ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SIAGSS and the Formal Verification of Signing Ceremonies

Master Thesis

Fabian Andreas Murer

April 4, 2018

Advisors: Prof. Dr. Adrian Perrig, Brian Trammell
Department of Computer Science, ETH Zürich

Abstract

Keeping the signing key save and secure is a crucial part in the public key infrastructure (PKI). If an attacker manages to steal a signing key, he is able to sign requests by himself and making everyone believe that this request has been certified by the original key holder. The goal of this thesis is to develop a new approach how to store a signing key in an isolated environment, still being able to accept and sign requests. We achieve this by using an air-gapped channel and two subsystems, where one of them is completely isolated from any network, has reduced to no radio frequency emissions and in the optimal case not even attached directly to a power supply. The whole system should be able to handle a fair amount of requests a day while still be of relative low cost. Furthermore, we want to formally verify ceremonies when initialising, changing or backing up secret keys.

Contents

Chapter 1

Introduction

For hundreds of years, people have been using signatures, typically to bind a person to a contract. Since the digital age, digital signatures have appeared which basically have the same ulterior motivates, i.e., binding two objects to each other. Typical examples are binding a name to a person or a name to an IP-address like it is done in the *Domain Name Security Extensions* (DNSSEC). Such signing mechanisms most likely use a *public key infrastructure* (PKI). The very crucial part in every system that uses PKI is to keep the private/signing key secure. This not always an easy task to solve, as a system where the signing key is stored might be compromised or even stolen physically. One goal of this thesis is to figure out if it is possible to build a rather simple signing system that is located in a totally isolated and safe environment such that it can be used to sign assertions (e.g., the mapping of a name to an address) over an air-gapped channel. Based on the idea of Matsumoto et al. [5], the air-gapped channel will consist of two monitors standing in front of each other and two cameras pointing to the opposite monitor. The reason why we want to use such a visual air-gap is that one can easily monitor and audit what happens on the screens, i.e., on the channel. Furthermore, the system should be of relatively low cost and yet effective, since this system will mainly be used to sign assertions from the RAINS protocol used in the new Internet architecture SCION [6], where a lot of requests have to be handled.

Sometimes, managing secret keys like the signing key needs involvement of a human being. While a protocol implemented in a program simply follows the steps that it needs to do, a human can make mistakes, e.g., leave some step out or lose some key. The *control-plane PKI* of SCION [6] uses an *online* and *offline* asymmetric key pair. While the online key is used for frequent mechanisms, the offline keys are used for infrequent and safety critical operations like adding or removing a core AS to the *Trust Root Configuration* (TRC). Unlike the automated use of online keys, the use of offline keys needs

interaction with an administrator. Since no human is perfect, even a trained administrator can make mistakes. For this reason, the second goal of this thesis is to formally verify key ceremonies based on the work of Basin et al. [1]. By modeling the ceremonies including human interaction, we want to verify a safe and secure execution of the protocol despite the existence of fallible humans.

In the first part of this thesis we want to describe the signing system, how it is composed, how it works and what is needed to operate it. This is followed by discussing the key management of the signing system in chapter 4. In chapter 5 we start describing the protocol of a ceremony, followed by modeling and verifying it.

Chapter 2

Signing System

In this chapter we want to describe the actual signing system, it's parts and how it works. Furthermore, this chapter includes advices how to use the system correctly and how to deploy it secure and in isolation.

2.1 Idea

Before we start describing the system, we first want to elaborate the idea behind the system. The idea is based on the system from Matsumoto et al. [5]. In their project, they had one system component that stands in an isolated glass box. The system is equipped with a monitor and a camera. Now the only channel to and from this box is an air-gapped channel based on this monitor and camera. All the data that needs to be exchanged is encoded into a QR-code and read by cameras from the screen. In [5], the interaction with the isolated box only happens with human interaction, i.e., system administrators come and scan the QR-code from the monitor and show their own QR-codes to the system.

Unlike [5], we use two components that do not need any human interaction except of the initialization process. The main thought here is that we use the idea of [5] to create a signing machine which is completely isolated. The second part of the system is used to receive requests from different outsiders and convert the requests to QR-codes readable by the signing component.

