UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

PKI For Automotive Applications

PKI für Anwendungen in Fahrzeugen

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Florian Dahlmann

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Sebastian Alexander Eckhardt

Die Arbeit ist im Rahmen einer Tätigkeit bei der Firma IAV GmbH entstanden.

Lübeck, den 27. September 2018

Abstract

We live in a world where technology is integrated in more and more parts of our environment. This means that nearly everything that contains electronics has computational power as well, which can be used to connect it to the internet. Therefore many devices are or will be able to communicate and exchange data with each other. This also applies to cars. As the technology in cars evolve, cars will be able to communicate with each other and with computers all around the globe. This raises a problem: The communication has to be secured against vicious attackers. We will therefore discuss in this thesis how this communication can be secured and focus especially on the embedded environment of the car with its performance restrictions. We will therefore discuss the challenges of the approach to build a Public-Key Infrastructure (PKI) and we will analyse the performance of typical cryptographic operations on an embedded system. Furthermore we will use our knowledge and build a practical implementation of a lightweight PKI, which could be used for the communication with cars and which solves the challenges that occur with a PKI.

Abstract

We live in a world where technology is integrated in more and more parts of our environment. This means that nearly everything that contains electronics has computational power as well, which can be used to connect it to the internet. Therefore many devices are or will be able to communicate and exchange data with each other. This also applies to cars. As the technology in cars evolve, cars will be able to communicate with each other and with computers all around the globe. This raises a problem: The communication has to be secured against vicious attackers. We will therefore discuss in this thesis how this communication can be secured and focus especially on the embedded environment of the car with its performance restrictions. We will therefore discuss the challenges of the approach to build a Public-Key Infrastructure (PKI) and we will analyse the performance of typical cryptographic operations on an embedded system. Furthermore we will use our knowledge and build a practical implementation of a lightweight PKI, which could be used for the communication with cars and which solves the challenges that occur with a PKI.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Lübeck, 27. September 2018

Contents

1 Introduction

As cars become more connected, the focus on security grows. Researchers like Weimerskirch and comittees from the EU and US are focusing more on how to secure vehicle communication. [Wei11] [Kar09] [74216]. The research focuses on three topics: authentication and privacy.
It becomes clear, that a Public-Key Infrastructure (PKI) as defined in RFC 5280 [CSF+08] presents an additional obstacle for the computational and bandwidth restricted microcontroller of a car. Certificate revocation lists could be avoided by providing multiple certificates with a short lifetime for cars.
There is no data that shows how costly it is to generate new certificates often and if there are ways to avoid this.
We will take a look at the research in a practical manner and will take a look which tasks in a PKI are most CPU-intensive. We will also examine the influence on computational time by using different libraries. Furthermore we will use a different way to avoid certificate revocation lists than many certificates with a short lifetime and implement this in a practical example.

2 Background

2.1 Car2car & car2x communication

Nowadays people are used to getting new apps on the smartphone every day and therefore expect that software matures, which means, that new features are being added when there is a high demand for them. As cars become more software driven nowadays and therefore run more and more software, the same behaviour will be expected from cars. Additionally there will arise mistakes in the process of software development, which will let the software work in a different way than intended. To fix these bugs and make the software better, the car has to receive software updates by communicating with the manufacturer (car2manufacturer). In the past cars have not been connected to the internet and therefore the only gateway to the manufacturer were motor vehicle workshops, but there are new possibilities with cars that are connected to the internet. The car would not have to be driven to a garage, where the owner would have to wait until the update is finished, but it would be possible to send updates over the air (OTA). This could happen at night, when the vehicle isn't used by the owner and therefore it would not cost any time of the owner. In this manner it is cheaper to deploy an update, because there is no middleman (like the car shop) that needs to be paid. Therefore a manufacturer can react quickly to software bugs and improve the software continuously.

On the other hand the new interfaces can be used to let cars become more connected. They can begin to communicate with their environment to gain more information about their surroundings and therefore increase the safety of the passengers. This begins with other cars, to exchange data about the position, speed and more to detect traffic jams and other dangers (car2car communication, c2c). This data can then be used to warn the driver in advance and prevent traffic accidents. Also traffic lights, traffic signs, etc. can broadcast their current status and therefore communicate with the cars. This information can then be processed to optimize the driving speed to have less red lights and improve the traffic flow overall (car2authority, c2a).

The problems that arise are that the car has to make sure that the data is from a trustworthy source. For software updates this means that, independent from the source that sends the update, it has to be proven that the software has been created by the manufacturer. For the c2c or c2a communication it is important that no one is able to forge their identity and irritate nearby cars, for example by sending wrong information about traffic signals.

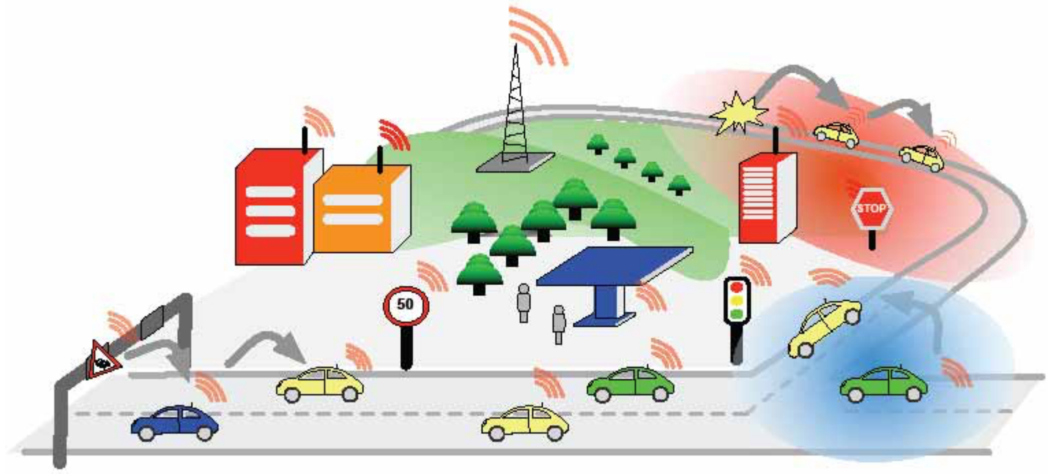


Figure 2.1: Car communication participants [ccc07]

2.2 Cryptography

When a car receives an update in a motor vehicle workshop, it has a private connection and therefore it receives the update in a way that could be compared with whispering to a person. We only have to assure that the content that is being whispered has really been created by the manufacturer.
With the connected car, it communicates with the internet and therefore sends information over channels that other parties can read or it sends the data wireless over the air. This can be compared to shouting into a room and therefore it is important to get a proof of the creator of the data and it should be secured that the other parties in the room cannot understand what is being shouted. For this purpose cryptography is often applied.
Cryptography is the science of information and communication security. Common use-cases are authentication, encryption, access control. There are three basic goals:

1. Confidentiality: A third party should not be able to gain any information from the communication of two parties.

2. Integrity: The receiver of information should know that the data has not been modified.

3. Authentication: The author of information should be verified.

An important principle in cryptography is that the security of the information should only rely on the secret that is used and not rely on the secrecy of a cryptographic algorithm itself, therefore it should be secure, even if the algorithm is not a secret. This is called the The Kerckhoffs Principle. [Vau06]

To illustrate different cases, we will look at Alice and Bob, which are two entities that communicate with each other. Eve will try to attack the communication, which means that she will try to undermine one of the goals.

2.2.1 Encryption

To keep the communication private (and fulfil Kerckhoffs Principle), it is usually necessary to encrypt the message. We will distinguish between symmetric and asymmetric encryption.
For symmetric encryption Alice and Bob need to share a common secret that only they both know and use it to encrypt the communication. This is called symmetric encryption. It works like a key to a safe in the real world: Alice can open the safe with the key and put a message inside and Bob can open the safe with the key to receive the message. The problem is that there could be Eve, who listens to their communication and therefore they would have to exchange a secret in private before they are able to communicate securely publicly. This is usually not possible via the internet, because there is usually no authentication of the communication participants. Another problem arises when Alice wants to communicate to more people than Bob. She has to store a secret for every person and so does Bob and every other person. This means that if we have n people who are communicating with each other, everyone has to store $n-1$ secrets and therefore there would exist $\frac{n*(n-1)}{2}$ secrets, which is way to much for embedded devices with small storage space, like a car, with millions of cars on the road.
The idea of asymmetric encryption has first been mentioned in 1976 Whitfield Diffie and Martin E. Hellman, and it is based on the following idea: To encrypt a message it is not necessary for Alice to use a secret, instead, she uses a public information that is specific to Bob and therefore only he can decrypt the message (with a secret only he knows). To realize this system, Bob has now a private key and a public key. He publishes the public key and keeps the private key secret. It works like a postbox on the street: Everyone can put a message inside (with the public key), but only the person with the private key can retrieve the letters. [DH76]
It is clear now that public key cryptography can be used to encrypt information, but it can do much more:

1. Key Exchange: The protocol can be used to exchange a key or even negotiate a key, to use symmetric encryption (which is much faster than asymmetric encryption, see table 2.1).

2. Non-repudiation & Integrity: By encrypting messages with the private key and de-

crypting them with the public key, they can be protected against malicious modifications and the source of the message can be proven.

3. Identification: Alice can check if she really communicates with Bob by sending him a challenge and checking the signature of the answer.

The remaining problem is the authentication of public keys. We can freely distribute public keys, but we can not be sure, who the real owner of the key is. We can solve this using certificates. They link a key to a identity. [PP10a]
The general idea is that Alice sends her public key (sK), an Identifier (ID) and a Signature which has been applied to the sK and the ID (sig(sK, ID)). Bob can now verify the sK and ID and therefore can be sure that it is Alice. The problem is now: How should the signature be created? If Alice would create her own, Bob had to save a public key for every other communication-partner in advance, so we have the problem that we wanted to solve. Instead the signatures are provided by a trusted third party, which is called Certificate Authority (CA). The CA verifies the IDs and then provides the tuple of sK, ID and the signature. Bob only has to retrieve the public key of the CA via a safe channel, but can verify the identity of every other participant. In practice there isn't just one CA, but for example one for the university and each institute would have their own. The university-CA would then sign the certificates of the institute-CAs and therefore they can create signatures and Alice and Bob only have to trust the university-CA. This is called chain of trust.
The CAs with all the services they provide are called Public-Key Infrastructure (PKI). The CA has to verify identities and issue, update and revoke Certificates.
Certificates do not only include the ID and the private key, they often have other information embedded as well. The most common public standard for certificates is the X.509 standard, which specifies which information can be embedded in a certificate. We will take a look at the most important fields of the X.509 certificate to show this at an example. [CSF+08]

1. Certificate Algorithm: There are multiple algorithms that can generate a signature. Here is specified which one has been used.

2. Issuer: There are multiple CAs that can generate trusted signatures. Here is specified which one generated this one.

3. Period of Validity: Normally a certificate has a date of expiry to prevent unlimited malicious use if the private key has been compromised.

4. Subject: This is the field for the ID.

5. Subject's Public Key: The public key that should be bound to the ID will be specified here.

6. Signature: The CA creates a signature over all the other fields of the certificate.

Therefore up to two signature algorithms can be involved: One for the signature and another one for the public key. [PP10b]
Before the period of validity reaches its end, the subject has to create a new certificate and prove it's identity to the CA to receive a signature for the new one.

2.2.2 Cryptographic hash function

Sometimes it could be really useful to prove to someone else that you know something, without revealing it. Therefore we have a secret that we want to reflect that on something, but prevent that someone could guess the secret from the portray. For example if we multiply a number with itself, the 25 can be calculated, but it is unclear if the original number is five or minus five.
This can be done in a more complex manner with cryptographic hash functions (will be called h). These functions portray data of any length to on data of a fixed length (the hash, e.g. 32 bytes). A one way hash function has the following properties:

1. To calculate the hash, only the input and the algorithm is needed. No additional information is needed to calculate the hash.

2. The hash has a fixed length of at least 2^n bits (with security parameter n), which is independent of the size of the input.

3. If we know X and the function h, the calculation of $h(X)$ should be easy.

4. The calculation only goes one-way and it is therefore hard to an unknown X for a known $h(X)$.

When it is hard to find two values that hash to the same result, a hash function is called collision resistant. This means furthermore that it is hard to find an $X \neq Y$ with $h(X) = h(Y)$. [PGV93]
Another use of a cryptographic hash function is to provide information that can be used to check whether a message has been modified. We assume that Alice and Bob have a shared secret S. When they send a message M they append $h(M||S)$, where $||$ means concatenation. Because of the secret, the hash can only be calculated by Alice and Bob and therefore they can calculate it when they receive a message and compare it with the attached one. If they differ, the message has been modified. This procedure is called Key-Hashing for message authentication (HMAC). [KBC97]

2.2.3 Elliptic curves

In cryptography there are four main realizations of asymmetric cryptography: RSA[1], DSA, El Gamal and elliptic curves over finite fields. We will explain elliptic curves and the advantages in comparison to RSA in the following section. For cryptography usually elliptic curves in a special field are being used, therefore the curves in a Galois field with p elements (p prime) can be defined with the equation

$$y^2 = x^3 + ax^2 + b, \text{ where } 4a^3 + 27b^2 \neq 0.$$

The important property is that two points of a curve can be added and will result in another point at the curve (see figure 2.2). The points and the addition form an abelian group. In addition there is the multiplication of a point with a positive integer k, which results in the sum of k copies of the point.

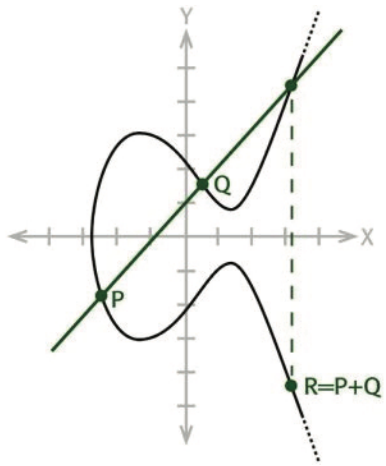


Figure 2.2: Addition of two points on an elliptic curve [KAS08]

In the cryptography Alice and Bob agree on a curve and a fixed point (F) on the curve. They then choose each a secret integer (A_k and B_k) which they multiply with the curve point and publish the result as their public keys (A_P, B_P). To encrypt the communication with each other they can simply multiply their private key with the others public key and therefore generate a shared secret that can be used with symmetric encryption. This is called Elliptic-Curve Diffie–Hellman (ECDH).

$$B_k \cdot A_P = B_k \cdot (A_k \cdot F) = A_k \cdot (B_k \cdot F) = A_k \cdot B_P$$

[1]RSA is a public-key cryptosystem, which is based on a problem that easy to solve if the factorization of a number is known, but appears to be hard if not. [KL07]

Time to break (in MIPS-years)	**RSA key-size** (in bits)	**ECC key-size** (in bits)
10^4	512	106
10^8	768	132
10^{11}	1024	160
10^{20}	2048	210
10^{78}	21000	600

Table 2.1: Comparison of strength of RSA and ECC [KAS08]

To calculate the private key, an attacker would have to solve the Elliptic Curve Discrete Logarithm problem (ECDLP). There is no mathematical nor theoretical evidence that the ECDLP is intractable, however the problem has been studied over many years and there are lower bounds for the problem in specific groups. [HMV04]
Without solving the ECDLP, an attacker would have to guess, which would take about $2^{\frac{n}{2}}$ operations. Because of the exponential increase, the keys and signatures do not have to be that large and can especially be smaller than RSA pendants with the same security. Also the point addition is computational expensive and therefore it is quite unlikely that there will be a general sub-exponential attack. There are sub-exponential attacks for special types of curves, but they can be avoided and there are no known attacks to recommended curves by NIST, Curve25519 by Bernstein and the Brainpool curves. Therefore ECC needs less computational power and space in comparison with RSA for the same security (see table 2.1). [KAS08]

2.2.4 Random number generators

A computer is a deterministic system and therefore always generates the same output for the same input. In this manner it is quite challenging to generate random data that is being needed by cryptography.
The common way to generate pseudo random numbers, is to start with a "seed" and perform mathematical operations on it to provide a stream of values that appear to be random. Therefore the randomness is directly dependent of the seed, which means that it is crucial to begin with a seed that can not be predicted and is as random as possible. Reliable sources are thermal noise, radioactive decay or a fast spinning oscillator, but not all computers have access to that data. Reliable sources can also be a spinning disk, noise from an unplugged audio device or a camera with lens-cap on. [rCS94]

3 Related Work

3.1 Common literature

The challenges of C2X Security and Privacy are often separated into several distinct parts by the literature. For example Weimerskirch et al describes the areas of communication security, privacy, certificate management and revocation, performance and physical security.

For communication security he refers to the US Standard IEEE 1609.2, which describes a basic security protocol which is based on certificates and elliptic curves.

