Research Internship mDL

Martijn Terpstra

July 20, 2017

1 Introduction

The mDL, short for mobile drivers license, is a system where a user can authenticate much the same way as with a physical drivers license. Compared to the physical drivers license, the mDL would offer some new advantages like selective disclosure of attributes and a shorter window of time in which a revoked drivers license could be passed off as unprovoked. On the other hand, the mDL introduces new challenges. The mDL's authenticity cannot be assured by physical inspection

1.1 Context

I worked on my research at UL Transactional security Leiden. UL's interest is in formalizing an ISO standard for the mDL to foster adaptation. To keep complexity down, currently only the scenarios where a verifier is physically present are being considered Currently the formalization of this standard is still in its early stages and in its current phase UL wants to look into how to manage potential security risks. The problem that I researched at UL was to map out the security requirements, threats and possible mitigations needed for an mDL.

2 Internship

I employed as an intern at UL for about three months. I enjoyed worked alongside and with UL employees. I worked under a team managed by Paul Hin. Most contact was with other members of this team. At UL there were different values than at the university.

At UL I was able to discuss technical problems with employees with relevant knowledge and did so to verify that I correctly understood the project potential security issues.

I first spend a few weeks familiarizing myself with the project. After that I described on a high level the parties and use cases. I looked at security measures taken in (Dutch) passports and physical drivers licenses. After that, I looked at security assessments of similar projects. Most interestingly mobile payment apps had a large amount of overlap in threats and possible security measures.

Based on that Information I informally made some subjective classifications. One interesting point here is that after researching it, I found that android malware was far more prevalent than I assumed. Based on the research and conversations with UL employees I found that certain risks were worth focusing on.

In between I spend two days at UL attending a meeting of the mDL ISO work group. Although most discussions, (e.g. should Bluetooth low energy be a hard requirement for reader?) where not relevant to my assessments of the security, I did learn what the parties value. Some parties attending has a surprisingly lax view of security, suggesting quite trivially broken ideas such as a static set of data without any data with winch the holder can be recognized.

After discussing the assessment of threats and mitigations, I looked more in depth into the storage of sensitive and secret information. This was with the intent of preventing the sharing of mDL data and limiting the damage malware can do. After that I realized that some risks can only be mitigated by the use of secure hardware, so I looked into the options there.

A promising solution was to use the ARM TrustZone technology. This is a method of secure storage embedded in ARM chips present in most smartphones, and can be used by the android keystore. In recent Android version it is possible to obtain a certificate chain of keys signed by this chip to verify that this key is stored and generated in hardware.

At the end to test this functionality I wrote an android script to make a phone generate such a key and try to verify that it has indeed been stored in hardware.

3 Overview mDL

An mDL is intended to be used as an alternative to the physical drivers license. It can be used for both conveying driving certification and as a legal method of identification.

Compared to the physical drivers license, the mDL offers some advantages. The attributes stored on the mDL can be updated far quicker than on

a physical drivers license. Physical drivers licenses can be valid for a decade without their stored attributes changing. The attributes stored on an mDL could be updated as soon as the mDL holder is online.

However, it also comes with new challenges. Unlike a physical drivers licenses, which has to be physically copied, a mDL can be duplicated by extracting information. Also, without a physical form it is hard to verify that the device the mDL is stored on is physically present.

3.1 Parties Involved

There are three types of parties involved with the mDL.

- The Issuer, who will issue the mDL to holders.
- The Holder, who uses an mDL specific to him to authenticate to a reader.
- The Reader, who want a holder to authenticate to it and must verify the authentication.

Multiple Issuers, holders and readers will exist. However, during any single authentication only a single issuer, holder and reader need to be taken into account. Before authentication, the Holder shall have received a personalized mDL and the reader shall learn the identity and public keys of valid Issuers.

3.2 Requirements successful authentication

On a high level the following needs to happen

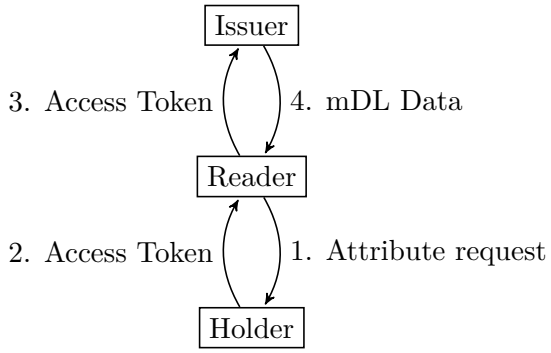
1. The reader has to specify what attributes it wants to verify from a holder. (e.g. a reader might want to know if a holder is of drinking age)
2. The holder has to proof that a valid Issuer assigned these attributes to his mDL. This will take the form of a digital signature of the Issuer.
3. The holder has to proof that his mDL is issued to him specifically. This will typically include a digital face image, linked to the mDL and signed by the issuer.
4. The Reader as to verify these proofs.

3.3 Online vs Offline authentication

When authenticating using an mDL there is a difference between online and offline authentication. The Issuer is an active participant during online authentication but is absent during offline authentication.

3.3.1 Online authentication

Authenticating with a connection between the Reader the Issuer is useful because the reader can be sure that the data received is up to date and the protocol is fundamentally simpler. The downsides to it are that each authentication of the holder is revealed to the issuer (privacy), selective attribute disclose is hard, the Issuer has to make sure that it is always available and authentications are not possible if the reader cannot establish a connection to the Issuer.



Online authentication can be done with access tokens, where an Issuer can refuse to reveal the mDL data is a token is reused. The generation of these token can be done in two ways.

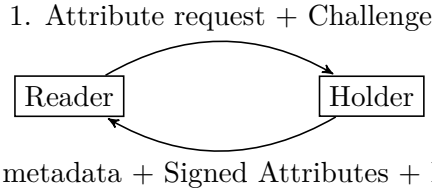
Firstly the Issuer can generate and distribute these tokens. The advantage of this is that the Issuer can limit the amount of tokens in circulation. The disadvantage is that tokens for disclosing specific combinations of attributes are non-trivial. Sending multiple token to a reader may be impractical and storing tokens for every combination of attributes is infeasible.

Secondly the holder itself could generate these tokens. The tokens would be signed authorizations signed with the holders secret key (known to the issuer), to let the issuer know which attributes it wants to disclose. These could be generated at the time of authentication.

A problem that is present with the use of tokens is that the reader can ask for an access token but not use it. A reader could then give the token to another reader and prevent to be the Holder.

3.3.2 Offline authentication

Authenticating without the Issuer needing to be present allows for authentication using just the reader and holder. Offline authentication can be used in situations where online authentication is not possible.



The reader first specifies the attributes that need to be proven and includes a unique challenge to prevent replay attacks. Without this challenge a non-holder could simply replay an earlier recorded authentication to the reader. The holder responds with some mDL metadata, which includes the identity of the Issuer, expiry date of the mDL (attributes), the public key of this mDL, and some physical identifier like a face image and is signed in whole by the Issuer. Next the holder includes a response for every attribute requested. This response includes the name and value of the attribute and is signed together with the mDL metadata. Lastly the holder responds with a response, signed with its secret key to verify that the authentication is specifically for this reader.

3.4 Privacy concerns

To protect he privacy of the mDL holder, no more personal information should be revealed than is necessary.

3.4.1 Information revealed to reader

The holder only interacts with readers during authentication. Any personal data which is an attribute in the authentication is revealed to the reader. Unlike the physical drivers license a holder may choose to reveal some but not all available attributes during an authentication.

Additionally, some information is revealed regardless of the attributes requested for authentication.

Issuer: The reader will learn which issuer issued the holders mDL. If specific issuers are responsible for issuing mDLs in specific regions, the identity of the issuer will reveal the holder's nationality.

Bio-metric: To verify that an mDL belongs to the physical person presenting the mDL a verification using bio-metrics is needed. If this bio-metric is a face image this reveals no more information than a reader could otherwise learn. However, the digital format allows for easy storage of the bio-metric. If the holder always presents the exact same bio-metric, the bio-metric can be used as a unique identifier

Signing key: The mDL holder may reveal his signing key to sign a challenge to prevent a replay attack. However, since this key is unique it can be used as a unique identifier across different readers if readers share information.

