AN12838

Strengthening Public Key Cryptography using CAAM Secure Key

Rev. 0 — June 2020

Application Note

Contents

1 Introduction..... 1

2 References..... 1

3 Overview..... 1

4 Implementation..... 3

4.1 RSA..... 3

4.2 ECDSA..... 5

5 Hands-on..... 7

5.1 Build..... 7

5.2 Usage..... 8

6 Conclusion..... 10

1 Introduction

Public Key Cryptography is a solid tool which ensures the transfer of confidential data upon insecure channels. Digital signature as one of the applications of public key cryptography ensures the identity of the signer and integrity of the signed data, hence the security of the private key is crucial.

In this work, we consider a scenario in which the software environment gets compromised due to attacks leading to the possibility of leaking the private key. We consider also the situation where the user does not simply want to make the key visible to software but rather keep it only known to hardware. The scheme we propose is based on a feature provided by the i.MX application processors called Black Key.

2 References

- i.MX Security Reference Manual, downloadable from nxp.com portal
- *i.MX Yocto Project User's Guide* (document IMXLXYOCTOUG)
- Linux mainline kernel archive, 2018 https://kernel.org

3 Overview

Cryptographic Accelerator and Assurance Module (CAAM) is a hardware module built-in i.MX SoCs which implements secure RAM and a dedicated public-key hardware accelerator (PKHA). It supports ECDSA signature generation and verification operations and elliptic curve key-pair generation. It also supports RSA encryption and decryption operations and key-pair generation.

A device specific random 256-bit One Time Programmable Master Key (OTPMK) key is fused in each i.MX processor at manufacturing time, this key is unreadable and can only be used by the CAAM for AES encryption/decryption of user data, through the Secure Non-Volatile Storage (SNVS) companion block. This mechanism can be used to encrypt the ECC and RSA private keys and turn them into a data structure called a **Black Key**.

As defined in the SRM, the black key scheme is intended for protection of user keys against bus snooping while the keys are being written to or read from memory external to the SoC. The keys are automatically encapsulated and decapsulated on-the-fly. The scheme also prevents key cloning, a black key generated on a specific device can only be used in the device where the key has been generated.

CAAM supports two different black key encapsulation schemes which are AES-ECB and AES-CCM. While AES-ECB encryption is intended for quick decryption, AES-CCM encryption is intended for high assurance. Regarding AES-ECB encryption, data is a multiple of 16 bytes long. If the private key is a multiple of 128-bit, as it will be if we use P-256 or P-384 curve for ECC, or 2048-bit RSA private key, then the AES-ECB encrypted private key would fit in the same buffer as the original unencrypted private key. If the private key is not a multiple of 16 bytes long, as it will be if we use P-521 for ECC, then it is padded before being encrypted. A CCM-encrypted black key is always at least 12 bytes longer than the encapsulated private key (the 12 bytes are for the storage of the nonce and of the IV and in addition as the CCM mode is an authenticated encryption mode, additional bytes are to be foreseen to store the authentication tag for the verification of the integrity and the authenticity of the encrypted data). CCM-

encrypted keys are preferred, unless we need the encrypted key to fit in the same space as the unencrypted key. Figure 1 illustrates the black key encapsulation and decapsulation mechanisms.