In the privacy area, he distinguishes between two main concerns: Privacy against third party entities and privacy against authorities. Firstly he thinks that it is important to guarantee anonymity and prevent that a certificate can be linked to the license plate or a VIN[2], as well as long-term unlink-ability of two messages of the car to prevent tracking. To implement this, certificates have to be anonymised and a car needs to change the certificate quite often. In practical terms he suggests that a car should have multiple (e.g. 30) certificates with a short time to live and it would switch between them over time.[Wei17]

Privacy against authorities is more complex: More privacy means less control over the network. Therefore he recommends to implement privacy on an institutional level, which means that e.g. two authorities would have to collaborate to gain certain information.

For the certificate management he sees CAs as necessary, which creates the certificates and the certificates should be renewed by communication with road-side-units, which are placed next to street and distribute certificates for a CA. The handling of revoked certificates can be done in two ways: Either there has to be a public list of revoked certificates or the CA has a private list and therefore simply does not renew revoked certificates. The hierarchy could be separated by the location (EU, USA, ...) and sub-CAs for car manufacturers. Another crucial point is the deployment of the certificates in the first place. The manufacturers would have to flash them on the devices, but have to make sure that the parties involved cannot forge certificates or use valid certificates for their own purpose.

Performance-wise will a microcontroller in the car not be able to read and verify 1,000 or more messages per second. To solve this problem the car has to either select only messages that are relevant to it and dump all other ones to reduce the amount of verifications or the car needs security hardware that is able to verify huge amounts of messages. [Wei11]

[2] vehicle identification number, a unique number to identify a specific car

3.2 PKI for smart metering

The BSI in Germany published guidelines for a PKI for smart metering in 2017. It is focused on IOT (Internet of things) applications that run in houses through a gateway. Therefore the document defines standards and recommendations on the communication between the gateway and electronic counters, devices in the home area network and the wide area network with authorised participants. It is important to integrate a bidirectional authentication and to create an encrypted and integrity protected channel.
Therefore the application is quite similar: Multiple manufacturers create devices that include small microcontrollers/CPUs, but need protected communication.
The main idea is to use certificates with a PKI to achieve the authentication. In this manner there has to be a root-CA and multiple sub-CAs which then provide certificates for the devices of the consumer. The approach is a usual Public-Key Infrastructure, with one important catch: The management of the certificates (e.g. update, revocation, etc.) does not do the gateway or the devices themselves, but an administrator which controls the gateway. Another difference to the car world, is that that devices do not communicate with the internet directly, they communicate always through the smart gateway. [fSidI17]

3.3 Secure Vehicle Communication

The European commission funded in 2009 a project which is called SEVECOM,[3] which should do research on the security of vehicle to vehicle communication. They therefore divided the different aspects of the security in multiple modules, where each has its own purpose.
The security manager is responsible for the initial configuration of all security modules and also for the communication between them.
The identification and trust management module has to manage identities and credentials and therefore is responsible for keeping them up-to-date. The main idea was to manage multiple anonymous identities (pseudonyms, short-term public keys) and one identity which was there to receive new anonymous identities. Therefore when the main identity has been revoked, the vehicle will not be able to receive new anonymous identities and therefore can not authenticate itself any more, because the time to live for one pseudonym is short. This means that other vehicles do not need lists with revoked identities (more about revocation in section 4.1).
The privacy management module is responsible for privacy-enabled communication. It leverages the pseudonyms and allows vehicles to have a definite level of privacy while al-

[3] Secure Vehicle Communication

lowing to identify them as valid vehicles. It improves the privacy significantly by switching the pseudonyms often and therefore preventing tracking by eavesdroppers.
The secure communication module is responsible for the communication and doing it in a secure way. It communicates with nearly all other modules and it takes care of the complete communication process. It is divided into the secure beaconing component, the secure flooding component and the secure routing component. Beaconing is the process of broadcasting data in regular time intervals to all nodes that are nearby. This data could contain information about the location, speed or heading of the vehicle. Flooding is quite similar, but it is used to send information that then is being forwarded by other entities as well. Therefore it will continue to be forwarded until a specific time or in a specific area. The routing component has to ensure that the communication that is being received is from a valid vehicle, has not been modified and has not been rerouted in the network.
The in-car security module ensures that the communication between the wireless communication system and the in-car networks is protected. It therefore controls the access to vehicle data and ensures the correct provision. It has a firewall to control the access and a intrusion detection system which can create new firewall rules and monitors the traffic.
The crypto support module implements the security functions which are being needed by the other modules. It is a crypto component with an API, which provides the functions and a HSM component[4] with its HSM API, which provides random data and saves data like the keys in a secure manner. [Kar09]

[4]Hardware security module that provides fast and secure cryptography operations

4 PKI Challenges

There are a few points to consider when creating a public key infrastructure, which we will discuss in the following sections.

4.1 Revocation of certificates

When the CA (e.g. the car manufacturer) notices that it created a certificate with false information or a private key has been leaked, the certificate has to be declared as invalid. The obstacle is, that the certificate isn't in the hands of the creator, it is being used by someone and therefore it can not be changed. As a solution the CA can publish information about revoked certificates and everyone who checks the validity of a certificate has to check whether the certificate has been revoked. As this introduces new attack surfaces, the revocation remains a main challenge for PKIs. We will discuss possible options in the next paragraphs and compare them at the end.

4.1.1 Certification revocation lists (CRLs)

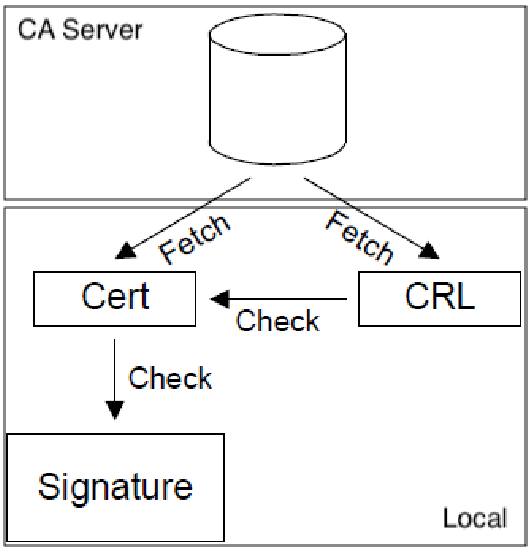


Figure 4.1: X.509 certificate usage model [Gut]

An obvious solution is to let the CA create a list with all revoked certificates (certificate

revocation list, CRL) and publish it online to make it available to all communication participants. When a certificate is being revoked by the CA, it will be added to the CRL. To check the validity of a signature, the validator has to take a look in the CRLs of the CAs in the chain, to prove, that no certificate in the chain is on one of the blacklists.
This results in a few problems: If the data has to be up to date in real time, the validator has to check the CRLs every time it checks a certificate, which creates additional bandwidth. Otherwise the CRLs could be updated in a scheduled interval (e.g. everyday), but then an attacker could use a certificate for up to one day (or another interval) after it has been revoked. Also a attacker could block traffic to the CA and then the validator would have no chance to check certificates for their validity.
And despite the bandwidth, the search in the list will always cost additional computation time. [Gut]

4.1.2 OCSP

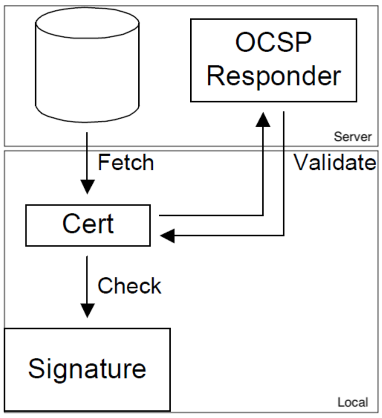


Figure 4.2: Certificate usage model with OCSP responder [Gut]

To prevent that the user has to fetch the CRL quite often and search through it, the OCSP[5] approach has been developed. The main idea is to let the validation be made by a server, the OCSP responder. Therefore the bandwidth and load with CRLs can be saved on the hardware of the validator, but an additional internet connection is needed. It is also possible to let the client send the OCSP response with the certificate (OCSP stapling), but the CA still has to answer a lot of OCSP requests. Which means that it has to search in the CRL and then sign a response, which signature also has to be checked by the validator.
Another problem is that the OSCP can only answer with "not-revoked", "revoked" and "unknown" where "not-revoked" doesn't necessarily mean good and for the status un-

[5]Online Certificate Status Protocol

known, the client still has to decide. It could mean, that the certificate has never been issued or the CRL was not reachable or no CRL has been found,... and therefore the client hasn't gained any knowledge about the validity. [Gut]

4.1.3 Certification revocation trees (CRTs)

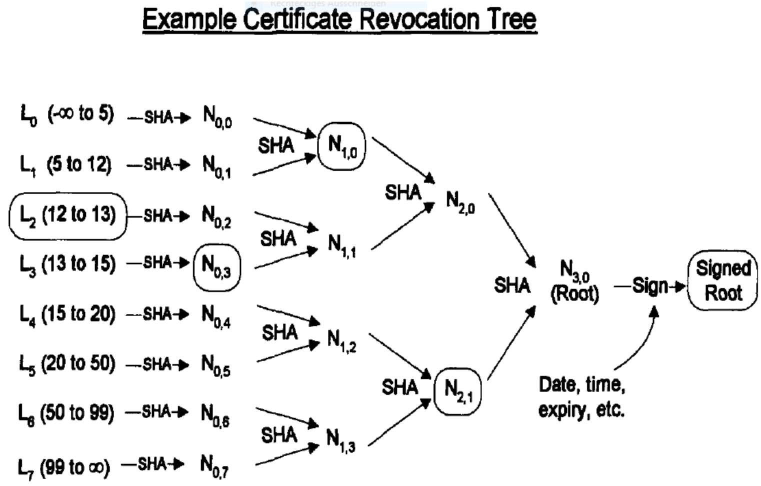


Figure 4.3: Certificate Revocation Tree [Koc98]

To solve the problems with CRLs and OCSP, Certificate Revocation Trees has been developed. The main idea is to have a data structure in which the OCSP Responder can search fast and give a useful answer back. For fast search, a tree is a plausible idea.
To give an advantage to the tree, the leaves are not just the certificates that have been revoked, they are ranges of certificate numbers (every range represents exactly one revoked certificate). A leaf (5,12) means that the certificate 5 has been revoked, but any certificate less than 12 and more than 5 is good. Of course the reason and date of revocation is also included (other information is possible). We then use the idea of Merkle-Hash-Trees for the nodes. Therefore a node $N_{i,j}$ of the tree is the hash of the nodes of the layer below them. For example $N_{2,1}$ is the hash of $N_{1,2}$ and $N_{1,3}$. The root will then be signed by the CA.
Because of the structure of the tree, the participants of the communication don't even need to save the full tree. It could be distributed by servers that answer validation requests from validators. The server just has to return a few nodes (circled in the graphic) and needs no cryptographic operations. The validator then has to check the hashes and the signed root. The client that tries to prove its identity can even provide the nodes itself and therefore the validator doesn't need an additional connection and the bandwidth-usage keeps low. [Koc98]

4.1.4 Novomodo

With the Certificate Revocation Tress there is still a lot of overhead from a bandwidth perspective, but we still have the goal to reduce it furthermore. One possible approach is provided by Novomodo, where only one hash is needed to prove that a certificate is still valid.
For Novomodo the CA generates for every certificate a random 160-bit value X_0, which is being kept secret. If we assume that we have a certificate that is valid for 365 days and it should be revoked in max. 24h, then the CA uses a public one-way hash function on X_0 for 365 times. This hash X_n (where $X_i = Hash(X_{i-1})$) is then being included in the certificate.
To prove now that a certificate is still valid on day i, just the hash X_{n-i} is being needed. Hashing it i times, it will be exactly the same as the hash in the certificate. The clue is, that there is no possibility to get the hash X_{n-1} when the only knowledge is X_n.2.2.2 Therefore only the CA can calculate the hash values, by knowing the secret value X_0. The CA will then provide a directory server which distributes hashes for all certificates that have not been revoked until the current day. The directory server doesn't even has to be trusted, because only the CA can calculate the values. Because of this, there could even be multiple directory servers by untrusted entities or the client that wants to be authenticated can even provide the X_{n-i} hash itself, to provide a proof of validity to the communication-partner. The additional bandwidth is only 160-bits and for the CA hashing is usually cheaper than signing. Especially the aspect that the communication with the directory-server does not have to be authenticated saves bandwidth and computational power. [Gen03]

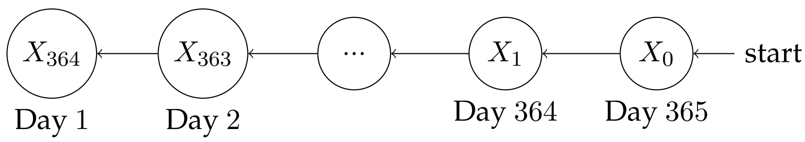


Figure 4.4: Order of hashing compared to the day of usage

4.1.5 Short lifetime

To avoid revocation altogether it is possible to just use very soon expiration dates and therefore give a certificate a really short lifetime. If every certificate is good for an hour, an attacker would only have an hour to hack it and when something wrong happens, a certificate would just be accepted for that hour. There are no additional CPU costs or additional bandwidth in the verification process for the client needed, but the certificate owner would have to generate new certificates quite often and let them being signed by a

CA.
One disadvantage is the cost of producing the certificates. A CA usually generates a new certificate for a car every year or even less and would then have to provide a renewed certificate every hour, which will result in heavy load. Also the client would have to generate many private keys, which is quite costly. This can be avoided when the client uses the same private key and just requests a new certificate from the CA.
The main problem is that cars could stay for a long time in a multi-storey without an internet connection and therefore it would not be able to get a new certificate if it has no internet connection for 6 months. Therefore a car would need to have multiple certificates in advance, especially with different private keys (otherwise it would not make a difference to a certificate with a longer lifetime. This would mean that it would need much more memory, because it needs to store multiple certificates (e.g. 365 certificates with a lifetime of one day). [Gut]
[MR01]
This approach can also be combined with other approaches, by not only differentiating between valid until revoked or expired. We would have three stages for a certificate: guaranteed valid (for a short period of time), valid until revoked (for the rest of the time) until it is expired. [Koc98]

4.1.6 Comparison

Over all it becomes clear that additional bandwidth or CPU usage can arise in different moments, depending on the method. Therefore the choice of the revocation mechanism has to be fitted to the purpose of the system, to minimize the additional costs. The decision can be based on a number of factors, e.g. the probability of a certificate being revoked, the amount of existing certificates, the infrastructure of the CA, the amount of computational power that is available and the size of the window between revocation and the time when no one accepts the certificate any more. Especially the last point can make quite a huge difference: In an example the certificate could be valid for 365 days, therefore if it shall be revoked within a day, with Novomodo up to 365 hashes would have to be calculated. If it shall be revoked within a week, only up to 52 hashes are necessary and a lot of computation can be saved.
To give a brief overview over the different methods, we created two tables that are in the following pages. We used symbols like $\oplus$ (good), $\circledcirc$ (not so good), $\ominus$ (bad) to illustrate the different areas of interest. The second table gives are more detailed look with short explanations.
Security-wise all the methods can be configured in a way that they suit the needs. Therefore the size of the window for an attack can be influenced by the configuration.

concept	validator effort general	validator effort at verification	bandwidth (general)	bandwidth at validation	memory for validator	effort for CA	Internet connection at validation
CRL	⊚	⊖	⊖	⊕[a]	⊖	⊚	No [b]
CRT	⊕	⊚	⊕	⊚	⊕	⊚	Yes [c]
OCSP	⊕	⊚	⊕	⊖	⊕	⊖	Yes
Novomodo	⊕	⊚	⊕	⊚	⊕	⊕	Yes [d]
Short lifetime	⊚ [e]	⊕	⊖	⊕	⊚	⊖	No

Table 4.1: Brief revocation method overview

⊕ good, ⊚ not so good, ⊖ bad

[a] Nothing needs to be fetched, if the CRL is already in the memory

[b] If the CRL has already been downloaded at another time and is being held up to date

[c] The client that wants to be authenticated can send the CRT values, therefore a connection to another server is not always necessary

[d] The client that wants to be authenticated can send the hash value, therefore a connection to another server is not always necessary

[e] It could be more if the client generates a new keypair for every new certificate

concept	validator effort general	validator effort at validation	bandwidth (general)	bandwidth at validation	memory for validator	effort for CA	internet connection at validation
CRL	Keep CRLs up to date	Search in CRL	Keep CRLs up to date	Nothing, if CRL in the memory	The CRLs	The CA has to provide a CRL	No, if CRL in the memory
CRT	Nothing needs to be precomputed	The hashes for the CRT and the root signature	Nothing has to be fetched regularly	Hashes, logarithmic to amount of revoked certificates	Nothing needs to be stored additionally	Provide the hashes	Yes, if the client doesn't it
OCSP	Nothing needs to be precomputed	Signature of the OCSP response	Nothing has to be fetched regularly	Certificate to OCSP server and the response	Nothing needs to be stored additionally	The CA has to handle the requests	Yes
Novomodo	Nothing needs to be precomputed	Only hashes	Nothing has to be fetched regularly	160-bits	Nothing needs to be stored additionally	The CA just has to calculate hashes	Yes, if the client doesn't send it
Short lifetime	New certificates have to be requested quite often [a]	Nothing needs to be calculated additionally	The new certificates have to be send to the CA	Nothing needs to be fetched	Multiple certificates have to be stored	The CA has to sign new certificates quite often	No

Table 4.2: Revocation method overview

[a] Or even new keypairs would have to be created

4.2 Compromise of the private key

When an attacker is able to retrieve the private key of a valid certificate, the foundation of the security of asymmetric encryption breaks down. The attacker is then able to create a signature for any message, which usually means that there is no possibility to distinguish between the attacker and the legitimate owner of the private key. Of course the certificate for this private key has to be revoked then, which will be topic in the section revocation. We will describe the consequences in the following sections.