Expiry date mDL: The mDL may reveal the expiry date, to prevent an expired mDL from authenticating. While this may give an indication of when the holder last updated his mDL, it does not reveal any private data.

Because authentication using the mDL is done using digital messages rather than inspecting a physical object, any disclosed information can be easily stored and processed. If multiple readers are controlled by the same party, this data can be accumulated and could be used to profile users.

3.4.2 Information revealed to issuer

To be able to authenticate the mDL holder has to interact with the issuer.

Issuance: During issuance the Issuer known the mDL attributes of the holder. During the issuing of a new mDL the issuer can learn that the holder obtains a mDL and the holder public keys, neither of which is very sensitive data.

Authentication: To whom, when are where a holder authenticates can be revealing to an issuer. To protect the holders privates, the issuer should not process or keep a record of the holders authentications. The only way for a holder can be sure the issuer does not record his authentications is if the holder never learns about the authentications. Since this is not possible with online authentication, offline authentication provides a significant advantage in terms of privacy.

Updating of mDL: mDL data should be regularly updated, to keep high confidence that the data is correct and lower the value of stolen mDL data. Since this requires the mDL application to contact the issuer, the connection setup to download the updated data can reveal sensitive information. The connection would reveal that a device is active at a certain time and will likely reveal an approximate location of the device. For users whose device is frequently internet connected, the ability to automatically setup a connection between the mDL application and the issuer could be a covert

way of tracking a users activity and location. Because of this potential to infringe the holder's privacy, and the fact that a delayed update poses no issue if the holder does not authenticate in the mean time, it is ill-advised to do unattended updates of mDL data.

4 Threats

4.1 Lookalike fraud

It is unreasonable to assume we have a perfect bio-metric, a bio-metric that easily matches the mDL holder but no other person. If an attacker obtains a valid mDL from a holder, with or without the holder's cooperation, and the bio-metric is similar enough to the attacker, the attacker can use this mDL to authenticate to readers.

4.2 Authentication using forged mDL

The authenticity of an mDL should be vetted by known issuers rather than the mDL holders themselves.

If the reader only verifies the authenticity based on the holder's response without taking into account the issuer, any attacker can claim to have a valid mDL with any attributes it wants.

4.3 Authentication using expired/revoked mDL

Simply because an mDL was once valid does not imply it is currently valid.

4.4 Leaking personal data without authenticating

The mDL should not authenticate simple because a reader is asking for an authentication. When it does, an attacker simply has to be within communication range of a holder to obtain his personal information.

4.5 Leaking personal data to issuer

Due to privacy concerns it is not desirable that the issuer learns of every authentication done by a holder. A mDL should not need to contact the issuer for every authentication.

4.6 Oversharing data to reader

It should not be possible for a reader to trick a holder into disclosing more attributes that are needed for the authentication.

5 Attacks

5.1 Lending out device with mDL

The simplest attack would be for a holder to lend out his device to a lookalike.

5.2 Use mDL after it has expired/revoked

If the reader does not verify the current status of an mDL, a holder could simply keep an mDL after it has been revoked or expires and try to authenticate with it.

5.3 Copy/Paste of mDL app and data

If the mDL data can be copied, any device holding the mDL application and mDL data is seen as a valid mDL by a reader. Unlike the lending out of a device this attack could allow multiple attackers to obtain a valid mDL.

5.4 Obtain personal data with malware

Since the mDL is stored on a device that allows for the installation of external software and the ubiquity of malware on smartphones.

5.5 Attacker-controlled reader

An attacker is able to create his own reader and may be able to trick a holder into authentication to it. Once this user has authenticated, the attacker learned the holders private data. When an mDL automatically authenticates without user interaction, an attacker can harvest a large amount of personal data simply by placing a reader in a location where many people pass by.

5.6 Legitimate reader asks for more attributes that needed

A reader can ask for any attribute that the mDL holder can have. When the holder is not able to tell the difference, the reader learn too much private information.

5.7 Attacker generates own mDL data

If there are no proper check in place to verify the authenticity of an mDL, the holder can make up his own data and authenticate with that.

6 Mitigations

6.1 Preventing attack

6.1.1 Require user confirmation before authenticating

A simple dialog box can prevent the mDL both from authenticating without the users consent and from revealing more attributes that reasonable.

Additionally, the holder application should be setup to only respond to authentications after the holder has informed the app that an authentication will take place. This would prevent ignorant holders from accepting unexpected requests.

6.1.2 Sign and verify signatures of mDL data

The readers should allow a limited amount of issuers and verify that any mDL was signed by known issuer.

6.1.3 Include challenge with request

If the holder has a private key for signing the challenge and the holders key if itself signed by a known issuer, the reader can be sure that the mDL data was not simple copied from a capture of previous authentication(s).

6.1.4 Hardware storage of keys

If we store the key used to sign challenges in tamper resistant hardware we prevent an attacker from copying the mDL and using it on devices other that the device it was issued to.

6.1.5 Punish mDL misuse

If the mDL is a legal identification, the misuse of mDL is identity fraud and can be treated as such. If the risk of misusing an mDL is high, it would make attacker less likely to try.

6.1.6 Allow mDL only on certain devices

If the level of security of a device cannot be guaranteed on certain devices, they can be excluded from use with mDL.

6.1.7 Terminal registration

Requiring terminals to register at all issuers, requiring readers to authenticate to the holders and requiring holder to verify these authentications before authenticating themselves, solves some problems but introduces new problems as well.

Firstly it would hinder an attacker from creating a terminal that tricks a holder into authenticating to obtain their personal data.

Secondly it would allow for classification of terminals, where a specific class of terminals could only ask for certain attributes (e.g. bars could ask only for age checks and border control could ask for all attributes). This would make it harder for readers to ask (and get) more attributes than needed.

However, terminal registration is not desirable because it comes with significant downsides. Firstly it makes mDLs non inter-operable across jurisdictions. An mDL that only authenticates to registered terminals won't register to terminals in jurisdictions that forego registration.

Secondly it would hamper adoption. Registering your own terminals at all issuers and keeping track of all registered terminals, including their revocation in specific authorizations around the world would be difficult.

Lastly the verification of terminals may in itself reveal personal information. If a holder has to query its issuer to verify that a specific terminal is registered, it will reveal who it is trying to authenticate to.

6.2 Limit damage possible with attack

6.2.1 Monitor mDL behavior as issuer

An issuer can keep track of how often a holder updates. When the Issuer notices the mDL is updated more often that reasonable, it can revoke the mDL.

6.2.2 Limit mDL issued per person

Would prevent the holder from easily lending out a spare device to a lookalike attacker.

6.2.3 Limit attributes

If an mDL stores only limited attributes, the attacker could only use the mDL only for limited purposes and devaluing the mDL.

6.2.4 Limit validity of mDL unless updated

This way even if the mDL is compromised, the issuer can stop updating the mDL and the mDL can only be used for a short amount of time.

7 Remaining attacks

Unless we have a perfect[1] bio-metric or an honest[2] mDL holder, there are unavoidable attacks. Although an attacker can be punished after executing, the attacker cannot be prevented from executing these attacks and detection is difficult.

7.1 Sharing physical device

Much like a physical drivers license, a device with an mDL stored can be shared with a lookalike.

7.2 Relay attack

Unlike the physical drivers license, the mDL's authentication is done via digitally send information. An attacker without mDL can foward any request to authenticate to a valid mDL, and relay the responses from the mDL to the reader. If the mDL device is set up to respond automatically, the mDL device could be used to authenticate for many users.

[1] A bio-metric that matches the holder but no other person
[2] A holder that does not share his device or keys with other

8 Appendix

8.1 Categorization of attacks and mitigations

8.1.1 Categorization of difficulty of attack

Category	Difficulty
Nation State	Requires nation state support
Expert	Requires some expert knowledge
Layman	Can be done by layman

The difficulty here is under the assumption that no mitigating measures have been taken yet.

