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

Kupper Benedek

AUTENTIKÁCIÓ ÉS REJTJELEZÉS AUTÓIPARI BEÁGYAZOTT RENDSZEREKBEN

KONZULENSEK

Dr. Sujbert László
Dr. Balogh András

BUDAPEST, 2014

MŰEGYETEM 1782

DIPLOMATERV-FELADAT

Kupper Benedek (DKQXM9)
szigorló villamosmérnök hallgató részére

Autentikáció és rejtjelezés autóipari beágyazott rendszerekben

A modern gépjárművek biztonságtechnikai és kényelmi funkcióinak megvalósításában, környezetvédelmi jellemzőinek javításában stb. egyre jelentősebb szerepet kapnak a számítástechnikai megoldások. Ma egy prémium személyautó gyártójának közel száz elektronikus vezérlőegységből (ECU) és számos fedélzeti kommunikációs sínből kell kialakítani egy megbízhatóan működő elosztott rendszert, amely komoly algoritmus- és kommunikációtervezési, illetve munkaszervezési kihívást jelent. Napjainkban már különböző külső eszközöket (mobiltelefon, multimédia lejátszó, internet átjáró) is csatlakoztatnak a járművek rendszereihez, így egy új aspektus, a biztonság (security) is egyre fontosabb szerepet kap a tervezés során.

A biztonsági megoldások többsége még előfejlesztési vagy koncepcionális fázisban van, a hatékony megvalósításhoz szükséges hardvertámogatás sem feltétlenül adott a beágyazott processzorokban. Ugyanakkor már részletesen megvizsgálták a lehetséges támadási módokat, illetve a szükséges funkciók halmazát. Jelen feladat célja az eddigi vizsgálati eredmények összegzése, az elektronikus kormányszervó rendszerek szempontjából releváns funkciók és támadási módok kiválasztása, majd néhány kiválasztott megoldás implementálása. A hallgató feladata magában foglalja az alábbiakat:

- A meglevő kutatási eredmények megismerése (pl. EVITA projekt)
- A rendelkezésre álló védelmi mechanizmusok áttekintése
- A kormányrendszerek esetén releváns támadási módok áttekintése
- Néhány védelmi mechanizmus implementálása céleszközön (SW-ben)
- Az új mikrokontrollerek hardver rejtjelező magjainak áttekintése
- Néhány védelmi mechanizmus implementálása (HW támogatással)
- A SW és HW megoldások összehasonlítása teljesítmény és védelmi szint szempontjából

A feladat megvalósításához szükséges eszközöket a ThyssenKrupp Presta Hungary Kft. biztosítja.

Tanszéki konzulens: Dr. Sujbert László
Külső konzulens: Dr. Balogh András (ThyssenKrupp Presta Hungary Kft.)

Budapest, 2014. március 3.

.....
Dr. Jobbágy Ákos
tanszékvezető

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

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Kupper Benedek**, szigorló hallgató kijelentem, hogy ezt a diplomatervet 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, 2014. 12. 20.

.....
Kupper Benedek

Összefoglaló

A modern személygépjárművek évről-évre több elektronikus vezérlőegységet tartalmaznak, egyre nagyobb sávszélességű adatkapcsolattal egymás között, és ez a fedélzeti kommunikációs felhő megkezdte terjedését a jármű határain túlra is. Egyre több vezeték nélküli kapcsolattal és egyre több autonóm vezérlési rendszerrel kerülnek az autók az utcákra, mégis egy nagyon fontos aspektus eddig igen kevés szerepet kapott az autóipari rendszerekben. Ez pedig nem más, mint a cybertámadások elleni hatékony védekezés. A munkám során bemutatom a közelmúltban feltárt sebezhetőségeket, és a biztonsági modell alapján megvizsgálom, hogy milyen intézkedések, autentikációs és rejtjelező algoritmusok, valamint hardver és szoftver megoldások szükségesek egy autóipari beágyazott környezetben.

A legsúlyosabb biztonsági rést jelentő diagnosztikai kommunikáció lett a kiválasztott célterület, melyhez fel kellett építenem egy teljes diagnosztikai környezetet. Ez egy cél ECU-ból, egy saját CAN-USB átjáróból valamint egy PC-n futó felhasználói alkalmazásból állt. Létrehoztam egy kompakt, hordozható, beágyazott rendszerekben alkalmazható kriptográfiai C könyvtárat, mely minden alapvető kriptográfiai eljárást támogat. Beágyazott platformon megvizsgáltam az eljárások teljesítményét, melyet összehasonlító elemzésnek vetettem alá. A biztonsági modell alapján kialakítottam a szabványos diagnosztikai kommunikációba ágyazva egy olyan ECU biztonsági hozzáférési sémát, mely a diagnosztikai klienst annak kriptográfiai aláírása alapján hitelesíti, emellett egy közös titkos kulcsot is létrehoz a felek számára. Ezután az ECU memóriájába való feltöltés és az onnan történő letöltés diagnosztikai szekvenciáit valósítottam meg többféle titkos kulcs alapú kriptográfiai eljárás használatával. Végül ezt a megoldást továbbfejlesztettem egy programmemória feltöltő alkalmazássá. Zárszóként áttekintem, hogy hardveres biztonsági modullal rendelkező ECU-ban ezek a megoldások mennyiben változnának meg.

Abstract

Modern automobiles include more and more electronic control units (ECUs) by the years, with these having ever so increasing coupling via on-board buses between them. This cloud of interconnected units has begun to expand beyond the physical limits of the vehicle. Today's automobiles feature a never before seen range of wireless communication capabilities and advanced driver assistance systems. One major aspect of communication systems has been mostly ignored by the automotive industry, the security of the vehicle networks and components against adversaries. In this present the recently identified attack surfaces and system vulnerabilities, which leads me to create a security model of the on-board and diagnostic communication. I investigate the cryptographic, hardware and software building blocks needed to create a secure ECU communications interface to the on-board networks.

Based on the identified system weaknesses I selected the most vulnerable attack surface of the ECU – the diagnostic communication – for a security application implementation. I constructed the necessary diagnostic hardware-software toolkit that can connect to the ECU, which is composed of a CAN-USB gateway and a PC user application. I have compiled a compact, portable cryptographic C software library, which features all basic cryptographic algorithms with the currently most widespread cryptographic ciphers and hash functions. I have conducted measurements on an embedded platform in order to determine the code size and run time costs of each cryptographic scheme. I created a custom security access protocol which is compliant to the underlying diagnostic standard, which verifies a diagnostic client by a cryptographic signature, and which also produces a shared secret key for the parties. I also added secret key-based cryptographic encryption and authentication capabilities to the security access-protected memory transfer services. This functionality was extended to support the upload of a new main software to the ECU. In the final chapter I make a brief overview of the security improvements provided by a hardware security module supporting ECU.

1 Bevezetés

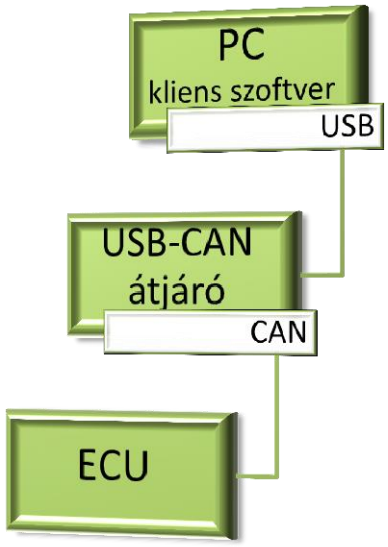
A diplomatervem témafelvetése onnan ered, hogy az elmúlt évtizedek során egyre nagyobb önállóságra és vezeték nélküli (GSM, Bluetooth, stb.) hálózatba kapcsoltságra szert tevő személygépjárművek esetén egyre több bizonyíték mutatkozott arra, hogy a cybertámadások egy potenciális célpontjai lehetnek a közeljövőben. Éppen ezért az ezredforduló után egymást érték az autóipari csoportok által indított kutatások, melyek a járművek beágyazott rendszereiben alkalmazható védelmi mechanizmusok implementációs lehetőségeit vizsgálták. Jelenleg már a piacon elérhetőek olyan autóipari célú mikrokontrollerek, melyek a kutatások eredményeként azonosított szükséges hardveres védelmi mechanizmusokat támogató modullal rendelkeznek. Azonban még így is éveket kell arra várni, mire az autógyártók kialakítanak egy teljes szoftverarchitektúrát, mely a járművek komplex beágyazott rendszereiben átfogó és megbízható védelmi szolgáltatásokat nyújt. Én a ThyssenKrupp Presta Hungary Kft-n keresztül ismertem meg ezt a problémát, mely cég egy innovációs és fejlesztő profilú autóipari beszállító:

„A liechtensteini központú ThyssenKrupp Presta budapesti leányvállalata 1999 óta foglalkozik személyautók elektromechanikus kormányrendszerének fejlesztésével a cégcsoporton belül egyedüli elektronikai és szoftverfejlesztési kompetenciaközpontként.” „A szoftverfejlesztés a ThyssenKrupp Presta Hungary Kft. egyik legfontosabb tevékenysége. A cég által tervezett és gyártott szervokormány rendszerek beágyazott programjai teljes egészében a budapesti fejlesztőközpontban készülnek, beleértve a fejlesztést, integrálást és tesztelést is. Ezekkel a tevékenységekkel a budapesti részleg szakembereinek több mint fele foglalkozik.” [1]

Mivel a személyautók kormányrendszerei igen magas minőségi és biztonsági követelményeknek kell eleget tenniük, ezért csak komoly fejlesztési kompetencia megléte esetén jöhet létre megfelelő termék. A szoftverfejlesztés a cégen belül sok ágra osztott, én az ECU szoftver csoport munkájában vettem részt a diplomatervezés során. Ennek a csoportnak a feladata az AUTOSAR szoftver architektúra ECU szoftverhez szükséges komponenseinek implementálása és komponens szintű tesztelése.

Munkám során megismerkedtem a kriptográfiával, az információtitkosítás és –hitelesítés tudományával. A közelmúlt autóipari kutatásainak vizsgálatával képet

nyertem arról, hogy a fedélzeti beágyazott rendszerek esetén milyen egyéb hardver, szoftver és egyéb követelmények kielégítése szükséges a biztonságos információtárolás és –átvitel megvalósításához. Létrehoztam egy hordozható, beágyazott rendszerekben alkalmazható kriptográfiai szoftverkönyvtárat, mely minden modern kriptográfiai sémára tartalmaz legalább egyféle megvalósítást. Elemeztem a könyvtár eljárásait biztonsági szint, memóriaigény és számítási igény szempontjából. Ezt a könyvtárat integráltam a cég által használt beágyazott platformon a diagnosztikai kommunikációs szofver stack-jébe, valamint egy dinamikusan betölthető könyvtárat készítettem belőle, mellyel Windows alkalmazások tudják használni. Felépítettem egy CAN busz alapú diagnosztikai kliens rendszert (lásd az alábbi ábrát), mely egy CAN-USB átjáróból és az azzal kommunikáló .NET kliens alkalmazásból áll. Erre az alkalmazásra építettem egy biztonságos diagnosztikai tesztkörnyezetet, mely szabványos üzeneteken keresztül titkosított és hitelesített adatcserét képes végezni a fedélzeti kontrollerrel. Ezt kibővítettem a kontroller szoftver frissítésének lehetőségével fordító kimeneti fájl alapján. Végül az elérhető irodalom alapján megvizsgáltam, hogy milyen előnyöket hordoz magában egy hardveres biztonsági modullal rendelkező kontroller.



1.1. ábra: A tesztkörnyezet vázlatos felépítése

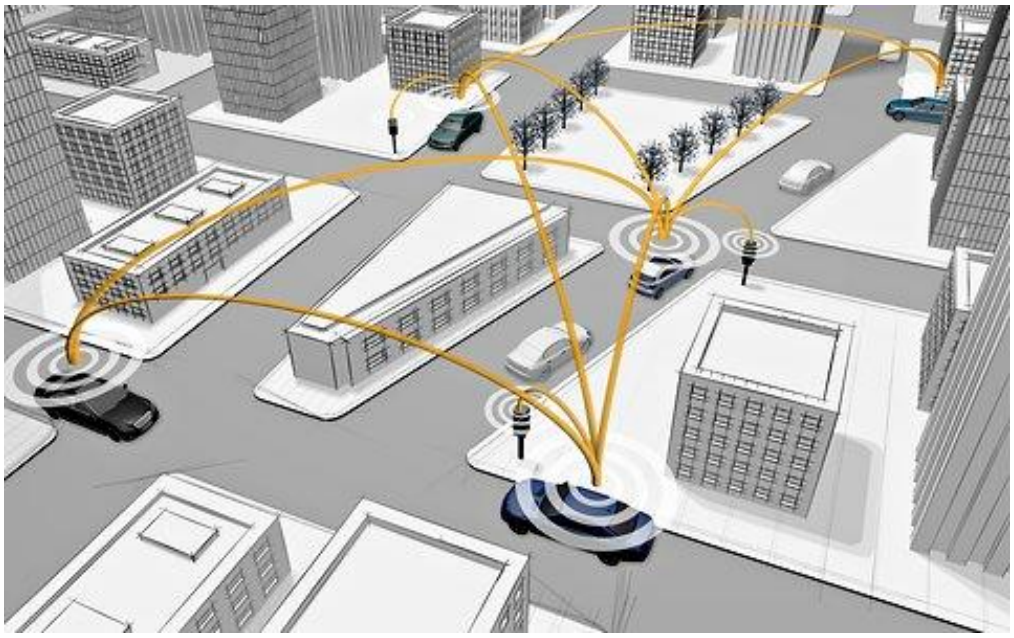
1.1 Autóipari fejlődési trendek

Az autóipar a világ reálgazdaságának egyik legnagyobb bevételt termelő ágazata, a világ ipari foglalkoztatásának 5 %-át közvetlenül foglalkoztatva [2]. Különösen nagy jelentősége van Magyarországon, ahol 2013-ban a GDP 10 %-át a járműipar termelte [3]. A manapság előállított személy- és tehergépjárművek a fogyasztói piac legösszetettebb termékei, melyben jelentős szerepet játszanak az egyre bővülő funkcionalitású fedélzeti beágyazott rendszerek. Egy modern személygépkocsiban akár 70 darab elektronikus vezérlőegység (ECU) is megtalálható, melyek működésüket az őket összekötő különböző kommunikációs hálózatok segítségével hangolják össze.

Piaci súlyának és nagymennyiségű eszközigényének köszönhetően az autóipar kiemelkedő vevői státuszhoz jutott az elektronikai piacon, valamint egy fontos célterülete a beágyazott rendszerek fejlesztésének. Becslések szerint az elektronika és a beágyazott szoftverek teszik ki egy mai jármű költségének 40 %-át, valamint az autóipari elektronikai iparág által és a célszoftverek fejlesztése során létrejött új megoldások teszik ki a modern gépjárművek innovációinak 90 %-át [4]. A három legmeghatározóbb fejlesztési irányzat a következő [5]:

1. Az autókba szerelt szenzorok számának és a beépített számítási kapacitás növekedésével egyre több és fejlettebb aktív biztonsági rendszert és vezetőt segítő rendszert (ADAS) képesek kivitelezni.
2. A mobilinternet széles körű elterjedésének köszönhetően egyre nagyobb a fogyasztói igény az olyan gépjárművek iránt, melyek képesek kapcsolódni a felhasználók mobilkészülékeihez és a világhálóhoz, melyek segítségével fejlett információs és szórakoztató rendszert tudnak nyújtani nekik – mindezt úgy, hogy az ne legyen alkalmas a vezető figyelmének elvonására (*Connected Cars*).
3. Az egyre szigorúbb károsanyag-kibocsájtási szabályozások miatt felértékelődött a hibrid és elektromos hajtásláncok technológiája. Komoly kutatások zajlanak többek között az újgenerációs akkumulátorok területén, ugyanis az elektromos autók jelenlegi legnagyobb gyengesége az akkumulátor kapacitásából adódó rövid hatótáv.

Az első két terület egymáshoz szervesen kapcsolódik, hiszen mindkettő a gépjármű fedélzeti elektronikai rendszerének bővítését célozza. Egy új fejlesztési irány jött létre a két terület egyesítésével, amely az autók érzékelési és számítási képességeit bővíti oly módon, hogy folyamatos kommunikációt hoz létre a járművek között (V2V), valamint a járművek és az intelligens közúti infrastruktúra között (V2I). Már napjainkban is a piacon vannak olyan gépjárművek, melyek rendelkeznek bizonyos kommunikációs képességekkel, például a beépített navigációs rendszerrel, nyomkövető rendszerrel vagy az automatikus segélyhívóval felszerelt autók.



1.2. ábra: Hálózatba kapcsolt közúti forgalom szemléltetése

Az autóforgalom kommunikációs csatornákkal történő összehangolására már az ezredforduló előtt történtek lépések. Az Egyesült Államok már 1999-ben kijelölt egy spektrumot az 5,9 GHz-es frekvencián, melyet kifejezetten a rövid hatótávú járműforgalmi kommunikációs célokra különített el, ez az ún. DSRC [6]. Azóta folyamatos kutatások zajlanak a témában, például 2002-ben egy erre a célra alakult, az autóipari piacot legnagyobb részben lefedő autógyártói konzorcium indította el a Vehicle Safety Communications projektet [7], melynek az volt a fő célja, hogy felmérje, a DSRC vezeték nélküli hálózat alkalmazásával mennyivel lehetne a közúti forgalmat biztonságosabbá tenni.

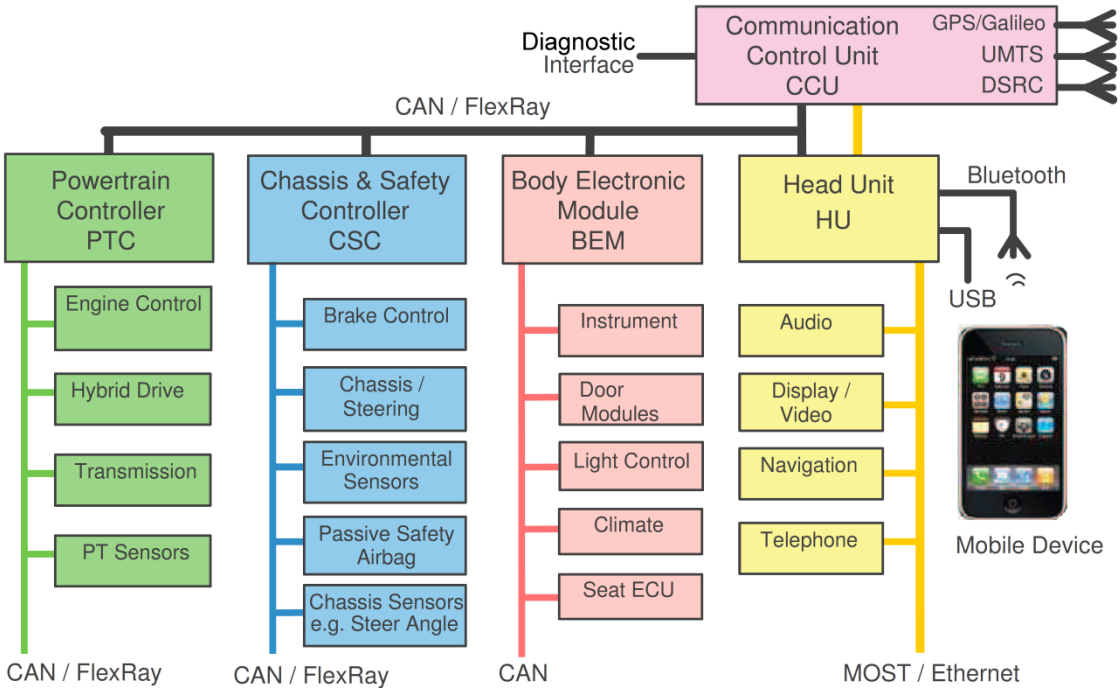
Összefoglalva tehát a következő évtizedek egyik fő autóipari fejlesztési trendje a járművek dedikált kommunikációs csatornákon keresztüli adatcseréjének és a bővülő képességű fedélzeti érzékelő, feldolgozó és beavatkozó rendszerek segítségével a közlekedésbiztonság növelése és a vezetés megkönnyítése. Azonban ahogyan a vezető feladatainak egyre nagyobb részét veszik át automata és távoli rendszerek, úgy egyre szélesebb körű megbízhatósági és biztonsági követelményeknek kell megfelelniük. Míg a funkcionális biztonságra már hosszú évtizedek óta komoly hangsúlyt fektetnek az autógyártók, addig a fedélzeti számítógép-rendszerek ellen indított szándékos beavatkozások kivédésével csak az ezredforduló után kezdtek el behatóan foglalkozni az autógyártók.

1.2 Fedélzeti buszrendszerek

A modern gépjárművek elosztott beágyazott rendszerek, a vezérlőegységek lokális mérési és beavatkozási feladatokat látnak el, közöttük buszrendszerek teremtenek kommunikációs kapcsolatot. Ennek a kialakításnak praktikus okai vannak. Centrális vezérlőrendszer esetén egyrészt a szükséges kábelezés a jármű különböző részein elhelyezett érzékelőkhöz és beavatkozószervekhez jelentős anyagi, valamint súly- és térfogatbeli költségekkel járna. Ezen felül e kábelek mindegyikét megfelelő zavarvédelemmel kellene ellátni, hogy se a vezérlőben, se a végberendezésben ne okozzon kárt. További szempont, hogy egy központi vezérlőegység esetén annak meghibásodása a teljes járművet használhatatlanná tenné. Ezen okokból kifolyólag alkalmaznak elosztott rendszereket, melyeket egy- vagy kétvezetékes kábelt igénylő, zavar- és hibatűrő buszokkal hoznak összeköttetésbe. A járművekben elterjedt buszrendszereket az alábbi felsorolás csoportosítja [8]:

- Másodlagos buszok: LIN, K-Line, I^2C
- Eseményvezérelt buszok: CAN, VAN
- Idővezérelt buszok: FlexRay, TTCAN, TTP
- Multimédiás célú buszok: MOST, D^2B, GigaStar
- Vezeték nélküli hálózatok: Bluetooth, GSM, WLAN

A másodlagos lokális buszok – mint például a LIN – kisméretű autonóm hálózatokat vezérelnek, például a központi zárrendszert, az elektromos ablakemelőket és tükröket, vagy különböző érzékelőket kezelő rendszereket. Az eseményvezérelt buszok – legelterjedtebb közülük a CAN – az ECU-k közötti nem valósidejű követelményű kommunikációra szolgál, például a motorvezérlő rendszer összehangolását végzi. Az idővezérelt, valósidejű működésű buszok – mint a FlexRay – a biztonságkritikus Drive-by-Wire rendszereknél használtak. A multimédiás célú buszok a járművek szórakoztató rendszereihez szükséges nagy sávszélességet biztosítják a jó minőségű hang és videó lejátszáshoz. A vezeték nélküli hálózatokat pedig a V2I kommunikációhoz használják. Az egyes buszok között kommunikációs átjárók teremtik meg a kapcsolatot, így például diagnosztika során a jármű bármelyik ECU-jával kapcsolatba lehet lépni. Egy általános jármű buszrendszert mutat a következő ábra:



1.3. ábra: Egy modern autó fedélzeti hálózati rendszere [9]

Általánosságban elmondható, hogy a jelenlegi fedélzeti buszrendszerek hitelesítés és titkosítás nélküli adatkommunikációt végeznek. További problémát jelent, hogy a jelenlegi járművekben a komplex vezérlési rendszerek miatt több ECU-nak is egynél több buszra is csatlakoznia kell, amiknek következtében a jármű bármely két ECU-ja között kommunikációs kapcsolatot lehet teremteni.

1.3 Diagnosztikai kommunikáció áttekintése

A járművek elektronikai egységeinek diagnosztikai képességei viszonylag rövid múltra tekintenek vissza. A fedélzeti diagnosztikai szolgáltatásokat a motorszabályozó elektronika üzemének pontos részleteinek megismerése hívta életre, melynek elsődleges motivációja volt a jármű károsanyag-kibocsátásának nyomon követése. Ennek eredményeként jött létre az OBD, majd az OBD-II szabvány (ISO 15031). Ebben tíz féle üzemmód (szolgáltatás) van definiálva, melyek elsősorban a diagnosztikai hibakódok (DTC-k) kezelését szolgálják. A szabvány specifikál egy csatlakozótípust is, melyet a szabvány szerint a kormány fél méteres körzetében kell elhelyezni [10].

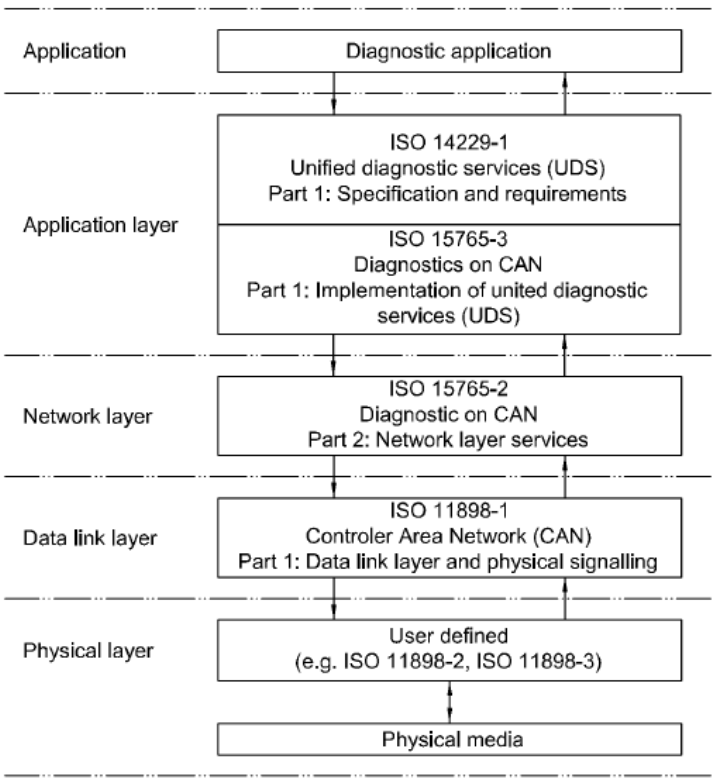
1.4. ábra: A jellemzően a kormánymű alá beszerelt OBD-II csatlakozó

Az OBD-II fedélzeti diagnosztikai szabvány 1996 óta kötelező része az Egyesült Államokban új autóknak, és 2003-ra az EU is kötelezővé tette. 2008 óta a szabvány előírja a CAN busz használatát (is). Az OBD funkcionalitását bővíti ki nagymértékben az UDS szabvány (ISO 14229), mely használatával a diagnosztikai kommunikáció kiegészül többek között tetszőleges memóriaadat fel- és letöltés, távoli rutinhívás, I/O vezérlés és kommunikáció menedzsment szolgáltatásokkal. Ezen diagnosztikai szabványok jelenlétében egy PC (egy átjáró segítségével) vagy egy diagnosztikai céleszköz képes csatlakozni az OBD csatlakozón keresztül a fedélzeti busz(ok)ra, melyen keresztül képes a buszon lévő ECU-kkal kommunikálni, továbbá a fedélzeti átjáróknak köszönhetően másodlagos buszokon lévő vezérlőkkel is kapcsolatba tud lépni.

Ezeket a diagnosztikai szabványokat többféle kommunikációs buszra illesztették, én a továbbiakban a CAN buszra épülő diagnosztikai kommunikációs stack-et fogom bemutatni. Egyrészt mivel ez a busz része az OBD-II szabványnak, másrészt pedig a relatíve egyszerű hardver és szoftver implementációja miatt. A diagnosztikai átjárót a PC-hez saját megvalósításban készítettem el, mert ez a megoldás jóval költséghatékonyabb és flexibilisebb, mint egy piacon elérhető kivitelezést beszerezni. Az átjáró egy USB virtuális soros porton keresztül teremti meg a PC-vel a kapcsolatot. Az adatok értelmezéséhez egy kliens alkalmazást is létre kellett hoznom, mely képes az átjárót konfigurálni, valamint képes megjeleníteni az átjáró által szolgáltatott adatokat. Ezen felül mind a kliensben, mind a tesztelés során használt ECU szerepet betöltő kontrollerekben implementálnom kellett a diagnosztikai szoftverréteg felett egy biztonságos adatátviteli réteget.

Mivel a CAN-re épülő diagnosztikai stack a cégnél már rendelkezésre állt, ezért csak nagy vonalakban vázolom az átvitel működését. A kommunikáció összesen öt

rétegből áll, ebből a CAN biztosítja a fizikai és az adatkapcsolati réteg működését, a hálózati rétegnek a CAN Transport Protocol (CAN TP) feleltethető meg, az alkalmazási réteg szerepe az UDS és OBD szabványokra esik, ezek felett pedig lesz a diagnosztikai biztonsági alkalmazásom.



1.5. ábra: CAN-ra épülő diagnosztika [11]

A CAN busz bemutatására remélhetőleg nincs szükség, a CAN TP (ISO 15765) szerepe és működése viszont megér néhány szót. Ennek a rétegnek a feladata egyrészt a legfeljebb nyolc bájtos CAN adatkeretek alapján maximum 4 kB-os csomagok átvitelének, másrészt több CAN busz közötti kommunikáció megteremtése.