2.2 Requirements

This section states the different requirements on the signing system, starting with the functional requirements followed by the non-functional requirements.

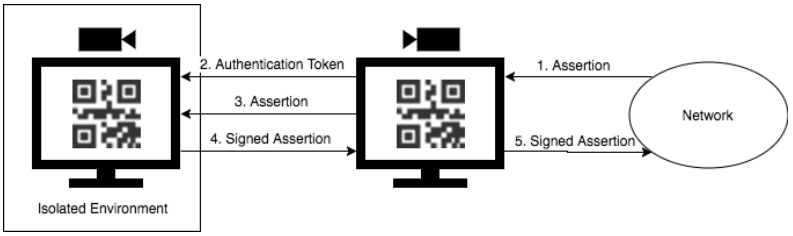


Figure 2.1: This image shows a basic overview of the system components and how they interact.

2.2.1 Functional Requirements

Let us start with the functional requirements of the system. The general goal of the system is to receive requests from the network containing some data. This data needs to be transfered over an air-gapped channel, signed and transfered back over the air-gapped channel and back to the network. As we have already described the idea before, this is done by using QR-codes and displaying and reading them from a screen respectively. So the system needs to be able to

1. receive requests from the network,
2. queue the incoming requests and handle them in *first come first serve* manner,
3. encode requests in a QR-code and display the QR-code on a screen,
4. read the QR-code from a screen using a camera,
5. decode the QR-code and get its data,
6. sign the data using the *Curve25519 Algorithm* [2],
7. send the signed request back to the originating sender

Furthermore, in order to avoid an adversary injecting some other data and getting it signed, the system needs to be able to authenticate the requests and thus only sign authenticated requests. Moreover, the system needs to be able to generate new public/private key pairs and to distribute the public keys.

2.2.2 Non-Functional Requirements

After describing the functional requirements, let us concentrate on the non-functional requirements. As we have already mentioned in section 2.1, one part of the system needs to be isolated as much as possible. In more detail, this means

1. *no* Internet connection,
2. *no* emitting radiation (Bluetooth, WiFi, etc.),

3. *no* direct connection to a power supply (in the ideal case)

where the last point could be achieved by charging the device over induction. Furthermore, the secret/signing keys need to be kept secret and should *never* leave the signing machine.

Since there is no certificate revocation in the SCION architecture [6] and they are using short lived certificates instead, there will be a lot of signing requests. An optimistic counting would yield in 2 million signatures a day, which is around 23 signatures a second. So in the ideal case, the system should be able to handle at least that amount of requests.

Moreover, in order to display and read the QR-codes, the System needs to have two screens as well as two cameras. Also, since the system is web-based as we will see in section 2.3, the system must run on a recent browser like Google Chrome.

2.3 System Components

We have now seen the system's requirements, so let us now describe the system itself. As already mentioned in section 2.1, the system mainly consists of two components. This is also visible in Figure 2.1, which shows a very basic overview of the system and how the components interact with each other. There, one can see the two main components. First on the left side in the isolated environment we have the signing system, we call it the *Signer*. The Signer is responsible for receiving and signing requests from the second component in the middle of Figure 2.1, which we call the *Signee*. The Signee receives requests from the network and transforms the requests into readable objects for the signer. Notice here, that points 2 and 3, i.e., authentication and sending the assertion, happens in the same message (see in Figure 2.2).

In the following sections we want to describe the two components in more detail. Both components are running `Node.js`[1] in version 8.6.0. In order to read and decode the QR-codes, generate QR-codes and sign the requests, we used the following external libraries:

Instascan: We used the Node module called `Instascan` from Schmich [7] to read and decode the QR-code from a monitor. This module makes it easy to select a provided camera, e.g. the internal webcam of a laptop, and add a listener that returns the content of the QR-code once the scanner has read and decoded one.

jQuery-qrcode: Unfortunately, the Instascan module does not provide a method to generate a QR-code, so we used a plug-in for `jQuery` pro-