8.1.2 Categorization of impact of attack

Category	Damages
Low	Minor
Medium	Major
High	Party abandons mDL

8.1.3 Threats

Holder shares mDL Device with lookalike

Threat	Holder shares mDL Device with lookalike
Impact	Low Only single mDL is misused and can only be done by people who look alike
Difficulty	Layman Can be as easy as lending out your phone
Mitigations	M-A01 Enforce strict rules for ID matching M-A14 Penalize mDL holder for misuse of mDL

Holder shares mDL Data with lookalike

Threat	Holder shares mDL Data with lookalike
Impact	Medium a single mDL could be shared with many people
Difficulty	Expert Requires copying of data stored on device
Mitigations	M-A01 Enforce strict rules for ID matching M-A14 Penalize mDL holder for misuse of mDL M-A15 Device binding M-A28 Hide holders private key using white-box cryptography M-A30 Store sensitive data in hardware

Holder authentications with expired mDL

Threat	Holder authenticates with expired mDL
Impact	Medium Expired mDL should be treated as invalid mDL
Difficulty	Layman Holder has to reuse old mDL
Mitigations	M-A12 Verify expiry date at reader

Holder authentications with revoked mDL

Threat	Holder authenticates with revoked mDL
Impact	Medium holder authenticates with mDL with attributes that are no longer valid
Difficulty	Layman Holder just keeps mDL when revoked
Mitigations	M-B01 Short expiry date/ frequent refresh M-A12 Verify expiry date at reader M-A17 Treat expired mDLs as revoked until updated M-C03 Issuer keeps track of status polling

Fake mDL Application

Threat	Attacker tricks user into using fake (repackaged) mDL application
Impact	High Loss in trust of mDL PII is leaked and mDL can be copied
Difficulty	Expert Requires application development knowledge
Mitigations	M-A18 Mandate use of issuer approved apps M-A19 Distribute mDL apps via Issuer M-A20 Sign official mDL applications M-A21 Issue official mDL as free software M-A25 Check if mDL is running in emulator M-A26 Check if USB debugging is enabled M-A27 Check if installation from unknown sources is allowed M-A28 Hide holders private key using white-box cryptography M-A29 Obfuscate mDL application M-B03 Integrity check of mDL app at runtime by holder device M-B05 Regular scan of App stores for fake mDL app

Malware

Threat	Attacker compromises holder device with other malware
Impact	Medium PII is leaked and mDL can be copied
Difficulty	Expert Requires application development knowledge
Mitigations	M-A23 Dissalow use of rooted devices M-A24 Integrated detection for malware M-A30 Store sensitive data in hardware M-B03 Integrity check of mDL app at runtime by holder device M-A31 Store sensitive data encrypted with holder-chosen password

Attacker successfully creates a new mDL without Issuer interaction

Threat	Attacker successfully creates a new mDL without Issuer interaction.
Impact	High Creation of non-issuer mDL would mean mDL are no longer trusted
Difficulty	Nation state Requires impersonation of publicly known participant
Mitigations	M-A04 Issuer must sign mDL with strong cryptographic key

Attacker compromises Issuer

Threat	Attacker compromises Issuer.
Impact	High Would mean no mDL signed by Issuer is trusted
Difficulty	Nation State Would likely require infiltration
Mitigations	M-A05 Issuer should protect its own signing keys (Out of scope)

Encryption scheme is broken

Threat	Encryption scheme is broken.
Impact	High Would allow attacker to sign anything
Difficulty	Nation State Would require backdoored or weak encryption to be the standard
Mitigations	M-A06 Use strong encryption

Thief authenticates with stolen mDL

Threat	Thief authenticates with stolen mDL
Impact	Medium Compromise of a single mDL
Difficulty	Expert Requires physical interaction Use may revoke mDL when stolen
Mitigations	M-A01 Enforce strict rules for ID matching M-A12 Verify expiry date at reader during authentication M-B04 Require holders to report stolen/lost phones to Issuer M-A31 Store sensitive data encrypted with holder-chosen password M-B01 Short expiry date/ frequent refresh

Attacker tricks mDL application to make it delete the mDL data

Threat	Attacker tricks mDL application to make it delete the mDL data
Impact	Medium Impacts a single holder, does not prevent holder from re-downloading mDL data
Difficulty	Medium Attacker has to understand the protocol
Mitigations	M-A10 Enforce security requirement holder devices

Issuer revokes wrong mDL by mistake

Threat	Issuer is tricked into revoking wrong mDL
Impact	Medium Single holder is affected, new mDL can be issued easily
Difficulty	Expert Requires knowledge of revocation process
Mitigations	M-A11 Hold Issuer liable for wrong commands send to mDL

Attacker sends wrong or invalid attributes to mDL

Threat	Attacker sends wrong or invalid attributes to mDL
Impact	Medium Single mDL is invalidated
Difficulty	Expert Requires knowledge of the update protocol
Mitigations	M-A06 Use strong encryption M-A11 Hold Issuer liable for wrong commands send to mDL M-A05 Issuer should protect its own signing keys (Out of scope)

Attacker learns attributes

Threat	Attacker learns attributes, by setting up own reader
Impact	Low
Difficulty	Layman
Mitigations	M-A07 Instruct users not to authenticate trivially

The fact that a specific mDL is revoked becomes known to unintended parties

Threat	The fact that a specific mDL is revoked becomes known to unintended parties.
Impact	Medium One time comprise of single mDL metadata
Difficulty	Expert attacker need to obtain and use mDL polling token
Mitigations	M-A07 Instruct users not to authenticate trivially M-A10 Enforce security requirement holder devices

Attacker forges access token

Threat	Attacker forges access token
Impact	High Potential to acces any mDL data of any holder
Difficulty	Nation State Required forgery of signatures
Mitigations	M-A06 Use strong encryption M-A13 Register terminals M-A16 Allow only offline authentication M-A31 Store sensitive data encrypted with holder-chosen password

8.1.4 Mitigations

Recommended mitigations are underlined

Mitigations that limit likelihood of attack

- M-A01 Enforce strict rules for ID matching
  - Would put restrictions on users on what pictures are acceptable for identification, same as with normal drivers license
- M-A02 Enforce strict rules for acceptable ID during mDL initialization
  - Would require extra scrutiny during initialization
- M-A03 Create ID during initialization
  - Would make initialization slower because ID cannot be taken beforehand
- M-A04 Issuer must sign mDL with strong cryptographic key
  - Would not impact user
- M-A05 Issuer should protect its own signing keys
  - Out of scope
- M-A06 Use strong encryption
  - Should be invisible to user
- M-A07 Instruct users not to authenticate trivially

  - Would require user to distinguish when authentication is actually needed

- M-A08 Setup secure channel for communication between mDL and reader
  - Might prevent fast and error-free channel from being used

- M-A09 Setup secure channel for communication between mDL and Issuer
  - Would prevent attacker from issuing fake updates.

- M-A10 Enforce security requirement holder devices
  - Would remove freedom of holder to secure device by own standards

- M-A11 Hold Issuer liable for wrong commands send to mDL
  - Would allow user to blindly trust issuer when it tells the holder their mDL is expired

- M-A12 Verify expiry date at reader during authentication
  - Simple check by reader

- M-A13 Register terminals
  - mDLs that would only authenticate to registered terminals, would be incompatible in other jurisdictions.
  - Would violate privacy of holder by disclosing to Issuer all authentications

- M-A14 Penalize mDL holder for misuse of mDL
  - Could be as simple issuing a fine when it is discovered a holder lend out his mDL.

- M-A15 User device binding on top of bio-metrics to verify identity
  - Might prevent holder from using hardware of choice
    * Would require holder to get new mDL each time he wants to use it on a new device.

  - Requires each holder device to be easily finger-printable, which might not be desirable outside of mDL usage.

- M-A16 Allow only offline authentication
  - Would require proper channel between holder and reader for authentication

- M-A17 Treat expired mDLs as revoked until updated
  - Would require holder to update mDL even when the data is not neccesarily invalid.

- M-A18 Mandate use of issuer approved apps

- M-A19 Distribute mDL apps via Issuer
  - Requires that holders learn how to install software from new sources

- M-A20 Sign official mDL applications
  - Requires policy for holders to verify signatures are valid and from issuer

- M-A21 Issue official mDL as free software
  - Would allow anyone to audit source
  - Would allow users to create hardened versions
  - Would not require repackaging for special functionality
  - Would create distrust in non-free (potential malware) implementations
  - Would make monetization of app difficult

- M-A22 Store only a limited set of attributes on mDL
  - Would prevent leakage of attributes not stored on compromise
  - Would limit situations in which mDL can be used to authenticate

- M-A23 Dissalow use of rooted devices
  - Denying people root access to their personal devices is quite dystopian

  - Would make it harder for malware to obtain mDL data

- M-A24 Integrated detection for malware
  - Traditional malware scanners are trivial to circumvent This approach would be either complex or quite limited in effectiveness.

