the information within the architecture. As soon as a compromised ILS is detected, a Validator informs all other parties in the AKI network but the actual interruption and rehabilitation of the compromised ILS is out of the protocol, so it must be carried out by the ILS administrator.
Chapter 2

AKI Certificates and Policies Creation

The creation of AKI policies and certificates is out of the protocol; meaning that this step is supposed to be carried out by a collaboration between domains and CAs before domains can ask the AKI servers to register them. As a part of my work I have created a program that CAs can use to generate AKI policies and certificates. They just need to receive from the domains the values they would like to have in the fields (e.g. subject name area, AKI parameters) and of course the public key to certify. The program was written in C++ and uses the OpenSSL APIs [YH14] for all cryptographic operations. Detailed information about the creation of AKI policies and certificates can be found in the appendix of this paper.

2.1 AKI Policy Creation

An AKI policy is the result of a concatenation of multiple X.509 objects, each one containing a different CA's signature. The reason why the X.509 standard is the one chosen for the creation of policies and certificates is its wide acceptance and therefore the need to keep a certain backward compatibility with the current PKIs. Nevertheless, AKI requires further information in order to function and such information is stored in X.509 extra fields.

One characteristic of these new policies is that the public key field is set in such a way to be the same for all of its parts (i.e. X.509 objects), as well as for the AKI parameters. This means that all these parts look exactly the same, exception made for their signature field. Figure 2.1 corresponds to a single part of a certificate. It looks very similar to the current certificates, just with some extra fields to support AKI. For instance, we can tell from the figure that this policy's part was signed by the CA ca1.ethz.ch and requested by the domain domain1.ch. Then we can also see how domain1.ch configured its AKI security parameters.

That is just the first step in order to build an AKI policy though. Supposing that domain1.ch asked four other CAs to certify its public key, we would expect four more parts. Therefore the result would be something like in Figure 2.2. We can see clearly that all parts were signed by different CAs, but in fact they all contain the same information. The purpose of this is first to allow a domain to choose a set of CAs to trust (and not only one) and consequentially enforce a stronger level of security (in case one of those CAs went rogue).
2.1. AKI Policy Creation

Figure 2.1: One signed part of an AKI policy.
2.1. AKI Policy Creation

Figure 2.2: Example of an AKI policy.
2.2 AKI Certificate Creation

An AKI certificate is very similar to an AKI policy: it is as well the concatenation of multiple X.509 objects signed by different CAs, all containing the same public key but different signatures. There are substantially two differences between AKI policies and AKI certificates. First, given that a domain may own multiple public keys, a domain may have multiple AKI certificates (i.e. one for each of its public keys). Second, an AKI certificate contains only the extra AKI field POL_VER_IN_CERT.

Figure 2.3 presents a certificate’s part. Also in this example I suppose that five CAs are used to sign the public key. The final result is showed in Figure 2.4.

Please note that the value of the field POL_VER_IN_CERT in the AKI certificates has to correspond to the value of the field POL_VER_IN_POL in the AKI policy, in order for the certificate to be valid.

```
Data:
Version: 3 (0x2)
Serial Number: 72545 (0x1b61)
Signature Algorithm: sha1WithRSAEncryption
Issuer: C=CH, ST=Zurich, L=Zurich, O=cal.ethz.ch, OU=cal.ethz.ch, CN=cal.ethz.ch
Validity
Not Before: Apr 13 17:25:50 2014 GMT
Not After : Apr 13 17:25:50 2015 GMT
Subject: C=CH, ST=Zurich, L=Zurich, O=ETH, OU=NetSec Group, CN=domain1.ch
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
Public-Key: (2048 bit)
04:34:97:bd:3a:0c:4e:21:9b:4c:bd:16:c4:04:af:
7d:9d:00:00:00:00:63:dd:6d:74:6e:87:77:47:dd:
f8:d9
Exponent: 65537 (0x10001)
X509v3 extensions:
X509v3 Basic Constraints: critical
CA:FALSE
X509v3 Key Usage: critical
Digital Signature, Key Encipherment
X509v3 Subject Key Identifier:
POL_VER_IN_CERT:
1
Signature Algorithm: sha1WithRSAEncryption
```
Figure 2.3: One signed part of an AKI certificate.
2.2. AKI Certificate Creation

Figure 2.4: Example of an AKI certificate.
Chapter 3

AKI Network Architecture

In this chapter the actual implementation of the Accountable Key Infrastructure is discussed. First the system requirements are mentioned, then a general overview of the network architecture is given to the reader. The programs used to run the entire system are all written in C++ and use the OpenSSL APIs for all cryptographic operations. The messages exchanged are formatted using the JavaScript Object Notation (JSON). Detailed information about the usage of the AKI prototype can be found in the appendix of this paper.

3.1 System Configuration and Requirements

Since this is an early implementation of AKI, a number of requirements and assumptions are necessary in order to guarantee a correct system behavior. These requirements are mostly out of the protocol, so it is responsibility of the system administrators to make sure they are satisfied.

- The lists of all CA and ILS domain names in AKI must be kept up to date on each CA/ILS machine.
- On all parties involved there must be installed one of the programs I created in this work. Such a program allows the parties to send and receive messages, as well as process them.
- CAs and ILSs must be started before domains start to send requests. To be able to receive messages they need to listen on a predefined port (the same one for all parties).
- All CA and ILS certificates are already distributed among all parties (i.e. domains, CAs, ILSs, and end users). This is achieved in the same way current PKIs handle the distribution of root CA public keys; thus they are integrated within the browsers. It is strongly recommended to use at least 2048 bit keys to sign these certificates.
- Each ILS has also a simple HTTP server written in Python running and listening on a predefined port (different from the one used to exchange messages). This server is needed by the Validators in order to update their local copies of the ILS logs, by actually downloading the copies in the format of files.
3.2 General Overview

Figure 3.1 shows the entire AKI network architecture. More representations will follow to illustrate all possible workflows, based on the specific use cases, and the reader will see that sometimes only a part of the network is actually used. This lets us to cut off some latency time and unnecessary steps wherever it is possible.

![Figure 3.1: Message flow in the AKI network architecture.](image)

Note that the user is included too in the schema as he also benefits from AKI but that is not the only reason. He actually requires a modified version of his browser since it has to be able to distinguish current certificates from AKI policies and certificates, and of course verify them accordingly. However, since that work is out of the scope of this thesis, from now on the user will not be taken into account and every process will start from the domain side. Validators are included too in the schema since can be deployed on separate machines but, as already mentioned, in this work they are considered to be integrated within the CAs.

3.3 Implementation

This section focuses on the implementation of the single parties. They are all pretty similar to each other, so the following considerations are valid for any CA or ILS. Domains are still very similar bust since all they need to do is send a message and wait for a reply, the reader can imagine them as a simplified version of CAs and ILSs (based on the same implementation concepts though). All these machines act as servers and they internally use multithreaded queues. For their implementation I took inspiration from an article written by the software architect and blogger Vic Hargrave [Har13]. I kept what I considered to be already good and I added the necessary parts to make AKI work properly. Basically I extended his concept of **Multithreaded Work Queue Based Server**.
Figure 3.2 illustrates how the parties are related to each other and gives an insight on the implementation of a single CA/ILS.