A hálózati rétegben minden eszközhöz saját egyedi cím van rendelve, ezen felül még lehetséges logikai (csoport-) címeket is használni. A küldő és a cél eszköz címei a CAN azonosítóban kerülnek elhelyezésre, amennyiben kiterjesztett vagy távoli címzést alkalmaz a küldő, akkor az adatmező első bájtját is elfoglalja a címinformáció. Ettől függően a maradék 7 vagy 8 adatbájton osztozik a CAN TP protokollinformáció és a felsőbb réteg adatszelete. A továbbiakban az egyszerűség kedvéért csak az utóbbit, vagyis a normál címzés esetét tárgyalom.

A CAN TP szabvány négyféle keretet definiál. Amennyiben a küldendő csomag mérete kisebb, mint hét bájt, úgy a hálózati kommunikáció egyetlen *SingleFrame*

keretből áll, mely első bájtja a protokollinformáció, a többi a hasznos adat. Ennél nagyobb adatkeret esetén adatfolyam-szabályozásra van szükség a küldő és a címzett között. A küldő először egy *FirstFrame* keret küldésével jelzi a teljes átküldendő adat méretét, valamint az első hat bájtot. Ezután a címzett *FlowControl* keretet küld, melyben megadja a küldőnek a további keretek fogadási feltételeit (egymás után hányat küldhet újabb FC keret érkezése nélkül, mekkora késleltetéssel közöttük). A további adatszeleteket *ConsecutiveFrame* keretben küldi a küldő, melyben egy négy bites számláló szolgál a sorrendtartás biztosítására, és legfeljebb hét bájt hasznos adatot tud átvinni.

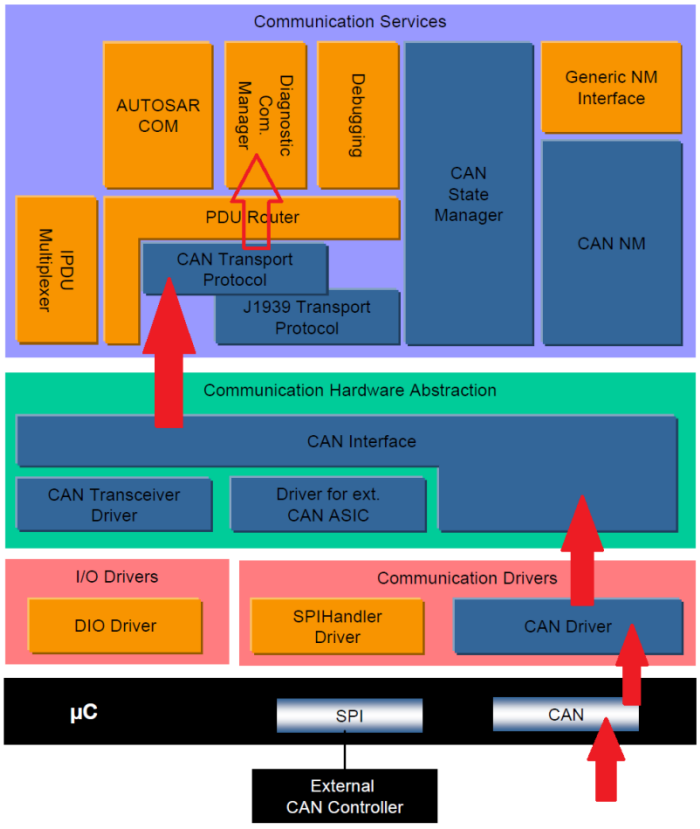
Az UDS és OBD diagnosztikai szabványok megegyező kommunikációs sémával rendelkeznek: a diagnosztikai kliens (más néven *tester*) a kezdeményező, aki a szerver ECU felé kéréseket küld, a szerver pedig ezekre ad választ. A nagyobb adattömbök átvitelére (például új ECU program feltöltésére) ezen a rétegen is a tömb feldarabolása és csomagonkénti átvitele a réteg által nyújtott megoldás.

1.4 Diagnosztikai stack az AUTOSAR architektúrában

Az ECU oldali biztonsági réteget az AUTOSAR architektúrájú diagnosztikai kommunikációs stack fölé kellett implementálnom. Az AUTOSAR egy olyan autóipari szabványkészlet, mely a komplex ECU szoftvereket elemi feladatú komponensekre, modulokra osztja, melyekre nagyon specifikus követelményrendszert szab. Az így létrejövő szoftverarchitektúrának az a célja, hogy rugalmasan skálázható és testreszabható, a modulok gyártótól, beszállítótól függetlenül összekapcsolhatóak, az alkalmazás szintű feladatok pedig teljesen függetleníthetőek legyenek a hardvertől.

Nekem az AUTOSAR alacsony szintű komponenseit felhasználva kellett egy tesztelési célú programot létrehoznom, melyben a biztonsági szoftverréteg képességeit kipróbálhattam. A 1.6. ábra mutatja az architektúrának az általam használt szeletét, melyen piros nyíllal jelöltem a CAN buszon érkező adat útját a diagnosztikai rétegig. A használt mikrokontroller rendelkezik belső CAN perifériával, ennek a szoftveres, hardvertől függő kezelésére szolgál a *CAN Driver* modul. Innen a *CAN Interface* modulhoz kerül a keret, mely komponens az alkalmazott hardvertől független absztrakciót nyújt a felsőbb szoftverrétegeknek. A keret továbbítódik a *CAN Transport Protocol* modulhoz, mely elvégzi a csomag összeállítását (és végzi az adatfolyam-szabályozást). A kész csomag a *PDU Router*-hez kerül, ami azonosító alapján képes

eldönteni, hogy az adott PDU-ra melyik felsőbb modulnak van szüksége (és ugyanezt a másik irányban is). Ha csak egy kommunikációs csatornát és egy kommunikációs szolgáltatást használunk, akkor ez a modul felesleges és kihagyásra kerül. Így a csomag a CAN TP-ből közvetlenül a *Diagnostic Communication Manager* modulhoz kerül.



1.6. ábra: CAN-re épülő diagnosztika az AUTOSAR architektúrában [12]

A DCM modul foglalja magában az UDS és OBD szabványok AUTOSAR szabvány szerinti részhalmazát. A diagnosztikai szolgáltatások egy részét önmaga képes lekezelni, más esetekben fix deklarációjú függvényeket kell definiálni, melyeket az adott szolgáltatások során meghív. A DCM részletesebb használatát a DCM feletti védelmi réteg megvalósítása során ismertetem.

2 Fedélzeti rendszerek biztonsági analízise

2.1 Terminológia

Mielőtt a témára térnék, szeretnék pár fontos, a továbbiakban gyakran használt fogalmat tisztázni, elsősorban a biztonság és védelem fogalmát. „A biztonság és védelem fogalmak a nemzetközi szakirodalomban leggyakrabban használt nyelv, az angol esetében több kifejezés formájában is előfordulnak. Ezek közé tartozik: a *'security'* (biztonság, valaminek a biztos jellege), a *'safety'* (biztonság, veszélytelenség), a *'protection'* (védelem, megvédelmezés), a *'defence'* (védelem, védekezés) és a jelen előadás témaköréhez kapcsolódóan legújabban az *'assurance'* (biztosítás, biztossá tétel).” [13]

Az autóipari nemzetközi szakirodalomban a *safety* fogalmat a megbízhatóság, funkcionális biztonság leírására használják, azaz hogy a rendszer mennyire képes a (véletlenszerűen bekövetkező) környezeti hibáktól függetlenül az elvárt működésre. Ez a megközelítés már régóta nagy jelentőségű az autóiparban, és igen komoly teljesítményeket értek már el ezen a téren, jelen munkában viszont a továbbiakban nem fog előkerülni. Ezzel szemben a *security*, mint a szándékos és jogosulatlan beavatkozások elleni védelem és biztonság lesz a következőkben az elsődleges értelmezése a védelem és biztonság fogalmaknak.

Munk Sándor szerint „[…] a leggyakoribb az az álláspont, amely - a köznapi nyelvhasználattal is összhangban - a biztonságot állapotnak, a védelmet tevékenység(rendszer)nek tartja. Ennek megfelelően a biztonság egy olyan állapot, amelyben valaki/valami a lehetséges fenyegető hatások ellen a megkívánt mértékben védett. A védelem pedig ebben az értelemben a fenyegetések elleni, a biztonság (mint megkívánt állapot) megteremtésére és fenntartására irányuló tevékenységek, rendszabályok összessége.”

2.2 A fedélzeti rendszer biztonsági alapmodellje

Annak érdekében, hogy megfelelő védelmi intézkedéseket tudjunk tenni a járművek fedélzeti rendszereiben, először azonosítanunk kell a védelmi rendszer szereplőit és tulajdonságaikat, ezt a Munk Sándor által alkotott biztonsági alapmodellen

keresztül fogjuk véghezvinni. Az alábbi ábra szemlélteti a biztonság alapmodelljét, melynek elemeit fogjuk a továbbiakban azonosítani:

2.1. ábra: A biztonság alapmodellje [13]

2.3 A védendő entitások

„A biztonság alapmodelljének elsődleges összetevője a biztonság alanya, a fenyegetések által veszélyeztetett objektum. […] Az adott objektum biztonságának értelmezéséhez meg kell határozni annak összetevőit. A biztonság összetevői, aspektusai a biztonság alanyának azon tulajdonságai (statikus és dinamikus állapotjellemzői), amelyeknek a megengedett mértéktől eltérő megváltozása a biztonság sérülését, megsértését jelenti.”

Egy gépjármű esetén az elsődlegesen veszélyeztetett objektumok a jármű viselkedését, működését befolyásoló vezérlőrendszerek és beavatkozószervek. Ezeknek az elemeknek a nem megfelelő működése a jármű feletti uralom elvesztését jelentheti a vezető számára, ami közvetlen kockázatot jelent az utasok és a közlekedés többi résztvevőjének testi épségében. A jármű rendszerei alábbi védendő tulajdonságokkal rendelkeznek:

- Az adott rendszer fizikai (mechanikai, hidraulikai) kialakítása.
- A rendszer elektronikai működési feltételei (a jármű üzeme alatt végig megfelelő tápellátással rendelkezik az összes elektromos alrendszer, nem éri a tervezettnél nagyobb mértékű EMI).

- A jármű viselkedését befolyásoló elektromos elosztott vezérlőrendszer rendeltetésszerű működése.

Természetesen ez a lista korántsem tekinthető teljesnek, de ez jelenleg nem is cél, sőt a továbbiakban a célirányosság érdekében leszűkítjük a vizsgálatunkat az utolsó pontra. Ez a tulajdonság bővebben kifejtve azt takarja, hogy a vezérlőrendszer építőkövei (az ECU-k) a helyes programot futtatják, valamint az összes szükséges kommunikáció ezen összetevők között akadálymentes.

2.4 Veszélyforrások

„A biztonság alapmodelljének második legfontosabb összetevőjét a biztonság alanyát (a védendő objektumot) veszélyeztető fenyegetések képezik. […] A fenyegetést jelentő kölcsönhatások végbemehetnek az adott objektum és környezete között, vagy érvényesülhetnek az objektumon belül, ennek megfelelően beszélhetünk külső, vagy belső fenyegetésekről.”

„A fenyegetések bekövetkezését a különböző sebezhetőségek teszik lehetővé. A sebezhetőség a biztonság alanyának egy olyan tulajdonsága, hiányossága, vagy gyengesége, amely lehetőséget teremt egy fenyegetést megvalósító [kölcsön]hatás érvényesülésére. A sebezhetőségek egy része a biztonság alanyának, vagy egyes összetevőinek természetes, a környezeti hatásokkal kapcsolatos tulajdonsága: ezekkel szembeni védtelensége, érzékenysége. Más sebezhetőségek – működő rendszerek, szervezetek esetében – a működési hiányosságok közé sorolhatóak.”

2.4.1 Belső fenyegetések

A belső fenyegetéseket célszerű a jármű, mint fő védendő entitás belsejében értelmezni, azaz amihez gyakorlatilag szükséges az autóhoz való hozzáférés (autókulcs) megléte, azt belső fenyegetésnek tekintem, minden egyéb, az autóval fizikai kapcsolatot nem igénylő fenyegetést távolinak tekintek.

Ha az ECU-val közvetlen fizikai kapcsolaton keresztül kerül egy entitás kölcsönhatásba, akkor lehetőség nyílik a mikrokontroller és a közvetlen perifériaegységek közötti jelek megfigyelésére és befolyásolására, továbbá JTAG vagy hasonló debug interfész segítségével a kontroller belső állapota is megismerhetővé és befolyásolhatóvá válik. (A mai kontrollerek már gyakran tartalmaznak olyan lehetőségeket, melyek a JTAG jellegű debug kapcsolatot képesek elérhetetlenné tenni,

megvédve így a kontroller belső memóriatartalmát és állapotát a nem kívánt hozzáférésektől.) Ezen felül felmerül annak a lehetősége is, hogy egyes hardver elemeket eltávolítanak vagy lecserélnek a nyomtatott huzalozású lemezről. Ez a típusú sebezhetőség gyakorlatilag nem szüntethető meg, és ilyen szintű hozzáféréssel a járműhöz gyakorlatilag bármi más fizikai paramétert meg lehetne változtatni rajta, így ezt a fenyegetést a továbbiakban nem tárgyaljuk.

A valódi fenyegetést az jelenti, hogy egyetlen kiszolgáltatott vagy feltört vezérlő a jármű szintű kommunikációs buszokkal való összekapcsoltság miatt (még ha közvetetten is) az egész jármű viselkedését befolyásolhatja. A Washington és a Kaliforniai San Diego Egyetem kutatói 2010-ben egy középkategóriás személyautó kísérleti sebezhetőségi analízise során a következőket tapasztalták [14]:

A jármű fedélzeti rendszeréhez való kapcsolódáshoz elég volt az OBD-II csatlakozóra csatlakoztatni egy rossz szándékú, CAN kommunikációval rendelkező eszközt, amit nem sokkal később el is lehetett távolítani a támadás gyors sikere miatt. A támadás a két CAN busszal rendelkező fedélzeti rendszer szabványos diagnosztikai és tesztelési célú funkcióin keresztül volt képes új programot feltölteni egy-egy vezérlőbe, bizalmas információt kiolvasni belőle, vagy közvetlenül befolyásolni az egyes beavatkozó szervek működését. A visszafejtett CAN kommunikáció alapján olyan kritikus műveleteket voltak képesek végrehajtani, mint például a motorvezérlő kontroller firmware frissítő módba átváltása járó motor és mozgó jármű mellett, vagy a fékek végleges kikapcsolása egy adott sebesség átlépésekor. A legtöbb ilyen beavatkozást ráadásul a vezetőnek nem állt módjában megszüntetni a kézi beavatkozásaival. A kutatók az alábbi fenyegetésekre hívják fel a figyelmet:

1. A CAN busz nem alkalmas biztonságos kommunikációként szolgálni. A buszra folyamatosan küldött nullás legmagasabb prioritású keret a busz kommunikációját teljes mértékben képes megszüntetni. További probléma, hogy a keretek broadcast jellegűek, nem lehet beazonosítani a feladót, így egyrészt a buszhoz egy ponton hozzáférve az azon elérhető összes adat hozzáférhetővé válik, másrészt bármelyik ECU-t meg lehet személyesíteni és a nevében keretet küldeni.
2. Az ECU-kban használt diagnosztikai alkalmazás hitelesítésre szolgáló kihívás-válasz protokollok megvalósításai elégtelenek a céljukra. A protokollok ugyanis fix 16 bites választ várnak, melyet *brute-force*

támadással 7-8 nap alatt (10 s várakozási idő van két próba között) fel lehet törni az összes ECU-ra.

3. A szoftver implementációk nem felelnek meg a szabványban előírt követelményeknek. Többek között lehetőség volt az egyes ECU-k CAN perifériájának kikapcsolására vagy újraprogramozási módba váltására menet közben.

Mindezen felül még az is problémát jelent, hogy sok parancs esetén a vezető azzal ellentétes manuális beavatkozásait a rendszer nem veszi figyelembe, így tehetetlenné téve a vezetőt. A kutatók egy olyan forgatókönyvet is megvalósítottak, melyben az ütközés észlelése után a rosszindulatú program eltávolítaná magát a kontrollerről. Az ilyen jellegű támadások – habár az általuk okozható károk jelentősek – összességében nem tekinthetők a legveszélyesebbnek, figyelembe véve, hogy feltételezik a támadó előzetes fizikai hozzáférését a járműhez, aminek fennállása esetén nem csak a fedélzeti rendszerben tehet kárt, hanem mechanikai kárt is okozhat, amely ugyancsak nagymértékű károkhoz vezethet. Továbbá ilyen esetben felmerül a tulajdonos felelőssége, hiszen ilyen támadáshoz (jellemzően, de nem kizárólagosan) az autó belsejében kell jelen lenni, ami csak kulcs birtokában lehetséges.

Más a helyzet, amennyiben egy multimédiás eszközt csatlakoztatunk a járműre (legyen az egy CD lemez, USB PenDrive, vagy okostelefon). Ezek az eszközök a tulajdonos tudta nélkül képesek kártékony kódot tartalmazni, mely a járműre csatlakozás során ugyanúgy képes befolyása alá venni más ECU-kat, mint az OBD-re csatlakoztatott rendszer, kezdve a rádió vezérlőjével. Teszi mindezt annak kihasználásával, hogy a kényelmesebb szoftverfrissítés érdekében többféle eljárással is frissíteni lehet a rádiók firmware-ét.

2.4.2 Külső fenyegetések

A korábban már idézett Washington és a Kaliforniai San Diego Egyetem kutatói 2011-ben elvégeztek egy újabb kísérletsorozatot, mely során már az indirekt és külső fenyegetéseket mérték fel egy modern személyautóban [15]. Az eredményeik nem sokban különböztek az előző évitől, minden rendelkezésre álló vezeték nélküli kommunikációs csatornán elindulva képesek voltak a jármű feletti irányítást megszerezni. A következő felsorolás ismerteti mindazokat a vezeték nélküli hálózatokat, melyeken keresztül be lehet hatolni a jármű belső fedélzeti rendszerébe:

- A vezetés közbeni telefonálást Bluetooth kihangosítók teszik lehetővé a legtöbb mai autóban.
- Az autók zárjait és riasztóját már hosszú évek óta rádiófrekvenciás távadóval is lehet vezérelni.
- A hagyományos autókulcs nem csak mechanikailag azonosítja a modern autókat, hanem egy beépített RFID eszköz is szerepet játszik az azonosítás során, és hibás azonosító esetén letiltja az autó elindítását.
- GPS navigációval vagy műholdas rádióvevővel is egyre több kocsit látnak el, ezen keresztül egy esetleges lopás esetén távolról lehet motorleállítási és zárrendszer oldási parancsokat küldeni.
- A sok autógyártó által használt távoli telemetriai rendszerek (ilyenek például a Ford Sync, a Toyota SafetyConnect, a BMW Assist, vagy a Mercedes-Benz mbrace rendszere) a mobilhálózaton keresztül állnak összeköttetésben az adatközpontokkal. Ezek a rendszerek többek között balesetekről, technikai hibákról küldenek értesítéseket, de lopás esetén nyomkövetésre és jármű letiltásra is képesek.

A kutatócsoport eredményei azt mutatják, hogy az autóipari tervezés során egyszerűen nincs jelen a biztonsági aspektus, mint tervezési szempont. A másik nagy probléma az autóipari szoftverrendszer kialakulásában van. Ezt ugyanis az autógyártók egyre gyakrabban szervezik ki beszállítóknak megvalósítási célzattal a megadott specifikáció alapján. A beszállító viszont csak a modult valósítja meg, és modul szintű teszteket végez, és csak a lefordított fájlokat adja át a gyártónak. Az pedig forráskód híján nehezen tudja az integrációs tesztek során kiértékelni a rendszer védelmi hiányosságait.

Persze ezek után jogosan merül fel a kérdés, hogy miért nem hall az ember nap, mint nap autófeltörésről híreket. A választ a „bizonytalanságon alapuló biztonság” [16] adja, ami egy tudásszakadékot jelent a lehetséges támadók és a jármű között azáltal, hogy a jármű egy komplex és teljesen ismeretlen belső működésű rendszer, így csak kitartó és hozzáértő munka után lehet feltérképezni a működését.

2.4.3 Kormányrendszerekre érvényes támadási lehetőségek

Az autók kormányrendszerének vezérlőegysége esetén a legtöbb járműben – az utasok fizikai hozzáférésének hiánya miatt – elsősorban nem a közvetlen, hanem a fedélzeti hálózatokon (CAN, FlexRay) keresztüli támadásokkal kell számolni. Mint az mostanra kiderült, egy általános ECU legnagyobb támadási felülete a diagnosztikai interfésze, és ez az állítás a kormányrendszerek esetén is megállja a helyét.

Vannak azonban egyéb jelentős károkozásra alkalmas támadási lehetőségek, hogyha az adott jármű rendelkezik olyan funkcióval, amely részben vagy teljesen az elektronika kezébe adja a kormányzás feladatát (például egy intelligens parkolási asszisztens). Ezek a funkciók tipikusan elosztott vezérléssel valósulnak meg, akár több tucat szenzor, mikrokontroller és beavatkozó összehangolásával. Ha az ezen elemeket összekapcsoló hálózatokat egy támadó a megfelelő üzenetekkel befolyásolja, az adott funkcióhoz hasonló beavatkozási hatalma lesz a jármű komponensek felett (a parkolási példánál maradva a kormányvezérlő hálózatán küldött szabotált üzenetek elhitetik a vezérlővel, hogy adott parkolási manővert kell végezni, vagyis a kormányszöget képesek befolyásolni). Ráadásul az elektromechanikus kormányszervók nyomatéka jelentősen meghaladja azt az emberi erőt, amely ellen tudna tartani a vezérelt mozgásnak.

2.2. ábra: Az első sorozatgyártott Steer-by-Wire kormányrendszer [17]

Összességében kijelenthető, hogy kormányvezérlők esetén az elsődleges sebezhetőségeket a hálózati kapcsolaton alapuló funkciók jelentik, tehát a kommunikáció biztonságossá tétele a legfontosabb feladat. Ennek viszont része kell

legyen az is, hogy a hálózatot használó felek képesek legyenek a hiteles működésükről bizonyítékot szolgáltatni.

2.5 Lehetséges támadók

„A biztonság alapmodelljének harmadik összetevőjét a veszélyeztetés forrásai, a biztonság alanyát veszélyeztető kölcsönhatásokban érintett objektumok képezik. A veszélyforrás az az objektum, amely a biztonság alanyát veszélyeztető hatást közvetlenül, vagy közvetve (áttételesen) kiváltja, vagy a veszélyeztető kölcsönhatásban érintett. Veszélyforrások lehetnek a természeti és az épített környezet (ezek egyes összetevői); technikai eszközök; vagy emberek, csoportok, szervezetek, stb. Amennyiben a veszélyeztető hatást a veszélyforrás egy másik - természeti, mesterséges, vagy társadalmi - 'eszköz' közvetítésével, felhasználásával váltja ki, beszélhetünk a veszélyeztetés alanyáról, eszközéről és tárgyáról (a biztonság alanyáról).

Tudatos szereplők esetében megkülönböztethetőek a gondatlan (véletlenül, nem akarattal előidézett) és a szándékos veszélyeztetések. A biztonságot szándékosan veszélyeztető fenyegetéseket a [biztonság elleni] támadásnak nevezzük. A biztonság elleni támadás lehet aktív, amikor a támadó - anyagi, információs, vagy szellemi - hatást gyakorol a biztonság alanyára és lehet passzív, amikor a támadó csak a biztonság alanyának, kibocsátott hatásainak, interakcióinak érzékelésére, megfigyelésére és felhasználására alapozva sérti meg annak biztonságát.”

Az már jól ismert tény, hogy a járművek megbízható működésére évtizedek óta nagy hangsúlyt fektetnek. Így a veszélyeztetés elsődleges forrásainak a szándékos támadásokat kell tekintenünk, de ehhez érdemes megvizsgálni, milyen motivációk bújhatnak meg egy lehetséges járműrendszert szabotáló személy mögött. Az eddigi kutatási projektek során a következő célokat sikerült azonosítani [18]:

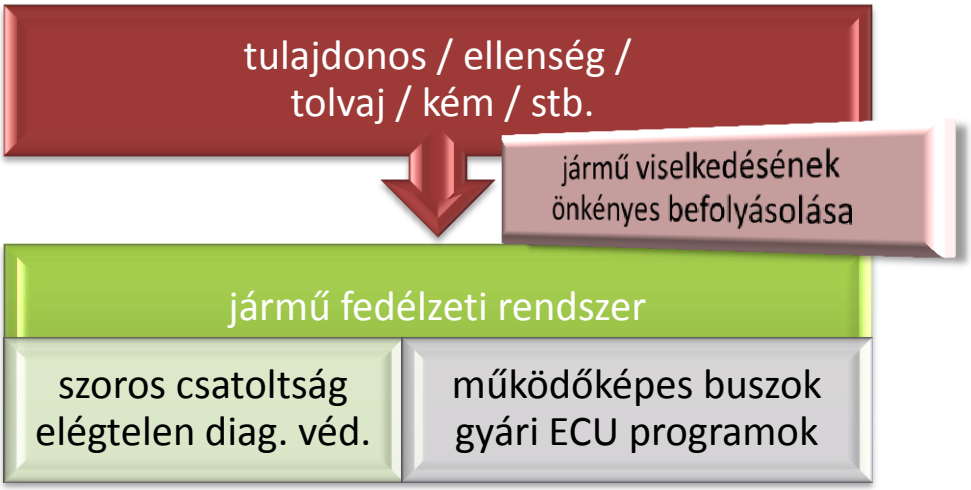
- Kártétel a járműben, az utasok testi épségében (baleset előidézése)
- Jármű vagy komponens használatának ellehetetlenítése
- A gépjármű vagy egy értékes összetevőjének eltulajdonítása
- Egyes kikapcsolt vagy korlátozott hardver vagy szoftver funkciók feltörése (például sebességplafon módosítás, extra szolgáltatás aktiválás, jogosulatlan szoftverfrissítés)

- Információszerzés a vezetőről (mobiltelefon adathozzáférés, tartózkodási hely vagy útvonalak megfigyelése)
- Megszemélyesítés (elektronikus rendszám segítségével)
- Járműkomponensek pénzügyi, jogi vagy jótállási célú befolyásolása (elektronikus díjszedő rendszerek, digitális tachográf manipuláció)
- Gyártók vagy beszállítók szellemi tulajdonának megszerzése, megismerése (ipari kémkedés, hamisítványok gyártása)
- Fedélzeti vagy külső kommunikáció megzavarása, tönkretétele
- Gyártó vagy beszállító hírnevének rontása (helytelen működés előidézésével)
- Baleset esetén a tárolt adatok módosítása a felelősség elhárítása céljából
- Hírnév szerzése a sikeres támadások által

E motivációk egy igen széles kört le tudnak fedni, ráadásul nagyban eltérő járműhöz férési lehetőségekkel, technikai tudással valamint anyagi és technikai háttérrel. Emellett mindenképpen hangsúlyozni kell, hogy ha a rendszer befolyásolása sikerrel jár, az potenciálisan életveszélyes szituációt hoz létre az autó utasai és a környező járművek számára, akár szándékában állt a támadónak, akár nem. Éppen ezért a kialakítandó védelmi rendszernek átfogó megoldást kell nyújtania, amely minden szintű támadás ellenére képes a rendszer biztonságát megőrizni.

2.6 Fedélzeti rendszer biztonsági modellje

A fejezetben megismertek alapján most már képesek vagyunk megalkotni a saját biztonsági modellünket. Az egyre jobban automatizált működésű modern gépjárművek elosztott fedélzeti vezérlőrendszerének működési feltételeit kell biztosítanunk. Láttuk, hogy az autóiparban elterjedt buszok nem alkalmasak arra, hogy egy potenciális támadó eszköz jelenlétében megbízhatóan működjenek. Az is kiderült, hogy a jelenlegi ECU-k nincsenek felkészítve arra, hogy ellenálljanak a szándékosan hibás inputoknak. A támadó személyét a motivációk sokfélesége miatt nem lehet igazán körülhatárolni, viszont minden esetben a jármű viselkedésének a gyáritól való megváltoztatása által éri el a célját.

