Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Pálics Marcell Ferenc

KRIPTOGRÁFIAI MŰVELETEK TELJESÍTMÉNYÉRTÉKELÉSE AUTÓIPARI MIKROKONTROLLEREN

KONZULENS
Bajor Péter

KÜLSŐ KONZULENS
Kiss Miklós

BUDAPEST, 2022

Tartalomjegyzék

Összefoglaló **6**

Abstract **7**

1 Bevezetés **8**

1.1 Kiberbiztonság megjelenése az autóiparban 8

1.2 Kiberbiztonság feladata 10

1.3 Feladat meghatározása 11

1.4 Kutatási célok 12

2 Elméleti háttér **13**

2.1 Vezérlőegységek kiberbiztonsági funkciói 13

2.1.1 Biztonságos rendszerindítás (Secure Boot) 13

2.1.2 Biztonságos szoftverfrissítés (Secure Update) 14

2.1.3 Biztonságos belső kommunikáció (SecOC) 14

2.1.4 Diagnosztikai hozzáférési szintek (Security Access) 15

2.2 Alkalmazott kriptográfiai műveletek 16

2.2.1 Hash algoritmusok 16

2.2.2 Szimmetrikus kulcsú rejtjelezés 17

2.2.3 Aszimmetrikus kulcsú rejtjelezés 17

2.2.4 Digitális aláírások 18

2.2.5 Üzenethitelesítési kód 19

2.2.6 Véletlen számok szerepe 20

2.3 Kriptográfiai műveletek végrehajtási ideje 21

2.3.1 Hardveres biztonsági modulok 21

3 Implementáció **23**

3.1 Mérési metodológia 23

3.2 Működési elv 25

3.3 Felhasznált eszközök 27

3.3.1 Hardver 27

3.3.2 Szoftver 27

3.4 Keretrendszer szoftverarchitektúrája 28

3.4.1 Hardver absztrakciós réteg 28

3.4.2 Platform absztrakciós réteg 30

3.4.3 Szolgáltatási réteg 32
3.4.4 Applikáció réteg 32
3.4.5 Közös függőségek 32
3.5 Kriptográfiai meghajtók 33
3.5.1 mbedTLS függvénykönyvtár 34
3.5.2 TC3XX: Hardware Security Module 35
3.5.3 K3XX: Hardware Security Engine 35
3.6 Fordítási szabályok 36
3.7 Programozó eszköz vezérlése 37
3.8 Integrációs lehetőségek 38
3.8.1 Új teszt hozzáadása 38
3.8.2 Új mikrovezérlő hozzáadása 39
3.9 Kezelőprogram 40
3.9.1 Felhasználói felület áttekintése 40
3.9.2 Belső működés 41
3.9.3 Statisztika nézet 43
3.10 Továbbfejlesztési lehetőségek 44
3.10.1 További kriptográfiai függvénykönyvtárak integrálása 44
3.10.2 AUTOSAR szoftverarchitektúrába illesztés 45
3.10.3 Megfelelés a MISRA-C irányelveknek 46
4 Eredmények 47
4.1 Elkészült tesztek 47
4.2 Szoftveresen számított eredmények összehasonlítása 48
4.3 Hardveresen gyorsított számítás összehasonlítása 49
4.4 Értékelés 49
5 Összefoglalás 51
6 Irodalomjegyzék 52
7 Függelék 54
A Abbreviációk 54
B Kódrészletek 55
B-1 Fordítási időben ki és bekapcsolt implementációk 55
B-2 Közösen használt típusok definíciója (részlet) 55
B-3 mbedTLS statikus buffer létrehozása (részlet) 56
B-4 mbedTLS fordítása statikus könyvtárként (részlet) 56

B-5 NXP S32 K3XX makefile (részlet) 57
B-6 Infineon Aurix TC3XX makefile (részlet).. 58
B-7 Támogatott kriptográfiai meghajtókat tartalmazó sémá 58
B-8 Programozó eszköz irányítása PRACTICE script-el (részlet)......................... 59
B-9 Teszt paramétereket tartalmazó fejlécfájl (példa)... 59

HALLGATÓI NYILATKOZAT

Alulírott **Pálics Marcell Ferenc**, 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, 2022. 05. 28.

...…………………………………………….

Pálics Marcell Ferenc

Összefoglaló

Napjaink modern gépjárművei több száz különböző elektronikai komponenssel rendelkeznek, melyek mind önálló beágyazott rendszerként üzemelnek, mindezen felül különböző autóipari kommunikációs technológiák révén hálózatot is képesek alkotni. Az alkatrészek beszállítóinak meg kell felelniük autóipari szabványoknak és egyedi gyártói igényeknek, melyekhez hozzátartoznak a kiber- és információbiztonságra irányuló követelmények.

A thyssenkrupp Components Technology Hungary Kft. elektromos kormányrendszerek (EPAS – Electric Power Assisted Steering) fejlesztésével és gyártásával foglalkozik. Egy erre dedikált – a vállalat termékeinek kiberbiztonsági kérdéseivel és megoldásaival foglalkozó – csoport tagjaként egy olyan projektben volt lehetőségem részt venni, mely a régebbi és újabb gyártású mikrokontroller családok kriptográfiai számítási képességeinek összehasonlítását hivatott biztosítani. A kormányrendszert vezérlő programkód integritását és hozzáférhetőségét biztosító digitális aláírás ellenőrzésének egyre növekvő számítási kapacitásigénye miatt nem megengedhető olyan vezérlők használata, melyek nem rendelkeznek a műveletek végrehajtásához szükséges erőforrásokkal.

Jelen dolgozat témája egy általam fejlesztett keretrendszer bemutatása, melynek célja különböző mikrovezérlők validációja kriptográfiai teljesítményük vizsgálata alapján. Fejlesztésekor fő szempont volt a szoftver architektúra megfelelő megtervezése generikus programozási modellek és a Standard C könyvtár segítségével, további mikrokontrollerek egyszerű integrálhatóságát támogatva. Az elkészített szoftver mellett bemutatásra kerülnek a vizsgált kriptográfiai algoritmusok által kiszolgált autóipari beágyazott rendszerekben használt információbiztonsági mechanizmusok, valamint a keretrendszer fejlesztéséhez létrehozott, az AUTOSAR konzorcium által megalkototthoz hasonlatos szoftverarchitektúra.

Abstract

Today's modern vehicles have hundreds of different electronic components, all of which operate as a stand-alone embedded system, but are also capable of forming a network through various automotive communication protocols. Component suppliers must meet automotive standards and individual manufacturer claims, which include cyber and information security requirements.

thyssenkrupp Components Technology Hungary Kft. (Ltd.) deals with the development and production of Electric Power Assisted Steering (EPAS) systems. As a member of a department dealing with cyber security issues and providing solutions for the company's products, I had the opportunity to participate in a project dedicated to comparing the cryptographic computing capabilities of older and newer microcontroller families. Due to the increasing computational power required to verify the integrity of the application that controls the steering system, the use of microcontrollers that do not have the resources to perform the operations is not permitted.

The topic of my thesis is to present a software framework I have developed to validate different microcontrollers by benchmarking their cryptographic performance. The main consideration during development was to create the most portable code possible, using generic programming models and the Standard C library, easing the later integration of additional microcontrollers. In addition to the framework, the information security mechanisms used in automotive components served by measured cryptographic algorithms are presented, as well as the software architecture created for the development of the framework, similar to that created by the AUTOSAR consortium.

1 Bevezetés

1.1 Kiberbiztonság megjelenése az autóiparban

A ’80-as évek autóipari mérnökei felismerték annak szükségességét, hogy a járművek egyes elektronikusan vezérelt komponensei képesek legyenek kommunikálni egymással. 1986-ban megjelent, a Robert Bosch GmbH által kifejlesztett CAN buszrendszer az első alacsony költségű, robosztus és hibatűrő megoldás melynek segítségével a járművek elektronikus vezérlőegységei (*Electronic Control Unit - ECU*) kommunikálhattak. Ezek a vezérlőegységek mind a jármű egy jól meghatározott funkcióját ellátó beágyazott rendszerei. Kezdetben a kifejezés csak a motorvezérlő egységet takarta, a digitális technika rohamos fejlődésével azonban megjelent az igény, hogy a járművek minél több komponense programozható logikával vezérelt legyen. A CAN rendszert rövidesen követte annak kiterjesztése (CAN-FD) majd a FlexRay és egyéb autóipari kommunikációs technológiák. Az Intel által már egészen 1987-től gyártott CAN vezérlő áramkör rohamos terjedése és általánosságban a digitális technika és a mikroprocesszorok rohamos fejlődése lehetővé tette az autóipar számára új funkciók vagy meglévő funkciók elektronikusan vezérelt megfelelőjének adaptálását. 1981-ben megjelenik az első navigációs rendszer (Honda), 1987-ben az első elektronikus kipörgésgátló rendszer (Mercedes-Benz), 1990-ben pedig megjelenik – még bőven a ma már széles körben használt és megszokott okoseszközeink előtt – az első érintéssel vezérelhető multimédiás rendszer.

Az ezredfordulót követően számos tényező tovább gyorsította az egy járműben megtalálható vezérlőegységek számának növekedését. Míg korábban csak gépjárművön belüli kommunikációról beszélhettünk, addig a mai modern járművek képesek információcserére külső eszközökkel, a smart megoldások világában egyre elterjedtebb vezeték nélküli technológiák révén mint például GPS, GSM, Bluetooth, ezen komponensek további vezérlőegységek meglétét igénylik. Az elektromos járművek térnyerése rákényszerítette az autóipari beszállítókat, hogy a mechanikus vagy hidraulikus energiával működő komponenseket is átdolgozzák elektromechanikus energiával használható változatra, amely további vezérlő logika fejlesztését igényli. Nem elhanyagolható tény továbbá, hogy a régebbi pusztán elektromechanikus rendszerek

végfelhasználó általi testreszabása is a legtöbb esetben valamely digitális technikát igényelt. Napjaink prémium kategóriás autóinak ECU száma 100-150-es nagyságrenden is mozoghat.

Nagyjából tízszeres ütemben növekszik emellett azoknak a szoftvereknek a mérete, melyet az ECU-ban található processzorok futtatnak. Mint bármely szoftver, az autó komponenseinek logikáját futtató szoftver is igényelhet karbantartást, frissítést. Az autóipari szabványok által leírt diagnosztikai célú külső eszközzel való fizikai csatlakozási lehetőségek (Fizikai réteg: *On-Board Diagnostic* csatlakozó) és magas szintű protokollok (Alkalmazási réteg: *Universal Diagnostic Service*) biztosítják, hogy egy komponens fejlesztése során, annak visszahívásakor, vagy akár szervizelés során az egyes ECU egységek szoftvere letölthető, feltölthető legyen, emellett különböző diagnosztikai alprogramok indítására és hibakódok olvasására is van lehetőség.

Az információbiztonság feladatai között szerepel annak megakadályozása, hogy a fent említett folyamatokat illetéktelen szereplő tudja kezdeményezni, irányítani, befolyásolni. A komponens gyártók szellemi termékének védelme mellett még fontosabb tényező hogy egyes vezérlőegységek biztonságkritikus rendszerek, azaz szoftverük módosítása vagy meghibásodásuk az ember egészségének és a környezetének károsításához vezethet. Ilyen beágyazott rendszernek minősülnek a fékrendszerek és kormányrendszerek, míg egy multimédiás rendszer például nem. Mivel a járművek diagnosztikai rendszerének kompromittálása alapvetően sokáig csak fizikai hozzáféréssel volt kivitelezhető ezért a gyártók és beszállítók nem fektettek nagy hangsúlyt sem interfészszintű, sem pedig komponensszintű információvédelmi (*security*) mechanizmusok implementálására, kizárólag a üzembiztosságot (*functional safety*) tartották szem előtt.

Az autóipari információbiztonság megjelenése mely ellátja ezeket a feladatokat és kezeli a kapcsolatos felmerülő problémákat egy a dolgozat írásakor kifejezetten friss ágazatnak számít, mindössze pár éves múltra tekint vissza. Az elmúlt két évtizedben több alkalommal is demonstrálták különböző kiberbiztonsági kutatással foglalkozó szakemberek és lelkes amatőrök az autók biztonsági problémáit és kihasználási lehetőségeit, a valódi töréspont azonban csak 2015-ben következett két IT biztonsági szakértő Charlie Miller és Chris Valasek által demonstrált problémán keresztül. Az említett szakemberek egy Jeep Cherokee modellen mutatták be, hogyan lehet a Uconnect

okostelefonra megjelent applikáción keresztül, a fejegységre kapcsolódva távoli hozzáférést szerezni a gépjármű belső kommunikációs hálózatához. A publikált sérülékenységet kihasználva képesek voltak a rádióállomások között váltogatni, elindítani az ablaktörlőt, és alacsony sebesség esetén akár az elektromos fékrendszert is letiltani. A gyártásért felelős Fiat Chrysler Automobiles vállalat 1.4 millió járművet hívott vissza, amely a vállalatcsoportnak súlyos anyagi és reputációs veszteséget okozott. Charlie Miller szavait idézve: „Ez lehet az első alkalom, hogy tömeggyártott terméket [..autót..] ilyen mennyiségben visszahívnak egy szoftverhiba miatt.”. [1] Azóta eltelt évek során a vezetéknélküli támadási felületek száma azonban csak tovább nőtt, a potenciálisan kihasználható funkciók listájával együtt: a közelmúltban megjelent a gyártói igények között az *OTA* (*Over-the-air*) vezetéknélküli frissítési lehetőség az vezérlőegységek számára, mely alternatívát kínál a hagyományos diagnosztikai csatolófelületen való frissítésre, valamint a napjainkban kialakuló *Connected Car* technológiák révén újabb információbiztonsággal kapcsolatos problémákkal kell szembenéznie az autóiparnak. Az ipar szereplői felismerték a kiberbiztonsági célok szükségességét, és az azóta kidolgozott szabvány (ISO/SAE-21434, *Road vehicles – Cybersecurity engineering*) és regulációk már előírnak aktív kiberbiztonsági részvételt a fejlesztésben, használt eszközök validációján keresztül, a szoftvermodulok felülvizsgálatán át, egészen hardverközeli biztonsági problémák felderítéséig. A 2024 után tömeggyártási szakaszba (*SoP – Start of production*) lépett személygépjárművek már nem kapnak típusengedélyt amennyiben nem felelnek meg a fentebb említett szabályozásoknak.

1.2 Kiberbiztonság feladata

A kiberbiztonság (*cybersecurity / information security*) feladata a digitális adatok és eszközök védelme, az ellenük irányuló fenyegetésekkel szemben. Hálózati rendszerek és alkalmazások védelmével hozható kapcsolatba, minden olyan külső és belső támadási potenciál elhárításának gyakorlatával foglalkozik, melyek a kiberbiztonsági CIA modell valamely pontját érinthetik. [2] A CIA mozaikszó egy általános szempontrendszert takar, az információbiztonsággal kapcsolatos problémák nagyrésze a három tényező valamelyikébe besorolható:

Confidentiality – bizalmasság

A *bizalmasság* biztosítása, az információ megvédése minden annak megtekintésére jogosulatlan egyedtől. [3]

Integrity – sértetlenség

Az *sértetlenség* vagy *adatintegritás* az információ helyességét jelenti, elvesztése használhatatlan, félrevezető információt eredményezhet. [3]

Availability – elérhetőség

Az *elérhetőség* az információ rendelkezésre állását jelenti, az információbiztonság kontextusában jellemzően időben mérhető. [3]

$$Elérhetőség\ (t) = \frac{Teljes\ működési\ idő\ (t)}{hasznos\ működési\ idő\ (t) + működésképtelen\ eltelt\ idő\ (t)}$$

1.3 Feladat meghatározása

Az thyssenkrupp Components Technology Hungary Kft. elektromechanikus kormányrendszerek (*Electronic Power Assisted Steering*) fejlesztésével foglalkozik, tevékenysége kiterjed a tervezési szakasztól egészen a gyártásig és az azt követő terméktámogatás biztosításáig. A vállalat többféle kész platformot biztosít a vevők (gyártók) számára, emellett lehetőséget a rendszer egyes paramétereinek, funkcióinak finomhangolására/megváltoztatására. A különböző kormányrendszer platformok, eltérő vezérlő elektronikával rendelkeznek, a kormányrendszer szoftverének futtatása más-más számosságú és típusú mikrovezérlőn történik.

