Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Fekets György

BIZTONSÁGI MEGOLDÁSOK AUTÓIPARI RENDSZEREKBEN

KONZULENS

Pintér Gergely
(Thyssenkrupp Components Technology Hungary Kft.)

Krébesz Tamás István
(BME-MIT)

BUDAPEST, 2019

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Fekets György**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2019. 12. 05.

...…….

Fekets György

Összefoglaló

A modern autóiparban egyre jelentősebb szerepet kapnak a közlekedést támogató szoftverek, mint például a sávkövetési funkció használata közben futó program, ami meggátolja, hogy a jármű, a sofőr hozzájárulása nélkül elhagyja a sávot.

Az ehhez hasonló funkciók napról napra fejlődnek, változnak, ennek következtében a gyártóknak minden szoftverfrissítés alkalmával el kell érni az autó fedélzeti számítógépét, hogy annak szoftverében változtatásokat eszközölhessen.

Amennyiben ezt szervizben kellene megtenni, úgy a járművek darabszáma és a frissítések gyakorisága miatt rendkívüli kellemetlenségeket okoznának a felhasználóknak. Ennek elkerülése érdekében a gyártók a járművek távolról való elérését látták helyes megoldásnak. Ez a megoldás viszont további problémákat vetett fel. Mivel a járművek távoli elérésére legtöbb esetben az internetet használják, így ez támadhatóvá válik. A járművek frissítéseit kezelő hardverre olyan szoftvert implementálnak, amely különböző ellenőrzéseket végez, mielőtt engedélyezné a frissítéseket az eszközön.

Az egyik ilyen megoldás a kriptográfián alapul, miszerint az autó csak azzal a féllel létesít kommunikációt, amely bizonyítani tudja hitelességét úgynevezett tanúsítvánnyal/tanúsítványokkal. Mivel a használt tanúsítványokat digitális aláírással hitelesítik, így a kriptorgráfia jelentős szeretpet játszik a feladatban.

Abstract

The driving assistance softwares become more important and widespread nowadays. For example the lane assist systems, that prevents the lane changes without the persmission of the driver.

Since the developers make new software changes very frequently, the companies have to reach the on board computer in their cars to change something in the software when an update is required.

Because of the large amount of vehicles, and because of the frequent updates it would be a very big problem for users and the services to handle all these software updates. To avoid this problem the companies has to reach the vehicles through the internet. Although this solution has other weaknesses. The internet connection is not secure and the connection can be attacked. To encounter this problem the cars must have a software component which is responsible for the security of the connection. This software has to run security checks before installing the software updates.

A working solution is based on cryptography. Whenever someone try to reach the car through the internet it has to provide a certificate, which will be verified by the car's software. The car must have trusted certificates and the verification process will use these certificates to make sure that the software updates are safe. Since digital signatures are used to make certifications, the cryptography is very important in this field.

1 Bevezetés

1.1 A feladat értelmezése

A félév során egy programot kell megvalósítanom az autóiparban használatos mikorvezérlőn, beágyazott környezetben, amely megvédi a jármű szoftvereit a külső támadásoktól. Amennyiben valaki csatlakozni szeretne a járműben található szotfverfrissítéseket kezelő eszközhöz, úgy annak biztosítania kell egy hiteles tanúsítványt, illetve tanúsítvány láncot. A kapott tanúsítvány láncon ellenőrzéseket kell elvégeznie a programnak és ennek az eredményétől függően elfogadnia, illetve elutasítania azt.

1.2 A feladat indokoltsága

A járművek távoli elérésének biztosítása érdekében védnünk kell az egyes járművek rendszerét. Amennyiben egy külső támadás során képes valaki megváltoztatni a jármű létfontosságú funkcióit ellátó szoftverek egyikét, úgy azok futása katasztrofális eredményekhez vezethet. El tudjuk képzelni, hogy ha egy autópályán a megengedett sebességgel haladó jármű kormányszervót irányító szoftvere helytelenül működik az hova vezethet.

1.3 A diplomaterv felépítésének rövid összefoglalása

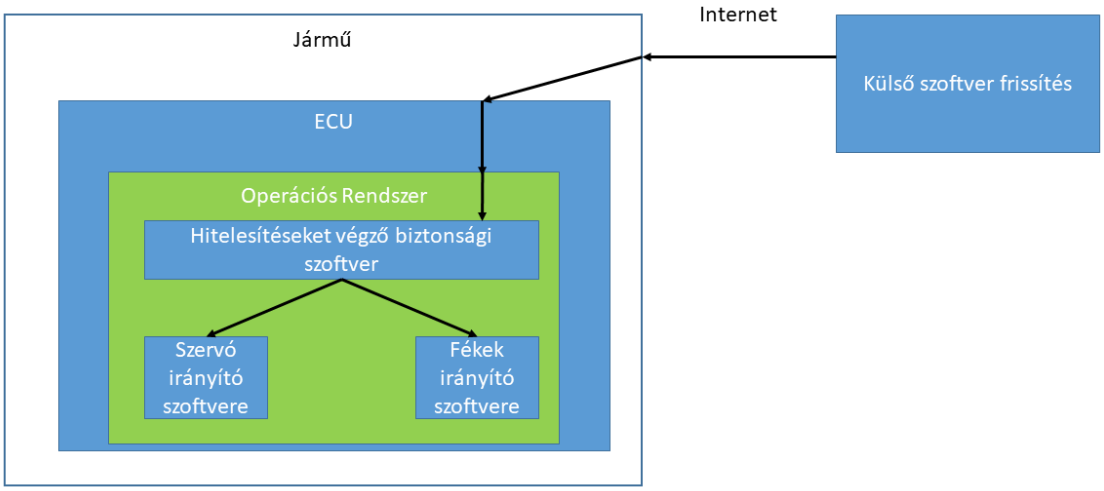
A bevezetést követő első fejezetben a feladat elvégzéséhez szükséges háttér információt ismertetem. Ezt követi a felhasznált könyvtárak funkcióinak leírása, és az AUTOSAR szabvány által specifikált API-k ismertetése. Majd a mikrovezérlő által nyújtott kriptográfiai funkciók leírása következik. Végül ismertetem a a tervezés és megvalósítás lépéseit és a tesztek eredményeit.

1.4 A rendszer felépítése

A modern járművekben találhatóak különböző folyamatokat vezérlő eszközök ún. ECU-k. Ezeken a mikrokontrollereken futó szoftvereek különböző feladatokért felelősek. A thyssenkrupp által használt eszközön lévő szoftver például az autó elektromos szervókormány rásegítését szabályozza, illetve a fékeket. Amennyiben egy gyártó az eszközre új szoftvert szeretne telepíteni, mert például hibát találtak a korábbi verzióval, úgy azt távoli eléréssel, interneten keresztül szeretnék megtenni.

Ezeket a frissítéseket a járműben elhelyezett ECU végzi, a kapott adatok alapján. Nyilvánvaló, hogy ezeket a frissítéseket az ECU nem fogadhatja el bárkitől, ugyanis előfordulhat, hogy illetéktelül szeretnének rajta frissítéseket telepíteni. Annak érdekében, hogy ez elkerülhető legyen az ECU-n futó operációs rendszernek egy új programot is kell futtatnia, amely biztosítja, hogy csak a megfelelő frissítések legyenek telepítve az eszközre.

Ahogy az 1. ábrán látható, a gyártó által küldött frissítés először beérkezik a járműben lévő ECU-ra. Ezután az operációs rendszeren futó biztonsági szoftver ellenőrzi, a frissítést telepíteni kívánó szervezet hitelességét és amennyiben az megfelelő, úgy az operációs rendszeren futó, járművet irányító szoftvereket frissíti.



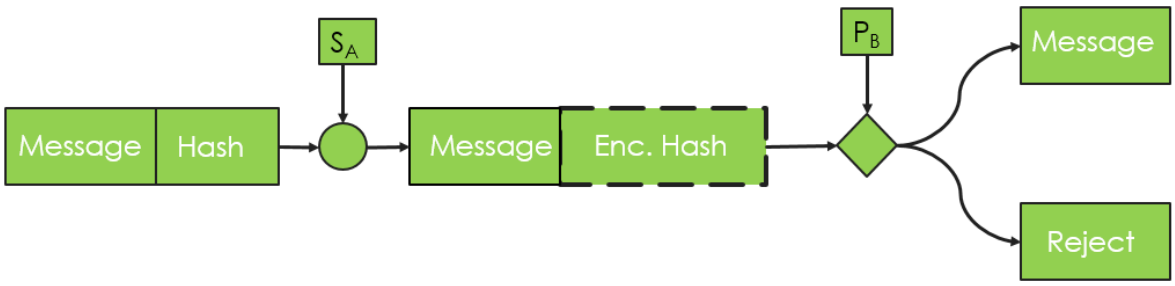
1. ábra Blokkdiagram

2 Az X509-es szabvány

A kriptográfiában széles körben használt X509-es szabvány standardizálja a tanúsítványok formáját, illetve azt, hogy milyen információkat kell tartalmazniuk. A szabványt több internetes protokollban használják, ahol a biztonság növelése a cél, mint például TLS/SSL kriptográfiai protokollban, ami a HTTPS alapját képzi. Ezen kívül az X509 tanúsítványok offline esetben is alkalmazhatók, például digitális aláírások esetében [1].

2.1 Digitális aláírás

A digitális aláírás egy kriptográfiai eszköz, amivel a hagyományos aláírást lehet helyettesíteni elektronikus formában. Lehetőséget nyújt az aláíró személyének azonosítására és annak megállapítására, hogy a dokumentum az aláírás óta változott-e. Tehát, hogy a dokumentum sérülten érkezett, illetve megváltoztatták.



2. ábra Digitális Aláírás

A 2. ábrán látható, amennyiben a számított és kapott ellenőrző összeg nem egyezik meg úgy elutasítható az üzenet, nem biztonságos forrásból származik. A digitális aláírás nem hamisítható, ezáltal letagadhatatlan. A fogadónak elég csak néhány nyilvános kulcsban megbíznia. Ezekkel a megbízható kulcsokkal írjuk alá a kommunikációs partnerek titkos kulcsát. Kommunikáció előtt ellenőrizni kell az aláírást a másik fél állítólagos kulcsán. Az aláírt nyilvános kulcsok tanúsítványokba rendezhetők.

2.2 Tanúsítvány felépítés

A szabvány magában foglalja, hogy mely mezőket kell tartalmaznia egy tanúsítványnak, illetve meghatározza a tanúsítványokkal végezhető műveleteket. A 3. ábárn látható, hogyan épül fel egy X509-es tanúsítvány.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            95:31:59:5a:2f:c9:88:f2
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = De, ST = Bayern, O = BMW AG, CN = BMW Supplier

        Validity
            Not Before: Apr  5 10:21:28 2019 GMT
            Not After : Apr  4 10:21:28 2020 GMT

        Subject: C = De, ST = Nordrhein-Westfalen, O = ThyssenKrupp AG, CN = ThyssenKrupp
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:d2:c6:e8:50:c6:11:39:83:80:61:59:4f:dc:f5:
                    [...]
                    7f:1f:8d:d7:76:72:9e:a7:63:d4:9d:f3:b7:7e:15:
                    58:dd
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                EF:FC:91:45:10:09:45:31:33:69:01:4D:A3:3E:0F:34:C8:78:34:1F
            X509v3 Authority Key Identifier:
                keyid:E0:10:6F:4D:A7:59:CF:88:7D:05:C3:E4:BB:4E:B7:96:EE:0E:BC:DF