[1] Node.js: https://nodejs.org/en/

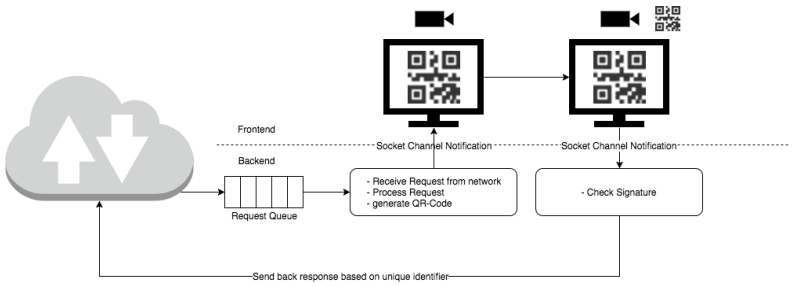


Figure 2.2: This figure shows a general overview of the *Signee*

vided by Jung [4]. This allows to use simple `jQuery`-method calls to generate a new QR-code.

TweetNaCl: In order to match the signing algorithm used in SCION [6], namely the algorithm called *ed25519* based on elliptic curves and introduced by Bernstein et al. [2], we used the Node module called `TweetNaCl` from Chestnykh [3].

2.3.1 The Signee System

In this section we want to describe the first part of the system that a request reaches, namely the *Signee*. The Signee listens on a predefined port (default port is 3000) for `POST` requests containing data that needs to be signed by the *Signer*. Thereby, the request needs to have the following parameters:

```
data=DATA_TO_SIGN&from=VALID_FROM&to=VALID_TO
```

Listing 2.1: Expected message format when sending a new request to the Signee

where DATA_TO_SIGN stands for the actual data that needs to be signed by the Signer, for example an assertion for a name resolution. The parameters *from* and *to* describe the desired validity range. Notice, that it is the desired validity range and that the Signer checks whether it is a valid range or not (more on this later).

Once a request is received, the Signee puts it in a queue, together with a callback to a request handler. Every request in the queue is then handled in *first in first out* (FIFO) order. In order to hinder an adversary to inject any irregular request, every request gets authenticated by appending the `HMAC` of the data to the request. We use the `HMAC` module from the Node.js crypto library[2] to achieve this. The request is then sent to the front-end, i.e., the browser, using *Socket Notifications*. Notice that the system expects a specific `JSON` format like in Listing 2.2. Moreover, notice the unique identification number. This identification number is computed by calculating the hash of

[2]Node.js Crypto Library: https://nodejs.org/api/crypto.html

the request. The ID is used to reference back to the original request such that the Signee is able to return to the right requester later on.

```
{
    id: UNIQUE_ID,
    data: { data: DATA_TO_SIGN,
            from: VALID_FROM,
            to: VALID_TO },
    mac: HMAC_OF_DATA
}
```

Listing 2.2: Expected message format from the Signee to the Signer

The browser then encodes this information into a QR-code using the *jQuery plug-in* mentioned before.

Once the Signee scans a new QR-code from the Signer's screen, the browser decodes it and sends the decoded content back to the back-end again using *Socket Notifications*. The Signee then checks whether the signature is indeed from the expected Signer and if yes, it sends the signed data back to its origin. In order to handle the right request and response objects, the Signee uses the unique id which it has computed before. The final response has the format like in Listing 2.3.

```
{
    assertion: { data: DATA_TO_SIGN,
                 valid_from: DATE_IN_UTC_FORMAT,
                 valid_until: DATE_IN_UTC_FORMAT },
    signature: SIGNATURE_OF_ASSERTION
}
```

Listing 2.3: Expected message format from the Signee back to the origin

You can find a graphical overview of the Signee in Figure 2.2. This overview shows the very high-level interaction between the front-end and the back-end of the Signee.