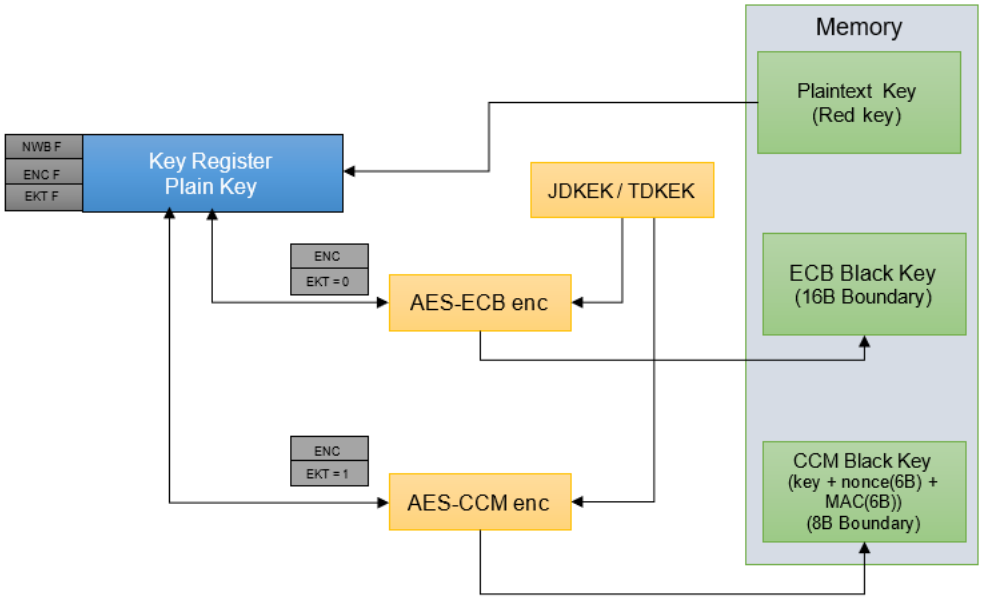

Figure 1. Encapsulating and decapsulating CAAM black keys

Black keys are loaded using CAAM's KEY command with ENC flag set.

The EKT bit defines the black key encapsulation scheme. If it is set, then AES-CCM is used to decrypt the black key otherwise AES-ECB is used.

A plaintext key can be converted to a black key, assume that we generated an RSA key and we want to turn the private key in the format (n, d) to a black key. Then, the private exponent d is loaded to CAAM PKHA E register, writing it back to memory will be in an encrypted form.

Black keys are session keys thus, are not power-cycles safe. CAAM's blob mechanism provides a method for protecting user-defined data across system power cycles. The black private key to be protected, is re-encrypted so that it can be safely placed into non-volatile storage before the SoC is powered down. The mechanism provides both confidentiality and integrity protection. Figure 2 illustrates the CAAM blob mechanism.

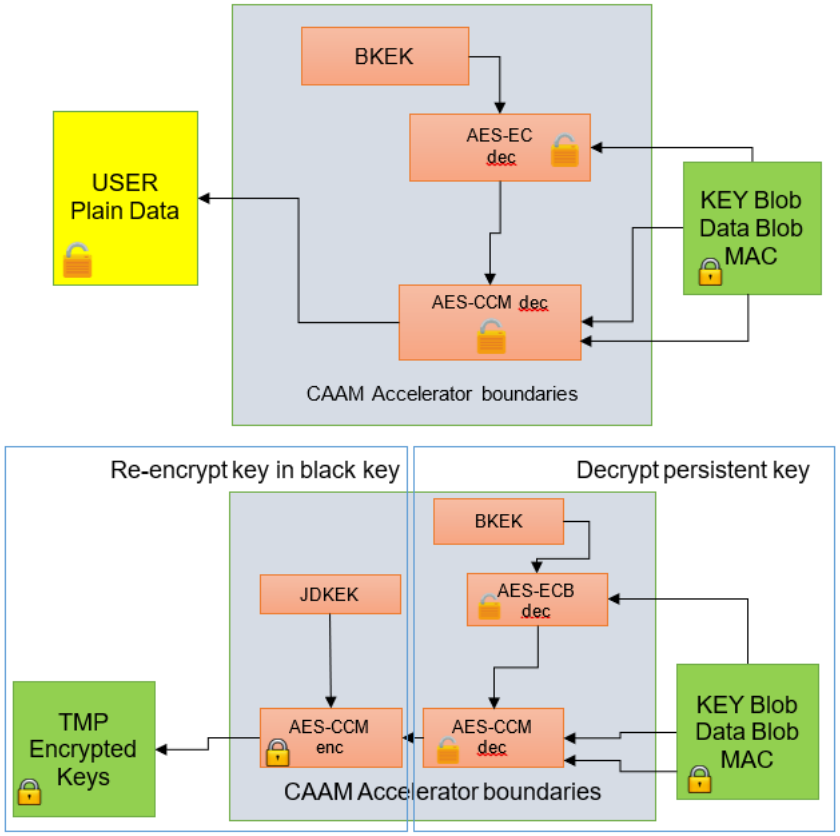

Figure 2. CAAM blob mechanism

CAAM implements operations that converts between blob encapsulation and black-key encapsulation without exposing the key in plaintext.

Finally, the CAAM Secure Memory can be used as confidentiality-preserving memory that protects the private key. We could imagine a scenario in which, a user wants to generate a black RSA key. First, two primes should be generating then the CAAM RSA finalize key generation function is called to perform primality checks and compute the private exponent, modulus and remaining CRT values. This flow involves having the primes visible even for a short time for the software. To protect any intermediate value against malicious software, secure memory can be used to store these values until the final black key is generated.