    Signature Algorithm: sha256WithRSAEncryption
         95:2e:84:ac:c4:3a:3b:7e:74:23:ea:7c:aa:91:1a:ad:31:e0:
         [...]
         26:d7:50:5e:2b:28:98:2d:92:95:58:88:66:df:16:98:8d:1b:
         c5:5a:c1:69
```

3. ábra Példa az X509 szabványnak megfelelő tanúsítványra

Az X509-as szabvány v3-as verziójának megfelelő digitális tanúsítvány struktúrája az alábbi [2]:

- Tanúsítvány
    - Verzió szám (Információt nyújt a támogatott verzióról)
    - Sorozat szám
    - Aláíró algoritmus típusa
    - Aláíró identitás (A tanúsítványt kiállító entitás főbb adatait tartalmazza)

    - Érvényességi időtartama (Megadja, hogy mily en dátum után és milyen dátumig lehet használni a tanúsítványt)
    - A tanúsítványt igénylő identitás neve
    - Subject Publikus Kulcsa (Ez a mező tartalmazza az igénylő publikus kulcsát, annak minden adatával együtt.)
    - Bővítmények (Opcionális)
- Tanúsítvány aláíró algoritmus (Az aláíró algoritmus típusa)
- Tanúsítvány aláírása (Maga a tanúsítványt hitelesítő aláírás)

2.3 Tanúsítványkezelés

Az X509-es szabvány szerint egy szervezetnek, amely érvényes aláírt tanúsítványt szeretne, igényelhet egyet egy ún. Certificate Signing Request (CSR) által. Ebben a folyamatban először egy kulcspárt kell generálnia, egy nyilvános és egy privát kulcsot és az utóbbit titokban kell tartania. A CSR tartalmaz információt a tanúsítvány tigénylő szervezetről illetve magát a publikus kulcsot. A hitelesítő ún. Certificate Authority (CA) ehhez a publikus kulcshoz adja a tanúsítványt.

A szabvány megengedi, hogy a tanúsítványok láncokba szervezhetőek legyenek, ennek alapja egy szervezet által megbízhatónak tekintett ún. Root Certificate. Ebben az esetben a tanúsítványok szerkezetében az összefüggést a hitelesítést végző Certificate Authority jelenti. Amennyiben a lánc legutolsó tanúsítványát kiállító Autority a szervezet számára nem megbízható, illetve ismeretlen úgy elkezdheti, a láncnak a hitelesítő Authority-ról szóló információt tartalmazó mezője alapján való visszafejtését, amíg el nem éri a lánc első elemét.

2.4 Tanúsítványok ellenőrzése

A tanúsítványok, illetve tanúsítvány láncok ellenőrzése a szabvány szerint nyújtott információk alapján történik. Ellenőrizhető például az érvényesség illetve az aláírás helyessége. A különböző ellenőrzések végrehajtását támogatják különféle kriptográfiai könyvtárak, szoftverek. Ezek közül a szakdolgozatom során három különbözővel ismerkedtem meg: OpenSSL, WolfSSL, Mbed TLS. A tanúsítványok ellenőrzésének folyamatát ezeken keresztül fogom bemutatni.

3 Tanúsítványkezelés

3.1 OpenSSL

Az OpenSSL egy könyvtár olyan alkalmazások számára, amelyek biztosítani hivatottak számítógépek közti kommunikációt és védenek az esetleges támadások ellen ellen. Széles körben felhasznált a legtöbb HTTPS weboldalnál is.

Az OpenSSL könyvtár használata jó választásnak tűnt, mivel az összes funkciót meg lehet valósítani a segítségével, amelyre az alkalmazásban szükség lehet, ezen kívül a könyvtár támogatja az X509-es szabvány használatát is. Maga a könyvtár C nyelven íródott, ami elősegíti a beágyazott rendszerekben való használatot.

A feladat a könyvtár által nyújtott funkciók segítségével megvalósítani a szoftvert, amely ellenőrzéseket végez a kapott tanúsítványokon.

3.1.1 OpenSSL command line

Az OpenSSL lehetőséget nyújt többek közt tanúsítványok és tanúsítvány láncok létrehozására, Certificate Signing Request kérésére. Ezeken kívül ellenőrizni is lehet a láncokat. Ezen kívül információt lehet lekérni a tanúsítványokról.

Az alábbi parancsokkal lehet a fent felsorolt műveleteket elvégezni [3]:

- `openssl genrsa -out Centerkey.pem 2048` RSA típusú kulcs generálása, 2048 bites változat. A pem kiterjesztés arra utal, hogy a kulcs egy olvasható formában lett létrehozva nem binárisan.
- `openssl req -new -key Centerkey.pem -out CenterCsr.csr` A parancs egy Certificate Signing Requestet hoz létre.
- `openssl ca -config /etc/ssl/openssl.cnf -outdir . -in CenterCsr.csr -extensions v3_ca -cert /home/fekets/BMW/BMW_AG/RootCAcert.pem -keyfile ../BMW_AG/RootCAkey.pem -create_serial -out Centercert.pem -days 365` Ezzel a paranccsal lehet egy meglévő Certificate Signing Requestet elfogadni és egy aláírt tanúsítványt generálni.

3.1.2 OpenSSL API

Az OpenSSL API a tanúsítványok kezelésére különböző funkciókat nyújt. A tanúsítványok tárolására több struktúra használható. Az *X509* struktúra egy X509-es szabványnak megfelelő struktúrát tárol annak minden paraméterével. Az *X509_STORE* struktúra hivatott tárolni a tanúsítványokat és a Certificate Revocation List, ami a visszavont tanúsítványokat tartalmazza. Ezen kívül tárol információt a hiteles tanúsítványokról, amelyek a hitelesítéshez szükségesek. Az *X509_STORE_CTX* struktúra támogatja az ellenőrzés elvégzését, ezen keresztül lehet a hiteles tanúsítványoknak megfeleltetni az hitelesítendőket [4].

Az *X509*-es struktúrák használatát a következő funkciók támogatják. A struktúrák használatához először létre kell hozni őket és inicializálni a könyvtárt, amelyet az `X509_STORE_new` és `X509_STORE_CTX_init` függvény támogatja. A tanúsítványkezelés esetében elengedhetetlen, hogy információt nyerjünk ki a felhasznált tanúsítványokból. Az `X509_get_issuer_name`, `X509_get_subject_name` és `X509_get_pubkey` funkciók rendre megadják a tanúsítvány-t kiállító entitás nevét, a tanúsítványt igénylő szervezet nevét és a benne foglalt publikus kulcsot. Az OpenSSL lehetőséget nyújt a tanúsítványok ellenőrzésére is. Ebben az esetben egy *X509_STORE_CTX* típusú Kontextusban tárolt Root tanúsítványok alapján az `X509_verify_cert_error_string` funkciót lehet meghívni. Ezt megelőzően a megfelelő sturktúrába az `X509_STORE_add_cert` függvénnyel lehet tanúsítványt tölteni. Az API-val a tanúsítványkezelés többi műveletét is meg lehet valósítani. Az elutasított tanúsítványokat be lehet tölteni a megfelelő struktúra CRL mezőjébe, illetve az érvényesség lejárta után vissza lehet vonni a tanúsítványokat. Amennyiben sérült egy tanúsítvány úgy el lehet utasítani [4].

Az OpenSSL API lehetőséget nyújt kulcsok generálására, CSR kérésére illetve tanúsítványok kiállítására, illetve minden olyan műveletre, amelyet parancssorból meg lehet valósítani.

3.2 WolfSSL

A WolfSSL egy szoftver könyvtár, amely többek között kriptográfiai feladatok elvégzését támogatja. A fent leírt OpenSSL könyvtárnál, sokkal kisebb méretű, így beágyazott rendszerek esetén sok esetben kiválthatja az OpenSSL könyvtárat.

3.2.1 WolfSSL konfigurációja

A feladat megvalósítása során kiderült, hogy az OpenSSL könyvtár túl nagy méretű, illetve túl sok funkciót tartalmaz. Mivel az programban viszonylag kevés beépített elemet kellett felhasználni, ezért a legtöbbet nélkülözni lehetett. Ezeken még a konfigurációkor van lehetőség módosítani úgynevezett kapcsolókkal lehet bizonyos funkciókat a már lefordított könyvtárból elhagyni, illetve amelyek alapértelmezetten le vannak tiltva, azokat engedélyezni. Ezekből pár példa, leírással az 1. táblázatban látható [5]:

1. táblázat WolfSSL kapcsolók

Option	Default Value	Description
--enable-static	Disabled	Building static wolfSSL libraries
--enable-crypttest	Enabled	Enable Crypt Bench/Test
--enable-certservice	Disabled	Enable certificate service
--enable-jobserver	Enabled	Enable one more than CPU count
--enable-shared	Disabled	Building shared wolfSSL libraries

A könyvtár alapértelmezetten két tanúsítvány illetve kulcs típust támogat amelyek a PEM, (szöveges formátumú) illetve DER (bináris). Az előbbi sok esetben megkönnyítheti a fejlesztést, mivel text editor segítségével megnyitható és ezáltal elolvasva azt értelmes információhoz juthat a fejlesztő. A WolfSSL segítségével a tanúsítványok alapértelmezetten fájlrendszerből olvashatóak be, a legtöbb esetben PEM formátumban megadva. Mint a legtöbb beágyazott eszköz esetében, az általam használt

ECU-n futó operációs rendszer sem támogatta fájlrendszer használatát. Ennek következtében a hitelesíteni kívánt tanúsítványokat nem lehet egy szöveges fájlból, a fájlrendszerből beolvasni. A WolfSSL könyvtár ennek a problémának a megoldására lehetővé teszi memória bufferek használatát. Ez egy új konfiguráció létrehozásával történik a NO_FILESYSTEM kapcsoló beállításával. Ezek után elérhetővé válnak a következő funkciók:

```
int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX* ctx,
const unsigned char* in, long sz, int format);
```

A függvény segítségével egy bufferben tárolt tanúsítványt lehet betölteni a később hitelesítésre használt kontextusba.

```
int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX* ctx,
const unsigned char* in, long sz, int format);
```