4.2.1 Perspective car

The attacker can now behave like a car and the manufacturer cannot separate between the two. Therefore it will be quite challenging for the car to renew the certificate, because both of them could request a new one and the manufacturer cannot decide which one should receive a new certificate. A second certificate (e.g. an expired one or better a second valid certificate as a fail safe method) could be helpful in that case. The attacker would have to hack two certificates to produce this dilemma, which is significantly more unlikely.

Another possibility would be that the car receives the information that it has to go to a work shop. At that place is a secure connection to the manufacturer and the car could obtain a new certificate, but this could become expensive, if many cars have to go to service.

4.2.2 Perspective CA

Another side is the loss of a trusted CA. This means that an attacker gains control over the private key of a CA and is therefore able to sign certificates that would be trusted by other cars. We have to divide between two different cases:

If the time of the attack is clear and the reaction is quick, this isn't a huge problem: All cars that had valid certificates before that time will get a new one from a different CA and all requests with certificates that were created after the attack are most likely the attacker.

In the case that the time of the attack isn't clear, all cars that have a certificate of the hacked CA need to receive new certificates, because it is not possible to distinguish between a certificate that has been signed legitimately or by the attacker. In this case all affected cars would probably have to go to a work shop.

Of course this isn't binary, which means that the time could be known roughly, so there would be a specific time zone in which the ownership of the certificates would be unclear and therefore only a few cars would need a secure channel.

4.3 Distribution of certificates

In the most common use of certificates, the encryption of the web (websites), certificates are issued after the party that needs one has proved its identity. This can be a simple check, like adding a special code by the choosing of the CA to the website and therefore proving ownership or even more checks e.g. the address. The CA can then be sure that the party that wants the certificate really owns the website and it will sign the certificate signing request. This method is being used for the initial setup and also in advance before a certificate expires.
In the automotive world, a car cannot prove that it is a car and therefore the certificates have to get to the car in a different manner. The initial setup is quite simple: When the computer is being produced, the manufacturer can add an valid certificate to it, but the certificate cannot be renewed in that way, because no one wants to exchange parts every few years from their car. Therefore the car itself has to communicate with the manufacturer and request a new certificate. It can prove the identity with the old certificate and the CA can then sign the new one.
In case that the certificate has been revoked in the meantime, it can of course not be used as prove of identity and therefore the certificate would have to be renewed by a work shop, which has a secure connection to the manufacturer and is trusted.

5 Benchmark

Manufacturers try to minimize the costs for the car production to maximize their profit and therefore only what is needed will be added. This means that cars do not have as much computational power as a desktop computer, because the computers have to be as small as possible and should be energy efficient. Therefore it is important to take a look on the performance of the cryptographic functions that will be needed for a PKI with cars. To do this we compared two different libraries in C, which provide the cryptographic functions we need. C is the common language that is being used for car-software, because it is on a low abstraction level and doesn't need a huge operating system or a runtime that costs resources. We chose the most common library openssl and a library for embedded systems wolfssl. To simulate the embedded environment, we used a Raspberry Pi 2B which runs on a 900MHz quad-core ARM Cortex-A7 CPU.

5.1 Building the libraries

The Raspberry Pi runs with a Raspbian OS, which is based on Debian and therefore on Linux and UNIX.
To be able to use the openssl library, we just had to install the libssl-dev package, it comes with all the functions we need and was already pre-installed. We wanted to use it as a reference for wolfssl and didn't try to modify it.

```
apt-get install libssl-dev
```

wolfssl on the other hand could be compiled to fit the system perfectly and had to be configured to include all functions that we need.[6]

```
./configure --enable-fasthugemath --enable-keygen --enable-certgen
        --enable-certreq --enable-harden --enable-hkdf --enable-eccencrypt
        --enable-testcert --enable-sp
make check
sudo make install
```

enable-fasthugemath: Enables the use of faster math operations.
enable-keygen: Allows us to generate new keys and not only use existing ones.
enable-certgen: Allows us to generate new certificates.

[6] using the GNU toolchain with GNU make

enable-certreq: Allows us to generate certificate signing requests.
enable-harden: Prevents timing attacks.
enable-hkdf: Allows us to use hash functions.
enable-eccencrypt: Allows us to use elliptic curve cryptography.
enable-testcert: Allows us to decode existing certificates.
enable-sp: Uses single precision math, which makes the calculation faster on the Raspberry Pi.

5.2 Generating data

As we have the libraries now available, we need to collect the data. To get accurate data, we will run every scenario that we want to benchmark for 1000 times and calculate the best, worst and average time. We do this to balance inaccuracy of the measurement and prevent that a single measuring could lead to a incorrect result.
We cannot use the system clock to get accurate time, because it communicates with a timeserver and can therefore make little time-jumps and ruin the data. Instead there are two possible options: Tick counting and monotonic clock. The C program itself always knows the amount of ticks[7] that have passed by since the start of it and therefore we could use this data to calculate the difference between the start and the end of each test and then divide it by the amount of ticks that pass by per second. The other possibility is to use the monotonic clock with clock_gettime. This has nanosecond precision and does not do any time-jumps. Therefore we decided to use the second possibility.
We can then calculate the difference in nanoseconds and seconds and therefore get the amount of microseconds that have passed by.

```
struct timespec startTime, endTime;
clock_gettime(CLOCK_MONOTONIC, &startTime);
//Calculations here
clock_gettime(CLOCK_MONOTONIC, &endTime);
long time = (endTime.tv_nsec-startTime.tv_nsec)/1000
        + (endTime.tv_sec-startTime.tv_sec)*1000000; //micsec
```

5.3 Scenarios

We then used the APIs to benchmark multiple scenarios.

[7] processor clock cycles

5.3.1 ECC-Key generation

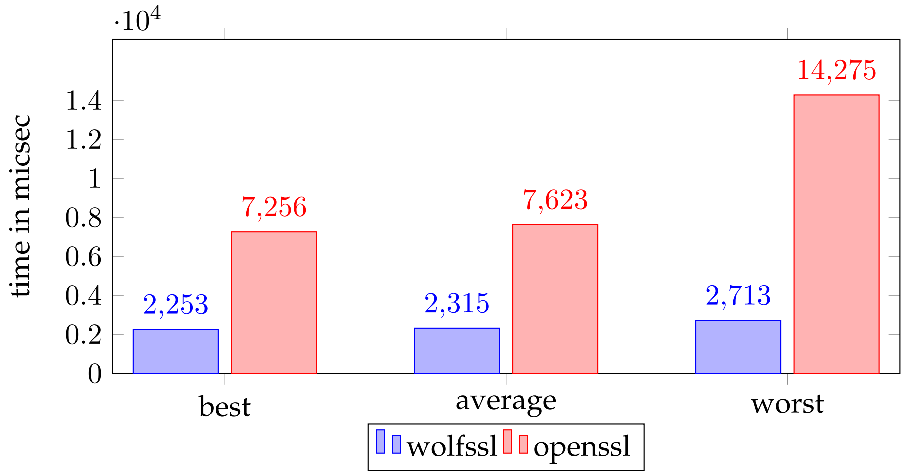
When a car renews its certificate, it should create a new keypair, because if an attacker is trying to guess the key, the attacker would have to begin again.

wolfssl: wolfssl has the data structure ecc_key, which saves the private and public key and the init function allocates memory for it. Additionally we need a RNG (random number generator), which is being needed for the generation of a new key. This has also be initialized to allocate memory and to get some pseudorandom data. We can then generate a key of 32 byte length.

```
ecc_key key;
RNG rng;

wc_InitRng(&rng);
wc_ecc_init(&key);

wc_ecc_make_key(&rng, 32, &key);
```

openssl: In openssl the private EC_KEY and public key EVP_PKEY are separate data structures and we don't need a RNG explicitly. Wolfssl give more responsibility to the developer by expecting a RNG, but openssl integrated the random number generation in the software to lighten the load of the developers. We then use the secp256r1 curve, which has 256 bits (32 bytes) and is the implementation of NIST P-256. It is a common curve that is recommended by the US department NIST and therefore suggest itself as a reference.

```
EVP_PKEY * pkey;
pkey = EVP_PKEY_new();
EC_KEY *key;

key = EC_KEY_new_by_curve_name(NID_secp256r1);

EVP_PKEY_assign_EC_KEY(pkey, key);
```



Wolfssl can generate three keypairs in the time that openssl needs to generate one and the worst case for openssl needs 97% more time than the best case, but it doesn't affect the average case much. Therefore the worst case happens rarely.

5.3.2 ECC certificate generation

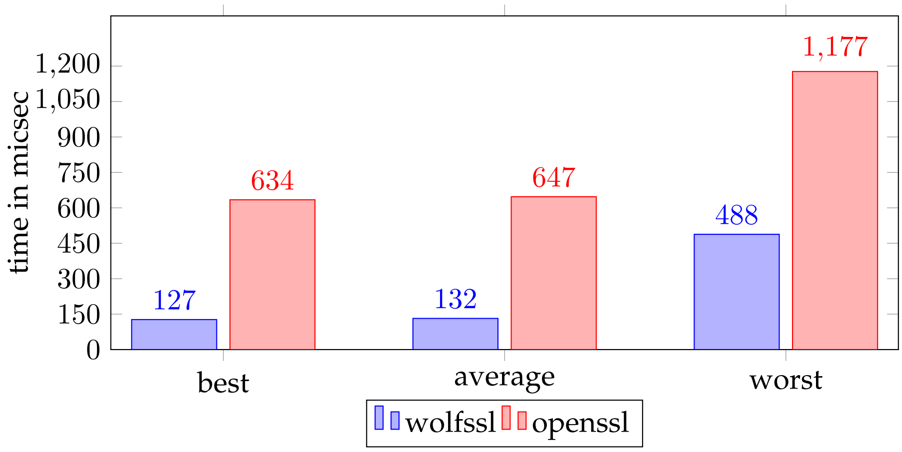
We are benchmarking the certificate generation with SHA256 as Hash and ECDSA as signature algorithm. This means that the library calculates the hash of the certificate and encrypts it with ECDSA with the private key of the signature.
wolfssl: To prepare the benchmark we loaded a certificate into derBuf to use this as a CA certificate. We then create a new certificate, add some data and sign it. As a signature algorithm we use CTC_SHA256wECDSA.
Before the certificate can be created, the issuer buffer has to be set to make clear which entity issued the certificate.
At the end we have an unsigned certificate.

```
Cert newCert;
wc_InitCert(&newCert);

strncpy(newCert.subject.commonName, "A car manufacturer", CTC_NAME_SIZE);
//[...] more X.509 information
newCert.isCA     = 0;
newCert.sigType = CTC_SHA256wECDSA;

wc_SetIssuerBuffer(&newCert, derBuf, derBufSz);
wc_MakeCert(&newCert, certBuf, FOURK_SZ, NULL, &newKey, &rng);
```

openssl: We loaded a CA certificate in caCert before the benchmark starts and then create a new certificate with some data which is then being signed.

We don't need to fill the complete CA certificate into it, the function just need the name to set the issuer correctly.

```
X509_NAME* name = NULL;
X509* x509;
x509 = X509_new();

name = X509_REQ_get_subject_name(x509_req);

ASN1_INTEGER_set(X509_get_serialNumber(x509), 1);
X509_gmtime_adj(X509_get_notBefore(x509), 0);
X509_gmtime_adj(X509_get_notAfter(x509), 31536000L);
X509_set_pubkey(x509, newpKey);

X509_NAME_add_entry_by_txt(name, "CN", MBSTRING_ASC,
        (unsigned char *)"A␣car␣manufacturer", -1, -1, 0);
//[...] more X.509 information

X509_set_issuer_name(x509, X509_get_subject_name(caCert));
```



Wolfssl needs only a fifth of the time to generate a signed certificate compared to wolfssl. Certificates are rarely generated this way in a practical manner (without a certificate signing request), but this benchmark shows that wolfssl can be fast in signing, which other benchmarks will underline.

5.3.3 ECC certificate signing request generation

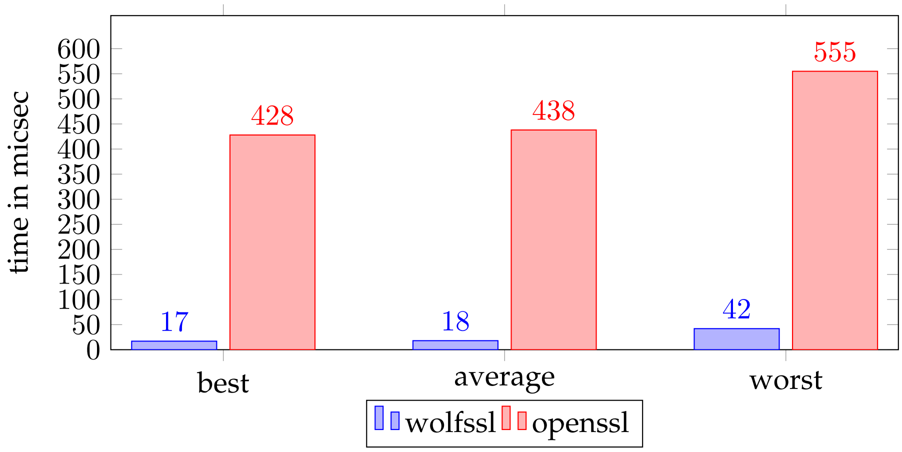
When a car needs a new certificate, it would create a certificate signing request, which is like a certificate, but missing the signature of the CA. It does contain a signature from the car, to protect the integrity of the data. This will then be send to the CA and the CA will send a signed certificate back.

wolfssl: This one is quite similar to the certificate generation, but instead of creating a certificate, we just create a certificate signing request and therefore don't need to set the issuer.

```
Cert newCert;
wc_InitCert(&newCert);

strncpy(newCert.subject.commonName, "A car manufacturer", CTC_NAME_SIZE);
//[...] more X.509 information

ret = wc_MakeCertReq(&newCert, certBuf, FOURK_SZ, NULL, &newKey);
```

openssl: We have to use a different data structure for the certificate signing request, but this can be used quite similar to the one in the certificate generation.

```
X509_REQ* x509 = NULL;
X509_NAME* name = NULL;

x509 = X509_REQ_new();
ret = X509_REQ_set_version(x509, 1);
if (ret != 1){
        goto fail;
}

name = X509_REQ_get_subject_name(x509_req);

ASN1_INTEGER_set(X509_get_serialNumber(x509), 1);
X509_gmtime_adj(X509_get_notBefore(x509), 0);
X509_gmtime_adj(X509_get_notAfter(x509), 31536000L);
X509_set_pubkey(x509, newpKey);

X509_NAME_add_entry_by_txt(name, "CN", MBSTRING_ASC,
        (unsigned char *)"A car manufacturer", -1, -1, 0);
//[...] more X.509 information

ret = X509_REQ_set_pubkey(x509, newpKey);
```



The size of the difference between wolfssl and openssl is surprising here. Openssl needs on average more than 25 times more time than wolfssl (420 micsec difference).