- M-A25 Check if mDL is running in emulator
  - Would prevent debugging attempts of attacker

- M-A26 Check if USB debugging is enabled
  - There are legitimate reasons for enabling USB debugging.

- M-A27 Check if installation from unknown sources is allowed
  - Would prevent issuer from distributing mDL itself outside app store

- M-A28 Hide holders private key using white-box cryptography
  - Would make it harder for malware to obtain unencrypted mDL data
  - Would make it harder for user to obtain mDL data for duplication of mDL
  - Encrypted mDL data may be just as valuable in combination with mDL app
  - Essentially this is just security by obscurity

- M-A29 Obfuscate mDL application
  - Security by obscurity

- M-A30 Store sensitive data in hardware

- M-A31 Store sensitive data encrypted with holder-chosen password
  - Would prevent data leaks when mDL is not used
  - Would require user to enter password each time it wants to use the mDL

Mitigations that limit impact of attack

- M-B01 Short expiry date/ frequent refresh
  - Would require user to update frequently
  - Would limit benefit to stolen mDL if attacker can't update
- M-B02 Require holders permission to check revocation status of mDL
  - Would prevent readers from learning if mDL was revoked after authentication
- M-B03 Integrity check of mDL app at runtime by holder device
  - Might detect compromise of mDL app or holder device
  - Easily bypassed if mDL app is easy to modify
- M-B04 Require holders to report stolen/lost phones to Issuer
  - Issuer would learn about loss of phone (privacy)
  - Issuer can revoke mDL before it is used by thief
- M-B05 Regular scan of App stores for fake mDL

Mitiagtions based on Issuer data collection While information about the holder revealed to the issuer is limited

- M-C01 Issuer notifies holder of unusual patterns
- M-C02 Issuer keeps track of token use
  - Would be privacy invasive.
  - Issuer can keep track of frequency authentication (e.g. 100 authentication per hour is suspicous)
  - Issuer can keep track of time in between authentication to different parties. Trying to authenticate to two parties within a minute would be suspicous.
  - Issuer could keep track of polling attempts with expired access tokens
- M-C03 Issuer keeps track of status polling
  - Status polling is not revealing perse

  - Issuer can keep track of frequent polling
  - Issuer could keep track of polling attempts with expired polling tokens (might indiccate theft of mDL)

- M-C04 Issuer keeps track of Updates issued to holder
  - Limit amount of successful updates per holder per time period.
  - Requires holder app to confirm of successful updates
  - Can be used to deny cloned mDLs update

8.2 Secure storage of key material

8.2.1 Assets

To perform authentication, some data has to be stored on the mDL itself. What has to be stored on the device depends on what type of authentication is necessary.

Type		Offline Authentication	Self-generated token	Issuer-token
Non-revealing	I: Issuer name	✓	✓	✓
	pkH: Identifier to check signature	✓	✓	
	E: Expiry date mDL	✓		
	Signature over (ID,E,pkH)	✓		
Personal	ID: Face image	✓	✓	
	Attribute responses	✓		
Secret	skH: Private key	✓	✓	
	Issuer generated tokens			✓

These can be split up into two categories

- Non-revealing: This data is useless by itself, however can link some authentication to the same holder.
- Personal: This data reveals private information
- Secret: This data, if leaked, would allow an attacker to authenticate as if it were the holder.

8.2.2 Threats

Attack without holder cooperation If the attacker is a third party, the data can be stored encrypted in such a way that it can only be decrypted (on use) using knowledge only known to the holder.

Root access adversary If the adversary can obtain (the equivalent of) root access on the holder's device, it is only possible to limit the time frame in which the holder's data is used. Whenever it is not used, the mDL data should be stored encrypted with a key not stored on the device itself, to prevent an attacker from learning the mDL private data even when the adversary can take full control over the device. Care has to be taken that after use, sensitive data is not only de-referenced within the mDL application, but also overwritten to prevent memory scraping from reveal the secrets.

Ultimately this measure still might not be sufficient. While adversary cannot obtain the secrets before the user decrypts the data, if the adversary is persistent and patient enough, eventually the secret will be decrypted because they will be used.

One possible mitigation that would permanently prevent a root access adversary from learning secrets is to limit the available attributes on an mDL. For instance an mDL might be usable for drivers license category and age checks but would not be usable to authenticate based on other attributes.

Malicious application The same mitigation that can be used against a root access adversary attacker can be used against a non-root access adversary. In addition, there are mitigations that would not be effective against a root-access attacker but would prove useful against non-root acces adversaries.

Attack with holder cooperation The mDL data should be available to the mDL application for authentication. However, if the holder cannot be trusted with the mDL data is may be useful to obfuscate some mDL data from the user.

The holder can be expected to be known or learn all but his own private key skH and token that are not yet used. Self-generated tokens and Offline authentication requires the use of the private key skH for signing. Authentication using Issuer-generated token does not require the use of the holder's private key during authentication.

However, the holder still has to retrieve new tokens from the issuer which likely involves authentication to the Issuer using the holder's private key.

Obfuscation of skH Merely obfuscating skH is not enough. We want to take the freedom from the user to sign using his own private key outside the mDL application on his own device. If a derivative signing function plus optionally a derived signing key, then the new signing function and key are just as valuable.

1. $$\text{sign}(m, skH) = m_{\text{signed}}$$

   Then the holder must not learn both the signing function sign and the key pkH.

2. $$\text{sign}_{\text{obfuscated}}(m, skH_{\text{obfuscated}}) = m_{\text{signed}}$$

   Then the holder must not learn both the signing function $\text{sign}_{\text{obfuscated}}$ and the key $pkH_{\text{obfuscated}}$.

3. $$\text{sign}_{\text{hard-coded}}(m) = m_{\text{signed}}$$

   Then the holder2 must not learn the signing function $\text{sign}_{\text{hard-coded}}$.

One possible way of bypassing this is to include some unspoofable external factor (hardware challenge, hardware fingerprint) to the signing function. However, if this is possible there is no point in obfuscating the signing function, the reader can simply include this external factor in the verification the holder's identity.

Issuer-holder relation In order to prevent the holder from duplicating the mDL without Issuer interaction, the mDL issuer may choose to limit the control the user has over the mDL, its data and its computations. However, if the mDL is expected to run on a holder's personal device limiting the control of the holder over his own devices may pose a moral dilemma.

The use of smartphones (and similar devices) devices may be essential for participation in modern society. If a holder is dependent on his personal devices, the control over personal computing will likely be valued highly by the holder. To a holder who values the control over his personal devices highly, limitations imposed which limit the holder's control over his device, would prevent adaptation or continued use of the mDL.

8.2.3 Mitigations

{NOTE: some following mitigations are specifically for the Android operating system. Similar mitigation may or may not be possible on other mobile operating systems}

Policies

Revocation of mDL on misuse A recommended mitigation to discourage mDL holder to cooperate with attackers (e.g. to clone mDLs), would be to (permanently) revoke any mDL whose holders are caught sharing their mDL data needed for authentication.

Limiting available attributes To limit value of the mDL data you could limit what attributes are stored on the mDL. This does not prevent an attacker from learning the mDL data. However, it does devalue the mDL an attacker would like to clone.

Hardware-based mitigations The use of a hardware secure element could be used to obfuscate the holder's private key. It would allow for the storage of and computation using skH without the holder knowing the key. A caveat here is that a holder may very well have multiple devices that he wishes to use his mDL on. The Issuer could choice to limit any holder to at most 1 mDL or would have to support multiple mDLs per holder. The holder would either be inconvenienced by only being to able to use a single device for mDL or by having to update each device separately since each would have its own secret key.

Trusted Execution Environment The TEE (Trusted Execution Environment) allows for a special operating system (e.g. Trusty[3]) to be run on a separate processor. The normal operating system can communicate with the TEE, asking it to encrypt/decrypt/sign etc. but cannot access its memory or its computations.