```
int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX*
ctx, const unsigned char* in,long sz);
```

Az első függvényhez hasonlóan itt egy kulcs illetve egy egész tanúsítvány lánc betöltése történik bufferből [6].

A konfiguráció következő lépése a dinamikus tárkezelés mellőzése volt. Ebben az esetben csak statikus memória használatát engedélyezi a WolfSSL könyvtár, aminek a memóriafoglalásból adódó problémák elkerülése a célja. Ehhez a STATIC_MEMORY kapcsoló nyújt segítséget [7].

A memóriahasználat minimalizálása érdekében, a PEM formátumú tanúsítványokat DER formátumba alakítva tovább csökkenthető a program memória igénye. Ezt a transzformációt egy segédprogram végzi a szöveges tanúsítványból egy `unsigned char` tömbbé.

A végső és legjobbnak bizonyuló konfigurációt a SMALL_STACK kapcsoló beállítása után értem el. Ennek következtében a memóriaigény 1kByte alá csökkent, ami messze meghaladta az OpenSSL esetén kapott értéket, és a dinamikus memóriakezelést alkalmazó konfiguráció 10kByte-os memóriahasználatánál is optimálisabbnak bizonyult. A végső konfigurációs beállítás: ./configure --prefix /c/Users/gyorgy.fekets/Downloads//wolf-install/ --enable-jobserver=no -enable-

crypttests=no --enable-examples=no --enable-static=yes --enable-shared=no --enable-certservice=yes

3.3 MbedTLS

A második kipróbált kriptográfiai funkciókat támogató könyvtár az MbedTLS. Ez a könyvtár sokkal kisebb méretű, mint az OpenSSL, viszont jóval kevesebb funkciót is tartalmaz. A könyvtárat első sorban beágyazott rendszereken történő használatra készítették. A könyvtár teljes mérete kevesebb, mint 64kByte RAM-ot használ, amely előnyös az alkalmazásban. Mivel ez a szám is túl nagy a jelenlegi esetben, nincs lehetőség a teljes könyvtár használatára, hanem bizonyos funkciókat el kell belőle hagyni.

A könyvtár támogatja az X509 szabványnak megfelelő tanúsítványok használatát, és az általános Hash funckiókat, illetve különböző Cipher funkciókat is támogat. Ezen kívül külböző kulcsok generálását, illetve szöveges PEM és bináris DER formátumú tanúsítványok kezelését is lehetővé teszi.

A könyvtár egy másik nagy előnye, hogy beépített függvényeket tartalmaz tanúsítványok adatainak kinyerésére. Tehát viszonylag egyszerűen elérhető egy publikus kulcs a tanúsítványból.

Annak ellenére, hogy kis mérete megfelelőnek bizonyult beágyazott környezetben és tanúsítványláncok ellenőrzését is meg tudtam vele valósítani, nem erre a könyvtárra esett a választás a végső feladat megoldásakor. Ennek az volt az oka, hogy a WolfSSL könyvtárral jóval többet foglalkoztam és mélységében ismertem annak felépítését, így jobban működő programot tudtam létrehozni vele. Az MbedTLS könyvtár viszont jó alternatíva, további fejlesztésekre.

4 AUTOSAR szabvány által nyújtott kriptográfiai API-k

4.1 Crypto Stack

A Crypto Stack a kriptográfiai szolgáltatások standard elérését biztosítja a különböző alkalmazások és rendszer funkciók számára.

A Crypto Stack kriptográfiai szolgáltatásai olyanok, mint hash értékek számítása, aszimmetrikus vagy szimmetrikus titkosítás. Ezek a szolgáltatások az alatta lévő kriptográfiai primitívektől és kriptográfiai sémáktól függenek. Ezért a Crypto Stack legfelső részének, a Crypto Service Manager-nek lehetőséget kell adni különböző alkalmazások számára, hogy ugyan azokat a szolgáltatásokat használják, de különböző primitívek és vagy sémák támogatásával. Például lehetséges, hogy egy alkalmazásnak szüksége van egy hash szolgáltatásra, hogy kiszámolja egy SHA2 hash értékét, míg egy másiknak egy SHA1 hash értékét kell kiszámítania. Előfordulhat az az eset is, amikor egy alkalmazásnak hitelesíteni kell egy aláírást amely RSA-val lett rejtjelezve és SHA2-t használ, de egy másik alkalmazásnak SHA1-et használt hash primitívként [8].

A Crypto Stack-nek lehetőséget kell nyújtani, hogy egy alkalmazás számára szükséges szolgáltatások konfigurálhatók legyenek és különböző konfigurációkat kell készítenie minden egyes szolgáltatásra, ahol a funkciók és primitívek választhatóak.

Továbbá, mivel számos kriptográfiai művelet számítása, meglehetősen számításigényes feladat, a hosszú számításokat ütemezni kell. A Crypto Job-ok konfigurációjában állíthatónak kell lennie, hogy azok szinkron vagy aszinkron módon legyenek végrehajtva.

A Crypto Stack által nyújtott kriptográfiai szolgáltatások szoftver könyvtárakon vagy hardver modulokon alapulnak, de lehetségesek összetett megvalósítások is például, ha egy hardver modul nem képes a szükséges feladatokat magában ellátni. Ez alapján a nevezhetünk minden alul lévő funkcionalitás példányt crypto könyvtárnak legyen az hardveres vagy szoftveres.

4.1.1 Crypto Stack Szolgáltatásai

A Crypto Stack számos kriptográfiai primitív implementációját foglalja magában, amelyeket több különböző Basic Software modul használ fel. A primitívek egy vagy két algoritmus felhasználásán alapulnak tipikusan egy cipher és egy hash algoritmusból állnak össze. A Crypto Stack a következő funkcionalitásokat biztosítja az alkalmazások számára [10]:

4.1.1.1 Primitívek a Crypto Stackben:

- **Hash számítás:** A kriptográfiában használt hash művelet egy olyan folyamat, amely tetszőleges méretű adat blokkot kap bemenetként és egy fix méretű bit string-et, a hash értéket adja vissza kimenetként. Bármilyen véletlen bekövetkező vagy szándékos változtatása a bemenetnek megváltoztatja a hash értéket. A legnagyobb előnye a műveletnek, hogy kivitelezhetetlen, hogy a hash értékből visszafejthető legyen az üzenet. Ezen kívül rendkívül számításigényes két külböző üzenetet találni amelynek hash értéke megegyezik.
- **MAC generálás és ellenőrzés:** A MAC (Message Authentication Code) egy üzenet hitelesítésére használt információ. A MAC algoritmus egy titkos kulcsot fogad bemenetként és egy tetszőlegesen hosszú üzenetet, hogy hitelesítse és kimenetként adja vissza a MAC-et. A MAC érték biztosítja az üzenetben foglalt adat hitelességét és sértetlenségét, azáltal, hogy akik ellenőrizni kívánják a kapott értéket meghatározhassák, ha változás történt az üzenetben. (A MAC-et ellenőrző entitás birtokában van a titkos kulcs.)
- **Titkosítás és titkosítás feloldása:** A titkosítás arra szolgál, hogy egy üzenetet vagy bármilyen információt olyan módon kódoljanak, hogy csak az arra jogosult felek érhessék el a benne lévő adatokat.
- **Aláírás generálás és ellenőrzés:** A Crypto Stack által nyújtott aláírás generálás és ellenőrzés teljes mértékben megegyezik a már korábban tárgyalt általános metódussal.

4.1.1.2 Crypto Stack által nyújtott legfőbb Cipher-ek:

- RSA: Az RSA egy titkos kulcsú, vagy másnéven aszimmetrikus titkosító algoritmus. A Crypto Stacknek mindenképpen meg kellett tudnia valósítani a titkosítás ezen módszerét, mivel ez napjaink egyik legelterjedtebb algoritmusa erre a célra. Ebben az esetben a nyilvános kulccsal titkosított üzenetet csak a titkos kulcs birtokában lehet elolvasni [9].
- AES: Az AES egy módszer üzenetek titkosítására. Ebben az esetben a blokkméret 128 bit, a kulcs pedig 128, 192 vagy 256 bites.
- 3DES: A Triple Data Encryption Algorithm egy szimmetrikus kulcsú block cipher 168 bites kulccsal és 64 es block mérettel. Lényegében egy DES cipher algoritmus minden blokkra háromszor alkalmazva.

4.1.1.3 Crypto Stack által nyújtott Hash funkciók:

SHA1: Egy kriptográfiai hash funkció a Secure Hash Algorithm 1 családból, amely 160 bit hosszú hash értéket eredményez.

SHA2-224: Egy kriptográfiai hash funkció a Secure Hash Algorithm 2 családból, amely 224 bit hosszú hash értéket eredményez, amely az SHA-256 csonkított változata.

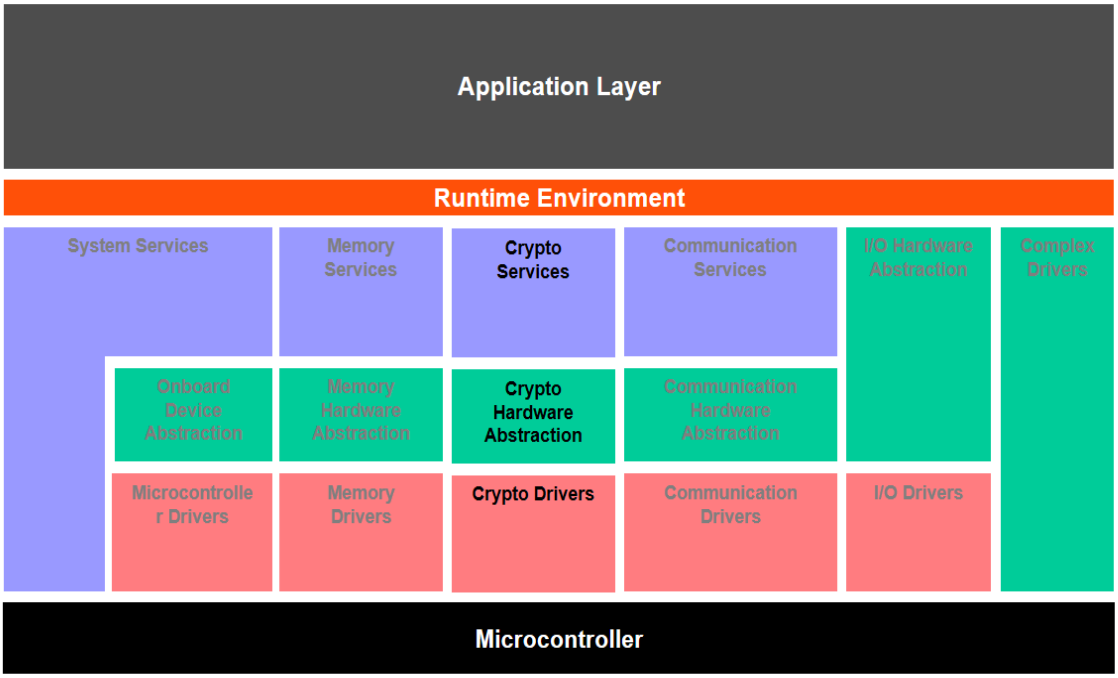
Ezekből pár példa összefoglalva a 2. táblázatban látható.

2. táblázat A Crypto Stack által nyújtott Hash műveletek

Kriptográfiai hash funkció	Hash érték bitszáma
SHA1	160
SHA2	256
SHA2	512
SHA3	256
SHA3	512
BLAKE-1	512
BLAKE-2	512

4.1.2 A Crypto Stack felépítése

A Crypto Stack a Crypto Service Manager-ből, a Crypto Interface-ből és a Crypto Driver-ből épül fel. A Crypto Stack és a Communication Stack közti kapcsolatot a Secure On-Board Communication Modul biztosítja, ez a 4. ábrán látható.



4. ábra AUTOSAR rétegei

4.2 Crypto Service Manager

A Crypto Service Manager szinkron és aszinkron szolgáltatásokat nyújt minden szoftver modul számára, annak érdekében, hogy mindegyik modulnak külön hozzáférése legyen az alapvető kriptográfiai funkciókhoz. A Crypto Service Manager-nek egy absztrakciós réteget is kell nyújtani, amely egy standard felületet ad a felette lévő rétegeknek, hogy azok elérjék az alapvető funkciókat [8].

Bizonyos szoftver modulok által elvárt funkcionalitások eltérhetnek más szoftver modulok által elvártaktól. Ebből kifolyólag minden egyes szoftver modulnak külön lehetőséget kell nyújtani, hogy a Crypto Service Manager funkciói és szolgáltatásai konfigurálhatók legyenek, mert bizonyos esetekben egy modulnak csak néhány szolgáltatásra van szüksége. Ez a konfiguráció tartalmazza a szinkron, valamint az aszinkron feldolgozású Crypto Service Manager szolgáltatásokat.

A Crypto Service Manager modul felépítése generikus megközelítésen alapul. Amennyiben egy struktúra vagy interfész egy részletes specifikációja limitálná a Crypto Service Manager-t, az interfész és a struktúra generikus módon kell, hogy definiálva legyen. Ez lehetőséget nyújt a jövőbeli változtatásokra.

A Crypto Service Manager-ben megvalósított API jelentősebb függvényei a következők:

HASH:

```
Std_ReturnType Csm_Hash( uint32 jobId, Crypto_OperationModeType mode, const uint8* dataPtr, uint32 dataLength, uint8* resultPtr, uint32* resultLengthPtr )
```

A Crypto Service Manager függvénye, amellyel kiszámítható a hash érték.

MAC:

```
Std_ReturnType Csm_MacGenerate( uint32 jobId, Crypto_OperationModeType mode, const uint8* dataPtr, uint32 dataLength, uint8* macPtr, uint32* macLengthPtr )
```

A függvény egy MAC-t generál.

Ez a MAC érték ellenőrizhető a következő függvénnyel:

```
Std_ReturnType Csm_MacVerify( uint32 jobId, Crypto_OperationModeType mode, const uint8* dataPtr, uint32 dataLength, const uint8* macPtr, const uint32 macLength, Crypto_VerifyResultType* verifyPtr )
```

Titkosítás:

```
Std_ReturnType Csm_Encrypt( uint32 jobId, Crypto_OperationModeType mode, const uint8* dataPtr, uint32 dataLength, uint8* resultPtr, uint32* resultLengthPtr )
```

```
Std_ReturnType Csm_Decrypt( uint32 jobId, Crypto_OperationModeType mode, const uint8* dataPtr, uint32 dataLength, uint8* resultPtr, uint32* resultLengthPtr )
```

A két függvény a kódolást illetve dekódolást oldja meg.

A Crypto Service Manager külön felületet nyújt a szimmetrikus, illetve aszimmetrikus kódolásra.

Aláírások:

```
Std_ReturnType Csm_SignatureGenerate( uint32 jobId, Crypto_OperationModeType mode, const uint8* dataPtr, uint32 dataLength, uint8* resultPtr, uint32* resultLengthPtr )
```