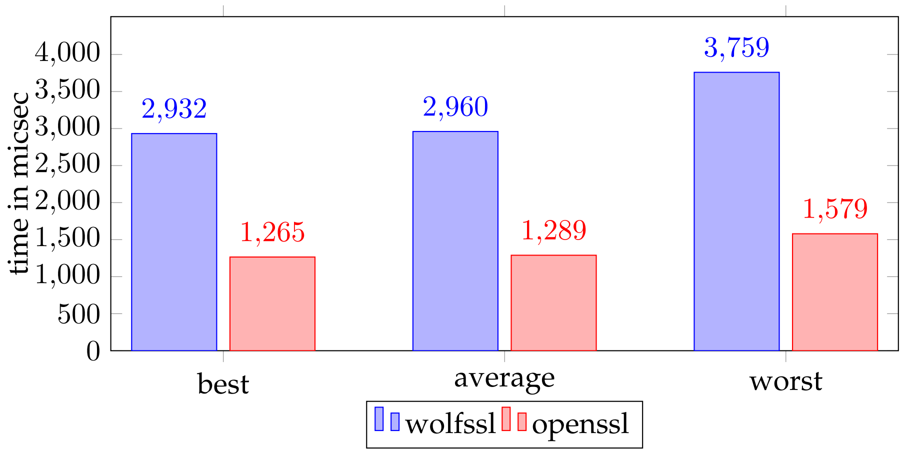
5.3.4 ECC CSR signature/certificate generation

This will be used by the CA when it receives a certificate signing request, which the CA has to check (integrity) and sign it, to create a certificate.
wolfssl: newCert is a CSR which is then being signed by the CA.

```
newCert.sigType = CTC_SHA256wECDSA;
wc_SignCert(newCert.bodySz, newCert.sigType, certBuf, FOURK_SZ, NULL, &caKey, &rng);
```

openssl: x509 is a $X509_REQ$ which is then being signed.

```
X509_REQ_sign(x509, caKey, EVP_sha256());
```



This balances the difference between wolfssl and openssl in the CSR generation. Wolfssl needs 1671 micsec more than openssl and therefore the overall process of CSR generation and signing would be faster with openssl.

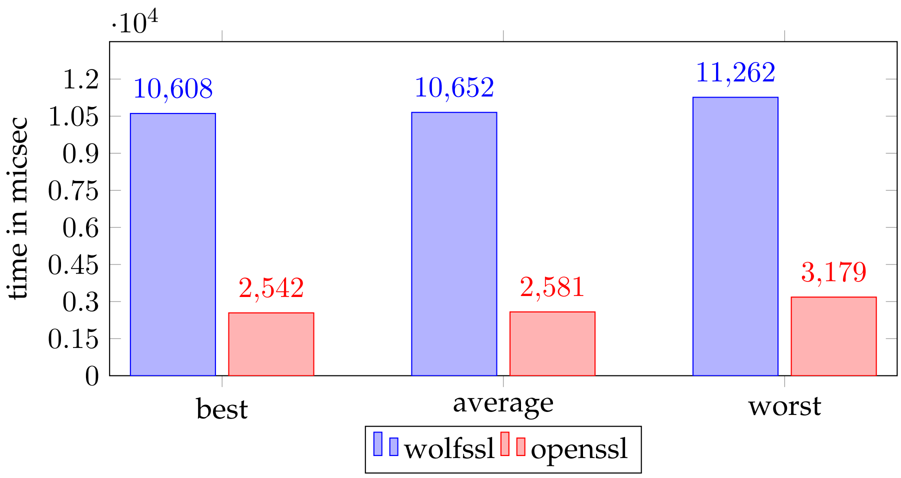
5.3.5 Verify certificate

When a communication participant receives a certificate, the validity and the chain to the root certificate has to be verified.
wolfssl: cm is a $WOLFSSL\_CERT\_MANAGER$ which has the CA certificate already loaded. We then just have to call it and it will verify the certificate and all the certificates in the chain.

```
wolfSSL_CertManagerVerifyBuffer(cm, certBuf, certBufSz, SSL_FILETYPE_ASN1);
```

openssl: ctx is a $X509\_STORE\_CTX$ which has the CA and root already loaded as trusted stack. We then just have to call it and it will verify the certificate and all the certificates in the chain.

```
X509_verify_cert(ctx);
```



Openssl can verify nearly four certificates in the time that wolfssl needs to verify one, which could be useful for a car, because it will receive a lot of data and will need to verify certificates quickly.

5.3.6 Extract ECC public key from certificate

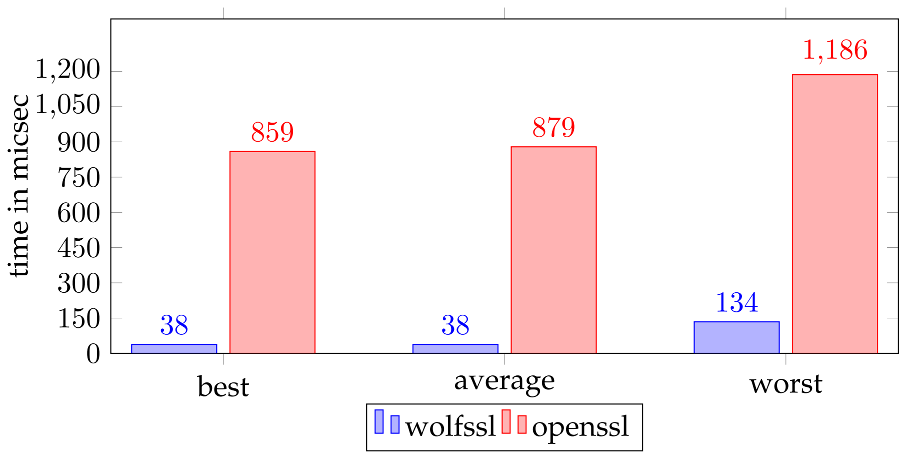
When we receive a certificate we usually want the public key to encrypt data and send it. To minimize the amount of data that is being sent, we can extract the public key from the certificate instead of sending it additionally.
wolfssl: The usual way would be to add the openssl-compatibility layer to wolfssl and then use the API to decode the certificate. We did not want to add this and therefore used the testcert environment to decode the certificate.

```
DecodedCert dcert;
InitDecodedCert(&dcert, derBuf, derBufSz, HEAP_HINT);
```

```

ParseCert(&dcert, CERT_TYPE, NO_VERIFY, 0);

ecc_key pubKey;
wc_ecc_init(&pubKey);

wc_EccPublicKeyDecode(dcert.publicKey, &idx, &pubKey, dcert.pubKeySize);
```

openssl: The certificate that should be used is in the derBuf, we then create the certificate and read the public key from it.

```
EVP_PKEY *pkey = NULL;
BIO *certbio = NULL;
X509 *cert = NULL;

certbio = BIO_new_mem_buf((void*)derBuf, 4096);

cert = PEM_read_bio_X509(certbio, NULL, 0, NULL);

pkey = X509_get_pubkey(cert);
```



As we can see, wolfssl is faster in extracting the public key than openssl. When openssl is chosen as a library, it should be considered to send the public key additionally to the certificate, as a trade-off of bandwidth vs. computational power.

5.3.7 ECC signature of 1024 random bits

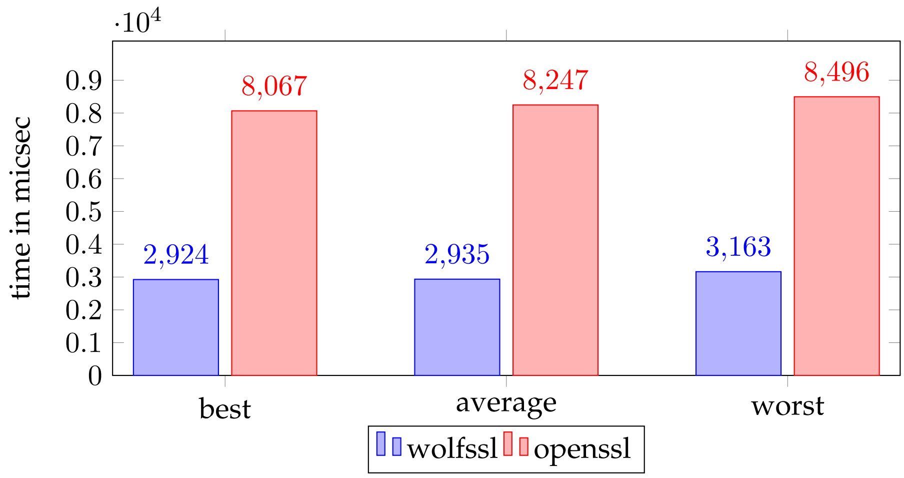
Information does not always have to be encrypted. Data that is being broadcasted to many other cars doesn't need to be hidden, but there has to exist a proof of the sender and protection of the integrity. Therefore the car has to generate a signature for the data that it sends.

wolfssl: We now have a *ecc_key* newKey and random 1024 bits in data. We then create a signature for this data.
The function is a general function and therefore need arguments like the hash type and signature type.

```
unsigned int sigLen = wc_SignatureGetSize(WC_SIGNATURE_TYPE_ECC, &newKey,
                                         sizeof(newKey));
byte* sigBuf = malloc(sigLen);

wc_SignatureGenerate(WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC, data, len,
                                         sigBuf, &sigLen, &newKey, sizeof(newKey), &rng);
```

openssl: newpKey contains our key and we then sign our data of 1024 random bits.
For the different signature types there are different functions and therefore this code is way smaller than the code with wolfssl.

```
ECDSA_SIG* signature = NULL;
signature = ECDSA_do_sign(data, len, newpKey);
```



Wolfssl doesn't even need half the time of openssl, but we can see that it takes a long time to generate a signature anyway.

5.3.8 Verify ECC signature of 1024 random bits

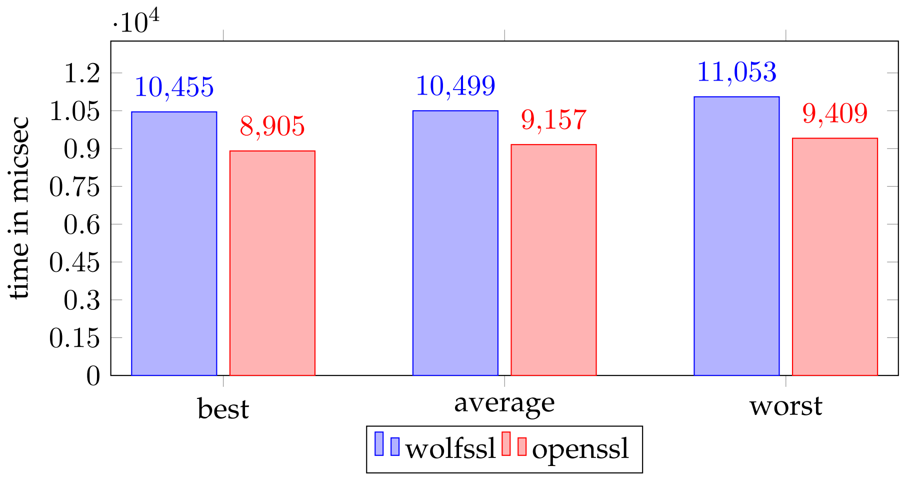
Cars will receive a lot of data from the cars around it. This data will have a signature, to offer evidence that it has been sent by a legitimate car (or traffic light, etc.). The receiver needs to verify the signature, to trust the data.
wolfssl: We generated a signature in advance and we will now check if the signature is correct.

```
wc_SignatureVerify(WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC, data, len,
```

```
                                sigBuf, sigLen, &newKey, sizeof(newKey));
```

openssl: We generated a signature in advance and we will now check if the signature is correct.

```
ECDSA_do_verify(data, len, signature, newpKey)
```



The verification process takes even more time than the generating process, but this time wolfssl and openssl need about the same time.

5.3.9 SHA2-256 hashing of 256 random bits

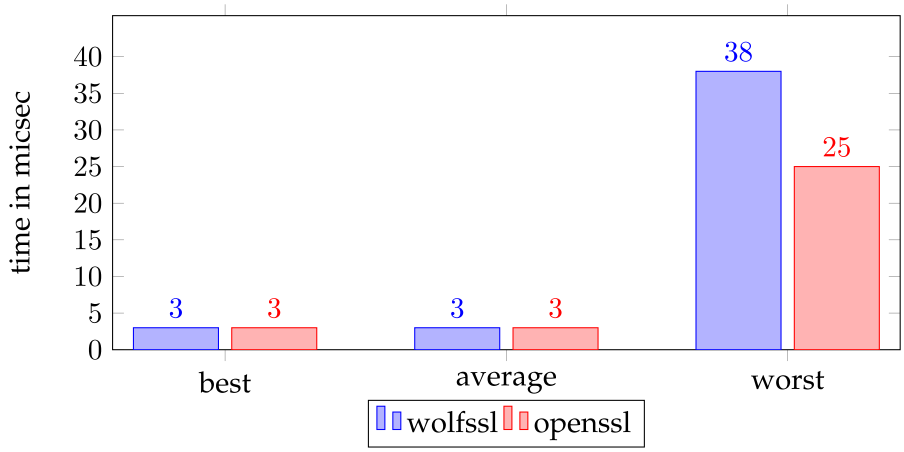
Hashing plays a central role for Novomodo, this means that the car has to hash quite often, to check the validity of a certificate.
wolfssl: We generate 256 Bits using the RNG and save it in data. We then initialize the sha256 and feed it with data with the update method. Final creates the hash and resets the sha object.

```
wc_Sha256 sha;

wc_InitSha256(&sha);
wc_Sha256Update(&sha, data, len);

byte hash[WC_SHA256_DIGEST_SIZE];
wc_Sha256Final(&sha, hash);
```

openssl: We see the similarities with wolfssl here very well, the functions are nearly the same.

```
byte hash[SHA256_DIGEST_LENGTH];
SHA256_CTX sha;

```

```
SHA256_Init(&sha);
SHA256_Update(&sha, data, len);
SHA256_Final(&sha, hash);
```



Hashing is one of the fastest tasks in the benchmark and wolfssl and openssl both need three micsec on average. There is a huge deviation to the worstcase, but it happens infrequently and best and average case are the same.

5.4 Conclusion

Overall we can see that the data for wolfssl and openssl usually is quite congruent: The average case is quite near to the bestcase ($1,638\%$ and $2,252\%$ deviation on average). But the worst case takes up to 252% more time than the average case (SHA2-256 hashing of 256 random bits with wolfssl).
For Novomodo it is quite interesting to see that we can calculate 1000 hashes in the same time that is needed for one signature and even 3000 hashes for one verification of a signature.
An interesting benchmark was the time that is needed to generate a CSR and sign it afterwards. We could see that the CSR generation is much faster with wolfssl, but the signing took so long that openssl was faster overall. In practical terms the CA server has more power than the car and it would be more important to keep the load of the car low. This means that wolfssl would probably be chosen nevertheless.
By comparing benchmarks of signing operations and verification operations, we can see strengths of the liberaries: Wolfssl is always faster in signing, while openssl is faster in verification and tasks that include verification, like the signing of a CSR.
In conclusion both APIs are quite similar in their function signature and it is easy to see which functions of wolfssl have been inspired by openssl (e.g. hashing). Performancewise they can be quite different, which means that wolfssl can be up to 23 times faster than

openssl (average case for ECC certificate signing request generation), but openssl can be up to 3 times faster (average case for verification of a certificate). Therefore it is important to evaluate which tasks will be executed the most in the practical use and decide which library suits the needs in functionality and performance best. If this is unclear in the beginning, wolfssl would be a good choice, it is faster than openssl in 5 cases and only slower in 3 cases. They have about the same speed in hashing.

6 Practical Implementation

As we have taken a look at the challenges and became familiar with the libraries, we will now build an implementation that could be used in practice and realizes the Novomodo concept in a practical manner. For this we will use wolfssl, because it provides a better performance foot print for this use-case and we can compile it for different platforms and adapt it to our needs.
We will create a protocol to let two cars communicate with each other and create a Novomodo server, which will provide the current hash. To give an example of the communication, we will use the protocol to let a car communicate with a software update server, which will reply whether the current version is up to date or not.

6.1 Challenges

First we need to address the challenges that exists with PKI. This means for us that we have to be able to revoke certificates, but the car should not have to compute a lot. We will use Novomodo to prove the validity of the car's certificate and a CRL[8] to prove the validity of the CA certificates. Each car manufacturer will have it's own CA and therefore there won't exist that many CA certificates and a CRL suits our needs. The car will have to check it occasionally, but a revocation of CA certificates is really unlikely and therefore won't create much effort for the car. We do not focus on the CRL and therefore excluded it from our implementation.
Additionally the CA has to save the secret Novomodo values. We will use a sqlite database in this case, as it will simplify the implementation and our focus is by the client and not the server application. Of course a car manufacturer would choose the database more deliberately to account for scalability and performance. Because the database doesn't directly support binary values which are not UTF or ASCII, we used a binary-blob entry for the hash, the random secret and the expiry date (see lines 23-28 in code listing A.9), but we converted the serial number to a hexadecimal string to use it as a database-key.
Lastly we have to integrate the Novomodo hash in a X.509 certificate. Wolfssl doesn't allow to add fields and therefore we will use the e-mail-field for the hash[9], because cars don't have an e-mail and therefore we won't need it (see line 116 in code listing A.11).

[8] Certificate Revocation List
[9] As this is just a demonstrator. In a final product wolfssl can be adapted to change the e-mail-field into the Novomodo-hash-field.

6.2 Communication

The next step is to think about how to secure the communication. Our focus is on authentication, integrity and prevention of replay attacks. Therefore the participants have to prove their identities by using a certificate and we will have to verify it and the provided Novomodo hash. We also have to add information to every message which can be used to detect modifications of the message (signature). We will also use additional information (a salt) to prevent a replay attack, which means that an attacker can not inject packets from a captured older communication into a new one. Lastly we will allow the participants to encrypt their messages and therefore prevent that someone else reads them.