2.3.2 The Signer System

After having covered the Signee, we now want to describe the second component of the system, namely the *Signer*. Similar to the Signee, the Signer runs a Node.js server. However, unlike the Signee, the Signer gets its requests not from the network, but reads it from the Signee's screen. One reason for this is the requirement that the Signer needs to be totally isolated and therefore can not be connected to any network. In order to read the QR-code from the Signee's screen, the Signer also uses the Instascan module as described before. Once the QR-code has been read and decoded by the module, the content is sent to the back-end using an `AJAX POST request`. The

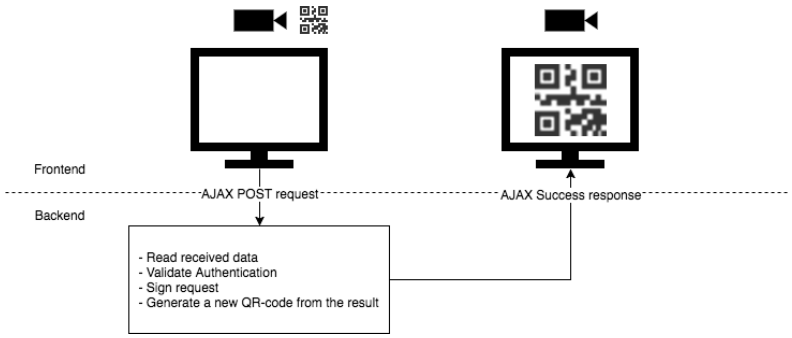


Figure 2.3: This figure shows a general overview of the *Signer*

back-end first verifies the authentication token, i.e., checking the hash of the data against the received hash-value. If it is correct, the Signer extracts the data from the request and checks the validity range. Thereby, the starting date must start in the future and the duration must not be longer than a predefined maximum value. If the validity range is valid, the Signer signs the data together with the validity range.

```
{
    id: UNIQUE_ID,
    assertion: { data: DATA_TO_SIGN,
                 valid_from: DATE_IN_UTC_FORMAT,
                 valid_until: DATE_IN_UTC_FORMAT },
    signature: SIGNATURE_OF_ASSERTION },
}
```

Listing 2.4: Expected message format from the Signer back to the Signee

The result is then sent back to the front-end using success callback of the `AJAX` request using the format in Listing 2.4. In the front-end, the response is encoded into a QR-code and displayed on the screen for the Signee to read it. The important part here is that the Signer sends back the request identifier so that the Signee can redirect the signed data to the right originator.

Figure 2.3 shows a graphical overview of the interaction between the front-end and the back-end of the Signer.

2.4 System Setup

Now that we have described the two system components, we want show our prototype setup. For our prototype setup we first met the assumption of complete isolation of the Signer system, i.e., we did not especially shield or isolate the system. However, we argue that since the system implementation does not need any Internet connection or some other connection, we could

simply use the method of [5] in order to make the Signer running in an isolated environment.

The overall system setup is very minimalistic. For each of the two components we simply used a commodity laptop. For the Signer we used a HP Aspire and for the Signee a Lenovo Thinkpad. Both laptops run a clean install of Ubuntu 16.04.3 LTS with the Google Chrome Browser and Node.js Server installed additionally.

2.4.1 Initialization

Before the system is ready to run correctly, both components need first to be initialized. The initialization includes downloading the corresponding source code and then in a next step installing the needed *node-modules*. The best way to do this is to simply run `npm install`. Notice that one will need the `package.json` file (see Listing 2.5) from the repository[3].

```
{
  "name": "signer",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Fabian Murer",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.2",
    "jquery": "^3.2.1",
    "morgan": "^1.9.0",
    "node-schedule": "^1.2.5",
    "qrcode-generator": "^1.3.1",
    "socket.io": "^2.0.4",
    "tweetnacl": "^1.0.0"
  }
}
```

Listing 2.5: Example `package.json` containing the needed node-modules

Next to installing the modules from `package.json`, one also needs to install *Instascan* [7] and the *jQuery plug-in* [4] for generating the QR-codes. Furthermore, a system administrator needs to place the Signer's public key on the Signee before starting the system, otherwise the Signee fails to verify the responses from the Signer. For the initial key generation, a script is provided (see Listing 2.6). So the administrator only needs to put the file `signer.pub` onto the Signee.