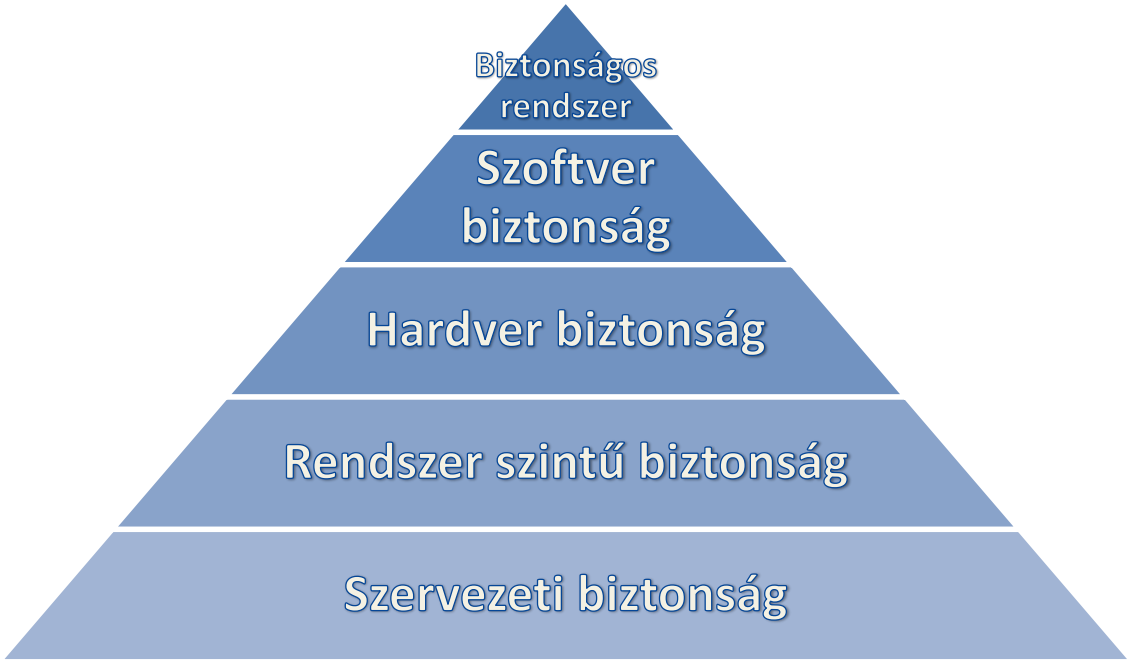


2.3. ábra: A fedélzeti rendszer biztonsági modellje

A következő fejezetekben ismertetem azokat a védelmi eljárásokat és alkalmazásokat, mely ismeretek birtokában megalkottam egy autóipari beágyazott rendszer diagnosztikai kommunikációjának védelmi rétegét. Ez a védelmi réteg lehetővé teszi, hogy az ECU ellenálljon a nem hitelesített forrásból származó diagnosztikai memória-átviteli kéréseknek és csak hiteles forrásból származó új firmware-t enged feltölteni.

3 Beágyazott rendszerek védelme

A fedélzeti számítógép-rendszer a biztonsági modell felállítása után azt kell megvizsgálnunk, hogy milyen védelmi rendszert kell felépítenünk ahhoz, hogy a megismert sebezhetőségeket eltávolítsuk a rendszerből. Ehhez azonosítani kell a szükséges fizikai, algoritmikus és rendszabályi biztonsági feltételeket, valamint ezek kapcsolatait. A következő ábra szolgál útmutatóul a biztonságos rendszer felépítésére:



3.1. ábra: A biztonságos rendszerhez szükséges biztonsági feltételek [19]

Mint az látható, az egyes biztonsági szintek egymásra épülnek, azaz egy felsőbb szintű védelmi rendszer mindenképpen kiszolgáltatott lesz az alsóbb szinteken meglévő sebezhetőségeknek. Tehát a védelmi rendszert alulról felfelé minden szint biztonságának garantálásával kell felépíteni.

3.1 Szervezeti biztonsági követelmények

Szervezeti biztonság alatt jellemzően azt értjük, hogy a bizalmas információk birtokában lévő szervezetek – az autógyártók, a beszállítóik – mélyreható belső rendszabályi védelemmel rendelkeznek, melyek csak a jogosult személyeknek adnak hozzáférést az érzékeny adatokhoz, és nincsenek fenyegetve a potenciális pszichológiai manipuláción (social engineering [20]) alapuló támadásokkal. A Verizon 2013-as

felmérése alapján az ilyen módszereket (is) használó támadások tették ki az összes szervezet elleni támadás 29 %-át [21], tehát ez egy korántsem elhanyagolható biztonsági szempont.

3.2 Rendszer szintű biztonság – kriptográfia

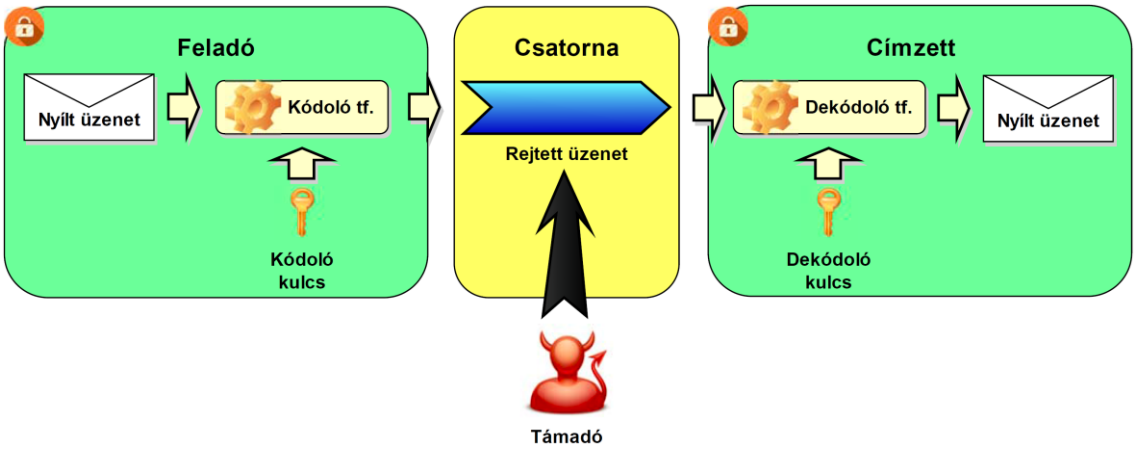
A kriptográfia a biztonságos információközvetítés és –tárolás megvalósításával foglalkozó tudományterület. Célja, hogy a védendő információn olyan transzformációt végezzen, mely képes az információ bizalmasságát, (forrásának) hitelességét, valamint sértetlenségét garantálni. Ezt a modern kriptográfia speciális matematikai algoritmusok segítségével éri el. Mivel ez a tudomány számomra azelőtt ismeretlen volt, ezért [22] alapján elsajátítottam a szükséges kriptográfiai alapismereteket.

A kriptográfiában az információhoz való jogosultságot egy bitsorozat biztosítja, melyet kulcsnak neveznek. A kriptográfiai algoritmusok működése arra épül, hogy a kulcs birtokában egyértelmű kimenetet tudjanak produkálni, melyeket a kulcs nélkül nem lehet meghamisítani, valamint abból a bemenetet visszafejteni. A kriptográfiai kulcsoknak két típusa van a kulcsot felhasználó algoritmustól függően: szimmetrikus kulcsú rejtjelező esetén egyazon titkos kulcs szükséges a kódolás és dekódolás műveletéhez, aszimmetrikus (vagy nyílt) kulcsú rejtjelező esetén egy nyilvános kulcs szükséges a kódolás, és egy (hozzá tartozó) titkos kulcs kell a dekódolás műveletéhez. Nyílt-titkos kulcspár mindig úgy jön létre, hogy egy megfelelő követelményeknek eleget tevő számon (ami a titkos kulcs lesz) egy olyan matematikai műveletet hajtunk végre, mely végeredményéből (a nyilvános kulcsból) visszaszámolni a titkos kulcsot praktikusan lehetetlen (hatalmas számítási kapacitást igényel).

A kriptográfia által nyújtott két legfontosabb szolgáltatás az információ titkosságának és hitelességének biztosítása. Az előbbire a rejtjelezés szolgál, utóbbira pedig az üzenethitelesítés (MAC) és az aláírás, ezen felül léteznek olyan kombinált eljárások, melyek mindkét igényt megvalósítják.

A rejtjelezés során a feladó egy nyílt szöveget (*plaintext*) szeretne eljuttatni a fogadónak. Hogy az üzenet az útja során értelmezhetetlen legyen mások számára, a nyílt szöveget egy rejtjelezővel és a kulcsával kódolja (*encrypt*), így létrejön a rejtjeles szöveg (*ciphertext*). Ezt megkapja a fogadó, és a kulcsával a rejtjelezővel dekódolja azt, visszakapva a nyílt szöveget. Szimmetrikus rejtjelezés esetén a küldő és a fogadó kulcsa meg kell, hogy egyezzen, aszimmetrikus rejtjelezés esetén a fogadó a saját titkos kulcsát

egyedül birtokolja, míg a nyilvános kulcsát bárki használhatja arra, hogy neki titkosított üzeneteket küldjön, melyeket a fogadó képes egyedül dekódolni.



3.2. ábra: A rejtjelezés klasszikus modellje

Az üzenethitelesítés (*message authentication*) során a feladó a (szimmetrikus) kulcsa felhasználásával egy kriptográfiai ellenőrzőösszeget számít, melyet hozzáfűz a nyílt szöveghez. A fogadó a megegyező kulcsa segítségével ugyanazt a számítást elvégzi, és ha az eredmény megegyezik a küldött értékkel, akkor elfogadja az üzenet hitelességét, különben elveti. Üzenet aláírásakor a feladó a titkos kulcsával és egy aszimmetrikus rejtjelezővel számít egy aláírást, melyet a fogadó(k) ellenőrizni tud(nak) a nyilvános kulcs felhasználásával.

3.2.1 Szimmetrikus kulcsú rejtjelező – AES

Az Advanced Encryption Standard a NIST négy éves versenye volt egy új szimmetrikus rejtjelező szabvány előterjesztésére, majd 2001-ben a Rijndael blokkrejtjelező egyes verziói lettek elfogadva a verseny lezárásaként és a szabvány publikálásaként. Az AES 128 bites blokkok rejtjelezésére képes, 128, 192 vagy 256 bites kulcsmérettel. Nem csak kiemelkedő biztonságú (mindeddig feltörhetetlennek bizonyult), de kevés műveleti memóriát igényel és relatíve gyors algoritmus [23]. Hardver megvalósításra is alkalmas, 2008-ban az Intel kifejlesztette az AES-NI (New Instructions) x86 utasításkészlet-kiegészítést, mely az Intel és AMD processzorokban felgyorsítja az AES kódolást és dekódolást.

3.2.2 Aszimmetrikus kulcsú rejtjelezés - ECC

A nyílt kulcsú rejtjelezők skálája (a szimmetrikussal szemben) igen szűkre szabott. Az RSA volt az első ilyen rejtjelező, és még ma is a legelterjedtebbnek számít.

Viszont az RSA egy nagy hátránnyal szenved, ez pedig a túl nagy kulcsméret igénye a megfelelő szintű biztonsághoz [24]. Ezért érdemesebb az újabban felfedezett elliptikus görbékre épülő kriptográfiát alkalmazni, mely esetén a szükséges kulcs mérete töredéke annak, amit egy ugyanolyan biztonságú RSA-hoz szükséges. A véletlenszám-alapú elliptikus görbéket leíró egyenlet az alábbi:

$$E: y^2 \equiv x^3 - 3x + b \pmod{p}$$

Ahol p a prím modulus, valamint x és y a görbe koordinátái, b pedig egy rögzített együttható (minden egész számok felett értelmezett). Az elliptikus görbe pontjain értelmezett két pont összeadása, valamint pont skalárral való szorzása (mindkettő a görbe egy pontját adja eredményül). A kriptográfiai használatát ez utóbbi művelet adja, ugyanis a két pont ismeretében a diszkrét logaritmusképzés elliptikus görbékre feladatot kellene megoldani, melynek megoldására csak exponenciális algoritmus ismert. Ezek alapján a titkos kulcs szerepét a skalár szám veszi fel, a nyilvános kulcs szerepét pedig a szám szorzata a bázisponttal (ami szintén egy rögzített görbe paraméter).

Az elliptikus görbéket nem csak titkosításra, hanem kulcscsere protokollra és aláírási algoritmusra is lehet használni. Előbbit elliptikus görbe Diffie-Hellman (ECDH) protokollnak hívják, és a lényege, hogy ha két fél szeretne egy közös titkot létrehozni, akkor mindketten létrehoznak egy kulcspárt, a publikus kulcsot kicserélik egymás között, és összeszorozzák a saját titkos kulcsukat a másik publikus kulcsával. Ekkor mindketten ugyanazt a görbe pontot kapják végeredményül a görbe ponton értelmezett szorzás asszociatív volta miatt. Az aláírási séma (ECDSA) már egy összetettebb protokoll, így azt már nem fejtem ki részletekbe menően, a koncepció lényege, hogy a titkos kulccsal egy a rejtjelező blokkméreténél kisebb számból létrehoz két új értéket, melyeket az aláírást ellenőrző fél a nyilvános kulcs és a kiindulási szám segítségével képes ellenőrizni [25].

3.2.3 Blokkrejtjelezési módok

A blokkrejtjelezők – ilyen az AES és az ECC is – hátránya, hogy csak a rögzített blokkméretüknek megfelelő hosszú üzeneteket lehet velük rejtjelezni. Természetesen egymás után több blokknyi adat feldolgozására is van lehetőség (ezt hívják ECB módnak), ám mivel ilyenkor a megegyező tartalmú nyílt szövegblokkok megegyező rejtett blokkokká képeződnek, amiből adott adatmennyiség után könnyű következtetni a

nyílt blokkokra, ezért ezt a módot a gyakorlatban nem használják. Az elterjedt blokkrejtjelezési módok mind egy kezdeti változóval (IV) indítják a rejtjelezést, mely nyilvánosan átadható a felek között, de azonos kulcs esetén eltérő értékeket kell használni.

Az első gyakorlatban is használt mód a CBC. Ez a mód a kódolás során az aktuális nyílt blokkot XOR-olja az előző rejtett blokkal (az IV a 0. rejtett blokk), majd az eredményt titkosítva hozza létre az aktuális rejtett blokkot. A dekódolás során pedig az aktuális rejtett blokk dekódolása után XOR-olja azt az előző rejtett blokkal, és így kapja meg a nyílt blokkot.

A következő mód, a CFB egy adatfolyam-rejtjelezővé alakítja a blokkrejtjelezőt. Az aktuális rejtett adatmennyiség az aktuális bejövő adatmennyiség XOR-olva a belső shift regiszteren alkalmazott kódolás kimenetének egyező hosszúságú részével. Dekódolás során az aktuális nyílt adatmennyiség az aktuális rejtett adatmennyiség XOR-olva a belső shift regiszteren alkalmazott kódolás kimenetének egyező hosszúságú részével. A shift regiszterbe – ami kezdetben az IV-vel van feltöltve – pedig mindkét irányban betolódik az új rejtett adatmennyiség.

Az OFB mód annyiban tér el a CFB-től, hogy a shift regiszterbe a rajta alkalmazott kódolás vagy dekódolás kimenetének meghatározott hosszú része tolódik be. Végül a számláló (CTR) mód a rejtjelezőbe bemenő adatot XOR-olja egy számláló kódolásának kimenetének egyező hosszúságú részével, mind kódolási, mind dekódolási oldalon. A számlálót minden blokkméretnyi bájt feldolgozása után inkrementálja.

3.2.4 Hash függvények

A hash függvények olyan transzformációk, melyek tetszőleges hosszú bináris sorozatot rögzített hosszúságú hash értékbe (lenyomatba) képeznek. A kriptográfiai felhasználásra alkalmas hash függvényekkel szemben a következő elvárásokat állítjuk:

- A hash függvény legyen egyirányú, azaz a bitsorozatból egyszerű feladat legyen a lenyomat kiszámítása, a lenyomatból viszont legyen nehéz megfejteni a bitsorozatból.
- A hash függvény legyen ütközés-ellenálló, azaz nehezen lehessen két különböző bitsorozatot találni, melyek egyező lenyomatot produkálnak.

Jelenleg a NIST által szabványosított SHA kriptográfiai hash család a legszélesebb körben használt protokollcsalád. Habár 2005-ben az SHA-1-ben már sikerült ütközéseket találni, így ez a verzió csak bizonyos feltételek mellett alkalmazható, az SHA-2 és SHA-3 implementációkban még nem sikerült ilyen támadást véghezvinni, így ezek mindenképpen biztonságosnak tekinthetők.

3.2.5 Üzenethitelesítés - MAC

Titkos kulcsú felek esetén egy üzenet eredetét, valamint tartalmának sértetlenségét egy MAC algoritmussal lehet igazolni, mely egy fix hosszú *tag*-et hoz létre a tetszőleges hosszú üzenet és a titkos kulcs alapján. A fogadó a *tag*-et újraszámolva az üzenetre, majd azt összevetve a kapott *tag*-gel el tudja dönteni, hogy az adott üzenetet figyelembe vegye-e. Jelenleg kétféle kriptográfiai elemre épített biztonságos MAC algoritmus terjedt el, az első a hash alapú MAC (HMAC), a második a blokkrejtjelező alapú MAC (CMAC). Előbbi tetszőleges kulcsméretet fel tud dolgozni, utóbbi csak a rejtjelező által elfogadott méreteket.

3.2.6 Hitelesített rejtjelezés

A hitelesített rejtjelezés célja, hogy a hitelesítés által nyújtott hitelességet és sértetlenséget, valamint a rejtjelezés által nyújtott bizalmasságot egyszerre képes legyen szolgáltatni. A hitelesített rejtjelezés háromféleképpen valósulhat meg:

1. Titkosítás utáni MAC, mely a rejtett szövegre számítja a hitelesítő kódot. Ez a mód a három közül a legbiztonságosabb.
2. Titkosítás és MAC, mely során a két művelet külön-külön fut le a nyílt szövegen.
3. MAC utáni titkosítás, mely során a nyílt szöveg és az azt hitelesítő kód is titkosításra kerül.

Erre a célra alkalmas egy blokkrejtjelező és egy MAC algoritmus kombinált alkalmazása, de léteznek integrált sémák is a feladatra, például a CCM mód [26], mely CBC-MAC-et kombinálja a számláló rejtjelező móddal (3-as típus). Egy másik elterjedt algoritmus a GCM, mely már egy sokkal összetettebb algoritmus segítségével képes jobb teljesítményre (1-es típus).

3.2.7 Kulcsméret választás

Az egyik legfontosabb biztonsági szempont a használt kulcs mérete, mely minél nagyobb, annál nehezebb úgynevezett *brute-force* támadással végigpróbálni a kulcsteret a keresett kulcsérték után. Azonban a kulcsméret növelésével a kriptográfiai algoritmusok számítási igénye is megnő, tehát végeredményben a kulcsméret mindig egy kompromisszum eredménye az elérhető biztonsági szint és a kriptográfiai eljárások sebessége között. A különböző kriptográfiai szabványosítási testületek és hivatalok által tett javaslatok és összevetésük itt tekinthető meg [24]. Az itteni adatok alapján 2030-ig biztonságosnak tekinthető:

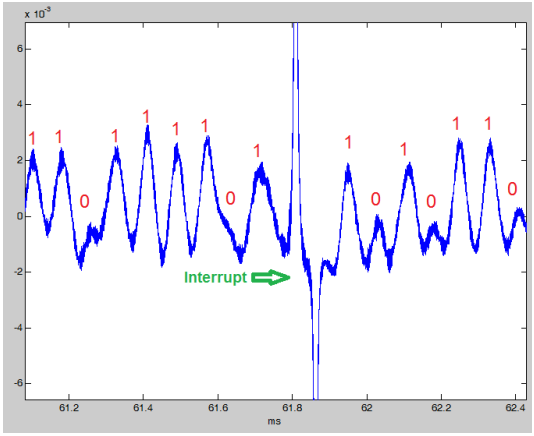
- a 128 bites szimmetrikus (AES) kulcs
- a 256 bites lenyomatú hash függvény
- a 256 bites elliptikus görbe kulcs

Elliptikus görbék esetén összetettebb feladat a biztonságosság megítélése, mint a többi kriptográfiai eljárás esetén. Itt ugyanis nem csak a kulcs mérete számít – sőt az még nem is elsődleges – hanem a görbét leíró paraméterek. Pietrosanti összefoglalása alapján [27] a jelenleg használt elliptikus görbék közül kettőt választott alkalmasnak több független testület is (ANSI, NSA, SAG, NIST, ECC BrainPool), használt több biztonsági protokollban és szerepel az NSA Suite B listán is, ezek pedig a NIST P-256 és P-384 elliptikus görbék. Paramétereik a NIST DSS szabványának D függelékében találhatóak [28].

3.3 Biztonságos hardver platform

A kriptográfiai védelem csak elméletben képes garantálni a biztonságot. Ha a támadó képes kinyerni jeleket a mikrokontroller működése során annak környezetéből, vagy belsejéből, akkor képes lehet megfejteni a kulcsot vagy a nyílt üzenetet. Ez többféleképpen is bekövetkezhet, például ha a kulcsot egy külső memóriában tároljuk, vagy ha a kontroller JTAG interfésze üzemképes. De ennél sokkal kevésbé egyértelmű információ is elegendő a feltöréshez. A kriptográfiai programok ugyanis az algoritmus bizonyos pontjain eltérő műveleteket végeznek attól függően, hogy az éppen feldolgozott bit 0 vagy 1. Ezek az eltérések mérhetőek a kontroller által felvett áram megfigyelésével. A jelalak alapján kétféle úgynevezett oldalcsatornás támadást (*Side-Channel Attack*) lehet indítani:

1. Amennyiben két eltérő periódusidejű jel véletlenszerű váltakozását lehet megfigyelni, akkor az időeltérés alapú oldalcsatornás támadásra ad lehetőséget, ahol az egyik periódusidőhöz 0, a másikhoz 1 bit tartozik.
2. Ha egyező periódusidővel, de eltérő amplitúdóval érkező jeleket mérünk, akkor az áramfelvétel alapú oldalcsatornás támadást jelenthet (lásd ábra).



3.3. ábra: Áramfelvétel alapú oldalcsatornás támadás

A biztonságos hardver külön, a rendszer többi részétől elzárt tárhelyet kell biztosítson az érzékeny adatoknak (kriptográfiai kulcsok, eseménynaplók). A külön memóriaterület nem ér semmit egy hozzárendelt kriptográfiai elemkészlet nélkül, ugyanis más esetben a kulcsot ki kellene hozni a biztonságos tárolási helyéről, hogy a kriptográfiai művelethez felhasználható legyen. A hardveres védelemmel szemben támasztott másik fontos követelmény a kontrolleren lévő program hitelesítésének feladata, melyet tipikusan a bootolás során tesz meg. Fontos kriptográfiai szerepe van a véletlenszám-generátornak, amely például kihívás-válasz protokollok során küld nonce seed-et, mellyel a visszajátszásos támadások esélyét nagyban megnehezíti. Továbbá azt se felejtsük el, hogy a biztonságos hardvertől elvárjuk, hogy szigorúan a mikrokontroller integrált áramkörén belül kell legyen, lehallgatás-biztos (*tamper-resistant*) tokozással.

3.4 Szoftver implementáció biztonsága

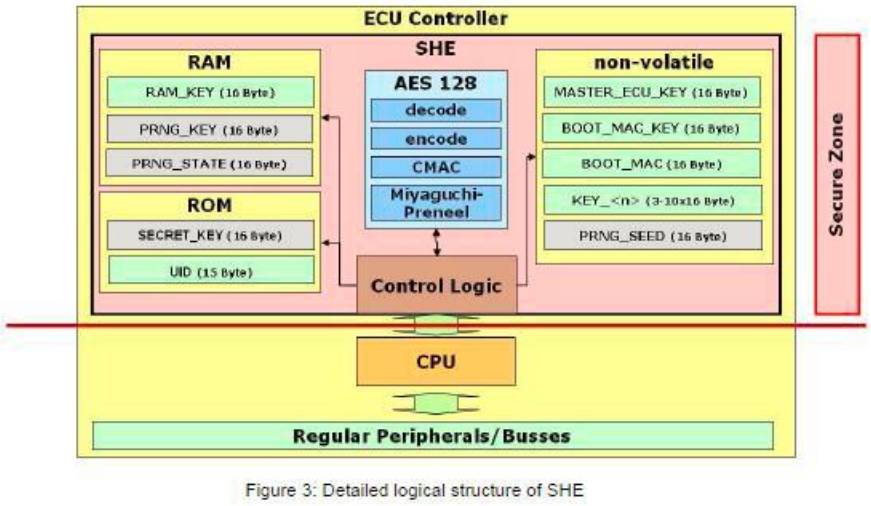
Amennyiben rendelkezésünkre állnak a korábban felvázolt biztonsági komponensek, a következő biztonsági követelmény a vezérlőn futó programok biztonságos, sebezhetőségektől mentes implementációja. Ez egy igen összetett feladat, és teljes mélységében nehéz ismertetni, de a kiemelendő az, hogy a szoftvertervezés során különös figyelmet kell szentelni a nem várt események biztonságos kezelésére. Az ilyen – véletlenül vagy szándékosan – hibás bemenetek elfogadása könnyen képes kárt okozni a futó kódban. Ezen felül az alkalmazott szabványok által megfogalmazott követelményeket mindenképpen meg kell valósítani, és meg kell győződni az elvárt

működéséről. További elvárás lehet az, hogy a futó program hitelességét az azt elindító bootloader program ellenőrizze. Ilyen esetben a hardvernek kell képesnek lennie a boot kód hitelesítésére, és blokkolására, ha nem érvényes az adott kód.

4 Eddig elért kutatási eredmények

4.1 HIS Secure Hardware Extension

Az első autóipari kutatási projektet, mely biztonságos autóipari beágyazott hardver megoldást kutatott, a Hersteller Initiative Software (HIS) – német autógyártók szövetsége – indította el 2009-ben. Az általuk javasolt hardver modul a minimális szükséges szolgáltatásokat nyújtja:



4.1. ábra: A SHE modul felépítése [29]

A kriptográfiai magban a 128 bites kulcsú AES található meg, melyhez a rejtjelezésen kívül hozzáadták a CMAC algoritmust, valamint a Miyaguchi-Preneel egyirányú tömörítő eljárást. A háromféle memória típus mindegyike külön célokat szolgál:

- A ROM tárolja a kontrollerhez tartozó egyedi titkos kulcsot és az egyedi azonosítót.
- A RAM-ban az ideiglenes kapcsolatkulcsok és a PRNG változók kapnak helyet.
- A flash memóriába kerül a PRNG seed-je, a Boot memória MAC *tag*-je és a hozzá tartozó titkos kulcs.

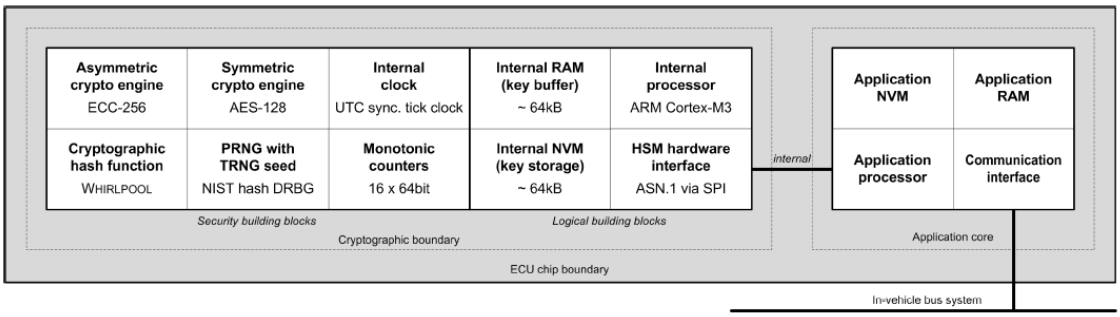
Ezt a modult kifejezetten a biztonságos szoftver bootolás céljára fejlesztették ki. A reset utáni induláskor ez a modul a programmemórián lefuttatja a CMAC algoritmust,

összeveti a flash-ében tárolt értékkel, és ha a kettő megegyezik, akkor elindulhat a rendes szoftver bootolása.

4.2 Az EVITA projekt

Az EVITA az Európai Unió és különböző európai egyetemek, autóipari és elektronikai cégek által 2008 nyarán indított 3 éves kutatási projekt volt [19]. Célja egy olyan prototípus hardver kifejlesztése volt, mely képes a biztonságkritikus alkalmazások számára egy védett futási környezetet nyújtani, mely segítségével hitelesíthető az ECU program és a kapcsolódó hálózatok működése, valamint érzékelhető a rosszindulatú beavatkozás. Emellett egy ECU védelmi architektúra kidolgozása is a feladata volt, ennek keretében létrejött a védelmi hardvermodul-készlet (HSM), szoftveres védelmi komponensek, továbbá hozzá tartozó védelmi protokollok.

A projekt során létrejött egy, az alábbi ábrán látható ECU periféria-készlet:



4.2. ábra: Az EVITA biztonsági moduljának teljes eszközkészlete

A korábban feltárt hardveres biztonsági követelményeknek megfelelően a modul az ECU chip része, így fizikai támadásokkal szemben ellenálló. Külön CPU vezérli a perifériákat, és egyben szolgál az ECU szoftverét hitelesíteni képes védett entitásként. Elkülönített ROM és RAM memóriával rendelkezik a titkos kulcsok és titkosítandó adatok tárolása céljából. A teljes modul tartalmaz szimmetrikus, aszimmetrikus és hash titkosító algoritmust futtató egységeket a gyors és védett titkosítási eljárások végrehajtása érdekében. Ezen kívül a kulcsok megfelelő menedzsmentje és a rossz kulccsal történő potenciális támadások kiszűréséhez szükséges egy valódi véletlenszám-generátor, egy rendszer óra, és egyszerű számlálók.