AKI parties use TCP sockets [DC09] to send and receive messages in the architecture. They all listen on one predefined port and make use of internal queues to process the messages. More precisely, they use a consumer that pushes the incoming messages into an arrival queue. Then an arbitrary number of workers or consumers (i.e. threads) extract the messages from the arrival queue and process them according to the type of message. Once a worker has finished to process a message, it stores the next message to send into one of the outgoing queues. Which queue is selected depends on the destination of the just processed message. Each CA/ILS has an outgoing queue for every other CA/ILS in the architecture. Each outgoing queue has its own consumer, which extracts the outgoing messages as soon as they are ready and sends them to the destination associated to that queue.

In general, incoming messages are queued sequentially, processed and dispatched into the outgoing queues in parallel (depending on the number of workers set and on the number of cores available), and sent to the next destination in parallel (sequentially if we consider a single outgoing queue).

Moreover, this implementation can support the exchange of hundreds of messages simultaneously as the queues are actually mutex queues, so these shared resources do not generate access conflicts.
3.4 Structure of the messages

All AKI messages are formatted in JSON [Sri13] and most of them have the same structure. For instance PolRegReq, CertRegReq, and IntegVR do not, as they do not contain the signature. So the most complete structure for the messages is the following one:

```
{
    "type": MSG_ID,
    "content": "B64(…)",
    "TA": Timestamp,
    "ID": "Domain_name",
    "SGN": "B64(…)
```

Where:

- "type":MSG_ID defines the type of message. MSG_ID is an integer greater or equal to 1 used by the parties to call the right processing function.
- "content": "B64(…)" contains the base 64 encoding of another message. This inner message could have the same structure of the outer message or it might be a simpler JSON string containing only the necessary data that the parties need to exchange with each other.
- "TA":Timestamp tells at what time the message was created by the sender. Timestamp corresponds to the current Unix timestamp (i.e. number of seconds since 00:00 hours, Jan 1, 1970 UTC).
- "ID": "Domain_name" is the field where the domain name of the sender is set. Domain_name could be domain1.ch for instance.
- "SGN": "B64(…)" contains the base 64 encoding of the signature computed on the rest of the message. This is used by the parties to verify the messages.

Example:

```
{
    "type": 5,
    "content": "eyJ2x[. . . ]pY3n0",
    "TA":1398542763,
    "ID": "ils1.ethz.ch",
    "SGN": "fd3h4[. . . ]s2hth"
```
According to the type of domain request there can be five different scenarios and therefore different workflows within the AKI network. Meaning that not only different types of messages will be exchanged (as they are request dependent), but also they will be exchanged by the parties following different paths.

### 4.1 Policy Registration

Registering a policy is the first step that a domain should address. It is the most resource demanding operation but on the other hand it is the least frequent to happen (in the order of years). In Figure 4.1 the workflow of messages exchanged to process such a request is shown.

![Figure 4.1: Message flow for a policy registration.](image)
4.1. Policy Registration

<table>
<thead>
<tr>
<th>Domain — PolRegReq —&gt; CA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the policy registration request a domain can specify the policy to register, the ILS which should actually register it and the two trusted CAs used to monitor the specified ILS. CA1 keeps this request in memory as long as it is not processed or up to a certain time limit, after which the domain can try again to send the policy registration request.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CA1 — PolRegReq —&gt; ILS1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILS1 temporarily stores this request in memory and performs some security checks (e.g. verification of the signatures). Then it needs to synchronize itself with all the other ILSs, since only one ILS is allowed to continue the registration process for a particular policy. This is done to handle the case in which requests with the same policy are accidentally (or not) sent to multiple ILSs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ILS1 — PolRegUnqReq —&gt; ILSs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Once ILS1 has sent to all other ILSs a policy registration uniqueness request they check their current state and if they are not already trying to register the policy contained in the uniqueness request, they send a positive message back to ILS1. In case they are processing another request for the same policy, the tie is won by whoever has the oldest request. That ILS will proceed, while the others will stop.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ILSs — PolRegUnqRes —&gt; ILS1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILS1 in the meantime gathers all other ILS responses and only if they are all positive it will add the policy in its log. Of course these responses are all signed and verified in order to prove the ILSs’ authenticity. Afterwards ILS1 sends a confirmation message to all other ILSs again, this time meaning that they are allowed to add the policy in their log too.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ILS1 — PolRegConf —&gt; ILSs</th>
</tr>
</thead>
<tbody>
<tr>
<td>By receiving this message an ILS knows that it is allowed to register the specified policy. This is not done blindly though; indeed a series of verifications on the given policy are performed before this operation is performed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ILS1 — MSG1v1 —&gt; CA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>With MSG1v1 CA2 is given the policy uniqueness responses generated by all ILSs and it uses them to check their validity. The policy parameters are also checked.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CA2 — MOK1v1 —&gt; CA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA1 performs a series of security checks to make sure that CA2 is trustworthy (e.g. a verification of the policy uniqueness responses). Among the things checked is also included a part of MOK1v1 that is directly signed by ILS1 and cannot be modified by CA2 without being detected by CA1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CA1 — MOK2 —&gt; Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>At this point the domain can assume that the process was successfully completed and the policy registered.</td>
</tr>
</tbody>
</table>
4.2 Certificate Registration

A domain can register an unlimited number of certificates; one for each of its public keys. The workflow (Figure 4.2) is definitely lighter and this helps a lot since this operation is done quite frequently.

![Certificate Registration Diagram]

**Figure 4.2:** Message flow for a certificate registration.

**Table 4.2:** Certificate registration process in detail.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>CertRegReq —&gt; CA1</td>
</tr>
<tr>
<td>CA1 — CertRegReq —&gt; ILS1</td>
<td></td>
</tr>
<tr>
<td>ILS1 — MSG1v2 —&gt; CA2</td>
<td></td>
</tr>
<tr>
<td>CA2 — MOK1v2 —&gt; CA1</td>
<td></td>
</tr>
<tr>
<td>CA1 — MOK2 —&gt; Domain</td>
<td></td>
</tr>
</tbody>
</table>

In the certificate registration request a domain can specify the certificate to register, the ILS which should actually register it and the two trusted CAs used to monitor the specified ILS. CA1 keeps this request in memory as long as it is not processed or up to a certain time limit, after which the domain can try again to send the certificate registration request.

ILS1 performs some security checks (e.g. verification of the existence of a policy for such domain, verification of signatures). Then, if they all pass, it adds the certificate to its log and sends a positive message to CA2.

The same security checks done by ILS1 are performed by CA2 too and if they all pass a positive message is then sent to CA1. One of the things checked is the policy version for instance, as it is not allowed to register a certificate with the policy version of an obsolete policy.

Like in the policy registration process, CA1 performs some security checks before sending a confirmation to the domain. Moreover, the request is finally removed from the pending requests.

At this point the domain can assume that the process was successfully completed and the certificate registered.
4.3 Integrity Verification

This process is triggered by a domain whenever it needs to verify the presence or absence of one of its certificates in the AKI logs. As for the certificate registration, only one ILS is needed in this scenario (the one that is supposed to have registered such a certificate).

