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

Saska Szilvia

ADAPTÍV FUZZING MÓDSZERTANA AUTÓIPARI KÖRNYEZETBEN

KONZULENS
Schulcz Róbert

KÜLSŐ KONZULENS
Babud Imre

BUDAPEST, 2023

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2023. 06. 10.

...…….
Saska Szilvia

Összefoglaló

Napjainkban a modern járművek egyre összetettebb egységekből épülnek fel. Ezzel a komplexitás növekedéssel azonos mértékben a támadási- és sérülékenységi lehetőségek száma is megnövekedett. Az új technológiák, például az önvezető autók és a kommunikációs hálózatok fejlődése új kihívásokat rejtenek magukban biztonsági szempontból is. Ennek megfelelően az autóipari kiberbiztonság területén is olyan változásokra van szükség, melyek új módszerek segítségével képesek lépést tartani ezzel a nagy léptékű kihívással, így hatékonyabb eszközök kifejlesztésével és felhasználásával sokkal biztonságosabb rendszerek születhetnek.

Az autóipari szegmens szereplőinek is alkalmazkodniuk kell a megnövekedett igényekhez, így a fejlesztési metodológiájukban új tesztelési eljárások számtalan variációja jelenik meg és épül be a termékfejlesztési folyamataikba.

Az egyik ilyen új módszer az információs technológiák világából már ismert fuzzing, ami az autóiparban a fenti jelenségek hatására kezd elterjedni. A fuzzing egy olyan technika, amely során a szoftver vagy rendszer bemeneteinek véletlenszerűen módosított változatait tesztelik, hogy felfedezzék a hibákat és sérülékenységeket.

Jelen szakdolgozatban az okos fuzzing módszertanát mutatom be, egy elméleti teszteset kidolgozásával, ami az autóiparban használt Unified Diagnostic Services és CAN protokollokra épül.

Abstract

In today's world, modern vehicles are being built from increasingly complex units. With this increase in complexity, the number of attack vectors and vulnerabilities has also grown. The development of new technologies, such as autonomous cars and communication networks, presents new challenges in terms of security as well. Accordingly, the field of automotive cybersecurity requires changes that can keep up with this large-scale challenge, leading to the development and utilization of more effective tools, thus enabling the creation of much safer systems.

The participants in the automotive industry segment also need to adapt to the increased demands, resulting in numerous variations of new testing procedures being introduced and incorporated into their product development processes. One such new method is fuzzing, which is already known in the world of information technology and is starting to spread in the automotive industry due to the aforementioned phenomena. Fuzzing is a technique in which randomly modified versions of software or system inputs are tested to discover errors and vulnerabilities.

In this thesis, I present the methodology of smart fuzzing, with the development of a theoretical test case based on the Unified Diagnostic Services and CAN protocols used in the automotive industry.

.

1. Bevezetés

Az autóipari termékek az elmúlt években jelentős fejlődésen mentek keresztül, melynek következtében az esetleges támadási felületek és sérülékenységek száma exponenciálisan megnőtt. A járművekben használt megoldások többségét biztonságkritikus rendszernek tekintjük, ugyanis nem megfelelő működés esetén (rosszindulatú támadás) kárt tehet a felhasználóban és a környezetben is. Emiatt kiemelten fontos, hogy a fenyegetések azonosításához kiberbiztonsági teszteléseket hajtsanak végre. A tesztelések fontosságát nemcsak az egyes autóipari szereplők egyéni belátása szerint kell elvégezni, hiszen több szabvány is megjelent, melyek előírják az átfogó tesztelés szükségességét, így például az ISO/SAE 21434. Ez az iparágban újdonságnak számít, emellett számos kritériumoknak meg kell felelni, melyeket gyártói követelmények és szabványok is magukba foglalnak, mint például az ISO/SAE 21434. A tesztelési folyamat elsődleges célja a sérülékenységek felfedezése, majd azok javítása. Kijelenthető, hogy a biztonsági tesztelések végrehajtása ma már elkerülhetetlen az autóipari termékek fejlesztési életciklusában. [1]

Napjainkban számos tesztelési eljárás létezik az esetleges hibák megtalálására, melyek közül az egyik a fuzzing módszer. Ez az autóipari környezetben még nem annyira elterjedt tesztelési eljárás, viszont szükségessége megkérdőjelezhetetlen az átfogó módszertana miatt, mellyel széles spektrumban azonosíthatók támadási felületek.

E dolgozat célja egy autóipari környezetben felhasználható okos fuzzer létrehozása, nyílt forráskódú keretrendszer segítségével a későbbiekben.

A bevezetés további részeiben ismertetem a kutatási céljaimat, illetve módszereimet, az olvasó megismerkedhet a kiber- és információbiztonság alapjaival (AAA vagy CIA triád), a kiberbiztonság és az autóipar kapcsolódási pontjaival és az autóiparban használt kommunikációs protokollokkal.