[3]GitHub Repository: https://github.com/fmurer/master_thesis

Figure 2.4: Example setup of the prototype showing the monitors while running

```
#!/bin/bash

keypair="$(node -e 'keyPair = require("../signer/node_modules/
    tweetnacl").sign.keyPair();
console.log(Buffer.from(keyPair.publicKey).toString("hex"));
console.log(Buffer.from(keyPair.secretKey).toString("hex"));
console.log("");')"

echo "${keypair}"

counter=0
while IFS='' read -r line || [[ -n "$line" ]]; do
        echo "$line" > key_"$counter"
        let counter++
done <<< "${keypair}"

mv ./key_0 ../signer/pk/signer.pub
mv ./key_1 ../signer/sk/sign_key
```

Listing 2.6: Script for initial key pair generation

After having installed all the needed modules, the system administrator should also manually launch the pairing of the systems by pressing the *p* button on the Signee's keyboard. This guarantees an initial shared secret between the Signer and the Signee which is then used for authenticating the messages sent from the Signee to the Signer.

2.4.2 Running Example

Once everything is initialized according to what we have seen before, the rest of the setup is really simple. The two laptops need to be placed opposed to each other such that the camera of the Signee can see the monitor of the Signer and vice versa.

Remark: *One thing that came across while implementing and testing the system was that a too bright monitor makes the camera unable to read the QR-code from it, because the white parts outshine the dark parts. So in order for the cameras to read properly, the brightness needs to be adjusted accordingly. In the current setup with the two laptops, this means reducing the screen brightness to maximum* 50%.

This is already all for a basic example. The only thing left is to start both servers with `node app.js`. Now one can send `POST` requests using the format mentioned in subsection 2.3.1 and send it to the Signee on the specific port (default is port 3000). Figure 2.4 shows a running example including a view on the monitors.

Chapter 3

Environment

In this chapter we want to describe the environment in which the system will be placed. This includes assumptions about the location like the building and the server room and also the attacker model and his capabilities.

3.1 Location

As we have seen in chapter 2, the system will be mainly used in the control-plane PKI of the new SCION architecture [6]. Since the control-plane is a very security critical environment, the system will be placed in the server rooms of the core ASes. Thereby, we assume and strongly rely on the fact that those infrastructures and buildings are secured accordingly, i.e., they are using *and* enforcing physical security measurements like secured entrance to the site, security guards controlling the terrain, surveillance cameras with operators 24/7 monitoring them and restricted areas where only authorised people have access to. Furthermore, every employee needs to have an own security code and every access to a room needs to be logged.

All those measurements are necessary to hinder unauthorised people like an attacker to access the site and getting access to the system (see section 3.2).

3.2 Attacker Model

In this section we want to describe the attacker's goal. The goal of the attacker is to get something signed by the Signer that is not supposed to be signed. He can achieve this using several approaches. For example an attacker could gain access to the signing key of the Signer by tricking the Signee to send some malicious code to the Signer who reveals the signing key. Another approach would be to inject a piece of paper containing a QR-code with the information encoded that he wants to have signed. However, the last point needs physical presence in the room where the system

is located, which could be achieved by either breaking into the building or by manipulating or compromising an administrator who could extract the signing key from the Signer.

We further assume that the attacker can not break cryptographic primitives like symmetric or asymmetric encryption as well as cryptographic hash functions. Moreover, we assume that the libraries and modules we used [3, 4, 7] are not vulnerable in any way.

Chapter 4

Key Management

In this chapter we want to describe how the system handles the different keys used for signing and authentication. To start with, let us explain what keys the system is actually using. All the keys explained in the following sections are saved in the local file system using the principle of least privilege.

4.1 Authentication

In order to prevent an unauthorised signing request getting signed by the Signer, we authenticate the incoming requests at the Signee. In subsection 2.3.1 we have already mentioned that we use the `HMAC` module from the crypto library of Node.js for this. To compute the `HMAC` of the received data and later verify it at the Signer, the Signee and the Singer need to have a shared secret.

This shared secret is computed using a *Elliptic Curve Diffie-Hellman* (ECDH) key exchange, which is supported by the crypto library of Node.js. We chose this algorithm over the standard Diffie-Hellman (DH) (which is also supported by Node.js), because ECDH is much faster than the standard DH used in Node.js. In ECDH we use the *secp521r1* curve. In order to exchange the DH half-keys, we use the air-gapped channel, i.e., encode them to a QR-code and display it on the screen. Once having both half-keys, the Signee and the Signer can compute the shared secret (let's call it *sk* for now).