![Image](153x465 to 441x637)

**Figure 4.3:** Message flow for an integrity verification.

**Table 4.3:** Integrity verification process in detail

<table>
<thead>
<tr>
<th>Domain — IntegVR —&gt; CA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the integrity verification request a domain can specify the certificate to verify, the ILS which should actually check the existence of the certificate in its log (the same one that registered it) and the two trusted CAs used to monitor the specified ILS. CA1 keeps this request in memory as long as it is not processed or up to a certain time limit, after which the domain can restart the process.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CA1 — IntegVR —&gt; ILS1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILS1 first retrieves the current policy for the specified domain and then computes the proof of presence or absence for the given certificate, linked to the policy found for the considered domain. Finally it sends such a proof to CA2.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ILS1 — MSG4 —&gt; CA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA2 uses its local copy of the log in ILS1 to compute the same proof, so that it can verify if ILS1 is stating the truth. If not it will raise an alarm, otherwise it will pass the message (containing the ILS’s proof) to CA1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CA2 — RVOK1 —&gt; CA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>As for CA2, CA1 goes through the same verification steps and if all the security checks pass, it will remove the original request from the list of pending requests.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CA1 — RVOK2 —&gt; Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>At this point the domain can assume that the process was successfully completed and that the received proof is authentic and reliable.</td>
</tr>
</tbody>
</table>
4.4 Policy Update

Updating the policy is also a rare operation. Still, a domain might have the necessity to set a higher level of security for its certificates or maybe its private key has been compromised. This can be solved by registering a newer policy. Because of the way AKI is currently designed and implemented, once that the policy is updated, all linked certificates must be replaced (otherwise the policy version field will not match).

---

**Table 4.4:** Policy update process in detail.

| Domain — PolUpdReq — ILS1 |
|---|---|
| In the policy update request a domain can specify the new policy and the ILS that should actually update it. This request is directly sent to the trusted ILS and it is signed by the domain, so that such ILS can verify the domain’s identity. ILS1 temporarily stores this request in memory and performs some security checks (e.g. verification of the signatures). Then it synchronizes itself with all the other ILSs, as only one ILS is allowed to continue the updating process for a particular policy. This is done to handle the case in which requests with the same policy are accidentally (or not) sent to multiple ILSs. |

| ILS1 — PolUpdUnqReq — ILSs |
|---|---|
| Once ILS1 has sent to all other ILSs a policy update uniqueness request they check their current state and if they are not already trying to update the policy contained in the uniqueness request, they send a positive message back to ILS1. In case they are processing another request for the same policy, the tie is won by whoever has the oldest request. That ILS will proceed, while the other will just drop its request. |

| ILSs — PolUpdUnqRes — ILS1 |
|---|---|
| ILS1 gathers all other ILSs’ responses and only if they are all positive it will update the policy in its log. Of course these responses are signed and verified to prove the ILSs’ authenticity. Afterwards it sends a confirmation message to all other ILSs again, this time meaning that they are allowed to update the policy in their log as well. |
4.5 Certificate Revocation

Revoking a certificate is definitely the lightest operation for the system. The reason is dual: firstly the domain can directly contact the trusted ILS, secondly such ILS does not need to be synchronized with the other ILSs. A domain might need to revoke one of its certificates because the related private key has been compromised for instance.

<table>
<thead>
<tr>
<th>ILS1 — PolUpdConf —&gt; ILSs</th>
</tr>
</thead>
<tbody>
<tr>
<td>By receiving this message an ILS knows that it is allowed to update the specified policy. This is not done blindly though; indeed, a series of verifications on the given policy are performed.</td>
</tr>
<tr>
<td>ILS1 — MOK3 —&gt; Domain</td>
</tr>
<tr>
<td>At this point the domain can assume that the process was successfully completed and the policy updated. However it will need to wait the cool-off period to be sure that the new policy is the actual one being used.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain — CertRevReq —&gt; ILS1</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the certificate revocation request a domain can specify the undesired certificate and the ILS that should actually revoke it. This request is directly sent to the trusted ILS and it is signed by the domain, so that such ILS can verify the domain’s identity. ILS1 performs some security checks (e.g. verification of the signatures, linkage to the current policy). Then it revokes the specified certificate and sends a reply back to the domain.</td>
</tr>
<tr>
<td>ILS1 — MOK4 —&gt; Domain</td>
</tr>
<tr>
<td>Upon the ILS’ reply the domain can assume that the certificate was successfully revoked. The domain does not need to wait any longer, as this action has immediate effect.</td>
</tr>
</tbody>
</table>

Figure 4.5: Message flow for a certificate revocation.

Table 4.5: Certificate revocation process in detail.
Chapter 5

Performance Analysis

This analysis focuses on the system’s performance in terms of time spent to process the requests, efficiency (i.e. number of processed requests per second), and bandwidth overhead introduced because of the AKI messages. In the first part of the analysis I exam for each type of request how the time is fractioned for cryptographic operations, I/O operations, computational operations and so on. Then I aim to find out how many requests the system can process per unit of time. So I first make some considerations about how differently requests can be affected by design decisions in terms of processing time (e.g. what are the effects of varying the number of ILSs involved) and then from the time I deduce an estimation of how many requests AKI would be able to perform per second. Finally I make a study on the introduced bandwidth overhead taking into account which messages are exchanged and how big they are.

For simplicity I run all tests locally on a single machine. Also I create the requests in a way that all domains would use the same CA1, CA2, and ILS1. This choice enables me to find the guaranteed performance on each one of these parties; indeed, if the requests were "spread" among different CA1s, CA2s, and ILS1s the architecture would naturally be able to result in higher performance. A similar consideration is made at the level of the single parties. Each one of these parties has an internal queue where the incoming requests are first inserted and then extracted to be processed by a certain number of workers as soon as they are free. Once again, in order to see how the system performs under the worst conditions, the number of workers is set to one, so that all jobs are internally processed in a sequential manner. Practically speaking these considerations mean that the presented results correspond to the minimum guaranteed system’s performance. Given the dependency of the requests (e.g. a certificate registration cannot succeed if a domain does not register a policy first), all the tests require me to send the requests in this order: policy registration request, certificate registration request, integrity verification request, policy update request, and certificate revocation request. For instance I first send 100 policy registration requests and I wait for all replies, then I send 100 certificate registration requests and I wait again, and so on.
5.1 Detailed time analysis

In AKI there are different types of computations involved, such as: cryptographic operations (i.e. encryption and decryption), encoding and decoding operations, hash functions, I/O operations, and other (e.g. messages parsing etc.). This section focuses on discovering how the time is partitioned among all these different computations with regard to the type of request considered.

The types of computations considered are:

- General computations: message parsing, array and string modification, etc.
- I/O operations: reading the policy/certificate/key files, writing the log files.
- Log copies updates: updates performed by the CAs (i.e. download of the ILS log files).
- Certificate/policy part verifications: standard OpenSSL verification of a policy/certificate part against the root CA certificate that signed such part.
- Hash functions: computation of hash codes.
- Encryption/Decryption operations: signature creation and verification of the AKI messages.
- Encoding/Decoding operations: base 64 encoding and decoding of the inner part of the AKI message’s content.

The AKI architecture that I examine for this analysis is the simplest possible: two CAs and one ILS. The reason is that CA1, CA2, and ILS1 play the most fundamental roles in the architecture. In order to retrieve reliable results I use samples of 100 requests (i.e. 100 policy registration requests, 100 certificate registration requests, etc.).

First I present my findings with regard to the total processing time, as sum of the processing time of all three machines, and then I focus on the single machines one by one. Every time I report the exact processing time, the corresponding percentages, a graph based on the percentages (just to give to the reader a more effective vision of the partitioning), and my considerations.
In Table 5.1 the fractions of the total processing time in seconds on the three machines (CA1, CA2, and ILS1) are reported, as well as the corresponding percentages (Table 5.2), for each type of request.