4 Implementation

The software presented in this work, utilizes all CAAM features described in the previous section by building a descriptor and then adding this descriptor to a Job Ring. The following sections illustrates the job descriptors used for various operation such as private decryption using black key (RSA signing), ECDSA signing using black key, ECC key-pair generation with blackened private key and converting a plain RSA key to a black key.

4.1 RSA

For RSA we provide job descriptors to perform decryption using private key which could be used also for signing data, exporting a black key into a blob and import a blob into a black key.

Regarding RSA key generation, a job descriptor which converts a plain key to a black key is illustrated.

In ECC a black key is generated directly on hardware. The same of RSA could be achieved by replacing the RSA Finalize Key Generation (RFKG) but it is no implemented in this work.

4.1.1 Wrapping a Plain RSA private key to a Black RSA private key

Assume we generated an RSA key-pair and we want to wrap the private key (n, d) into a Black key. The operations would be:

- Load d to PKHA E
- Store PKHA E to memory (encrypted version of d)

To go more in detail, the private exponent "d" should be loaded to CAAM's PKHA E memory. The PKHA E Size Register is used to indicate the size of the private exponent that will be loaded into or unloaded from the PKHA E Memory.

The PKHA E Size Register must be written before the private exponent is written into or read from the PKHA E Memory. This will reserve the PKHA for the current job. Since the PKHA E Size Register is automatically written by the KEY Command, then we load the private exponent to PKHA E memory using the KEY command.

FIFO STORE commands are used to move data from the output data FIFO to external memory by means of the DMA.

PKHA E Memory is one of data that can be output from the output data FIFO. This data is encrypted as a Black Key prior to being written to memory.

The output data type for PKHA E can be one of the following:

- PKHA E, encrypted using AES-ECB with the job descriptor key-encryption key (Bit 22).
- PKHA E, encrypted using AES-ECB with the trusted descriptor key-encryption key (Bit 23).
- PKHA E, encrypted using AES-CCM with the job descriptor key-encryption key (Bit 12).
- PKHA E, encrypted using AES-CCM with the trusted descriptor key-encryption key (Bit 13).

Following is an example of Job descriptor which wraps a 2048-bits long private exponent d into a black key.

```
[00] B0800005 jobhdr: stidx->[00] len=5
[01] 02010100 key: class1-pke len=256
[02] BE05D7C0 ptr->@0xbe05d7c0
[03] 60220100 fifostr: pke-jdk len=256
[04] BE05D7C0 ptr->@0xbe05d7c0
```

The result is an ECB-blackened private exponent, encrypted using JDKEK and written to the same buffer of the plain key (`0xbe05d7c0`).

4.1.2 Signing/Decrypting with a Black RSA key

CAAM implements an RSA decrypt operation that can be used either to decrypt data or to create a signature (sign data). The RSA Decrypt function is implemented via the CAAM's OPERATION Command. The private key may be supplied in the Black Key form.

CAAM allows the private key input to be provided in any of four different forms that enable increased efficiency of computation:

- #1 (`n`, `d`)
- #2 (`p`, `q`, `d`)
- #3 (`p`, `q`, `dp`, `dq`, `c`)
- #4 (`p`, `q`, `dp`, `dq`, `cR`, `RRp`, and `RRq`)

NOTE

`n` is never encrypted.

Referring to the Security Reference Manual, when using RSA Decrypt Protocol, the field 8-10 of `PROTINFO` defines if the input key is encrypted (black key) or not (plain key) and how it is encrypted.

- `000` private key is not encrypted.
- `001b` private key components are each encrypted with the JDKEK using ECB mode.

- 011b private key components are each encrypted with the JDKEK using CCM mode.
- 101b private key components are each encrypted with the TDKEK using ECB mode.
- 111b private key components are each encrypted with the TDKEK using CCM mode.

The following job descriptor decrypts a message using an ECB-blackened private key Format 1 (n, d).

```
[00] B0860007 jobhdr: stidx->[06] len=7
 PDB: RSA Decrypt PDB Private Key Form 1:
[01] 00100100 #d=256 #n=256
[02] BE243C90 g->@0xbe243c90
[03] BFC03000 f->@0xbfc03000
[04] BE05D600 n->@0xbe05d600
[05] BE05D4C0 d->@0xbe05d4c0
[06] 80190100 operation: uni-pcl rsa-decrypt prot_info=0x0100 nd(blk-nrm) f(red)
```

- g-> is a pointer to public exponent (e).
- f-> is a pointer to data private decrypt.
- n-> is a pointer to public modulus (n).
- d-> is a pointer to Private exponent (d).