1.1. Kutatási célok

Szakdolgozatom elkészítésével az elsődleges célom az volt, hogy különböző tesztelési eljárásokat, módszertanokat ismertessek, különös tekintettel az okos fuzzerekre. Ezenfelül általam kidolgozott teszteset elméleti bemutatását is végre kívánom hajtani, mely a későbbiekben alkalmazható lesz autóipari, kiberbiztonsági tesztelésekhez . Céljaim megvalósításhoz több mérföldkövet határoztam meg magamnak kutatásom nyomon követhetőségének érdekében:

1. A kiber- és információbiztonság alapjainak ismertetése, megjelenésük a mindennapokban, hatásuk azautóiparra.
2. Tesztelési eljárások és módszertanok bemutatása.
3. Az adaptív fuzzing technika bemutatása és ismertetése.
4. Az okos fuzzing elméleti felépítése, példák az UDS szolgáltatás tesztelésére.
5. Korlátok, problémák, illetve előnyök és hátrányok.
6. Tapasztalataim összegzése, a kutatásom továbbfejlesztési ötletei.

1.2. Kutatási módszerek

Az előbb említett alfejezetben megemlített kutatási célok és mérföldkövek teljesítéséhez széles spektrumban próbáltam kutatási módszertanokat felhasználni, növelve ezzel a tudományosság alapját. A kooperatív képzésem keretében tanult autóipari, illetve kiberbiztonsági módszereket alkalmaztam, melyek kapcsolódtak a szakdolgozatom témaköréhez. Széleskörű hazai és nemzetközi irodalomkutatást, forráselemzést hajtottam végre, kiemelt figyelemmel a fuzzingra, a tesztelésekre és az autóipari kiberbiztonság problémáira. SWOT-analízist alkalmaztam az okos fuzzing tesztelési módszertanon annak érdekében, hogy átfogó képet kapjak annak előnyeiről és hátrányairól rendszerezett formában.

1.3. Kiberbiztonság és információbiztonság

A kiberbiztonság magában foglalja a számítógépes rendszerek, hálózatok, a bennük tárolt és a rajtuk keresztülfutó adatok és információk védelmét a különböző kibertámadások és vírusok ellen. Fő feladata a potenciális sérülékenységek és kockázatok

mielőbbi felfedezése és azok javítása. Ahogy életünk szinte minden szegmensét egyre jobban átszövik a számítógépek és a különböző hálózatok, úgy azzal együtt rohamosan növekszik a felhasználók száma, az eltárolandó adatmennyiség is, mely támadási felületként is szolgálhat a hackerek számára. A digitális biztonság szemléletmód kialakítására több megközelítést dolgoztak ki, melyek közül az AAA és CIA modelleket kívánom kifejteni. Előbbi, az AAA modell tagjai a következők [2]:

- Hitelesítés (Authentication),
- Engedélyezés (Authorization),
- Ellenőrizhetőség (Accounting).

A hitelesítés az a folyamat, mely során megbizonyosodhatunk a hitelesítendő személy vagy program kilétéről. A hitelesítésnek három megközelítése ismert [2]:

- valami, ami tudás (pl. jelszó, hiszen az ember a jelszavát jobb esetben, megjegyzi);
- valami, ami rendelkezésre áll (pl. belépőkártya, hiszen az ember birtokolja azt);
- valami, ami vagy (pl. ujjlenyomat, hiszen ez adott egyénen teljes mértékben egyedi).

A hitelesítésről általánosságban elmondható, hogy amíg a személyazonosságok ellenőrzésére vonatkozik – vagy másik szoftver esetén annak azonosítására -, az utána következő folyamat, vagyis az engedélyezés, az adott felhasználó jogosultságait határozza meg.

A NIAG- vagyis NATO Industrial Advisory Group definíciója alapján az engedélyezés, hozzáférési jogosultságok, amelyeket megadott felhasználó, program vagy folyamat alkalmazhat. [2]

Az ellenőrizhetőség lényege az, hogy a tevékenység, ami történt, visszavezethető legyen hitelesített vagy nem hitelesített személyekhez is. [2]

A CIA modell tagjai pedig a következők:

- Bizalmasság (Confidentality),
- Sértetlenség (Integrity),

- Rendelkezésre állás (Availability).

A bizalmasság lényege, hogy a kritikus adatokhoz és információkhoz kizárólag az a személy vagy entitás férhessen hozzá, aki rendelkezik az ahhoz szükséges jogosultságokkal. A sértetlenség azt jelenti, hogy az adatok és információk hiánytalan állapotban vannak, azok módosítására vagy manipulálására észrevétlenül nincsen lehetőség. A rendelkezésre állás az adatok és információk megfelelő helyen és megfelelő időben történő hozzáférést foglalja magában.