```
Std_ReturnType Csm_SignatureVerify( uint32 jobId, Crypto_OperationModeType mode, const uint8* dataPtr, uint32 dataLength, const uint8* signaturePtr, uint32 signatureLength, Crypto_VerifyResultType* verifyPtr )
```

A fenti két függvény az aláírás generálását, illetve az aláírás ellenőrzését végzi.

A kulcsok kezelését végző API működését később, a megvalósítás résznél tárgyalom.

4.3 Crypto Interface

A Crypto Interface a Crypto Service Manager és az alatta lévő Crypto Driver-ek között foglal helyet. A Crypto Interface egy egyedi réteg, amely a kriptográfiai funkciók elérését biztosítja az összes felette lévő réteg számára. Az absztrakciós réteg megában foglal különböző szoftver eléréseket és ezzel biztosítja, hogy a Crypto Interface implementációja független legyen minden alatta lévő Crypto Driver-től, amelyek hardverben vagy szoftverben vannak megvalósítva. Ezen kívül biztosítja a különböző crypto funkciók elérését, ezáltal lehetővé téve különböző crypto task-ok egyidejű futását [11].

4.4 Crypto Driver

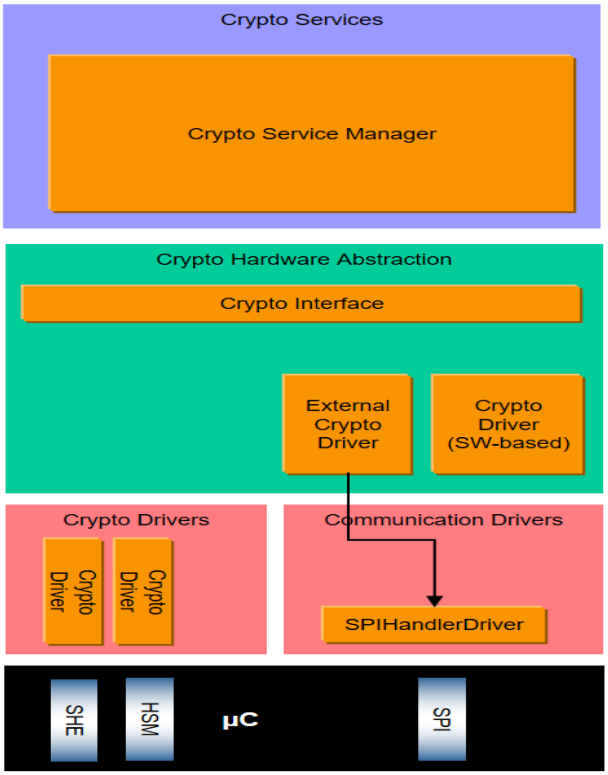
A Crypto Driver modul a mikrokontroller absztrakciós szintjén helyezkedik el a Crypto interface és a Crypto Service Manager modul alatt és egy generikus interfészt valósít meg szinkron és aszinkron kriptográfiai primitívek számára. Emellett támogatja a kulcsok tárolását, illetve konfigurálását és a kulcskezelést a kriptográfiai funkciók számára [12].

Ahhoz, hogy megfelelő kriptográfiai funkciókat nyújtson az ECU, integrálnia kell egy Crypto Service Manager modult és egy Crypto Interface-t. Bár a Crypto Interface elérhet számos Crypto Driver-t, mindegyik az alatta lévő Crypo Driver Object szerint van konfigurálva. Egy Crypto Driver Object reprezentálja egy független kripto hardver eszköz (pl AES gyorsító) példányát. Létezhet egy csatorna a gyors AES és CMAC számításokra a HSM-en, olyan Crypto Job-ok számára, amelyek magasabb priorítással bírnak.

A Crypto Driver-t további két részre bomlik. A mikrokontrolleren két külön egység foglal helyet. A Tricore egység illetve a HSM. Minden kérés a Tricore oldalra érkezik be először és csak bizonyos számításokhoz használja fel a Crypto Driver a HSM által nyújtott lehetőségeket.

4.4.1 Kommunikáció a rétegek között

A Crypto Stack részére érkezett kéréseket először a Crypto Service Manager kezeli. Mivel a Crypto Service Manager egy standardizált felületet szolgáltat a magasabb rétegekből érkező kéréseknek, ezért ennek a modulnak a feladat a kéréseket alsóbb szintekre továbbítani már specifikusabban. A Crypo Interface megfelelő szolgáltatását hívja a Crypto Service Manager amely kérés továbbítódik az Crypto Interface-n keresztül a Crypto Driver-nek. A Crypto Driver a kéréstől függően vagy a mikrokontrollert használja a számítások elvégzésére, illetve nagy számítás igényű kérés esetén a mikrokontrollerbe beleépített HSM gyorsító lehetőségeit használja ki a számítási idő lecsökkentéséhez. A számítások befejeztével a Crypto Driver a megfelelő adatokat a Crypto Interface-n keresztül a Crypto Service Managernek szolgáltatja, amely továbbítja azokat a feljebb lévő rétegeknek. A kommunikáció az 5. ábrán látható felépítésű.



5. ábra Crypto Stack szerkezete

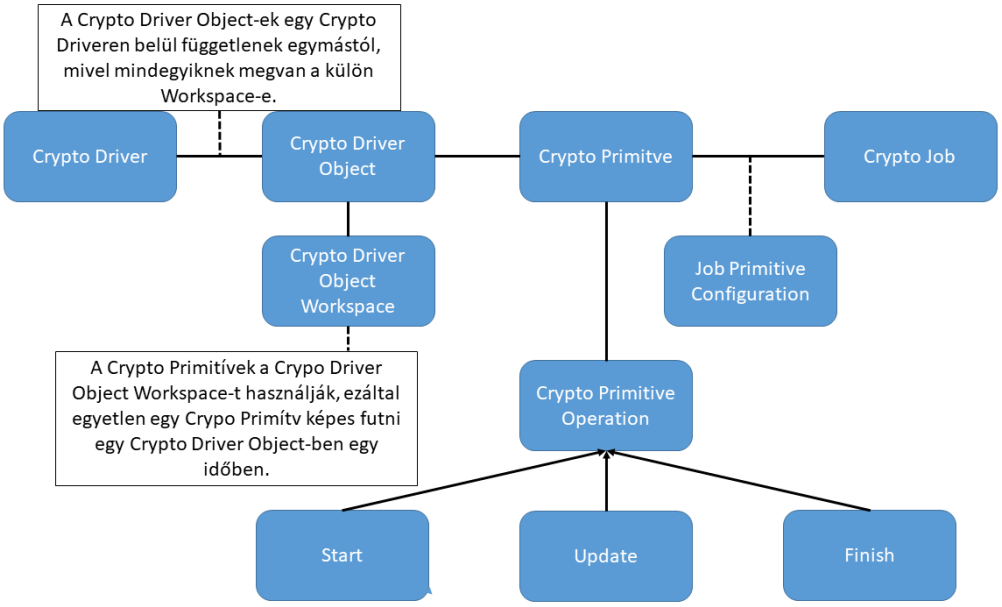
4.5 Objects, Primitives and Jobs

A Crypto Driver tetszőleges számú Crypto Driver Object-et implementál. Minden Crypto Drivernek megvan a saját Workspace-e, amely a részleges eredmények tárolására szolgál, mint például egy hosszú üzenet első néhány bájtjának hash értéke. A Crypto Driver Object tetszőleges számú Crypto Pimitive-t implementálhat. Mivel minden egyes Workspace egy Crypto Driver Object-hez van hozzárendelve és nem minden egyes Crypto Primitive-hez külön, ezért bármely időpillanatban tetszőleges számú Crypto Driver Object lehet aktív, de minden Crypto Drivernek legfeljebb egy Crypto Primitive-e állhat feldolgozás alatt. Ebből látszik, hogy az egyes Workspace-k működésükben függetlenek egymástól.

A Crypto Primitiv-ek konfigurálhatóak aszerint, hogy milyen kontextusban lesznek felhasználva és tipikusan egy kulcsra van szükségük a működéshez. A Crypto Primitive-et egy kulccsal párosítva Crypo Job-nak nevezzük. A Cypto Job-ok a konfigurálási időben már definiáltak nem futási időben. Ezen kívül prioritás van rendelve hozzájuk (a magasabb érték a fontosabb), ami a konfiguráció része, tehát futási időben például a prioritás már nem változtatható meg. Mivel a Crypto

Job-ok a Crpyto Driver Objec-ek által nyújtott Crpyo Primitive-ből vannak példányosítva, ezért egy időben egyszerre csak egy Crypto Job állhat feldolgozás alatt.

A 6. ábrán jól megfigyelhetőek a fent leírt kapcsolatok:



6. ábra Crypto Primitve-k és Crypto Job-ok

4.5.1 Job feldolgozás állapotai

A felhasználók a Crypto Stack interface-n keresztül küldhetnek kéréseket egy bizonyos művelet, vagy több művelet végrehajtására egy Crypto Job-on.

OPERATION: Egy művelet a Crypto Primitve-en meghatározza, hogy a primitív mely része hajtódjon végre. A műveletek a következőek lehetnek: START, UPDATE, FINISH illetve ezeknek bármilyen kombinációja.

START: A művelet egy Crypto Primitve új kérését jelzi. A műveletnek minden előző kérést vissza kell vonni és a szükséges inicializálást el kell végezni. Ezen kívül figyeli, hogy a kripto primitív végrehajtható-e.

UPDATE: A művelet jelzi, hogy a Crypto Primitve egy új bemeneti adatot vár. Az update művelet köztes eredményeket is szolgáltathat.

FINISH: A művelet jelzi, hogy minden bemeneti adat megfelelően be lett töltve és a Crypto Primitive be tudja fejezni a számításokat. A finish műveletnek minden esetben eredményeket kell szolgáltatnia.

Alapértelmezetten minden Crypto Job Idle állapotban van. A START művelet visz át egy Idle Job-ot Started állapotba. Bármilyen másik állapotban meghívva ezt a műveletet az elveti az addig megkapott eredményeket és újra indítja az új Job-ot. Az Update művelet Started állapotból Updating állapotba mozdítja, illetve Updating állapotban hagyja, ha már ott volt eredetileg is. A Finish művelet Updating állapotból viszi Idle állapotba a Job-ot és az eredményeket a hívónak visszaszolgáltatja.

4.5.2 Állapotváltozások:

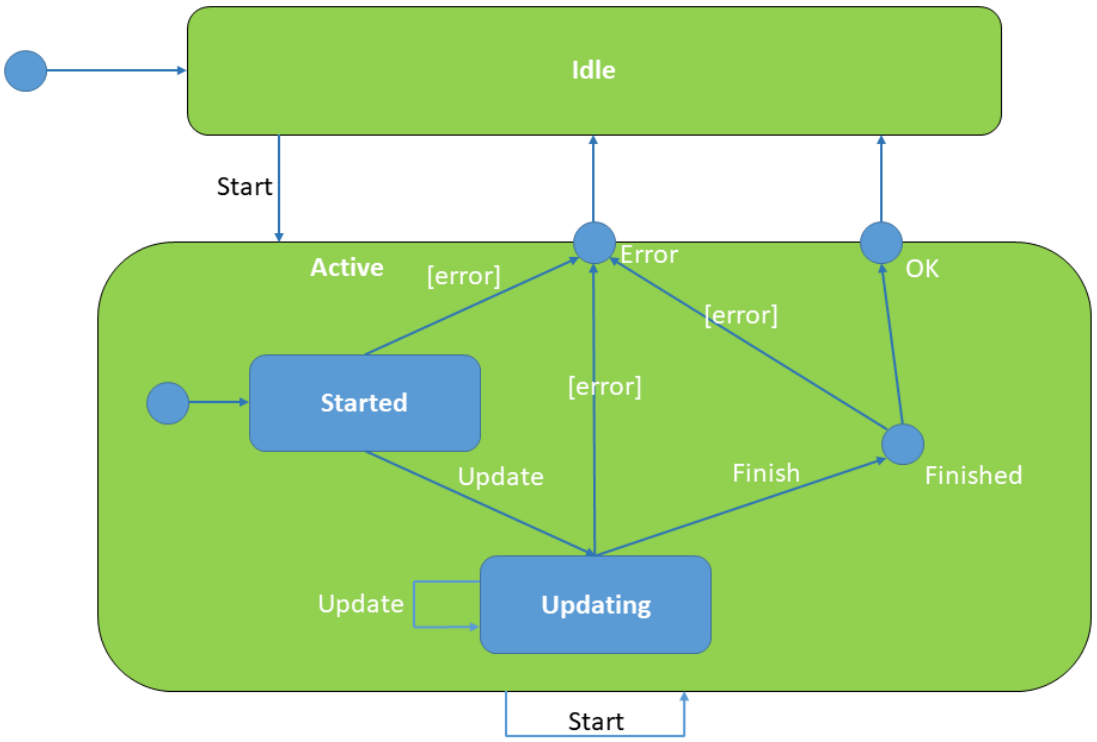
Idle-ból Started: A Crpyto Driver Object hardver erőforrásai és a workspace inicializálása egy Job futtatásához.

Started-ból Updating: A Crypto Primitive megkapja a felhasználó által nyújtott első adatrészt. Bármely köztes számítás eredménye a Workspace-ben lesz eltárolva.

Updating-ből Updating: A Crypto Primitive megkapja a következő adatrészt a felhasználótól.

Updating-ből Idle: Bármely szükséges feldolgozás utáni művelet elvégzése itt történik. A végső eredmény elküldésre kerül a felhasználónak és a Job Idle állapotba kerül. A Workspace felszabadul.

A 7. ábrán látható a működés, illetve az állapot változások:



7. ábra Állapotgép a Crypto Job-okról

A műveletek kombinálhatóak és a kombinált műveletek megegyeznek azzal, mintha külön hívná őket a felhasználó START, UPDATE, FINISH sorrendben.

4.5.3 Crypo Job-ok feldolgozása

Mivel némely Crypto Job-ok feldolgozása jelentősen hosszú időbe telhet és hardveres erőforrások használhatóak bizonyos Crypo Primitve-k gyorsítására, amelyek nem lehetnek megosztva egymással konkurens módon futó Job-ok között, ezért a Crypto Stack lehetőséget nyújt szinkron és aszinkron végrehajtásra.

4.5.3.1 Aszinkron végrehajtás

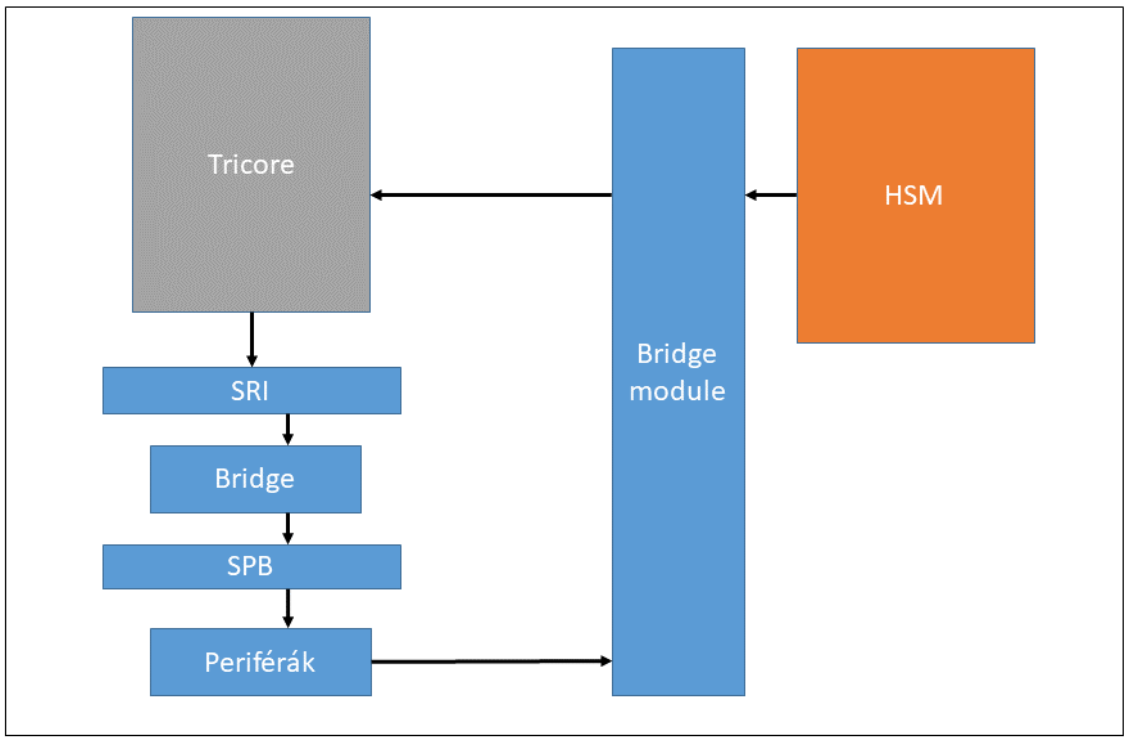
Az aszinkron végrehajtás lehetővé teszi a hívónak, hogy a Job eseményszerűen legyen végrehajtva és az eredményről később legyen értesítve, anélkül, hogy folyamatosan várnia kelljen a Crypto Stack-re, amíg az végrehajtja az egyes lépéseket, a valószínűleg hosszú Job-on.

4.5.3.2 Szinkron végrehajtás

A 4.2. pontban felsorolt interfészekhez tartozó függvények végrehajtása mind szinkron.

5 Mikrovezérlő által nyújtott Kriptográfiai Funkciók