Now in order to compute the authentication token of the received request, the Signee generates a `HMAC` using the `SHA256` algorithm together with the previously computed shared key *sk*.

We call the just described shared key establishment process *pairing* the two systems. In the very first initialisation phase, i.e., when setting up the environment, the two DH half-keys are not authenticated when sending over the

air-gapped channel. We argue that this can be handled since the setup needs to be done manually by a qualified administrator, so no malicious QR-code can be injected between the two systems. In later pairing steps, initiated by the Signee, the currently used shared key is used for authenticating the DH half-keys. Once the new shared key is established, the old one gets replaced.

4.2 Signing

As mentioned in section 2.3, we use the `TweetNaCl` library from Chestnykh [3] to sign the received requests. The library includes a method to generate new random key pairs. The keys have the following sizes:

Public Key: 256*bits*

Secret Key: 512*bits*

In the initialisation phase, the Signee needs to know the Signers public key, therefore the public key needs to be placed on the Signee. On the very first boot up of the Signee, it immediately asks the Signer for a *key schedule*.

Key Schedule: In our context, a *key schedule* describes a list of public keys together with their validity ranges. You can find an example in Listing 4.1. Thereby, the public keys are `hex` encoded.

```
{
  keys:
   { key1:
      { valid_from: 'Thu Jan 04 2018 09:52:15 GMT+0100 (CET)',
        valid_to: 'Thu Jan 04 2018 09:52:15 GMT+0100 (CET)',
        public_key: 'PUBLIC_KEY_1' },
     key2:
      { valid_from: 'Thu Jan 04 2018 09:54:15 GMT+0100 (CET)',
        valid_to: 'Thu Jan 04 2018 09:56:15 GMT+0100 (CET)',
        public_key: 'PUBLIC_KEY_2' },
     key3:
      { valid_from: 'Thu Jan 04 2018 09:56:15 GMT+0100 (CET)',
        valid_to: 'Thu Jan 04 2018 09:58:15 GMT+0100 (CET)',
        public_key: 'PUBLIC_KEY_3' }
    },
  signature: 'SIGNATURE'
}
```

Listing 4.1: Example of a Key Schedule of length 3

Notice that the key schedule needs to be signed, otherwise an adversary could inject his own key schedule and the Signee would simply accept it. In this example, the keys are valid for 2 minutes. However, one can choose the validity range for his needs.

When the Signee receives a key schedule, he sets a new scheduled job for every key. Those jobs are executed exactly when the next key starts to be valid. Furthermore, a short amount of time before the last key expires (in this case it is 30 seconds), the Signee asks for a new key schedule. The same key renewal rule is also applied at the Signer, i.e., when he creates a new key schedule, he also sets new scheduled jobs which fetch the new signing key from the list. All those time spans mentioned can be chosen arbitrarily. However, one should make sure that they match the specification of the system where the keys are integrated in. For example, the recommended usage period of a core AS key in the SCION architecture [6] is *6 months*.

Chapter 5

Formal Verification of Ceremonies

Bibliography

[1] D. Basin, S. Radomirovic, and L. Schmid. Modeling human errors in security protocols. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 325–340. IEEE, 2016.

[2] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, pages 1–13, 2012.

[3] D. Chestnykh. Tweetnacl. https://github.com/dchest/tweetnacl-js, 2017.

[4] L. Jung. jquery-qrcode. https://github.com/lrsjng/jquery-qrcode, 2016.

[5] S. Matsumoto, S. Steffen, and A. Perrig. Castle: Ca signing in a touch-less environment. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 546–557, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4771-6. doi: 10.1145/2991079.2991115. URL http://doi.acm.org/10.1145/2991079.2991115.

[6] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat. *SCION: A Secure Internet Architecture*. Springer, 2017.

[7] C. Schmich. Instascan. https://github.com/schmich/instascan, 2017.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SIAGSS and the Formal Verification of Ceremonies

Authored by (in block letters):

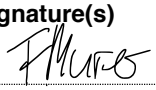
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Murer	Fabian

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, March 30	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.