Az elektromos kormányrendszer biztonságkritikus beágyazott rendszernek minősül, emellett különböző autóipari, információbiztonsági és elektronikai szabványoknak is meg kell felelnie. Nem csak a működés során felmerülő problémákat kell tesztelni és szükségszerűen kezelni, hanem mint a vállalat szellemi termékének védelme is hangsúlyt élvez. A program utólagos módosítása, vagy futtatási időben történő befolyásolása különböző biztonsági mechanizmusok beépítésével elkerülhető, ehhez kriptográfiai algoritmusok használata szükséges. A szoftver feltöltése és letöltése az autóiparban szinte minden vezérlőegység által használt diagnosztikai protokollon keresztül, szintén igényel kriptográfiai műveleteket, ilyen például az asszimmetrikus

kulcspárok és digitális aláírások használata. Nem utolsósorban végrehajtási időben periódikusan több alkalommal is történhet integritás ellenőrzés és az egyes vezérlőegységek közötti kommunikáció is igényelhet rejtjelezési eljárásokat. Akadályt jelent azonban, hogy a platformok tervezésekor költséghatékonyan kell eljárni, az autóipari mikrovezérlő termékcsaládok általában erősen skálázhatók, viszont a tervezés óta eltelt évek alatt megváltozott kiberbiztonsági követelmények ugyanazon a hardveren erőforrásigényesebb kriptográfiai műveleteket igényelhetnek. A teljesítmény profilozás során előállhat olyan eredmény hogy egy adott kriptográfiai művelet használata miatt már nem teljesülnek egyes követelmények, például a központi feldolgozó egység kihasználtsága nem haladhat meg egy meghatározott szintet. A fentebb említett funkciók biztosítása, a számítási költség, végrehajtási idő (*overhead*) meghatározása és ezzel együtt az újonnan piacra dobott mikrovezérlők gyártói specifikációban meghatározott képességeinek validálása indokolta a 3. fejezetben tárgyalt szoftver keretrendszer létrehozását.

1.4 Kutatási célok

A vállalat termékeinek kiberbiztonsági kérdéseivel foglalkozó szervezeti egység gyakornokaként feladatként kaptam, hogy létrehozzak egy kriptográfiai műveletek futási idejének mérését végző szoftver keretrendszert. Feladata hogy a végfelhasználó által beállított paraméterek alapján egy sablonból létrehozzon egy végrehajtható programot, a támogatott autóipari mikrovezérlők számára. A paraméterek alapján a létrehozott futtatható állomány tartalmazni fogja egy kiválasztott kriptográfiai algoritmus egy szintén választott implementációját, amennyiben abból több áll rendelkezésre. A szoftver elkészítése során nagy hangsúlyt kap az erősen generalizált felépítés és kódolási paradigmák alkalmazása, annak érdekében, hogy a támogatott mikrovezérlők száma egyszerűen bővíthető legyen. A cél, a különböző platformok kriptográfiai számítási képességeinek összehasonlítása, legyen szó szoftveresen implementált vagy az egyes mikrovezérlők esetén hardveresen gyorsított algoritmusokról.

2 Elméleti háttér

Az alábbi fejezetben kerülnek bemutatásra az elektronikus vezérlőegységek jellemző kiberbiztonsági funkciói, emellett azon kriptográfiai műveletek melyek szükségesek ezen funkciók működéséhez. Tárgyalásra kerül ezen műveletek futási idejének és erőforrásigényeinek fontossága valamint a véletlen számok szerepe a kriptográfiában.

A fejezet tanulmányozása során az olvasó képet kaphat a dolgozat témájaként szolgáló keretrendszer szükségességéről, emellett, hogy milyen kihívásokkal kell szembenéznie az autóipar kiberbiztonsági területének.

2.1 Vezérlőegységek kiberbiztonsági funkciói

Az alábbi alfejezet tárgyalja az autóipari elektronikus vezérlőegységek jellemző kiberbiztonsági célokat szolgáló funkcióit.

2.1.1 Biztonságos rendszerindítás (Secure Boot)

A biztonságos rendszerindítás (*Secure Boot*), egy modern számítástechnikai eszközök által gyakran használt eljárás, manapság a számítógépektől kezdve az okostelefonokon át szinte minden eszköz támogatja és használja. Célja a futtatható szoftverek betöltése előtti ellenőrzés végrehajtása. Legismertebb változata a személyi számítógépek hagyományos BIOS (*Binary Input Output System*) rendszerét leváltó UEFI (*Unified Extensible Firmware Interface*) egyik funkciója, mely a személyi számítógépek számára nyújt többek között védelmet az operációs rendszereket betöltő programok módosítása és potenciálisan ezt kihasználó kártékony szoftverek ellen. [4] A Secure Boot működéséhez a betöltendő szoftver entitásokat előzetesen el kell látni egy *digitális aláírással*. A digitális aláírások őrzik egy adat pontos tartalmának lenyomatát. Az aláírásokat az úgynevezett *Secure Boot Manager* fogja ellenőrizni, ami az első szoftver entitás mely futtatásra kerül a indítási folyamat során. Amennyiben a betöltendő szoftvert módosították és nem került sor újabb aláírásra, a Secure Boot Manager nem fogja betölteni. A biztonságos rendszerindítás megvalósításához az aláírás ellenőrzéshez szükséges kulcsot tároló tanúsítvány a vezérlő védett memóriaterületén található, egy hardveres biztonsági modulban. A Boot Manager jellemzően a vezérlő egyszer írható

memóriájába kerül (*One-Time Programmable -OTP*), annak érdekében hogy a forgalomba helyezését követően semmiképp se legyen módosítható és az indítás mindig megbízható forrása legyen.

2.1.2 Biztonságos szoftverfrissítés (Secure Update)

A Secure Update feladata hasonlatos a Secure Boot-hoz, a frissítési folyamat során a feltöltött szoftver entitások aláírásának ellenőrzése a feladata. A frissítési folyamat történhet többféle protokollon keresztül, diagnosztikai vagy hibakereső interfészre való fizikai csatlakozás segítségével, azonban egyre nagyobb teret nyer a járműiparban még friss igénynek számító OTA (*Over-the-air*) szoftverfrissítési mechanizmus mely további kihívásokat jelent a megfelelő Secure Update megvalósítása során. A vezeték nélküli csatlakozás alapjaiban egy újabb támadási felületet képez, a letöltött szoftver forrásával szembeni bizalom elvesztése újabb fenyegetést jelenthet.

Az OTA frissítések letölthetők mobilhálozat, Wi-Fi vagy egyéb rádiófrekvenciás technológiák segítségével, a dolgozat írásának idejében főként elektromos járművek de a belső égésű motorral hajtott típusok prémium szegmenseiben is megjelenik. Az ellátási lánc elleni támadások súlyos problémákat okozhatnak az autóipar számára, például egy hamis forrásból érkező - magát szoftverfrissítésnek álcázó - kártékony szoftver felbecsülhetetlen károkat okozhat, ráadásul kivitelezésükhöz fizikai hozzáférés sem szükséges a járművekhez. [5] Az UNECE R156-os regulációja Európában már szabványosítja a gépjárművek szoftverfrissítési rendszerét. [6]

2.1.3 Biztonságos belső kommunikáció (SecOC)

A *Secure On-Board Communication* (továbbiakban SecOC) a különböző vezérlőegységek között megvalósított biztonságos kommunikációért felelős funkció egyben az autóipari szoftverfejlesztésben használt szoftverarchitektúra egyik szoftverkomponense. A jármű belső kommunikációs hálózatán létrehozott hálózati szegmensek (alhálózatok) tagjai adatokat küldhetnek egymásnak, például a sebességmérő szenzorok elküldik a műszerfal számára a megjelenítendő mennyiséget. A SecOC célja, hogy ezen adatok biztonságosan, integritásukat megőrizve továbbíthatók legyenek.

Egy alhálózaton található eszköz, kulcs elosztó kiszolgálóként működik, a kommunikációban résztvevő felek számára biztosít egy szimmetrikus kulcsot. A kulcs

segítségével létrehozható egy hitelesítő kód (2.2.5), melyet az elküldött üzenet mellé csatolva a kommunikációban résztvevő másik fél ellenőrizni tudja annak integritását és hitelességét, olyan módon, hogy a beérkező üzenetre helyileg számított hitelesítő kódot összehasonlítja a fogadott hitelesítő kóddal. A kulcs tartalmának egy része véletlenszerűen generált az első indítás alkalmával, míg a másik felét egy pillanatnyilag érvényesnek tekintett érték teszi ki, a csomag ismétléses/visszajátszott támadásokat megakadályozva ezzel.

2.1.4 Diagnosztikai hozzáférési szintek (Security Access)

Valamennyi vezérlőegységen megtalálható a *Universal Diagnostic Service* elnevezésű kiszolgáló szoftver. Az diagnosztikai kiszolgáló egy felsőbb rétegbeni magasszintű funkciókat megvalósító autóipari kommunikációs protokollt használ (UDS), mely megvalósítható a legtöbb adatkapcsolati rétegekre építve. (CAN, FlexRay) Funkciói között megtalálható a jármű szervízben értelmezhető hibakódjainak kiolvasása, azok kijelzésének törlése, ECU szoftverek frissítése, emellett használható egyes alacsonyabb szintű műveletek például bizonyos memóriacímek olvasására is. A diagnosztikai szolgáltatásokat a megfelelő szolgáltatási azonosító (*Service ID - SID)* segítségével érhetjük el, azonban bizonyos funkciók korlátozottan hívhatók, használatukra csak bizonyos hozzáférésére szintekre való jogosultság ellenőrzését követően kerülhet sor. A UDS protokoll működését leíró szabvány (ISO 14229-1) meghatároz ilyen szinteket: alapesetben például az indítást követően minden vezérlőegység diagnosztikai kiszolgálója az 0x01-es azonosítóval ellátott alapértelmezett hozzáférési szintről indul. (*DefaultSession*) Továbbá a szabvány leírja, hogy ezen a hozzáférési szinten milyen szolgáltatások legyenek elérhetők, a szoftver feltöltéséhez, letöltéséhez már más hozzáférési szintek szükségesek, de a szabványon felül a jármű gyártója és a komponens gyártója is meghatározhat a gyártás különböző szakaszain használtakat.

A 0x27-es szolgáltatási azonosító (*Security Access*) segítségével lehetséges hozzáférési szintet váltani, a jogosultság ellenőrzése *Challenge-Reponse* elven működik. Diagnosztikai szoftverek a jármű egyik komponensén futó UDS szolgáltatástól kérnek a SID megfelelő paraméterezésével egy véletlenszerű értéket (*seed*) a komponenstől. Ezen információból számítja ki a szoftver a megfelelő választ egy közösen a felek által ismert algoritmus alapján. A válasz visszaküldését követően a program fogad egy pozitív vagy egy negatív választ. Előbbi esetén tudja hogy sikeres volt az autentikáció, és használhat

korlátozott szolgáltatásokat. A Security Access megfelelő implementálásának kulcsa egy helyes működésű véletlenszám generátor használata illetve a szabványnak és kiberbiztonsági elvárásoknak megfelelő algoritmus alkalmazása, mely akár érvényes seed-válasz párokkal való rendelkezés esetén sem visszafejhető ameddig a diagnosztika szolgáltatás oldalán tárolt titkos kulcs ismeretlen. A SecurityAccess szolgáltatást a jövőben kiegészíti az UDS szabvány által előírt újabb lehetőség, a 0x29-es azonosítóval ellátott ún. *token* alapú autentikáció, mely az UDS 2020-as szabványmódosításában került definiálásra.

2.2 Alkalmazott kriptográfiai műveletek

A kriptográfia rejtjelezéssel, titkosírással, azok előállításával és megfejtésével foglalkozó tudományág. Az információbiztonság fenntartásának elengedhetetlen eszközeit ezen matematikai koncepciók összesége biztosítja. A keretrendszerben vizsgált és a fejezetben tárgyalt kriptográfiai algoritmusok szükségesek a fentebb említett kiberbiztonsági funkciók biztosításához, elemi műveleteknek is tekinthetők.

2.2.1 Hash algoritmusok

Hash függvények egy - az adott algoritmustól függő - fix méretű kimenetre tudnak leképezni egy tetszőleges méretű bemenetet. Működésük determinisztikus, azaz ugyanazon X bemenet esetén mindig a hozzá tartozó Y kimenetet kapjuk. Halmazelméleti szempontból egy egyértelmű hozzárendelésnek feleltethető meg, azonban nem kölcsönösen egyértelmű, hiszen az alaphalmazunk (tetszőleges méretű bemenetek) mérete jelentősen nagyobb mint a képhalmaz (hash értékek). Lehetséges tehát két különböző bemenetre ugyanazon hash érték a kimenet ($H(x) = H(y)$ ahol $x \neq y$), egyben az is látszik hogy a hash függvények nem rendelkeznek sem matematikai vagy informatikai értelemben vett inverz függvénnyel.

Algoritmuselméleti szempontból sokféle felhasználási lehetőséget biztosít, hálózati és biztonsági szempontokból szinte mindig valamely üzenet sértetlenségének ellenőrzésére, jelszavak tárolására, üzenethitelesítésre, digitális aláírások létrehozására használják. [7]

2.2.2 Szimmetrikus kulcsú rejtjelezés

A klasszikus, már az ókorban is használt rejtjelezési megoldások során az olvasható üzenetekből (*plaintext*) legtöbbször egy „szimmetrikus” célú kulcs segítségével készítettek titkosított üzeneteket (*ciphertext*). Ennek jelentése, hogy egyazon kulcs volt használható a titkosítási és visszafejtési művelet elvégzésére, a kommunikáló partnerek mindegyikének ismernie kellett tehát kulcsot. [8]

Számos biztonsági problémát jelenthet a kommunikációs partnerek között használt kulcsok cseréje, ezért nem nevezhető a legbiztonságosabb rejtjelező technikának, azonban a bonyolultabb több kulcsot használó megoldásokkal szembeni egyszerűsége, sebessége és üzenet méretének megtartása illetve egyes esetekben csökkentése miatt léteznek egyértelmű a felhasználási területei, például nagyobb mennyiségű adat küldése. A járművek vezérlőegységei közötti biztonságos belső kommunikáció megvalósítása során is használnak szimmetrikus titkosítást.

2.2.3 Aszimmetrikus kulcsú rejtjelezés

Az aszimmetrikus rejtjelező algoritmusok egy kulcspár segítségével működnek. A kulcspárt egy privát (titkos) és egy publikus láthatóságú információ alkotja, melyeknek egymással matematikai relációban kell lenniük, viszont a titkos kulcs nem előállítható a publikus kulcsból. A működési elve, hogy a publikus kulcs segítségével titkosított üzenetek csakis annak titkos párjával fejthetők vissza, a publikus kulcs ismeretében nem. A publikus kulcs mint nevéből is adódik szabadon megosztható, ellenben a privát kulcs kiszivárgása kompromittálja a kommunikációs csatornát.

Publikus-privát kulcspárokat használó titkosító algoritmusok között legszélesebb körben használt az *RSA (Rivest–Shamir–Adleman)*, azonban beágyazott rendszerek esetén a jelenleg még kevésbé széles körben adoptált *ECDSA (Elliptic Curve Digital Signature Algorithm)* algoritmust célszerű használni. Előnye, hogy kulcshossz mellett lényegesen nagyobb erőforrásigénnyel fejthető *brute-force* módszerekkel.

A rejtjelező algoritmusok biztonsága mérhető annak függvényében, hogy legalább hány próba szükséges a titkosított információ megfejtéséhez vagy nagyságrendileg nagyon nagy valószínűséggel megtalálni a megoldást. A legtöbb algoritmus esetében a biztonság mértéke nem egyenlő a kulcshosszal, ami csupán ezen próbálgatások számának maximumát adja, de nem feltételezhető hogy a teljes tartomány

próbája szükséges a visszafejtéshez. [9] Az ECDSA használatával rövidebb kulcsok segítségével nagyobb mértékű biztonságot érhetünk el az RSA algoritmussal szemben, emellett a számítási igény is alacsonyabb, így ideális kisebb teljesítményű rendszerek, mint például egy mikrokontroller számára. A számítási kapacitás a számítástechnika területének fejlődésével együtt növekedik, különösen a ma már elosztott felhő alapú rendszerek bevonásával olyan titkok (például jelszavak) visszafejtése is lehetséges belátható időn belül melyek 5-10 évvel ezelőtt nagyságrendekkel hosszabb időt vettek volna igénybe.

Az RSA használata esetén szükséges pár évenkénti kulcshossz és egyéb paraméterek auditálása nem megoldható olyan rendszerek esetén mint például a gépjárművek komponensei melyek akár évtizedekig futtatják ugyanazt a szoftvert.