Refer to Security Reference Manual for detailed information.

4.1.3 Exporting a blob RSA key

The following is an example of job descriptor which exports a black key originated from general memory to an ECB-black blob.

```
[00] B0800008 jobhdr: stidx->[00] len=8
[01] 14400010 ld: ccb2-key len=16 offs=0
[02] BE040368 ptr->@0xbe040368
[03] F8000130 seqoutptr: len=304
[04] BE05C440 out_ptr->@0xbe05c440
[05] F0000100 seqinptr: len=256
[06] BE05C5C0 in_ptr->@0xbe05c5c0
[07] 870D0004 operation: encap blob reg=memory, black, format=normal
```

Refer to Security Reference Manual for detailed information.

4.1.4 Importing a blob RSA key

The following is an example of a job descriptor that decapsulates a black blob into an ECB-black key.

```
[00] B0800008 jobhdr: stidx->[00] len=8
[01] 14400010 ld: ccb2-key len=16 offs=0
[02] BE040368 ptr->@0xbe040368
[03] F8000100 seqoutptr: len=256
[04] BE05C340 out_ptr->@0xbe05c340
[05] F0000130 seqinptr: len=304
[06] BE05C480 in_ptr->@0xbe05c480
[07] 860D0004 operation: decap blob reg=memory, black, format=normal
```

For a long-lived key, CCM-Blackened keys are recommended. This is the only way to get integrity checking. If the black key gets modified, AES-CCM will detect that, while AES-ECB will not.

4.2 ECDSA

The following sections describe ECDSA-related jobs.

4.2.1 Generating ECC black key

The following job descriptor instructs CAAM to generate an EC key-pair where the private key is blackened. In contrast to RSA, the private key is only visible inside CAAM. It uses the CAAM ECC built-in domains.

```
[00] B0840005       jobhdr: stidx->[04] len=5
                       PDB: EC PUBLIC KEYGEN
[01] 02000100               ECC domain: (0x2) P-256
[02] 9611B000               s->@0x9611b000
[03] 9611B020               Wx|Wy->@0x9611b020
[04] 80140006    operation: uni-pcl pk-pairgen  prot_info=0x0006 ecc enc_pri
```

P-256 is selected as ECC domain.

s is a pointer to input private key.

Wx|Wy is a pointer to public key.

The `enc_pri` bit of the Protocol Information(prot_info) is set. The input private key s is handled as an encrypted key (black key) and is encrypted after being generated.

4.2.2 Generating ECDSA signature with black key

The following job descriptor performs ECDSA signature generation using a black EC key. It uses ECDSA PDB with Elliptic curves built-in CAAM. The same could be achieved using non-built-in curves, with large pointers (curve parameters, and so on).

```
[00] B0860007       jobhdr: stidx->[06] len=7
                       PDB: ECDSA SIGN Msgrep
[01] 00400100               ECC domain: (0x2) P-256
[02] 9607F000               s->@0x9607f000
[03] 9607F020               f->@0x9607f020
[04] 9607F040               c->@0x9607f040
[05] 9607F060               d->@0x9607f060
[06] 80150006    operation: uni-pcl dsa-sign  prot_info=0x0006 ecc dec_pri (msgrep)
```

- `s` is a pointer to input private key.
- Wx|Wy is a pointer to input public key.
- `f` is a pointer to input message.
- `c` and `d` are respectively pointers to output signature (`r`, `s`).
- `enc_pri` bit of the Protocol Information (`prot_info`) is set. The input private key s is handled as an encrypted key (black key) and is decrypted after being read.

4.2.3 Verifying ECDSA signature

The following job descriptor performs ECDSA signature verification. It uses ECDSA PDB with Elliptic curves built-in-in CAAM. The same could be achieved using non-built-in curves, with large pointers (curve parameters etc).

```
[00] B0870008       jobhdr: stidx->[07] len=8
                       PDB: ECDSA VERIFY Msgrep
[01] 00400100               ECC domain: (0x2) P-256
[02] 9607F000               Wx|Wy->@0x9607f000
[03] 9607F040               f->@0x9607f040
[04] 9607F060               c->@0x9607f060
[05] 9607F080               d->@0x9607f080
[06] 9607F0A0               tmp->@0x9607f0a0
[07] 80160002    operation: uni-pcl dsa-verify  prot_info=0x0002 ecc (msgrep)
```

- Wx|Wy is a pointer to public key.