A beágyazott rendszerben egy TriCore mikrokontroller foglal helyet, amely egy 32 bites mikrokontroller-DSP egyesítése. Egy magos felépítésű és real-time beágyazott rendszerek megvalósítására van optimalizálva. A TriCore mikrokontrolleren fut a beágyazott (thyssenkrupp által fejlesztett) operációs rendszer, így ezen futnak a főbb feladatokat ellátó programok, mint például a szervó vezérlése. A Crypto Stack jelentős rész is itt foglal helyet. A Tricore-on kívül a beágyazott egységben van egy Hardware Security Module úgynevezett HSM bővítmény. Ennek feladata, hogy megfelelő számítások elvégzését gyorsítja. A Crypto Driver egy része ezen a processzoron fut. A Tricore memóriájának egy része meg van osztva a HSM-el és ezen keresztül képesek kommunikálni. A HSM oldalon futó program észreveszi, ha a memóriában változás történt olyan módon, hogy elvégzendő számítása van. A HSM oldalon több gyorsító foglal helyet, így elősegítve például a Hash illetve AES műveletek gyorsítását. Ahogy a 8. ábrán is látható, a kommunikáció több csatornán keresztül folyik. Például a TriCore perifériákkal az SPB (Simple Peripherial Bus-on) keresztül. Illetve a Shared Resource Interconnect Bus-on keresztül.

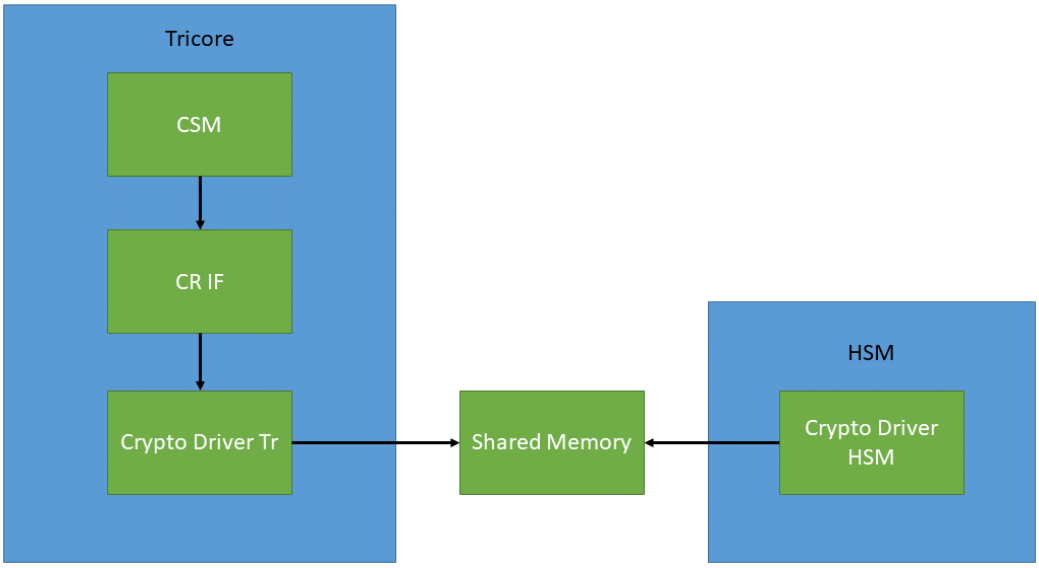


8. ábra TriCore és HSM kapcsolata

6 Megvalósítás

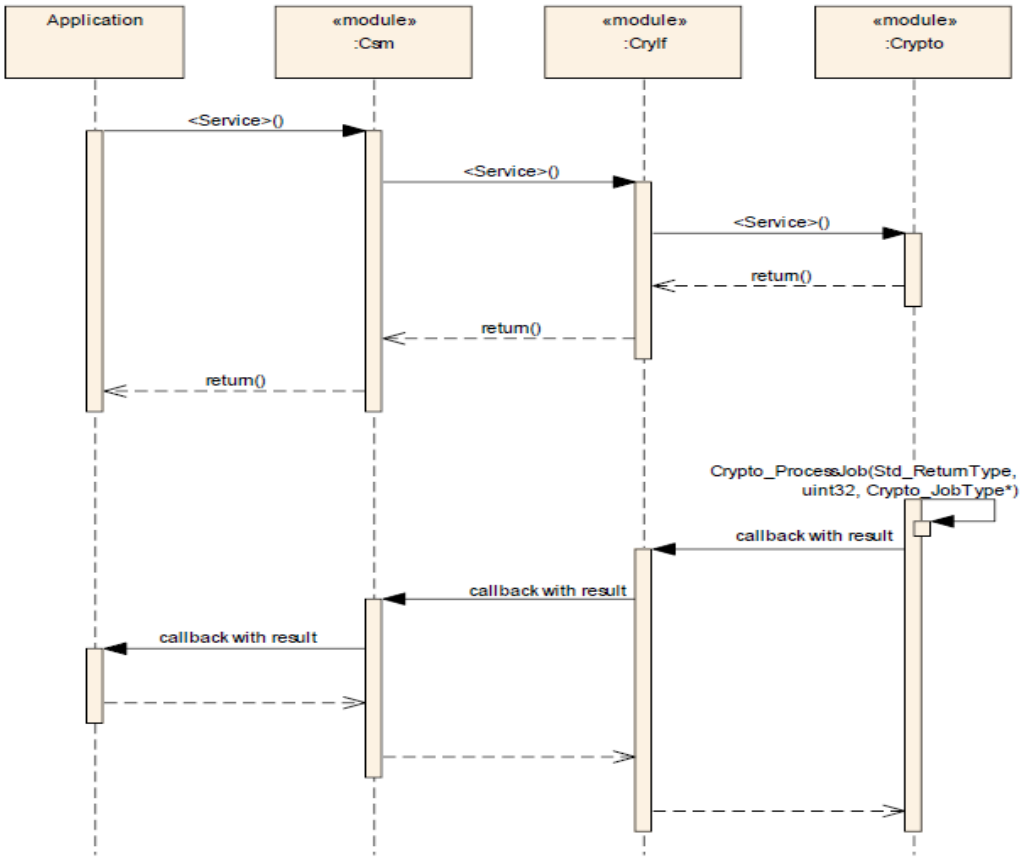
6.1 Tervezés

A megvalósítandó függvényeknek két fő feladatuk van, a kapott tanúsítványok adott mezőinek elérése és azoknak megfelelő struktúrákba való elhelyezése illetve egy kapott tanúsítvány illetve tanúsítvány lánc ellenőrzése. A megvalósítandó két függvény neve a `CryptoHSM_CertificateParse` illetve `CryptoHSM_CertificateVerify` kell, legyen a szabvány alapján [12]. A megvalósítandó függvényeknek meg kell felelnie az AUTOSAR Crypto Driver szabványában leírtaknak. Erre a leírásra lehet alapozni a funkcionalitást és ezáltal ismertek a bemeneti és kimeneti paraméterek, illetve az is, hogy milyen hibákat kell kezelni, illetve milyen módon kell kezelni azokat. A Crypto Driverben megvalósított adat struktúrákhoz kell alkalmazkodni, mivel a Crypto Driver-ben foglalnak helyet a függvények. A Crypto Driver-en belül pedig a HSM oldalon. A függvények ez esetben csak a Crypto Driver HSM oldali függvényeit látják és a közös memóriát sem érik el közvetlenül. Ez a kommunikáció a 9. ábrán látható. A tervezés során ezt figyelembe kellett venni és ennek megfelelően a saját szintről hívható függvényekkel elérni a megfelelő adatokat. A függvényeknek tehát a kapott adatokkal a HSM oldalon el kell végezni a számításokat, illetve ellenőrzéseket és ezek után a megfelelő értékekkel visszatérni a felsőbb rétegekbe.



9. ábra Hívási lánc a Crypto Stack-en

A 10. ábrán látható, hogy a függvényeket nem közvetlenül, fogja meghívni a felhasználó, hanem a Crypto Service Manager által nyújtott standardizált interfészt fogja használni. Ha egy *CryptoKeyElement* egy bizonyos mezőjét szeretne elkérni a felhasználó, például egy tanúsítványt akkor a `Csm_KeyElementGet` függvényt fogja meghívni. Ez a függvény még nem éri el közvetlenül a memóriában tárolt tanúsítványt ezért az eggyel alatta lévő szintre küld egy kérést a Crypto Interface-nek. A Crypto Interface-nek egy hasonló függvénye, a `CryIf_KeyElementGet` hívódik meg. Ez a Tricore oldalon lévő Crypto Driver `Crypto_KeyElementGet` függvényét hívja meg, amely a közös memória megfelelő helyére beírja a felhasználó és a felsőbb rétegek által adott adatokat és beállítja az `isUnderProcessing` nevű változót igaz értékűre. A HSM oldalon lévő Crypto Driver-nek egy függvénye a `cryptoHsm_KeyServiceDispatcher` bizonyos időnként kiolvassa ennek a változónak az értékét, és amennyiben igaz értékű, úgy elkezdi feldolgozni azokat az adatokat, amelyeket a felsőbb rétegek nyújtanak. Ezután a HSM oldali függvény elvégzi a kért számítást és átírja a közös memóriának azt a részét, amivel a TriCore oldali függvények visszatérhetnek a felhasználóig.



10. ábra Szekvenci diagram egy hívásról a Crypto Stack-ben

A függvényeknek mindkét esetben a megfelelő adatokat kell megszerezni, hogy azokon műveleteket tudjanak elvégezni. Mivel a függvények által használt struktúrák eltérőek lehetnek a Crypto Driver-ben használtaktól, ezért ehhez olyan függvényekre lesz szükség, amelyek a Crypto Driver-ben használatos struktúrák adatait átadják a megvalósítandó függvényeknek. Emellett olyan függvény is fog kelleni, amely, a kiszámított eredmény a megfelelő struktúrába menti. Az adatok átvezetése után a számításokat a WolfSSL által nyújtott API segítségével lesz célszerű megoldani.

A programot minél kisebb egységekre célszerű szétbontani, hogy a segédfüggvények elemi feladatokat lássanak el, ne legyenek túl hosszúak. Ez a megoldás átláthatóvá teszi a programot és sokkal beszédesebb wrapperekkel lehet hívni a függvényeket.

6.1.1 Kulcs Struktúra a Crypto Stack-ben

A Crypto Stackben eltárolt tanúsítványok felépítésével a feladat során meg kellett ismerkednem, ugyanis ebből az adatstruktúrából kell kinyerni a megfelelő adatokat.

Egy kulcs a Crypto Driverben egy specifikus kulcs típusra utal vagyis *CryptoKey*-re. A kulcs elemek szolgálnak adatok tárolására. Ez az adat lehet egy tanúsítvány vagy egy kezdőérték, amely egy AES titkosításhoz szükséges. A továbbiakban *CryptoKeyElement* [10].

Az AUTOSAR alapján megvalósított Crypto Stack-ben a tanúsítványok úgy nevezett *CryptoKey* struktúrák *CryptoKeyElement* mezőiben foglalnak helyet. A megvalósításban ennek a típusa a *CryptoHSM_KeyType* struktúra. Ebben a típusban négy mező foglal helyet, amely a 3. táblázatban látható:

3. táblázat CryptoKey Struktúra

Típus	**Mező neve**
`uint32`	`cryptoKeyId`
`CryptoHSM_KeyElementType*`	`keyElementRef`
`uint32`	`keyElements`
`boolean`	`isValid`

A `cryptoKeyId` azonosító, amely egyértelműen meghatároz egy *CryptoKey* példányt. A `keyElementRef` mezőben van eltárolva, a kezdőcíme annak a tömbnek, amely a *CryptoKey* típus *CryptoKeyElement* részeit tartalmazza. A `keyElements` mező *CryptoKeyElement* elemek darabszámát adja meg. Az `isValid` mező *CryptoKey* struktúrának az állapota lehet érvényes illetve érvénytelen, ez ebben a mezőben tárolódik.

A megvalósításban a *CryptoKeyElement* struktúra típusa a *CryptoHSM_KeyElementType,* amelynek felépítése a 4. táblázatban látható:

4. táblázat CryptoKeyElement struktúra

Típus	**Mező neve**
`uint32`	`keyElementId`
`CryptoKeyElementAccessType`	`elementReadAccess`
`CryptoKeyElementAccessType`	`elementWriteAccess`
`uint8*`	`keyElement`
`uint32`	`keyLength`
`boolean`	`allowPartialAccess`
`boolean`	`isRomElement`

A `keyElementId` azonosító egyértelműen meghatároz egy *CryptoKeyElement* példányt. Erre azért van szükség, mivel a függvényhívások során ezzel lehet beazonosítani, illetve megkeresni azokat az adatokat, amelyekkel műveleteket kell végezni. Az `elementReadAccess` mező definiálja, hogy kik olvashatják a megadott *CryptoKeyElement*-et. Az `elementWriteAccess` mező pedig ugyanezt az írási jogokkal teszi. Ezekre azért van szükség, mivel ha olyan függvény írná vagy olvasná ezeket az adatokat, akinek nincs rá jogosultsága azzal hibás vagy még rosszabb esetben meghamisított adatok kerülhetnének a memóriába. A `keyElement` pointer mutat a tényleges adatot tartalmazó memóriacímre. Valójában egy tanúsítvány ezen a mezőn keresztül érhető el. A típusból jól látszik, hogy a

tanúsítványok bájt tömbökben vannak tárolva, bináris formában. A `keyLength` mező meghatározza, hogy maga a *CrytoKeyElement*, illetve maga a tanúsítvány milyen hosszú. Erre azért van szükség mivel egy függvénynek tudnia kell, hogy milyen hosszan helyezkedik el a memóriában az adott kezdőcímtől a hasznos adat, nehogy túlcímezzen és érvénytelen adatot olvasson ki, illetve túl kevés adatot, ezzel hibás eredményeket szolgáltatva. Az `allowPartialAccess` mező engedélyezi, vagy letiltja az *CryptoKeyElement* írását és olvasását, olyan adatokra amlyek a *CryptoKeyElement*-nél kisebbek. A `isRomElement` mező megadja, hogy a *CryptoKeyElement* a ROM-ban foglal-e helyet.