Android KeyStore The Android developer notes[4] of suggest that since android 4.3 there exist and easy way to ensure that keys are hardware bound.

[3] `https://source.android.com/security/trusty/`

[4] `https://developer.android.com/about/versions/android-4.3.html`

> Android also now supports hardware-backed storage for your KeyChain credentials, providing more security by making the keys unavailable for extraction. That is, once keys are in a hardware-backed key store (Secure Element, TPM, or TrustZone), they can be used for cryptographic operations but the private key material cannot be exported. Even the OS kernel cannot access this key material. While not all Android-powered devices support storage on hardware, you can check at runtime if hardware-backed storage is available by calling `KeyChain.IsBoundKeyAlgorithm()`.

Furthermore, the latest versions of the Android KeyStore allow their keys to be stored in hardware.[5]

In the application itself this can be verified using `KeyInfo.isInsideSecureHardware()`

Verifying a Hardware-backed Key Pair Android helpfully explains how to verify if the key is stored in hardware[6]. Specifically Android provides a method, `getCertificateChain` [7] to verify the certificate chain. If such a certificate chain contains a root certificate that attest that the key was generated using hardware (e.g. Samsung devices with Samsung KNOX[8])

On interesting detail listed is that the root certificate of keys generated Android 7.0 phones with hardware-level key attestation, are signed with the Google attestation root key[9]. If such a key can be verified with Google root certificate, the mDL does not need to know the root certificates for specific manufacturers.

Example code of how to implement a key attestation check can be found here : `https://github.com/googlesamples/android-key-attestation`

Requiring User Authentication For Key Use form the KeyStore It is possible to require user authenticating[10] to use to the keys. The authentication can be done using the same met hods as the lock screen or using

[5] `https://source.android.com/security/keystore/`

[6] `https://github.com/googlesamples/android-key-attestation`

[7] `https://developer.android.com/reference/java/security/KeyStore.html#getCertificateChain%28java.lang.String%29`

[8] `https://www.samsungknox.com/en`

[9] `https://developer.android.com/training/articles/security-key-attestation.html#verifying`

[10] `https://developer.android.com/training/articles/keystore.html#UserAuthentication`

fingerprint authentication (if the hardware permits it). While this adds a layer of security, since merely having physical acces to a phone is not enough to use the mDL application, it impacts usability noticeable.

- The lock screen has to be enabled *globally* (i.e you can't have a lock screen just for the mDL)
- There is a "feature"[11], that *deletes* your keys that require user authentication if you change your lock screen. [12]

ARM TrustZone The ARM Trustzone is a System on Chip included with recent ARM processors frequently used on smartphones. The usage of ARM trustzone would be the recommended method of ensure hardware bound key usage. Because the ARM TrustZone already present on many (future) holder devices, we would not have to require holder's to use new hardware. Additionally, the TrustZone processor is generally faster than other secure elements like SIM cards. This is particularly noteworthy since slow authentication times could hamper adoption.

A major downside is that the ARM TrustZone is not designed to be tamper proof. On the other hand tampering of the holder with the TrustZone has the potential to break his own device, so there is noticeable incentive not to tamper with it.

SmartSD One method of implementing a secure element would be using a SmartSD[13]. The SmartSD can run java-card applications and comes with builtin NFC capabilities.

The downside here is that new hardware is needed and the holder has to have a spare SD card slot.

SIM Card SIM cards could also be used as a secure element. However, re-issuing each mDL holder a new SIM card (replacing their current one) may not be feasible.

Software based mitigations

[11] `https://doridori.github.io/android-security-the-forgetful-keystore/`

[12] There is a "fix" in android 6.0, You can change your pincode or swap pattern but have to delete your keys if you want to change the lock method

[13] `https://www.sdcard.org/developers/overview/ASSD/smartsd/`

Environment checks A practical way to mitigate the cloning of an mDL is to use environment checks. Since these are all done in software and holder's can have root access to their devices, the results of the environment checks can be falsified. On the other hand many of the following checks can be programmed easily and would not impact usability significantly.

Device identifiers Checking the device the mDL runs on allows the mDL to detect when it is being run on new hardware. This would mean that to clone a working mDL application, an attacker not only would have to copy the application and its local data, but would also have to copy and spoof the source device's fingerprint.

On Android, I recommend the `ANDROID_ID` [14] Identifier. This identifier is generated when a user first sets up his device and by itself does not reveal any private information about the holder.

Additionally, the following variables could be checked, however unlike `ANDROID_ID` these may reveal private information.

- Build.BOARD
- Build.BRAND
- Build.DEVICE
- Build.DISPLAY
- Build.FINGERPRINT
- Build.HARDWARE
- Build.MANUFACTURER
- Build.MODEL
- Build.PRODUCT
- Build.SERIAL

When contemplating if other Identifiers are useful, keep in mind that no identifier should change in the lifetime of the mDL. For instance the language setting could be used as an additional identifier on most devices but is something that can be expected to change in the future on at least some devices.

[14] `https://developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID`

Potentially unwanted environments To slow down reverse engineering, it is possible to add a few extra hurdles.

Emulator use Running applications is a valuable tool in both developing and reverse engineering applications. However, during normal use, the mDL is not expected to run in an emulator.

Comparing the identifier to that of common emulators is a simple method of checking if the application is run in an emulator.

```
  public static boolean isEmulator() {
    return Build.FINGERPRINT.startsWith("generic")
            || Build.FINGERPRINT.startsWith("unknown")
            || Build.MODEL.contains("google_sdk")
            || Build.MODEL.contains("Emulator")
            || Build.MODEL.contains("Android SDK built for x86")
            || Build.MANUFACTURER.contains("Genymotion")
            || (Build.BRAND.startsWith("generic") &&
                Build.DEVICE.startsWith("generic"))
            || "google_sdk".equals(Build.PRODUCT);
}
```

[15]

Android debug bridge adb (Android Debug Bridge) is command line utility that allows users to easily manage their android device, generally via a USB cable. It is not only useful for developing and reverse engineering but is also useful for personal use. It is particularly useful for file management (transferring files and making backups). Because of this, it is not very unusual to have adb enabled.

Ether adb is enabled can be check using `Settings.Global.ADB_ENABLED` [16]

However, this is not recommended, a layman can bypass this by quickly toggling the option in the settings menu and it can be expected to be enabled during normal use.

Root Checking for common methods of root access is relatively easy. A simple snippet can be found quite easily. [17]

```
/** @author Kevin Kowalewski */
public class RootUtil {
```

[15] Shamelessly stolen from `https://stackoverflow.com/questions/2799097/how-can-i-detect-when-an-android-application-is-running-in-the-emulator`

[16] `https://developer.android.com/reference/android/provider/Settings.Global.html#ADB_ENABLED`

[17] `https://stackoverflow.com/questions/1101380/determine-if-running-on-a-rooted-device`

```
    public static boolean isDeviceRooted() {
        return checkRootMethod1() || checkRootMethod2() || checkRootMethod3();
    }

    private static boolean checkRootMethod1() {
        String buildTags = android.os.Build.TAGS;
        return buildTags != null && buildTags.contains("test-keys");
    }

    private static boolean checkRootMethod2() {
        String[] paths = { "/system/app/Superuser.apk", "/sbin/su", "/system/bin/su", "/syste
                "/system/bin/failsafe/su", "/data/local/su", "/su/bin/su"};
        for (String path : paths) {
            if (new File(path).exists()) return true;
        }
        return false;
    }

    private static boolean checkRootMethod3() {
        Process process = null;
        try {
            process = Runtime.getRuntime().exec(new String[] { "/system/xbin/which", "su" });
            BufferedReader in = new BufferedReader(new InputStreamReader(process.getInputStre
            if (in.readLine() != null) return true;
            return false;
        } catch (Throwable t) {
            return false;
        } finally {
            if (process != null) process.destroy();
        }
    }
}
```

However, this is not a recommended check. Root access is common and a policy refusing to allow user to have root access to their own devices is quite dystopian.

Obfuscation Software based obfuscation, unlike hardware-based obfuscation provides little advantage. Software and the locally stored data is easily copied. Even if the application tried to detect a change in hardware, any identifier that can be checked using software (e.g. IMEI, MAC address, Model number, etc.) can easily be copied and spoofed on another device.