Az EVITA projekt során figyelembe vették az autóiparra nehezedő költséghatékonysági elvárásokat, ezért a teljes modult háromféle, gyakorlati megközelítésű implementációra bontották attól függően, hogy az adott ECU milyen

feladatot lát el. A három verzió gyakorlatilag a teljes modul egymás részhalmazai, ahogy az az alábbi ábrán látható.

HSM / Feature	EVITA full	EVITA medium	EVITA light	HIS SHE	TCG TPM/MTM	Usual smartcard
Bootstrap integrity protection	Authentic and/or secure	Authentic and/or secure	Authentic and/or secure	Secure	Authentic	None
HW crypto algorithms (incl. key generation)	ECDSA,ECDH, AES/MAC, WHIRLPOOL/ HMAC	ECDSA,ECDH, AES/MAC, WHIRLPOOL/ HMAC	AES/MAC	AES/MAC	RSA, SHA-1/ HMAC	ECC, RSA, AES, 3DES, SHA-x & more possible (but seldom in parallel on chip)
HW crypto acceleration	ECC,AES, WHIRLPOOL (FPGA/ASIC)	AES (ASIC)	AES (ASIC)	AES (ASIC)	None	None
Internal CPU	Reprogrammable firmware & hardware (FPGA)	Reprogrammable firmware	None	None	Preset	Reprogrammable firmware
RNG	TRNG	TRNG	PRNG w/ external seed	PRNG w/ external seed	TRNG	TRNG
Counter	16x64bit	16x64bit	None	None	4x32bit	None
Internal NVM	Yes	Yes	Optional	Yes	Indirect (via SRK)	Yes
Internal clock	Yes w/ external UTC sync	Yes w/ external UTC sync	Yes w/ external UTC sync	No	No	No
Parallel access	Multiple sessions	Multiple sessions	Multiple sessions	No	Multiple sessions	No
Tamper protection	Indirect (passive, part of ASIC)	Indirect (passive, part of ASIC)	Indirect (passive, part of ASIC)	Indirect (passive, part of ASIC)	Yes (mfr. depended)	Yes (active, up to EAL5)

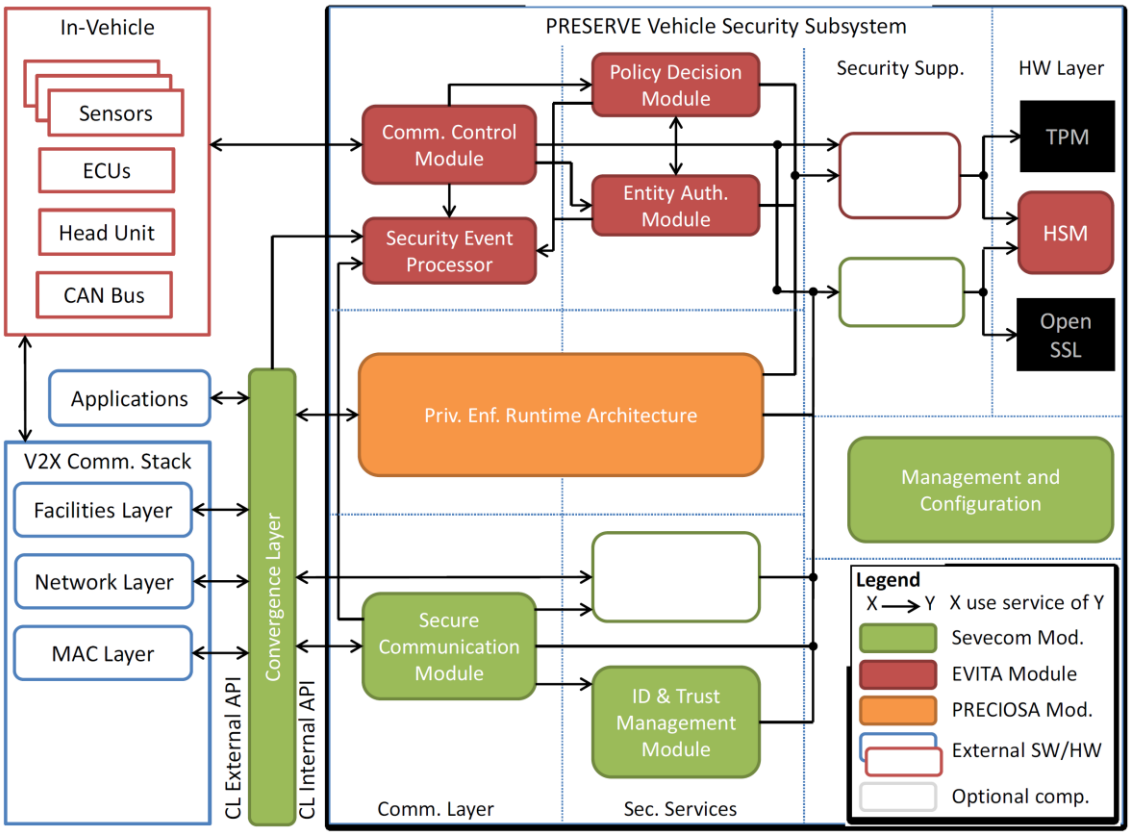
4.3. ábra: Az EVITA modulválasztéka
egyéb hardver biztonsági modulokkal összevetve [30]

A full modul a mai autókban még ritkán szükséges, hiszen azt a járművön kívüli kommunikációt bonyolító eszközökhöz szánták, azonban a medium és light verziók a fedélzeti kommunikációban részt vevő ECU-kba tervezettek, így azok alkalmazása már a mai járművekben is kívánatos volna. A light verzió során kifejezett cél volt annak kompatibilitása a SHE modullal.

4.3 PRESERVE projekt

A 2011-ben indult 4 éves PRESERVE projekt célja egy biztonságos V2X kommunikációs architektúra létrehozása a SEVECOM, EVITA és PRECIOSA korábbi kapcsolódó projektek alapján. A projekt által kialakított jármű védelmi alrendszer

(VSS) az EVITA full HSM-ét, a SEVECOM szoftvermoduljait valamint a PRECIOSA futási idejű architektúráját ötvözte magába.



4.4. ábra: A PRESERVE által létrehozott biztonságos rendszerarchiektúra (VSS) [31]

A hardver modul itt is a biztonságos kulcstárolás és a kriptográfiai műveletek gyorsítása érdekében került a projektbe. Ezen kívül egy tanúsítvány infrastruktúrát is ki kellett alakítani, mely a V2X kommunikáció során a jármű kilétét igazolni tudja. Továbbá a biztonságos kommunikációhoz és az anonimitáshoz szükséges szoftver komponensek is elkészültek.

5 Beágyazott kriptográfiai könyvtár létrehozása

Ahhoz, hogy beágyazott környezetben tudjak biztonságos szoftver alkalmazásokat készíteni, szükségem volt egy kriptográfiai primitíveket tartalmazó könyvtárra. Ehhez már meglévő, elérhető forrásokat használtam fel és alakítottam át a saját igényeimnek megfelelően. A felhasznált forrásoknak az alábbi követelményrendszernek kellett eleget tenniük:

- Nyílt forráskódú, szabadon felhasználható projekt legyen.
- Kizárólag C nyelvű, architektúra független kód alkalmazható.
- Csak a kriptográfiai funkciók megléte szükséges, minden egyéb kiegészítő szolgáltatás felesleges.
- Legyen könnyen áttekinthető felépítésű, csak a szükséges mennyiségű konfigurációs beállítással.

A másik mérlegelendő feladat annak eldöntése volt, hogy milyen kriptográfiai eljárásokra lehet szükség, és ezek közül melyik protokoll a legalkalmasabb választás (kódméret illetve sebesség szempontjából). Mivel a biztonságos információcsere jelenthet alkalmazástól és támadó modelltől függően titkosított vagy hitelesített adatátvitelt, vagy ezek kombinációját, ezért mindenképpen szükség lesz szimmetrikus blokkrejtjelezőre (a titkosítás biztosítására, valamint CMAC hitelesítésre). A felhasznált beágyazott hardverekben nincs lehetőség titkos kulcsok biztonságos tárolására, így kapcsolatkulcsokra kell hagyatkozni, amelyeket kulcscsere algoritmussal hoz létre a két fél. Erre a célra szükség van egy nyílt kulcsú rejtjelezőre. Ezen kívül még egyirányú függvényt is hozzá érdemes venni a könyvtárhoz, amennyiben a nyílt kulcsú rejtjelezőt tetszőleges hosszú üzenet aláírására akarjuk használni (ekkor az üzenet hash értékét lehet a DSA algoritmussal aláírni). Ezen felül biztosítani kell egy véletlenszám-generáló eljárást is, melyre a kapcsolatkulcs létrehozásakor illetve a kihívás-válasz jellegű hitelesítési eljárásoknál van szükség.

Az EVITA projekt által tett biztonsági modulban mindezek a kriptográfiai eljárások megtalálhatóak, így az itt használt rejtjelezőket és hash függvényeket használtam kiindulásként.

5.1 Kiindulási források

5.1.1 Nyílt kulcsú rejtjelező – ECC

Az elliptikus görbe kriptográfia megvalósítások közül az ARM és AVR beágyazott platformokra optimalizált, nyílt forráskódú (*BSD 2-clause* licencű) µECC [32] könyvtárat használtam fel. Ez a modul kis memóriaigényű, négy szabványos görbe közül választhatunk: *secp160r1, secp192r1, secp256r1,* és *secp256k1* görbéket. Előnye, hogy a fordítási idejű görbeválasztással kisebb kódot, és a görbére optimalizált műveletek segítségével gyorsabb futási időt eredményez.

5.1.2 PolarSSL

A *PolarSSL* egy kettős licencű (GPL vagy kereskedelmi) könyvtár, mely az SSL és TLS protokollokat, valamint az azok által használt kriptográfiai eljárásokat tartalmazza [33]. „A PolarSSL a könnyű érthetőség, alapos dokumentáltság, a teszteltség, a szétcsatolható moduláris felépítés és az architektúra-függetlenség jegyében lett tervezve.” Ebből a könyvtárból én az AES blokkrejtjelezőt a rendelkezésre álló blokkrejtjelezési módokkal, az AES-t CCM módban használó hitelesített rejtjelezőt, valamint az SHA-1 és SHA-2 (256) hash függvényeket emeltem ki. A modulok teljesen különálló egységeket képeznek, valamint külön öntesztelő függvény áll rendelkezésre, amely egyrészt remekül bemutatja a felhasználási sémát, másrészt a használt architektúrán futtatva ellenőrizhető a tesztvektorok segítségével a helyes működés.

5.1.3 Whirlpool hash függvény

Az EVITA projektben a hash függvények közül a *Wirlpool*-ra esett a választás, ezért én is felvettem a könyvtáramba. A Whirlpool C nyelvű forráskódja a szerzők honlapján közvetlenül elérhető, „bármilyen célra szabad felhasználású, és az is marad” [34].

5.1.4 Egyéb open-source alternatívák

A fentebb említetteken kívül számtalan nyílt forráskódú könyvtár elérhető, melyek kriptográfiai funkciókat megvalósítanak (pl: *OpenSSL* [35]). Ezek felhasználásánál figyelembe kell venni a licenc típusát, valamint mindenképpen érdemes lefuttatni a kriptográfiai eljárásokat a szabványos tesztvektorok segítségével annak érdekében, hogy megbizonyosodjunk a megfelelő működésről az adott fordító és

architektúra környezetben. Ezen felül az adott alkalmazástól és támadó modelltől függően érdemes időzítési és fogyasztási analízisnek alávetni az adott eljárásokat, hiszen ezek alapján lehetségesek az oldalcsatornás támadások.

5.2 Könyvtár megvalósítása

5.2.1 Koherens elnevezési struktúra

A forráskódok függvényei és struktúrái többféle elnevezési logika alapján lettek elnevezve, ezeket egységesítettem a `Modul_FüggvényNév` szintaxis alapján. A legtöbb kriptográfiai eljárás felépítése megegyező sémát követ:

- Első lépésként (tipikusan a kulcs segítségével) létre kell hozni egy kriptográfiai kontextust (Start);
- Ez a kontextus szolgál a továbbiakban a bemenő adatok transzformálására, és amennyiben adatfolyam jellegű az eljárás, akkor kimenő adatot is szolgáltat (Update);
- Ha az eljárás kimenete Update műveletek akkumulációja során jön létre, akkor az eljárás utolsó lépéseként jön létre a kimenő adat (Finish);
- A kontextus tartalmát annak felhasználásának befejeztével törölni kell (Clear).

Az AES kódjában két jelentősebb módosítást végeztem. Az egyik változtatás az volt, hogy az AES kontextus struktúrát – mely alapból csak a kulcsütemező puffert tartalmazta – kibővítettem két 128 bites pufferrel, melyek a blokkrejtjelezési módok során felhasználhatóak, mint belső shift regiszter és kimeneti puffer. A másik változtatás, hogy az egyes blokkrejtjelezési módokhoz külön Start és Update függvényeket írtam. Előbbi beállítja a rejtjelező irányát (titkosítás vagy visszafejtés) a kontextusban, a módtól és az iránytól függően létrehozza a kulcsütemező puffert, és betölti a shift regiszterbe a kezdeti változót (IV). Utóbbi a megadott hosszúságban rejtjelez a bemenet blokkjaiból a kimenet blokkjaira a kontextusból kinyert irányban.

Mivel a *PolarSSL*-ben nem valósítottak meg blokkrejtjelező-alapú üzenethitelesítő protokollt (CMAC), én viszont szerettem volna összevetni a HMAC algoritmussal, ezért magam megvalósítottam az AES-CMAC algoritmust [36] alapján. Az ugyanitt megadott tesztvektorokkal sikeresen teszteltem a működését.

5.2.2 Dinamikusan betölthető könyvtár létrehozása

Miután meggyőződtem a kód futtathatóságáról és helyes eredményeiről, a *Visual Studio* segítségével létrehoztam egy C nyelvű dinamikusan betölthető könyvtárat (DLL-t). Ennek az volt a célja, hogy a forráskódot ne kelljen portolnom *C#*-ra, hanem ugyanazt a megvalósítást használhassam a beágyazott rendszerben valamint a PC-s oldali kliensben. A fejlesztőkörnyezet által előre generált sablonba csak be kellett illesztenem a forrásfájlokat, valamint az exportálandó függvények elé a `CRYPTOLIB_API` makrót kellett tennem, ahol a makró a következőképpen értelmezett:

```
#ifdef CRYPTOLIB_EXPORTS
#define CRYPTOLIB_API __declspec(dllexport)
#else
#define CRYPTOLIB_API __declspec(dllimport)
#endif
```

Ahogy azt a mellette lévő komment leírja a Visual Studio-ban, a dll fordítás során az összes fájlt a `CRYPTOLIB_EXPORTS` parancssori argumentummal fordít, a dll használata során pedig ezt az argumentumot nem használva lehet importálni a függvényeket.

5.2.3 DLL importálás és nem menedzselt kód kezelése

A kriptográfiai könyvtárból elkészített dinamikus csatolású könyvtár függvényeinek meghívására a `System.Runtime.InteropServices` névtér ad lehetőséget az ún. *PInvoke* [37] használatával. Miután elhelyeztük a *dll*-t a kliens futási mappájába, a főablak osztályban az alábbi módon lehet meghívni a *dll*-ből exportált függvényeket:

```
typedef struct {
   uint32_t Buffer[64];
   uint8_t ShiftReg[16];
   uint8_t Tmp[16];
   uint32_t NbrOfRounds;
   uint32_t *RoundKeys;
   uint32_t ShiftRegPos;
   uint8_t Mode;
} AES_Ctx;

CRYPTOLIB_API
uint8_t AES_CTR_Update
   (AES_Ctx *ctx,
   uint32_t length,
   uint8_t *input,
   uint8_t *output);
```

```
[StructLayout(LayoutKind.Sequential)]
private unsafe struct AES_Ctx {
      fixed uint Buffer[64];
      fixed byte ShiftReg[16];
      fixed byte Tmp[16];
      uint NbrOfRounds;
      uint* RoundKeys;
      byte ShiftRegPos;
      byte Mode;
}

[DllImport("CryptoLib.dll",
CallingConvention = CallingConvention.Cdecl)]
static extern byte AES_CTR_Update
(ref AES_Ctx ctx, uint length,
[In]byte[] input, [Out]byte[] output);
```

5.1. ábra: Nem menedzselt kód importálása PInvoke használatával

Mint látható, a *dll*-ből importált függvényeket statikus függvényekként kell deklarálni. Alapértelmezésben a hívási konvenció a *stdcall*, de mivel én a dll-t *cdecl* konvencióval fordítottam, ezért ezt explicit módon meg kell adni. A beépített típusokat (`int` és társai) a méretüknek és előjelességüknek megfelelő *C#*-os változatukká kell cserélni. (String-ek kezelése esetén a karakterkészletet is meg kell adni.) A natív kódban a cím szerinti paraméterátadás két célt szolgálhat: az egyik egy memóriaterület (tömb) átadása a kezdőcímével, a másik pedig a függvényen belüli módosítása a paraméter értékének. Előbbi esetben a menedzselt kód függvénydeklarációjában a cím szerinti paramétert ki lehet cserélni tömb paraméterre (ekkor nem követelmény az `unsafe` kontextus). Utóbbi esetet a menedzselt kódban a `ref` kulcsszóval lehet megvalósítani.

Egy *PInvoke* hívás 10-30 közötti x86 utasításnyi overhead-et okoz, amelyhez hozzáadódik a *Marshalling* – vagyis a menedzselt és natív adatstruktúrák közötti konverzió és megfeleltetés – miatti késleltetés. Többek között ezért is célszerű a függvény (nem beépített típusú) paramétereinél attribútumban megadni a *Marshalling* irányát (`[In]`, `[Out]`, vagy `[In,Out]`). Tömböket használó natív függvények importálása során érdemes lehet egy csomagolófüggvényt létrehozni a menedzselt kódban, mely például a neki átadott menedzselt tömb hossz tulajdonságával hívja meg az importált függvényt, így elkerülve a natív kód által okozható stack tönkretételt.

A nem menedzselt kódból nem lehet struktúra- és osztálytípusokat importálni, ezeket a kliens kódban is definiálni kell. A natív kódú struktúrákból menedzselt struktúrákat alkotni nem triviális feladat, főleg azokban az esetekben, ahol mutatók vagy tömbök szerepelnek. Az 5.1. ábrán látható `AES_Ctx` struktúrán keresztül mutatom be a legfontosabbakat. Rögzített méretű tömböket kétféleképpen lehet létrehozni. Az első, általam is használt megoldás csak a beépített típusokra érvényes, ekkor az adott tömböt a `fixed` kulcsszóval rögzített méretűvé lehet tenni (a struktúrát `unsafe` típusúnak definiálva). A másik megoldás az adott tömb attribútumában megadni a tömb méretét:

```
[MarshalAs(UnmanagedType.ByValArray, SizeConst=64)] uint[] Buffer;
```

Amennyiben mutatót is tartalmaz a struktúra, úgy az `unsafe` típusjelzés szintén kötelező.

5.3 Memória- és számítási igény összehasonlítása

A kriptográfiai könyvtár eljárásainak memória- és számítási igényét az eddig is használt beágyazott platformon, az ARM Cortex M4 magú kontrolleren mértem. A

függvények futási idejének méréséhez úgy konfiguráltam be a rendszert, hogy egy időzítő a CPU órajelével megegyező frekvencián járjon (84 MHz-en), a mérés kezdésekor elindítom, befejeztével leállítom. A memóriaigény felméréséhez az *arm-none-eabi-readelf* programot használom, mely az *elf* formátumú fordító fájlból ki tudja nyerni a szimbóluminformációkat, így többek között a függvények memóriaméretét is. Mivel a kriptográfiai eljárásokon végeztem saját optimalizációt, így a következő elemzések elsősorban összehasonlító szerepűek. A számítási igény felméréséhez minden esetben 1024 bájtos adatblokkon végzett műveletek alapján vonatkoztattam az egy bájtra vett értékeket.

5.3.1 AES blokkrejtjelezési módok

Az AES-re épülő blokkrejtjelezési módok ugyanazt a kulcsütemezőt és blokkrejtjelezőt használják, a mérések után azonban mégis jelentős eltérésekre derült fény. Először vessünk egy pillantást magának a rejtjelező magnak a jellemzőire:

	Memóriaigény (bájt)	Kulcsütemezőhöz szükséges CPU ciklusok			
		$E_{K[128]}()$	$D_{K[128]}()$	$E_{K[256]}()$	$D_{K[256]}()$
AES	2564	2640	9009	2912	11438

Az általam használt blokkrejtjelezési módok közül csak a CBC használja a dekódolás során a kulcsütemezőt dekódoló irányban, ami jóval időigényesebb lefutású, mint a titkosító irány (többek között mivel ez előbbi alatt le kell futtatni az utóbbit is). Itt fontos észrevenni, hogy a kulcsméret nem befolyásolja arányos mértékben a kulcsütemező lefutási idejét (10 és 20 %-kal tart tovább a kétszer nagyobb kulcs alapú ütemezés), így megfontolandó a nagyobb kulcsméret általános használata a nagyobb biztonság érdekében. Most pedig következzen a blokkrejtjelezési módok teljesítménye:

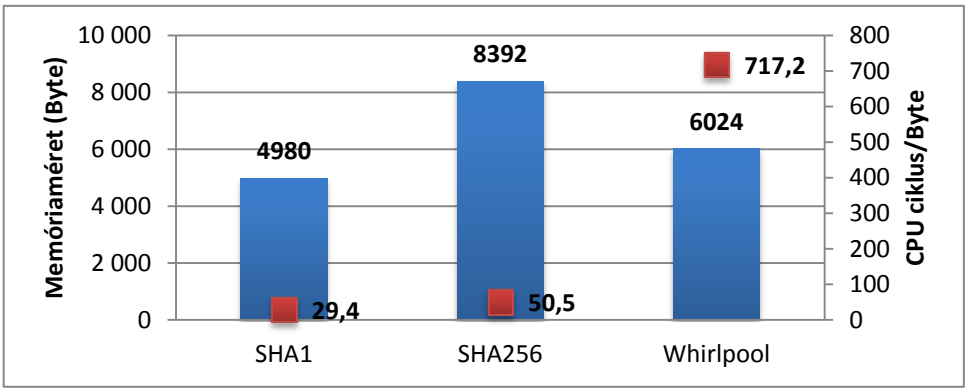
	Memóriaigény (bájt)	Egy bájt rejtjelezéséhez szükséges CPU ciklusok			
		$E_{K[128]}()$	$D_{K[128]}()$	$E_{K[256]}()$	$D_{K[256]}()$
CBC	222	166,6	172,2	219,1	224,5
CFB128	212	184,5	185,3	236,9	237,4
CFB8	150	2466,1	2474,5	3305,8	3309,9
CTR	162	182,8	182,8	235,3	235,3

Mindenekelőtt leszögezhető, hogy minimális extra memóriaigényük van az alap rejtjelezőhöz képest. A CFB8 egyértelműen rossz választás sebesség szempontjából, hiszen a többi blokkrejtjelezővel ellentétben minden 8 bit rejtjelezéséhez egy teljes AES blokkot használ fel, így gyakorlatilag körülbelül 16-szor annyi időbe kerül rejtjelezni

vele egy adatot, mint a többi móddal. A többi versenyző között nincs számottevő eltérés, habár a CBC az első, ha nem 16 bájtos blokkokból álló adatot kell átvinni, akkor ehhez még hozzáadódik az üzenet végi kitöltés létrehozása vagy leválasztása.

5.3.2 Hash függvények

A felhasznált hash függvények esetén nemcsak a megvalósításban vannak eltérések, hanem a lenyomat méretében is. Elsőként vessünk egy pillantást az alábbi diagramra:



5.2. ábra: Felhasznált hash függvények teljesítménymutatói

A két SHA algoritmus hasonlóan teljesít a saját kategóriájában, bár a lenyomat méret arányához képest (32/20 bájt) az SHA-256 majdnem kétszer akkora helyet igényel, mint az SHA1. A számítási igény esetén viszont már megegyezik a lenyomat méretének aránya és a szükséges CPU ciklusok számának aránya. A sokkal megdöbbentőbb eredményt a Whirlpool nyújtotta, mely 64 bites lenyomata ellenére kisebb, mint az SHA-256, viszont a számítási igénye messze felülmúlja az előbbieket. Ennek a fő oka az lehet, hogy a Whirlpool nem bájtfolyamot, hanem bitfolyamot dolgozik fel, és a bitkezelés nagyban lelassítja a folyamatot. Másrészt a Whirlpool lassúságában a szoftver műveletek sorrendisége is közrejátszhat, hiszen párhuzamos hardver implementáció esetén nagyon bíztató eredményeket értek már el vele (többek között ezért kerülhetett az EVITA HSM-be is).

5.3.3 MAC algoritmusok

Ennek a fejezetnek a fő célja, hogy a blokkrejtjelezőt használó MAC-et összehasonlítsa a hash függvényre épülővel. Itt a memóriaigény nincs feltüntetve, az nem sokban tér el a MAC-re használt kriptográfiai elem méretétől.

	Overhead CPU ciklusokban		Egy bájt hozzáadásához szükséges CPU ciklusok	
	K[128]	K[256]	K[128]	K[256]
AES-CMAC	7165	8252	155,9	215,8
SHA1-HMAC	10145	10625	29,4	29,4
SHA256-HMAC	16434	16877	50,5	50,5

Az itteni eredmények azt mutatják, hogy a HMAC megvalósítás mindenképpen kifizetődőbb hosszú üzeneteknél, illetve a kulcs mérete sem befolyásolja számottevően a futási idejét. Ezzel szemben a CMAC kisebb indulási overhead-del rendelkezik, valamint az AES memóriaigénye is fele az SHA1-nek, így ezt az algoritmust rövid üzenetek hitelesítésére alkalmasabb (amihez rövid, 16 bájtos *tag* társul, szemben az SHA1 20 és az SHA-256 32 bájtjával). Végeredményben 128 bites kulcs esetén az SHA1 22-nél több, az SHA-256 82-nél több bemeneti bájt esetén gyorsabb, mint az AES-CMAC, 256 bites kulcs esetén ezek a küszöbök rendre 13 és 32.

5.3.4 Hitelesített titkosítás

A dedikált hitelesített titkosító eljárások közül csak a CCM módot vettem át, mely nem csak egy egyszerű, de igen hatékony séma is. Ha összevetjük az alábbi táblázatot a fenti CMAC és CTR módok eredményével, látható, hogy az itteni eredmények visszatükrözik a CCM mód felépítését. Amennyiben csak a hitelesítési funkcióját használjuk, annak teljesítménye megegyezik a CMAC-éval, hiszen a CBC-MAC eljárást használja mindkettő. A hitelesített titkosítás során minden bájt hitelesítésén felül annak rejtjelezése is megtörténik CTR módban, ebben az esetben a CCM-en mért eredmény a CMAC és a CTR módok összege.

	Memóriaigény (bájt)	Egy bájt hitelesítéséhez szükséges CPU ciklusok		Egy bájt rejtjelezéséhez és hitelesítéséhez szükséges CPU ciklusok	
		K[128]	K[256]	K[128]	K[256]
CCM	956	170	224,8	332,1	438,8

Az 5.3. ábra mutatja be több elterjedt hitelesített rejtjelező teljesítményét (egy bájt rejtjelezéséhez szükséges CPU ciklusban, x86 architektúrán). Az ábrán jól látszik, hogy a CCM a saját mezőnyében igen jó teljesítményt nyújt, amelyet több egyéb faktor megnövel. Egyrészt az OCB módot az Egyesült Államokban szabadalom védi, aminek következtében ott nem lehet kereskedelmi célú szoftverben felhasználni (egyébként elérhető a GPL licenc alapján is, de ez alapján viszont nyílt forráskódúvá kellene tenni az OCB-t felhasználó szoftvert). Másrészt a másik vetélytárs, a GCM mód csak úgy tud

jobb teljesítményt elérni, hogy nagy méretű LUT-okkal puffereli a műveleteket (ezeket a kulcs birtokában lehet előre számítani).

Mode	Message size (bytes)				
	16	64	256	1024	8192
CBC-HMAC-SHA1	1270	342	124	68.4	51.2
CCM	159	75.6	54.5	49.2	47.6
CWC	227	102	72.7	63.3	61.2
EAX	239	93.8	59.4	51.1	48.0
GCM, 64Kb storage	60.8	44.8	36.1	36.6	38.1
GCM, 8Kb storage	89.9	51.9	42.9	43.0	40.1
GCM, 4Kb storage	118	69.1	46.5	54.1	53.5
GCM, 256b storage	179	108	89.5	85.4	84.6
OCB	89.4	43.3	31.4	29.3	29.0

5.3. ábra: Hitelesített titkosítási sémák számítási igénye egy bájtra jutó CPU ciklusban [38]

5.3.5 Aszimmetrikus rejtjelező

Az aszimmetrikus ECC rejtjelező képes a legnagyobb szintű védelem megteremtésére, viszont az előző eljárásokhoz képest mérhetetlenül nagyobb a számításigénye. Ennél a táblázatnál már mértékegység váltásra volt szükség a szemléletesebb megjelenítés érdekében, összehasonlításként a 100 ms CPU ciklusokban 8.400.000-nek felel meg.