- `f` is a pointer to message.
- `c` and `d` are respectively pointers to signature (`r`, `s`).

5 Hands-on

In this hand-on we provide detailed steps how to install and deploy the demo code for ECDSA signature using black key. This guide demonstrates how to generate secure elliptic curve key-pair or where the private key is encrypted into a black key and use these keys for signature generation and verification. The black key is used in this implementation are encrypted using AES-ECB and can be easily modified to support AES-CCM.

NOTE

The code provided in this work is for demo purpose only and it does not fit for production neither for upstream.

The demo is based on 5 main components which are a caam_ecdsa, caam_blob, cryptodev, eckey and openssl.

caam_ecdsa a kernel module which exports ECDSA primitive functions for signature generation/verification and key-pair generation with ECB-encrypted private key.

caam_blob a kernel module which registers a device */dev/caam_blob* which serves blob encapsulation and decapsulation routines.

cryptodev module enables user space application access to Crypto API backend modules already present in the kernel. Linux kernel does not provide Crypto API interface to access accelerated hardware implementations of public key cryptography. Cryptodev has been extended to use functions exported by caam_ecdsa driver.

OpenSSL cryptodev engine has been modified accordingly to offload ECDSA operations to CAAM.

eckey a tool which exports/import ECC keys to and from CAAM blobs. It is a user-space tool that access /dev/caam_blob device.

In this demo, shared (non-trusted) job descriptors are used. Black keys and blobs are generated in non-secure world (Linux) through caam_blob module. The software maintaining the Jobring is the same building job descriptors (caam_ecdsa driver) To access the feature on User-space, Cryptodev module and OpenSSL's cryptodev engine have been extended to add support for public key cryptography and to specifically invokes ECDSA primitive functions exposed by caam_ecdsa driver.

To test the feature, you can either compile the BSP source code after adding the ***meta-imx-ecdsa-sec*** recipe to it.

5.1 Build

The following steps have been executed on a x86_64 machine running Ubuntu 16.04.

1. Install the essential packages:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo build-essential chrpath
libsdl1.2-dev xterm curl
```

2. Install repo tool:

```
$ mkdir -p ~/bin
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo >  ~/bin/repo
$ export PATH=$PATH:~/bin
$ chmod a+x ~/bin/repo
```

3. Download the BSP source:

```
$ cd ~
$ mkdir imx-yocto-bsp
$ cd imx-yocto-bsp
$ repo init -u https://source.codeaurora.org/external/imx/imx-manifest -b imx-linux-warrior -m
imx-4.19.35-1.1.0.xml
$ repo sync
```

4. Create a local branch for the demo:

```
$ repo start imx-ecdsa-sec-demo --all
```

5. Download meta-caam-pkc-sec layer source:

```
$ git clone https://source.codeaurora.org/external/imxsupport/imx_sec_apps
$ cp -r imx_sec_apps/meta-imx-ecdsa-sec/ sources/
```

6. Select your board:

   In our case it is an i.MX8QXP MEK board

```
$ DISTRO=fsl-imx-xwayland MACHINE=imx8qxpmek source fsl-setup-release.sh -b build
```

7. Add meta-caam-pkc-sec key layer:

```
$ bitbake-layers add-layer ../sources/meta-imx-ecdsa-sec
```

8. Add cryptodev module, OpenSSL and eckey utility by editing conf/local.conf file and append

```
IMAGE_INSTALL_append = " openssl openssl-bin cryptodev-module cryptodev-linux eckey"
```

9. Build:

   Build an image that fully supports the target device hardware and have the minimal required components to demonstrate the feature.

```
$ bitbake core-image-base
```

   This is the right time to foresee a coffee break because this is a time-consuming task!

10. Flash your SD-Card:

```
$ cd tmp/deploy/images/imx8qxpmek
$ bunzip2 -dk -f core-image-base-imx8qxpmek.sdcard.bz2
$ sudo dd if=core-image-base-imx8qxpmek.sdcard of=/dev/sd<X> bs=1M conv=fsync
```

5.2 Usage

After successfully building the components a simple use-case would be generating an EC key-pair with blackened private key, sign a message using the black key and verify it using the public key.

On the target:

5.2.1 Loading modules

1. Verify that Linux kernel version is 4.19.35_1.0.0

```
# uname -r
4.19.35-1.1.0+g0f9917c56d59
```

2. Verify that OpenSSL version is 1.1.1a

```
# openssl version
OpenSSL 1.1.1b  26 Feb 2019
```

3. Load caam_ecdsa module If you selected to build it as loadable module

```
# modprobe caam_ecdsa
```

4. Load cryptodev module

```
# modprobe cryptodev
```

Loading cryptodev module should automatically loads caam_ecdsa module since it depends on it.

5. Load caam_blob module

```
# modprobe caam_blob
```

5.2.2 Signing and verifying

1. List supported curves.