White-box cryptography To obfuscate the holder's signing key from the holder, The use of white-box cryptography is ill-advised.

- White-box cryptographic schemes are slower than their unobfuscated counterparts. This is a serious problem since authentication should not take too long for the mDL to remain practical.

- While obfuscation can increase the time needed to extract the key, it does not prevent a persistent attacker from deobfuscating. The same signing key is usable for an extended period and the mDL may be valuable if usable as a legal identification. This combination may make it reasonable for an attacker to invest sufficient time to undo the obfuscation.

- In order to prevent the signing function from being used outside the mDL, the signing function would have to be tightly integrated with the rest of the mDL. This requires to entire mDL application to be obfuscated, which likely leads to slower execution, larger file size and increased complexity.

- There are no well-known public key white-box implementations. All publicized white-box cryptography schemes are based on symmetric key cryptography. Even if a symmetric key algorithm is used at store the signing key while not in use, if the key is not obfuscated in memory while it is being used, it could reasonably be extracted by memory scraping.

- The security benefits are questionable. Gray box cryptanalysis not only applicable, but is often more efficient in breaking white box cryptography. [] [] [] [] Since Observations and fault injections can be done in software, unlike gray box cryptography, there is no noise in the measurements. Because of this lack of noise, far fewer measurements, and thus time, is needed to break white box cryptographic schemes.

- There is no trivial way to prevent code lifting. Preventing the signing function from being cloned would require the signing function not only to be tightly integrating with the mDL but

8.2.4 Remaining risks

Even after implementing all the previously mentioned mitigations there will still be some risks that could lead to the disclosure of information that needs to be secure.

In particular, we need to take into account that face matching is not guaranteed to be perfect. Leeway in matching faces is needed because a persons face can change over time (different lighting conditions, aging, injuries, etc.), so one face image could occasionally match two people.

Physical sharing of primary device While binding keys to hardware would prevent an attacker from using the same mDL on another device. It does however not prevent an attacker from using the mDL storing device of a valid mDL holder to authenticate himself. The attacker could get the device either through theft or by the user lending out the device. If the holder cooperates with the attacker, or if the holder neglects to report its theft, the unwanted usage of this mDL can remain undetected.

Since this attack requires the physical possession of the device the damage is limited only a single attacker can authenticate as a specific mDL holder who is similar enough in appearance that the face matching fails.

However, physical drivers licenses also have the same exact problem. In that regard the mDL is as secure as the physical drivers license.

Sharing of secondary devices If the issuer allows a holder to use his mDL on multiple devices, the holder could lend out on of his devices to a lookalike.

More dangerously, a malicious holder could hoard several devices before obtaining his device for the only purpose of lending/selling these devices once the mDL is installed on it.

Relaying of authentication An attacker could eliminate the need to have the mDL device present by forwarding any mDL challenge it gets to an mDL device. If that device is set up such that it will automatically respond to the challenge then the non-mDL device will appear as if it were the mDL device to any reader.

Disclosure of hardware stored keys In theory hardware-stored keys are not extractable from the hardware. If this turns out not to be the case in practice, for instance through a bug in the TEE, then an mDL holder could extract its own key. With this key the holder could copy the mDL.

Disclose of root certificate signing keys If the root certificate signing keys are not properly secured an attacker would be able to sign its own keys and pretend they are stored in hardware. After that, a malicious mDL holder could share his mDL with many other users. If the signing keys are disclosed to multiple people, multiple users could share their mDL with arbitrary users.

References

8.3 Mdl Initialization protocol

8.3.1 Data Structure

mDL metadata

Issuer identifier

Public key

1. Nonce signing key
2. Token signing key

Biometric

mDL expiry date

Attribute responses Each attribute response contains

- attribute_name
- attribute_value
- attribute_epiry_date
- Signature over the concatination of above and mDL metadata

Private Keys

Nonce signing key

Issuer connection key

Holder/Issuer communication key

Chip authentication key

8.3.2 New variables to be stored by Issuer

Public keys

Nonce signing key

Issuer connection key

Holder/Issuer communication private key

Chip authentication key

8.3.3 assumptions

- Issuer already knows the biometrics and attributes of the holder
- There exists a channel such that
  - The information send is not eavesdropped
  - The information send is not modified
  - The holder can confirm it is talking to the issuer
  - The issuer can confirm it is talking to the holder

8.3.4 PROTOCOL:

Pre-connection phase

HOLDER: Generate Nonce Signing keypair

HOLDER: Generate Token Signing keypair

HOLDER: Generate Holder/Issuer communication keypair

HOLDER: Generate Chip authentication keypair

HOLDER: Generate certificate chains for each key (Optional)

HOLDER: Verify certificate chains (Optional)

Sending data from holder to Issuer

HOLDER: Send Nonce signing public key

HOLDER: Send Nonce signing public certification chain

HOLDER: Send Issuer connection public key

HOLDER: Send Issuer connection public certification chain

HOLDER: Send Update public key

HOLDER: Send Update public certification chain

HOLDER: Send Chip authentication public key

HOLDER: Send Chip authentication public certification chain

Verification of holder given data

ISSUER: For every key

- Verify that it is a properly formatted key
- Verify that the certificate chain is valid
  - Verify that the root of the certificate chain is a known trusted certificate
  - Verify that the certificate chain belongs to the users public key

Generate mdL data

Generate mDL header

Generate attribute responses

Sending data from Issuer to holder

mDL Header

Attribute responses

Verification of issuer given data

8.3.5 Formats

mDL header format `ISSSUER_ID|ISSUER_PUBLIC_KEY|HOLDER_BIOMETRIC|HOLDER_AA_KEY|HOLDE`

Attribute response format `ATTRIBUTE_NAME|ATTRIBUTE_VALUE|ATTRIBUTE_SIGNATURE`

Format for sending public key

Format for sending certificate chain

8.3.6 Expections

Missing certificate chain

8.3.7 Samples

Generating key stored in hardware

Extracting certificate chain of key

Signing using key

Verifiying certificate chain

```
cat cert3.pem cert2.pem cert1.pem > concat.pem
openssl verify concat.pem
```