**Table 5.1: Partitioning of the total processing time in seconds.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolReg</td>
<td>0.299s</td>
<td>0.041s</td>
<td>0.102s</td>
<td>0.402s</td>
<td>0.078s</td>
<td>1.895s</td>
<td>0.131s</td>
<td>3.948s</td>
</tr>
<tr>
<td>CertReg</td>
<td>0.781s</td>
<td>0.063s</td>
<td>2.230s</td>
<td>0.484s</td>
<td>0.101s</td>
<td>2.188s</td>
<td>0.368s</td>
<td>6.216s</td>
</tr>
<tr>
<td>IntegVR</td>
<td>0.569s</td>
<td>0.063s</td>
<td>2.590s</td>
<td>0.322s</td>
<td>0.062s</td>
<td>2.604s</td>
<td>0.190s</td>
<td>6.399s</td>
</tr>
<tr>
<td>PolUpd</td>
<td>0.095s</td>
<td>0.006s</td>
<td>0.000s</td>
<td>0.081s</td>
<td>0.015s</td>
<td>0.371s</td>
<td>0.032s</td>
<td>0.600s</td>
</tr>
<tr>
<td>CertRev</td>
<td>0.067s</td>
<td>0.005s</td>
<td>0.000s</td>
<td>0.023s</td>
<td>0.014s</td>
<td>0.349s</td>
<td>0.023s</td>
<td>0.400s</td>
</tr>
</tbody>
</table>

**Table 5.2: Partitioning of the total processing time in percentages.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolRegReq</td>
<td>8 %</td>
<td>1 %</td>
<td>28 %</td>
<td>10 %</td>
<td>2 %</td>
<td>48 %</td>
<td>3 %</td>
</tr>
<tr>
<td>CertRegReq</td>
<td>13 %</td>
<td>1 %</td>
<td>36 %</td>
<td>8 %</td>
<td>2 %</td>
<td>35 %</td>
<td>6 %</td>
</tr>
<tr>
<td>IntegVR</td>
<td>9 %</td>
<td>1 %</td>
<td>40 %</td>
<td>5 %</td>
<td>1 %</td>
<td>41 %</td>
<td>3 %</td>
</tr>
<tr>
<td>PolUpdReq</td>
<td>16 %</td>
<td>1 %</td>
<td>0 %</td>
<td>13 %</td>
<td>3 %</td>
<td>62 %</td>
<td>5 %</td>
</tr>
<tr>
<td>CertRevReq</td>
<td>14 %</td>
<td>1 %</td>
<td>0 %</td>
<td>5 %</td>
<td>3 %</td>
<td>73 %</td>
<td>5 %</td>
</tr>
</tbody>
</table>

In general, the computations keeping the whole architecture busy for most of the time are the cryptographic operations (i.e. encryption and decryption). This makes sense as they are known to be very expensive operations, time and hardware resources wise. Further more, they are performed for every message sent or received in the architecture, included those case in which a signed message is encapsulated into another message.
5.1. Detailed time analysis

The second most time consuming operation is given by the updates of the log copies on the CAs’ side. This component is missing for the policy update requests and certificate revocation requests because in those processes domains can contact ILS1 directly, without involving the CAs. In these tests, for each one of the first three processes, both CAs need to update exactly once their local copy of ILS1. So for each one of the first three columns in the graph the update of the log copies occurs twice, one on CA1 and one on CA2. Afterwards there are the general computations and the verification of the policy/certificate parts. A minor part of the time is spent for all the other types of operations.

In Table 5.3 the fractions of the processing time in seconds on CA1 are reported, as well as the corresponding percentages (Table 5.4), for each type of request.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolReg</td>
<td>0.096s</td>
<td>0.011s</td>
<td>0.526s</td>
<td>0.172s</td>
<td>0.018s</td>
<td>0.290s</td>
<td>0.054s</td>
<td>1.167s</td>
</tr>
<tr>
<td>CertReg</td>
<td>0.331s</td>
<td>0.023s</td>
<td>1.131s</td>
<td>0.227s</td>
<td>0.015s</td>
<td>0.410s</td>
<td>0.175s</td>
<td>2.312s</td>
</tr>
<tr>
<td>IntegVR</td>
<td>0.152s</td>
<td>0.015s</td>
<td>0.796s</td>
<td>0.157s</td>
<td>0.022s</td>
<td>0.239s</td>
<td>0.084s</td>
<td>1.466s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolRegReq</td>
<td>8 %</td>
<td>1 %</td>
<td>45 %</td>
<td>15 %</td>
<td>2 %</td>
<td>25 %</td>
<td>5 %</td>
</tr>
<tr>
<td>CertRegReq</td>
<td>14 %</td>
<td>1 %</td>
<td>49 %</td>
<td>10 %</td>
<td>1 %</td>
<td>18 %</td>
<td>8 %</td>
</tr>
<tr>
<td>IntegVR</td>
<td>10 %</td>
<td>1 %</td>
<td>54 %</td>
<td>11 %</td>
<td>2 %</td>
<td>16 %</td>
<td>6 %</td>
</tr>
</tbody>
</table>

Figure 5.2: Partitioning of the processing time on CA1.
5.1. Detailed time analysis

By far, on CA1 the most time consuming operation is the update of the log copy of ILS1. This operation occurs only once for each type of request but it is enough to overshadow the cryptographic operations. This is reasonable if we consider that updating the log copy actually means contacting the HTTP server on ILS1, establishing a connection and downloading the log files created in the last update window of time.

The second reason why the cryptographic operations do not occupy CA1 so much is that the requests coming from the domains are not even verified, as already discussed, so the only messages CA1 verifies are the ones coming from CA2 and the inner message parts signed by ILS1. Of course CA1 still needs to sign the messages to send back to the domains, that is more expensive than just verifying a signature.

In Table 5.5 the fractions of the processing time in seconds on CA2 are reported, as well as the corresponding percentages (Table 5.6), for each type of request.

### Table 5.5: Partitioning of the processing time in seconds on CA2.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolReg</td>
<td>0.081s</td>
<td>0.015s</td>
<td>0.575s</td>
<td>0.092s</td>
<td>0.024s</td>
<td>0.597s</td>
<td>0.041s</td>
<td>1.425s</td>
</tr>
<tr>
<td>CertReg</td>
<td>0.224s</td>
<td>0.024s</td>
<td>1.099s</td>
<td>0.095s</td>
<td>0.024s</td>
<td>0.788s</td>
<td>0.147s</td>
<td>2.400s</td>
</tr>
<tr>
<td>IntegVR</td>
<td>0.328s</td>
<td>0.042s</td>
<td>1.794s</td>
<td>0.093s</td>
<td>0.025s</td>
<td>1.797s</td>
<td>0.075s</td>
<td>4.155s</td>
</tr>
</tbody>
</table>

### Table 5.6: Partitioning of the processing time in percentages on CA2.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolRegReq</td>
<td>6 %</td>
<td>1 %</td>
<td>40 %</td>
<td>6 %</td>
<td>2 %</td>
<td>42 %</td>
<td>3 %</td>
</tr>
<tr>
<td>CertRegReq</td>
<td>9 %</td>
<td>1 %</td>
<td>46 %</td>
<td>4 %</td>
<td>1 %</td>
<td>33 %</td>
<td>6 %</td>
</tr>
<tr>
<td>IntegVR</td>
<td>8 %</td>
<td>1 %</td>
<td>43 %</td>
<td>2 %</td>
<td>1 %</td>
<td>43 %</td>
<td>2 %</td>
</tr>
</tbody>
</table>

**Figure 5.3:** Partitioning of the processing time on CA2.
5.1. Detailed time analysis

Unlike for CA1, it appears that CA2 occupies more time at signing and verifying the messages, which is actually what occurs. Indeed CA2 needs to verify the messages coming from ILS1 and it must sign twice the packets to send to CA1; once the outer message and once an inner part of the message (for all three types of requests). This amounts to a total of two signature creations and one signature verification, for each process. Instead, CA1 has to perform one signature creation and two signature verifications, which is computationally lighter. That is why CA2 spends more time than CA1 at performing cryptographic operations.