2.2-1. Táblázat - Részlet az ajánlott kulcsméretekből a BSI kutatási eredményeiből [12]

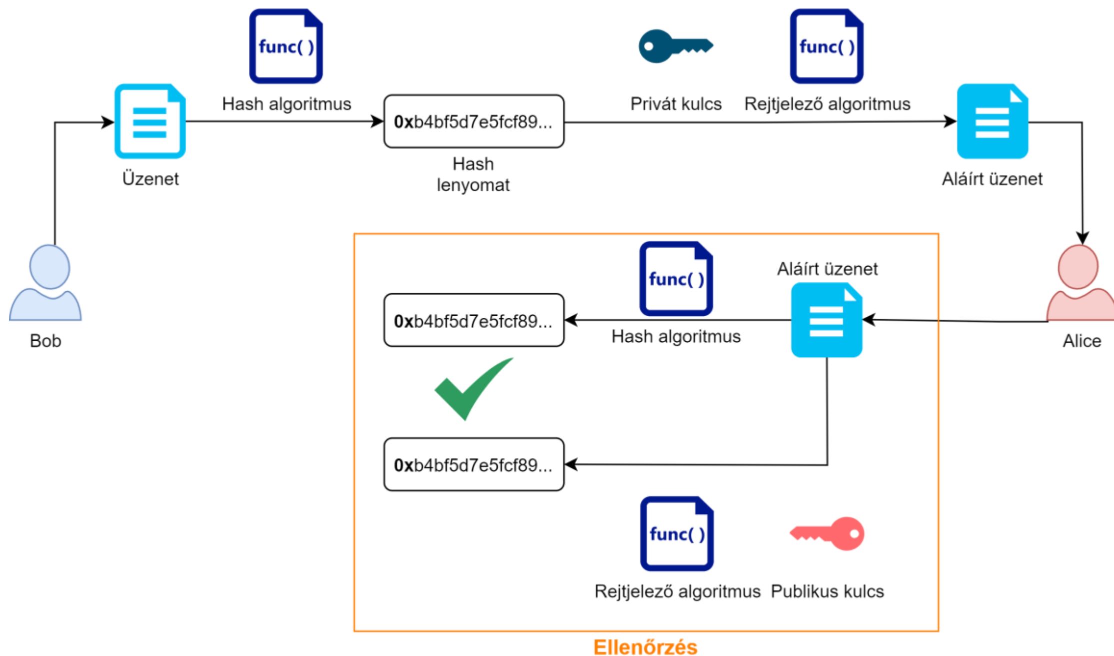
Várhatóan meddig tekinthető biztonságosnak	Ajánlott RSA kulcshossz (Factoring modulus)	Ajánlott ECDSA kulcshossz
2020-2022	2000	250
2023-2026	3000	250

2.2.4 Digitális aláírások

A digitális aláírás egy üzenethez mellékelt információ, a szerepe, hogy ellenőrzésével kiderül, hogy az üzenet az érkezése előtt megsérült vagy módosították-e, emellett az üzenet küldője megegyezik e az elvárttal: az üzenet *hitelességét* és *integritását* hivatott bizonyítani. Elengedhetetlen eszköze a fentebb tárgyalt információbiztonságot szolgáltató mechanizmusok implementálásához. Digitális aláírások létrehozásához és hamisíthatóságuk elkerüléséhez két különböző célú kriptográfiai eljárásra van szükség: egy hash algoritmusra és egy aszimmetrikus kulcsú titkosító algoritmusra.

Első lépésként az aláírandó üzenetből, egy aktuális követelményeknek megfelelő hash algoritmus segítségével képezni kell egy lenyomatot. Második lépésként az elkészült hash lenyomatot titkosítani kell a küldő privát kulcsával, így jön létre az üzenet digitális aláírása. Az elküldött üzenethez mellékelni kell az arról készült aláírást. A fogadó fél megkapja az aláírást, majd a publikus kulcs segítségével visszafejti és az eredményként

kapott hash-t, összehasonlítja a megkapott üzenetből számított hash értékkel. Amennyiben a két lenyomat megegyezik a fogadó megbizonyosodhat afelől, hogy egy kommunikációs csatornán megjelenő harmadik személy nem változtathatta meg az eredeti üzenetet, mert a privát kulcsot birtokolnia kellett volna ahhoz, hogy aláírja az üzenetet.



2.2-1. Ábra: Üzenet aláírása, majd ellenőrzésének folyamata (saját szerkesztés)

Felvethet biztonsági problémákat a publikus kulcsok tárolása is. Ugyan a birtoklása egy támadó számára nem jelentheti a titkosítás megtörését, azonban amennyiben az eszközön nem-felejtő memóriájában eltárolt publikus kulcsot képes lecserélni, abban az esetben egy saját kulcspár használatával, saját digitális aláírással ellátott üzeneteket fogalmazhat meg, adott esetben akár az eredeti szoftvert is lecserélheti egy saját aláírt példánnyal.

2.2.5 Üzenethitelesítési kód

Az üzenethitelesítési kód (*Message Authentication Code - MAC*), üzenetek tartalmának a hitelességét és integritását megőrző információ mely az üzenettel együtt kerül elküldésre. Működési elvben hasonlít a digitális aláírásokhoz, azonban szimmetrikus titkosítást használ, azaz a kommunikációban résztvevő feleknek előzetesen meg kell egyezniük egyetlen titkos kulcsban mely a rejtjelezésért és a visszafejtésért egyaránt felelős, emellett azonos titkosító algoritmust kell használniuk. Lényeges

különbség, hogy míg a aszimmetrikus kulcsú (nyilvános kulcsú) titkosítás esetén, az titkos aláíró kulcsot egyetlen fél birtokolhatja, addig a szimmetrikus titkosítás esetén minden résztvevő rendelkezik vele, azonban lényegesen gyorsabb és használatával kisebb adatcsomag érhető el, ami folyamatos üzenetváltás mellett nem valósítható meg a bonyolultabb, több kulcsot használó eljárások segítségével.

Az üzenethitelesítési kódot használó biztonsági mechanizmusok gyengepontja a megosztott titkos kulcs kiszivárgása, a támadó, a kulcs birtokában hitelesnek látszó üzeneteket hamisíthat. [10]

2.2.6 Véletlen számok szerepe

A számítástechnika egyik legrégebbi problémája, hogy nehezen állíthatók elő véletlen előfordulású adatok. A digitális technika alapja hogy egy megtervezett stabil rendszer determinisztikusan működik, ideális üzem esetén nem keletkeznek váratlan állapotok (például hazárdjelenség). Ennek ellenére sok felhasználási területen, de különösen a kriptográfia területén szükség van arra véletlenszerű paramétekre.

Az digitális aláírásokhoz használt kulcspárok előállítása során, a diagnosztika jogosultság ellenőrzéshez szükséges *seed* érték generálásához és a biztonságos belső kommunikáció működése is egy megfelelő véletlenszám generátor használatán múlik. Egy hibásan implementált véletlenszám generátor kiszámíthatóvá teszi a rendszer biztonsági funkcióit, például egy nem megfelelően inicializált véletlenszám generátor eredményezheti, hogy indítást követően a diagnosztikai szolgáltatás minden *seed* kérésre ugyanaz a válasz érkezik, ebben az esetben elég egyetlen *seed* érték megfelelő válasz párját megkeresni, akár *brute-force* módszerek segítségével.

A személyi számítógépek alkalmasak rá, hogy kriptográfiai szempontból biztonságosnak mondható pseudo véletlenszám generátorokat (CSPRNG) futtassanak, melyek a számítógép érzékelői illetve akár a felhasználói aktivitás alapján rövid idő alatt képesek szolgáltatni elegendő mennyiségű rendezetlenséget (*entrópiát*), az algoritmus működtetéséhez. [11] A rendezetlenség szükséges a véletlenszám generátorok inicializálásához, ellenkező esetben kimenetük erősen kiszámíthatóvá válhat. A begyázott rendszerek működése a számítógépekkel szemben lényegesebb kiszámíthatóbb, nem minden esetben rendelkeznek komplex operációs rendszerekkel, sem az azokkal járó feladatütemezési és memóriakezelési folyamatokkal vagy akár folyamatos felhasználói

aktivitással. Az entrópia gyűjtéséhez különböző analóg forrásokat, szabadon hagyott „lebegő” lábakat használnak melyek azonban nem mindig elérhetők. A rendszerek számítógépekkel szembeni egyszerűsége okán a megfelelő mennyiség összegyűjtése hosszabb időtartamot vehet igénybe mint amennyit a követelmények meghatároznak, így különös odafigyelést igényel a forrásának meghatározása, különben súlyosan meghibásodik a legtöbb információbiztonságért felelős funkció. A közelmúltban kiberbiztonsággal foglalkozó kutatók több millió IoT (*Internet of Things*) eszköz esetén azonosítottak véletlenszámok generálásával kapcsolatos problémákat. [13]

2.3 Kriptográfiai műveletek végrehajtási ideje

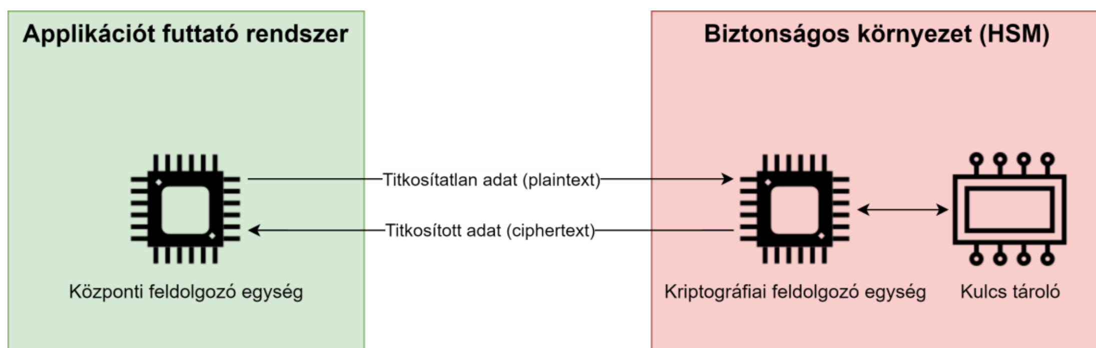
A növekedő gyártói igények és a különböző minősítésekből, szabványokból fakadó egyre számításigényesebb kriptográfiai műveletek akadályozhatják a kormányrendszer fő funkcióinak működését, melyek szintén folyamatosan fejlődnek és bővülnek (például sávtartás, parkolási segéd, oldalszél kompenzáció), szoftveres eljárásaik egyre nagyobb erőforrásigényekkel rendelkeznek. A kriptográfiai algoritmusok végrehajtási ideje egy kritikus tényező, nagyobb mértékben akadályozhatja a jármű minél előbbi üzembe helyezését . Emellett míg a biztonságos rendszerindítás használatonként egyszer, addig a SecOC és egyes periodikusan futó integritás ellenőrzések végrehajtása már jelentős többletköltséget képezhetnek a feldolgozásban. Az mikrokontrollerek egy részénél rendelkezhetünk gyártói információkkal egyes műveletek végrehajtási idejére vonatkozóan, azonban az eltérő fordító verziók optimalizációja és a gyártói szoftverektől való eltérés megváltoztathatja. További problémát jelent a mikrokontrollerek összehasonlítása, mely nem határozható meg egyszerűen azok konfigurációjának teljes ismeretében sem. Sok termék rendelkezik hardveres biztonsági modullal, amely további lehetőségeket biztosít a kriptográfiai műveletek végrehajtására és kritikus adatok biztonságos tárolására.

2.3.1 Hardveres biztonsági modulok

A legtöbb modern autóipari felhasználási területre pozícionált mikrovezérlő tartalmaz egy többnyire független erőforrásokkal rendelkező biztonsági modult, mely a fentebb tárgyalt információbiztonsági célok elérését támogatja. Funkciói között szerepel a titkos kulcsok biztonságos tárolása, *„valódi”* véletlenszám generátor funkció (TRNG) és a hardveresen gyorsított kriptográfiai műveletek biztosítása egy elkülönített

környezetben. A hardveres biztonsági modulok (HSM) az autóipari vezérlők mellett széles körben megtalálhatók más beágyazott rendszerekben például IoT (Internet of Things) megoldásokban, okostelefonokban, személyi számítógépekben, szerverekben de akár „*Plug and Play*” egyszerűségével csatlakoztatott külső eszközök formájában is.

A keretrendszer céljai között szerepel, hogy a mikrokontrollerek hardveres biztonsági modulja által hardveresen gyorsított kriptográfiai algoritmusait támogassa, majd ugyanazon mérési metodológia segítségével lemérje azok végrehajtási idejét. Eredményként összehasonlíthatók lesznek a mikrokontrollerek egymással, emellett a segédprocesszor által produkált idő összemérhetővé váljon a kriptográfiai könyvtárak által biztosított, hagyományos processzoron elért eredménnyel.



2.3-1. ábra Hardveres biztonsági modul (saját szerkesztés)

3 Implementáció

Az alábbi fejezetben kerül bemutatásra a szakdolgozat témájaként meghatározott önálló munka: a kriptográfiai műveleteket értékelő keretrendszer, annak működése, felépítése, mérési módszertana, a dolgozat írásának idejében támogatott mikrokontrollerek egyedi lehetőségei valamint a bővíthetőséget elősegítő kódolási konvenciók és használt programozási technikák.

3.1 Mérési metodológia

A vizsgálni kívánt kriptográfiai műveletek végrehajtási idejének méréséhez két időpont közötti eltérés minél pontosabb meghatározására van szükség. Az eltelt idő mérésének egyik módja a mikrovezérlő egy általános felhasználási célú lábának (*GPIO – General Purpose Input/Output*) célzott billegtetése a kriptográfiai algoritmusok végrehajtása során, majd a fel és lefutó élek közötti stabil állapot időtartamának meghatározása, például oszcilloszkóp vagy logikai analizátor segítségével. Ebben az esetben, a mérőműszerek pontosságától és kalibrációjától függenek a mérési eredmények, nagyobb hiányosságot jelent azonban, hogy mellette semmilyen egyéb kimenet vagy információt nem szereztünk a végrehajtás körülményeiről és a mérési elrendezés is felesleges túlbonyolított.

A választott megoldás a minden modern mikrovezérlőben megtalálható belső számlálók egyikének használata. Az ideális belső időzítő/számláló modulok az utasítás végrehajtástól független módon működnek, vagy egy abból előállított fázis-zárt hurok, azaz egy leosztott frekvenciával vagy más forrásból kapják az vezérlő órajel ciklust. Komplex beágyazott rendszerek mindegyike rendelkezik egy *System Timer*, *System Tick* vagy hasonló elnevezésű számláló modullal. Ezek a vezérlő indítását követően egyszerű n-bites felbontású szinkron számlálóként működnek, alacsony szintű konfigurációtól függően a maximális érték elérése esetén túlcsordulnak, ezéáltal visszatérnek a kiinduló állapotba. (*free-running counter*). Ideálisak szinte bármilyen alkalmazási terület számára, használhatóak például beágyazott operációs rendszerek időalapjának szolgáltatására, mely az eseménykezelő és feladatkezelő alrendszerek szükséglete. Megfelelő konfiguráció esetén a szoftveres és hardveres megszakítások nem szüneteltetik működését kivéve a explicit módon megszakítás kérést küldünk ezzel a szándékkal. A

számlálók működési frekvenciáját meghatározó oszcillációs forrás, előskálázó és osztó beállítások valamint a vezérlő kommunikációs lehetőségei mikrovezérlő függők, a termékek keretrendszerbe integrálása során a gyártói dokumentáció alapján jártam el.

A kriptográfiai műveletek vizsgálatához különböző tesztek kerültek létrehozásra, C programnyelven, az elemi kriptográfiai primitív procedúráktól, a komplex több alapműveletet magukba foglaló eljárásokig (például digitális aláírások kezelése). Közös tulajdonságuk, hogy a számlálók állapota lementésre kerül a tesztek végrehajtása előtt és után is. A tesztek feladata továbbá a műveletek eredményének ellenőrzése, ahol erre van lehetőség, mivel a hibás működés során mért időtartam nem fogadható el érvényesnek, ha például egy memóriacímzési probléma miatt a teszt bemenetének csak egy része került feldolgozásra.

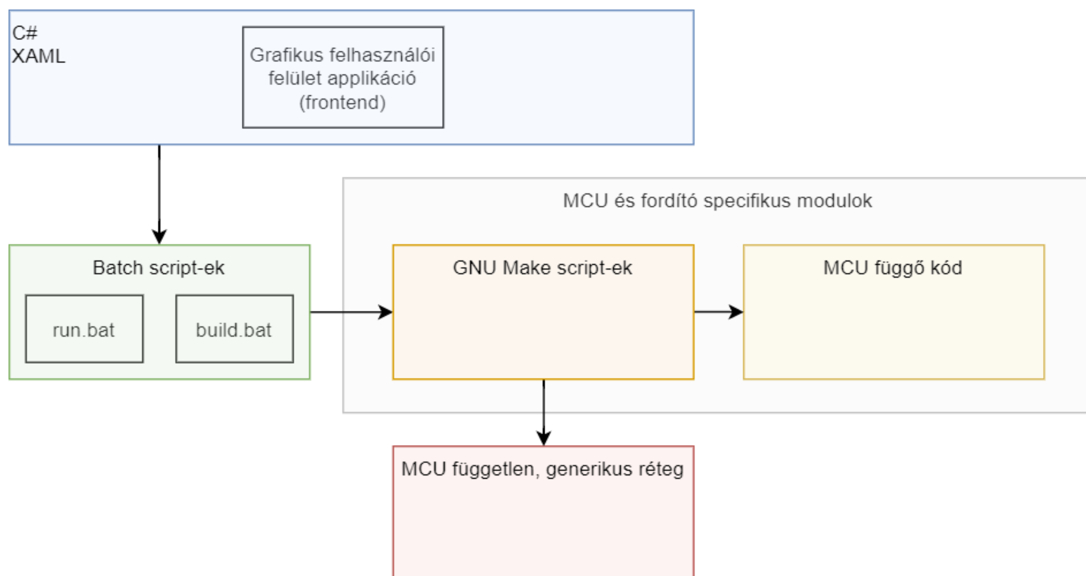
A tesztek felépítése lehetővé teszi a tartalmazott művelet implementációjának fordítási időben történő cseréjét. Forráskódjuk módosítása nélkül, fordító direktívák segítségével a tesztet lefedő futtatható állomány elkészítése során megadható azon forrás mely tartalmazza a kriptográfiai műveletek implementációját (*kriptográfiai meghajtók*). Ennek előnye, hogy azonos kódbázis, típusok, adatstruktúrák, bemenetek mellett közvetlen összehasonlíthatóvá válnak az egyedi implementációk, legyen szó akár az általános szoftveres úton számításokat végző kriptográfiai függvénykönyvtárakról, vagy a vezérlőkben található hardveres biztonsági modulok egyedi programozásáról.