Verifiying signed nonce

8.4 Test applications

8.4.1 Secure storage test

```
package com.example.terp.keystoragetest;

import android.content.Context;
import android.security.keystore.KeyGenParameterSpec;
import android.security.keystore.KeyInfo;
import android.security.keystore.KeyProperties;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
```

```
import android.widget.TextView;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.InterruptedIOException;
import java.io.UnsupportedEncodingException;
import java.io.Writer;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.SignatureException;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.spec.InvalidKeySpecException;


//**REQUIRES ANDROID 7 OR NEWER//

// specifically the method getCertificateChain does not exist in older
// version of android. Older version of android may be able to store keys in hardware, but

public class MainActivity extends AppCompatActivity
{

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    // Name to reference our key in the keystore. The key is only
    // usable by application that created it, so benefit of named keys
    // is to manage multiple keys used by a single application
    String hardware_keyname = "key1";
```

```
// Commend the lines with RSA, and uncomment the lines with ECDSA
// to make the application use eliptic curves.
String signature_algorithm = "SHA256withRSA/PSS";
//    String signature_algorithm = "SHA256withECDSA";

String keystore_name = "AndroidKeyStore";


// The following functions get called when the user presses a
// button in the UI and provide feedback by modifying the text in
// the texview object call "MainText".

void CreateHardwareKey(View v)
{
    TextView t = (TextView) findViewById(R.id.MainText);
    String txt = "";
    txt += "\nStep 1\n";
    try {
        // First we create a key pair generator
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.
            getInstance(KeyProperties.KEY_ALGORITHM_RSA,
                        keystore_name);
        // KeyPairGenerator keyPairGenerator = KeyPairGenerator.
        // getInstance(KeyProperties.KEY_ALGORITHM_EC, keystore_name);
        txt += "\nStep 2\n";

        // An attestationchallenge has to be specified if a key is
        // generated in hardware, it can be used to verify that a
        // key was generated based on a request. This value is not
        // checked later so this is just 4 made up bytes.
        byte[] attestationchallenge = new byte[] { (byte)0xAA,
                                                   (byte)(0xBB),
                                                   (byte)(0xCC),
                                                   (byte)(0xDD),
        };


        // setAttestationChallenge is essential, without it the
        // key is not stored in hardware
        KeyGenParameterSpec keygenparameterspec = new KeyGenParameterSpec.
            Builder(hardware_keyname, KeyProperties.PURPOSE_SIGN)
            .setDigests(KeyProperties.DIGEST_SHA256,
                        KeyProperties.DIGEST_SHA512)
            .setSignaturePaddings(KeyProperties.SIGNATURE_PADDING_RSA_PSS)
            .setAttestationChallenge(attestationchallenge)
            .build();
        keyPairGenerator.initialize(keygenparameterspec);
        txt += "\nStep 3\n";

        // Only at this point is the actual key pair generated
        KeyPair keyPair = keyPairGenerator.generateKeyPair();
        txt += "\nStep 4\n";
```

```
            // We create a new uninitialized object for signing
            Signature signature = Signature.getInstance(signature_algorithm);
            txt += "\nStep 5\n";

            // We initialize the signing object with our freshly
            // generated keypair
            signature.initSign(keyPair.getPrivate());
            txt += "\nStep 6\n";
            txt += ">>\n";

            // We check if the attestation challenge is present, if
            // not this throws an error.
            txt += keygenparameterspec.getAttestationChallenge();
            txt += "<<\n";
        } catch (NoSuchAlgorithmException |
                 NoSuchProviderException |
                 InvalidKeyException |
                 InvalidAlgorithmParameterException e) {
            txt += "\nERROR [[" + e.toString() + "]]\n";
            e.printStackTrace();
        }
        txt += "\nStep 7\n";
        t.setText(txt);
    }

    void ReadHardwareKey(View v)
    {
        TextView t = (TextView) findViewById(R.id.MainText);
        String txt = "";
        try {
            // Load the keystore
            KeyStore keyStore = KeyStore.getInstance(keystore_name);
            keyStore.load(null);
            // Load the certificate chain of our hardware stored
            // keylist. This is null if there is no certificate chain.
            Certificate[] chain = keyStore.
                getCertificateChain(hardware_keyname);
            keyStore.getKey(hardware_keyname, null);

            // Try to obtain private key. We should expect null to be
            // returned. If we do somehow obtain the private key we
            // know it wasn't stored properly.
            PrivateKey privateKey = (PrivateKey) keyStore.
                getKey(hardware_keyname, null);
            txt += "Privatekey encoded: "+privateKey.getEncoded()+ "\n";


            PublicKey publicKey = keyStore.
                getCertificate(hardware_keyname).
                getPublicKey();
            KeyFactory factory = KeyFactory.
                getInstance(privateKey.getAlgorithm(),
                            keystore_name);
```

```
KeyInfo keyInfo;
keyInfo = factory.getKeySpec(privateKey, KeyInfo.class);


// Check if the OS thinks the key is in hardware.

// This check is only trustworthy if we can assume the OS
// is not altered.
if (keyInfo.isInsideSecureHardware()) {
    txt += "The key *IS* inside secure hardware\n";
} else {
    txt += "The key is *NOT* inside secure hardware\n";
}


// Loop over the certificate chain, reading every
// certificate and writing it to a file.

// Validity of the certificates is not checked on the device right
int n = 0;
String[] certname = {"hardware","intermediate","root" };
for (Certificate cert : chain) {
    String filename_crt = "certificate_encoded_" +
        certname[n] + ".crt";
    String filename_txt = "certificate_plaintext_" +
        certname[n] + ".txt";
    File file_crt = new File(getExternalCacheDir(), filename_crt);
    File file_txt = new File(getExternalCacheDir(), filename_txt);
    try {
        // Write encoded certificate to file
        if(file_crt.exists())
            file_crt.delete();
        file_crt.createNewFile();
        FileOutputStream outputStream_crt;
        outputStream_crt = new FileOutputStream(file_crt);
        outputStream_crt.write(cert.getEncoded());
        outputStream_crt.flush();
        outputStream_crt.close();

        // Write plaintext representation to file
        if(file_txt.exists())
            file_txt.delete();
        file_txt.createNewFile();
        FileOutputStream outputStream_txt;
        outputStream_txt = new FileOutputStream(file_txt);
        outputStream_txt.write(cert.toString().getBytes());
        outputStream_txt.flush();
        outputStream_txt.close();

        //Show plaintext version on screen
        txt += "Certificate " + Integer.toString(n) +
            " :\n" + cert.toString() + "\n";
        n+=1;
```

```
            } catch (Exception e) {
                e.printStackTrace();
                txt += e.toString();
            }
        }
        txt += "Written certificate files to  " +
            getExternalCacheDir().toString() + "\n";
        txt += "\n\n\n\n\n\n\n\n\n\n\n"; // Fixes some text being cut off
    } catch (IOException |
             CertificateException |
             NoSuchAlgorithmException |
             UnrecoverableKeyException |
             KeyStoreException |
             NoSuchProviderException |
             InvalidKeySpecException e) {
        txt += "\nERROR [[" + e.toString() + "]]\n";
        e.printStackTrace();
    }
    t.setText(txt);


}

void ShowMDLData(View v)
{
    // After the mDL has been personalized, a file containing the
    // metadata of that mDL is written in plaintext to
    // "mdl_header.txt"

    // This function checks if that file exists and displays its
    // contents if it exists.

    // The personalization should also add a file
    // "mdl_header.txt.sig", a signature over the header file,
    // however this is not verified by the holder.

    TextView t = (TextView) findViewById(R.id.MainText);
    String txt = "Reading mDL data";
    String filename_header = "mdl_header.txt";
    File file_header = new File(getExternalCacheDir(), filename_header);

    if(file_header.exists()) {
        try {
            final InputStream inputStream =
                new FileInputStream(file_header);
            final BufferedReader reader =
                new BufferedReader(new InputStreamReader(inputStream));

            final StringBuilder stringBuilder = new StringBuilder();

            boolean done = false;

            while (!done) {
                final String line = reader.readLine();
```

```
                done = (line == null);

                if (line != null) {
                    stringBuilder.append(line);
                    stringBuilder.append("\n");
                }
            }

            reader.close();
            inputStream.close();

            txt = stringBuilder.toString();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    } else {
        txt = "mDL is not yet initialized";
    }
    txt += "\n\n\n\n\n\n\n\n\n\n\n\n\n\n"; // Fixes some text being cut off
    t.setText(txt);
}

void ShowAttributes(View v)
{
    // After initialization, the issuer script will write a file
    // to storage for every attribute issued. This function will
    // look if any known attribute file exists and output it.

    // The holder does not verify the signatures

    String[] attribute_names = {"birthdate", "firstname", "lastname" };
    TextView t = (TextView) findViewById(R.id.MainText);
    String txt = "";
    for (String attribute_name : attribute_names) {

        String filename = "attribute_"+ attribute_name + ".txt";
        File file = new File(getExternalCacheDir(), filename);

        if(file.exists()) {
            try {
                final InputStream inputStream = new FileInputStream(file);
                final BufferedReader reader =
                    new BufferedReader(new InputStreamReader(inputStream));

                final StringBuilder stringBuilder = new StringBuilder();

                boolean done = false;

                while (!done) {
                    final String line = reader.readLine();
                    done = (line == null);
```

```
                    if (line != null) {
                        stringBuilder.append(line);
                        stringBuilder.append("\n");
                    }
                }

                reader.close();
                inputStream.close();

                txt += stringBuilder.toString();
                txt += "\n\n";
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            txt += "*** MISSING ATTRIBUTE: " + attribute_name + " ***\n";
        }

    }

    txt += "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n";  // Fixes some text being cut off
    t.setText(txt);
}


void SignChallenge(View v)
{
    // The user can input a number into the field "challenge"

    // This function will sign the string representation of that
    // number (e.g. 0x31,0x32 for the number "12") and write that
    // signature to a file.

    TextView t = (TextView) findViewById(R.id.MainText);
    TextView challenge_textview = (TextView) findViewById(R.id.challenge);
    String challenge = challenge_textview.getText().toString();
    String txt = "";
    txt += "Signing the challenge " + challenge + " ...\n";

    byte[] challenge_bytes = new byte[0];
    try {
        challenge_bytes = challenge.getBytes("UTF-8");
        String challenge2= new String(challenge_bytes, "UTF-8");
        txt += "Signing the challenge " + challenge2 + " ...\n";
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }

    KeyStore keyStore = null;
    try {
        keyStore = KeyStore.getInstance(keystore_name);
        keyStore.load(null);
```

```
            PrivateKey privateKey = (PrivateKey) keyStore.
                getKey(hardware_keyname, null);
            Signature signature = Signature.getInstance(signature_algorithm);
            signature.initSign(privateKey);
            signature.update(challenge_bytes);
            byte result[] = signature.sign();

            String filename = "response";
            File file = new File(getExternalCacheDir(), filename);
            if(file.exists())
                file.delete();
            file.createNewFile();
            FileOutputStream outputStream;
            outputStream = new FileOutputStream(file);
            outputStream.write(result);
            outputStream.flush();
            outputStream.close();

            txt += "Written response to  " +
                getExternalCacheDir().toString()+ "/" +
                filename + "\n";


        } catch (KeyStoreException e) {
            e.printStackTrace();
        } catch (CertificateException e) {
            e.printStackTrace();
        } catch (UnrecoverableKeyException e) {
            e.printStackTrace();
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InvalidKeyException e) {
            e.printStackTrace();
        } catch (SignatureException e) {
            e.printStackTrace();
        }


        txt += "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n";  // Fixes some text being cut off
        t.setText(txt);
    }

}
```