To authenticate themselves the participants will exchange their certificates and the hashes in the beginning. We can extract the public key out of the certificate and we will use it later. We then have to check the validity of the certificate (see lines 42-54 in code listing A.6) and then validate the hash (see lines 60-71 in code listing A.6).

The challenging part was the extraction of the begin date of the certificate and use it to calculate the hash. There is no public wolfssl API to extract the date, so a few internal methods of wolfssl had to be modified (see lines 108-151 in code listing A.8). The next problem was now, that that only the begin date with an internal offset could be extracted. The begin date can than be calculated by adding an offset of 2 and the expiration date can be extracted by adding an offset of 19. In the end we used the public API to gain information about the date and used this with the offset to extract it as a $struct\ tm$ (see lines 71-73 in code listing A.8).

Now we have checked the certificates and have the public key of the other participant and therefore need to exchange salts. We used the wolfssl API for that and did the exchange via plain message (see lines 131-156 in code listing A.7). A man in the middle attacker could submit a "bad" salt, but the sender of the salt would notice that a wrong salt has been used in the reply.

From now on we are able to encrypt the message with the wolfssl API. We just have to make sure that it has the correct padding (length has to be a multiple of 16, see lines 297-310 in code listing A.7). Because the encryption with just the public key is quite expensive, we are using our private key and the public key of the other participant to create a shared secret (see section 2.2.3). This will then be used for symmetric AES-128-CBC encryption.

To secure the message from modifications, we then apply HMAC-SHA256 with the shared secret on the message, concatenated with the salt. Therefore an attacker will not be able to to create the HMAC (because of the secret) and it will be different for every new communication (because of the salt). This means that we can prevent replay attacks and we will notice when the data has been modified.

We will then send the message with the HMAC (see lines 143-144 in code listing A.2). The recipient then just has to check the HMAC and decrypt the message (see lines 160-161 in code listing A.2). [Ous13]

6.3 Architecture

In our example we will need three certificates: One for the root, which acts as a CA, one for the car and another one for the software update server. Therefore we created scripts, that create these certificates, sign them and add the hash to the database (see lines 84-124, 186-211 code listing A.11). We separated repetitive tasks like hashing, extracting the date, writing to database into different files and created a function for each task.
We also created three other runnable files. Firstly the Novomodo server which runs endlessly and listens for new requests for the hash. Additionally the software update server which listens for incoming connections endlessly as well. The third one is the automotive client, which can communicate with either the Novomodo server or the software update server (see A.1 for an example of the console output).

6.4 Conclusion

In the end the API of wolfssl reached its limit in multiple points, but we were able to overcome these drawbacks and extract or put in the information that we needed anyway. The most difficult part was the juggling with pointer (or pointers of pointers) and to use the offset for the methods in the correct way. It can happen quickly that a mistake arises by using pointers, which can then lead to a security problem in the software. These are both topics that someone, who would develop the code further, needs to have in mind. It would also be useful to expand the wolfssl API to allow the addition of more fields to a certificate and to create methods that do not need an offset.

7 Conclusion

7.1 Summary

In this thesis we have taken a look at already existing standards for the car2car and car2authority world, as well as standards in similar environments. We then compared them briefly and highlighted the main ideas. We also stated the reasons why other environments are quite similar and showed the differences.
Furthermore we have taken a look at different challenges that someone who builds a PKI has to be aware of. We had a look at the revocation of certificates in detail and compared several ways which solve the problem. In this manner we noted that for cars the most efficient ways are certificates with a short lifespan or Novomodo.
After that we created a benchmark to compare the crypto libraries wolfssl and openssl in an embedded environment. We therefore used a Raspberry Pi and performed operations that will be needed in the context of a PKI. Therefore we had 9 comparisons, including the hashing for Novomodo. We learned that hashing is faster to a factor of 1000-3000 compared to signing or verifying a signature. This emphasized the advantage of Novomodo compared to certificate revocation lists furthermore.
To proof that Novomodo can work in practice, we implemented a small infrastructure with two communication participants and one Novomodo server. We experienced that it can be quite challenging to adapt a library to special needs and therefore uncommon solutions can be needed.

All in all it became clear that there is not the one perfect solution to implement a Public-Key Infrastructure, rather decisions depend highly on the application and the most common operations that will be used. Therefore in the car background it is important to choose the best fitting library to optimize performance and to avoid certificate revocation lists to be able to communicate with many other participants.

7.2 Discussion and open problems

To continue the development of the practical implementation, the architecture could be expanded. This means, that one or multiple CAs could be integrated in the hierarchy and a server that distributes updated certificates could be created.

The servers could also be expanded to support multithreading and therefore to serve multiple cars at once. This could then be used to benchmark the communication and to examine how many requests per second can be answered.

A problem that we have not looked into is the privacy. To prevent that an attacker can track cars and therefore knows if a car is at home or somewhere else and which route it is currently driving, it will be important to anonymise the certificates. This is a challenge that could be looked into furthermore.

References

[74216] Ieee standard for wireless access in vehicular environments–security services for applications and management messages. *IEEE Std 1609.2-2016 (Revision of IEEE Std 1609.2-2013)*, pages 1–240, March 2016.

[ccc07] car 2 car consortium. Car 2 car communication consortium manifesto. Technical report, 08 2007.

[CSF+08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280, RFC Editor, May 2008. `https://tools.ietf.org/html/rfc5280`.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. on Info. Theory*, IT-22:644–654, 11 1976.

[fSidI17] Bundesamt fuer Sicherheit in der Informationstechnik. Technische richtlinie. *Smart Metering PKI - Public Key Infrastructure fuer Smart Meter Gateways*, 08 2017.

[Gen03] Craig Gentry. Certificate-based encryption and the certificate revocation problem. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 272–293, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Gut] Peter Gutmann. Pki: It's not dead, just resting.

[HMV04] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography (Springer Professional Computing)*. Springer, 2004.

[Kar09] Frank Kargl. Secure vehicle communication baseline security specification. Technical report, European Commission, 04 2009.

[KAS08] Vivek Kapoor, Vivek Sonny Abraham, and Ramesh Singh. Elliptic curve cryptography. *Ubiquity*, 2008(May):7:1–7:8, May 2008.

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. RFC 2104, RFC Editor, February 1997. `https://tools.ietf.org/html/rfc2104`.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[Koc98] Paul C. Kocher. On certificate revocation and validation. In Rafael Hirchfeld, editor, *Financial Cryptography*, pages 172–177, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[MR01] Patrick McDaniel and Aviel Rubin. A response to "can we eliminate certificate revocation lists?". In Yair Frankel, editor, *Financial Cryptography*, pages 245–258, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Ous13] Todd A Ouska. wolfssl btle secure message exchange. Technical report, wolfSSL Inc., 01 2013.

[PGV93] Bart Preneel, René Govaerts, and Joos Vandewalle. Information authentication: Hash functions and digital signatures. In Bart Preneel, René Govaerts, and Joos Vandewalle, editors, *Computer Security and Industrial Cryptography*, pages 87–131, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[PP10a] Christof Paar and Jan Pelzl. *Introduction to Public-Key Cryptography*, pages 149–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[PP10b] Christof Paar and Jan Pelzl. *Key Establishment*, pages 331–357. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[rCS94] D. Eastlake 3rd, S. Crocker, and J. Schiller. Randomness recommendations for security. RFC 1750, RFC Editor, December 1994. `https://tools.ietf.org/html/rfc1750`.

[Vau06] Serge Vaudenay. *A Classical Introduction to Cryptography - Applications for Communications Security*. Springer Science & Business Media, Berlin Heidelberg, 2006.

[Wei11] Andre Weimerskirch. V2x security & privacy: The current state and its future. *Proceedings 18th ITS World Congress*, 10 2011.

[Wei17] Andre Weimerskirch. V2x security and privacy. In *SANS - Automotive Cybersecurity*, 05 2017.

A Practical Implementation

A.1 Output

We startet the novomodo-server first, then the software update server and lastly the automotive client. We are only printing the first 16 bytes of the hash.
Novomodo-Server:

```
Server starting
Waiting for connection...
Receiving a connection...
Sending hash: 271987F1541CF48A423343F7CE5D75444A4EACFF8BC071CA06E9F55C795FCA08

Waiting for connection...
Receiving a connection...
Sending hash: FD619BCCEDA6442CD0945D48D87472717FE1DF2B40ABD0400CDD2A3ACD6577A0

Waiting for connection...
```

Software-Update-Server

```
Current hash: 271987F1541CF48A423343F7CE5D75444A4EACFF8BC071CA06E9F55C795FCA08
Waiting for connection...
Receiving a connection...
Client certificate successfully verified!
Exchanging salts...
Version is up to date: 1

Waiting for connection...
```

Automotive-Client

```
Current hash: FD619BCCEDA6442CD0945D48D87472717FE1DF2B40ABD0400CDD2A3ACD6577A0
    0: Check for software update
    1: Get current Novomodo hash
    Please choose a number: 0

Check for software update
0: Send current version
1: Send older version
Please choose a number: 0

```

```
Creating a connection...
Server certificate successfully verified!
Exchanging salts...
Version ok

    0: Check for software update
    1: Get current Novomodo hash
    Please choose a number:
```

A.2 automotive-client.c

```c
#include <stdio.h>
#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/sha512.h>

#include "connection-worker.h"
#include "hasher.h"
#include "certificate-manager.h"

#define HEAP_HINT NULL
#define KEY_SZ 2048
#define FOURK_SZ 4096

int speakToSoftwareUpdateServer(byte* derBuf, char version);

byte* hash;

/*
 Can fetch Novomodo hash and check for software update
 */
int main(int argc, char const *argv[]) {
    hash = NULL;

    byte* derBuf = NULL;
    int derBufSz = loadAutomotiveCert(&derBuf);

    byte* rootBuf = NULL;
```

```
    int rootBufSz = loadRootCert(&rootBuf);

    hash = malloc(32);
    fetchCurrentHash(0, "127.0.0.1", &hash, derBuf, derBufSz, rootBuf, rootBufSz);

    while (1) {
        char choice, temp;

        printf("␣␣␣␣0:␣Check␣for␣software␣update\n");
        printf("␣␣␣␣1:␣Get␣current␣Novomodo␣hash\n");
        printf("␣␣␣␣Please␣choose␣a␣number:␣");

        scanf("%c%c", &choice, &temp);

        printf("\n");

        if (choice == ’0’) {
            char version;

            printf("Check␣for␣software␣update\n");
            printf("0:␣Send␣current␣version\n");
            printf("1:␣Send␣older␣version\n");
            printf("Please␣choose␣a␣number:␣");
            scanf("%c%c", &version, &temp);
            printf("\n");

            if (version == ’0’) {
                speakToSoftwareUpdateServer(derBuf, ’1’);
            } else if (version == ’1’) {
                speakToSoftwareUpdateServer(derBuf, ’0’);
            } else {
                printf("Invalid!␣\n");
            }
        } else if (choice == ’1’) {
            printf("Get␣current␣Novomodo␣hash\n");
            fetchCurrentHash(0, "127.0.0.1", &hash, derBuf, derBufSz,
                             rootBuf, rootBufSz);
        } else {
            printf("Invalid!␣\n");
        }

        printf("\n");
    }

    if (hash != NULL) free(hash);
    if (derBuf != NULL) free(derBuf);
```

```
    if (rootBuf != NULL) free(rootBuf);
    return 0;
}

/*
 Establishes a secure connection with the SoftWareUpdateServer and checks
 wether software is up to date
 */
int speakToSoftwareUpdateServer(byte* derBuf, char version) {
        int ret;
        WC_RNG rng;
        ecEncCtx* cliCtx = NULL;
        const byte* mySalt;
        byte peerSalt[EXCHANGE_SALT_SZ];
        //byte peerSalt[EXCHANGE_SALT_SZ];
        byte plain[16];
    byte buffer[sizeof(plain) + 32]; //Adding digest size
    word32 bufferSz = sizeof(buffer);;
        word32 plainSz;
        ecc_key myKey, peerKey;
    int sock = 0;
    byte* rootBuf = NULL;
        byte* peerBuf = malloc(FOURK_SZ);

        wolfSSL_Init();

        /* make my session key */
        ret =  wc_ecc_init(&myKey);
        ret |= wc_ecc_init(&peerKey);
        if (ret != 0) {
                printf("wc_ecc_init␣failed!\n");
                goto cleanup;
        }

        ret = wc_InitRng(&rng);
        if (ret != 0) {
                printf("wc_InitRng␣failed!␣%d\n", ret);
                goto cleanup;
        }

        //Load root certificate
        int rootBufSz = loadRootCert(&rootBuf);

        //Load my key
        loadAutomotiveKey(&myKey);

```

```
    printf("Creating␣a␣connection...\n");

        //Establish connection
        ret = openConnectionAsClient(&sock, "127.0.0.1", rng, rootBuf, rootBufSz,
                                     derBuf, peerBuf, &peerKey, hash, &cliCtx);
        if(ret != 1) goto cleanup;

    //Exchange salts
    ret = clientSideSaltExchange(&mySalt, peerSalt, sock, cliCtx);
    if (ret != 1) goto cleanup;

    /* get message to send */
    plainSz = sizeof(plain);
    strcpy((char*)plain, &version);    //current version is 1
    plainSz = strlen((char*)plain);
    msg_pad(plain, &plainSz);

    /* Encrypt message */
    ret = wc_ecc_encrypt(&myKey, &peerKey, plain, sizeof(plain), buffer,
                         &bufferSz, cliCtx);
    if (ret != 0) {
        printf("wc_ecc_encrypt␣failed␣%d!\n", ret);
        goto cleanup;
    }

    /* Send message */
    send(sock, buffer, bufferSz, 0);

    /* Get message */
    bufferSz = sizeof(buffer);
    ret = read(sock, buffer, bufferSz);

    /* Decrypt message */
    bufferSz = ret;
    plainSz = sizeof(plain);
    ret = wc_ecc_decrypt(&myKey, &peerKey, buffer, bufferSz,
                         plain, &plainSz, cliCtx);
    if (ret != 0) {
        printf("wc_ecc_decrypt␣failed␣%d!\n", ret);
        goto cleanup;
    }

    if(plain[0] == '1') {
        printf("Version␣ok\n");
    } else {
        printf("Update␣neccessary\n");
```

```
    }

    /* reset context (reset my salt) */
    ret = wc_ecc_ctx_reset(cliCtx, &rng);
    if (ret != 0) {
        printf("wc_ecc_ctx_reset␣failed␣%d\n", ret);
        goto cleanup;
    }

cleanup:
    if (peerBuf != NULL) free(peerBuf);
    if (peerBuf != NULL) free(rootBuf);

    wc_ecc_free(&myKey);
    wc_ecc_free(&peerKey);
    wc_FreeRng(&rng);

        wolfSSL_Cleanup();
        return ret;
}
```

A.3 novmodo-server.c

```
#include <stdio.h>
#include <sqlite3.h>

#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/sha512.h>

#include "connection-worker.h"
#include "sqlite-worker.h"
#include "hasher.h"

#define HEAP_HINT NULL
#define KEY_SZ 2048
#define FOURK_SZ 4096
#define PORT 8080
```

```

/*
 Answers requests for the current novomodo hash of a certificate
 */
int main(int argc, char const *argv[]) {
     printf("Server␣starting\n");
    int server_fd, new_socket, ret;
    WC_RNG rng;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    byte* peerBuf = malloc(FOURK_SZ);
    byte* hash = malloc(32);

    wolfSSL_Init();

    ret = wc_InitRng(&rng);
    if (ret != 0) {
        printf("wc_InitRng␣failed!␣%d\n", ret);
        return -1;
    }

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket␣failed");
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port 8080
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR /*| SO_REUSEPORT*/,
                   &opt, sizeof(opt)))
    {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons( PORT );

    // Forcefully attaching socket to the port 8080
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))<0) {
        perror("bind␣failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 3) < 0) {
        perror("listen");
```

```
        exit(EXIT_FAILURE);
    }

    while (1) {
        printf("Waiting␣for␣connection...\n");

        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                                 (socklen_t*)&addrlen))<0) {
            perror("accept");
            exit(EXIT_FAILURE);
        }

        printf("Receiving␣a␣connection...\n");

        //Receive certificate
        ret = read(new_socket, peerBuf, FOURK_SZ);

        //Create structure for certificate
        DecodedCert dcert;
        InitDecodedCert(&dcert, peerBuf, FOURK_SZ, HEAP_HINT);

        //Decode Certificate from the previously set buffer
        ret = ParseCert(&dcert, CERT_TYPE, NO_VERIFY, 0);
        if (ret != 0) return 0;

        sqlite3 *db;
        openDatabase(&db);
        ret = getCurrentHash(db, &dcert, &hash);
        closeDatabase(db);

        printf("Sending␣hash:␣");
        printByteAsHexa(hash);

        /* Send hash */
        send(new_socket, hash, 32, 0);

        printf("\n");

        FreeDecodedCert(&dcert);
    }

    return 0;
}
```