Az eredmények, hibák visszajelzésért, a vezérlők UART kommunikációra képes moduljai, egy USB-soros átalakító interfész segítségével képesek kommunikálni a teszt fordítását is végző számítógéppel, egy felhasználóbarát grafikus felületen.

Az önálló munka során elkészített szoftveregyüttes a végfelhasználói kezelőprogram, az egyes mikrovezérlő típusokra szabott fordítási szabályokat megadó és hibakereső (*debugger*) interfészeikkel való kommunikációt támogató script-ek és programok összeségéből áll. Keretrendszer jellegét azon lehetőség adja, hogy egymástól akár jelentősen eltérő mikrovezérlők számára biztosít vázat amelyben a megfelelő absztrakt függvények az adott platformra történő implementálásával lehetővé teszi lehetővé mérések elvégzését és kijelzését egy egységes felületen.

3.2 Működési elv

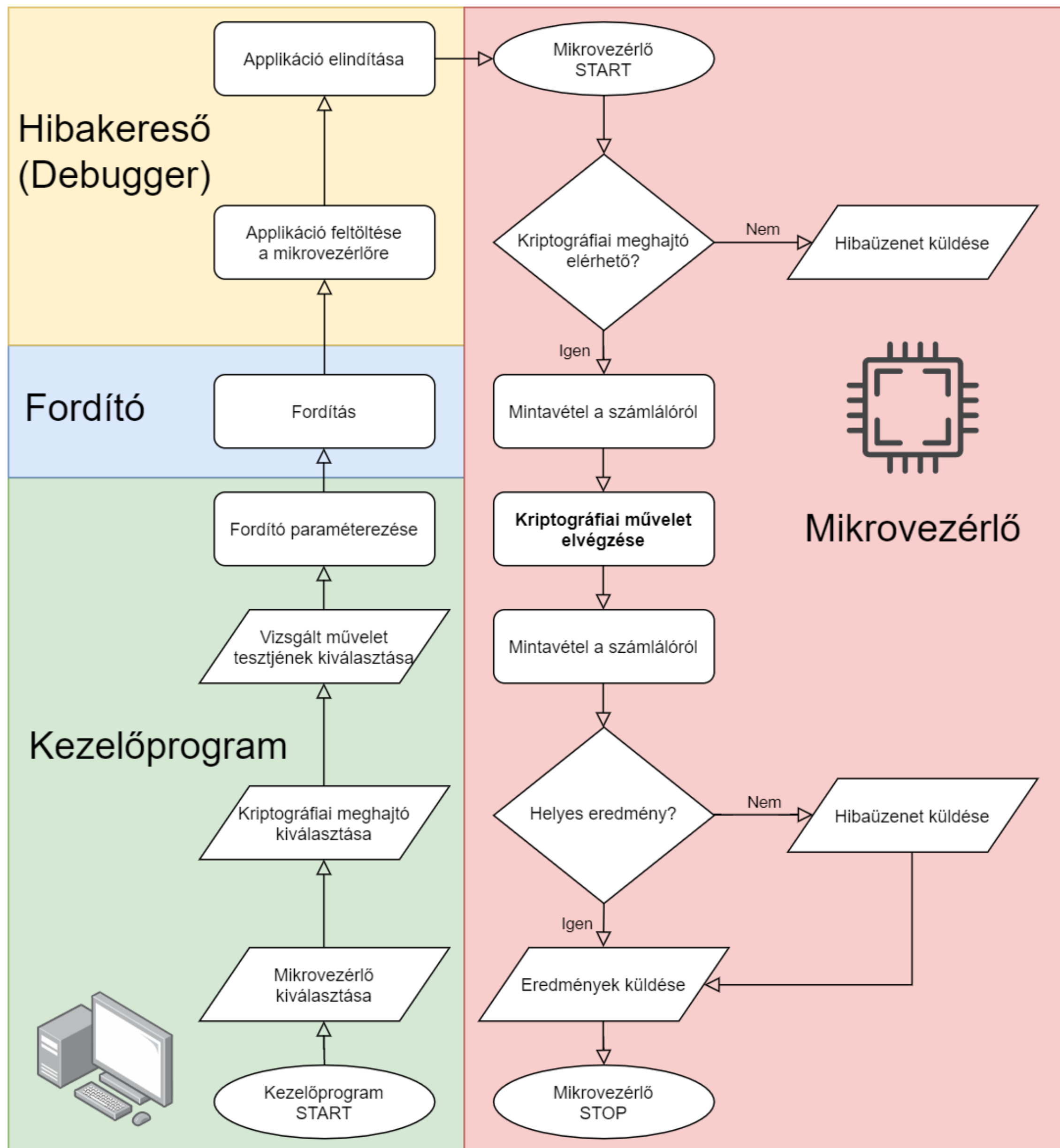
A keretrendszerbe integrált kód fordítható, egy felhasználói interakcióért felelős grafikus kezelőprogramból. A szoftver C# programnyelven készült, a grafikus felületet XAML (*Extensible Application Markup Language*) leíró nyelv realizálja, a projekt a Visual Studio 2017 integrált fejlesztői környezetében fejlesztett, .NET 4.6.1 keretrendszer felhasználásával. A használt grafikus keretrendszer a WPF (*Windows Presentation Foundation*). A grafikus felületen elérhető beviteli mezőkön keresztül paraméterezhető a keretrendszer back-end részének fordítása, a vizsgált algoritmus és a célplatform kiválasztása, mely a kiválasztott mikrovezérlővel kompatibilis fordítási szabályokat leíró script változóinak szerkesztésével történik. A fordító script-ek egyetlen közös paramétereket tartalmazó állomány kivételével vezérlő specifikusak, mivel minden mikrokontroller családhoz tartozhat egyedi fordítóprogram és linker.



3.2-1. Ábra Keretrendszer szoftverkomponensei és meghívásuk iránya (saját szerkesztés)

Amennyiben a fordítás sikeres, a kezelőprogram a PC-hez csatlakoztatott programozó/hibakereső eszköz (*debugger)* a megfelelő porton keresztül feltölti a *linker* által létrehozott futtatható kódot tartalmazó bináris állományt, a programozó eszközre specifikus script-ek segítségével. A szintén a kezelőprogram által futtatott debugger/programozó eszközt vezérlő program elindítja a feltöltött applikációt a célhardveren, majd egy - a kezelőszoftverben kiválasztott soros átviteli aljzaton keresztül - visszajelzésre vár UART kommunikációs protokollt feltételezve. Az elindított mikrovezérlő inicializálja a vizsgálathoz és a kommunikációhoz szükséges moduljait,

majd lefuttatja a tesztet melyet a kezelőszoftverben választottunk. A számítógép oldali kezelőszoftver a választott COM port-ról érkező adatokra várakozik, a felhasználói felület programszálától, aszinkron módon. A mikrokontrollerről érkező információ megjelenítésre kerül a kezelőszoftverben, beleértve az eredményt illetve az esetleges hibakódokat, egyúttal a fordító kimenetét is jelzi. A teljes folyamat látható az alábbi folyamatábrán. (3.2-2. Ábra)



3.2-2. Ábra – A keretrendszer működésének folyamatábrája (saját szerkesztés)

3.3 Felhasznált eszközök

Az alábbi alfejezetben kerül felsorolásra a Benchmarking szoftver fejlesztéséhez használt szoftveres és hardveres eszközök:

3.3.1 Hardver

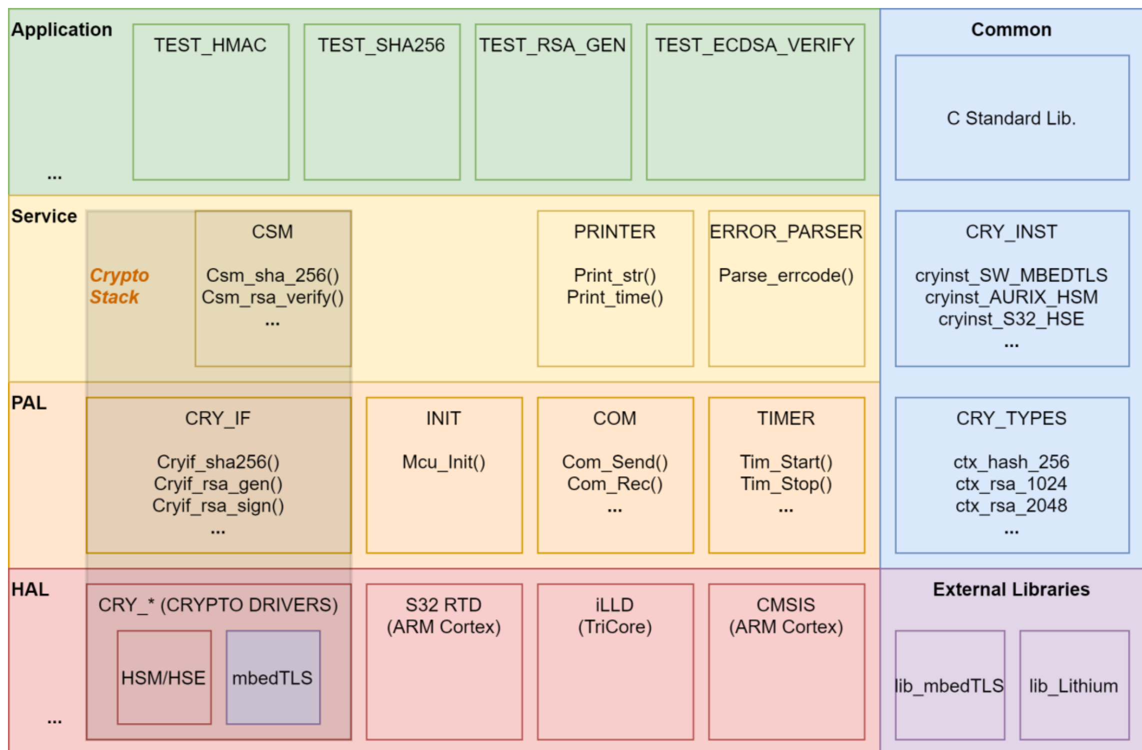
- **Lauterbach PowerDebug Pro**: Többféle processzor architektúrát támogat, jelen esetben a TC3XX Tricore és ARM feldolgozóegységinek egyidejű elindítását teszi lehetővé.
- **NXP S32K3XX** mikrokontroller családhoz tartozó vezérlő.
- **Infineon TC3XX** mikrokontroller családhoz tartozó vezérlő.
- Egyedi gyártású kiértékelő kártya: egy thyssenkrupp által tervezett egyedi kiértékelő kártya TC3XX mikrokontrollereket támogat. Belső fejlesztésekhez készítették, nincsen kereskedelmi forgalomban.
- **NXP S32 K3X4EVB-Q257**: Az NXP S32 K3XX termékcsaládjával kompatibilis kiértékelő kártya, gyári állapotában felszerelt mikrovezérlővel.
- **FTDI TTL-232R-5V**: USB-n csatlakoztatható soros kapcsolatot biztosító interfész, 5V-os jelszintet használ és feltételez.

3.3.2 Szoftver

- **TRACE32**: Lauterbach debugger kezelő szoftvere.
- **S32 Design Studio**: Eclipse alapú, integrált fejlesztői környezet (IDE), az NXP S32 mikrokontroller családjához való fejlesztésekhez, chip konfigurációhoz.
- **Visual Studio Code**: Szövegszerkesztő program, bővítmények segítségével erősen testre szabható.
- **HexEdit**: hexadecimális szerkesztő, futtatható állományok, memória dump-ok elemzéséhez.
- **mbedTLS**: Kriptográfiai függvénykönyvtár.

3.4 Keretrendszer szoftverarchitektúrája

A keretrendszer forráskódja négy egymásra épülő rétegből áll. A többrétegű szoftverarchitektúrát használó tervezésnek megfelelően az alsóbb rétegek implementációja cserélhető a felette álló rétegek módosítása nélkül, minden réteg csak az alatta lévő absztrakciós réteg szolgáltatásait használja. Ezen megoldással egyszerűen hozzáadhatók új mikrokontrollerek, algoritmus tesztek, az alapfunkciók változtatása, újra írása nem szükséges, a mikrokontroller specifikus rétegek felcserélhetősége okán.



3.4-1. ábra Keretrendszer forráskódjának többrétegű szoftverarchitektúrája (saját szerkesztés)

A kriptográfiai modulok szerepe és nevezéktana az autóiparban használatos AUTOSAR szoftverarchitektúra alapján kerültek megtervezésre. Az alábbi alfejezetek tárgyalják az egyes rétegek fontosabb moduljait és keretrendszerben betöltött szerepüket.

3.4.1 Hardver absztrakciós réteg

A legalsóbb szinten helyezkedik el a hardver absztrakciós réteg (HAL). Ezen réteg tartalmazza a legelemibb hardverközeli alapműveleteket a vezérlő alapkonfigurációját inicializálását, eszközmeghajtók, perifériák, I/O portok és minden alacsony szintű funkció implementációját. A vezérlők gyártója jellemzően keverten C nyelven, egyes modulok esetén Assembly kód formájában teszi elérhetővé. További szerepe, hogy a

használatához mellékelt példák tartalmaznak egy kiinduló konfigurációt, az induláskor beállítandó regiszterek alapértékét, megszakítási és kivételkezelő táblákat definícióját (IVT, TVT), az vezérlő indításához szükséges alapműveletek implementációját. Ezen szoftverentitás az indítókód (*startup code*), melynek általában a nem-felejtő memória egy meghatározott területére kell kerülnie, annak érdekében, hogy processzor beépített logikája (*mikrokód*) el tudja índítani. Az indítókód feladata, hogy inicializálja a hardvert és felkészítse az applikáció futtatására.

Feladatai között szerepel:

- A HAL réteg fordítás előtt meghatározott konfigurációs fájljainak megfelelően beállítja a regisztereket
- Stack és heap allokálása a memóriában, stack pointer alapértékének beállítása
- Megszakítások letiltása, majd engedélyezése a folyamat végén
- Felügyeletidőzítő (*watchdog*) beállítása
- Újraindítás során végrehajtandó első utasítás memóricímének meghatározása (*Reset vector*)
- A folyamat utolsó lépéseként a programszámlálót (IP/PC) az applikáció belépési pontjára (main() függvény) állítja.

Az Infineon TC3XX 32 bites mikrokontroller család termékei két különböző architektúrájú feldolgozó egységgel is rendelkezhetnek (TriCore és ARM Cortex M), emiatt a HAL rétegét két külön fordított kódbázis alkotja, a TriCore processzorok számára a gyártói iLLD (Infineon Low Level Drivers), az ARM feldolgozóegységek számára a nyílt forrású CMSIS (Common Microcontroller Software Interface Standard) csomagját használtam. Az NXP S32 K3 mikrokontrollercsaládja számára, a gyártótól elérhető S32 RTD (Real-Time Drivers) kereskedelmi változatát vettem igénybe.

Az platform absztrakciós réteg és minden hierarchikusan felette elhelyezkedő réteg általam került implementálásra, míg a hardver absztrakciós réteg és indítókód minden esetben a gyártótól származó példákból származik.

3.4.2 Platform absztrakciós réteg

A vezérlő gyártója által kiadott szoftvercsomag (HAL réteg, példaprogramok) alapján a kerültek implementálásra a keretrendszerhez szükséges magasabb szintű funkciók, mint például a számlálók kezelése, az UART modulon keresztüli kommunikáció, vagy a megszakítások letiltása.

A platform absztrakciós réteg függvényeinek hívási szignatúrái platform függetlenek, a fentebb említett magasabb szintű műveleteket generalizált formában teszik elérhetővé a felsőbb rétegek számára. Az alábbiakban rövid bemutatásra kerülnek az egyes modulok.

INIT modul

Azon alapértékek beállítására szolgáló eljárásokat tartalmazza, melynek futtatására a Startup Code futtatását követően kerül sor, az applikáció első meghívott komponense. A hardver absztrakciós réteg szolgáltatásainak segítségével elvégzi a keretrendszerben definiált modulokat, például a megfelelő lábak kiválasztását a COM modul működéséhez. Tartalmazza továbbá az *Mcu.h* fejléc fájlt, melyben minden INIT modulban található segédfüggvény definíciója megtalálható valamint minden szükséges hardver absztrakciós rétegbeli fejléc beillesztéséért is felel. A platform top-level fejlécének is tekinthető, emellett olyan konstansokat és makrókat tartalmaz, melyek azonosítják az adott vezérlőn mely kriptográfiai meghajtók támogatottak, ennek megfelelően a releváns modulok fordítása befolyásolható, például a szoftveres kriptográfiai meghajtóként használt mbedTLS használatához szükséges statikus átmeneti tároló nem kerül definiálásra ha az *Mcu.h* fájlban nem jelezzük mely az mbedTLS támogatását. (B-3)

COM modul

A mikrovezérlő UART protokollt használó kommunikációjáért felelős modul. A felsőbb rétegek ezen modul függvényeivel küldenek adatot a kezelőszoftvert futtató PC számára. Felelős a vezérlő UART interfészén a megfelelő átviteli sebesség beállításáért. A baudrate (továbbított bitek száma / másodperc) valamely kompatibilis fáziszárt hurok (PLL) alapján előállított érték, a megfelelő előskálázó, osztók és túl-mintavételezés mértékének beállításával. Ezen paraméterek a támogatott mikrokontrollerek esetén

manuálisan a megfelelő regiszterek beállításával is megadhatók, de a hardver absztrakciós réteg is biztosít függvényeket a beállításhoz.