	Memória-igény (bájt)	Az adott művelet számításigénye (84 MHz CPU frekvencia mellett, ms)				
		kulcspár létrehozás	publikus kulcs ki-tömörítés	közös kulcs létrehozás	aláírás	aláírás hitelesítés
secp160r1	4380	91,588	10,882	91,954	102,694	112,147
secp192r1	3848	108,960	12,261	106,707	117,453	131,670
secp256r1	4144	277,659	22,348	276,734	295,013	340,688
secp256k1	3736	233,475	30,068	233,442	247,853	253,905

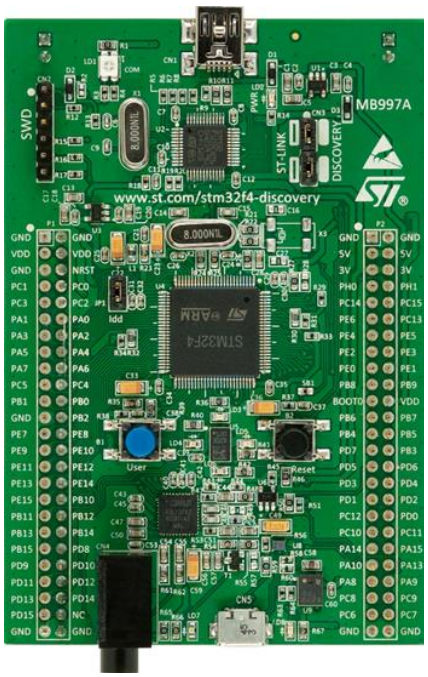
Az eredmények alapján látszik, hogy adatok titkosítására használni semmiképpen nem érdemes, hiszen ilyenkor a blokkméretének megfelelő adatonként kellene egy közös kulcs létrehozás szintű műveletet elvégezni, ami megengedhetetlen. Ehelyett célszerű csak olyan protokollokra használni, melyek aszimmetrikus rejtjelezőre épülnek (lásd ECDH, ECDSA). Mindemellett ez egy igen kis méretű implementáció, mely azt is magában hordozza, hogy nem kerül sokba a felhasználása, így ritka alkalmazása mellett sem jelent problémát a jelenléte.

6 CAN-USB átjáró megvalósítása

Áttérve a diplomatervem diagnosztikai rendszerének kivitelezési fázisába, első lépésként a PC alkalmazás és az ECU közötti kommunikációs kapcsolatot létrehozó átjárót kellett kiviteleznem. A saját kivitelezésű átjáró választását az motiválta, hogy egy egyszerű, olcsó megoldás jöjjön létre, mely egyúttal széleskörű konfigurációt tud biztosítani. A költséges és időigényes hardvertervezés elkerülése érdekében egy fejlesztőkártyát egészítettem ki a szükséges CAN fizikai réteggel, így csak a szoftver implementációval kellett foglalkoznom. A szoftverfejlesztést Eclipse alapon, nyílt forráskódú GNU ARM fordító és OpenOCD debugger segítségével végeztem.

6.1 Hardver kialakítás

A fejlesztői kit az általam már korábban is használt STM32F4Discovery [39], mely tartalmaz egy 168 MHz-es maximális órajelű, ARM Cortex M4F magú, többek között 1 MB flash memóriával, CAN és USB perifériákkal rendelkező mikrokontrollert. A kiten található ezen felül a rajta lévő – vagy akár egy külső – kontroller programozására alkalmas ST-LINK interfész, ezen keresztül kap tápot is a kártya. A másik, alsó USB csatlakozó a kontrollerre van kötve, ezen keresztül lehet a kontroller beépített USB perifériáját használni.



6.1. ábra: Az STM32F4Discovery fejlesztőkártya felülnézete

Az egyetlen kiegészítés, amit a kártyán meg kellett ejteni, az a CAN fizikai rétegének hozzáadása volt. Ehhez egyrészt szükség volt egy fizikai meghajtó áramkörre, itt az MCP2551-es típusra esett a választás, mivel képes a szabványos CAN bitsebességeken üzemelni, valamint elterjedt, egyszerű és olcsó kivitel. Másrészt a CAN busz recesszív állapotának beállításához szükséges 120 Ω-os ellenállást is biztosítani kellett, ugyanis tesztkörnyezet lévén valódi CAN busz nem áll rendelkezésre, csupán egy ugyanilyen kialakítással rendelkező, ECU szerepet betöltő

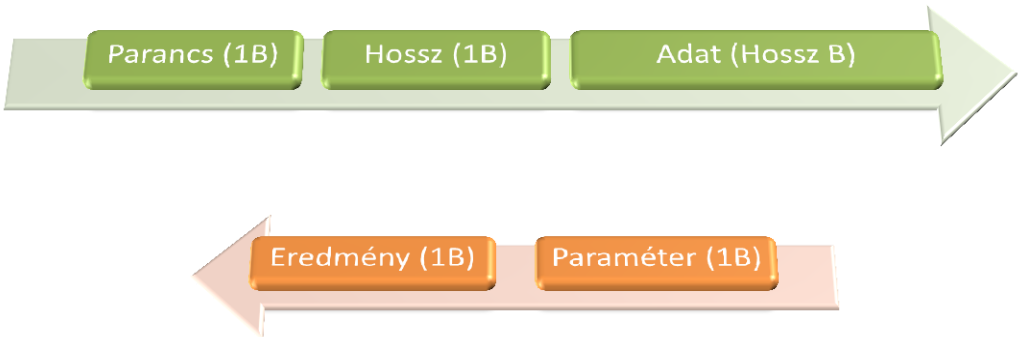
kártya. Ezek rögzítése a kártyához nem volt bonyolult, hiszen a DIP8-as meghajtó is 100 mil-es lábtávolsággal rendelkezik, ugyanúgy, mint a kit tüskesora, így egy prototípuspanel segítségével össze lehetett állítani az átjáró hardverét.

6.2 USB virtuális soros port interfész

A fejlesztőkártyához nem csak az USB 2.0 Full Speed periféria hardver oldala volt előre elkészítve, hanem a szükséges USB CDC (Communications Device Class) osztályt megvalósító forráskód is a gyártótól [1]. Mindösszesen néhány konfigurációs beállítást kellett elvégeznem: sebesség kiválasztása (Full Speed), lábkiosztás beállítása, Low Power mód engedélyezése, valamint a szükséges két megszakítás (adatátvitel és ébresztési célú) konfigurációja a NVIC modulban. A kommunikáció mindkét irányban dedikált USB végponton keresztül történik, FS mód miatt a maximális 64 bájtos keretmérettel. Az adatküldés egy 2 kB-os FIFO-ból történik.

6.3 Átjáró soros port protokoll

Az átjáró PC oldali megvalósításához létre kellett hoznom egy olyan protokollt, mely az USB virtuális soros port bájtfolyamát az átjáró funkcionalitásának megfelelő adatcsomagokként képes értelmezni. A feltételek adottak voltak: mindkét irányban különféle, eltérő adathosszúsággal rendelkező csomagok áramolhatnak, ezek eredményéről mind pozitív, mind negatív kimenetel esetén érkezzen visszajelzés, jelezve a hiba forrását is. Emellett cél volt a minél hatékonyabb sávszélesség kihasználás, vagyis a minél tömörebb, felesleges vagy redundáns információ nélküli adatcsomagok átvitele. Az átvitel általános vázlata az alábbi ábrán látható:



6.2. ábra: A soros port csomagküldési sémája (bájtméretekkel)

A legelső, még fel nem dolgozott bájt felső két bitje árulja el, hogy az éppen fogadott adatobjektum egy csomag, vagy pedig egy eredmény visszajelzés. Ha az értéke 00, akkor egy csomag, ha 01, akkor egy sikeres eredményt jelez, ha pedig 10, akkor feldolgozási hibát. Az alsó hat bit a csomag típusát jelöli minden esetben. Ez alapján válik el kétfelé az új adat feldolgozása:

- Csomag esetén meg kell várni a következő bájtot, valamint ennek a bájtnak az ismeretében, ennek az értékének megfelelő mennyiségű bájtot, ezután pedig fel kell dolgozni a csomagot a csomag típusának megfelelően.
- Eredmény visszajelzés és pozitív kimenetel esetén nincs teendő, az adott azonosítóval rendelkező küldött csomag feldolgozása sikeresen megtörtént (a paraméter tartalma jellemzően másodlagos, viszont csomag típustól függő). Negatív kimenetel esetén a paraméter bájt szolgál információval a hiba típusáról (ez esetben csomag független az értelmezése, viszont nyilván nem minden hibakód következhet be minden csomag esetén).

Egy nagy hátránya ennek a formátumnak az, hogy szinkronizációs célú start és stop karakterek hiányában egy hibásan küldött vagy értelmezett adatobjektum potenciálisan tönkreteheti az utána következő teljes kommunikációt. Ennek a hibának az elkerülésére az átjáró funkcióinak eléréséhez a kliensnek először egy jelszót tartalmazó csomagot kell elküldenie, csak ennek hiteles formátuma és tartalma esetén válik elérhetővé az átjáró CAN interfésze. Amennyiben az átvitel során az átjáró hibás formátumú csomagot detektál, növeli a hibaszámlálóját, ami ha egy adott határt átlép, akkor lezárja a CAN kapcsolatot, és kiindulási állapotba helyezi az átjárót. Ezzel a megoldással elérhető, hogy csak az átjáró indításakor kelljen egy karaktertömböt küldeni, a kommunikáció további részében az átjáró szoftvere és a kliens alkalmazás helyes implementációja garantálja a megfelelő csomagértelmezést.

A fejlesztés egy későbbi fázisában nyilvánvalóvá vált, hogy mivel az átjáró csak a CAN busz eseményeit (keretfogadás, busz hibák) közvetíti adatcsomagokkal a kliens felé, ezért neki nincs szüksége a kliens visszajelzésére az adott csomagról, hiszen ezek nem utasítást jelentenek, pusztán adattartalmat.

6.3.1 Csomag típusok

A csomag által hordozott adat tartalmát a csomag típusától függően kell értelmezni. Az alábbiakban felsorolom az eddig kivitelezett csomag típusokat:

1. `AccessGateway`: Az átjáró feloldásához szükséges jelszót tartalmazza, üres string segítségével lehet lezárni az átjárót. A pozitív visszajelzés paraméterében az eddigi sikertelen próbálkozások száma szerepel.
2. `CANConfig`: A CAN periféria inicializálásához szükséges adatokat küldjük el, az első bájt tartalmazza a kiválasztott Baudrate-et, és a globális CAN konfigurációs flag-eket. Az ezt követő bájtok 9-es csoportonként alkotnak egy-egy keretfogadó ID szűrőt. Ezekben a csoportokban az első bájtok a szűrő típusát (16 vagy 32 bit széles, lista vagy maszk üzemmód), a következő 8 bájt pedig a szűrő regiszterek értékét tartalmazzák. A pozitív visszajelzés során a küldési puffer méretét jelzi az átjáró a paraméterben.
3. `SingleStdFrame`: Egy standard ID-val rendelkező CAN keretet tartalmaz, az első két adatbájt az ID, az azt követőek a CAN keret adatbájtjai. Ezt a csomagot mind a kliens alkalmazás, mind az átjáró küldheti, előbbi egy CAN keret küldése céljából, utóbbi egy fogadott keret esetén. Az átjáró a negatív visszajelzést azonnal elküldi, a pozitív visszajelzés küldése csak az után történik meg, hogy a keretet sikeresen adta a buszon az átjáró (a pozitív paraméter a küldő puffer szabad helyeinek száma).
4. `SingleExtFrame`: Egy extended ID mezővel rendelkező CAN keretet tartalmaz, az előzővel megegyező tulajdonságokkal, azzal a kivétellel, hogy az ID az első négy adatbájtot kitölti.
5. `MultiStdFrame`: Olyan adatcsomag, mely ugyanazon standard CAN ID-val egyszerre több CAN keretet definiál. Az első két ID bájt kivételével a csomag tartalma egymás után folyamatosan tölt fel CAN kereteket azok maximális 8 bájtjáig, amíg el fel nem használta a csomag teljes tartalmát (ha nem lenne elég hely a CAN pufferben, akkor egyetlen keretet sem hoz létre. Ezt a kerettípust csak a kliens küldheti, a bevezetése az adatfeltöltés gyorsítása érdekében történt olyan protokollok esetén, melyek egy adott ID-val több keretből álló csomagokat küldenek CAN-en.
6. `MultiExtFrame`: Az előbb elmondottak alapján könnyen kitalálható, az előző típussal megegyező csomag, az első négy bájt az ID, az utána jövő adatok a CAN keretek adatai.

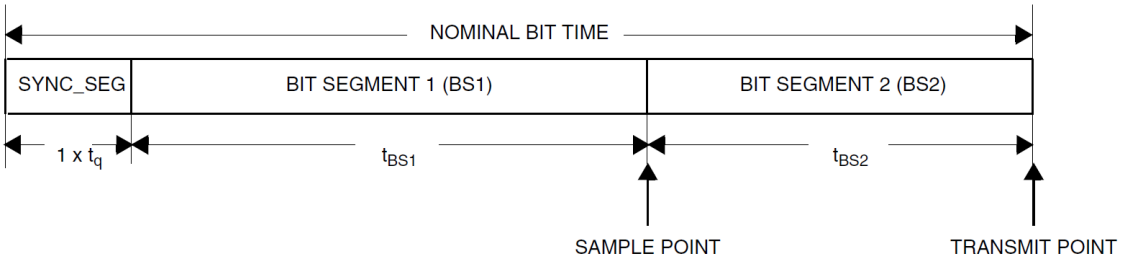
7. `CancelStdFrame`: Az átjáró pufferéből kitörölteti az első két bájtban megadott standard ID-jú keretek közül az első, a harmadik bájtban specifikált darabot. A pozitív visszajelzés során a sikeresen eltávolított keretek száma szerepel a paramétermezőben.
8. `CancelExtFrame`: A korábbiakkal analóg módon értelmezett.
9. `CANError`: Egyetlen adatbájtot tartalmazó csomag, melyet az átjáró küld a kliensnek, ha hibaesemény történik a CAN buszon. Az adatbájt tartalmazza a legutolsó buszhiba forrását, valamint a CAN periféria hibaállapotát.

Ez a jelenlegi parancskészlet még tovább bővíthető egészen 64 parancsig, és nem is teljes értékű, hiszen például nincs megvalósítva a gyakorlatban elvétve használt CAN Remote Transmit Request kerettípusa, illetve az átvitel szüneteltetése és folytatása lehet egy jövőbeli bővítési lehetőség, de ezt a kliensben meg lehet kerülni.

6.4 CAN periféria kezelése

Az STM32 mikrokontroller szériában az úgynevezett Basic Extended CAN (bxCAN) periféria található meg, mely a szabvány 2.0A és B verzióit támogatja (azaz a standard 11 bites ID-val és az extended 29 bites ID-val rendelkező kereteket). A periféria támogatja a szabványban megadott maximális 1 Mbit/s-os bitsebességet, valamint a TTCAN-hez (Time-Triggered CAN) szükséges hardver funkciókat.

Az első és legfontosabb dolog a CAN periféria beállítása során a kiválasztott bitsebességnek megfelelő bitidőzítések konfigurációja volt. A CAN periféria a kontroller egyik APB buszán található, amelyiket a maximális 42 MHz-es órajelre állítottam be. Ebből az órajelből kell előállítani egy órajel-osztó (prescaler) segítségével egy leosztott mintavételezési órajelet (ún. time quantum – TQ periódusú jelet), mely a CAN busz bit értékeit olvassa le a buszról. Kiindulásként a maximális 1 Mbit/s bitsebesség helyes beállítása volt a cél. A szabvány szerint egy bitidő alatt 8 és 25 közötti TQ-nak kell eltelnie. Tehát a TQ-nak 8 és 25 MHz közötti frekvencián kell járnia, méghozzá úgy, hogy a 42 MHz annak egész számú többszöröse legyen, valamint a TQ-nak bitidőnek az egész számú többszörösének kell lennie. Ezek alapján az egyetlen szóba jöhető érték a 21 MHz.



6.3. ábra: A kontroller CAN perifériájának bitidőzítési paraméterei [2]

A CAN szabványban négy bit szegmenst specifikálnak, a SYNC (szinkronizációs), a PROP (busz terjedési idejét kompenzáló), valamint a PHASE_SEG1 és 2, melyek határán történik a mintavételezés. A periféria konfigurációjában a középső két szegmens össze van vonva. 21 MHz esetén 21 TQ-ot kell az ábrán látható módon szétosztani úgy, hogy a szabvány szerinti BS1 ≥ BS2 szabályt betartsuk, és a megengedett értékek közül válasszuk meg BS1-et és BS2-t (előbbi [1; 16], utóbbi [1; 8] közötti értéket vehet fel). Ezen követelmények alapján BS1-et 12 TQ-ra, BS2-t 8 TQ-ra választottam. Ezeken kívül még egy bitidőzítéssel kapcsolatos paramétert kellett beállítanom, ez a Synchronization Jump Width (SJW), mely egy és négy közötti érték, ennyivel lehet a mintavételi pontot későbbre csúsztatni egy bitidőn belül. Ezt az értéket a maximális négyre állítottam be.

Az átjáró tervezése során igény volt több elterjedt bitsebesség támogatására, például az 500 kbit/s és a 125 kbit/s sebességekére. Mivel ezek a sebességek egymás többszörösei, ezért elégséges az órajel-osztót megduplázni illetve megnyolcszorozni. A jelenlegi konfigurációban az 1 Mbit/s-hoz tartozó 2 értéket lehet egy kétbites konfigurációs paraméterrel balra tolni, így megoldott az összes kívánt bitsebesség használhatósága.

A CAN inicializáció során a bitidőzítésen kívül a nyolc darab CAN funkció flaget kellett kiválasztani. Ezek közül néhányat célszerűen fix értékre állítottam, másokat a `CANConfig` paranccsal lehet kiválasztani. A fix beállítások közül az egyik a küldési prioritás sorrendje, mely alapértelmezésben ID prioritás alapján történik a prioritás inverzió elkerülése érdekében. Az átjáróban viszont az a cél, hogy érkezési sorrendben történjen a keretek küldése, így ezt a funkciót be kellett kapcsolnom. A másik fix beállítás másodlagos, az összes többit pedig az alkalmazás tudja megválasztani.

A keretek fogadására a perifériában két darab háromelemű fogadó FIFO elérhető. Ezekre külön-külön megszakítás állítható be, külön prioritással és kezelő függvénnyel. A keretküldésre pedig három küldő postaláda szolgál.

6.5 Az átjáró belső működése

6.5.1 Soros port adatfogadás és csomagfeldolgozás

Az USB CDC interfész egy alkalmazás szintű függvénnyel szolgál az adatfogadásra, melyet az USB megszakítás-kezelő hív meg, ebben adott egy tömb kezdőcím és egy tömb hossz, ebben a függvényben történik az érkezett adat átmásolása a csomagegyesítő pufferbe. Mivel amíg ez a függvény nem tér vissza, addig az interfész blokkolt állapotba kerül és nem képes újabb adatcsomagokat fogadni, ezért végképp nem merült fel a csomag feldolgozása ebben a függvényben. Az érkezett csomagok összerakása és feldolgozása az egyetlen feladat, amelyet nem megszakítás-kezelő függvény végez el, így nincs szükség beágyazott operációs rendszer megvalósítására. A main függvény figyeli, hogy nőtt-e a csomagegyesítő puffer tartalma, ha igen, akkor mindaddig meghívja a csomagfeldolgozó függvényt, amíg a pufferben következő adathalmaz egy teljes csomagot alkot. Amennyiben a csomagfeldolgozó függvény visszatérési értéke jelzi, hogy nem tudja értelmezni az adott csomagot, úgy a main egy hibaszámlálót inkrementál, mely egy adott szint után elveti a teljes puffert, és lezárja az átjárót. Ezek után eltávolítja a feldolgozott csomagokat a pufferből (az utána következő, csonka adathalmazt a puffer elejétől kezdve reallokálja). Ezen teendők végeztével a WFI utasítással Sleep módba helyezi a kontrollermagot (leállítja a belső órajelet), melyből csak megszakítás bekövetkeztekor indul el újra.

6.5.2 CAN keretek küldésének menedzsmentje

A CAN perifériában egy keret tárolására négy darab 32 bites regiszter szolgál:

1. A keret ID-t és a keret flag-eket (extended vagy standard ID, Remote Transmit Request vagy adatkeret) tartalmazó regiszter.
2. A keret adatmezőjének hosszát meghatározó Data Length Code, valamint a keret elküldésének vagy fogadásának 16 bites időpecsétjét tartalmazó regiszter.
3. A keret első négy adatbájtját tartalmazó regiszter.

4. Valamint a keret második négy adatbájtját tartalmazó regiszter.

Az USB CDC interfészen keresztül az átvitel maximum 64 bájtos tömbökben történik, a buszra csatlakozott eszközök számától és átviteli igényétől függő gyakorisággal. Annak érdekében, hogy az CAN adatküldés során a lehető legnagyobb sávszélességgel tudjon kereteket küldeni az átjáró, célszerű volt létrehoznom egy cirkuláris küldési RAM puffert, ami a CAN periféria által szolgáltatott 3 küldő postaláda kapacitását jelentősen megnöveli. Így a küldés során nem fog függeni az USB átvitel nehezen prediktálható késleltetéseitől a CAN busz kihasználtsága. A cirkuláris puffer jelen konfigurációban 64 darab keretet képes tárolni, és ugyanazt a kerettárolási felépítést alkalmazza, mint a CAN periféria, így a pufferből az éppen üres postaládába egy egyszerű memóriamásolással kerül be az adat (a kódban ez két azonos típusú struktúra közötti értékadásként jelenik meg).

A periféria biztosít megszakításkérést a sikeres keretküldésre, ennek a kezelő függvényében történik a keretküldés parancs visszaigazolásának elküldése (pontosabban csak az USB kimenő pufferbe írása), valamint innen is meghívódik a CAN küldést ütemező függvény. Az ütemező függvény megvizsgálja, hogy van-e még küldendő keret a pufferben, illetve egy szemafor segítségével meggyőződik arról, hogy nem fut egy konkurens függvény (ebbe a függvényt magát is bele kell érteni) egy eltérő prioritási szinten. Ezen feltételek mellett mindaddig, amíg talál üres küldő postaládát, abba a puffer legelső keretét bemásolja, továbbá az adott keret típusát elmenti a küldés visszajelzésre szolgáló FIFO-ba, innen tudja a kezelőfüggvény, hogy milyen típusú keretet küldött az átjáró utoljára. Az ütemező függvényt a megszakítás-kezelőn kívül a csomagfeldolgozás során is meghívódik az olyan utasítások miatt, amik a puffer tartalmát módosítják. Ilyen esetben csak akkor van erre szükség, amennyiben a periféria postaládái üresek, ekkor ugyanis nem fog küldési megszakítás bekövetkezni, aminek következtében folytatódhatna a küldés ütemezése.

6.5.3 CAN keretek, busz hibák fogadása és kezelése

A keret fogadás során egyszerű az átjáró feladata: a CAN fogadó megszakítás-kezelő függvényben mindössze a fogadott keret ID típusától és adatának hosszától függő megfelelő csomagformátumot kell létrehoznia, majd át kell másolnia a csomag fejlécet és a periféria FIFO-ból a keret adatait az USB CDC interfész küldési FIFO-jába.

CAN busz hiba esetén a `CANError` csomagot kell összeállítania az átjárónak, jelenleg ez a csomag csak a legutolsó buszhiba típusát, illetve a busz hibaállapotát jelzi. Ezeket az információkat a periféria regiszterekből egyszerűen ki lehet nyerni, egy bájtban kódolható.

7 CAN diagnosztikai alkalmazás

Az USB-CAN átjáró belső szoftverének megalkotása után elkezdhettem annak a PC oldali kezelőalkalmazását megalkotni. Ehhez a Kliensalkalmazások fejlesztése című szakirányos tárgyam során elsajátított ismeretek alapján egy Windows Forms alkalmazást hoztam létre, C# nyelvben. Ennek az alkalmazástípusnak az egyik előnye, hogy egyszerűen létre lehet hozni a grafikus interfészt, hiszen a meglévő vezérlőelemekből az igények túlnyomó többségét meg lehet valósítani, ezeket pedig csak drag-and-drop módszerrel el kell helyezni az alkalmazás Design nézetében. A másik hasznos tulajdonsága a .NET keretrendszer, mely egyrészt garantál egy felügyelt környezetet, másrészt egy nagyon széles körű osztálykönyvtárat biztosít, melyek nagyban le tudják egyszerűsíteni a programozást. A Windows Forms alkalmazásfejlesztés hátránya viszont, hogy csak Windows operációs rendszereken futtatható.

7.1 CAN adatstruktúrák osztályai

Kezdetnek létrehoztam az absztrakt CAN osztályt, egyrészt a CAN buszhoz kapcsolódó enumerációk osztályhoz kötéséhez, másrészt a CAN buszhoz kapcsolódó entitások megvalósításához, mint osztályon belüli osztályok. Utóbbiba tartoznak a CAN Rx szűrő, a keret, a buszhiba és a busz hibaállapot osztályok. Ezen osztályok segítségével az átjáró protokolljának ismerete nélkül lehet kezelni a következőkben bemutatott átjáró interfészt. Megfelelő konstruktorokat és metódusokat kellett alkotnom attól függően, hogy küldés, fogadás vagy mindkettő esetén használt objektumról van szó. CAN Rx szűrőt csak küldeni tudunk az átjárónak, így ennek a konstruktorai csak felhasználói oldalúak, és egy – az `IGatewayPacket` interface-hez tartozó – metódussal lehet az átjáró által értelmezhető bájttömbbé alakítani. A kereteket küldeni és fogadni is lehet, így kétféle konstruktorra volt szükségem, az egyik a felhasználói oldalhoz tartozik, a másik az átjáró által küldött csomag alapján hozza létre a keret objektumot. Az adat- és hibakeretek átadásának eseményvezérelt megvalósításához létrehoztam egy-egy `delegate` típust, melyek segítségével létre lehet hozni CAN struktúrákat átadó `event`-eket. Az alábbi szövegdobozban található kódrészlet pontosabb, bár korántsem teljes rálátást nyújt a CAN osztályszerkezetre.

```
public abstract class Can
{
    public enum IdType
    {
        Standard = 0,
        Extended = 1
    }
    public delegate void FrameHandler(Frame Frame);
    public delegate void ErrorHandler(Error Error, State State);
    public class State {…}
    public abstract class Event
    {
        protected DateTime eventTime;
        protected void SetTime() { eventTime = DateTime.Now; }
        public string Timestamp{
          get{
               return eventTime.ToLongTimeString() + ":" +
               eventTime.Millisecond.ToString("D3");
          }
        }
        virtual public string[] ToStringArray(){…}
    }
    public class Error : Can.Event {…}
    public class Frame : Can.Event, IGatewayPacket
    {
        protected byte[] data;
        protected uint id;
        protected IdType idtype;
        public byte[] Data { get { return data; } }
        public uint ID { get { return id; } }
        public IdType IDType { get { return idtype; } }
        protected bool isSent = false;
        public bool IsSent {
          get { return isSent; }
          set {
               this.isSent = value;
               if (value == true)
                    SetTime();
          }
        }
        public Frame(uint id, byte[] data, IdType idtype) {…}
        public Frame(byte[] gatewayPacket) {…}
        virtual public byte[] GatewayPacket { get {…}}
        override public string[] ToStringArray(){…}
        …
    }
    public class Filter : IGatewayPacket {…}
}
public interface IGatewayPacket
{
     byte[] GatewayPacket { get; }
}
```

7.1. ábra: Kliens CAN osztály vázlatos áttekintése

7.2 Átjáró soros port interfész

A CAN objektumok implementálása után egy olyan osztályt kellett létrehozzak, mely egyrészt képes kezelni egy soros portot, másrészt a felhasználó felé olyan metódus-, és eseménykészletet nyújtani, melyek az átjáró képességeit kihasználják, viszont a konkrét implementációt (a csomagformátumot) elfedik. Ez a gyakorlatban azt jelentette, hogy a kapcsolat nagy részét kizárólag a CAN osztályok használatával végzi el.

7.2.1 Felhasználói metódusok

Az alapvető koncepció a felhasználói metódusok esetén, hogy a bemenő paraméterek alapján összeállítanak és a soros porton elküldenek egy megfelelő csomagot, majd blokkolják a meghívó szálat. A blokkolást a soros port fogadott eseményének kezelőfüggvénye szinkronizációs objektumok (`AutoResetEvent`-ek) jelzettsége, vagy az időtúllépés oldja fel, a függvény visszatérési értéke az adott művelet sikerességét jelzi. Időtúllépés vagy átjáró hiba esetén a kimenő string paraméter írja le a hiba okát.

A CAN busz beállításaihoz kétféle paraméterezés érhető el: szűrők nélküli paraméterezéssel egy minden keretet átengedő szűrőt állít be a metódus, mellyel az összes, a buszon megjelenő keretek továbbítja az átjáró, egyébként a megadott szűrőket továbbítja az átjárónak.