A.4 software-update-server.c

```
// Server side C/C++ program to demonstrate Socket programming
#include <stdio.h>
#include <sqlite3.h>

#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/sha512.h>

#include <unistd.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>

#include "connection-worker.h"
#include "certificate-manager.h"

#define HEAP_HINT NULL
#define FOURK_SZ 4096
#define PORT 8081

/*
 Answers the request if the version is still up to date
 */
int main(int argc, char const *argv[]) {
        int server_fd, new_socket, ret;
        WC_RNG rng;
        ecEncCtx* srvCtx = NULL;
        const byte* mySalt;
        byte peerSalt[EXCHANGE_SALT_SZ];
        word32 bufferSz;
        word32 plainSz;
        ecc_key myKey, peerKey;
        struct sockaddr_in address;
        int opt = 1;
        int addrlen = sizeof(address);
        byte* derBuf = malloc(FOURK_SZ);
        byte* peerBuf = malloc(FOURK_SZ);
```

```
        byte* hash = malloc(32);

        wolfSSL_Init();

        /* make my session key */
        ret =  wc_ecc_init(&myKey);
        ret |= wc_ecc_init(&peerKey);
        if (ret != 0) {
                printf("wc_ecc_init␣failed!\n");
                return -1;
        }

        ret = wc_InitRng(&rng);
        if (ret != 0) {
                printf("wc_InitRng␣failed!␣%d\n", ret);
                return -1;
        }

        //Load root certificate
    byte* rootBuf;
        int rootBufSz = loadRootCert(&rootBuf);

        //Load my key
        loadSoftwareUpdateKey(&myKey);

        //Load my certificate
        int derBufSz = loadSoftwareUpdateCert(&derBuf);

    //Get Novomodo Hash
    fetchCurrentHash(0, "127.0.0.1", &hash, derBuf, derBufSz, rootBuf, rootBufSz);

        if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
        {
                perror("socket␣failed");
                exit(EXIT_FAILURE);
        }

        // Forcefully attaching socket to the port 8081
        if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR /*| SO_REUSEPORT*/,

        {
                perror("setsockopt");
                exit(EXIT_FAILURE);
        }
        address.sin_family = AF_INET;
        address.sin_addr.s_addr = INADDR_ANY;
```

```
        address.sin_port = htons( PORT );

        // Forcefully attaching socket to the port 8081
        if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))<0) {
                perror("bind␣failed");
                exit(EXIT_FAILURE);
        }
        if (listen(server_fd, 3) < 0) {
                perror("listen");
                exit(EXIT_FAILURE);
        }

        while (1) {
        printf("Waiting␣for␣connection...\n");

        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                                 (socklen_t*)&addrlen))<0) {
            perror("accept");
            exit(EXIT_FAILURE);
        }

        printf("Receiving␣a␣connection...\n");

        srvCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);
        if (srvCtx == NULL) {
            printf("wc_ecc_ctx_new␣failed!\n");
            return -1;
        }

        ret = acceptConnectionAsServer(new_socket, rootBuf, rootBufSz,
                                       derBuf, peerBuf, &peerKey, hash);
        if (ret != 1) return -1;

                ret = serverSideSaltExchange(&mySalt, peerSalt, new_socket, srvCtx);
                if (ret != 1) return -1;

                /* Get message */
                byte* buffer = malloc(FOURK_SZ);
        bufferSz = read(new_socket, buffer, FOURK_SZ);

                /* Decrypt message */
        byte plain[bufferSz];
                plainSz = sizeof(plain);

                ret = wc_ecc_decrypt(&myKey, &peerKey, buffer, bufferSz, plain,
                             &plainSz, srvCtx);
```

```
                if (ret != 0) {
                        printf("wc_ecc_decrypt␣failed␣%d!\n", ret);
                        return -1;
                }

                if (plain[0] == '1') {
                        //Version is 1 = ok
            printf("Version␣is␣up␣to␣date:␣%c\n", plain[0]);
                        strcpy((char*)plain, "1");
                } else {
                        //Version is not ok
            printf("Version␣is␣not␣up␣to␣date:␣%c\n", plain[0]);
                        strcpy((char*)plain, "0");
                }

        plainSz = strlen((char*)plain);
        msg_pad(plain, &plainSz);

                /* Encrypt message */
                ret = wc_ecc_encrypt(&myKey, &peerKey, plain, plainSz, buffer,
                             &bufferSz, srvCtx);
                if (ret != 0) {
                        printf("wc_ecc_encrypt␣failed␣%d!\n", ret);
                        return -1;
                }

                /* Send message */
                send(new_socket, buffer, bufferSz, 0);

                /* reset context (reset my salt) */
                ret = wc_ecc_ctx_reset(srvCtx, &rng);
                if (ret != 0) {
                        printf("wc_ecc_ctx_reset␣failed␣%d\n", ret);
                        return -1;
                }

        if (buffer != NULL) free(buffer);

        printf("\n");
        }

    return 0;
}
```

A.5 certificate-manager.c

```c
#include <stdio.h>
#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/sha512.h>

#define HEAP_HINT NULL
#define FOURK_SZ 4096

/*
 Loads a certificate (.der)
 */
int loadCert(byte** derBuf, char certToUse[]) {
    FILE* file;
    *derBuf = (byte*) XMALLOC(FOURK_SZ, HEAP_HINT,
                              DYNAMIC_TYPE_TMP_BUFFER);
    if (*derBuf == NULL) return -1;

    XMEMSET(*derBuf, 0, FOURK_SZ);

    file = fopen(certToUse, "rb");
    if (!file) {
        printf("failed to find file: %s\n", certToUse);
        return -1;
    }

    int size = fread(*derBuf, 1, FOURK_SZ, file);

    fclose(file);
    return size;
}

/*
 Loads root certificate
 */
int loadRootCert(byte** rootBuf) {
    return loadCert(rootBuf, "./certs/root-cert.der");
}

```

```c
/*
 Loads car certificate
 */
int loadAutomotiveCert(byte** derBuf) {
    return loadCert(derBuf, "./certs/automotive-cert.der");
}

/*
 Loads software update server certificate
 */
int loadSoftwareUpdateCert(byte** derBuf) {
    return loadCert(derBuf, "./certs/su-cert.der");
}

/*
 Loads a ecc_key
 */
int loadKey(ecc_key* myKey, char keyFile[]) {
        int ret = 1;
    word32 idx = 0;
        FILE* file;
        byte* keyBuf = (byte*) XMALLOC(FOURK_SZ, HEAP_HINT,
                                   DYNAMIC_TYPE_TMP_BUFFER);
    if (keyBuf == NULL) return -1;

        file = fopen(keyFile, "rb");
        if (!file) {
                printf("failed to open file: %s\n", keyFile);
                return -1;
        }

        int keyBufSz = fread(keyBuf, 1, FOURK_SZ, file);
        if (keyBufSz <= 0) {
                printf("Failed to read caKey from file\n");
                return ret;
        }

        fclose(file);

        ret = wc_EccPrivateKeyDecode(keyBuf, &idx, myKey, (word32)keyBufSz);
    if (ret != 0) {
        printf("wc_EccPrivateKeyDecode failed %i\n", ret);
        return ret;
    }

    ret = 1;
```

```

        return ret;
}

/*
 Loads root key
 */
int loadRootKey(ecc_key* myKey) {
    return loadKey(myKey, "./certs/root-key.der");
}

/*
 Loads car key
 */
int loadAutomotiveKey(ecc_key* myKey) {
    return loadKey(myKey, "./certs/automotive-key.der");
}

/*
 Loads software update server key
 */
int loadSoftwareUpdateKey(ecc_key* myKey) {
    return loadKey(myKey, "./certs/su-key.der");
}
```

A.6 certificate-validity-check.c

```
#include <stdio.h>
#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/sha512.h>

#include "hasher.h"

#define HEAP_HINT NULL
#define FOURK_SZ 4096

/*
```

```
 Checks certificate for validity and checks the novomodo hash
 */
int checkCertificate(byte* certBuf, int certBufSz, byte* rootBuf,
                     int rootBufSz, byte* hash) {
        int ret = 0;
    ecc_key pubKey;
    byte* hashToCompare = NULL;
    WOLFSSL_CERT_MANAGER* cm = NULL;

        DecodedCert cert;
        InitDecodedCert(&cert, certBuf, certBufSz, HEAP_HINT);

        ret = ParseCert(&cert, CERT_TYPE, NO_VERIFY, 0) + 1;
        if (ret != 1) goto clearReturn;

        wc_ecc_init(&pubKey);

    word32 idx = 0;

        ret = wc_EccPublicKeyDecode(cert.publicKey, &idx, &pubKey,
                                    cert.pubKeySize) + 1;
    if (ret != 1) goto clearReturn;

        //Verify certificate chain
        cm = wolfSSL_CertManagerNew();
        ret = 0;
        if (cm == NULL) goto clearReturn;

        ret = wolfSSL_CertManagerLoadCABuffer(cm, rootBuf, rootBufSz,
                                              SSL_FILETYPE_ASN1);
    if (ret != SSL_SUCCESS) {
        printf("Errorcode␣1␣%i\n", ret);
        goto clearReturn;
    }

        ret = wolfSSL_CertManagerVerifyBuffer(cm, certBuf, certBufSz,
                                              SSL_FILETYPE_ASN1);
    if (ret != SSL_SUCCESS) {
        printf("Errorcode␣2␣%i\n", ret);
        goto clearReturn;
    }

        byte* finalHash = (byte*) cert.subjectEmail;

    //How many times have to be added on the hash
    int times = calculateVerifyTimes(cert);
```

```

        hashToCompare = malloc(32);

        memcpy(hashToCompare, hash, 32);

        hashFunc(hash, hashToCompare, times);

        ret = memcmp(hashToCompare, finalHash, 32);
    if (ret != 0) {
        printf("Oh␣no...\n");
        goto clearReturn;
    }
    ret = 1;

clearReturn:
        FreeDecodedCert(&cert);
    wc_ecc_free(&pubKey);
    wolfSSL_CertManagerFree(cm);
    if (hashToCompare != NULL) free(hashToCompare);
        return ret;
}
```

A.7 connection-worker.c

```
//  C/C++ program to demonstrate Socket programming
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <math.h>

#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/sha512.h>

#include "certificate-validity-check.h"
```

```
#include "hasher.h"

#define HEAP_HINT NULL
#define FOURK_SZ 4096
#define BLOCK_SIZE 16
#define PORTNOVOMODO 8080
#define PORTSU 8081

/*
 Creates a secure connection to Server with Port 8081
 int sock - Socket
 char* address - Adress, e.g. "127.0.0.1"
 WC_RNG rng - has to be initialised
 byte* derBuf - own certificate
 byte* peerBuf - peer certificate
 will be the peer key afterwards
 ecEncCtx* cliCtx - can be null, will be initialized
 */
int openConnectionAsClient(int* sock, char* address, WC_RNG rng,
                           byte* rootBuf, int rootBufSz, byte* derBuf,
                           byte* peerBuf, ecc_key* peerKey, byte* hash,
                           ecEncCtx** cliCtx) {
    int ret;
    struct sockaddr_in serv_addr;
    byte* peerHash = malloc(32);

    if ((*sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n␣Socket␣creation␣error␣\n");
        ret = -1;
        goto cleanup;
    }

    memset(&serv_addr, ’0’, sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORTSU);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, address, &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid␣address/␣Address␣not␣supported␣\n");
        ret = -1;
        goto cleanup;
    }

```

```
    if (connect(*sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("\nConnection␣Failed␣\n");
        ret = -1;
        goto cleanup;
    }

    *cliCtx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
    if (*cliCtx == NULL) {
        printf("wc_ecc_ctx_new␣failed!\n");
        ret = -1;
        goto cleanup;
    }

    /* exchange public keys */
    /* send my public key */
    send(*sock, derBuf, FOURK_SZ, 0);

    //SEND NOVOMODO
    send(*sock, hash, 32, 0);

    /* Get peer key */
    //Read certificate
    int peerBufSz = read(*sock, peerBuf, FOURK_SZ);
    //Read Novomodo Hash
    ret = read(*sock, peerHash, 32);

    //Create structure for certificate
    word32 idx = 0;
    DecodedCert dcert;
    InitDecodedCert(&dcert, peerBuf, FOURK_SZ, HEAP_HINT);

    //Decode Certificate from the previously set buffer
    ret = ParseCert(&dcert, CERT_TYPE, NO_VERIFY, 0);
    if (ret != 0) {
        printf("ParseCert␣failed␣%i\n", ret);
        goto cleanup;
    }

    //Decode the Public Key from certificate
    ret = wc_EccPublicKeyDecode(dcert.publicKey, &idx, peerKey, dcert.pubKeySize);
    if (ret != 0) {
        printf("EccPublicKeyDecode␣failed␣%i\n", ret);
        goto cleanup;
    }

```

```c
    ret = checkCertificate(peerBuf, peerBufSz, rootBuf, rootBufSz, peerHash);
    if (ret != 1) {
        printf("Server␣certificate␣verification␣failed␣%i\n", ret);
        goto cleanup;
    }

    printf("Server␣certificate␣successfully␣verified!\n");

    ret = 1;
cleanup:
    if (peerHash != NULL) free(peerHash);
    FreeDecodedCert(&dcert);
    return ret;
}

/*
 Exchanges salts with a server to secure connection
 */
int clientSideSaltExchange(const byte** mySalt, byte* peerSalt, int sock,
                           ecEncCtx* cliCtx) {
    printf("Exchanging␣salts...\n");

    int ret;
    /* get my salt */
    *mySalt = wc_ecc_ctx_get_own_salt(cliCtx);
    if (*mySalt == NULL) {
        printf("wc_ecc_ctx_get_own_salt␣failed!\n");
        return -1;
    }

    /* Send my salt */
    send(sock, *mySalt, EXCHANGE_SALT_SZ, 0);

    /* Get peer salt */
    read(sock, peerSalt, EXCHANGE_SALT_SZ);

    ret = wc_ecc_ctx_set_peer_salt(cliCtx, peerSalt);
    if (ret != 0) {
        printf("wc_ecc_ctx_set_peer_salt␣failed␣%d\n", ret);
        return 0;
    }

    return 1;
}

/*
```

```
 Establishes a secure connection with a client
 */
int acceptConnectionAsServer(int new_socket, byte* rootBuf, int rootBufSz,
                             byte* derBuf, byte* peerBuf, ecc_key* peerKey,
                             byte* hash) {
    int ret;
    byte* peerHash = malloc(32);

    /* exchange public keys */
    /* Get peer certificate & key */
    //Read certificate
    int peerBufSz = read(new_socket, peerBuf, FOURK_SZ);

    //Read Novomodo Hash
    ret = read(new_socket, peerHash, 32);

    //Create structure for certificate
    word32 idx = 0;
    DecodedCert dcert;
    InitDecodedCert(&dcert, peerBuf, FOURK_SZ, HEAP_HINT);

    //Decode Certificate from the previously set buffer
    ret = ParseCert(&dcert, CERT_TYPE, NO_VERIFY, 0);
    if (ret != 0) {
        printf("ParseCert␣failed␣%i\n", ret);
        goto cleanup;
    }

    //Decode the Public Key from certificate
    ret = wc_EccPublicKeyDecode(dcert.publicKey, &idx, peerKey, dcert.pubKeySize);
    if (ret != 0) {
        printf("EccPublicKeyDecode␣failed␣%i\n", ret);
        goto cleanup;
    }

    ret = checkCertificate(peerBuf, peerBufSz, rootBuf, rootBufSz, peerHash);
    if (ret != 1) {
        printf("Client␣certificate␣verification␣failed␣%i\n", ret);
        goto cleanup;
    }

    printf("Client␣certificate␣successfully␣verified!\n");

    /* send my public key & certificate */
    send(new_socket, derBuf, FOURK_SZ, 0);