Egy-egy küldő és fogadó függvény felelős az átvitt információkért, a keretrendszer mindkét eljárás szinkron módú működését feltételezi (blokkoló utasítások), amennyiben a küldéshez vagy fogadáshoz tartozó küldő/fogadó (TX/RX) átmeneti tárolók nem telnek meg a beállított időtúllépési határ elérését követően, a program futtatása folytatódik.

TIMER modul

A TIMER modul felelős a méréshez használt számláló beállításáért, aktiválásért, és az eltelt idő kiszámításáért. A hardver absztrakciós réteg általánosságban biztosít a számlálók kezelésére függvényeket, azonban figyelembe véve a TIMER modul egész feladatra kiterjedő kritikusságát, az első támogatott vezérlők esetén egy saját megoldást implementáltam szem előtt tartva a minél kisebb futtatási költséget és felesleges függvényhívásokkal járó stack műveletek elkerülését. A támogatott vezérlők tartalmaznak valamely órajel forrásból leosztott frekvenciával működtethető akár 64 bites felbontású számlálókat. A nagyfelbontású számlálók értékét egyetlen betöltő utasítással egy 32 bites adatbuszokkal rendelkező processzor nem képes egy utasítás keretein belül olvasni, anélkül, hogy az érték másik fele a betöltés ideje alatt megváltozzon. Ezen probléma megoldására *Capture* regisztereket lehet alkalmazni (amennyiben elérhető), melyeken keresztül a teljes felbontás egy kisebb szeletén olvasható a számláló állapota. Az eltelt idő számítása mindkét vezérlő esetén megegyezett, az kezdetben letárolt idő, és az algoritmus végrehajtása után letárolt idő különbsége osztva a számláló frekvenciájával.

$$eltelt\ idő(s) = \frac{számláló\ \ 2.lekérdezés - \ számláló\ 1.lekérdezés}{számláló\ frekvencia\ (Hz)}$$

CRY_IF modul

A kriptográfiai interfész (CRY_IF) modulban található azon programlogika mely kiválasztja a felhasználó által a kezelőprogramban megadott algoritmushoz megjelölt kriptográfiai meghajtó implementációját. Mivel az egyes hardveresen gyorsított megoldások egyediek az adott vezérlőre, illetve nem feltételezhető, hogy minden kriptográfiai függvénykönyvtár vagy hardverspecifikus megoldás fordítható minden eszközre, ezért a vezérlő INIT moduljában definiált makrók alapján az adott vezérlőre

nem értelmezhető/fordítható részek, kihagyásra kerülnek a C fordítók előfeldolgozó lépése által (B-1). A szolgáltatási réteg CSM moduljából kerülnek a meghívásra függvényei, az átadott paraméterek alapján választják ki a használt kriptográfiai meghajtó felhasználásával implementált algoritmust. A modul tervezésekor az objektum orientált szoftverfejlesztésben gyakran megjelenő *Strategy* nevű tervezési minta szolgált példaként.

3.4.3 Szolgáltatási réteg

CSM modul

A kriptográfiai szolgáltatások modulja (*Crypto Service Module* - *CSM*), a tesztelni kívánt algoritmusok meghívását biztosítja a tesztek számára. Az algoritmusok paraméterezése egy a teszttel kompatibilis struktúra létrehozásával történik, melyre mutató memóriacím az alsóbb rétegek számára átadásra kerül egészen az algoritmus tényleges implementációjáig. A kimenet ugyanezen struktúrába kerül, ezáltal mivel a változó a teszt kontextusában született, az applikációs réteg által ellenőrizhető és megjelenítésre feldolgozható a tartalma.

3.4.4 Applikáció réteg

A legfelső réteg tartalmazza a keretrendszerből fordított applikáció belépési pontját (*main()* függvényét) és az algoritmusok tesztjeit. A szolgáltatási réteg funkcióit használja, így a tesztek az alsó két mikrokontroller függő rétegtől eltérően teljesen hordozható kódot kell tartalmaznia.

Egy felhasználó által a kezelőprogramon keresztül beállított konfigurációs fejlécfájl alapján kerül beillesztésre a választott teszt függvény fordítási időben, ennek módja a 3.6 fejezetben olvasható. A tesztek forráskódjának struktúrája és kötelező elnevezési konvenciók a 3.8.1 fejezetben olvashatók.

3.4.5 Közös függőségek

Az közös függőségek között találhatók azon struktúrák, típusdefiníciók és kapcsolódó segédfüggvények melyeket valamennyi más rétegnek ismernie kell (objektum-orientált szövegkörnyezetben: entitások), azaz közösen függenek tőlük. A fordítási időben történő típusbiztonság ellenőrzéséhez elengedhetetlen, az explicit típus konverziók (*type-casting*) túlzott használata a tesztekben is megelőzhető.

CRY_TYPES

Az tesztelt algoritmusok bemenetei paraméterek formájában átadásra kerülnek több rétegen keresztüli hívási láncban, ennek megfelelően egy közös fejlécfájlban kerülnek definiálásra a kód újra felhasználási elvnek megfelelően. (*CryTypes.h*)

Példa: Egy hash függvény futtatásához a tesztet leíró kódban definiált bemenet, a Service réteg CSM modulja számára kerül átadásra, majd az Platform réteg CRY_IF moduljának függvénye kapja meg és hívja a választott algoritmust a megfelelő kriptográfiai meghajtó kiválasztása után.

Példa végrehajtási lánc:

- `Csm_hash_sha256(CRYINST_TC_HSM, *hash_ctx)`
- `Cryif_hash_sha256(CRYINST_TC_HSM, *hash_ctx)`
- `Cry_hsm_hash_sha256(*hash_ctx)`

Részlet a fájl tartalmából: (B-2)

CRY_INST

A CRY_INST modul tartalmaz egy felsorolást (*enumerációt*) a keretrendszer által támogatott összes platformfüggő és független kriptográfiai meghajtóról. A enumeráció értékeihez tartoznak makrók melyeket a platformok a saját inicializációs moduljukban definiálhatnak ezzel jelezve hogy támogatják az adott meghajtót, ellenkező esetben az meghajtó függvényei nem kerülnek be a fordított elemek közé.

A dolgozat írásakor támogatott meghajtók:

- *CRYINST_TC3XX_HSM* (Infineon TC3XX család HSM meghajtója)
- *CRYINST_MBED_TLS* (Multiplatform kriptográfiai függvénykönyvtár)
- *CRYINST_S32_K3XX_HSE* (S32 K3XX család HSE meghajtója)

3.5 Kriptográfiai meghajtók

Az alábbi alfejezetben kerülnek bemutatásra a keretrendszer által jelen dolgozat írásakor támogatott, szoftveres módon számításokat végző és célhardverrel gyorsított kriptográfiai implementációkat biztosító források.

3.5.1 mbedTLS függvénykönyvtár

Az mbedTLS egy nyílt forrású, alacsony erőforrás igényű és kis digitális lábnyommal rendelkező TLS protokollcsomagot megvalósító függvénykönyvtár. Megtalálható benne kriptografikus elemi függvények illetve X.509-es szabvány alapján leírt tanúsítványok kezelésének implementációja egyaránt. A könyvtárat C nyelven írták, nagy előnye, hogy funkcionalitása nagy mértékben skálázható, az olyan alacsony erőforrásokkal rendelkező célplatformok mint a beágyazott rendszerek számára is fordítható. A konfiguráció történhet a mellékelt *config.h* fájlban manuálisan, vagy az előbbit szerkesztő *config.py* Python script segítségével. Az egyes szoftvermodulok ki/be kapcsolhatók annak megfelelően hogy bizonyos makrók definiálásra kerülnek-e vagy sem. Az alapkonfigurációtól az alábbi makrók esetén tértem el, egyéb módosításra nem volt szükség a sikeres fordításhoz egyik támogatott mikrokontroller esetén sem:

#define MBEDTLS_PLATFORM_MEMORY: Bekapcsolásra került, ebben az esetben a mbedTLS lehetőséget ad a dinamikus memória kezelését biztosító *malloc() calloc()* és *free()* függvények újra definiálására. Erre azért volt szükség mert a Benchmarking szoftver külön kérésére kizárólag statikus memóriakezelést használ, ennek megfelelően a linker script fájlokban a *heap* terület allokációjára nem kerül sor, kizárólag *stack* terület, emellett a mikrovezérlő inicializálását végző startup szoftverből sem kerülnek behívásra a dinamikus memóriakezelést biztosító rutinok, így fordítási hibát eredményez a makró nem definiálása.

#define MBEDTLS_MEMORY_BUFFER_ALLOC_C: Az előző beállításhoz kapcsolódóan szintén bekapcsolásra került makró, ebben az esetben minden olyan mbedTLS függvény melyek használnának dinamikus memóriafoglalást, ehelyett egy statikusan a *stack*-en létrehozott *buffer* változóra tekintenek dinamikus memóriaterületként. Ezt a tárolót a fejlesztőnek külön kell definiálnia, megemlítendő, hogy a kulcs generáláshoz és aláíráshoz szükséges műveletek miatt több kilobyte méretűnek választottam mindkét integrált platform számára. [A)B-3]

A mbedTLS három különálló függvénykönyvtárba fordítható (*libmbedtls.a*, *libmbedx509.a*, *libmbedcrypto.a*). A forráskódhoz mellékelt makefile dinamikus könyvtárakba fordít, ezt a fájlt átalakítottam statikus könyvtárak fordítására és csomagálására, melyet az adott vezérlő fordítójának eszközkészletéhez tartozó *archiver* program végez el. [A)B-4] Mindkét platform saját fordítási útmutatója szerint (makefile)

statikus csomagokként kerülnek hozzáadásra a futtatható állományhoz a linkelési folyamat során.

3.5.2 TC3XX: Hardware Security Module

A TC3XX mikrovezérlő család rendelkezik egy dedikált belső komponenssel a kriptográfiai műveletek gyorsításához, Hardware Security Module néven. A HSM rendelkezik saját ARM Cortex M3 processzorral, memóriával de egyes erőforrásaiban osztozik az applikációt futtató TriCore processzorokkal. A modullal való kommunikáció egy tűzfalon keresztül megvalósított, ezen keresztül van lehetőség üzeneteket küldeni és válaszokat kapni. A HSM szintén egy flash memóriáról betöltött firmware segítségével működik, a kriptográfiai algoritmusok hardveres implementációk formájában érhetők el, a rendszer ismeretlen belső működése szándékosan nem kerül közlésre a gyártótól, így nem láthatjuk a műveletek pontos megvalósítását, például biztonsági okokból nem látható hogy a véletlen szám generátor működéséhez szükséges rendezetlenség (entrópia), milyen forrásból érkezik.

A dokumentáció alapján elérhető hardveresen gyorsított algoritmusok között szerepel MD5, részleges SHA-2 (csak SHA-256), AES-128, illetve maximum 256 bit hosszúságú ECDSA kulcspárokat használó aláírások készítése, ellenőrzése, valamint egy valódi véletlen szám generátor (TRNG).

A TC3XX HSM támogatásához készített CRY modulhoz tartozó implementációk többsége a HSM kommunikációs buszra írt algoritmust kiválasztó parancsok és paraméterek átadásából áll, majd megszakítás kérés küldéséből.

3.5.3 K3XX: Hardware Security Engine

A szintén vizsgált NXP gyártmányú K3XX termékcsalád is rendelkezik hardveres biztonsági modullal, mely a Hardware Security Engine (HSE) névre hallga. A TC3XX megoldásával szemben a támogatott kriptográfiai műveletek száma lényegesen magasabb. A teljesség igénye nélkül: AES-128/192/256 valamennyi elérhető láncolási móddal, SHA-256/512, 1024-4096-ig terjedő hosszúságú RSA kulcsok kezelése, tárolása. A HSE hasonlóan a TC3XX megoldásához hasonlóan szintén egy tűzfalként is funkcionáló buszon kommunikál a központi feldolgozó egységgel. A megszakítás rutinok hívása mellett, további lehetőség a „*polling*” megoldás, ahogy a HSE-n futó alapprogram

egy folyamatos ciklusban figyeli a busz-ra érkező parancsokat, melyek szükségszerűen sorban is állhatnak, kiszolgálásuk FIFO jelleggel történik.

3.6 Fordítási szabályok

Az C nyelven írt programok előnye, hogy a nyelvi eszköztár és a szabványos C könyvtár (*C Standard Library*) szinte bármely rendszer számára elérhető, a személyi számítógépekben használt mikroprocesszoroktól kezdve a legszerényebb képességű mikrokontrollerekig. Erősen hordozható kód írható C nyelven, amennyiben a támogatott platformok fordítóprogramjai a nyelv azonos változatát támogatják (például *C89, C99, C11*) és nem szerepelnek az applikációban platform specifikus függőségek.

Ezzel szemben a különböző platformok fordítócsomagjainak használata eltérnek egymástól, a fordító utasítások sokkal kevésbé általánosíthatók. A fordítási útmutatót tartalmazó forrásállományok (*makefile*) az egyedi fordítóprogram szintaxis miatt minden platform számára külön létrehozott fájlok, melyeket a választott platform könyvtárából hív meg a kezelőprogram. A makefile-ok meghatározott sorrendben végrehajtandó feladatokból (*makerule*) állnak, melyek akár egymással párhuzamosan is képesek futni. Az egyes feladatrészek útmutatóként szolgálnak a fordítóprogramok számára, a programokat egyes részeinek milyen egymással szembeni függőségeik is megadhatók, így a feladatok párhuzamos végrehajtása során szükség szerint várakozhatnak egymásra. a További előnye, hogy az első fordítás során létrejött érvényes tárgykódot nem fordítja újra, csak a megváltozott forrásállományokat. Ennek megfelelően amikor a felhasználó egy újabb tesztet szeretne futtatni nem szükséges fordítani a teljes keretrendszer kódját, csupán az applikáció fő programrészét (main.c), majd a meglévő tárgykód felhasználásával új futtatható állomány készíthető, mely jelentősen meggyorsítja a fordítási folyamatot.

A kezelőprogram egy minden platformra érvényes közös makefile segítségével képes paraméterezni az elkészítendő futtatható állományt, azaz mely vezérlőre készül, és melyik algoritmus teszt került kiválasztásra a felhasználó által. Ennek formája általános, a további vezérlő specifikus fordítási szabályokat ezen makefile hívja be. Az elérési útvonalak kötöttsége miatt, ezen struktúra alapján kell a platformfüggő kódot elhelyezni minden vezérlő esetén. [3.8.2]

```
MCU = vezerlo_neve
TEST = vizsgalni_kivant_algoritmus_neve
SRC_DIRS ?= ./src/core/main ./src/core/modules ./src/mcu/$(MCU)/bsw/
./src/mcu/$(MCU)/modules/ ./src/core/tests/$(TEST)

include ./src/mcu/$(MCU)/config/Makefile
```

A kódrészletben látható **MCU** és **TEST** változók írása a kezelőprogram feladata, mely a módosítást követően meghívja az állományon a makerule-ok feldolgozására használható a *make* programot, mely behelyettesíti és meghívja a kontroller specifikus következő makefile-t. A vezérlők és tesztek nevei a tartalmazó könyvtáraik nevével kell megegyezniük. A „belső" makefile-ok elkészítése a hivatkozott webhelyen található cikk sablonja alapján történt. [14]

A behelyettesítést követően a **SRC_DIRS** változó tartalmazza a keretrendszer gyökérkönyvtárához relatívan hivatkozott könyvtárak listáját. A két vezérlőspecifikus script ezen könyvtárakból a *find* program segítségével megkeresi az összes fejléc és implementációs fájlt, majd beállítja a fordító és linker számára szükséges paraméterváltozókat. (jellemzően **CFLAGS** és **LDFLAGS** névvel szokás ellátni őket). (B-5, B-6)

3.7 Programozó eszköz vezérlése

Az elkészült futtatható állomány többféle módon feltölthető a vezérlőre. Egyes fejlesztői kártyák rendelkeznek lapkára helyezett hibakereső eszközzel, ilyenkor általában rendelkezésre állnak kompatibilis szoftverek a feltöltéshez/hibakereséshez.

A vezérlők többsége rendelkezik a Joint Test Action Group által szabványosított tesztlábakkal (*JTAG port*), melyek alacsony szintű hozzáférést biztosítanak a lapkán elhelyezett processzorokhoz a csatlakoztatott számítógép számára egy soros kommunikációs interfésszel egyetemben. Többek között JTAG lábakat használ a feltöltéshez, a dolgozat írásakor használt *Lauterbach* hibakereső eszköz is. Legnagyobb előnye a modularitása: alkatrészei cserélhetők ezáltal különböző processzor architektúrákkal is kompatibilis, emellett a kezelőprogramja (*TRACE32*) teljeskörű saját script nyelvvel rendelkezik (*PARCTICE*). A program minden funkciója elérhető a parancsfájlok formájában elhelyezett script, így a feltöltési folyamat automatizálható.