In Table 5.7 the fractions of the processing time in seconds on ILS1 are reported, as well as the corresponding percentages (Table 5.8), for each type of request. Note that only one ILS was used, so no time is spent on synchronizing the ILSs for policy registration/update requests.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolReg</td>
<td>0.123s</td>
<td>0.013s</td>
<td>0.000s</td>
<td>0.138s</td>
<td>0.037s</td>
<td>1.009s</td>
<td>0.036s</td>
<td>1.356s</td>
</tr>
<tr>
<td>CertReg</td>
<td>0.227s</td>
<td>0.015s</td>
<td>0.000s</td>
<td>0.163s</td>
<td>0.062s</td>
<td>0.990s</td>
<td>0.046s</td>
<td>1.504s</td>
</tr>
<tr>
<td>IntegVR</td>
<td>0.086s</td>
<td>0.008s</td>
<td>0.000s</td>
<td>0.072s</td>
<td>0.015s</td>
<td>0.568s</td>
<td>0.030s</td>
<td>0.778s</td>
</tr>
<tr>
<td>PolUpd</td>
<td>0.095s</td>
<td>0.006s</td>
<td>0.000s</td>
<td>0.081s</td>
<td>0.015s</td>
<td>0.371s</td>
<td>0.032s</td>
<td>0.778s</td>
</tr>
<tr>
<td>CertRev</td>
<td>0.067s</td>
<td>0.005s</td>
<td>0.000s</td>
<td>0.023s</td>
<td>0.014s</td>
<td>0.349s</td>
<td>0.023s</td>
<td>0.600s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolRegReq</td>
<td>9 %</td>
<td>1 %</td>
<td>0 %</td>
<td>10 %</td>
<td>3 %</td>
<td>74 %</td>
<td>3 %</td>
</tr>
<tr>
<td>CertRegReq</td>
<td>15 %</td>
<td>1 %</td>
<td>0 %</td>
<td>11 %</td>
<td>4 %</td>
<td>66 %</td>
<td>3 %</td>
</tr>
<tr>
<td>IntegVR</td>
<td>11 %</td>
<td>1 %</td>
<td>0 %</td>
<td>9 %</td>
<td>2 %</td>
<td>73 %</td>
<td>4 %</td>
</tr>
<tr>
<td>PolUpdReq</td>
<td>16 %</td>
<td>1 %</td>
<td>0 %</td>
<td>13 %</td>
<td>3 %</td>
<td>62 %</td>
<td>5 %</td>
</tr>
<tr>
<td>CertRevReq</td>
<td>14 %</td>
<td>1 %</td>
<td>0 %</td>
<td>5 %</td>
<td>3 %</td>
<td>73 %</td>
<td>5 %</td>
</tr>
</tbody>
</table>
5.1. Detailed time analysis

The results for ILS1 are quite different from the ones obtained for the CAs. As a first thing we can notice the total absence of the component due the updates of the logs. Naturally this part is not needed on the logs themselves.

Clearly the processing time is dominated by the cryptographic operations that are by far the most expensive operations. Similarly to CA2, ILS1 needs to verify the messages coming from CA1 and it must sign twice the packets to send to CA2: once the outer message and once an inner part of the message (for all five types of requests).

Note that the results for ILS1 would become very similar to what we would obtain for CA1 and CA2 if we removed the component given by the updates of the log copies (the reader can imagine to remove the green component in the CAs’ charts). However, this characteristic of the CAs was intentionally triggered to show what could happen in the worst case scenario. The analyses in the next section also include this factor, as it makes these performance analyses closer to the architecture’s performance in a real scenario.

Figure 5.4: Partitioning of the processing time on ILS1.
5.2 Efficiency analysis

Given the possible workflows of messages that may result upon the type of request considered, I distinguish two sets of tests according to the consideration that some of these workflows involve multiple ILSs, while the others do not. More precisely, policy registration and policy update requests have different processing times according to the number of ILSs introduced in the system. While certificate registration, integrity verification and certificate revocation requests are independent from this (they always require only one ILS).

Figure 5.5 shows the time that it takes to the entire AKI architecture to process 100 policy registration/policy update requests, for different numbers of ILSs. Thus, this time is calculated as the difference between the arrival time of the last reply minus the sending time of the first request.

![Figure 5.5: Time needed to the whole architecture in order to process 100 policy registration/policy update requests.](image)

As expected the relationship between the two dimensions is linear: the higher the number of ILSs, the higher the processing time. This is quite normal if we consider the fact that ILS1 needs to be synchronized with all other ILSs. Meaning that it needs to send a uniqueness request to all other ILSs, collect all responses and send back to all of them a confirmation message (Chapter 4).

Given the linearity of the processing time for these two types of requests, we can say that the system scales quite well and does not result in exponential increments.
5.2. Efficiency analysis

While for the policy updates only ILSs are involved, for policy registrations also the CAs play their part. So it is interesting to see how much time is spent on each one of these machines. This is what comes up from Figure 5.6. While interpreting the graph, it is important to keep in mind one thing; that is, when a batch of requests is sent (100 in this case), it is very likely that CAs and ILSs will end up working in parallel at some point (even if the internal jobs are processed sequentially). That is the reason why the blue line (time the entire architecture needs to process 100 policy registration requests) must not be seen as the sum of the processing time on the CAs and ILSs, instead it must be simply seen as the total time from the first request until the last reply.

For instance, from the graph one can tell that with 70 ILSs a little bit more than 20 seconds are needed to AKI in order to process 100 policy registration requests. In these 20 seconds CA1 and CA2 both worked for about 7 seconds, while the entire cloud of ILSs was busy for about 13 seconds.

It is easy to see how rapidly the processing time on the ILS cloud increases as the number of ILSs increases, while the one of the CAs increases very slowly. Another thing worth of being noticed is the relation between the two CAs. One may expect CA1 working more than CA2, given that CA1 needs to process two messages (PolRegReq and MOK1v1) and CA2 only one (MSG1v1). That is true but it is also true that CA1 basically only forwards PolRegReq, after performing the signature verification (which is not very time consuming).

From these results we can assume that the system is able to perform 5 policy registration or policy update requests per second, with 70 ILSs (which would be a very big number in reality). Still it might seem low but we should also remember that these machines can make use of more than one worker (so that jobs are internally processed in parallel, of course depending on the number of available cores), and more importantly that in reality different CA1s, CA2s and ILS1s would be used; hence, the entire work load would be better distributed all over the architecture.
The other three types of requests are independent from the number of ILSs, so I wanted to test the system’s scalability by gradually increasing the number of requests sent to the system. Figure 5.7 shows the time that it takes to the entire AKI architecture to process 100 certificate registration/integrity verification/certificate revocation requests. Thus, this time is calculated as the difference between the arrival time of the last reply minus the sending time of the first request.