A Crypto Driver-ben a *CryptoHsm_KeyType* típus példányai egy *CrpytoHsm_Keys* nevű tömbben foglalnak helyet. Ebben a tömbben konfigurációtól függően több különböző *CryptoKey* található. A konfiguráció az AUTOSAR ARCHITECT nevű szoftverrel, történik, ahol megadható, hogy hány *CryptoKey* foglaljon helyet a *CryptoHsm_Keys* tömbben, illetve azok tulajdonságait. Megadható, hogy milyen ID-k tartozzanak az egyes elemekhez, mi legyen a nevük, illetve kezdőértékként, megadható például egy tanúsítvány, amely a megfelelő *CryptoKeyElement*-ben lesz eltárolva, a program indulásakor. Az alapértelmezett konfigurációban 13 elem foglal helyet a tömbben. Arról, hogy az egyes *CryptoKeyElement*-ekben milyen adat van tárolva, nincs túl sok információ, ezért a `keyElementId`-hoz tartozó beszédes makrók definiálása megkönnyíti a megértést, ezek az 5. táblázatban találhatók.

5. táblázat CryptoKeyElement példányok

Key Element	**Key Element Name**	**Key Element Id**
Certificate	CRYPTO_KE_CERTIFICATE_DATA	0
Signature Algorithm	CRYPTO_KE_CERTIFICATE_SIGNATURE_ALGORITHM	22
Subject Public Key	CRYPTO_KE_CERTIFICATE_SUBJECT_PUBLIC_KEY	1

Signature	CRYPTO_KE_CERTIFICATE_SIGNATURE	28

6.1.2 WolfSSL tanúsítvány struktúra

A WolfSSL könyvtár több féle támogatást nyújt a tanúsítványok kezelésére és tárolására. Jelen esetben, amikor a tanúsítványok bináris formában állnak rendelkezésre, a legpraktikusabb módja a tárolásnak, a *WOLFSSL_CTX* struktúrában való tárolás, másnéven wolfssl kontextusban. A kontextus globális változókat tárolhat SSL kapcsolatokhoz és a hozzájuk tartozó információkat a tanúsítványokról. Egy *WOLFSSL_CTX* használható tetszőleges számú WolfSSL Objektumra. Ez lehetőséget nyújt például a hiteles CA tanúsítványok listájának betöltésére. Amennyiben csak egy tanúsítványt töltünk be a *WOLFSSL_CTX* struktúrába, úgy az több mezőben is eltárolódik, viszont amennyiben egy láncot töltünk be úgy a lánc egyes elemei a *WOLFSSL_CERT_MANAGER* mezőben tárolódik. Ezen belül a *Signer* típusokat tartalmazó caTable tömb tartalmazza az egyes lánc elemeket, illetve egy tanúsítvány esetén is tartalmazza azt [13]. A struktúra megoldáshoz szükséges mezői a 6. táblázatban láthatók:

6. táblázat WolfSSL-ben használt struktúrák főbb mezői

Típus	**Mező Neve**
word32	pubKeySize
const byte*	publicKey
int	nameLen
char*	name
DerBuffer*	derCert

A feladat során fontos lesz elérni a tanúsítványhoz tartozó publikus kulcsot, `pubKeySize` illetve a tanúsítványt, `derCert`. Ehhez a publikus kulcs mérete is kell, mivel ezt a kulcsot át kell másolni, egy másik memóriacímre.

6.2 Segédfüggvények a Crypto Driver oldalon

6.2.1 CryptoHsm_KeyElementGet

A Crypto Stack-ben az egyes *CryptoKey*-ek illetve *CryptoKeyElement*-ek elérésének módja függ attól, hogy egy függvény melyik szintről kívánja elérni azokat. A legfelső rétegben lévő Crypto Service Manager két függvényt nyújt a megfelelő adatok, írására, illetve olvasására. Ezek a már fent leírt módon hívják az egyre lejjebb lévő rétegek hasonló funkcióit, amíg elérnek a Crypto Driver-be. Ezek a `Csm_KeyElementSet` és a `Csm_KeyElementGet` nevű függvények.

Ezek a függvények, illetve a standardizált interfész nem elérhető a Crypto Driver-ből, így ezekhez hasonló Crypto Driver-ben helyet foglaló függvényre van szükség a program helyes futásához. A fent említett két függvény mintájára két függvényt kell létrehozni a Crypto Driver-ben. Ezek a `CryptoHsm_KeyElementGet`, illetve a `CryptoHsm_KeyElementSet`. Ezeken kívül még szükség van egy wrapper függvényre, amely másolást nem végez.

A `CryptoHsm_KeyElementGet` egy adott *CryptoKeyElement* bizonyos adatait fogja átmásolni a megadott memória címre. A függvény végső soron meg fogja keresni az adott azonosítókkal ellátott *CryptoKeyElement*-et és a megadott memóriacímre fogja másolni azt. A függvénynek a Csm-es változathoz hasonlóan 4 bemeneti paramétere kell, hogy legyen. A bemenetek a következők:

- `uint32 cryptoKeyId` azonosító, amely alapján egyértelműen meghatározható, hogy, melyik *CryptoKey*-ből van szükség adatokra
- `uint32 keyElementId` azonosító, amely megadja, hogy melyik *CrpytoKeyElement*-et szükséges beolvasni
- `uint8* keyPtr` egy pointer, amely arra a memóriacímre mutat, ahová a *CryptoKeyElement*-et át kell másolni.
- `uint32* keyLengthPtr` egy pointer, amely arra a memória területre mutat, ahol a *CryptoKeyElement* számára fenntartott buffer mérete van eltárolva. Azért kell pointeresen átadni ezt az értéket, mert a függvény visszatértével ide fogja beírni, hogy mekkora volt az adott bufferbe másolt *CryptoKeyElement* mérete.

A függvény egy `CryptoKeyServiceInputOutput` típusú példányt hoz létre lokálisan keyServiceIo néven. Ez a változó alkalmas *CrpytoKeyElement*-ek tárolására, illetve emellett az eredmény számára fenntartott memóriára mutató pointer és hossz is eltárolható benne. Miután a `keyServiceIo` példány megfelelő mezőit kitöltötte a függvény egy másolást végző függvényt hív, amely a Crypto Driver HSM oldalán van megvalósítva és elvégzi a másolást a megadott memória területre. A visszatérési értéke Std_ReturnType típusú, tehát három féle eredményt nyújthat, amelyet kezelni kell. Az érték lehet `E_OK`, `E_NOT_OK`, `CRYPTO_E_KEY_NOT_AVAILABLE`, `CRYPTO_E_KEY_READ_FAIL`, `CRYPTO_E_SMALL_BUFFER`. Ezek rendre a művelet sikerességét illetve sikertelenségét jelzik, továbbá, hogy az azonosító alapján meghatározott *CryptoKey* nem volt elérhető, illetve az olvasási engedély meg lett tagadva. Az utolsó visszatérési érték pedig azt jelzi, hogy a művelet során beolvasott *CryptoKeyElement* mérete meghaladta, a másolásra kijelölt buffer méretét és emiatt sikertelen volt a másolás.

6.2.2 CryptoHsm_KeyElementSet

A második függvény a *CryptoHsm_KeyElementSet*, az előzőhez hasonlóan egy másolást végez, azzal a különbséggel, hogy ebben az esetben a bemeneti adatot, tehát amelyet szeretnénk eltárolni, az eszköz memóriában, kívülről kell megadni. Ennek a függvénynek is 4 bemeneti paramétere van.

- `uint32 cryptoKeyId` az előző esettel ellentétben ez az azonosító azt a *CryptoKey*-t határozza meg ahova az általunk ismert *CryptoKeyElement*-et másolni szeretnénk.
- `uint32 keyElementId` azonosító, meghatározza a *CryptoKeyElementet*, ahova másolni szeretnénk a megfelelő adatokat.
- `const uint8* keyPtr` egy pointer, amely arra a kezdőcímre mutat ahol az általunk ismert *CryptoKeyElement* van tárolva.
- `uint32 keyLength` egy számérték, amely megadja, hogy a *CryptoKeyElement* hossza mekkora.

A `Get` függvényhez hasonlóan itt is egy `CryptoKeyServiceInputOutput` típusú `keyServiceIo` példányt használ a függvény a bemeneti paraméterek tárolására, lokálisan. Ezek után hívódik meg a `cryptoHsm_InternalKeyElementSet`

függvény, amely a `keyServiceIo` változóban tárolt értékek alapján elvégzi a másolást. A másolást a háttérben egy `cryptoHsm_MemCopy8` nevű függvény végzi, amely egy for ciklussal a megadott címtől kezdődően bájtonként átmásolja az adott hosszúságú *CryptoKeyElement*-et a megadott címre.

6.2.3 CryptoHsm_KeyElementPointerGet

A `CryptoHsm_KeyElementGet` a megadott azonosítóval rendelkező *CryptoKeyElement*-et átmásolja, egy másik memóriaterületre. Mivel az eszköz memóriája igen kevés, ezért annak érdekében, hogy ne kelljen új memóriaterületet lefoglalni, egy olyan függvényt kell létrehozni, amely csak a *CryptoKeyElement*-ek kezdőcímét, illetve hosszát adja át. Ebben az esetben az adatokat használó függvények az eredeti memóriacímen lévő példánnyal fognak dolgozni, illetve fogják azt módosítani. Erre a célra, lett létrehozva ez a rövid függvény. Bemenetként három paramétert vár egy azonosítót, amely alapján a *CryptoKey* meghatározható. A *CryptoKey*-ek egy *CryptoHsm_KeyType* típusú példányokat tartalmazó tömbben foglalnak helyet, amely globálisan elérhető. Ebben a tömbben határoz meg egy *CryptoKey*-t a `cryptoKeyId`. Ezen belül több *CryptoKeyElement* foglalhat helyet, ez a konfigurációtól függ. Ebben az esetben feltételezzük, hogy minden *CrpytoKey* csak egyetlen *CryptoKeyElement*-et tárol, így ennek az azonosítóját már nem kell továbbadni, ezért csak 3 bemeneti paramétert kap a függvény. A másik két paraméter egy-egy pointer egy `uint8**` típusú `cert` és egy `uint32*` típusú `length` változó. A függvény a bemeneti cert pointert állítja a *CryptoKeyElement* megfelelő mezőjére, ahol az adat kezdődik, jelen esetben egy keyElement nevű mezőre. Ennek elérése a következő:

```
CryptoHsm_Keys[cryptoKeyId].keyElementRef[1].keyElement;
```

A harmadik bemeneti paramétert, a `length` pointert pedig hasonló képpen a *CryptoKeyElement* megfelelő mezőjére a következő módon:

```
CryptoHsm_Keys[cryptoKeyId].keyElementRef[1].keyLength;
```

Így paraméteresen átadja a megfelelő adatokra mutató pointereket a függvény. A híváskor figyelni, kell, hogy a `cert` és `length` változókat cím szerint kell átadni.

6.3 Crypto_CertificateParse

Az első megvalósítandó függvény neve a `Crypto_CertificateParse`. A függvény feladata, hogy egy adott tanúsítványból, kiolvassa, hogy a tanúsítványt milyen algoritmussal írták alá, a publikus kulcsot, illetve az aláírást és ezeket rendre mentse el a `CRYPTO_KE_CERT_SIGNEDDATA`, `CRYPTO_KE_CERT_PARSEDPUBLICKEY` és `CRYPTO_KE_CERT_SIGNATURE`, nevű *CryptoKeyElement*-ek megfelelő mezőibe. A függvény visszatérési értéke Std_ReturnType típusú, amely megadja, hogy sikeres volt-e a művelet. A típusnak három értéke lehet. `E_OK` amennyiben a számítás sikeres volt. `E_NOT_OK` amennyiben a művelet sikertelen volt, illetve `E_BUSY` amennyiben sikerteltelen volt a függvényhívás mivel a Crypto Driver Object használatban van [12].

A függvény egyetlen paramétert kap bemenetként, egy `uint32` típusú `cryptoKeyId` nevű azonosítót. Ezzel az azonosítóval egyértelműen meghatározható egy *CryptoKey*. A függvény több lokális változót használ, ezek a 7. táblázatban vannak összefoglalva:

7. táblázat Crypto_CertificateParse lokális változói

Típus	**Változó neve**
Std_ReturnType	ret
const uint32	keyElementId
WolfSSL_CTX*	wolfsslContext
WOLFSSL_CERT_MANAGER*	certManager
wolfSSL_method_func	method
Word32	pubkeySize
const byte*	pubkeyBuffer
uint8*	cert
uint32	length

Ahogy a táblázatban is látszik, mivel a függvény visszatérési értéke `Std_ReturnType` típusú ezért a ret változóval fog visszatérni a függvény, amelynek az értéke a futás során több ellenőrzéskor módosíthat.

A *CryptoKeyElement*-eket azonosító `uint32` típusú számot többnyire kívülről adják meg, de jelen esetben ismerjük, hogy minden *CryptoKey* melyik *CryptoKeyElement*-je milyen adatot tartalmaz. Így az azonosító értéke mindig konstans `CRYPTO_KE_CERTIFICATE_DATA` makró, amely mindig azt a *CryptoKeyElement*-et jelöli, ahol a tanúsítványt tartalmazó adatok vannak.

6.3.1 WolfSSL inicializálása

Futás során az adatokat a tanúsítványból a WolfSSL könyvtár segítségével lehet kinyerni. A könyvtár használatához azt először inicializálni kell. Az inicializálást, egy külön wrapper függvény végzi, az initWolfSSL, amely a WolfSSL könyvtárat felkészíti a használatra, alaphelyzetbe állítja annak értékeit. A WolfSSL a tanúsítványokat, illetve tanúsítvány láncokat a wolfsslContext nevű struktúrában tárolja. Ennek kezdő értéke NULL. Mivel ebben a pontban a kontextus még nem mutat értelmes adatra ezt be kell állítani. Ezen a `wolfSSL_CTX_new` függvényhívással lehet változtatni, amely visszatérési értéke egy *WOLFSSL_CTX* struktúrára mutat, erre a címre állítjuk rá a már korábban létrehozott kontextust. Ezen kívül be kell állítani, hogy milyen verziójú protokollt szeretnénk használni az ellenőrzésekhez, jelen esetben 1.2 és verziójú TLS-t használunk ezt a method változóban `wolfTLSv1_2_client_method_ex` értékkel adjuk meg. Az utolsó lépés a `wolfSSLContext` struktúra `WOLFSSL_CERT_MANAGER` mezőjének elérése, ugyanis ebben foglalnak helyet a tanúsítványok a kontextuson belül. Erre külön függvényt biztosít a könyvtár és egy `getCertManager(wolfSSLContext)` hívással a korábban létrehozott certManager pointer már a valódi certificate managerre mutathat.

6.3.2 Adatok szétbontása

Az inicializálás után az első lépés a megfelelő *CryptoKeyElement* elérése. A `CryptoHsm_KeyElementPointerGet` meghívásával a cert pointert a függvény a megfelelő tanúsítványt tartalmazó memóriacímre állítja, illetve ennek a hosszát tartalmazó length változónak adja át a visszatérési értékét.