A fordítási folyamat befejeztével a felhasználó elindíthatja a szoftver feltöltési folyamatot mely egy háttérben meghívott parancsfájl futtatási jelenti. A parancsfájlok nem tartalmaznak generikus részeket, a vezérlő gyártók de akár a termékcsaládokon belüli vezérlők flash memória szervezésében megjelenő eltérések, nem teszik lehetővé egységes script készítését, így minden vezérlő keretrendszerbe integrálása során újabb parancsfájlok létrehozására van szükség. (B-8)

3.8 Integrációs lehetőségek

Az alábbi alfejezetben kerül bemutatásra a keretrendszer által nyújtott váz lehetőségei, a további mikrovezérlők és kriptográfiai algoritmusokat vizsgáló tesztek integrálása a meglévő projektstruktúrába. Az erősen moduláris felépítés, a többrétegű szoftverarchitektúra alkalmazása a keretrendszer bővíthetőségét hivatott támogatni, az alább bemutatott példák egyben útmutatóként szolgálhatnak szoftverfejlesztők számára.

3.8.1 Új teszt hozzáadása

Új teszt hozzáadását elősegíti a kész tesztek között megtalálható *test_dummy* elnevezésű sablon mely áll egy implementációs és két fejlécfájlból. (test_dummy.c, test_dummy.h, test_input.h)

test_dummy.c

```
double runtime;

test_func LoadTest() {
     return Test_dummy;
}

uint32 Test_dummy() {

     int32_t test_validity = 0;
     CryCtx_Dummy dummy_ctx;

     Tim_StartTimer();
     Csm_Do_Nothing(selected_cryinst, &dummy_ctx);
     Tim_StopTimer();

     runtime = Tim_TimerInMs();
     Printer_time_results(runtime);

     return dummy_ctx.validity;
}
```

A kódrészletben látható példa, egy kriptográfiai meghajtók függőségétől mentes „üres” teszt, mely semmilyen kriptográfiai algoritmust nem hajt végre, teszt

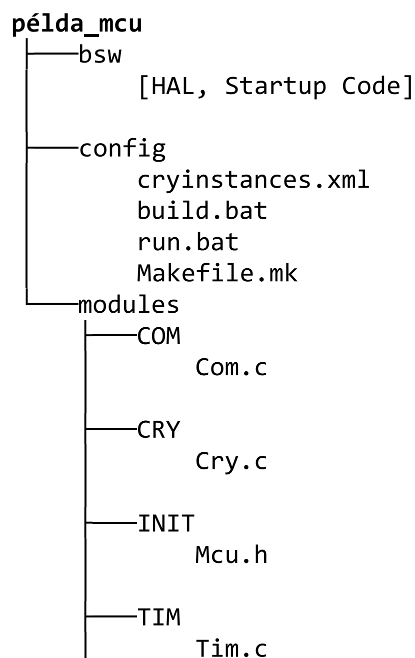
fordításokhoz használható egy újonnan integrált vezérlő számára, emellett sablonként szolgál a valódi tesztek megvalósításához.

A vizsgálathoz használt számláló állapotát eltároló függvények között kell elhelyezni a CSM modul meghívható kriptográfiai műveletét, a kiválasztott implementációs csatorna a közös kódbázisban található *cryinst.h* fájlban kerül beállításra a kezelőszoftver által. A kontextusfüggő adattípusokat definiálni kell a *CryTypes.h* fejlécfájlban, ezt minden teszt forráskódjába beillesztendő az *#include* fordító direktíva segítségével. Azonos módon szintén be kell illeszteni az *Com.h* és *Tim.h* fájlokat, annak érdekében hogy használhassa a teszt a PAL réteg nyújtotta absztrakciót. Az adott teszt eredményének elküldése a kezelőszoftveren való megjelenítéshez, a Service réteg Printer_* függvényei használhatóak, attól függően milyen adattípust szeretnénk küldeni.

A *test_input.h* fejlécfájlnak kell tartalmaznia az adott algoritmus tesztparamétereit és előre ismert kimenet esetén az eredményt, melyet a teszt szükségszerűen ellenőrizhet. (B-9)

3.8.2 Új mikrovezérlő hozzáadása

A fejlesztőnek hozzá adnia egy új könyvtárat a vezérlő elnevezésével az *mcu* könyvtárban, ezen nevek alapján listázza ki a kezelőprogram az elérhető célhardverek listáját. Az újonnan létrehozott könyvtárban az alábbi struktúra kialakítása szükséges:

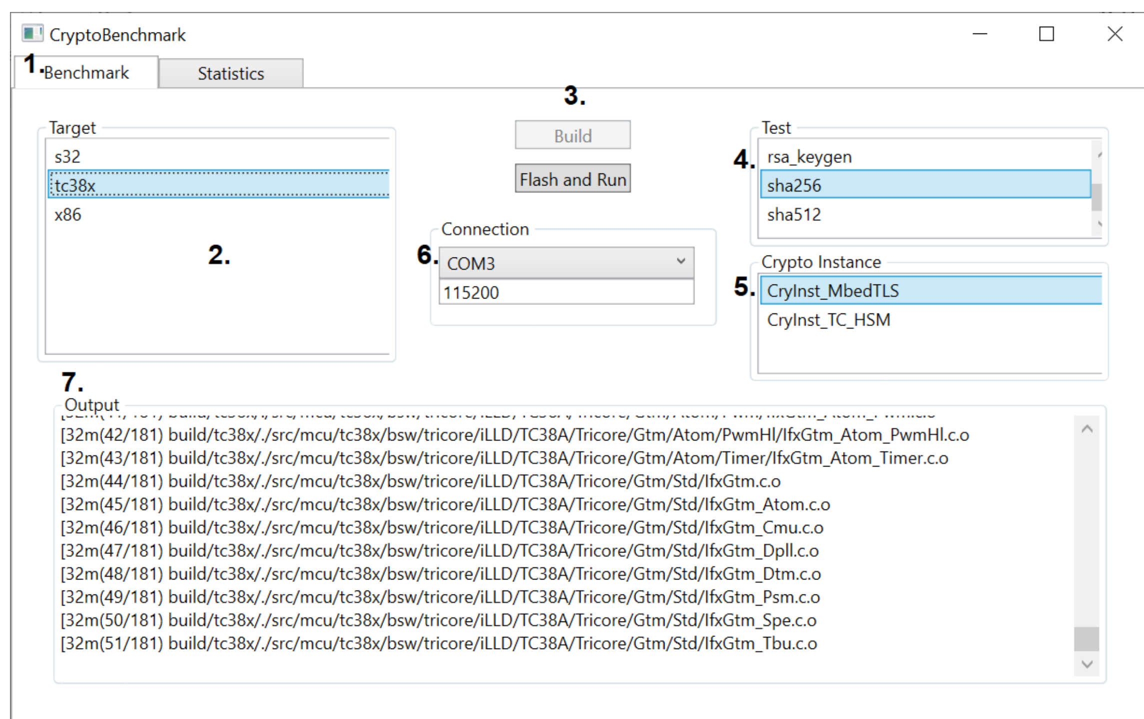


A főprogram által beillesztett *Mcu.h* fájl a mikrovezérlő fő fejlécfájlja, tartalmaznia kell minden olyan függőséget mely segítségével egy üres fő függvénnyel ellátott implementációs fájl is lefordítható lenne az adott platformon. A hardver absztrakciós réteget, indítókódot, bármely egyéb szükséges alapkonfigurációval egyetemben el kell helyezni a vezérlő *bsw* (*Basic Software*) könyvtárában, melyet általában a gyártó vagy harmadik személy biztosít, vagy mellékel példafájlok formájában. Az *Mcu.c* fájlnak tartalmaznia kell egy *Mcu_Init()* függvényt mely meghívja a platform réteg moduljait inicializálja.

A *config* alkönyvtárnak kell tartalmaznia a vezérlő specifikus fordítási útmutatót *Makefile.mk* néven illetve az esetlegesen fordítás előtt végrehajtandó parancsok listáját *build.bat* parancsállományban. A kezelőprogramban való megjelenítéshez és felhasználó általi kiválasztáshoz a vezérlő által támogatott kriptográfiai meghajtókat tartalmazó listát a *cryinstance.xml* állományban kell elhelyezni az előírt séma alapján. (B-7)

3.9 Kezelőprogram

3.9.1 Felhasználói felület áttekintése



3.9-1. Ábra - Kezelőszoftver felülete: a kimeneten fordítási folyamat látható (saját szerkesztés)

1. Nézetváltó fülek.
2. Célplatform kiválasztása: az opciókat, az *src/mcu* könyvtár tartalma adja.
3. A **Build** gombra kattintva, a felhasználó elindíthatja a fordítási folyamatot a kiválasztott opciók alapján. A **Flash and Run** elindítja a célplatformhoz készített scriptet mely elindítja a programozó eszközt/hibakeresőt, majd várakozik egy kiválasztott soros porton érkező adatokra.
4. Teszt kiválasztása: az opciókat az *src/core/tests* mappa tartalma adja.
5. Kiválasztható a használt kriptográfiai könyvtár/meghajtó. Az opciók a kiválasztott célplatform alapján változnak.
6. A soros port kiválasztása, melyek keresztül a mikrovezérlő kommunikálhat a kezelőprogrammal. Az átviteli sebesség (*baudrate*) manuálisan megadható.
7. A fordító, programozó scriptek és a vezérlő irányából érkező üzenetek kimenete látható. Tartalma színkódolt, a kék szöveggel megjelenő információk a választott COM portról érkeznek.

3.9.2 Belső működés

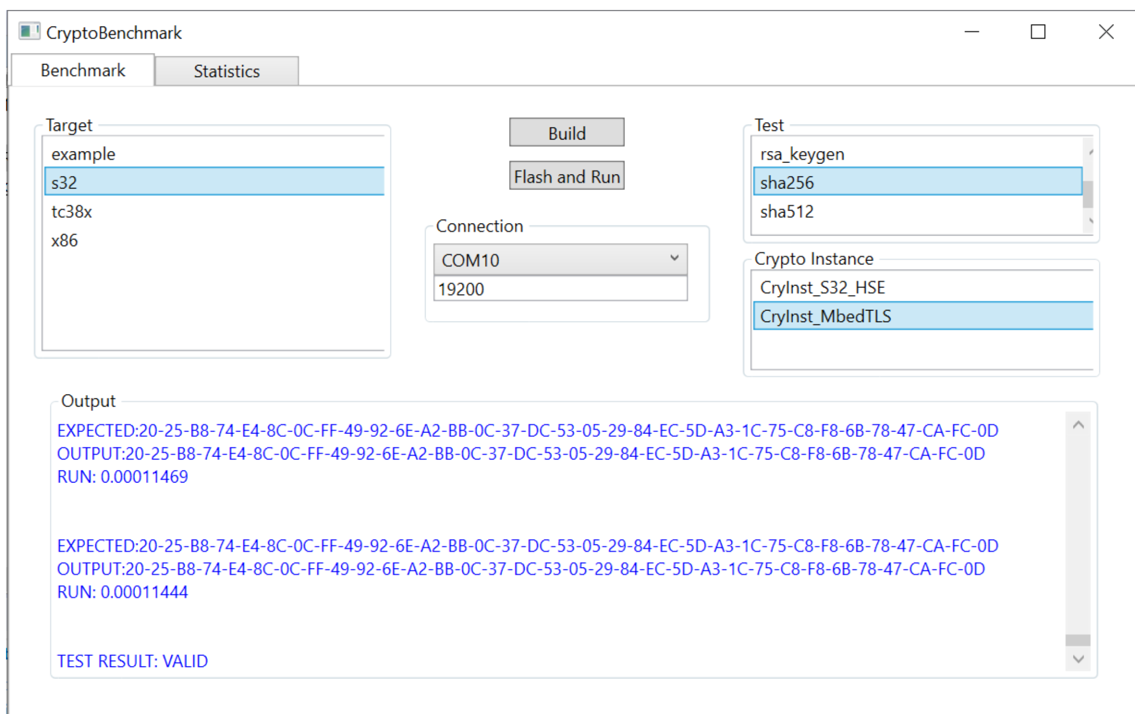
A kezelőprogram a .NET keretrendszer, a Windows Presentation Foundation (WPF) UI könyvtárának felhasználásával. A grafikus felület manuálisan XAML nyelv használatával készült mely egy XML-hez hasonló leíró nyelv kifejezetten ilyen felületek készítésére. A felület nincsen soros kapcsolatban a back-end kóddal, adatkötést nem használ, mivel relatíve kevés interakció történik a felhasználóval, így a UI komponensek által kiváltott eseményekre iratkoznak fel eseménykezelő függvények.

```
<Button x:Name="btn_Build" Content="Build" Width="80" Margin="5"
Click="Btn_Build_Click" ></Button>
<Button x:Name="btn_Run" Content="Flash and Run" Width="80" Margin="5"
Click="Btn_Run_Click" ></Button>
```

A *Btn_Build_Click* függvény a gomb megnyomásakor, elvégzi a top-level makefile-ban szükséges változtatásokat a listákból választott opciók alapján majd lecseréli a választott teszt fejlécfájlját az applikációs réteg kódjában. Sikeres írás esetén elindítja a választott *mcu* könyvtárában található *build.bat* parancsfájlt, majd kimenetét a képernyőkép (3.9-1. Ábra) alsó felén látható csak olvasható tartalmú szöveges mezőbe irányítja.

```
...
foreach (var line in TestHeaderContent)
            {
                if (line.StartsWith("#include \"test"))
                {
NewTestHeaderContent.Add($"#include \"test_{lst_test.SelectedValue}.h\"");
                }
                ...
            }
...
```

A *Btn_Run_Click* függvény meghívja az *mcu* könyvtárában található *run.bat* fájlt, majd egy új programszálon várakozik a kiválasztott COM portra, ezzel együtt letiltva a UI komponenseket.

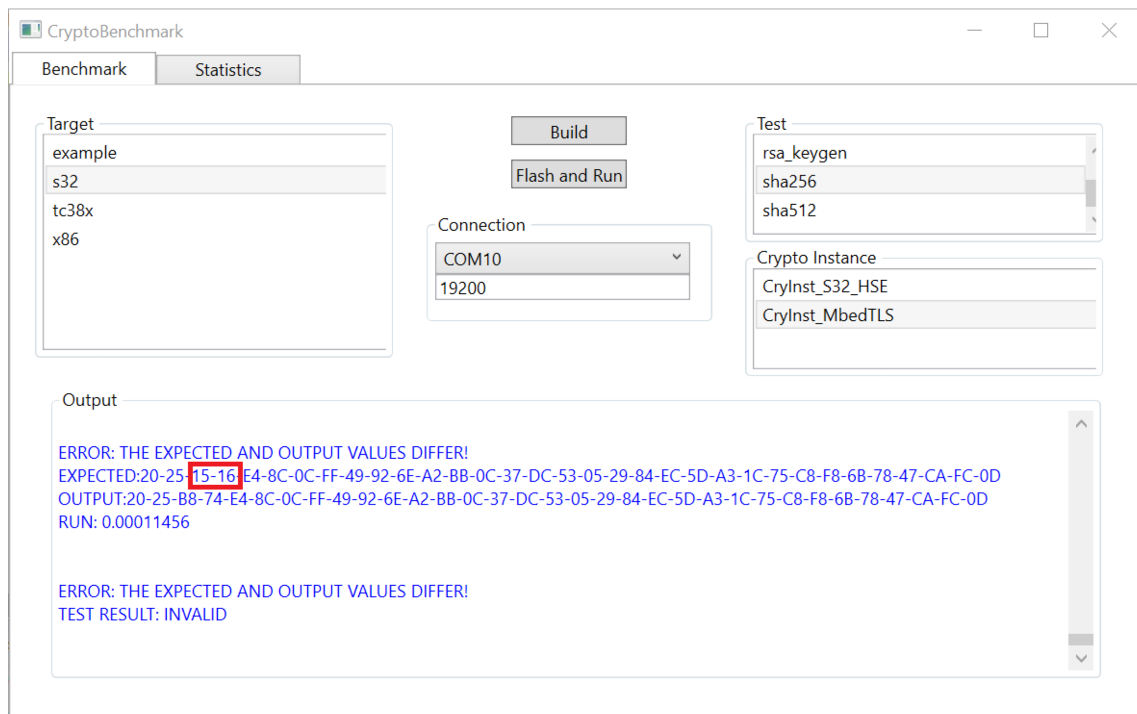


3.9-2. Ábra – Kezelőprogram felülete: sha256 teszt eredményeinek megjelenítése (saját szerkesztés)

A vizsgálatok validációja, a teszt feladatkörébe tartozik. A feltöltött applikáció utolsó lépése hogy nyugtát küldjön a teszt sikerességéről. Amennyiben megerősíti a kezelőprogramot abban, hogy nem történt hiba és a művelet elvégzése az utólagos ellenőrzést követően helyesnek bizonyult (3.9-2. Ábra: „**TEST RESULT: VALID**”), úgy kerülhet eltárolásra az eredmény, ellenkező esetben a legtöbb jelenleg implementált teszt egy hibaüzenetet is küld a negatív válasz mellett.