At a first glance, it appears very clearly that there is a big difference in the steepness of the curves. While for the certificate registration and integrity verification requests the increment is rapid, for the certificate revocation requests the processing time increases very slowly. This corresponds to the expectations since the certificate revocation request is by far the lightest operation in AKI: it just requires a direct interaction between the domain and ILS1, also without the need of synchronization among ILSs. Registration and integrity verification of the certificates require instead the usage of CA1 and Ca2, therefore the difference in processing time.
5.2. Efficiency analysis

The next step then is to find the number of requests processed per second once that the system is "warmed up"; meaning that there is a sufficient amount of requests keeping the system always busy. Figure 5.8 shows this information.

![Figure 5.8: Processed requests per second.](https://via.placeholder.com/150)

From this graph we can assume that the whole architecture can process about 40 certificate registration and integrity verification requests per second and about 160 certificate revocation requests per second. Considering once again that these numbers correspond to the minimum performance guaranteed, since the system is put under stress conditions, they can be considered to be good results.
5.3 Bandwidth overhead and latency time analysis

For the bandwidth overhead analysis we are interested in discovering how much data is exchanged between the parties in AKI. This analysis does not take into account the actual network packets’ size, rather the mere payload’s size needed in AKI.

There are two kinds of data transmitted in the system: first there are the messages exchanged to make the parties communicate with each other and secondly there are the ILS log files that are periodically downloaded by the CAs in order to update their local log copies.

For the first type of data it is possible to make a quite precise estimation. For each type of process I report in the following tables the sizes of all exchanged messages and the total amount of data exchanged within one process.

Table 5.9: Policy registration process.

<table>
<thead>
<tr>
<th>Message</th>
<th>Size (bytes)</th>
<th>Quantity</th>
<th>Total (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolRegReq*</td>
<td>11893</td>
<td>2</td>
<td>23786</td>
</tr>
<tr>
<td>PolRegUnqReq</td>
<td>517</td>
<td>No. ILSs - 1</td>
<td>517 * (ILSs-1)</td>
</tr>
<tr>
<td>PolRegUnqRes</td>
<td>517</td>
<td>No. ILSs - 1</td>
<td>517 * (ILSs-1)</td>
</tr>
<tr>
<td>PolRegConf*</td>
<td>16293</td>
<td>No. ILSs - 1</td>
<td>16293 * (ILSs-1)</td>
</tr>
<tr>
<td>MSG1v1*</td>
<td>14376</td>
<td>1</td>
<td>14376</td>
</tr>
<tr>
<td>MOK1v1</td>
<td>3484</td>
<td>1</td>
<td>3484</td>
</tr>
<tr>
<td>MOK2</td>
<td>1944</td>
<td>1</td>
<td>1944</td>
</tr>
</tbody>
</table>

43590 + 17327 * (ILSs-1)

Table 5.10: Certificate registration process.

<table>
<thead>
<tr>
<th>Message</th>
<th>Size (bytes)</th>
<th>Quantity</th>
<th>Total (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CertRegReq**</td>
<td>9149</td>
<td>2</td>
<td>18298</td>
</tr>
<tr>
<td>MSG1v2**</td>
<td>26208</td>
<td>1</td>
<td>26208</td>
</tr>
<tr>
<td>MOK1v2</td>
<td>23400</td>
<td>1</td>
<td>23400</td>
</tr>
<tr>
<td>MOK2</td>
<td>1944</td>
<td>1</td>
<td>1944</td>
</tr>
</tbody>
</table>

69850

Table 5.11: Integrity verification process.

<table>
<thead>
<tr>
<th>Message</th>
<th>Size (bytes)</th>
<th>Quantity</th>
<th>Total (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntegVR**</td>
<td>9149</td>
<td>2</td>
<td>18298</td>
</tr>
<tr>
<td>MSG4**</td>
<td>26225</td>
<td>1</td>
<td>26225</td>
</tr>
<tr>
<td>RVOK1</td>
<td>17268</td>
<td>1</td>
<td>17268</td>
</tr>
<tr>
<td>RVOK2</td>
<td>23772</td>
<td>1</td>
<td>23772</td>
</tr>
</tbody>
</table>

85563
5.3. Bandwidth overhead and latency time analysis

### Table 5.12: Policy update process.

<table>
<thead>
<tr>
<th>Message</th>
<th>Size (bytes)</th>
<th>Quantity</th>
<th>Total (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolUpdReq*</td>
<td>12194</td>
<td>1</td>
<td>12194</td>
</tr>
<tr>
<td>PolUpdUnqReq</td>
<td>517</td>
<td>No. ILSs - 1</td>
<td>517 * (ILSs-1)</td>
</tr>
<tr>
<td>PolUpdUnqRes</td>
<td>517</td>
<td>No. ILSs - 1</td>
<td>517 * (ILSs-1)</td>
</tr>
<tr>
<td>PolUpdConf*</td>
<td>16701</td>
<td>No. ILSs - 1</td>
<td>16701 * (ILSs-1)</td>
</tr>
<tr>
<td>MOK3*</td>
<td>16701</td>
<td>1</td>
<td>16701</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>28895 + 17735 * (ILSs-1)</td>
</tr>
</tbody>
</table>

* These messages are affected by the size of the AKI policy (8660 bytes in these tests).

### Table 5.13: Certificate revocation process.

<table>
<thead>
<tr>
<th>Message</th>
<th>Size (bytes)</th>
<th>Quantity</th>
<th>Total (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CertRevReq**</td>
<td>9446</td>
<td>1</td>
<td>9446</td>
</tr>
<tr>
<td>MOK4**</td>
<td>13037</td>
<td>1</td>
<td>13037</td>
</tr>
</tbody>
</table>

\[22483\]

** These messages are affected by the size of the AKI certificate (6630 bytes in these tests).

The data downloaded by each CA grows over time as it is periodically downloaded (every a certain prefixed window of time, i.e. every two hours) but we can roughly assume that:

- If an ILS is idle in this window of time: an almost empty file will be transmitted (37 bytes).
- If an ILS is active (it has at least one request to process): 37 bytes plus the size of each PolRegReq/CertRegReq/PolUpdReq message processed in that window of time will be transmitted.

Given the estimations made and the sizes of the policies/certificates used, it is safe to conclude that the data overhead introduced is quite negligible.
Chapter 6

Security Analysis

In this chapter I discuss the system’s resiliency to well known attacks, mostly from a network perspective (i.e. Denial of Service attack, replay attack, etc.) and I suggest which improvements could be implemented in order to fix eventual vulnerabilities.

**Eavesdropping** Data confidentiality is defined as the non disclosure of data which must be kept secret from untrusted parties. Encryption is a common mean used to ensure confidentiality. In AKI the exchanged messages are not encrypted (i.e. they are sent in clear text). However, this is not really needed, since there is no actual confidential data exchanged among the parties involved. This relieves the parties from additional unnecessary work.