```
CryptoHsm_KeyElementPointerGet(cryptoKeyId, &cert,
&length);
```

A tanúsítvány kezdőcímének és hosszának ismeretében azt már a WolfSSL által nyújtott kontextusba lehet tölteni. Erre a célra egy újabb segédfüggvény szolgál. Az

`addCert` nevű függvény feladata a cím szerint kapott tanúsítványt a kapott kontextusba tölteni. A WolfSSL könyvtár `wolfSSL_CTX_load_verify_buffer` függvénye a kapott adatokkal tölti fel a megfelelő mezőket jelen esetben a `Signer` típusú `caTable` első elemét. A függvényhívásnál, meg kell adni a tanúsítvány típusát, amely jelen esetben `SSL_FILETYPE_ASN1`, tehát bináris formátumú.

Ezek után már a megfelelő mezőket elérve külön változókba menthetők ki a várt adatok például publikus kulcs esetén:

```
pubkeySize = certManager->caTable[0]->pubKeySize;
pubkeyBuffer = certManager->caTable[0]->publicKey;
```

Hasonlóan az előzőekhez ebben az esetben is külön kell a elmenteni, a tömb kezdőcímét, illetve hosszát. Ezeket a másik két elvárt mezőre is meg kell tenni az aláírás-ra és az aláíró algoritmusra is.

Miután megvannak a szükséges információk, azokat az előírt *CryptoKeyElement*-ben kell tárolni. Erre a célra lett létrehozva a `CryptoHsm_KeyElementSet` függvény, amelyet mind a három esetben meg kell hívni.

A függvénynek a megadott `cryptoKeyId`-t kell átadni, mint *CryptoKey*-t jelölő azonosítót, a másolni kívánt tömböt, illetve annak hosszát és a három *CryptoKeyElement* azonosítójaként szolgáló makrót: `CRYPTO_KE_CERTIFICATE_SUBJECT_PUBLIC_KEY`, `CRYPTO_KE_CERTIFICATE_SIGNATURE`, `CRYPTO_KE_CERTIFICATE_SIGNATURE_ALGORITHM`.

6.3.3 Hibakezelés

A futás közben előforduló esetleges hibák esetén a specifikációban megadott módon kell visszatérnie a függvénynek. Amennyiben a WolfSSL függvényhívásai vagy azok visszatérése közben adódna probléma, úgy azokat is kezelni kell.

- Amennyiben a modul nincs inicializálva, és a Crypto Driver error detection funkciója be van kapcsolva, úgy `E_NOT_OK` értékkel kell visszatérni.
- Ha a `cryptoKeyId` értéke túlmutat a *CryptoHsm_Keys* tömb elemein, `E_NOT_OK` értékkel kell visszatérni. Mivel a tömb mérete ismert és az ID-k

egyre növekvőek, elég azt megnézni, hogy a `cryptoKeyId` ebben a tartományban van-e benne.

A WolfSSL inicializálása során minden függvény után meg kell vizsgálni, hogy az helyesen futott-e le. Ennek ellenőrzését támogatja a könyvtár ugyanis a függvények visszatérési értéke az `SSL_SUCCESS` érték, amennyiben az sikeresen végrehajtódott.

6.4 Crypto_CerificateVerify

A második megvalósítandó függvény a `Crypto_CerificateVerify`. Ennek a feladata a megadott azonosítójú tanúsítványt hitelesíteni egy másik azonosítóval megadott tanúsítvány alapján [12].

A függvénynek három bemeneti paramétere van két azonosító uint32 típusú id. `CryptoKeyId` és `verifyCryptoKeyId`. Illetve egy `Crypto_VerifyReslutType*` típusú verifyPtr. Az első két paraméter jelöli a tanúsítványokat, a `cryptoKeyId` határozza meg azt a tanúsítványt, amely alapján a másikat hitelesíteni kell. A harmadik, `verifyPtr` pedig arra a memóriacímre mutat, ahová a hitelesítés eredményét kell menteni. A visszatérési érték a `Crypto_CertificateParse` függvényhez hasonlóan `Std_ReturnType`, amely azt jelzi, hogy a művelet sikeres, illetve sikertlen volt.

Ahogy a 8. táblázatban is látszik, a függvény több lokális változót használ, a WolfSSL könyvtár funkcióinak használatára, illetve a visszatérési értékek ellenőrzésére, tanúsítványok tárolására.

8. táblázat Crypto_CertificateVerify lokális változói

Típus	**Változó Neve**
WOLFSSL_CTX*	wolfSSLContext
WOLFSSL_CERT_MANAGER*	certManager
wolfSSL_method_func	method
uint8*	cert
uint32	length

CryptoHsm_ReturnType	ret

A WolfSSL könyvtár inicializálása az előző függvény esetén alkalmazottal teljesen megegyezik, mivel ugyan olyan tanúsítványokkal dolgozik a függvény. Az inicializálás egyes lépéseinek ellenőrzésére hasonlóan a visszatérés helyességét vizsgáljuk, `SSL_SUCCESS` értéknek kell lennie.

Az első tanúsítvány megszerzéséhez a `CryptoHsm_KeyElementPointerGet` függvény használható. Ez a cert pointert a megfelelő azonosítójú *CryptoKeyElement* címére állítja és visszatért annak hosszával, átadva azt a length változónak. Ezek után az ellenőrzés következik, amelyhez a kapott tanúsítványt a WolfSSL által nyújtott kontextusba töltünk az `addCert` nevű segédfüggvénnyel. `addCert(wolfSSLContext, cert, length);`

Ebben a pontban a kontextusban foglal helyet az ellenőrzéshez használt tanúsítvány. Ez alapján fog történni a hitelesítés. Mivel a `cryptoKeyId` alapján meghatározott megbítható tanúsítvány mentésre kerül, ugyan azt a változót lehet használni, a hitelesíteni kívánt tanúsítvány tárolására. Ezek után az ellenőrzés a WolfSSl egyik függvényével az alábbi módon néz ki:

`wolfSSL_CertManagerVerifyBuffer(certManager, cert, length, SSL_FILETYPE_ASN1);` A függvény a paraméterként kapott tanúsítványt a kontextusban lévő tanúsítványok alapján hitelesíti, amelyben jelen esetben csak egy tanúsítvány van. A függvény visszatérési értéke igen sokféle lehet. Amennyiben például hibás egy tanúsítvány, úgy külön hibaüzenettel tér vissza. Ezt részletesebben a 7. pontban van kifejtve. Amennyiben a visszatérés érteke `SSL_SUCCESS`, úgy az ellenőrzendő tanúsítványt hozzáadjuk a kontextushoz, illetve az előző tanúsítványokhoz. Erre azért van szükség, hogy a későbbiekben amennyiban nem csak egy tanúsítványt, hanem tetszőleg hosszú tanúsítvány láncot szeretnénk ellenőrizni, úgy ilyen módon elemenként végig lehessen haladni a láncon és egyesével hitelesíteni az elemeit. A szabvány meghatározza, hogy bizonyos esetekben milyen értékkel kell visszatérnie a függvénynek.

- Amennyiben a modul nincs inicializálva úgy `E_NOT_OK` értéket kell adnia.
- Amennyiben a `cryptoKeyId` kimutat a *CryptoKey*-eket tartalamazó tömbből, úgy `E_NOT_OK` jelzéssel kell visszatérni.

- Ha a `verifyPtr` érétke NULL a művelet után, akkor `E_NOT_OK` a visszatérési érték.
- Amennyiben a hitelesítés sikeres, akkor `E_OK` a visszatérési érték.

7 Tesztelés

A függvények implementálása közben folyamatosan ellenőriznem kellett az egyes részfeladatok helyes működését, amennyiben azt lehetett külön vizsgálni. Első körben például, csak a WolfSSL által nyújtott funkciókkal független tanúsítványok ellenőrzését végeztem a Crypto Stack-en kívül, PC-n.

A PC-n való tesztelést sok eszköz segíti, például egyes változók értékét egész egyszerűen lehet kiiratni a képernyőre. Illetve a debug funckiós segítségével utasításról utasításra haladva lehet a programot végignézni és az esetleges hibákat megkeresni.

A függvények működését különböző tesztesetekkel vizsgáltam. A függvényeknek megfelelő értékkel kellett visszatérni minden esetre. Az ellenőrzést két részre lehetett bontani. Amennyiben a Crypto Stack részéről adódott hiba vagy ha a függvények nem működtek helyesen. A tesztelés már az eszközön folyt és az ehhez fejlesztett debug szoftver segítette az ellenőrzéseket. A teszteléshez a *CryptoHsm_Keys* tömbben lévő 8-as ID-val rendelkező *CrpytoKey*-t használtam, mivel ez egy custom, felhasználó által személyre szabható *CryptoKey* a tanúsítványok tárolására. Ez a *CryptoKey* több *CryptoKeyElement*-et tartalmaz, ezekben az 5. táblázat mezői közül is többet tartalmaznak. A tanúsítványt a tömbnek a megfelelő adatokat tartalamazó *CryptoKeyElement*-be kellett tölteni ez a következő függvényhívással történt:

```
ret = CryptoHsm_KeyElementSet(cryptoKeyId, keyElementId, keyPtr, keyLength);
```

A debug szoftver lehetőséget nyújt az egyes változók értekeinek vizsgálatára így a globálisan láthato crypokeys tömb megfelelő elemét vizsgálva jól láthatóvá vált, hogy a művelet sikeresen végrehajtódott-e.

A `Crypto_CerificateVerify` függvény teszteléséhez, két tanúsítványt kellett betölteni a megfelelő struktúrába, a fent említett módon. Az első az ellenőrzés alapjául szolgáló hiteles tanúsítvány, a második pedig az ellenőrzendő tanúsítvány.

Miután a tanúsítványok a helyükre kerültek, megtörténhetett a függvények ellenőrzése. Ehhez azokat megfelelő `cryptoKeyId`-val meghívva, lépésről lépésre követtem a futásukat. A debug szoftver lehetőséget nyújtott erre, illetve breakpoint-ok használatára is. A futás során először azt kellett vizsgálni, hogy a cryptoKeyId alapján

eléri-e a függvény a tanúsítványokat. Ehhez első sorban a `CryptoHsm_KeyElementGet` függvényt használtam és utána a `CrytoiCryptoHsm_KeyElementPointerGet`-et. Mindekét függvénynek sikeresen kellett működni, de a lényegesebb a második volt, mivel az nem másol, csak pointert állít a megfelelő memóriaterületre, így csökketve a memóriafelhasználást. Ennek a működésnek a helyességét úgy lehetett ellenőrizni, hogy a debug szoftver változók értékének figyelését lehetővé tévő funkcióját használtam. Ezzel egyértelműen meg lehetett vizsgálni, hogy az utasítások lefutása után a pointerek megfelelő adatra mutatnak-e.

Ezt követte a WolfSSL könyvtár használatának vizsgálata, amely egyszerűbb visszatérési értékek vizsgálatával történt, a korábbi pontokban tárgyaltak szerint.

A tanúsítványok betöltése után már csak két dolgot kellett vizsgálni, az ellenőrzés helyességét vagy parse helyességét, illetve a megfelelő visszatérési értékek *CrypotKeyElement*-ekbe való visszatöltését. Az ellenőrzéshez meg kellett vizsgálni, hogy a megfelelő parancsok után a változókban megfelelő értékek kerültek-e. Például a publikus kulcs kinyerése után:

`pubkeySize = certManager->caTable[0]->pubKeySize;` az általam ismert tanúsítványhoz tartozó publikus kulcs hossza került-e a `pubkeySize` változóba.

A helyes adatok CryptoKeyElement-ekbe való töltését ugyan csak a `CryptoHsm_KeyElementSet` függvénnyel végeztem és a fent leírtaknak megfelelően ellenőriztem a helyes máködést.

A tesztelés végére minden esetre helyes futást mutattak a függvények.

8 Összefoglalás

A féléves feladatom során, sikeresen elsajátítottam az AUTOSAR szabvány által meghatározott Crypto Stack működését és felépítését. Ezen kívül megismertem a kriptográfia alapjait és a tanúsítványkezeléssel kapcsolatos problémákat. Megismertem három, kriptográfiai problémákra használható szoftver könyvtárat, és ezekből eggyel megoldottam a féléves feladatom. A feladat során egy, az autóiparban használatos mikrokontrolleren futó szoftvert egészítettem ki, olyan módon, hogy az képes legyen tanúsítványláncokat is ellenőrizni, ezáltal növelve a biztonságot. Az erőforrástigényeket tovább lehet csökkenteni és adott esetben másik szoftver könyvtárral megoldani a problémát. A szoftvert további funkciókkal lehet kiegészíteni a későbbiekben, tovább növelve a biztonságot.

Irodalomjegyzék

[1] https://tools.ietf.org/html/rfc4158

[2] https://www.ietf.org/rfc/rfc1422

[3] OpenSSL User Manual (1. fejezet) 2018, Version 1.1.0, OpenSSL Software Foundation https://www.openssl.org/docs/man1.1.1/man1/

[4] OpenSSL User Manual (3. fejezet) 2018, Version 1.1.0, OpenSSL Software Foundation https://www.openssl.org/docs/man1.1.1/man3/

[5] wolfSSL User Manual (2. fejezet), August 2019, Version 4.1.0, wolfSSL Inc. https://www.wolfssl.com/docs/wolfssl-manual/ch2/

[6] wolfSSL User Manual (7. fejezet), August 2019, Version 4.1.0, wolfSSL Inc. https://www.wolfssl.com/docs/wolfssl-manual/ch7/

[7] wolfSSL Documentation and User's Guide, June 2017, Version 4.1.0, wolfSSL Inc. https://www.wolfssl.com/docs/wolfssl-manual/static-buffer-allocation/

[8] Requirements on Crypto Stack 4.3.1 Classic Platform AUTOSAR https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SRS_CryptoStack.pdf

[9] Lovász - Pelikán - Vesztergombi: Diszkrét matematika. 255. old. Typotex Kiadó, 2006. ISBN 963-9664-02-2

[10] Specification of Crypto Service Manager 4.3.1 Classic Platform AUTOSAR https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_CryptoServiceManager.pdf

[11] Specification of Crypto Interface 4.3.1 Classic Platform AUTOSAR https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_CryptoInterface.pdf

[12] Specification of Crypto Driver 4.3.1 Classic Platform AUTOSAR https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_CryptoDriver.pdf

[13] https://www.wolfssl.com/doxygen/structSigner.html