```
# openssl ecparam -list_curves
```

2. Generate a key-pair selecting a curve where point is a multiple of 128-bits. Example prime256v1

```
# openssl ecparam -engine devcrypto -genkey -out blackkey.pem -name prime256v1
```

3. Dump the generated key:

```
# openssl ec -in blackkey.pem -noout -text

read EC key
Private-Key: (256 bit)
priv:
    65:3d:6a:da:9f:a5:0d:11:a3:02:55:d4:21:13:41:
    21:c0:ab:01:41:83:0e:44:e7:55:c5:49:79:e6:cc:
    bd:df
pub:
    04:df:40:54:88:7e:e1:24:59:c4:3b:41:b0:ff:d2:
    79:42:fd:bc:e1:71:22:29:49:af:c4:d1:da:7b:40:
    31:11:5d:e3:04:ba:cb:94:cc:27:b7:3b:11:5e:6d:
    f9:c8:45:07:e5:80:e9:91:21:f9:93:c3:ee:bd:b7:
    ac:f9:5b:b1:d0
ASN1 OID: prime256v1
NIST CURVE: P-256
```

The key may fail if the private key starts with null byte.

4. Extract the public key:

```
# openssl ec -in blackkey.pem -pubout > pub.pem
```

5. Generate a sample message to sign.

```
# printf "sign me" > msg.txt
```

6. Sign it using the black key:

```
# openssl dgst -engine devcrypto -sha256 -sign blackkey.pem msg.txt > sig.bin
```

7. Verify with public key:

```
# openssl dgst -engine devcrypto -sha256 -verify pub.pem -signature sig.bin < msg.txt
```

If you take the private key and generate a signature in another target, signature verification in the current target should fail. Only target where the key-pair is generated can sign messages.

5.2.3 Exporting and importing blob

When the board is rebooted, the generated black key is no longer usable. Here comes the role of the caam_blob module. It leverages the functionality of exporting the black key to a black blob (before device shutdown) and import the black key from the blob during boot for example.

In this hand-on we decided to store the black key in PEM format to be OpenSSL compatible. Thus, before sending it to the caam_blob module to export it to blob, we need first to extract/dump it in binary form from the PEM, also, when we want to import the black key from the blob, we need to create a new PEM file with the black key.

This works provides a tool to parse/construct PEM files to replace the private key with a new session black key. The same goal could be achieved by other manner depending on how you decided to persist the key and the blob.

```
eckey
Usage: eckey <cmd>
export : Export black key to a black blob
import : Import black key from a black blob
```

After applying the blob patch and booting the device you should be able to see a device /dev/caam_blob

This has been developed as a proof of concept and as a simple way to communicate data between the user-space and the kernel.

1. Export the black key to a blob:

```
eckey export blackkey.pem blob.bin
```

2. Import the blob to the black key (The new black key will be written to the previous key, to preserve the public key):

```
eckey import blob.bin blackkey.pem
```

   Then you should be able to use again the black key for signing.

3. Sign a gain using the black key:

```
openssl dgst -engine devcrypto -sha256 -sign blackkey.pem msg.txt > sig.bin
```

4. Verify with public key:

```
openssl dgst -engine devcrypto -sha256 -verify pub.pem -signature sig.bin < msg.txt
```

6 Conclusion

In this work we presented a mechanism for strengthening public key cryptography implementations using CAAM secure key (i.e black key). We provided a set of CAAM job descriptors to perform RSA and ECDSA primitives using encrypted private keys, meaning that private keys are not visible to software but only known to hardware. The mechanism also prevents private key cloning, a secure key generated on a specific device can only be used in the device where the key has been generated. While in the hands-on we focused only on ECDSA, adding support to other PKC schemes is straightforward. Extending the current ECDSA implementation to cover more EC curves with secure key padding and handle CCM-blacken keys are other interesting development directions.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, UMEMS, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2020. **All rights reserved.**

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: June 2020

Document identifier: AN12838

arm