A következő képernyőképen látható a teszt ismételt futtatása, viszont az előre számolt érték 3. és 4. byte-jának megváltoztatásával (3.9-3. Ábra). A teszt ekkor is kiírja a

végrehajtási időt, azonban jelzi, hogy a számolt érték eltér az elvárttól. (**TEST_RESULT: INVALID**).



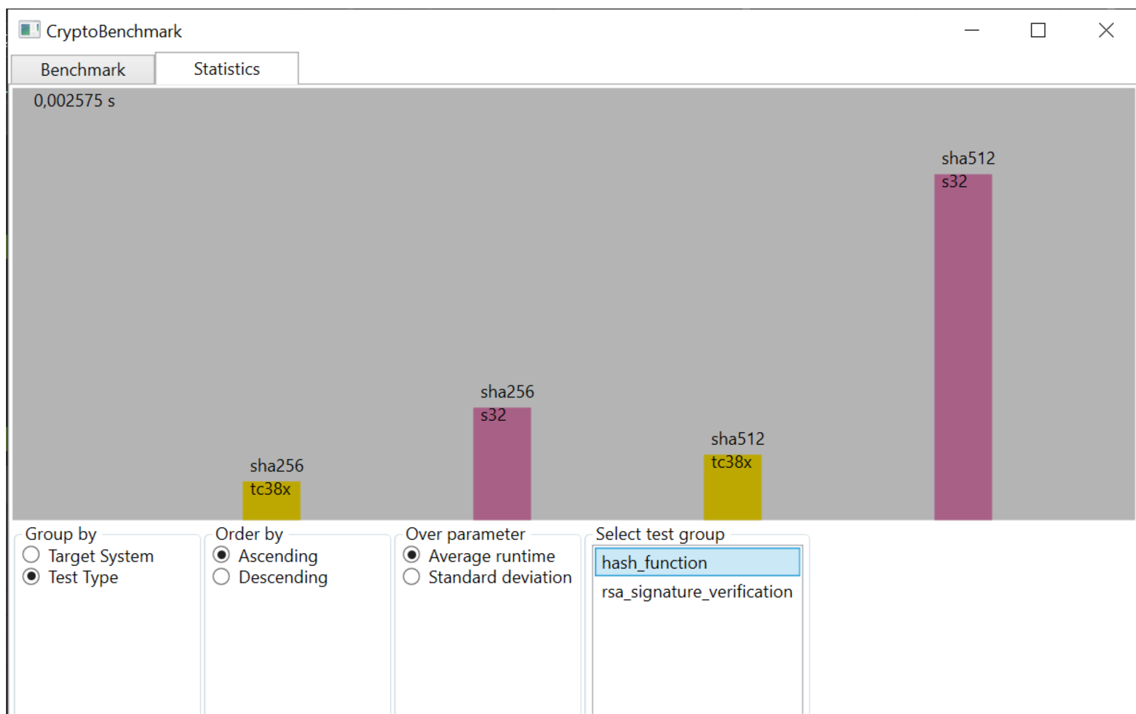
3.9-3. Ábra - Kezelőprogram felülete: Érvénytelen eredményű teszt (saját szerkesztés)

3.9.3 Statisztika nézet

A program indításakor fogadó alapértelmezett nézetből (3.9-1. Ábra) a felhasználónak lehetősége van átváltani a *Statistics* nézetbe a címsor alatt látható fülek segítségével, melyen egyszerű összesítő információt is kaphatunk a korábbi vizsgálatok eredményéről. A tárolt adatok megjelennek egy oszlopdiagramon, melyen platformonként színkódolva láthatók a különböző műveletek tesztjeinek átlagos végrehajtási ideje és szórása. A diagramon látható oszlopok csoportosíthatók vezérlő vagy tesztek szerint, emellett növekvő és csökkenő sorrendbe is rendezhetők. Az diagram felső szélsőértéke automatikusan mindig az adott tesztcsoport (például hash műveletek) maximális értékénél 25%-al nagyobb, így a program ezen nézete képes skálázni a megjelenített időtartományt.

Az adatok betöltése XML állományból történik, melynek szerkesztése a kiemeneten megjelenő „RUN:” feliratot követő számérték eltárolásával történik, egy teszt végrehajtása során. A dolgozat írásának idejében a statisztika nézet és a használt XML séma nem teszi lehetővé a használt kriptográfiai meghajtók szerinti különbségtételt, sem

az azok szerinti rendszerezést, az átlagok számításához a teszt adott vezérlőn történő futtatásának összes eredménye beleszámít.



3.9-4. Ábra – Kezelőprogram felülete, statisztika nézet: átlagok tesztenkéni csoportosításban, növekvő sorrendbe rendezve (saját szerkesztés)

3.10 Továbbfejlesztési lehetőségek

Az alábbi alfejezetben kerülnek tárgyalásra a keretrendszer tervezésekor és használata során felmerült továbbfejlesztési lehetőségek, ötletek melyek jelen dolgozat megírásának idejében nem valósultak meg.

3.10.1 További kriptográfiai függvénykönyvtárak integrálása

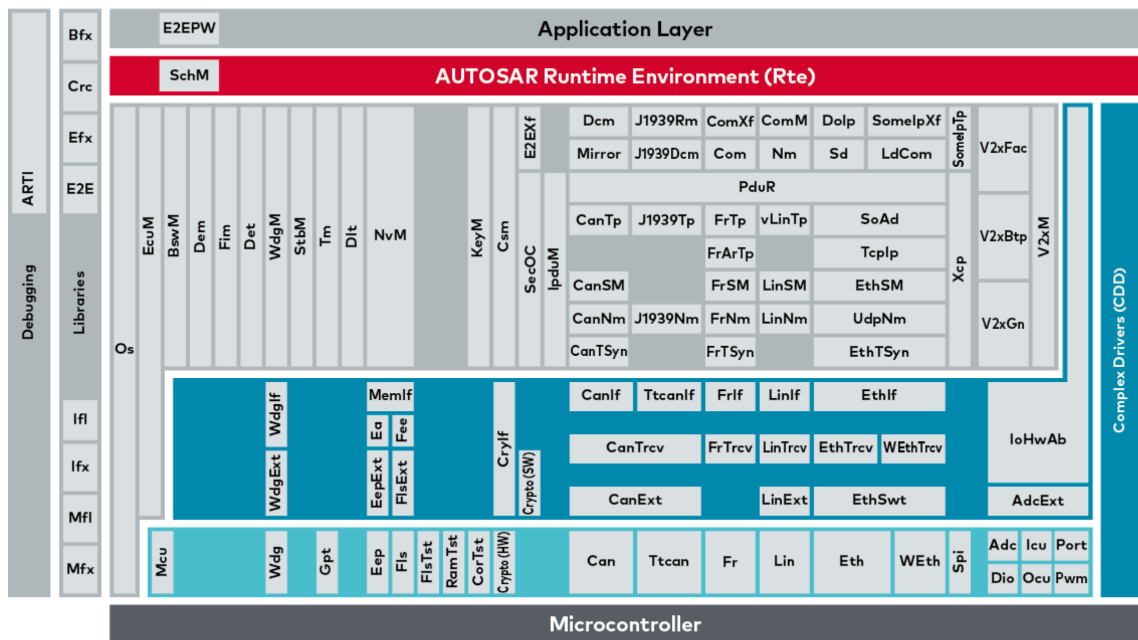
Az mbedTLS széles körben használt, de nem az egyetlen beágyazott rendszerek számára is alkalmas kriptográfiai könyvtár. A keretrendszer továbbfejlesztésének tervei között szerepel, a *liblithium*, a Tesla vállalat és közösség által fejlesztett, nyílt forrású függvénykönyvtárának integrálása kriptográfiai meghajtóként. A projekt leírása alapján, egy hordozható kódbázisú, alacsony energiaigényű rendszerekhez tervezett könyvtárat ígér, mely mindössze egy C99-es szabványt támogató C fordítóval használható további függőségek nélkül. [16]

Az ingyenesen hozzáférhető nyílt opciók mellett léteznek zárt licenszelésű kereskedelmi forgalomban lévő termékek, a mikrovezérlők gyártói, más autóipari cégek vagy a thyssenkrupp vevőinek egyedi implementációi.

3.10.2 AUTOSAR szoftverarchitektúrába illesztés

Az AUTOSAR, egy autóipari mérnökök és vállalatok bevonásával közösen létrehozott és használt szoftverarchitektúra. Létrehozásának célja hogy a elektronikus vezérlőegységeket fejlesztő vállalatok számára egységes szoftverfejlesztési környezetet, nyelvezetet, rendszert biztosítson. Az autóiparban használt mikrovezérlők szoftverének három fő rétegét különíti el egymástól. Előírja, hogy a legfelső applikációs réteg teljes mértékben hardverfüggetlen, számára egységes interfészt biztosít a *Runtime Environment* (RTE), a *Basic Software* (BSW) funkcióinak eléréséhez, mely a vezérlő meghajtóit és ECU absztrakciós alrétegeket és azokat összefogó szolgáltatási modulokat tartalmaz. [17]

A keretrendszer tervezése során az AUTOSAR architektúra jelent meg elsődleges mintaként lévén a tanulmányaim során használt a személyi számítógépekre írt alkalmazások fejlesztésekor gyakran használt klasszikus háromrétegű architektúra (*Presentation Layer, Business Layer, Data Access Layer*) elmélete nem értelmezhető. A fejlesztés során merült fel, hogy az újabb vezérlők integrálása lényegesen egyszerűbb lenne, ha a fejlesztés csupán az applikációs réteg szintjén lenne szükséges. A jelenleg támogatott két vezérlő is rendelkezik teljes mértékben AUTOSAR kompatibilis szoftvercsomaggal, így a támogatásuk az AUTOSAR modulok beállításán kívül, további kódolási feladatokat nem igényelnének. A jövőben egy reális és logikus továbbfejlesztési cél lehet a keretrendszer átalakítása egy AUTOSAR kompatibilis applikációvá.



3.10-1. Ábra - AUTOSAR Classic szoftverarchitektúra (Forrás: [18])

3.10.3 Megfelelés a MISRA-C irányelveknek

A MISRA-C, egy főként beágyazott rendszerek számára megfogalmazott kódolási irányelv gyűjtemény. Megbízható kód írása C programozási nyelven kifejezetten nehéz feladat, a MISRA-C olyan bevált gyakorlatokat, szabályokat és kódolási technikákat tartalmaz melyek használata elengedhetetlen a megfelelő üzembiztosság eléréséhez különösen olyan rendszerek programozása esetén melyek biztonságkritikus szerepet töltenek be egy nagyobb rendszerben és/vagy a programnyelv régebbi változatát (ANSI C, C90) használják (például *legacy rendszerek*). Míg a tárgyalt keretrendszer tervezése során nem merültek fel ilyen követelmények azonban a tipikusan C nyelvben megjelenő szemantikai hibák nagyrésze elkerülhető az irányelvek figyel. A statikus kódanalitikai eszközök mint például a *SonarQube* vagy a *cppcheck* sok esetben támogatják a MISRA szabályok betöltését és forráskód validálását.

4 Eredmények

Az alábbi fejezetben olvasható a dolgozat írásának idejében megvalósított tesztek végrehajtási ideje a két mikrovezérlőn futtatva. A szoftveres kriptográfiai meghajtóval végzett számítási teljesítmény összehasonlításához mindkét platform a keretrendszer által támogatott mbedTLS 2.25.0 verziószámú változatát használta. A végrehajtható állományok létrehozásakor mindkét vezérlőhöz felhasznált fordítók a válaszható legmagasabb optimalizációs beállításokkal kerültek paraméterezésre. A táblázatokban tesztenkénti 10 futtatásból vont átlag értékek szerepelnek.

4.1 Elkészült tesztek

Az alábbi alfejezetben olvashatók a jelenleg megvalósított vizsgált algoritmusok/komplex műveletek tesztjeinek leírásai, a hozzájuk tartozó forrásfájlok elnevezései alapján. A tesztek a fejezetben szereplő összehasonlító táblázatokban azonos névvel szerepelnek.

sha256, sha512: SHA-256 és SHA-512 hash műveletek elvégzése két különböző hosszúságú bemeneten. A teszt részét képzi az eredmény ellenőrzése, a teszparaméterek között elhelyezett előzetesen kiszámolt konstansokkal összevetve.

rsa_pss_sign: Előzetesen létrehozott RSA kulcspár általi aláírás elvégzése egy 64 byte hosszúságú bemenetnek. A teszt hash függvényként SHA-256 algoritmust használ. a kiválasztottal azonos kriptográfiai meghajtót használva.

rsa_pss_verify: Előzetesen létrehozott RSA-PSS-SHA256 típusú digitális aláírás ellenőrzése.

TRNG: 256 darab 4 byte méretű véletlen szám generálása. A teszt egyedül a hardveres biztonsági modulok valódi véletlen szám generáló alrendszereik tesztelésére jött létre. A 2.2.6 fejezetben is leírtak szerint, ezen rendszerek analóg forrásokból származó rendezetlenséget, véletlenszerű fizikai paramétereket használnak a generátor beállítására, melyet ismeretlen időnként meg is kell ismételniük.

4.2 Szoftveresen számított eredmények összehasonlítása

A szoftveres kriptográfiai meghajtón (mbedTLS) végrehajtott tesztek során előre elvárható eredmények születtek jelentős kilengések nélkül. A tesztek a TC3XX esetén az Infineon saját tervezésű TriCore architektúrájú processzorán futottak 300Mhz órajelű processzorciklussal, míg a K3XX esetén egy ARM Cortex-M7 feldolgozó egységen 120Mhz órajellel.

4.2-1. Táblázat - Vizsgált kriptográfiai algoritmusok végrehajtási ideje (Szoftveres meghajtó)

Mikrovezérlő / Vizsgált művelet	TC3XX (mbedTLS)		K3XX (mbedTLS)	
Bemenet mérete	**24 byte**	**64 byte**	**24 byte**	**64 byte**
sha256	19 µs	37 µs	68 µs	114 µs
sha512	39 µs	40 µs	206 µs	212 µs
RSA kulcs mérete	**1024 bit**	**2048 bit**	**1024 bit**	**2048 bit**
rsa_pss_sha256_sign	0.137 s	0.758 s	0.394 s	1.784 s
rsa_pss_sha256_verify	4.538 ms	17.316 ms	9.112 ms	30.903 ms

A kriptográfiai primitív műveletek (például hash műveletek) esetén az eltérés jelentősebb (kb. 2.5-3 -szoros), míg a több elemi műveletet magába foglaló komplex feladatok (például aláírás ellenőrzés) ez az arány csökkenni látszik. Egyetlen tesztelési munkamenet során a műveletek elvégzése többször megtörténik ugyanazon bemeneten. Az vezérlők erősen determinisztikus viselkedése miatt az átlagtól nagyobb eltérés, nem tapasztalható, elenyésző standard hibával rendelkezik, azonban egy jelenleg ismeretlen tényező miatt, az összes művelet első futtatása 5-15% nagyobb végrehajtási időt vesz igénybe mindkét platform esetén, ezt az átlagok számításakor nem vettem figyelembe. Az anomália kivizsgálására a dolgozat írásának idejében nem került sor.

4.3 Hardveresen gyorsított számítás összehasonlítása

A dolgozat írásakor a hardveres kriptográfiai meghajtók implementációi közül a feltüntetett teszteket támogató hardveresen gyorsított műveleteket irányító programkód el. (4.3-1. Táblázat) A TRNG teszt jellegéből fakadóan nem támogat szoftveres meghajtót. Fontos kiemelni, hogy a TRNG teszt csupán a hardveres biztonsági modulok véletlenszám generátorának végrehajtási ideje vizsgálható, a kimenetben szereplő véletlen számok minősége, eloszlása nem. Léteznek programok melyek alkalmasak, ilyen például a *NIST* amerikai szabványügyi szervezet és laboratóriumtól származó nyílt forrású *Statistical Test Suite (NIST-STS).* A szoftver különböző statisztikai és valószínűségi modellek alapján pontozza a számhalmazt, hasonló szolgáltatás a keretrendszer feladatkörébe jelenleg nem tartozik.

4.3-1. Táblázat - Vizsgált kriptográfiai algoritmusok végrehajtási ideje (Hardveres meghajtók)

Mikrovezérlő / Vizsgált művelet	TC3XX (HSM)		K3XX (HSE)	
Bemenet mérete	**24 byte**	**64 byte**	**24 byte**	**64 byte**
sha256	13 µs	24 µs	31 µs	39 µs
sha512	Nem támogatott		51 µs	52 µs
TRNG	17.654 ms		41.249 ms	

4.4 Értékelés

Véleményem szerint a keretrendszer a követelményeknek megfelelően elkészült, azonban fejlesztés során, felmerült újabb igények, továbbfejlesztési lehetőségek és a dolgozat írásának idejében még el nem készült tesztek mennyiségének ismeretében kijelenthető, hogy nem nyerte el végleges formáját. A tervezési időszakban és a két felhasznált mikrovezérlő képességeinek tanulmányozása során világossá vált, hogy a futtatási környezet, fordítók és minden egyéb szükséges segédprogram meghatározása időigényes feladat, a keretrendszer jelenlegi állapotában még nem áll készen sem a fejlesztők, sem a véghasználók kiszolgálására.