A keretek küldéséhez egy FIFO-t hoztam létre az interfészen belül, mely tárolja azokat a kereteket, melyeket küldésre átadott az átjárónak, de még nem lettek visszaigazolva az átjáró által. A FIFO-ba bekerül egy keret, amikor a keretküldési metódust meghívják, és kikerül, amikor keret küldési visszaigazolást küld az átjáró. Ennek a FIFO-nak a tartalma gyakorlatilag megegyezik az átjáró pufferének tartalmával, így nyomon követhetjük azt is, hogy az átjáró puffere tele van-e, és többféle módon kezelhetjük a teli puffer esetét. Mivel a magasabb szintű szoftver kezelése során sokféle igény szerinti küldési megoldás merült fel, ezért három különböző metódust implementáltam a keretküldésre. Az első megvizsgálja, hogy van-e hely a pufferben, ha igen, akkor elküldi az átjárónak a keretet, és a visszatérési érték jelzi a sikerességet. A második verzió csak az után tér vissza, hogy lett hely a pufferben, és el lett küldve az átjárónak az adat. A harmadik verzió pedig egy paraméterként megadott időkorlátig

próbálkozik az átjárónak elküldéssel, illetve utána a küldés visszaigazolásának megvárásával.

7.2.2 Soros port kezelés

A .NET keretrendszer `System.IO.Ports` névterében található `SerialPort` osztály egy könnyen használható implementációt nyújt a soros portok kezelésére. A virtuális soros port esetén a megfelelő konfigurációhoz csak az adott COM csatorna neve a lényeges. Rendelkezésre áll egy esemény, melyre feliratkozva értesülhetünk a soros porton érkezett adatról. Az interfész konstruktorában inicializáljuk fel és nyitjuk meg a megadott COM portot, valamint küldjük el a hozzáférési parancsot az átjárónak (hiba esetén kivételt dob).

Fontos szempont volt a többszálúság követelményeinek kielégítése, vagyis olyan megvalósítás alkalmazása, mely szinkronizációs objektumok segítségével biztosítja az elvárt működést többszálú környezetben is. Az egyik veszélyforrás a soros porton keresztüli adatküldés, itt garantálni kell, hogy a csomagok sorrendhelyesen kerülnek a kimenő pufferbe, márpedig ha egyszerre több szálon írhatjuk a puffert, akkor a csomagintegritás sérülhet. A .NET keretrendszerben többféle szinkronizációs megoldást lehet használni, ezek közül a legegyszerűbb és leggyorsabb a `lock(object o){}` utasítás használata, mely egy tetszőleges objektumot használ mutex-ként, vagyis egyszerre csak nulla vagy egy szálhoz lehet rendelve. Ha az objektum jelzett, akkor a többi szál várakozó állapotba kerül a blokk előtt, amíg hozzá nem jut az objektumhoz. (Ezt a megoldást alkalmazom a küldési FIFO kezelése során is.)

A soros porton keresztüli adatfogadás eseményvezérelt, a soros port `DataReceived` eseményére feliratkozva történik. Ez az eseménykezelő függvény kiolvassa a rendelkezésre álló adatmennyiséget a port bejövő pufferéből, összefűzi az eddig megérkezett, de még nem értelmezhető bájtokkal. Ezután az új adatok segítségével a teljes terjedelmű csomagokat és visszajelzéseket feldolgozza. Csomag érkezése esetén megvizsgálja, hogy CAN keret vagy hiba, és annak megfelelően létrehoz egy új objektumot, majd meghívja a saját megfelelő eseményét. Az interfész a következő eseményeket nyújtja:

```
public class GatewayIF : IDisposable
{
      public event Can.FrameHandler FrameReceived;
      public event Can.FrameHandler FrameSent;
      public event Can.ErrorHandler ErrorReceived;
      …
}
```

7.2. ábra: Kliens átjáró eseményeinek áttekintése

A `FrameReceived` és `ErrorReceived` eseményeket egy hozzájuk tartozó átjáró által küldött csomag hatására süti el az interfész. A `FrameSent` esemény kivitelezéséhez van szükség a keret küldési FIFO-ra az interfészen belül, melyből visszaigazolás fogadása esetén a legelső elemet kivéve tudjuk elsütni az eseményt.

7.3 Átjáró konfigurációs ablak

A grafikus felhasználói felület koncepcióját úgy alakítottam ki, hogy a program egy kezdőképernyővel indít, melyen az átjáró kezdeti konfigurációja állítható be. A szükséges beállításokat a 7.3. ábra szemlélteti. A legfontosabb paraméter az átjáróhoz tartozó COM port, ez a legördülő lista tartalmazza az éppen elérhető soros kommunikációs portokat. A lista indításkor, valamint egérrávitel hatására frissíti a listát a `SerialPort.GetPortNames()` statikus metódus alapján (ha nincs jelen aktív port, a None felirat jelenik meg). Ezután kell megadni az átjáróhoz tartozó jelszót. Mivel a jelszó tekinthető kommunikációt szinkronizáló karaktersorozatnak is, ezért egy jövőbeli revízióban nem a felhasználónak kell majd megadnia azt, hanem egy konfigurációs fájlban lesz tárolva. A CAN konfigurációnak ezen a felületen egy korlátozott, csak a főprogram által használt funkcióválasztéka elérhető. A keretszűrők

7.3. ábra: Az átjáró konfigurációs párbeszédablaka

beállítását is csak a főablak menüjében lehet elérni, az alapértelmezett konfigurációban egy minden keretet átengedő szűrő kerül beállításra.

A konfigurációs párbeszédablakon a beállítások elvégzése után az OK gombra kattintva a program megvizsgálja a paraméterek helyességét, majd létrehoz egy új átjárót. Ha az átjáró megfelelő visszajelzéssel elfogadta az elküldött jelszót és a CAN konfigurációt, akkor a `DialogResult` tulajdonságát `OK`-ra állítja, egyébként `Retry` lesz az értéke. Az átjáró bezárásakor ez a tulajdonság `Cancel` értéket vesz fel.

Annak érdekében, hogy a főablak betöltése előtt jelenjen meg ez az ablak, viszont későbbi újrakonfiguráció céljából a legutóbbi beállításokkal lehessen megjeleníteni, a főablak konstruktorában hozom létre és jelenítem meg, mint dialógusablak. A dialógusablak `DialogResult` értékétől függően a következők történnek:

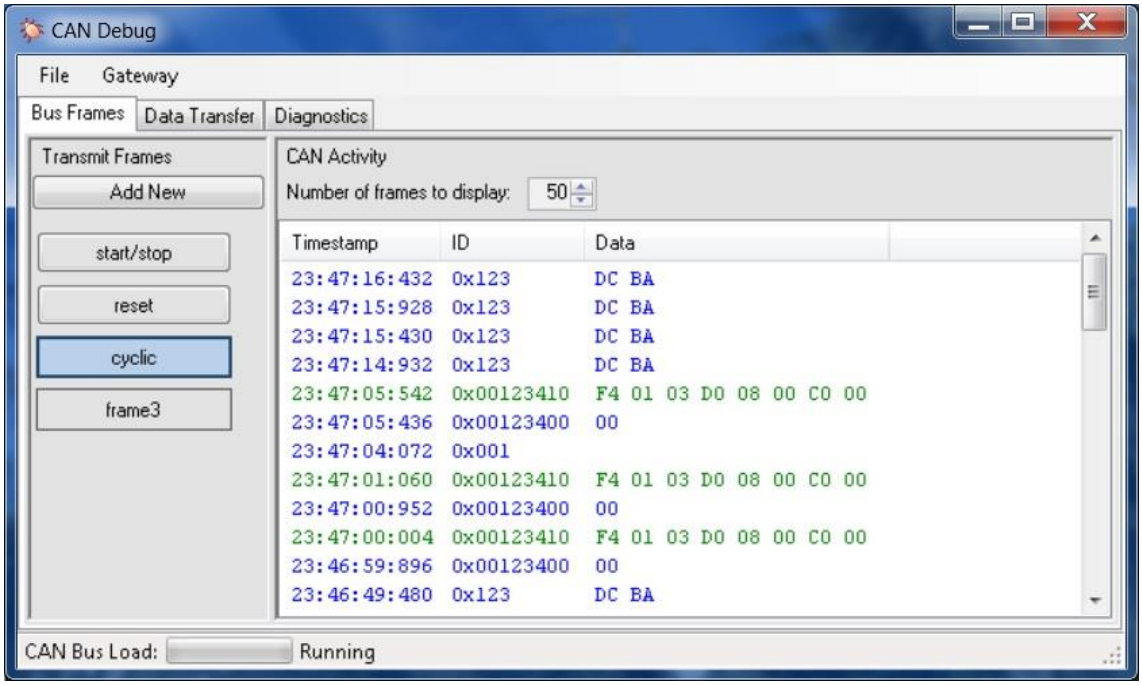
- `OK` esetén lefut a konstruktor további része és megnyílik a főablak,
- `Retry` esetén újra megnyitja a párbeszédablakot,
- minden egyéb érték esetén pedig azonnal visszatér.

Utóbbi esetben azt lehetett tapasztalni, hogy megjelent egy teljesen üres ablak, mivel a konstruktorból való egyszerű visszatérés esetén létrejön az ablak. Mivel nem akartam kivételdobással megoldani ezt a problémát, ezért a következő megoldást alkalmaztam: a Form `Load` eseményére feliratkozik a `Close()` metódusa, melynek eredményeképp az ablak betöltése pillanatában lefut az azt bezáró függvény.

7.4 Főablak

A főablakot egy többcélú felhasználói interfésznek készítettem, mely elsősorban az átjáróra épülve nyújt bizonyos szolgáltatásokat, melyeket a későbbi fejezetekben mutatok be. Először szeretném áttekinteni a felhasználói felület (GUI) alapvető működését. Először is, a szoftver kialakítása során szimpatikusabbnak láttam az egyes programmodulokat külön tab-okra helyezni, minthogy külön ablakokban jelenjenek meg, mert így kompaktabb és könnyebben kezelhető a felület meglátásom szerint. Hogy könnyebben áttekinthető kódot írjak, az egyes tab-oknak külön forrásfájlt biztosítottam, melyekben ugyanazt az osztályt bővítem tagváltozókkal és tab függvényekkel (az osztály felbontására több blokkba a `partial` kulcsszó segítségével van lehetőségem). A jelenlegi állapotok szerint három fő funkciócsoport van megvalósítva, a keret szintű

kezelésre szolgáló *Bus Frames* tab, (a kiforrott funkcióval nem rendelkező *Data Transfer* tab) valamint az UDS autóipari diagnosztikai szabványnak megfelelő hitelesítést és titkosított adatátvitelt biztosító *Diagnostics* tab. A főmenü egyelőre csak az átjáró újrakonfigurálását, valamint az átvitel megállítását és folytatását végző menüpontokat tartalmazza. Az ablak a következőképpen jelenik meg:



7.4. ábra: CAN adatfolyam megjelenítése a kliensben

Az ablak alján található egy státuszsáv, mely különböző értesítéseket jelenít meg a CAN busszal kapcsolatosan. Balról jobbra haladva az első egy `ProgressBar`, ami a CAN busz arányos terheltségét jeleníti meg. Ezután pedig a busz mintavételezés állapota (*Running/Stopped*) látható.

7.4.1 CAN adatfolyam megjelenítés

A program az átjáró interfésztől minden információt a három darab eseményén (7.2. ábra) keresztül nyer. Ezen események kezelőfüggvényeinek három fő feladata van:

- adatkeret esetén kiszámolni, hibakeret esetén becslést adni az adott CAN esemény bitidőben mért hosszára,
- megjeleníthető formátumra alakítva felvenni az eseménylistára,
- jelezni a megjelenítést frissítő szálnak az új adat meglétét.

A *Bus Frames* tabon lehet egyrészt megfigyelni a CAN busz aktivitását fordított időrendi sorrendben (azaz a legfrissebb esemény található legfelül) a tab jobb felén, másrészt létre lehet hozni küldésre szánt kereteket, melyek ezután gombokként megjelennek a bal oldali listán, és kattintásra küldésre kerülnek. A keretek megjelenítésénél RGB színkódolás különbözteti meg a háromféle eseményt: kékkel a küldött keretek, zölddel a fogadottak, vörössel pedig a hibakeretek vannak kijelezve.

Az időbélyeget a 7.1. ábrán is látható `Can.Event.SetTime()` metódus állítja be, amikor átjáró csomag alapján jön létre egy új keret vagy hiba objektum, vagy az átjáró beállítja a küldött keret `IsSent` tulajdonságát. Ez a metódus a `DateTime.Now` statikus *property*-t használja fel az időbélyeg beállítására. Ez az időadat az adott futtató gép lokális idejét szolgáltatja, és az adott gép hardverétől és az aktuális processzorterheltségtől is függően képes – a saját tapasztalataimból kiindulva – 1-10 ms közötti felbontásban aktuális időt szolgáltatni.

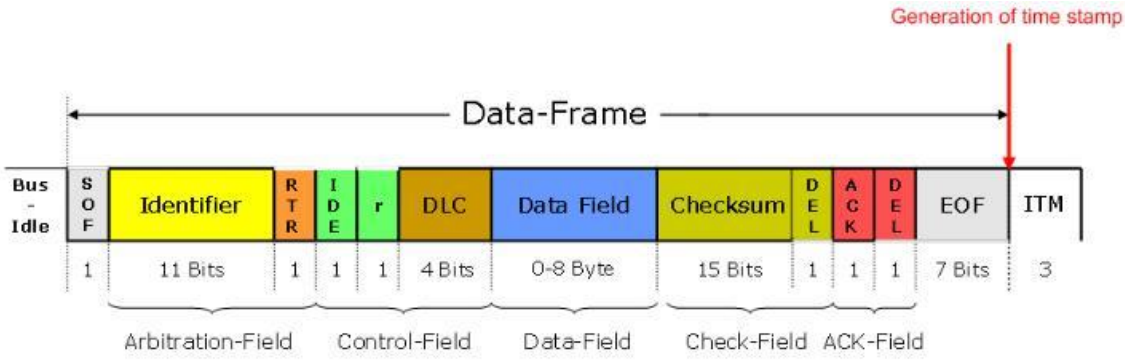
A keretek azonosítójának és adatának kijelzése fixen hexadecimális formátumban történik, nem csak az értékes karakterek, hanem mindegyik ki van írva. Ennek segítségével lehet a *Standard* ID-t is megkülönböztetni az *Extended*-től, előbbi háromjegyű, utóbbi nyolc. A hibák megjelenítésekor a hiba típusa szövegesen kerül kiírásra a *Data* oszlopba, az *ID* mezőt üresen hagyva.

7.4.2 Busz terheltség jelzés

A CAN busz kihasználtságának hozzávetőleges arányát egy `ProgressBar` jelzi. A terheltség számításához egy külön szálat használok, mely 200 ms-onként frissíti egyrészt a terheltségi kijelzőt, másrészt a kijelzett eseménylistát. Az időintervallumon belül érkezett keretek bitjeinek összegét szorozva a beállított bitidővel, majd ezt elosztva az időintervallummal kapjuk meg a busz terheltség aktuális arányát. A keret bitek számítása a következőképpen zajlik az eseménykezelő függvényekben:

- adatkeret esetén a 7.5. ábra alapján 36 darab fix bitmező van egy keretben, melyhez hozzáadódnak az adatbájtok bitmérete és az *ID* mező mérete, mely típustól függően 11 vagy 11 + 18 + 2 bitet jelent,
- hibakeret esetén csak becsülni lehet a keretbiteket, hiszen nem ismert, hogy melyik adatkeret adása zajlott éppen, és annak melyik pontján lépett fel a hiba. Itt így egy becsült értékkel növeljük a bitidő számlálót, 30-nak vesszük a keretcsonk méretét, valamint tudjuk, hogy a hibakeret

(*Error Active* módban) 17 és 23 bitidő között mozog, ezért ez 20-nak vesszük, tehát a hibaesemény idejét 50 bitre becsüljük.

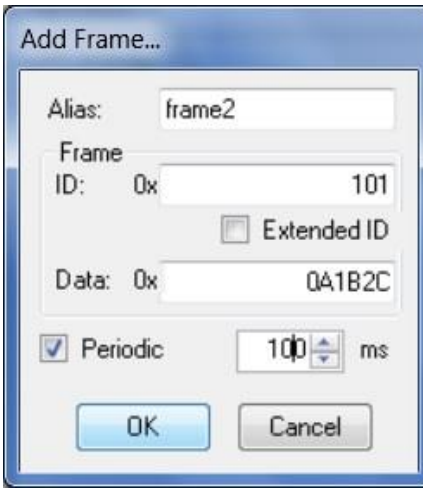


7.5. ábra: CAN 2.0A adatkeret felépítése [42]

A bitidő számítása a lehetséges Baudrate beállítások fényében igen egyszerű, 1 Mbit/s esetén 1 µs, a többi bitsebesség pedig ennek a fele, negyede, nyolcada, a többi bitidő tehát értelemszerűen 2, 4, 8 µs lehet. Az eseménykezelők az előbbieknek megfelelően növelnek egy bitszámlálót, melyet a szálunk a terheltség kiszámítása után lenulláz. A `ProgressBar` maximum értékét 200.000-nek beállítva nem kell az eltelt intervallummal osztani, egyszerűen a kitöltés mértéke egyenlő a bitszámláló és a bitidő szorzatával. Ez a számítási módszer csak szimbolikus jelleggel tudja reprezentálni a buszterheltséget, de egyrészt a hibák esetén fellépő bizonytalanságok, másrészt a körülményesen számításba vehető *bit stuffing* (NRZ kódolás miatti bitbeszúrás) miatt a pontosság növelése nem ésszerű.

7.4.3 Keretküldés megvalósítása

Annak érdekében, hogy egy egyszerű, manuális felületen lehessen a buszra kereteket adni, a tab bal oldalán elhelyeztem egy `FlowLayoutPanel` vezérlőt, mely képes más vezérlőket (jelen esetben gombokat) dinamikusan tárolni és elrendezni, valamint egy gombot, mellyel ebbe a panelbe lehet új elemeket létrehozni. Az *Add New* gombra kattintva a 7.6. ábrán szereplő párbeszédablak jelenik meg. Itt kell megadni a keret fantázianevét, amelyet majd a gomb visel, a keret azonosítóját és adatát hexadecimális



7.6. ábra: Küldendő keret létrehozásának párbeszédablaka

formátumban (utóbbinak minden jegyét ki kell írni, ugyanis az alapján tudja meghatározni az adatbájtok számát), valamint azt, hogy a keret egyszeri küldésre, vagy adott periódusidővel való ismételt adásra szánt-e (utóbbi esetben a milliszekundumban mért periódusidejét is meg kell adni). Elfogadáskor az ablak leellenőrzi a paraméterek helyességét, majd a paraméterek alapján létrejön egy `TransmitButton` vezérlő.

A `TransmitButton` egy saját leszármazott `Button` vezérlő, mely tartalmazza a beállított keretet és periódust, valamint két tulajdonság flag-et, az egyik a gomb periodikus voltát jelzi, a másik pedig – amennyiben periodikus – a keretküldés aktív voltát. Mindkét tulajdonsághoz eltérő vizuális megjelenítés társul. Az egyszerű, nyomásra egyszeri keretküldést végző gomb a hagyományos megjelenéssel rendelkezik, míg a periodikus küldést végző gomb lapos, szögletes (lásd 7.4. ábra, a felső kettő az előbbi, az alsó kettő az utóbbi kategóriába tartozik). A vizuális megkülönböztetés célja, hogy azt sugallja, az előbbi típus egy rugós nyomógomb, ami csak a nyomás idejére lesz aktív, míg az utóbbi egy ki-be kapcsológomb, melynek megnyomására állapotváltása történik. A periodikus gomb bekapcsolt állapotát pedig az ábrán látható kék színű kitöltés jelzi.

A gombra történő kattintást a `TransmitButton` vezérlő saját maga kezeli le, a főablak programjának csak a `SendFrame` eseményére kell feliratkoznia. Egyszeri küldésre beállított gomb esetén a kattintásra egyszerűen elsüti ezt az eseményt, periodikus esetben átállítja az `IsActive` tulajdonságát. Amikor aktív üzemmódba vált a gomb, akkor elindít egy háttérszálat, ami a periódusideig várakozik a kilépést jelző szinkronizációs objektum jelzettségére, majd sikertelen várakozás esetén elsüti a `SendFrame` eseményt, és újrakezdi a várakozást. Az aktív üzemmód kikapcsolásakor pedig csak jelzettbe kell állítani a szinkronizációs objektumot. A kódbeli megvalósítást a 7.7. ábra mutatja. A periodikus keretküldés pontossága nem nagyobb 5-10 ms-nál, és természetesen függ az aktív gombok számától, valamint a CAN busz terheltségétől is.

A gomb létrehozása után a programban hozzáadunk egy kontextus menüt, mely a meglévő gombok módosítását és törlését teszi lehetővé, valamint felvesszük a `FlowLayoutPanel` elemei közé és feliratkozunk a keret küldési eseményére. A keretküldést kezelő függvény az átjárónak egy olyan küldési metódusát alkalmazza, mely csak akkor küldi el a keretet, ha az adott pillanatban van hely az átjáró küldő pufferében, a küldés sikerességét pedig a visszatérési értékével jelzi. Sikertelen esetben a főablak lezárja a küldési GUI-t, amíg nem érkezik egy keretküldési visszaigazolás.

```
public class TransmitButton : System.Windows.Forms.Button
{
      private static int activeButtonCount = 0;
      public static int ActiveButtonCount {get{return activeButtonCount;}}
      private bool isPeriodic;
      private Thread periodicThread = null;
      private ManualResetEvent exit;
      private bool active = false;
      public bool IsPeriodic {
            get { return isPeriodic; }
            set {
                  if (!value && isPeriodic) {
                        this.IsActive = false;
                        this.FlatStyle =
                        System.Windows.Forms.FlatStyle.Standard;
                  }
                  else if (value && !isPeriodic){
                        this.FlatStyle =
                        System.Windows.Forms.FlatStyle.Popup;
                  }
                  isPeriodic = value;
            }
      }
      public bool IsActive {
            get { return active; }
            set {
                  if (!isPeriodic) return;
                  if (value && !active) {
                        this.BackColor =
                        System.Drawing.SystemColors.GradientActiveCaption;
                        activeButtonCount++;
                        exit = new ManualResetEvent(false);
                        periodicThread = new Thread(transmitting);
                        periodicThread.IsBackground = true;
                        periodicThread.Start();
                  }
                  else if (!value && active) {
                        this.BackColor =
                        System.Drawing.SystemColors.ControlLight;
                        activeButtonCount--;
                        exit.Set();
                  }
                  active = value;
            }
      }
      public event Can.FrameHandler SendFrame;
      private void transmitting(){
            do {
                  if (SendFrame != null) SendFrame(frame);
            } while (!exit.WaitOne(period));
            exit.Dispose();
      }
      …}
```

7.7. ábra: Küldő gomb osztály állapotváltásai `Property`-kkel megvalósítva

7.4.4 Adatfolyam megállítás, folytatás, törlés

A főablak *Gateway* menüpontja három elemet tartalmaz:

- *Pause*, mely hatására megáll a keretek fogadása, illetve küldése,
- *Resume*, mely hatására folytatódik a keretek fogadása, illetve küldése,
- *Reconfigure*, mely újra megnyitja az átjáró konfigurációs ablakot (7.3).

Az adatátvitel megállításakor az átjáró eseményeiről leiratkozik a főablak, valamint letiltja a keretek küldését, és az ablak státuszsávján kiírja a *Stopped* jelzést, míg az átjáró interfész továbbra is fogadja az adatokat az átjárótól. A folytatás során ugyanezek az események zajlanak le, csak fordítva, és a *Running* felirat jelenik meg.

7.4.5 Kilépés a programból

A program megfelelő bezárása korántsem triviális feladat. Habár a .NET keretrendszer szemétgyűjtője biztosítja azt, hogy a dinamikusan lefoglalt memória felszabaduljon, viszont vannak olyan nem felügyelt erőforrások, melyek használatának végeztével a programnak kell biztosítania az erőforrás felszabadítását. A jelen alkalmazásban ilyen a soros port, amelyet az átjáró interfész használ, ezért ez az osztály implementálja az `IDisposable` interfészt. Ezen kívül még azt is biztosítani kell, hogy a GUI szálon kívüli futó szálak is kilépjenek. Erre a célra bevezettem egy új `ManualResetEvent` szinkronizációs objektumot, melyet az összes háttérszál használ a várakozásai során, és ha ez jelzett állapotba kerül, akkor az összes ilyen szál felébred és kilép (utána az objektumra is `Dispose()`-ot kell hívni). Ez alól kivételek a küldő gombok szálai, melyeket gombok kikapcsolásával lehet kiléptetni.

Egy fontos működésbeli részlete a `SerialPort` osztálynak, hogy a `Close()` metódusa a GUI szálból meghívva *deadlock*-ot képes okozni abban az esetben, ha a `DataReceived` eseményének szálán `Invoke()` hívás történik. A `Close()` során ugyanis a szál megvárja, míg az eseményszál lefut, az viszont arra várakozik, hogy az `Invoke()` művelet sikerességét visszajelezze a GUI szál. Kétféle megoldás van erre a problémára [43]:

- Az eseménykezelők `Invoke()` hívásait le kell cserélni az aszinkron működésű `BeginInvoke()` hívásokra.
- A `Close()`-t a GUI száltól különböző szálon kell meghívni.

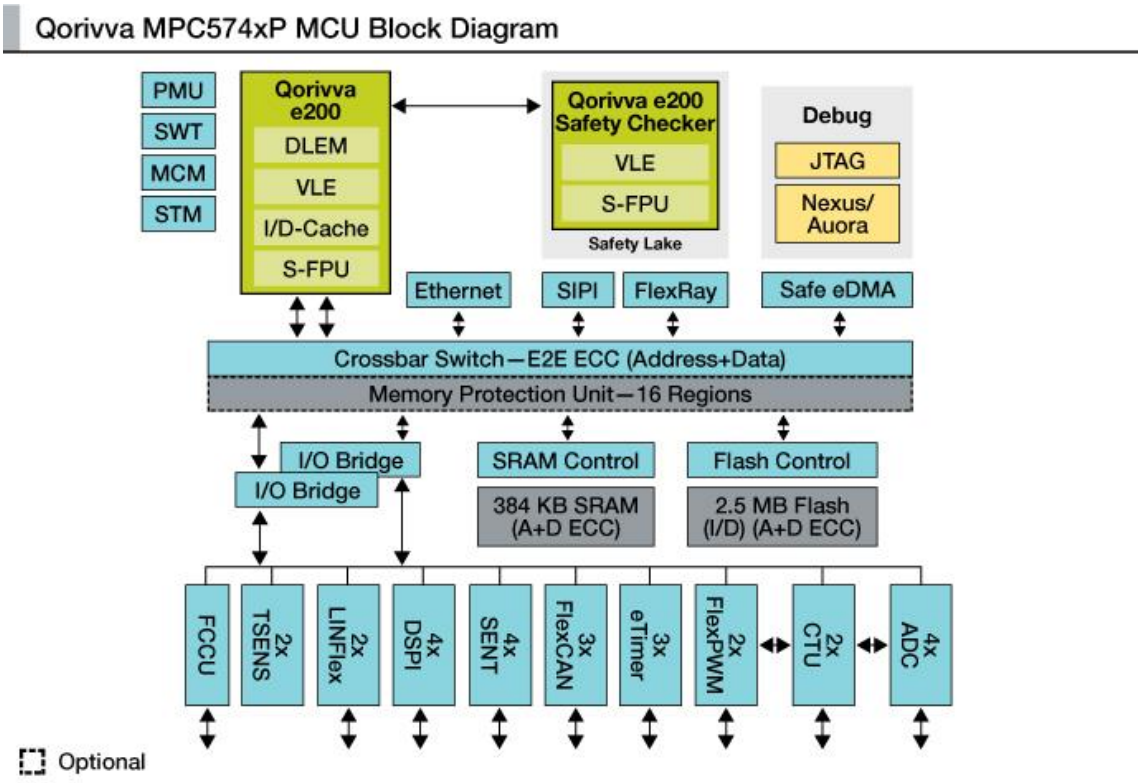
8 Hitelesítés és rejtjelezés autóipari platformon

A tesztelt kriptográfiai elemkészlet és a szükséges diagnosztikai architektúra birtokában el lehet kezdeni kifejleszteni ezek felhasználásával egy biztonságos kommunikációs stack-et az ECU-n és a PC-s kliensen. Előbbihez a TKP által jelenleg használt autóipari platformot, a Freescale Qorivva MPC5744P típusú mikrokontrollert használtam, melynek megismertem az elérhető hardveres biztonsági képességeit, ezek alapján egy, az adott feltételek mellett a legalkalmasabbnak tűnő biztonságú védelmi réteget implementáltam a diagnosztikai kommunikáció fölé. Ezzel a réteggel lehetőség nyílik a mikrovezérlő tetszőleges memóriaterületéről adatot olvasni vagy oda adatot írni titkosított és/vagy hitelesített módon, mindezt az UDS szabványnak [44] (és az azt megvalósító DCM implementációnak) megfelelő módon. Külön funkcióként a diagnosztikai kliens képes beolvasni a mikrovezérlő fordítási kimeneti fájlja alapján megalkotni a kívánt memóriaképet, és azt biztonságosan feltölteni a kontroller flash-ébe, illetve ugyanez a funkció elérhető visszafelé is (a vezérlő kódjának letöltése a PC-re).