**Data Modification** An attacker could actually modify the data in the packets but not without the knowledge of the receiver. The reason is that almost all messages are signed by the sender and can therefore be verified by the receiver. Only PolRegReq, CertRegReq and IntegVR are not signed; the first two naturally because they are used to certify a public key for the first time in the system, while the latter is not signed because the integrity check can be asked by anyone.

**IP Address Spoofing** An attacker might use special programs to construct IP packets that appear to originate from other addresses but what matters is the ID field in the AKI messages. Such field is used by any party to retrieve the right public key to be used to verify the messages. So, even if an attacker changed that field, the verification would still fail.

**Brute force Attacks** Because of the length of the keys used it is simply infeasible for an attacker trying to send a huge number of requests and hoping to be able to guess the right domain’s key pair. The key pairs used are formed by 2048 bits RSA keys which are widely considered to be safe enough.

**Denial of Service Attack** One considered threat is Denial of Service Attack (DoS), which consists in limiting the availability of machine or network resources to their intended users. This attacker is actually the most likely to succeed against this version of AKI. As a first countermeasure every party has a queue in which it stores every request for a prefixed amount of time (quite similarly to the concept of cookies), so that if other identical requests are received, they are consequentially ignored. This helps but of course the system is still weak against a powerful DoS attack, such as a Distributed Denial of Service attack (DDoS).

A possible defense could be the usage of multiple machines and the implementation of a protocol that immediately enables a backup server as the primary one goes down.
**Reply Attack** These attack is quite pointless since, whenever there is the risk of a reply attack, a timestamp is inserted as part of the message’s hashed data; so that tempering it would be of no use to an attacker. On the CAs, there is a further freshness check which consists in verifying whether the request contains a timestamp equal or greater than the last time the local ILS copies have been updated. This is done in order to guarantee a certain level of consistency with the information currently stored on the ILSs’ side.

**Log Copies Forgery Attack** If an attacker starts sniffing the network, he will see that the files of the log copies are transmitted in clear text and they are not signed. He could then decide to alter them, so that the system would result in an inconsistent state, therefore availability would come less. However, since the copies are periodically downloaded by the CAs, the attacker should temper all downloads for an endless (or big enough) window of time because as soon as he stops the correct data is downloaded again (even the one logged in the past).

A solution to this attack would consist in storing together with a log file another file containing the signature over the log file’s content. It would then become easy for the CAs verify the downloaded data.

**Taking Over AKI** This attack would be very hard to implement but ideally possible. The attacker would require to compromise at least two CAs and one ILS, and somehow force a domain to use only the parties under his control. If the attacker does not succeed in compromising two CAs and one ILS, then this attack would not work as the parties are able to detect each other’s misbehavior, and one would be enough to detect two compromised parties.

In case the attacker succeeded, that compromised part of AKI would remain such as long as a not compromised party interacts with it. Once this interaction occurs an alarm could be raised and a gossiping protocol could be used to warn all other trustworthy parties.
In this paper I have introduced the importance of the Accountable Key Infrastructure and the new things that this model implies (i.e. new certificates, new parties, etc.). I presented the actual infrastructure on which AKI runs, which requirements need to be satisfied, which are the possible workflows, and how the parties are actually implemented. I also discussed in detail a set of possible use cases, according to the domain requests.

In the second part I showed the results from my tests and I made my considerations on them. The time tests met my expectations and gave me precise insights on which things are sometimes more time demanding (i.e. update of the Validators). From the efficiency tests came up that the architecture scales quite well, it presents a good stability and from the results obtained it is also safe to conclude that the system can guarantee a good performance. Further more, the bandwidth overhead introduced in the system by the AKI messages and the log files is quite low. Maybe it could be find a way to further compress this data or even avoid to create and exchange certain files (i.e. the log files created after an idle window of time).

In the security analysis the system’s resiliency to well known network attacks was discussed. For most of them AKI is already free and for others some improvements need to be done.

The most urgent and important part to work on regards the handling of rogue parties. Meaning, how the system should response once that a CA or ILS is misbehaving. The means to detect if a party is misbehaving and to disable it are there, but it must be defined exactly how to use them and what other steps this operation implies. For instance, in such a case, there would be need of a gossiping protocol that allowed trustworthy parts to warn the rest of the architecture.

It could also be decided to have the Validators to pseudo-randomly and periodically check the CAs and ILSs (i.e. by sending them some random integrity verification requests) and not only when domains ask them to.

Future improvements can be done in order to prevent DDoS attacks too. For example there could be a smart protocol that allowed the entire architecture to make use of backup machines. Supposing that a server is temporarily down or compromised, it could be designed and implemented a protocol able to substitute such a server with another one. This would increase the overall level of availability and resilience of the architecture.

Another necessary modification is the utilization of files containing the signature of a corresponding log file. This change is very important to the Validators side because they need to be sure that what they are downloading from the ILSs is authenticated and not forged by a malicious party.
Appendix A

Appendix

This appendix aims to help the reader at using the programs implemented in my work. The first part focuses in generating AKI policies and certificates; the second part helps to actually run AKI. Please note that since this is a prototype, it is assumed that all the public/private key pairs are stored locally (even the ones belonging to the CAs and ILSs). In reality this would not be the case and such a constraint would imply a few more steps as the domain should send a certificate request to a CA and get it signed. However, these steps are out of the scope of this thesis and were therefore skipped.

A.1 AKI Policy/Certificate Creation

The first step is the creation of all necessary private/public key pairs and CA certificates. Thus, the user needs to generate with OpenSSL all the key pairs that belong to the domains, all key pairs that belong to the CAs, and their corresponding root CA certificates. For this purpose the script gen_keys can be used. This script generates in the keys folder a folder for each domain containing a certain number of key pairs for the AKI certificates (i.e. one pair for each certificate) and a key pair for the AKI policy. Moreover, the script generates a folder for each CA/ILS containing a key pair and a self signed root certificate. To simplify things domain names and CAs names are set as: domain1.ch, domain2.ch, etc. and ca1.ethz.ch, ca2.ethz.ch, etc. In reality domain owners can set their own domain names, as well as CA administrators can set theirs. The only important condition is that the generated files respect the following formats:

<table>
<thead>
<tr>
<th>Object</th>
<th>Format</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain’s certificate key pairs</td>
<td><code>&lt;Domain_Name&gt;_&lt;Key_ID&gt;_priv.pem</code></td>
<td>domain1.ch_1_priv.pem</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Domain_Name&gt;_&lt;Key_ID&gt;_pub.pub</code></td>
<td>domain1.ch_1_pub.pub</td>
</tr>
<tr>
<td>Domain’s policy key pair</td>
<td><code>&lt;Domain_Name&gt;_priv.pem</code></td>
<td>domain1.ch_priv.pem</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Domain_Name&gt;_pub.pub</code></td>
<td>domain1.ch_pub.pub</td>
</tr>
<tr>
<td>CA’s key pair</td>
<td><code>&lt;Domain_Name&gt;_priv.pem</code></td>
<td>ca1.ethz.ch_priv.pem</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Domain_Name&gt;_pub.pub</code></td>
<td>ca1.ethz.ch_pub.pub</td>
</tr>
<tr>
<td>CA’s certificate</td>
<td><code>&lt;Domain_Name&gt;.pem</code></td>
<td>ca1.ethz.ch.pem</td>
</tr>
</tbody>
</table>
A.1. AKI Policy/Certificate Creation

Afterwards, the AKI certificate/policy parts can be created. First it is necessary to set the desired values for the AKI fields in the `config.cnf` file. Then every certificate part can be created, one by one, using the `certs` executable.