8.4.2 Issuer script

```
#!/bin/sh

# USAGE: ./issue_mdl.sh USERID
# This script requires the phone to be connected via a usb cable,
# adb to be enabled on the phone and a host that has openssl and adb software.
```

```
# users are folders in the folder "users", attributes and biometrics
# are simply files inside the users folder.

# Right now only "test_user" is included, so the command
# ./issue_mdl.sh test_user
# Will issue an simple mDL

# Quit script when an error occurs
set -e

USER=$1

# Change this to the location where this script is located
ISSUERPATH="/media/sf_vmshare/issuer"

USERPATH=$ISSUERPATH"/users/"$USER
# Test if user exists
ls -d $USERPATH/attributes > /dev/null
ls -d $USERPATH/biometric > /dev/null
# Location where files are stored on the phone
MDLDATAPATH="/storage/self/primary/Android/data/com.example.terp.keystoragetest/cache"

echo "==DOWNLOADING KEY FROM PHONE=="
# Download all files to a temporary directory
rm -fr /tmp/cache
adb pull $MDLDATAPATH /tmp/cache
cd /tmp/cache

echo "==VERIFYING KEY FROM PHONE=="
# Convert certificates to PEM format
openssl x509 -inform der -in "certificate_encoded_hardware.crt"\
        -out "hardwarekey.pem"
openssl x509 -inform der -in "certificate_encoded_intermediate.crt"\
        -out "intermediate.pem"
openssl x509 -inform der -in "certificate_encoded_root.crt"\
        -out "root.pem"
# Concatinate the certificates, this is what openssl expect when veryfing
cat root.pem intermediate.pem hardwarekey.pem > concat.pem

# Verify the certificate chain when we trust the root certificarte
openssl verify -CAfile root.pem concat.pem

# Verify using a trusted root certificate
openssl verify -CAfile $ISSUERPATH/certificates/google-root.pem concat.pem
# Right now google-root.pem is copied from a phone

echo "==BUILDING HEADER=="
# Create a file with the essential metadata concatinated
cat $ISSUERPATH/keys/public_key.pem > mdl_header.issuer
cat $USERPATH/biometric/face.jp2 > mdl_header.biometric
cat hardwarekey.pem > mdl_header.hardwarekey
date > mdl_header.expiry        # For testing purposes this just the current date and time
```

```
cat mdl_header.issuer mdl_header.biometric mdl_header.hardwarekey mdl_header.expiry > mdl_hea
echo "==SIGNING HEADER=="
# Sign the header file
openssl dgst -sha256 -binary -sign $ISSUERPATH/keys/private_key.pem -out mdl_header.txt.sig m
# Sanity check: verify the signature with our own public key
openssl dgst -sha256 -verify $ISSUERPATH/keys/public_key.pem -signature mdl_header.txt.sig  m


echo "==SIGNING ATTRIBUTE RESPONSES=="
# Loop over the names of all files in the attributes folder
for file in $USERPATH/attributes/*
do
    # Strip the file path, leaving only the attribute name
    ATTRIBUTE="$(echo $file | sed -s 's,^.\+/\([^/]\+\),\1,')"
    echo $ATTRIBUTE

    # Create a file with our attribute information
    cp $USERPATH/attributes/$ATTRIBUTE "attribute_"$ATTRIBUTE".txt"
    echo -n "epiry_date:" >>  "attribute_"$ATTRIBUTE".txt"
    date >> "attribute_"$ATTRIBUTE".txt"
    # Sign our created file
    openssl dgst -sha256 -binary -sign $ISSUERPATH/keys/private_key.pem -out "attribute_"$ATT
    # Sanity check, verify own signature
    openssl dgst -sha256 -verify $ISSUERPATH/keys/public_key.pem -signature "attribute_"$ATTR
done

echo "==UPLOADING DATA TO PHONE=="
# Now copy all files , including the newly created ones back to the phone
adb push /tmp/cache/* $MDLDATAPATH/

# Not cleaning up for now to make debugging easier
# echo "==CLEANING UP=="
# rm -fr /tmp/cache

echo "==END OF SCRIPT=="
```

8.4.3 Verification script

```
#!/bin/sh

# USAGE: ./readout_mdl.sh CHALLENGE
# The response to this challenge should be written to a file by the holder.

# Quit script when an error occurs
set -e

CHALLENGE=$1
ISSUERPATH="/media/sf_vmshare/issuer"
MDLDATAPATH="/storage/self/primary/Android/data/com.example.terp.keystoragetest/cache"

echo "==DOWNLOADING DATA FROM PHONE=="
rm -fr /tmp/cache
adb pull $MDLDATAPATH /tmp/cache
```

```
cd /tmp/cache

echo "==VERIFYING MDL HEADER SIGNATURE=="
# Right now we assume there is only 1 Issuer, so we verify against the only known one.
cat mdl_header.issuer mdl_header.biometric mdl_header.hardwarekey mdl_header.expiry > mdl_hea
diff mdl_header.txt mdl_header.concat
openssl dgst -sha256 -verify $ISSUERPATH/keys/public_key.pem -signature mdl_header.txt.sig  m
# Only once we have verified the header data can we trust the mDL holders signing key.

echo "==VERIFYING ATTRIBUTE SIGNATURES=="
for file in /tmp/cache/attribute_*.txt
do
    openssl dgst -sha256 -verify $ISSUERPATH/keys/public_key.pem -signature $file.sig  $file
done

echo "==VERIFYING  RESPONSE TO NONCE=="
# Write out own challenge to a file
echo -n $CHALLENGE > challenge
openssl x509 -pubkey -noout -in mdl_header.hardwarekey > pub.pem

# Verify that the holder supplied signature corresponds to our chose challenge
openssl dgst -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -verify pub.pem -signatu

# Below is the verification if the signing key uses ECDSA
# openssl dgst -ecdsa-with-SHA256 -verify pub.pem -signature response challenge

echo "==MDL OVERVIEW=="
# Convert the face image to png so we can easily view it
mv /tmp/cache/mdl_header.biometric /tmp/cache/face.jp2
convert face.jp2 -resize 256x256 face.png
# Show face image
feh face.png                    # feh is a image viewing program
echo -n "MDL Expires at :"
cat /tmp/cache/mdl_header.expiry
echo -e "\n"
echo "MDL Signing key"
cat /tmp/cache/mdl_header.hardwarekey
echo "Issuer public key"
cat /tmp/cache/mdl_header.issuer

echo "==ATTRIBUTES=="
for file in /tmp/cache/attribute_*.txt
do
    cat $file
    echo ""
done

# Not cleaning up for now to make debugging easier
# echo "==CLEANING UP=="
# rm -fr /tmp/cache
echo "==END OF SCRIPT=="
```