Mindenek felett a használt szoftver nagy segítséget nyújthat további vezérlők integrációjában, míg az elsőként kipróbált vezérlő saját hardverfüggő forráskódja a keretrendszerrel egy időben készült el, addig a második esetén csak minimális módosításokat kellett végezni a független kódbázison, és a fejlesztés időigénye is lényegesen csökkent.

A vállalat hosszútávú tervei között szerepel, hogy a vevői kiberbiztonsági követelmények teljesíthetőségi analízise során a keretrendszer biztosította lehetőségekkel támogassa a jövőbeli kormányrendszerek korai szoftvertervezési fázisait.

5 Összefoglalás

A feladat elvégzése során megismerkedtem a járműipari szoftverfejlesztés folyamataival, biztonságkritikus beágyazott rendszerek működésével magas absztrakciós szinten. Találkoztam továbbá számos, az iparágban használt kommunikációs technológiával, AUTOSAR szoftverarchitektúrát, elmélyítettem tudásomat a C programnyelv hardverközeli alkalmazásában, és nem utolsó sorban a dolgozatban szereplő mikrovezérlő termékcsaládok tanulmányozása során megtanultam feldolgozni nagy kiterjedésű, részletes dokumentációt. A vállalati termékek kiberbiztonsági követelményeinek megismerése kapcsán tanulhattam a különböző kriptográfiai eljárások szerepéről, gyakorlati alkalmazási területeiről. Hasznosítani tudtam a képzésen hallgatott ismereteket, külön kiemelve a *Hardver alapok, Kódolás és IT biztonság* illetve a *Szoftvertechnológia és -technikák* tárgyak tananyagát.

Szakmai mentorom támogatásával részben beleláthattam szoftverfejlesztői, integrátori feladatkörökbe és elkészíthettem egy olyan szoftvercsomagot mely reményeim szerint hasznosnak bizonyul a vállalat számára. Bízom benne, hogy a gyakornokként végzett munkámból származó tapasztalataim hozzásegítettek, hogy a jövőbeli szakmai tevékenységem maradandó értéket képviseljen.

6 Irodalomjegyzék

[1] F. Y. Rashid, "*Hacker History: The Time Charlie and Chris Hacked a Jeep Cherokee | Decipher*," 2018 Május 28. [Online]. Elérhető: https://duo.com/decipher/hacker-history-time-charlie-chris-hacked-jeep-cherokee.

[2] „*IBM - What is cybersecurity?,*" [Online]. Elérhető: https://www.ibm.com/topics/cybersecurity.

[3] K. David és S. G. Michael, *Fundamentals of Information Systems Security*, Burlington: Jones & Bartlett Learning, 2018, pp. 16-17.

[4] „*Frequently Asked Questions about Secure Boot*" [Online]. Elérhető: https://www.intel.com/content/www/us/en/support/articles/000006942/boards-and-kits/desktop-boards.html

[5] Clark, C., 2021. *Automotive Cybersecurity & OTA Vehicle Software Update*s. [Online] Synopsys Blog - From Silicon To Software. Elérhető: https://blogs.synopsys.com/from-silicon-to-software/2021/07/08/ota-software-updates-automotive-cybersecurity.

[6] „*UN R155 and UN R156 set framework for automotive cybersecurity ESCRYPT*" [Online]. Elérhető: https://www.escrypt.com/en/news-events/un-r155_un-r156

[7] J.-P. Aumasson, "Ch. 6, Sec. Hash Functions," in *Serious Cryptography - A Practical Introduction to Modern Encryption*, San Fransisco: No Starch Press, 2018.

[8] J.-P. Aumasson, "Ch. 1, Sec. The Basics," in *Serious Cryptography - A Practical Introduction to Modern Encryption*, San Fransisco: No Starch Press, 2018.

[9] J.-P. Aumasson, "Ch. 3, Sec. Measuring Security in Bits," in *Serious Cryptography - A Practical Introduction to Modern Encryption*, San Fransisco: No Starch Press, 2018.

[10] J.-P. Aumasson, "Ch. 7, Sec. Forgery and Chosen-Message Attacks," in *Serious Cryptography - A Practical Introduction to Modern Encryption*, San Fransisco: No Starch Press, 2018.

[11] J.-P. Aumasson, "Ch. 2, Sec. Random Number Generators (RNGs) and Pseudorandom Number Generators (PRNGs)," in *Serious Cryptography - A Practical Introduction to Modern Encryption*, San Fransisco: No Starch Press, 2018.

[12] D. Giry, „*Keylength - BSI Cryptographic Key Length Report (2020),”* 2020. Május 25. [Online]. Elérhető: https://www.keylength.com/en/8/. [Hozzáférés dátuma: 25 Március 2022].

[13] R. Lakshmanan, „*A Critical Random Number Generator Flaw Affects Billions of IoT Devices*,” 9 Augusztus 2021. [Online]. Elérhető: https://thehackernews.com/2021/08/a-critical-random-number-generator-flaw.html. [Hozzáférés dátuma: 21 Április 2022].

[14] J. Vranish, „*A Super-Simple Makefile for Medium-Sized C/C++ Projects*,” 2016 Augusztus 26. [Online]. Available: https://spin.atomicobject.com/2016/08/26/makefile-c-projects/.

[15] „*Let mbed TLS use static memory instead of the heap*,” [Online]. Elérhető: https://tls.mbed.org/kb/how-to/using-static-memory-instead-of-the-heap.

[16] “*liblithium,*” [Online]. Elérhető: https://github.com/teslamotors/liblithium [Hozzáférés dátuma: 19 Május 2022.]

[17] „*AUTOSAR - Classic Platform*,” AUTOSAR, 2022. [Online]. Elérhető: https://www.autosar.org/standards/classic-platform/.

[18] „*AUTOSAR Classic | Vector*,” Vector Informatik GmbH, 2022. [Online]. Elérhető: https://www.vector.com/at/en/know-how/autosar/autosar-classic/.

7 Függelék

A Abbreviációk

MCU: Microcontroller Unit

ECU: Electronic Control Unit

HSM: Hardware Security Module

HSE: Hardware Security Engine

CPU: Central Processing Unit

iLLD: Infineon Low-Level Drivers

CMSIS: Common Microcontroller Software Interface Standard

U(S)ART: Universal (Synchronous) Asynchronous Receiver Transmitter

PLL: Phase-Locked Loop

UDS: Universal Diagnostic Service

SID: Service Identifier

IVT: Interrupt Vector Table

TVT: Trap Vector Table

IP/PC: Instruction Pointer / Program Counter

PRNG: Pseudo-Random Number Generator

TRNG: True Random Number Generator

CSPRNG: Cryptographically Secure Pseudo Random Number Generator

HAL: Hardware Abstraction Layer

PAL: Platform Abstraction Layer

CSM: Cryptography Service Module

BSW: Basic Software

B Kódrészletek

B-1 Fordítási időben ki és bekapcsolt implementációk

```
...
void cryif_hash_sha512(CryInstance cryinst, CryCtx_Hash512* hash_ctx)
{
     switch(cryinst) {
        #ifdef CRYINST_MBEDTLS
            case CryInst_MbedTLS:
                cry_mbedtls_hash_sha512(hash_ctx);
                break;
        #endif
        #ifdef CRYINST_TC_HSM
            case CryInst_TC_HSM:
                NOT_IMPLEMENTED_EXCEPTION;
                break;
        #endif
        #ifdef CRYINST_S32_HSE
            case CryInst_S32_HSE:
                cry_hse_hash_sha512(hash_ctx);
                break;
        #endif
            default : return;
    }
}
...
```

B-2 Közösen használt típusok definíciója (részlet)

```
#ifndef CRYTYPES_H
#define CRYTYPES_H

/*** Támogatott crypto eljárások ***/
typedef enum {
    CryInst_MbedTLS,
    CryInst_TC_HSM,
    CryInst_S32_HSE
} CryInstance;

/*** Crypto műveletekhez tartozó típusok ***/
typedef struct {
    char input[64];
    int input_len;
    int is_224;
    char output[32];
    char error[32];
} CryCtx_Hash256;

typedef struct {
    char input[32];
    int input_len;
    int is_384;
    char output[64];
    char error[32];
```

```
} CryCtx_Hash512;

typedef struct {
    unsigned char sig[1024];
     unsigned char hash[32];
    unsigned char key[32];
    unsigned char textToEncrypt[32];
    int error;
} CryCtx_RSA;
...
```

B-3 mbedTLS statikus buffer létrehozása (részlet)

```
void Csm_Init(void) {
...
     #ifdef CRYINST_MBEDTLS
          //allocate a buffer for mbedTL (Spec: target MCUs musn't use heap)
          static unsigned char memory_buf[20000];
          mbedtls_memory_buffer_alloc_init(memory_buf,
sizeof(memory_buf));
     #endif
...
}
```

[15]

B-4 mbedTLS fordítása statikus könyvtárként (részlet)

```
...
static: libmbedcrypto.a libmbedx509.a libmbedtls.a

# tls
libmbedtls.a: $(OBJS_TLS)
     @tput setaf 6
     echo "Archiving $@..."
     $(AR) $(ARFLAGS) $@ $(OBJS_TLS)

# x509
libmbedx509.a: $(OBJS_X509)
     @tput setaf 6
     echo "Archiving $@..."
     $(AR) $(ARFLAGS) $@ $(OBJS_X509)

# crypto
libmbedcrypto.a: $(OBJS_CRYPTO)
     @tput setaf 6
     echo "Archiving $@..."
     $(AR) $(ARFLAGS) $@ $(OBJS_CRYPTO)
.c.o:
     #echo "  CC    $<"
     @tput setaf 3
     @echo "($(PROGRESS)/$(OBJ_COUNT)) $@"
     $(eval PROGRESS=$(shell echo $$(($(PROGRESS)+1))))
     $(CC) $(LOCAL_CFLAGS) $(CFLAGS) -w -o $@ -c $< -g3
...
```

B-5 NXP S32 K3XX makefile (részlet)

```
#fordító toolchain
CC = utils/gcc-10.2-arm32-eabi/bin/arm-none-eabi-gcc
LD = utils/gcc-10.2-arm32-eabi/bin/arm-none-eabi-gcc

MBEDTLS_DIR = ./src/lib/mbedtls/library/
BUILD_DIR ?= ./build/$(MCU)/
LIB_ARCHIVES ?= build/$(MCU)/lib/
LIB_DIRS ?= ./src/lib/mbedtls_arm

MAIN_EXEC ?= benchmark

#fájl listák feldolgozása
SRCS := $(shell find $(SRC_DIRS) -name *.c)
SRCS += $(shell find $(MBEDTLS_DIR) -name *.c)
OBJS := $(SRCS:%=$(BUILD_DIR)/%.o)
DEPS := $(OBJS:.o=.d)

INC_DIRS := $(shell find $(SRC_DIRS) -type d)
INC_DIRS += $(shell find $(LIB_DIRS) -type d)
INC_FLAGS := $(addprefix -I,$(INC_DIRS))

STARTUP_CODE = src/mcu/s32/bsw/Startup_Code/
LINKER_SCRIPT = src/mcu/s32/bsw/linker_flash.ld
OPTIONS = -std=c99 \
-DD_CACHE_ENABLE \
-DI_CACHE_ENABLE \
-DENABLE_FPU \
-DGCC \
-DS32K3XX \

...
...
...

all: startup_cm7 Vector_Table mbedtls $(BUILD_DIR)/$(MAIN_EXEC)
main: $(BUILD_DIR)/$(MAIN_EXEC)

$(BUILD_DIR)/$(MAIN_EXEC): $(OBJS)
	@tput setaf 7
	@echo "Copying libs for gcc..."
	@$(MKDIR_P) $(LIB_ARCHIVES)
	cp $(MBEDTLS_DIR)libmbedcrypto.a $(LIB_ARCHIVES)libmbedcrypto.a
	cp $(MBEDTLS_DIR)libmbedtls.a $(LIB_ARCHIVES)libmbedtls.a
	cp $(MBEDTLS_DIR)libmbedx509.a $(LIB_ARCHIVES)libmbedx509.a
	@echo "Building ELF..."
	$(LD) $(LDFLAGS) $(OBJS) $(ASM_OBJS) -o $@.elf
...
```

B-6 Infineon Aurix TC3XX makefile (részlet)

```
#elérési útvonalak
CC = utils/tasking/ctc/bin/cctc
AS = utils/tasking/ctc/bin/astc
AR = utils/tasking/ctc/bin/artc
LD = utils/tasking/ctc/bin/cctc

MBEDTLS_DIR = ./src/lib/mbedtls/library/
BUILD_DIR ?= ./build/$(MCU)/
LIB_BUILD_DIR ?= build/$(MCU)/lib/
LIB_DIRS ?= ./src/lib/mbedtls

MAIN_EXEC ?= benchmark

#fájl listák feldolgozása
SRCS := $(shell find $(SRC_DIRS) -name *.c)
OBJS := $(SRCS:%=$(BUILD_DIR)/%.o)
DEPS := $(OBJS:.o=.d)

INC_DIRS := $(shell find $(SRC_DIRS) -type d)
INC_DIRS += $(shell find $(LIB_DIRS) -type d)
INC_FLAGS := $(addprefix -I,$(INC_DIRS))

#tasking compiler flagek
LINKER_SCRIPT                                                           =
utils/tasking/ctc/include.lsl/Lcf_Tasking_Tricore_Tc_HSM_Aligned.lsl
OPTIONS_OUT_FORMAT_SREC = .sre:SREC:4
OPTIONS_OUT_FORMAT_HEX = .hex:IHEX:4
OPTIONS_OUT_FORMAT_ELF = .elf:ELF
OPTIONS_LSL = --lsl-file $(LINKER_SCRIPT)
OPTIONS_LSL_CORE = --lsl-core tc0
OPTIONS_CORE = --core tc1.6.2
ERRORS_FILE = #$(LOG_DIR)/LastBuild_errors.log
OPTIONS_DEBUG_SYMBOLS = -g3
CFLAGS = $(OPTIONS_CORE) $(OPTIONS_DEBUG_SYMBOLS)

#linker flagek
LD_USER_LIBS = -lmbedtls -lmbedcrypto -lmbedx509
LDFLAGS = -L$(LIB_BUILD_DIR) $(LD_USER_LIBS) $(OPTIONS_LSL)
$(OPTIONS_LSL_CORE)
```

B-7 Támogatott kriptográfiai meghajtókat tartalmazó sémá

```
<?xml version="1.0"?>
<cryinstances>
<instance type="sw">Példa: Szoftveres kriptográfiai könyvtár</instance>
<instance type="hw">Példa: Hardveres biztonsági modul</instance>
</cryinstances>
```

B-8 Programozó eszköz irányítása PRACTICE script-el (részlet)

```
...
PRIVATE &script_location
&script_location = OS.PresentPracticeDirectory()
ChDir "&script_location"

DATA.LOAD.Elf \build\tc3xx\benchmark_TC.elf /DUALPORT /QUAD /NOCODE /INCLUDE

InterCom.Execute localhost:10001 SYStem.CPU &CPUTYPE
InterCom.Execute localhost:10001 SYStem.CONFIG.CORE 2 1
InterCom.Execute localhost:10001 SYStem.Mode Attach
InterCom.Execute localhost:10001

DATA.LOAD.Elf \build\tc3xx\benchmark_ARM.elf /DUALPORT /QUAD /NOCODE /INCLUD
...
```

B-9 Teszt paramétereket tartalmazó fejlécfájl (példa)

```
//****** Test vector ******
#define NUM_OF_TEST_INPUTS 2
const unsigned char tst_input[NUM_OF_TEST_INPUTS][64] = {
"idFvYujJscByrWZKvIM7G5by",
"utSD27aX3Ag8JvBBBblYP5YjAvwCFpwOV0Ht7nvawSJ8xuiFXRge2Y2pHFfRVo5K" };
const unsigned int tst_input_len[NUM_OF_TEST_INPUTS] = { 24, 64 };
const unsigned char tst_expected[NUM_OF_TEST_INPUTS][32] = {
    {
        0x45, 0xb9, 0xdf, 0xc1, 0xa3, 0x64, 0x06, 0x65, // 0x00000000
        0xe7, 0xdd, 0xae, 0x40, 0xf7, 0xf9, 0xc0, 0x99, // 0x00000008
        0xfc, 0x4e, 0x62, 0xc4, 0x4b, 0x85, 0x4f, 0x3e, // 0x00000010
        0xc1, 0xe3, 0xae, 0xf7, 0x91, 0x66, 0x4b, 0xa7  // 0x00000018
    },
    {
        0x20, 0x25, 0xb8, 0x74, 0xe4, 0x8c, 0x0c, 0xff, // 0x00000000
        0x49, 0x92, 0x6e, 0xa2, 0xbb, 0x0c, 0x37, 0xdc, // 0x00000008
        0x53, 0x05, 0x29, 0x84, 0xec, 0x5d, 0xa3, 0x1c, // 0x00000010
        0x75, 0xc8, 0xf8, 0x6b, 0x78, 0x47, 0xca, 0xfc  // 0x00000018
    }
};

const unsigned int tst_test_count = 3;

//****** Crypto driver instance ******
//****** GUI által kiválasztott! ******
const CryInstance selected_cryinst = CryInst_MbedTLS;
```