8.1 A hardver platform tulajdonságai

A Freescale MPC5744P egy 200 MHz-es órajelű, kétmagú, Power Architecture felépítésű (8.1) nagy rendelkezésre állású célalkalmazásokra fejlesztett mikrokontroller. A dupla processzormagú kialakítás nem a teljesítménynövelés érdekében történt, hanem hogy a programvégrehajtás helyességét valósidőben ellenőrizze a második mag, így a kontroller alkalmas ISO 26262 ASIL-D besorolású biztonságkritikus rendszerekben való felhasználásra. Bár a funkcionális biztonságra nagy hangsúlyt helyeztek a tervezés során, a támadások elleni védelem nem volt a tervezési szempontok között, így a mi szempontunkból elhanyagolható hardveres támogatás van jelen. [45] (Még az átjárónak használt STM32 kontrollerbe is sikerült legalább egy TRNG perifériát betervezni). Az egyedüli hardveres védelmet a flash memória és a debug interfészek élvezik, ezeket védett módba lehet állítani, így lezárva őket a külvilágtól. A védett módot 64 bites jelszóval lehet feloldani, viszont a jelszó feltöltését védtelen csatornán keresztül kell feltölteni, ami lehetőséget ad annak lehallgatására. Összefoglalva tehát a teljes védelmi

funkcionalitás kivitelezése a szoftverre hárul, amelyet ezen hardveres tulajdonságok fényében kell megtervezni.



8.1. ábra: Freescale MPC574xP blokkvázlata [46]

8.2 Védelmi rendszer koncepció

A védelmi rendszer koncepciójának az alapfeltevése az volt, hogy a már meglévő DCM implementáció felhasználásával, annak szolgáltatásainak segítségével, illetve kibővítésével valósítsuk meg a diagnosztikai kliens hitelesítését. Az UDS szabvány *SecurityAccess* (27h) szolgáltatása pont ezt a célt szolgálja. A kétlépéses hozzáférési metódus a kihívás-válasz séma alapján működik:

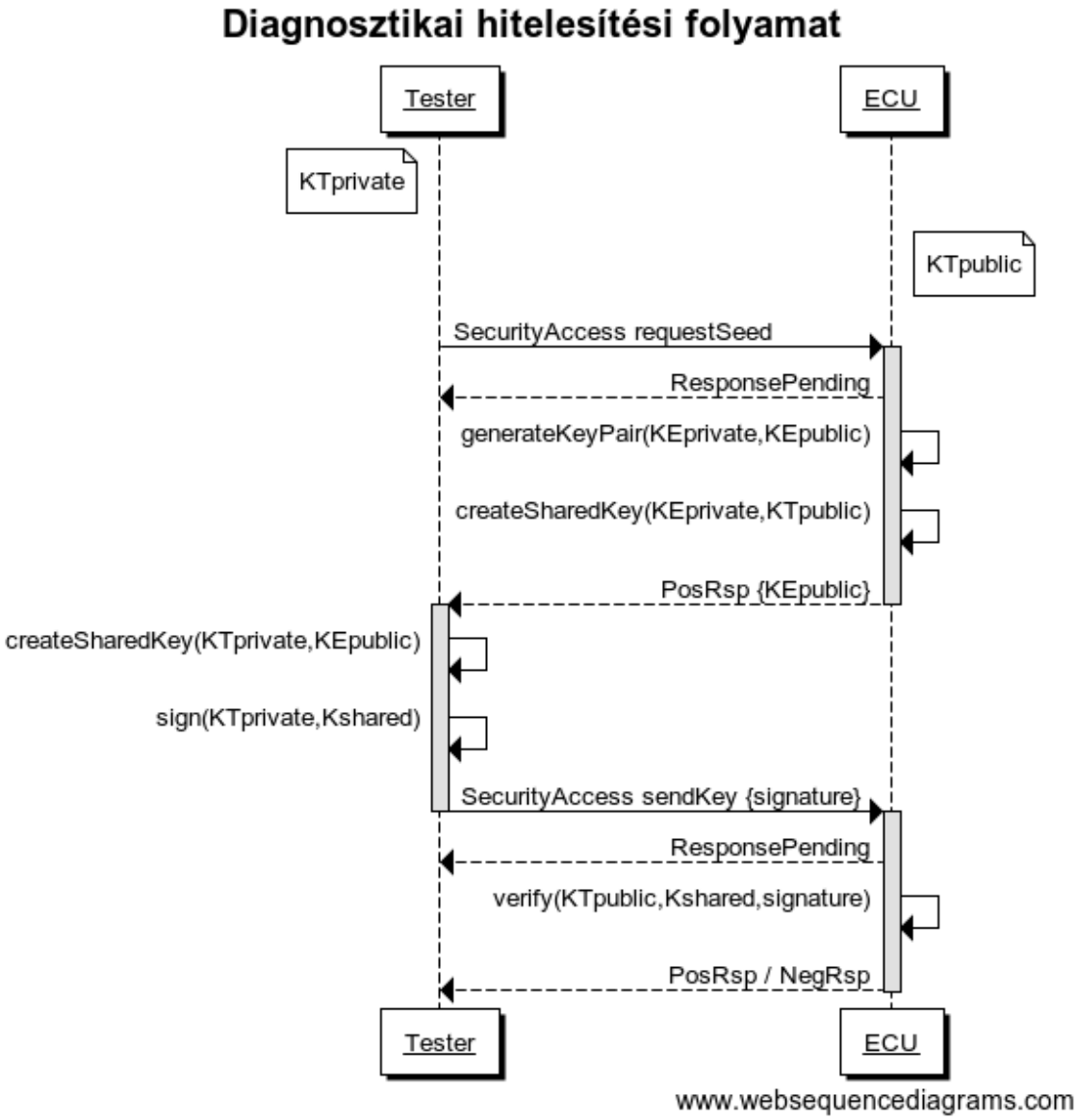
1. A diagnosztikai kliens (*tester*) az első körben kér egy *seed*-et (*nonce*-t) a szerver ECU-tól, melyre az ECU a válaszban visszaküld egy generált véletlen számot.
2. A kliens elvégzi ezen az egyedi számon az őt hitelesítő műveletet, majd annak eredményét visszaküldi, mint hozzáférési kulcs. Ezt az ECU leellenőrzi a saját műveletével, és ha a saját eredménye megegyezik a kliens által küldött kulccsal, akkor engedélyezi a hozzáférést, egyébként elutasítja.

Ezt a sémát kriptográfiai eljárások segítségével kétféleképpen lehet megvalósítani:

- Fix szimmetrikus kulcs használatával, mely mindkét fél számára rendelkezésre áll, és a seed-en a kliensnek egy MAC műveletet kell a szimmetrikus kulcs segítségével végrehajtania.
- Fix aszimmetrikus kulcspár használatával, mely során a seed-et a kliens a privát kulccsal aláírja, az aláírását pedig az ECU a publikus kulcs segítségével hitelesíti.

A korábbiakban elég részletesen megtárgyaltuk, hogy egy olyan fedélzeti hardveren, melyben nincs HSM, nem lehet biztonsággal meggyőződni a futó program eredetiségéről és hitelességéről, valamint a memóriájában tárolt bármilyen adat sincs biztonságban. Ez egyrészt azt jelenti, hogy felesleges lenne egy olyan eljárást választani, amelyben az ECU is hitelesíti magát a kliens felé, mielőtt érzékeny adatok átvitele történhetne. Másrészt a szimmetrikus kulcs használata egyértelműen aláássa az eljárás biztonságosságát, hiszen azt előre kiolvasva a kontroller memóriájából már bárki képes a hozzáférésre. Éppen ezért én az aszimmetrikus aláírási sémát választottam.

További megfontolandó dolog volt ezen a metóduson kívül az adatátvitel megoldása, mely esetén szintén igény tartunk a titkosságra és/vagy a hitelességre. Az 5.3-ban bemutattam, hogy az aszimmetrikus rejtjelezés teljesen alkalmatlan nagyobb adatcsomagok hatékony titkosítására az arányaiban hatalmas számítási ideje miatt. Viszont azt is rögzítettük, hogy nem lehet fix szimmetrikus kulcsokat használni. Ezen okokból a hozzáférési metódust úgy módosítottam, hogy egy ECDH kulcscsere algoritmus is lezajlik a két eszközön úgy, hogy a séma felépítése változatlan marad. A teljes hozzáférési algoritmus a következőképpen zajlik:



8.2. ábra: A diagnosztikai hitelesítés folyamatábrája

Kezdetben a diagnosztikai kliens rendelkezik egy fix privát kulccsal (*KTprivate*), melynek publikus párja (*KTpublic*) az ECU birtokában van. A seed kérését követően az ECU létrehoz egy ideiglenes kulcspárt (egy PRNG alapján létrejött privát kulcsot (*KEprivate*) és az abból számolt publikus kulcsot (*KEpublic*)), majd az új privát kulcs és a fix publikus kulcs alapján létrehozza a megosztott szimmetrikus kapcsolatkulcsot (*Kshared*). Az új publikus kulcs (*KEpublic*) kerül visszaküldésre a kliensnek, mint seed. Ez alapján a publikus kulcs alapján, valamint a fix privát kulcs alapján a kliens is létrehozza a szimmetrikus kapcsolatkulcsát, ami megegyezik az ECU által számítottal. Ez a már ismertetett ECDH kulcscsere algoritmus, azzal a különbséggel, hogy jelen esetben csak az egyik kulcspár született nonce alapján, a

másik kulcspár fixen rögzített. Ez a kapcsolatkulcs lesz a titkos kulcs a továbbiakban a biztonságos adatátvitel során.

Tehát idáig létrejött egy olyan kapcsolatkulcs, melyet csak hiteles kliens tud létrehozni, vagy olyan támadó, amely képes kiolvasni az ECU flash és/vagy RAM memóriáit. Amennyiben a memóriák befolyásolása más biztonsági szintet képes képviselni, mint az azokba való betekintés képessége, márpedig ez egy lehetséges forgatókönyv jelen esetben is, úgy a hozzáférés második fele ezt ki is használja. A kliens feladata a kapcsolatkulcs létrehozása után, hogy azt aláírja a saját fix privát kulcsával (mivel a kapcsolatkulcsot ugyanaz az ECC rejtjelező hozta létre, mint ami azt aláírja, ezért a kulcsot teljes hosszában felhasználja az aláíró algoritmus). Ezt az aláírást küldi át az ECU-nak, mint hozzáférési kulcs, melynek hitelességét az (a megegyező értékű kapcsolatkulcsával és a fix publikus kulcsával) képes igazolni vagy cáfolni. Ezzel a lépéssel már garantált, hogy csak akkor lehet a diagnosztikai biztonsági szint műveleteihez hozzáférni, ha valaki rendelkezik a fix privát kulccsal.

A rendelkezésre álló elliptikus görbék közül a legnagyobb biztonságút szerettem volna kiválasztani, így [27] alapján a három szabványosítási szervezet (SECG, NIST, ANSI X9.62) által is elfogadott NIST P-256 görbét állítottam be az ECC rejtjelezőben.

A blokkrejtjelezési módok esetén a titkos kulcson felül szükség van egy kezdeti értékre (IV) is. Ennek az AES használata miatt 16 bájt hosszúnak kellett lennie, én a biztonságosabb megoldást választottam, és a titkos kulcs első 16 bájtját XOR-oltam a második 16 bájttal (így nem nyilvános IV értéke). Mivel az IV nem használható fel egynél több titkosítási folyamatnál, ezért minden új titkosításnál inkrementálom az értékét.

8.3 Pszeudovéletlen-szám generátor

Mivel a használt mikrovezérlő nem rendelkezik TRNG perifériával, így a kulcsgeneráláshoz elengedhetetlen véletlenszám-generátort is szoftveresen kellett megvalósítani. A leggyakrabban használt PRNG sémát alkalmaztam: egyrészt adott egy entrópia puffer, mely kezdeti érték alapján egy kriptográfiai eljárás ciklikusan dolgozik, és ennek kimenete a PRNG érték. Az entrópia pufferben gyűjtjük a rendszerben fellépő „entrópiát”, vagyis a rendszer bizonytalan állapotainak értékét, bitsorozatok formájában. A tesztprogramban erre egyetlen nagyfrekvenciás szabadonfutó időzítőt (System Timer Module) használtam, melyet mindig az összes rendszertaszk lefutása után

mintavételeztem, így próbálva meg a tesztrendszer összes jelenlévő entrópiáját akkumulálni. Nyilván egy valós alkalmazásban több tucat megfelelő entrópia forrást is lehet találni (például zajterhelt analóg csatornán konvertált alsó helyi értékű bitek, kommunikációs üzenet időbélyegek, stb.). Az entrópia puffer méretét akkorára választottam, hogy feltöltött állapotban egy 256-bites AES-CTR kontextust fel tudjon tölteni kulccsal és kezdeti értékkel (ez 48 bájtot jelent). A véletlen bájtsorozatot nyújtó függvényben történik az entrópia puffer alapján a kulcsütemező indítása, majd a szükséges mennyiségben nulla bájtokat titkosítunk az AES-CTR rejtjelező módban, és a rejtett szöveg bájtjait szolgáltatjuk, mint véletlen szám. Egy adott bájtmennyiség rejtjelezése után a kulcsütemezés újrakezdődik a frissült entrópia puffer alapján. Amennyiben még nem gyűlt össze elég entrópia, a függvény hibával tér vissza, ekkor kénytelen várni a felsőbb réteg.

8.4 DCM feletti védelmi réteg

Az AUTOSAR szoftverarchitektúrának és a már összeállított CAN alapú DCM diagnosztikai kommunikációs stack-nek köszönhetően a beágyazott szoftver fejlesztése során nem kellett foglalkoznom mással, mint hogy a DCM konfigurációban a kiválasztott biztonsági szint (1-es) paramétereit beállítsam, valamint megalkossam az alábbi *callout* függvényeket, melyeket a DCM az adott diagnosztikai szolgáltatás kérése esetén meghív:

```
Std_ReturnType SecurityAccess_DcmDspSecurityLevel_1_GetSeed(
             Dcm_OpStatusType OpStatus,
             uint8 * Seed,
             Dcm_NegativeResponseCodeType * ErrorCode);
Std_ReturnType SecurityAccess_DcmDspSecurityLevel_1_CompareKey(
             uint8 * Key);
Std_ReturnType Dcm_ProcessRequestDownload(Dcm_OpStatusType OpStatus,
             uint8 DataFormatIdentifier,
             uint32 MemoryAddress,
             uint32 MemorySize,
             uint32 * BlockLength,
             Dcm_NegativeResponseCodeType * ErrorCode);
Std_ReturnType Dcm_ProcessRequestUpload(Dcm_OpStatusType OpStatus,
             uint8 DataFormatIdentifier,
             uint32 MemoryAddress,
             uint32 MemorySize,
             Dcm_NegativeResponseCodeType * ErrorCode);
Std_ReturnType Dcm_ProcessRequestTransferExit(Dcm_OpStatusType OpStatus,
             uint8 * ParameterRecord,
             uint32 ParameterRecordSize,
             Dcm_NegativeResponseCodeType * ErrorCode);
```

```
Dcm_ReturnWriteMemoryType Dcm_WriteMemory(Dcm_OpStatusType OpStatus,
            uint8 MemoryIdentifier,
            uint32 MemoryAddress,
            uint32 MemorySize,
            uint8 * MemoryData);
Dcm_ReturnReadMemoryType Dcm_ReadMemory(Dcm_OpStatusType OpStatus,
            uint8 MemoryIdentifier,
            uint32 MemoryAddress,
            uint32 MemorySize,
            uint8 * MemoryData);
```

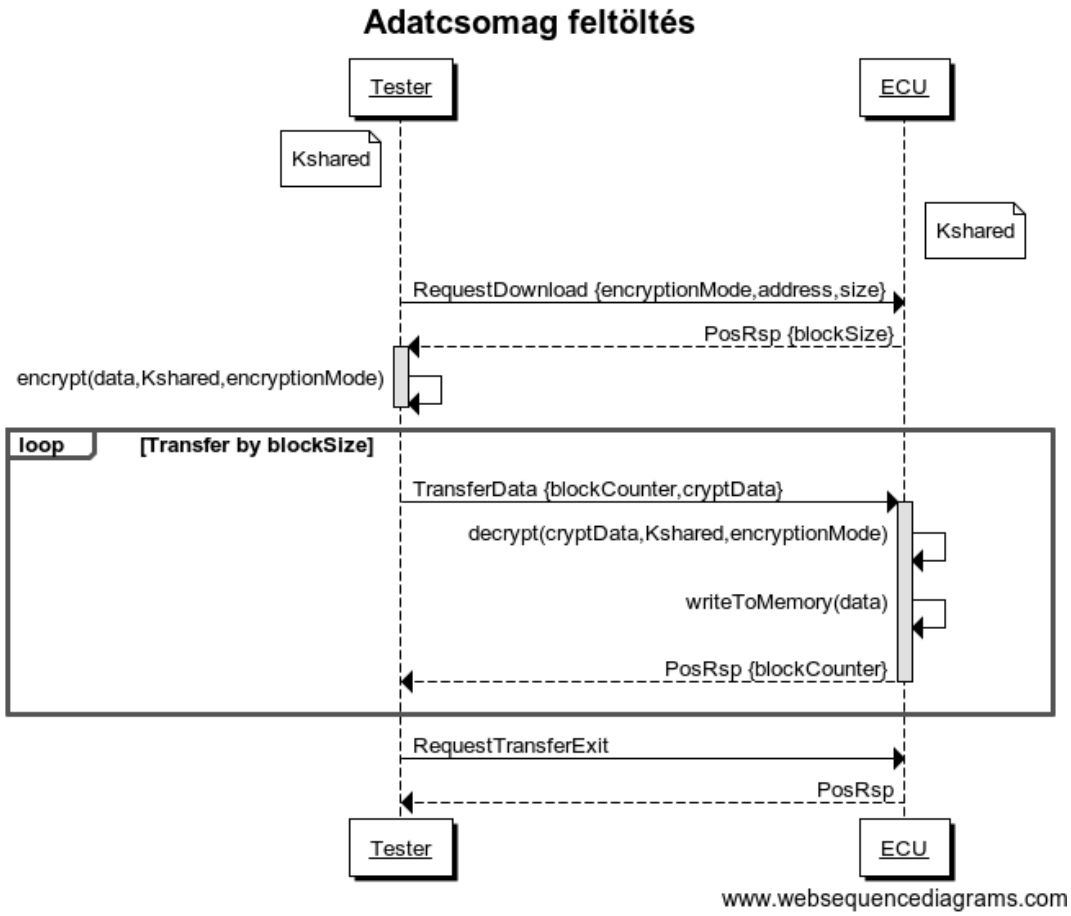
Az első két függvény feladatát már 8.2-nél bemutattam, így ezekre a következőkben nem térek ki. A DCM konfigurációja során az alábbiakat kellett beállítani:

- A kiválasztott biztonsági szinthez tartozó seed és kulcs méretét bájtban.
- A megengedett újrapróbálkozások számát.
- A bootolás utáni és az újrapróbálkozások közötti várakozás időmértékét.
- A használandó UDS szolgáltatások *session* szintjét *extendedProgrammingSession*-ra kellett állítanom.
- E szolgáltatások megengedett biztonsági szintjét be kellett állítanom a kizárólagos 1-es szintre.

A `Dcm_ProcessRequestDownload` és `-Upload` függvényeket hívja meg a DCM, amikor egy hozzá tartozó adatátviteli kérést fogad. Itt a kérésből kinyert `DataFormatIdentifier` változó bájt értéke alapján történik az adatátvitel során használandó tömörítési (felső négy bit) és rejtjelezési eljárás kiválasztása. Ez azt jelentette, hogy – a 0 érték nyílt szöveghez rendelése mellett – 15 féle titkosítási sémát lehetett megvalósítani ebben a rétegben, ami gyakorlatilag azt jelentette, hogy a kriptográfiai könyvtár összes blokkrejtjelezési, MAC és hitelesített rejtjelezési algoritmusát fel lehetett használni legalább egy kulcsméret mellett. Ezekben a függvényekben kell rögzíteni az eljárás típusát, a kérés memóriahosszát és a kezdőcímet, továbbá a szükséges kriptográfiai kontextust inicializálni kell.

A következő lépés a konkrét adatátvitel fázisa, mely során a DCM-től függő fix méretű csomagok átvitele történik az UDS *TransferData* szolgáltatásának megfelelően. Ezekről a csomagokról a `Dcm_ReadMemory` és `Dcm_WriteMemory` függvényeken keresztül értesít a DCM. Itt azzal a problémával kellett szembenéznem, hogy üzenethitelesítés használata esetén az üzenet hosszabb lesz, mint a memóriába szánt vagy a memóriából

jövő adat mérete, amelynek kezelésére a DCM nincs felkészítve. Így az átvitel kezdetekor letárolt méret és kezdőcím változók alapján kell tájékozódni, és egy külön globális változó segítségével nyomon követni a valós eddig felhasznált memóriaméretet. A MAC használata során azt kellett eldöntenem, hogy a hitelesítő *tag*-et a teljes memóriaterületre számítva, az utolsó csomagban elhelyezve küldjem el, vagy pedig minden átvitt csomag külön MAC-et kapjon. Az előbbi eset kivitelezési szempontból egyszerűbb, és az átvitel is gyorsabb, viszont felveti a kérdést, hogy mi legyen a sorsa az átvitt adatmennyiségnek mindaddig, amíg nem érkezett meg a hozzá tartozó *tag*. Ha már az átvitel során letárolódik a memóriába, azzal potenciálisan felülír más adatokat, és sikertelen hitelesítés után már csak a memória törlésére van lehetőség, visszaállításra nem. Ideiglenes pufferben tárolás esetén biztosítani kell a megfelelő méretű RAM memóriát, amellyel elfogadható megoldást teremtünk, ekkor viszont az átviteli kérés kezdeményezésekor az ilyen rejtjelezési módoknál vissza kell utasítani az átvitelt, ha az nagyobb memóriaterületet kíván használni, mint a puffer mérete. A csomagonként küldött *tag* a DCM által használt puffer méretétől függően alkalmazható. Kis puffer esetén nagyban megnöveli a számítási igény miatti késleltetést, továbbá az átvitel hasznos sávszélességét is lecsökkenti. Ha azonban a puffer abban a nagyságrendben van, melyre a saját puffert méreteznénk, akkor a függvényekben csak a DCM puffer használatával egyből képesek vagyunk a szükséges kriptográfiai műveletekre, melyekből megkapjuk a memóriába szánt nyílt szöveget.



8.3. ábra: Az adatcsomag ECU-ra töltésének folyamatábrája

Az átvitel lezárását a `Dcm_ProcessRequestTransferExit` függvény jelzi, melyben lezajlik a kriptográfiai kontextusok törlése, valamint a globális változók kezdeti állapotba állítása. Amennyiben hibát detektálunk valamelyik függvényben, azt egyrészt a függvény visszatérési értékében, másrészt (ahol erre lehetőség nyílik) az UDS szabványnak megfelelő negatív válasz azonosítót (NRC) lehet megadni, melyben az előre definiált hibaokok közül kiválaszthatjuk az adott hiba természetét legjobban összefoglaló elemet.

8.5 Kliens oldali megvalósítás

A kliens alkalmazásba importálnom kellett 5.2.3 alapján a kriptográfiai könyvtárat, valamint azt alkalmassá kellett tennem arra, hogy az előbb ismertetett diagnosztikai szolgáltatásokat végrehajtsa a megfelelő biztonsági réteg hozzáadásával. A CAN átviteli réteg és az UDS szolgáltatások megvalósításával nem kellett foglalkoznom, ugyanis arra a cégnél már rendelkezésre állt egy megfelelő implementáció, amelyet könnyen tudtam importálni. Két dolog volt hátra, az egyik egy

grafikus felület létrehozása, a másik pedig a 8.2. és 8.3. ábrán szereplő szekvenciák megvalósítása. Összesen négyféle szekvencia kivitelezésére volt szükség:

1. A biztonsági hozzáférés során a *DiagnosticSessionControl extendedDiagnosticSession* kéréssel session-t kellett váltani, majd a 8.2. ábrának megfelelő kéréseket kellett elvégezni.
2. ECU-ra adatfeltöltés során a megadott adaton a kiválasztott kriptográfiai eljárást kellett végrehajtani, majd erre az adatra kellett a 8.3. ábrán látható szekvenciát futtatni.
3. ECU-ról adatlekérés során a megadott mérettel (kiegészítve az esetleges hitelesítő *tag* hosszával) és a kiválasztott kriptográfiai eljárással elküld egy *RequestUpload* kérést, majd fogadja, összefűzi és hitelesíti/dekódolja az adatot.
4. A biztonsági kontextust a *DiagnosticSessionControl defaultSession* kéréssel lehet visszaállítani az alapértelmezett szintre.

Az új funkcionalitás GUI-ját egy külön *Diagnostics* fülön valósítottam meg, a kialakítása a 8.4. ábrán látható, azzal a különbséggel, hogy az első verzióban a fájlnév szövegdoboz helyett egy *RichTextBox* vezérlő volt jelen, melybe feltöltés során egy karaktersorozatot kellett írni, letöltés során pedig a letölteni kívánt bájtok számát, melyet az átvitel befejeztével a program helyettesített a letöltött karaktersorozattal.

Mind a négy művelet végrehajtását külön szálban hajtom végre. A meglévő CAN TP drivernek és az átjáró interfésznek egymás (kerek küldési és fogadási) eseményeire kell feliratkozniuk a megfelelő függvénnyel Az UDS drivernek a CAN TP drivert kell átadni, valamint egy `AddressInfo` objektumot, mely tartalmazza a szükséges küldő (kliens) és fogadó (ECU) címeket, valamint azok leképezésének típusát CAN ID-kre. Az UDS driver használatát az alábbi kódrészlet illusztrálja:

```
UDS.Service serv = new
     UDS.SecurityAccess(UDS.SecurityAccess.Subfunction.requestSeed, 1);
serv.Append(signature);
if (!serv.Request(cantp, ai, diagtimeout, out nrc, out diagdata))
    throw new Exception("Server denied " + serv.Name + " service with " +
     nrc.ToString());
```

Minden szolgáltatás egy `Service` leszármazott osztály, mely saját konstruktorral és saját alfunkció-listával (feltéve, hogy a szolgáltatáshoz tartozik alfunkció) rendelkezik. Ezután a szolgáltatáshoz az `Append`-del kell az esetleges hasznos terhet

hozzáfűzni. A `Request` során zajlik le a kérés-válasz szekvencia a CAN TP segítségével, pozitív válasz esetén a `diagdata` bájttömb tartalmazza a válaszban lévő hasznos (nem protokoll-) információt, negatív esetben megkapjuk az NRC hibakódot.

Egy fontos dolgot viszont nem nyújt az UDS driver, ez pedig a diagnosztikai kapcsolat életjeleként értelmezhető periodikus *TesterPresent* kérés, mely csak arra szolgál, hogy a nem alapértelmezett diagnosztikai kapcsolatszintet életben tartsa. Erre akkor van szükség, ha több másodpercig nem folytatunk UDS kommunikációt az ECU-val, viszont szeretnénk, ha például a biztonsági hozzáférési szintet megőrizné. Ezt a feladatot a CAN kliens keretküldő funkciójával viszont meg lehet egyszerűen valósítani, az ECU címét kell ID-nek megadni, az adatmező pedig a `02 3E 80` hexa bájtsorozat, a periódus pedig 1 másodperc.

8.5.1 Mérési eredmények

Mind a biztonsági hozzáférési eljárást, mind az adatátvitelt teszteltem és mértem az eltelt időt. Utóbbi esetében nem okozott érdemleges késleltetést a kriptográfia alkalmazása a nyílt szöveges átvitelhez képest, ugyanis a vezérlő szoftverében 10 ms-os ciklikus lekérdezéssel és nem megszakítás alapon van kezelve a CAN periféria keretfogadás eseménye.

A hozzáférés lépéseinek időigénye az alábbi:

1. Az első blokk, vagyis a seed kérés elküldése az ECU-nak, majd annak létrehozása, valamint visszaküldése átlagosan 984 ms-ba telt.
2. A kliens 29 ms alatt hozta létre a közös kulcsot, majd írta azt alá.
3. Az aláírás elküldése, majd annak hitelesítése az ECU-ban 615 ms alatt zajlott le.

A teljes hozzáférési folyamat tehát több mint másfél másodpercet vesz igénybe. Összehasonlításképpen 1 kB adat letöltése átlagosan 240 ms alatt zajlott le, míg a hozzáférés során csak kétszer 64 bájt hasznos adat átvitele zajlik, tehát jól érzékelhető késleltetést okoz az aszimmetrikus rejtjelező használata (természetesen ez egy várt eredmény volt).