To generate an AKI certificate part:

```
./certs 1 <domain> <Cert_ID> <Cert_Part> <Key_ID> <Signing_CA>
```

Where:

- `<domain>` is the domain name (e.g. domain1.ch)
- `<Cert_ID>` is the certificate ID (e.g. 123). This argument is any arbitrary number greater or equal to 1 and is used as suffix in the file name.
- `<Cert_Part>` is the ID associated to a specific part of the AKI certificate (e.g. 3). Every certificate part has an ID and it is important that the sequence of IDs is a continuous sequence of numbers starting with 1.
- `<Key_ID>` identifies which public key has to be set in the certificate. (e.g. 1). It is the same suffix used in the creation of the domain’s key pairs.
- `<Signing_CA>` is the domain name of the CA that is supposed to sign the certificate part (e.g. ca1.ethz.ch)

Example:

```
./certs 1 domain1.ch 123 1 1 ca3.ethz.ch
./certs 1 domain1.ch 123 2 1 ca4.ethz.ch
```

Notes:

- The first part is signed by ca3.ethz.ch.
- The second part is signed by ca4.ethz.ch.
- They both contain as public key: domain1.ch_1_pub.pub (in the keys/domain1.ch folder).
- The resulting parts will have a "Cert_ID"."Cert_Part" appended in their file names (i.e. 123_1 and 123_2 in this example). Once they are concatenated in the final AKI certificate, only Cert_ID will be left in the file name (e.g Cert_domain1.ch_123.pem).
- The certificate parts are created in the `cert_parts/domain1.ch` folder.
To generate an AKI policy part:

```
../certs 2 <domain> <Policy_Part> <Key_ID>
```

Where:

- `<domain>` is the domain name (e.g. domain1.ch)
- `<Policy_Part>` is the ID associated to a specific part of the AKI policy (e.g. 3). Every policy part has an ID and it is important that the sequence of IDs is a continuous sequence of numbers starting with 1.
- `<Key_ID>` identifies which public key has to be set in the certificate. (e.g. 1). It is the same suffix used in the creation of the domain’s key pairs.

Example:

```
../certs 2 domain1.ch 1 2
../certs 2 domain1.ch 2 2
../certs 2 domain1.ch 3 2
```

Notes:

- These parts are all self signed.
- They all contain as public key: domain1.ch_2_pub.pub (in the keys/domain1.ch folder).
- The resulting parts will have a "Policy_Part" appended in their file names (i.e. 1, 2, and 3 in this example). Once they are concatenated in the final AKI policy, the file name will correspond to the domain name (e.g Policy_domain1.ch.pem).
- The policy parts are created in the cert_parts/domain1.ch folder.

Alternatively, the user could execute the script `gen_cert_parts`. This generates a set of AKI policy/certificate parts for a certain number of domains.

Finally all the parts need to be concatenated. At this purpose the `cat` tool can be used, for instance:

```
cd cert_parts/domain1.ch
cat Cert_domain1.ch_123_1.pem 
    Cert_domain1.ch_123_2.pem > Cert_domain1.ch_123.pem

cat Policy_domain1.ch_1.pem 
    Policy_domain1.ch_2.pem 
    Policy_domain1.ch_3.pem > Policy_domain1.ch.pem
```

These concatenations can also be performed in batch by using the script `gen_certs`, which places all policies and certificates in the domain_certs folder.
A.2 AKI Prototype Usage

The following instructions help the user to test the AKI prototype on a single machine. For a real scenario utilization only a few things would change, such as: the single programs should be placed on the separate machines, the domain addresses adjusted, etc.

ILSs’ key pairs and certificates:
The first step is to check whether in the keys folder there are the key pairs for all parties and the root CA/ILS certificates (previously created). If not, the script gen_keys can be used.

CAs and ILSs’ domain names:
The next thing to configure is the list of domain names of all CAs and ILSs to use. These lists are contained in the files CAs and ILSs respectively. A necessary condition is that for each one of the CAs and ILSs specified in these two files there is a corresponding folder containing the key pair and the self signed certificate, in the keys folder.

Virtual interfaces and hosts file:
At this point the virtual interfaces must be created. Executing the script 1_interfaces with root privileges, the user can create on top of the localhost interface 100 virtual interfaces for the domains, 100 for the CAs, and 100 for the ILSs. The user must also add into his machine’s hosts file the mappings between the domain names (of all domains, CAs, and ILSs) and the local IP addresses just used to create the virtual interfaces.

Logs execution:
Now it is possible to run the logs by executing the script 2_logs. This starts a simple HTTP server for each ILS pointing to a folder where the ILS can log the processed requests and from which all Validators will download the files created. The HTTP activity will be displayed in the shell from which the script was executed.

ILSs execution:
On a separate shell the user can execute the script 3_ilss to actually start the ILSs. From this moment all the ILSs specified in the ILSs file are listening for messages and ready to process them.

CAs execution:
For design reasons, each CA must be started on a separate shell by using the executable file ca. Anyway, this is not too bad since for the testing the system two CAs are already enough. The program expects as a input argument the domain name of the CA to start, so ca1.ethz.ch can be started by executing:

```
./ca ca1.ethz.ch
```
Domains execution:
At this point the CAs are up and listening for messages, as well as the ILSs, and the logs are ready to serve the log copies. Therefore the entire architecture is ready to process domain requests.
In order to send a single request, on a separate shell the executable *domain* must be launched in this way:

```
./domain <Req_ID> <domain> <PolicyA/AKICertA> <ILS1> <CA1> <CA2>
```

Examples:
- `./domain 1 domain1.ch domain_certs/domain1.ch/Policy_domain1.ch.pem ils1.ethz.ch ca1.ethz.ch ca2.ethz.ch`
- `./domain 2 domain1.ch domain_certs/domain1.ch/Cert_domain1.ch_1.pem ils1.ethz.ch ca1.ethz.ch ca2.ethz.ch`
- `./domain 3 domain1.ch domain_certs/domain1.ch/Cert_domain1.ch_1.pem ils1.ethz.ch ca1.ethz.ch ca2.ethz.ch`
- `./domain 4 domain1.ch domain_certs/domain1.ch/Policy_domain1.ch.pem ils1.ethz.ch`
- `./domain 5 domain1.ch domain_certs/domain1.ch/Cert_domain1.ch_1.pem ils1.ethz.ch`

Notes:
- The mappings between *Req_ID* and the types of requests are: 1 = PolRegReq, 2 = CertRegReq, 3 = IntegVR, 4 = PolUpdReq, 5 = CertRevReq.
- PolUpdReq and CertRevReq do not require CAs, so the *CA1* and *CA2* can be omitted.

The user, however, may need to send batches of requests. At this purpose, the script *4_domains* can be used to send batches of 100 requests. According to the input argument given to the script (between 1 and 5), different types of requests can be sent:
- `./4_domains 1` sends 100 different policy registration requests (one per each domain).
- `./4_domains 2` sends 100 different certificate registration requests (one per each domain).
- `./4_domains 3` sends 100 different integrity verification requests (one per each domain).
- `./4_domains 4` sends 100 different policy update requests (one per each domain).
- `./4_domains 5` sends 100 different certificate revocation requests (one per each domain).
Bibliography