```

```c
    //send novomodo
    send(new_socket, hash, 32, 0);

    ret = 1;
cleanup:
    if (peerHash != NULL) free(peerHash);
    FreeDecodedCert(&dcert);
    return ret;
}

/*
 Exchanges salts with a client to secure connection
 */
int serverSideSaltExchange(const byte** mySalt, byte* peerSalt,
                           int new_socket, ecEncCtx* srvCtx) {
    printf("Exchanging salts...\n");

    int ret;
    *mySalt = wc_ecc_ctx_get_own_salt(srvCtx);
    if (*mySalt == NULL) {
        printf("wc_ecc_ctx_get_own_salt failed!\n");
        return -1;
    }

    /* Get peer salt */
    ret = read(new_socket, peerSalt, EXCHANGE_SALT_SZ);

    /* Send my salt */
    /* You must send mySalt before set_peer_salt, because buffer changes */
    send(new_socket, *mySalt, EXCHANGE_SALT_SZ, 0);

    ret = wc_ecc_ctx_set_peer_salt(srvCtx, peerSalt);
    if (ret != 0) {
        printf("wc_ecc_ctx_set_peer_salt failed %d\n", ret);
        return 0;
    }

    return 1;
}

/*
 Creates connection with Novomodoserver as Client to Port 8080
    to receive current Hash
 */
int fetchCurrentHash(int sock, char* address, byte** hash, byte* derBuf,
                     int derBufSz, byte* rootBuf, int rootBufSz) {
```

```c
    struct sockaddr_in serv_addr;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n␣Socket␣creation␣error␣\n");
        return -1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORTNOVOMODO);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, address, &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid␣address/␣Address␣not␣supported␣\n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("\nConnection␣Failed␣\n");
        return -1;
    }

    //send certificate
    send(sock, derBuf, FOURK_SZ, 0);

    //receive hash
    read(sock, *hash, 32);

    printf("Current␣hash:␣");
    printByteAsHexa(*hash);

    if (checkCertificate(derBuf, derBufSz, rootBuf, rootBufSz, *hash) != 1) {
        printf("Hash␣or␣certificate␣invalid!\n");
        return -1;
    }

    return 0;
}

/*
 Padds a message to make the length as a multiple of
    the block size (16)
```

```
 */
void msg_pad(byte* buf, word32* len) {
    word32 newLen = *len;
    word32 odd = (newLen % BLOCK_SIZE);

    if (odd != 0) {
        word32 addLen = (BLOCK_SIZE - odd);
        newLen += addLen;

        memset(&buf[*len], 0, addLen);
    }

    *len = newLen;
    return;
}

/*
 Writes a number to byte array
 */
byte* toArray(int number) {
    int n = log10(number) + 1;
    int i;
    byte* numberArray = calloc(n, sizeof(char));

    for (i = 0; i < n; ++i, number /= 10) {
        numberArray[i] = number % 10;
    }

    return numberArray;
}

/*
 Reverts a byte array to a number
    handles padding as well
 */
int revertToInt(byte* array) {
    int arraySz = sizeof(array);
    int number = 0;
    int padding = 1;

    for (int i = arraySz - 1; i >= 0; i--) {
        int toAdd = array[i];

        if (padding) {
            if (!toAdd) continue;
            padding = 0;
```

```
        }

        for (int j = i; j > 0; j--) {
            toAdd *= 10;
        }

        number += toAdd;
    }

    return number;
}
```

A.8 hasher.c

```
#include <stdio.h>
#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/sha512.h>

#define HEAP_HINT NULL
#define FOURK_SZ 4096

/*
 Prints first 32 hexadecimal numbers of a byte buffer
 */
void printByteAsHexa(byte* buf) {
    for (int i = 0; i < 32; i++)
    {
        printf("%02X", buf[i]);
    }
    printf("\n");
}

/*
 Hashes data for times times
 data has to have the length WC_SHA256_DIGEST_SIZE (32)
 */
int hashFunc(byte* data, byte* hash, int times) {
```

```
        int ret = 0;

        memcpy(hash, data, 32);

        for (int i = 0; i < times; i++) {
                //Create new sha2-256
                wc_Sha256 sha;

                //Init sha2-256
                ret = wc_InitSha256(&sha);
                if (ret != 0) goto cleanup;

                //Begin hashing
                ret = wc_Sha256Update(&sha, hash, WC_SHA256_DIGEST_SIZE);
                if (ret != 0) goto cleanup;

                //Get hash
                ret = wc_Sha256Final(&sha, hash);
        if (ret != 0) goto cleanup;

    cleanup:
        wc_Sha256Free(&sha);
        if (ret != 0) return ret;
        }

        return ret;
}


/*
 Calculates for the CA how many times the random has to be hashed
 */
int calculateHashTimes(DecodedCert* cert) {
        //Calculate the weeks the certificate has to be used in the future
        int length;
    const byte *datePtr = NULL;
    byte format;

    wc_GetDateInfo(cert->source, cert->maxIdx, &datePtr, &format, &length);

    struct tm before;
    int idx = 19;
        ExtractDate(cert->beforeDate, format, &before, &idx);

        int days = (int) difftime(mktime(&before), time(NULL)) / 60 / 60 / 24;

```

```
    days += 1;

        //How many times have to be added on the hash
        return (days + 6) / 7;
}

/*
 Calculates for communication participants how many times
    have to be added on a hash
 */
int calculateVerifyTimes(DecodedCert cert) {
        //Calculate the weeks the certificate has alreade been used
        int length;
    const byte *datePtr = NULL;
    byte   format;

    wc_GetDateInfo(cert.source, cert.maxIdx, &datePtr, &format, &length);

        struct tm after;
    int idx = 2;
        ExtractDate(cert.afterDate, format, &after, &idx);

        int days = difftime(time(NULL), mktime(&after)) / 60 / 60 / 24;

        //How many times have to be added on the hash
        return days / 7;
}

/*
 Helper function that converts a ascii character to the representing number
 */
word32 btoi(byte b) {
    return (word32)(b - 0x30);
}

/*
 Helper funtion to extract the date
 */
void GetTime(int* value, const byte* date, int* idx) {
    int i = *idx;

    *value += btoi(date[i++]) * 10;
    *value += btoi(date[i++]);

    *idx = i;
}
```

```c

/*
 Extracts the date of a decoded certificate
 idx needs to be chosen correctly
 */
int ExtractDate(const unsigned char* date, unsigned char format,
                struct tm* certTime, int* idx) {
    XMEMSET(certTime, 0, sizeof(struct tm));

    if (format == ASN_UTC_TIME) {
        if (btoi(date[0]) >= 5)
            certTime->tm_year = 1900;
        else
            certTime->tm_year = 2000;
    } else  { /* format == GENERALIZED_TIME */
        certTime->tm_year += btoi(date[*idx]) * 1000; *idx = *idx + 1;
        certTime->tm_year += btoi(date[*idx]) * 100;  *idx = *idx + 1;
    }

    /* adjust tm_year, tm_mon */
    GetTime((int*)&certTime->tm_year, date, idx); certTime->tm_year -= 1900;
    GetTime((int*)&certTime->tm_mon,  date, idx); certTime->tm_mon  -= 1;
    GetTime((int*)&certTime->tm_mday, date, idx);
    GetTime((int*)&certTime->tm_hour, date, idx);
    GetTime((int*)&certTime->tm_min,  date, idx);
    GetTime((int*)&certTime->tm_sec,  date, idx);

    return 1;
}

/*
 Generates a new Novomodo value including the secret
 */
void generateNovomodo(WC_RNG* rng, byte* hash, byte* data, int daysValid) {
    //Generate 32 Byte random
    wc_RNG_GenerateBlock(rng, data, 32);

    //Hash it ceil(daysValid / 7) times
    hashFunc(data, hash, (daysValid + 6) / 7);
}
```

A.9 sqlite-worker.c

```c
#include <stdio.h>
```

```c
#include <sqlite3.h>

#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/ssl.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/sha512.h>

#include "hasher.h"

/*
 Creates the novomodo table
 */
int createDatabase(sqlite3 *db) {
        char *sql;

        /* Create SQL statement */
        sql = "CREATE␣TABLE␣IF␣NOT␣EXISTS␣Secrets("  \
         "SERIAL␣␣␣␣␣␣␣␣␣␣TEXT␣PRIMARY␣KEY␣NOT␣NULL," \
         "secretValue␣␣␣␣␣BLOB␣␣␣␣␣␣␣␣␣␣␣␣␣NOT␣NULL," \
         "currentHash␣␣␣␣␣BLOB␣␣␣␣␣␣␣␣␣␣␣␣␣NOT␣NULL," \
         "currentWeek␣␣␣␣␣INT␣␣␣␣␣␣␣␣␣␣␣␣␣␣NOT␣NULL," \
         "validUntil␣␣␣␣␣␣BLOB␣␣␣␣␣␣␣␣␣␣␣␣␣NOT␣NULL);";

        /* Execute SQL statement */
        return sqlite3_exec(db, sql, NULL, 0, NULL);
}

/*
 Opens the novomodo database
 */
int openDatabase(sqlite3 **db) {
    int ret = sqlite3_open("novomodo.db", db);
    if (ret != SQLITE_OK) return ret;

    return createDatabase(*db);
}

/*
 closes the novmodo database
 */
void closeDatabase(sqlite3 *db) {
```

```
    sqlite3_close(db);
}

/*
 Adds a new decodedcertificate to the novomodo table
 db - database
 cert - decoded certificate
 value - secret value
 hash - current hash of the value
 before - expiry date of certificate
 */
int addSecretValue(sqlite3 *db, DecodedCert cert, byte* value, byte* hash,
                   struct tm before) {
        int ret;
        char time[11];

        byte* serial = cert.serial;

        strftime(time,11,"%Y-%m-%d", &before);

    int hashTimes = calculateHashTimes(&cert);

        sqlite3_stmt *stmt;
        ret = sqlite3_prepare_v2(db, "INSERT␣INTO␣Secrets␣(SERIAL,␣secretValue,␣"
                                 "currentHash,␣currentWeek,␣validUntil)␣VALUES␣"
                                 "(?,?,?,?,?);", -1, &stmt, NULL);
        if (ret != SQLITE_OK) return ret;

        ret = sqlite3_bind_text(stmt, 1, (char*) serial, 16, SQLITE_TRANSIENT);
    if (ret != SQLITE_OK) return ret;
        ret = sqlite3_bind_blob(stmt, 2, value, 32, SQLITE_TRANSIENT);
        if (ret != SQLITE_OK) return ret;
        ret = sqlite3_bind_blob(stmt, 3, hash, 32, SQLITE_TRANSIENT);
        if (ret != SQLITE_OK) return ret;
        ret = sqlite3_bind_int(stmt, 4, hashTimes);
        if (ret != SQLITE_OK) return ret;
        ret = sqlite3_bind_blob(stmt, 5, time, 11, SQLITE_TRANSIENT);
        if (ret != SQLITE_OK) return ret;

        ret = sqlite3_step(stmt);

        return sqlite3_finalize(stmt);
}

/*
 Adds a new certificate to the novomodo table
```

```
 db - database
 cert - certificate
 value - secret value
 hash - current hash of the value
 before - expiry date of certificate
 */
int addSecretValueCert(sqlite3 *db, Cert cert, byte* value, byte* hash,
                       struct tm before) {
    int ret;
    char time[11];

    byte* serial = cert.serial;

    strftime(time,11,"%Y-%m-%d", &before);

    int hashTimes = (cert.daysValid + 6) / 7;

    sqlite3_stmt *stmt;
    ret = sqlite3_prepare_v2(db, "INSERT␣INTO␣Secrets␣(SERIAL,␣secretValue,␣"
                             "currentHash,␣currentWeek,␣validUntil)␣VALUES␣"
                             "(?,?,?,?,?);", -1, &stmt, NULL);
    if (ret != SQLITE_OK) return ret;

    ret = sqlite3_bind_text(stmt, 1, (char*) serial, 16, SQLITE_TRANSIENT);
    if (ret != SQLITE_OK) return ret;
    ret = sqlite3_bind_blob(stmt, 2, value, 32, SQLITE_TRANSIENT);
    if (ret != SQLITE_OK) return ret;
    ret = sqlite3_bind_blob(stmt, 3, hash, 32, SQLITE_TRANSIENT);
    if (ret != SQLITE_OK) return ret;
    ret = sqlite3_bind_int(stmt, 4, hashTimes);
    if (ret != SQLITE_OK) return ret;
    ret = sqlite3_bind_blob(stmt, 5, time, 11, SQLITE_TRANSIENT);
    if (ret != SQLITE_OK) return ret;

    ret = sqlite3_step(stmt);

    return sqlite3_finalize(stmt);
}

/*
 Calculates the current hash or fetches it from database
 */
int getCurrentHash(sqlite3 *db, DecodedCert* cert, byte** hash) {
        int ret;
        int times = calculateHashTimes(cert);
        byte* serial = cert->serial;
```

```

        sqlite3_stmt *stmt;
        ret = sqlite3_prepare_v2(db, "SELECT␣*␣FROM␣Secrets␣WHERE␣SERIAL␣=␣?;",
                                 -1, &stmt, NULL);
        if (ret != SQLITE_OK) return ret;

        ret = sqlite3_bind_text(stmt, 1, (char*) serial, 16, SQLITE_TRANSIENT);
    if (ret != SQLITE_OK) return ret;

        ret = sqlite3_step(stmt);

        if (sqlite3_column_int(stmt, 3) == times) {
        *hash = (byte*) sqlite3_column_text(stmt, 2);
        } else {
                hashFunc((byte*) sqlite3_column_blob(stmt, 1), *hash, times);
                ret = sqlite3_finalize(stmt);
                if (ret != SQLITE_OK) return ret;

                sqlite3_stmt *stmt;
                ret = sqlite3_prepare_v2(db,
                                 "UPDATE␣Secrets␣SET␣currentHash␣=␣?,␣"
                                 "currentWeek␣=␣?␣WHERE␣SERIAL␣=␣?;", -1,
                                 &stmt, NULL);
                if (ret != SQLITE_OK) return ret;

                ret = sqlite3_bind_blob(stmt, 1, *hash, 32, SQLITE_TRANSIENT);
                if (ret != SQLITE_OK) return ret;
                ret = sqlite3_bind_int(stmt, 2, times);
                if (ret != SQLITE_OK) return ret;
                ret = sqlite3_bind_text(stmt, 3, (char*) serial, 16,
                                SQLITE_TRANSIENT);
                if (ret != SQLITE_OK) return ret;

                ret = sqlite3_step(stmt);

                ret = sqlite3_finalize(stmt);
                if (ret != SQLITE_OK) return ret;
        }

        return 0;
}
```

A.10 certgen_root.c

```
/*
This script generates a self signed ecc certificate,
which could be used as root in a PKI enviroment.

Uses a 32 byte ecc key and writes the key and certificate
to files (root-key.der, root-cert.der).
*/

#include <stdio.h>
#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>

#define HEAP_HINT NULL
#define FOURK_SZ 4096

/*
 Generates self-signed root certificate
 */
int main(void) {

        //Return values
        int ret = 0;

        //The certificate
        Cert newCert;

        //File and location to save certificate and key
        FILE* file;
        char newCertOutput[] = "./certs/root-cert.der";
        char newKeyOutput[] = "./certs/root-key.der";

        int derBufSz;

        //Buffer for certificate and key
        byte* derBuf    = malloc(FOURK_SZ);
        byte* pemBuf    = malloc(FOURK_SZ);
        byte* rootKeyBuf = malloc(FOURK_SZ);

        /* Random number generator for MakeCert
        and SignCert and the ecc key*/
        WC_RNG rng;
        ecc_key rootKey;
```

```

    /* Generate new ecc key */
        printf("initializing␣the␣rng\n");
        ret = wc_InitRng(&rng);
        if (ret != 0) goto fail;

        printf("Generating␣a␣new␣ecc␣key\n");
        //Initialize key
        ret = wc_ecc_init(&rootKey);
        if (ret != 0) goto fail;

        //Create Key
        ret = wc_ecc_make_key(&rng, 32, &rootKey);
        if (ret != 0) goto fail;

        //Convert key to der to save it later
        ret = wc_EccKeyToDer(&rootKey, rootKeyBuf, FOURK_SZ);
        if (ret < 0) goto fail;

        printf("Successfully␣created␣new␣ecc␣key\n\n");

    /* Create a new certificate using header information from der cert */
        printf("Setting␣new␣cert␣issuer␣to␣subject␣of␣signer\n");

        //Initialize the certificate
        wc_InitCert(&newCert);

        //Add some X.509 information to the certificate
        strncpy(newCert.subject.country, "DE", CTC_NAME_SIZE);
        strncpy(newCert.subject.state, "NDS", CTC_NAME_SIZE);
        strncpy(newCert.subject.locality, "Gifhorn", CTC_NAME_SIZE);
        strncpy(newCert.subject.org, "IAV", CTC_NAME_SIZE);
        strncpy(newCert.subject.unit, "TD-S1", CTC_NAME_SIZE);
        strncpy(newCert.subject.commonName, "IAV␣root", CTC_NAME_SIZE);
        strncpy(newCert.subject.email, "florian.dahlmann@iav.de", CTC_NAME_SIZE);
        newCert.isCA    = 1;
        newCert.sigType = CTC_SHA256wECDSA;

        //Create the certificate
        ret = wc_MakeCert(&newCert, derBuf, FOURK_SZ, NULL, &rootKey, &rng);
        if (ret < 0) goto fail;

        printf("MakeCert␣returned␣%d\n", ret);

        //Self sign it
        ret = wc_SignCert(newCert.bodySz, newCert.sigType, derBuf, FOURK_SZ,
```

```
                     NULL, &rootKey, &rng);
        if (ret < 0) goto fail;

        derBufSz = ret;

        printf("Successfully␣created␣new␣certificate\n");

    /* write the new cert to file in der format */
        printf("Writing␣newly␣generated␣certificate␣to␣file␣\"%s\"\n",
           newCertOutput);
        file = fopen(newCertOutput, "wb");
        if (!file) {
                printf("failed␣to␣open␣file:␣%s\n", newCertOutput);
                goto fail;
        }

        ret = (int) fwrite(derBuf, 1, derBufSz, file);
        fclose(file);
        printf("Successfully␣output␣%d␣bytes\n", ret);

    /* convert the der to a pem and write it to a file */
    {
        char pemOutput[] = "./certs/root-cert.pem";
        int pemBufSz;

        printf("Convert␣the␣der␣cert␣to␣pem␣formatted␣cert\n");

        pemBuf = (byte*) XMALLOC(FOURK_SZ, HEAP_HINT, DYNAMIC_TYPE_TMP_BUFFER);
        if (pemBuf == NULL) goto fail;

        XMEMSET(pemBuf, 0, FOURK_SZ);

        pemBufSz = wc_DerToPem(derBuf, derBufSz, pemBuf, FOURK_SZ, CERT_TYPE);
        ret = pemBufSz;
        if (pemBufSz < 0) goto fail;

        printf("Resulting␣pem␣buffer␣is␣%d␣bytes\n", pemBufSz);

        file = fopen(pemOutput, "wb");
        if (!file) {
            printf("failed␣to␣open␣file:␣%s\n", pemOutput);
            goto fail;
        }
        fwrite(pemBuf, 1, pemBufSz, file);
        fclose(file);
        printf("Successfully␣converted␣the␣der␣to␣pem.␣Result␣is␣in:␣␣%s\n\n",
```

```
                pemOutput);
    }

    /* write the new key to file in der format */
        printf("Writing␣newly␣generated␣key␣to␣file␣\"%s\"\n", newKeyOutput);
        file = fopen(newKeyOutput, "wb");
        if (!file) {
                printf("failed␣to␣open␣file:␣%s\n", newKeyOutput);
                goto fail;
        }

        ret = (int) fwrite(rootKeyBuf, 1, FOURK_SZ, file);
        fclose(file);
        printf("Successfully␣output␣%d␣bytes\n", ret);

        goto success;

fail:
        printf("Failure␣code␣was␣%d\n", ret);
        return -1;

success:
        printf("Generation␣successful\n");
        return 0;
}
```