8.6 Biztonságos firmware frissítés

Az egyik elsődleges célja annak, hogy ezeket az UDS szolgáltatásokat valósítottam meg, az az, hogy ezek felhasználásával lehetőség nyíljon a diagnosztikai interfész sebezhető alapján kritikus sebezhetőség, az ECU szoftverének frissítésének biztonságos megvalósítására. Ehhez csak a kliensben kellett kiegészítéseket tennem, hogy be tudjon olvasni és fel tudjon dolgozni egy fordító kimeneti fájlt, melyből memória felépítési adatstruktúrát alkot, és a fenti szolgáltatások segítségével azt feltölti.

8.6.1 Programkód beolvasása fordítófájlból

A céleszközre fejlesztett beágyazott programból a GHS fordító több kimeneti fájlt hoz létre. Ezek közül az *out* kiterjesztésű fájl egy *elf* formátumú fájl (ez a formátum mindig `0x7F 'E' 'L' 'F'` karaktersorozattal kezdődik), mely gyakorlatilag minden fordítási információt rögzít. Mivel ennek a fájlnak a feldolgozása közvetlenül igen összetett feladat, valamint nekem csak a programmemória tartalmára van szükségem a firmware frissítéshez, ezért találnom kellett egy olyan megoldást, amellyel ki tudom ezt az információt nyerni. A keresett megoldást az *objcopy* [47] program szolgáltatta. Ez egy GNU eszköz, mely object fájlok tartalmát képes átmásolni egy kimeneti fájlba különféle beállításokkal. Ha ezt a programot a `-O ihex "input.out" "output.hex"` argumentumokkal futtatjuk, akkor létrehoz egy Intel HEX32 formátumú fájlt, mely csak a programmemória képét tartalmazza és sokkal könnyebben értelmezhető [48] (többek között a tananyag része is volt). A hex fájl tetszőleges számú hexadecimális formátumú rekordot tartalmazhat, melyek a következő felépítéssel rendelkeznek:

```
:llaaaatt[dd...]cc
```

`:`	A kettőspont a kötelező kezdőkarakter.
`ll`	A rekord adatmezőjének hosszát adja meg bájtban.
`aaaa`	A kezdőcím (alsó két bájtja) az adatmező számára.
`tt`	A rekord típusa.
`dd`	A rekord egy adatbájtja.
`cc`	A rekord ellenőrzőösszege, melyet a rekord bájtjainak összegének LSB-jének kettes komplemenseként kapunk meg.

A rekord típusa többféle lehet, a memóriatartalom kinyeréséhez csak háromféle típust kell kezelnünk:

1. A 00 értékű típus az adatrekord, ami a memória tartalmát írja le egy adott kezdőcímtől.
2. A 01 értékű típus a fájlvége rekord, kötelezően a fájl utolsó sora, üres adatmezővel.
3. A 04 értékű típus a kibővített lineáris cím, mely az adatmezőjében lévő két bájttal meghatározza az utána következő adatrekordok címének felső két bájtját, kibővítve ezzel a címezhető tartományt 64 kB-ról 4 GB-ra.

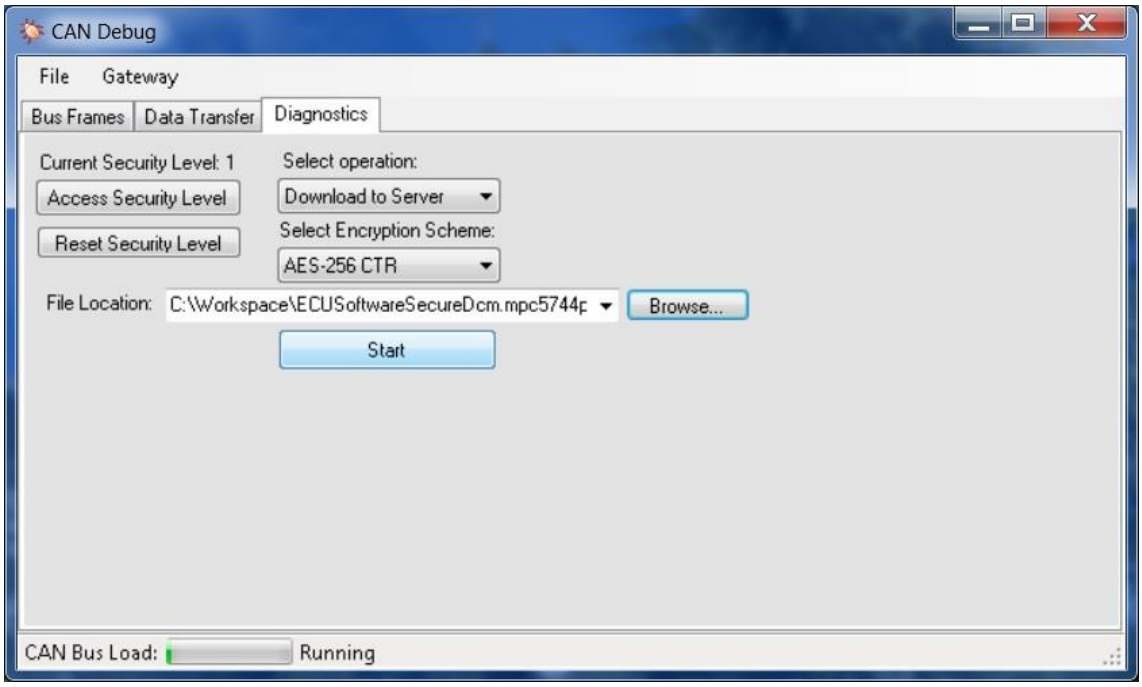
A kliens programban létrehoztam egy `IntelHEX` osztályt, mely egy bemenő fájlnévvel jön létre. Ha a fájl nem hex típusú, akkor előbb meghívja az objcopy-t, majd annak kimenetén dolgozik, egyébként az eredeti fájllal. A konstruktor a `StreamReader` beépített osztály segítségével olvassa be a fájl rekordjait, és dolgozza fel őket a típusuknak megfelelően. Ha a fájl megfelelő formátumú volt (fájlvége sor jelen volt, ellenőrzőösszegek helyesek voltak), létrejön az osztály, mely két tagváltozóval rendelkezik:

```
protected List<byte[]> memoryData = new List<byte[]>();
protected List<uint> memoryStartAddresses = new List<uint>();
```

Az előbbi tartalmazza az összefüggő adatbájtok tömbjeit egy listában, az utóbbi pedig meghatározza az egyes adattömbök kezdőcímét a memóriában. Mindkettőhöz tartozik egy olvasást lehetővé tevő publikus property.

8.6.2 Programmemória feltöltése

Az alábbi ábrán látható a diagnosztikai interfészen keresztüli programmemória frissítést lehetővé tevő minimalista grafikus felület, mely gyakorlatilag önmagáért beszél:



8.4. ábra: Diagnosztikai kommunikáció GUI-ja

Az `IntelHEX` osztály létrejötte után az ECU-n át kell váltani a bootloader programra, melyet egy reseteléssel lehet könnyen elérni. Erre a célra is rendelkezésre áll egy UDS szolgáltatás, az *ECUReset,* melyet *hardReset* alfunkcióval meghívva az ECU reseteli önmagát, és reset után a bootloader indul el automatikusan. A reset után *TesterPresent* kérésekkel tesztelem, hogy megtörtént-e az inicializáció, és képes-e válaszolni az UDS kérésekre. Ezt követően el kell végezni a biztonsági hozzáférést, majd az egyes összefüggő memóriablokkokra külön-külön egyesével le kell futtatni egy 8.3-as szekvenciát az összetartozó memóriablokk-kezdőcím párok felhasználásával, és le is zajlott a biztonságos firmware frissítés.

8.7 Programhitelesítő bootloader

Az ECU programmemória frissítéséhez szorosan kapcsolódik a bootloader működése, hiszen a bootloader végzi a főprogram betöltését reset után, valamint az menedzseli a főprogram frissítését is. A biztonságos szoftver platform megteremtésének egy következő állomása a főprogram hitelesítését elvégző bootloader megvalósítása. A program hitelességét az garantálja, ha a programmemória tartalma megegyezik a hitelesített kliens által feltöltött tartalommal. Amennyiben tehát a kliens aláírja a memória tartalmát a privát kulcsával, akkor azt a bootloader hitelesíteni tudja a publikus kulcs segítségével, és az aláírás hamisításához szükség lenne a privát kulcshoz, amit az ECU nem tartalmaz. Közvetlenül természetesen nem lehetséges, csak az ECC

kulcsméretének megfelelő adat aláírása, de hash függvénnyel (secp256r1 görbe esetén célszerűen SHA-256-tal) lenyomatot számolva, majd a lenyomatot aláírva már kész is az eljárás.

A bootloadernek ehhez az eljáráshoz a következőket kell előre tudnia:

- Mely kezdőcímektől indul, és milyen hosszúak a főprogram memóriaterületei, valamint ezekből az adatpárokból mennyi van.
- A programmemória lenyomatának aláírása.

Ezeket az információkat a firmware frissítés utolsó lépéseként kellene a diagnosztikai kliensnek feltöltenie egy előre meghatározott flash memóriaterületre.

Persze itt jogosan merülhet fel a kérdés, hogy szükség van-e egy ilyen biztonsági funkcióra, ha már garantált, hogy biztonsági hozzáférés nélkül nem lehet (távolról) újraprogramozni az ECU-t, és egy ilyen hitelesítés minden reset után fél másodperc nagyságrendű holtidőt okozhat. Továbbá mivel mindez csak szoftveres szintű megoldás, ezért végképp nem tud biztonságot nyújtani, hiszen maga a bootloader program is ugyanúgy átírható egy sikeres támadás során, mint a fő ECU alkalmazás. Így tehát ezt az eljárást nem alkalmaztam a jelen környezetben, viszont HSM-mel rendelkező kontroller esetén mindenképpen szükséges lesz megvalósítani egy olyan funkciót, ami képes a fő memória programjának hitelesítésére. Ilyen esetben már biztonságos kulcstároló memória is rendelkezésre áll majd, ami azt jelenti, hogy a hitelesítés alapja egy titkos kulcsú séma is lehet (MAC).

9 Eredmények

A diplomamunkám során megismerkedtem a kriptográfia tudományával, összeállítottam egy kompakt, beágyazott rendszerben alkalmazható kriptográfiai elemkészletet. Létrehoztam egy CAN-USB átjárót, valamint hozzá egy drivert és egy grafikus felhasználói felületet. Mind az ECU-ra, mind a kliensre illesztettem a meglévő DCM/UDS diagnosztikai stack-et, mely fölé integráltam egy szoftveres védelmi réteget. Ebben a védelmi rétegben alkalmazott megoldások garantálják a szoftver megvalósítás mellett az elérhető legnagyobb biztonságot a diagnosztikai kommunikáció felett.

Végezetül azonban szeretném előrevetíteni, hogy mi lett volna, ha a rendelkezésemre állt volna egy olyan ECU, ami tartalmaz HSM-et. Egy ilyen ECU esetén ki lehetne használni a biztonságos kulcstároló memóriákat, amely megteremtette volna a lehetőséget arra, hogy ne kelljen aszimmetrikus rejtjelezéssel hitelesítenem a diagnosztikai alkalmazást, hanem egy ezerszer gyorsabb MAC sémát használhattam volna.

A másik nagy változtatást a biztonságos bootolás lehetősége jelentette volna, mely szintén MAC művelettel képes lenne ellenőrizni a program eredetiségét, és szükség esetén képes lett volna beavatkozni a szoftver futásába. Továbbá azt se felejtsük el, hogy a hardveres rejtjelezőnek köszönhetően a titkosítás is sokkal gyorsabb művelet lenne. Való igaz az, hogy a diagnosztikai kommunikáció során ennek nincs nagy jelentősége, viszont az ECU-k közötti valósidejű kommunikáció (egy része) is a jövőben majd üzenethitelesítést igényel, és abban az esetben már nem lesz mindegy, hogy a CPU kódol, vagy egy dedikált periféria, sokkal gyorsabban.

Nem mellesleg pedig létrehoztam egy CAN átjárót egy hozzá tartozó klienssel, mely a jövőben tetszőleges célokra továbbfejleszthető. Ezen kívül van egy kriptográfiai könyvtáram, amelyen még bőven lehet mit módosítani illetve lehet még hozzáadni új eljárásokat.

Irodalomjegyzék

[1] ThyssenKrupp Presta Hungary, „ThyssenKrupp Presta honlap,” 2014. [Online]. Available: http://www.thyssenkrupp-presta.hu/hu/ceg. [Hozzáférés dátuma: 18. november 2014.].

[2] International Organization of Motor Vehicle Manufacturers (OICA), „Economic Contributions,” 2014. [Online]. Available: http://www.oica.net/category/economic-contributions/. [Hozzáférés dátuma: 18. november 2014.].

[3] Index.hu, „Az autógyártás adta a GDP tizedét,” 10. február 2014. [Online]. Available: http://index.hu/gazdasag/2014/02/10/az_autogyartas_adta_a_gdp_tizedet/. [Hozzáférés dátuma: 18. november 2014.].

[4] L. Delgrossi, „The Future of the Automobile - Vehicle Safety Communications,” 8. április 2014. [Online]. Available: http://web.stanford.edu/class/me302/PreviousTerms/2014-04-08-VSC02-Data&Applications.pdf. [Hozzáférés dátuma: 18. november 2014.].

[5] Deloitte, „Trends and outlook of the auto electronics industry,” 2013. [Online]. Available: http://www.deloitte.com/assets/dcom-china/local%20assets/documents/industries/manufacturing/cn_mfg_autoelectronicindustry_210314.pdf. [Hozzáférés dátuma: 18. november 2014.].

[6] Wikipedia, „Dedicated Short Range Communications,” 21. augusztus 2014. [Online]. Available: http://en.wikipedia.org/wiki/Dedicated_short-range_communications. [Hozzáférés dátuma: 18. november 2014.].

[7] H. Krishnan, „Vehicle Safety Communications Project,” 2006. [Online]. Available: www.sae.org/events/ads/krishnan.pdf. [Hozzáférés dátuma: 18. november 2014.].

[8] M. Wolf, A. Weimerskirch és C. Paar, „Security in Automotive Bus Systems,” 2004. [Online]. Available: http://www.weika.eu/papers/WolfEtAl_SecureBus.pdf. [Hozzáférés dátuma: 18. november 2014.].

[9] O. Henniger, L. Apvrille, A. Fuchs, Y. Roudier, A. Ruddle és B. Weyl, „Security requirements for automotive on-board networks,” október 2009. [Online]. Available: http://biblio.telecom-paristech.fr/cgi-bin/download.cgi?id=10024. [Hozzáférés dátuma: 18. november 2014.].

[10] Wikipedia, „On-board diagnostics,” 10. december 2014. [Online]. Available: http://en.wikipedia.org/wiki/On-board_diagnostics. [Hozzáférés dátuma: 13. december 2014].

[11] International Organization for Standardization, *Road vehicles — Diagnostics on Controller Area Networks (CAN),* Genf: ISO, 2004.

[12] AUTOSAR, „Layered Software Architecture,” 6. október 2011. [Online]. Available: www.autosar.org/fileadmin/files/releases/4-0/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf. [Hozzáférés dátuma: 18. november 2014.].

[13] S. Munk, „Információbiztonság vs. informatikai biztonság,” 27. november 2007. [Online]. Available: http://hadmernok.hu/kulonszamok/robothadviseles7/munk_rw7.html. [Hozzáférés dátuma: 18. november 2014.].

[14] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham és S. Savage, „Experimental Security Analysis of a Modern Automobile,” május 2010. [Online]. Available: http://www.autosec.org/pubs/cars-oakland2010.pdf. [Hozzáférés dátuma: 18. november 2014.].

[15] S. Checkoway, D. McCoy, B. Kantor, K. Koscher, A. Czeskis, F. Roesner és T. Kohno, „Comprehensive Experimental Analyses of Automotive Attack Surfaces,” augusztus 2011. [Online]. Available: www.autosec.org/pubs/cars-usenixsec2011.pdf. [Hozzáférés dátuma: 18. november 2014.].

[16] Wikipedia, „Security through obscurity,” 3. november 2014. [Online]. Available: http://en.wikipedia.org/wiki/Security_through_obscurity. [Hozzáférés dátuma: 18. november 2014.].

[17] M. Jancer és Wired, „Take a Look Inside the First Steer-by-Wire Car,” 13. május 2013. [Online]. Available: http://www.wired.com/2013/05/al_drivebywire/. [Hozzáférés dátuma: 18. november 2014.].

[18] J. Stotz, N. Bißmeyer, F. Kargl, S. Dietzel, P. Papadimitratos és C. Schleiffer, „Security Requirements of Vehicle Security Architecture,” június 2011. [Online]. Available: http://www.preserve-project.eu/sites/preserve-project.eu/files/PRESERVE-D1.1-Security%20Requirements%20of%20Vehicle%20Security%20Architecture.pdf. [Hozzáférés dátuma: 18. november 2014.].

[19] M. Wolf, „Vehicular Security Hardware,” 18. november 2009. [Online]. Available: http://www.evita-project.org/Publications/Wolf08.pdf. [Hozzáférés dátuma: 18. november 2014.].

[20] Wikipédia, „Pszichológiai manipuláció (informatika),” 18. november 2013. [Online]. Available: http://hu.wikipedia.org/wiki/Pszichol%C3%B3giai_manipul%C3%A1ci%C3%B3_%28informatika%29. [Hozzáférés dátuma: 18. november 2014.].

[21] Verizon, „2013 Data Breach Investigations Report,” 2013. [Online]. Available: www.secretservice.gov/Verizon_Data_Breach_2013.pdf. [Hozzáférés dátuma: 18. november 2014.].

[22] L. Buttyán és I. Vajda, Kriptográfia és alkalmazásai, Budapest: Typotex Kiadó, 2004.

[23] Wikipedia, „Advanced Encryption Standard,” 12. december 2014. [Online]. Available: http://en.wikipedia.org/wiki/Advanced_Encryption_Standard. [Hozzáférés dátuma: 13. december 2014.].

[24] BlueKrypt, „Cryptographic Key Length Recommendation,” október 2014. [Online]. Available: http://www.keylength.com/en/compare/. [Hozzáférés

dátuma: 18. november 2014.].

[25] Wikipedia, „Elliptic Curve Digital Signature Algorithm,” 13. december 2014. [Online]. Available: http://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm. [Hozzáférés dátuma: 13. december 2014.].

[26] D. Whiting, R. Housley és N. Ferguson, „Counter with CBC-MAC (CCM),” szeptember 2003. [Online]. Available: http://tools.ietf.org/html/rfc3610. [Hozzáférés dátuma: 18. november 2014.].

[27] F. Pietrosanti, „Not every elliptic curve is the same: trough on ECC security,” 26. szeptember 2010. [Online]. Available: http://infosecurity.ch/20100926/not-every-elliptic-curve-is-the-same-trough-on-ecc-security/. [Hozzáférés dátuma: 7. december 2014.].

[28] National Institute of Standards and Technology, „Digital Signature Standard (DSS),” július 2013. [Online]. Available: nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. [Hozzáférés dátuma: 18. november 2014.].

[29] Fujitsu Semiconductor, „Secure Hardware Extension,” február 2012. [Online]. Available: http://www.escrypt.com/fileadmin/escrypt/pdf/WEB_Secure_Hardware_Extension_Wiewesiek.pdf. [Hozzáférés dátuma: 18. november 2014.].

[30] A. Groll, J. Holle, M. Wolf és T. Wollinger, „Next Generation of Automotive Security: Secure Hardware and Secure Open Platforms,” 2010. [Online]. Available: http://www.oversee-project.com/fileadmin/oversee/scientific_publications/OVERSEE_Paper_ITS_World_2010_Final.pdf. [Hozzáférés dátuma: 18. november 2014.].

[31] C. Jouvray, „Ensuring Secure Communication for V2X - the PRESERVE Solution -,” 24. szeptember 2013. [Online]. Available: https://project.inria.fr/scoref/files/2013/10/Preserve-Score@F-Workshop-2013-part-2.pdf. [Hozzáférés dátuma: 18. november 2014.].

[32] K. MacKay, „micro-ecc,” 2014. [Online]. Available: https://github.com/kmackay/micro-ecc. [Hozzáférés dátuma: 18. november 2014.].

[33] Offspark B.V., „PolarSSL Features,” 2014. [Online]. Available: https://polarssl.org/features. [Hozzáférés dátuma: 7. december 2014.].

[34] V. Rijmen és P. S. L. M. Barreto, „The Wirlpool Hash Function,” 2008. [Online]. Available: http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html. [Hozzáférés dátuma: 7. december 2014.].

[35] The OpenSSL Project, „OpenSSL,” 15. október 2014. [Online]. Available: https://www.openssl.org/. [Hozzáférés dátuma: 7. december 2014.].

[36] J. Song, R. Poovendran, J. Lee és T. Iwata, „The AES-CMAC Algorithm,” június 2006. [Online]. Available: http://tools.ietf.org/html/rfc4493. [Hozzáférés dátuma: 18. november 2014.].

[37] Microsoft Developer Network, „Calling Native Functions from Managed Code,” 2014. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms235282.aspx. [Hozzáférés dátuma: 18. november 2014.].

[38] D. A. McGrew és J. Viega, „The Galois/Counter Mode of Operation (GCM),” 31. május 2005. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf. [Hozzáférés dátuma: 7. december 2014.].

[39] ST Microelectronics, „Discovery kit for STM32F407/417 lines - with STM32F407VG MCU,” [Online]. Available: http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419. [Hozzáférés dátuma: 18. november 2014.].

[40] ST Microelectronics, „STM32F105/7, STM32F2 and STM32F4 USB on-the-go Host and device library (UM1021),” [Online]. Available: http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/PF257882#. [Hozzáférés dátuma: 18. november 2014.].

[41] ST Microelectronics, „Reference Manual, STM32F405xx/07xx,

STM32F415xx/ 17xx, STM32F42xxx and STM32F43xxx advanced ARM-based 32-bit MCUs," [Online]. Available: www.st.com/st-web-ui/static/active/en/resource/technical/document/reference_manual/DM00031020.pdf. [Hozzáférés dátuma: 18. november 2014.].

[42] Vector, „Vector KnowledgeBase," 4. november 2013. [Online]. Available: http://vector.com/kb/index.php?q=47. [Hozzáférés dátuma: 18. november 2014.].

[43] K. Hamilton, „Top 5 SerialPort Tips," 10. október 2006. [Online]. Available: http://blogs.msdn.com/b/bclteam/archive/2006/10/10/top-5-serialport-tips-_5b00_kim-hamilton_5d00_.aspx. [Hozzáférés dátuma: 18. november 2014.].

[44] International Organization for Standardization, *ISO14229: Road vehicles — Unified diagnostic services (UDS)*, Genf: ISO, 2006.

[45] Freescale Semiconductor, „MPC5744P Reference Manual," április 2014. [Online]. Available: http://cache.freescale.com/files/32bit/doc/ref_manual/MPC5744PRM.pdf?fasp=1&WT_TYPE=Reference%20Manuals&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&fileExt=.pdf. [Hozzáférés dátuma: 7. december 2014.].

[46] Freescale Semiconductor, „MPC574xP: Qorivva 32-bit MCU for Automotive and Industrial Functional Safety Applications," 2014. [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC574xP. [Hozzáférés dátuma: 7. december 2014.].

[47] GNU Project, „objcopy," 2013. [Online]. Available: https://sourceware.org/binutils/docs/binutils/objcopy.html. [Hozzáférés dátuma: 18. november 2014.].

[48] Wikipédia, „Intel HEX," 21. augusztus 2014. [Online]. Available: http://hu.wikipedia.org/wiki/Intel_HEX. [Hozzáférés dátuma: 7. december 2014.].

Rövidítésjegyzék

ADAS	Advanced Driving Assistance Systems – Fejlett vezetést segítő rendszerek
AES	Advanced Encryption Standard – Rijndael blokkrejtjelezőn alapuló titkosítási szabvány
API	Application Programming Interface – Alkalmazásprogramozási interfész
AUTOSAR	AUTomotive Open System ARchitecture – Nyílt autóipari szoftverarchitektúra szabvány
CAN	Controller Area Network – Mikrokontrollerek közötti kommunikációs busz
CAN TP	CAN Transport Protocol – A CAN busz feletti hálózati réteg protokoll
CBC	Cipher Block Chaining – Blokkrejtjelezési mód
CCM	Counter with CBC-MAC – Egyesített hitelesítő és rejtjelező eljárás
CDC	Communications Device Class – Az USB egyik eszközosztálya
CFB	Cipher FeedBack – Blokkrejtjelezési mód
CMAC	Cipher-based MAC – Blokkrejtjelezőre épülő üzenethitelesítő kód
CTR	CounTeR – Blokkrejtjelezési mód
DCM	Diagnostic Communication Manager – UDS alapú AUTOSAR diagnosztikai kommunikációs szoftvermodul
DLL	Dinamically Loaded Library – Dinamikusan betölthető programkönyvtár
DSA	Digital Signature Algorithm – Digitális aláírásséma
DSRC	Dedicated Short Range Communications – Rövid hatótávú kommunikáció az erre a célra elkülönített 5,9 GHz-es frekvencián

DTC	Diagnostic Trouble Code – Diagnosztikai hibajelző kód, egyedi hibaazonosító érték
ECB	Electronic CodeBook – Blokkrejtjelezési mód
ECC	Elliptic Curve Cryptography – Elliptikus görbéken alapuló kriptográfia
ECDH	Elliptic Curve Diffie-Hellman – Elliptikus görbe kriptográfián alapuló Diffie-Hellmann kulcscsere algoritmus
ECU	Electronic Control Unit – Fedélzeti elektronikus vezérlőegység (mikrokontroller)
ELF	Executable and Linkable Format – Rugalmas, architektúrafüggetlen programleíró fájlformátum
EMI	ElectroMagnetic Interference – Elektromágneses zavarjel
EVITA	E-safety Vehicle Intrusion proTected Applications – Autóipari hardver védelmi kutatási projekt
GCC	GNU Complier Collection – GNU fordítógyűjtemény
GCM	Galois/Counter Mode – Hitelesített rejtjelező algoritmus, mely a Galois MAC és CTR sémák kombinációja
GHS	Green Hills Software – Beágyazott szoftver megoldásokat fejlesztő vállalat
GNU	GNU's Not Unix – Az Unix operációs rendszer mintája alapján fejlesztett nyílt forráskódú szoftver
GPL	GNU General Public License – Általános célú nyílt forráskódú licenc
GUI	Graphical User Interface – Grafikus felhasználói felület
HMAC	Hash-based MAC – Hash függvényre épülő üzenethitelesítő kód
HSM	Hardware Security Module – Hardver védelmi modul
ID	Identification (Field) – A keretarbitráció alapjául szolgáló mező a CAN busz esetén

IV	Initialization Variable – Blokkrejtjelezési módok során az átmeneti regiszter kezdeti értéke
JTAG	Joint Test Action Group – Szabványos (többek között) CPU debuggolásra szolgáló interfész
LUT	LookUp Table – Olyan memóriatömb, amely a számításigényes műveleteket egy olyan mátrixszal helyettesíti, melynek indexei a művelet bemenetei, és az indexek által kijelölt cella a végeredmény
MAC	Message Authentication Code – Üzenethitelesítő kód
NIST	National Institute of Standards and Technology – Amerikai szövetségi tudományos kutatóintézet
nonce	Number only sent ONCE – Egyszer használatos szám, melyet jellemzően kriptográfiai sémák során használnak (pl.: IV)
NRC	Negative Response Code – Negatív válasz azonosító, az UDS szabvány által rögzített hiba okok listája értelmezi a bájt értéket
OBD	On-Board Diagnostics – Személygépjármű fedélzeti diagnosztikai kommunikáció
OFB	Output FeedBack – Blokkrejtjelezési mód
PRESERVE	PREparing SEcuRE V2X communication systems – A V2I-hez és V2V-hez szükséges védelmi követelményeket kutató projekt
PRNG	PseudoRandom Number Generator – Kvázivéletlen-szám generátor
RNG	Random Number Generator – Véletlenszám-generátor
RTR	Remote Transmit Request – Távoli adatkérés típusú keret a CAN buszon
SEVECOM	SEcure VEhicle COMmunication – Európai Uniós, biztonságos járműkommunikációs projekt
SHA	Secure Hashing Algorithm – Kriptográfiai hash függvények családja, melyeket a NIST szabványosított
SHE	Secure Hardware Extension – Az első autóipari ihletésű hardver biztonsági modul szabvány

SSL	Secure Socket Layer – Az internetes adatforgalom egyik biztonsági protokollja
TKP	ThyssenKrupp Presta Hungary vállalat
TLS	Transport Layer Security – Az SSL utódja
TRNG	True Random Number Generator – Valódi (egyenletes eloszlású fizikai mennyiségre építő) véletlenszám generátor
UDS	Unified Diagnostic Services – ISO14229 autóipari diagnosztikai szabvány
V2I	Vehicle to Infrastructure communication – Jármű és közúti infrastruktúra közötti kommunikáció
V2V	Vehicle to Vehicle communication – Járművek közötti kommunikáció
V2X	Vehicle eXternal communications – V2I és V2V összevonva
VSS	V2X Security Subsystem – Jármű környezeti kommunikációjához a PRESERVE projekt által tervezett biztonsági alrendszer

Ábrajegyzék