A.11 certgen_automotive.c

```
/*
This script generates a self signed ecc certificate,
which could be used as root in a PKI enviroment.

Uses a 32 byte ecc key and writes the key and certificate
to files (root-key.der, root-cert.der).
*/

#include <stdio.h>
#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>

```

```
#define HEAP_HINT NULL
#define FOURK_SZ 4096

#include "certificate-manager.h"
#include "sqlite-worker.h"
#include "hasher.h"

/*
 Generates certificate for car, signed by root
 */
int main(void) {

        //Return values
        int ret = 0;

        //The certificate
        Cert newCert;

        //File and location to save certificate and key
        FILE* file;
        char newCertOutput[] = "./certs/automotive-cert.der";
        char newKeyOutput[] = "./certs/automotive-key.der";

        int derBufSz;

        //Buffer for certificate and key
        byte* derBuf   = malloc(FOURK_SZ);
        byte* pemBuf   = malloc(FOURK_SZ);
        byte* keyBuf = malloc(FOURK_SZ);

    byte* rootBuf = (byte*) XMALLOC(FOURK_SZ, HEAP_HINT,
                                    DYNAMIC_TYPE_TMP_BUFFER);
    ecc_key rootKey;

    ret = wc_ecc_init(&rootKey);
    if (ret != 0) goto fail;

    int rootBufSz = loadRootCert(&rootBuf);
    loadRootKey(&rootKey);

        /* Random number generator for MakeCert
        and SignCert and the ecc key*/
        WC_RNG rng;
        ecc_key key;

    /* Generate new ecc key */
```

```c
        printf("initializing the rng\n");
        ret = wc_InitRng(&rng);
        if (ret != 0) goto fail;

        printf("Generating a new ecc key\n");
        //Initialize key
        ret = wc_ecc_init(&key);
        if (ret != 0) goto fail;

        //Create Key
        ret = wc_ecc_make_key(&rng, 32, &key);
        if (ret != 0) goto fail;

        //Convert key to der to save it later
        ret = wc_EccKeyToDer(&key, keyBuf, FOURK_SZ);
        if (ret < 0) goto fail;

        printf("Successfully created new ecc key\n");

    /* Create a new certificate using header information from der cert */
        //Initialize the certificate
        wc_InitCert(&newCert);

    printf("Generating secret and hash for Novomodo\n");

    byte* hash = malloc(32);
    byte* data = malloc(32);

    generateNovomodo(&rng, hash, data, newCert.daysValid);

    printf("Secret: ");
    printByteAsHexa(data);

    printf("Hash: ");
    printByteAsHexa(hash);

    printf("Setting new cert issuer to subject of signer\n");

        //Add some X.509 information to the certificate
        strncpy(newCert.subject.country, "DE", CTC_NAME_SIZE);
        strncpy(newCert.subject.state, "NDS", CTC_NAME_SIZE);
        strncpy(newCert.subject.locality, "Gifhorn", CTC_NAME_SIZE);
        strncpy(newCert.subject.org, "IAV", CTC_NAME_SIZE);
        strncpy(newCert.subject.unit, "TD-S1", CTC_NAME_SIZE);
        strncpy(newCert.subject.commonName, "Car", CTC_NAME_SIZE);
    strncpy(newCert.subject.email, (char *) hash, 32);
```

```
        newCert.isCA    = 0;
        newCert.sigType = CTC_SHA256wECDSA;

    //Set issuer (the root certificate)
    ret = wc_SetIssuerBuffer(&newCert, rootBuf, rootBufSz);
    if (ret != 0) goto fail;

        //Create the certificate
        ret = wc_MakeCert(&newCert, derBuf, FOURK_SZ, NULL, &key, &rng);
        if (ret < 0) goto fail;

        printf("MakeCert␣returned␣%d\n", ret);

        //Self sign it
        ret = wc_SignCert(newCert.bodySz, newCert.sigType, derBuf, FOURK_SZ, NULL,
                              &rootKey, &rng);
        if (ret < 0) goto fail;

        derBufSz = ret;

        printf("Successfully␣created␣new␣certificate\n");

    /* write the new cert to file in der format */
        printf("Writing␣newly␣generated␣certificate␣to␣file␣\"%s\"\n",
           newCertOutput);
        file = fopen(newCertOutput, "wb");
        if (!file) {
                printf("failed␣to␣open␣file:␣%s\n", newCertOutput);
                goto fail;
        }

        ret = (int) fwrite(derBuf, 1, derBufSz, file);
        fclose(file);
        printf("Successfully␣output␣%d␣bytes\n", ret);

    /* convert the der to a pem and write it to a file */
    {
        char pemOutput[] = "./certs/automotive-cert.pem";
        int pemBufSz;

        printf("Convert␣the␣der␣cert␣to␣pem␣formatted␣cert\n");

        pemBuf = (byte*) XMALLOC(FOURK_SZ, HEAP_HINT, DYNAMIC_TYPE_TMP_BUFFER);
        if (pemBuf == NULL) goto fail;

        XMEMSET(pemBuf, 0, FOURK_SZ);
```

```

        pemBufSz = wc_DerToPem(derBuf, derBufSz, pemBuf, FOURK_SZ, CERT_TYPE);
        ret = pemBufSz;
        if (pemBufSz < 0) goto fail;

        printf("Resulting␣pem␣buffer␣is␣%d␣bytes\n", pemBufSz);

        file = fopen(pemOutput, "wb");
        if (!file) {
            printf("failed␣to␣open␣file:␣%s\n", pemOutput);
            goto fail;
        }
        fwrite(pemBuf, 1, pemBufSz, file);
        fclose(file);
        printf("Successfully␣converted␣the␣der␣to␣pem.␣Result␣is␣in:␣␣%s\n\n",
               pemOutput);
    }

    /* write the new key to file in der format */
        printf("Writing␣newly␣generated␣key␣to␣file␣\"%s\"\n", newKeyOutput);
        file = fopen(newKeyOutput, "wb");
        if (!file) {
                printf("failed␣to␣open␣file:␣%s\n", newKeyOutput);
                goto fail;
        }

        ret = (int) fwrite(keyBuf, 1, FOURK_SZ, file);
        fclose(file);
        printf("Successfully␣output␣%d␣bytes\n", ret);

    /* Add the hash to sqlite table */
    DecodedCert dcert;
    InitDecodedCert(&dcert, derBuf, derBufSz, HEAP_HINT);

    ret = ParseCert(&dcert, CERT_TYPE, NO_VERIFY, 0);
    if (ret != 0) goto fail;

    int idx = 19;
    int length;
    const byte *datePtr = NULL;
    byte format;

    wc_GetDateInfo(dcert.source, dcert.maxIdx, &datePtr, &format, &length);

    struct tm before;
    ExtractDate(dcert.beforeDate, format, &before, &idx);
```

```

    printf("Before␣Date:␣%s\n", asctime(&before));

    printf("Serial:␣");
    printByteAsHexa(dcert.serial);

    printf("Adding␣certificate␣to␣Novomodo␣table\n");
    sqlite3 *db;
    openDatabase(&db);
    addSecretValueCert(db, newCert, data, hash, before);
    closeDatabase(db);

        goto success;

fail:
        printf("Failure␣code␣was␣%d\n", ret);
        return -1;

success:
        printf("Generation␣successful\n");
        return 0;
}
```

A.12 certgen_su_server.c

```
/*
This script generates a self signed ecc certificate,
which could be used as root in a PKI enviroment.

Uses a 32 byte ecc key and writes the key and certificate
to files (root-key.der, root-cert.der).
*/

#include <stdio.h>
#include <wolfssl/options.h>
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/wolfcrypt/asn_public.h>
#include <wolfssl/wolfcrypt/asn.h>
#include <wolfssl/wolfcrypt/error-crypt.h>

#define HEAP_HINT NULL
#define FOURK_SZ 4096

```

```
#include "certificate-manager.h"
#include "sqlite-worker.h"
#include "hasher.h"

/*
 Generates certificate for software update server, signed by root
 */
int main(void) {

        //Return values
        int ret = 0;

        //The certificate
        Cert newCert;

        //File and location to save certificate and key
        FILE* file;
        char newCertOutput[] = "./certs/su-cert.der";
        char newKeyOutput[] = "./certs/su-key.der";

        int derBufSz;

        //Buffer for certificate and key
        byte* derBuf   = malloc(FOURK_SZ);
        byte* pemBuf   = malloc(FOURK_SZ);
        byte* keyBuf = malloc(FOURK_SZ);

    byte* rootBuf = (byte*) XMALLOC(FOURK_SZ, HEAP_HINT,
                                    DYNAMIC_TYPE_TMP_BUFFER);
    ecc_key rootKey;

    ret = wc_ecc_init(&rootKey);
    if (ret != 0) goto fail;

    int rootBufSz = loadRootCert(&rootBuf);
    loadRootKey(&rootKey);

        /* Random number generator for MakeCert
        and SignCert and the ecc key*/
        WC_RNG rng;
        ecc_key key;

    /* Generate new ecc key */
        printf("initializing␣the␣rng\n");
        ret = wc_InitRng(&rng);
        if (ret != 0) goto fail;
```

```

        printf("Generating␣a␣new␣ecc␣key\n");
        //Initialize key
        ret = wc_ecc_init(&key);
        if (ret != 0) goto fail;

        //Create Key
        ret = wc_ecc_make_key(&rng, 32, &key);
        if (ret != 0) goto fail;

        //Convert key to der to save it later
        ret = wc_EccKeyToDer(&key, keyBuf, FOURK_SZ);
        if (ret < 0) goto fail;

        printf("Successfully␣created␣new␣ecc␣key\n");

    /* Create a new certificate using header information from der cert */
        //Initialize the certificate
        wc_InitCert(&newCert);

    printf("Generating␣secret␣and␣hash␣for␣Novomodo\n");

    byte* hash = malloc(32);
    byte* data = malloc(32);

    generateNovomodo(&rng, hash, data, newCert.daysValid);

    printf("Secret:␣");
    printByteAsHexa(data);

    printf("Hash:␣");
    printByteAsHexa(hash);

    printf("Setting␣new␣cert␣issuer␣to␣subject␣of␣signer\n");

        //Add some X.509 information to the certificate
        strncpy(newCert.subject.country, "DE", CTC_NAME_SIZE);
        strncpy(newCert.subject.state, "NDS", CTC_NAME_SIZE);
        strncpy(newCert.subject.locality, "Gifhorn", CTC_NAME_SIZE);
        strncpy(newCert.subject.org, "IAV", CTC_NAME_SIZE);
        strncpy(newCert.subject.unit, "TD-S1", CTC_NAME_SIZE);
        strncpy(newCert.subject.commonName, "Software␣Update␣Server", CTC_NAME_SIZE);
    strncpy(newCert.subject.email, (char *) hash, 32);
        newCert.isCA    = 0;
        newCert.sigType = CTC_SHA256wECDSA;

```

```
    //Set issuer (the root certificate)
    ret = wc_SetIssuerBuffer(&newCert, rootBuf, rootBufSz);
    if (ret != 0) goto fail;

        //Create the certificate
        ret = wc_MakeCert(&newCert, derBuf, FOURK_SZ, NULL, &key, &rng); //ecc certificate
        if (ret < 0) goto fail;

        printf("MakeCert␣returned␣%d\n", ret);

        //Self sign it
        ret = wc_SignCert(newCert.bodySz, newCert.sigType, derBuf, FOURK_SZ,
                          NULL, &rootKey, &rng);
        if (ret < 0) goto fail;

        derBufSz = ret;

        printf("Successfully␣created␣new␣certificate\n");

    /* write the new cert to file in der format */
        printf("Writing␣newly␣generated␣certificate␣to␣file␣\"%s\"\n",
           newCertOutput);
        file = fopen(newCertOutput, "wb");
        if (!file) {
                printf("failed␣to␣open␣file:␣%s\n", newCertOutput);
                goto fail;
        }

        ret = (int) fwrite(derBuf, 1, derBufSz, file);
        fclose(file);
        printf("Successfully␣output␣%d␣bytes\n", ret);

    /* convert the der to a pem and write it to a file */
    {
        char pemOutput[] = "./certs/su-cert.pem";
        int pemBufSz;

        printf("Convert␣the␣der␣cert␣to␣pem␣formatted␣cert\n");

        pemBuf = (byte*) XMALLOC(FOURK_SZ, HEAP_HINT, DYNAMIC_TYPE_TMP_BUFFER);
        if (pemBuf == NULL) goto fail;

        XMEMSET(pemBuf, 0, FOURK_SZ);

        pemBufSz = wc_DerToPem(derBuf, derBufSz, pemBuf, FOURK_SZ, CERT_TYPE);
        ret = pemBufSz;
```

```
        if (pemBufSz < 0) goto fail;

        printf("Resulting pem buffer is %d bytes\n", pemBufSz);

        file = fopen(pemOutput, "wb");
        if (!file) {
            printf("failed to open file: %s\n", pemOutput);
            goto fail;
        }
        fwrite(pemBuf, 1, pemBufSz, file);
        fclose(file);
        printf("Successfully converted the der to pem. Result is in:  %s\n\n",
               pemOutput);
    }

    /* write the new key to file in der format */
        printf("Writing newly generated key to file \"%s\"\n", newKeyOutput);
        file = fopen(newKeyOutput, "wb");
        if (!file) {
                printf("failed to open file: %s\n", newKeyOutput);
                goto fail;
        }

        ret = (int) fwrite(keyBuf, 1, FOURK_SZ, file);
        fclose(file);
        printf("Successfully output %d bytes\n", ret);

    /* Add the hash to sqlite table */
    DecodedCert dcert;
    InitDecodedCert(&dcert, derBuf, derBufSz, HEAP_HINT);

    ret = ParseCert(&dcert, CERT_TYPE, NO_VERIFY, 0);
    if (ret != 0) goto fail;

    int idx = 19;
    int length;
    const byte *datePtr = NULL;
    byte format;

    wc_GetDateInfo(dcert.source, dcert.maxIdx, &datePtr, &format, &length);

    struct tm before;
    ExtractDate(dcert.beforeDate, format, &before, &idx);

    printf("Before Date: %s\n", asctime(&before));

```

```
    printf("Serial:␣");

    printByteAsHexa(dcert.serial);

    printf("Adding␣certificate␣to␣Novomodo␣table\n");
    sqlite3 *db;
    openDatabase(&db);
    addSecretValueCert(db, newCert, data, hash, before);
    closeDatabase(db);

        goto success;

fail:
        printf("Failure␣code␣was␣%d\n", ret);
        return -1;

success:
        printf("Generation␣successful\n");
        return 0;
}
```