1.4. Kiberbiztonság az autóiparban

A járművek egyre fokozottabb kapcsolódása a digitális világhoz komoly veszélyeket is hordozhat a felhasználók számára. Önvezető járművet fejleszteni több millió kódsor megírását és több tucat elektronikus eszköz beépítését jelenti, mely számos potenciális támadási felületet biztosít a támadók számára [3]. Az autókban alkalmazott vezérlési rendszerek függőségi viszonyban állnak a kapcsolt rendszerekkel, ugyanis azok érzékeny adatokat tárolhatnak, melyek egy kiberbűnöző számára lehetőséget biztosítanak támadási tevékenység végrehajtására. A kinyert adatok olyan kárt tudnak okozni, amelyek nemcsak az járművezető és az utasok sérülését eredményezheti, hanem azon túlmenően a környezetre is káros hatással lehet, amely akár terrortámadásként is elkönyvelhető. Például gondoljunk arra, mi történhet, ha egy üzemanyaggal teli tartályautó felett veszik át az irányítást. Az ipar számára először a 2015-ben történt események hívták fel a figyelmet a digitális biztonság fontosságára, ugyanis ekkor két kiberbiztonsági szakember átvette egy Jeep Cherokee felett az uralmat, és távolról irányították egy laptop segítségével [3]. Másik intő példa lehetett az a 2016-ban történt esemény, amikor egy Tesla Model S-t sikeresen feltörtek, és a sebességbeállítások megváltoztatásával vezérelték a járműrendszert [4].

A technológia fejlődésével egyre több fenyegetettség jelent meg az autóipar és a modern személygépjárművel rendelkező felhasználók számára. Ahogy az autóipar egyre jobban támaszkodik az elektronikára, digitális logikára, bonyolultabb járműfedélzeti kommunikációra, vezetéknélküli kapcsolódási lehetőségekre, úgy növekszik a biztonsági kockázatok száma is. Az önvezetéshez használt technikák megjelenésével – elég csak a fék vezérlésre vagy a sebességtartó elektronikára gondolni – megjelenik a „beavatkozó” elektronika, ami csak növeli a kockázatokat. Az előbb említett példa is felhívja a

figyelmet arra, hogy az okos eszközökkel végrehajtott távoli hozzáférésű támadások (például telefon, számítógép) valós veszélyt jelentenek. [5]

Az elektromos vagy elektromechanikus rendszereknek része lehet a beágyazott rendszer, mely olyan céláramkör, ami egy jól definiált feladatot hajt végre. Az Electrical Control Unit –vagyis az ECU az autóiparban használt beágyazott rendszert jelenti, amely manapság szinte kivétel nélkül minden egyes járműben megtalálható egyre nagyobb számban A modern járművekben az ECU-k számának a növekedésével egyre több szoftver is található, amelyek befolyásolják az autók működését. Az ilyen szoftvereket célzott támadásokkal lehet megbénítani, ami vezetési nehézségekhez és súlyos balesetekhez vezethet.

A fentebb említett példák is hozzájárultak ahhoz, hogy a járművek tesztelésének fontosságára nagyobb hangsúly helyezzen az autóipar, és több tesztelési módszert és technikát is alkalmaznak a fejlesztés során.

A járműfedélzeti kommunikációt, a különböző egységek közötti gyors és megbízható adatcserét, többféle protokollal is meg lehet valósítani. Az iparágban a legjellemzőbbek a FlexRay, a Control Area Network (CAN), a Local Interconnect Network (LIN) és a Media Oriented Systems Transport (MOST) protokollok. Ezek más-más tulajdonságokkal rendelkeznek, hiszen már az alrendszerek között is különbség van, például néhány közülük nem igényel nagy sávszélességet. A Society for Automotive Engineers a különböző protokollok csoportosítására 4 osztályt határozott meg [6]:

- Az A osztályba olyan protokollok tartoznak, melyek 10 Kb/s-nál alacsonyabb adatátviteli sebességgel rendelkeznek, mint például a LIN.
- A B osztály képviselője a CAN, 10 és 125 Kb/s-os adatátviteli sebességre képes, tehát ez is alacsonynak számít.
- A C osztályban a 125 Kb/s-nél nagyobb, viszont 1 Mb/s-os sebességre is képes, itt a nagy sebességű CAN hálózatot tudjuk megemlíteni.
- A D osztályban a FlexRay és MOST található, ez a két protokoll képes az 1 Mb/s-os adatcserét is biztosítani.

A járművek digitális biztonsági helyzetét tovább színesíti az, hogy az autókban használt szoftver- és hardverkomponensek kibővültek az Internet of Things (IoT) világával, ami magasabb szintű hálózat-biztonsági kockázatot jelent. [6]

Az iparban megjelenő egyes nemzetközi szabványok irányt adnak már kiberbiztonsági területen is a járművek megfelelő szoftveres adaptálására. A kiberbiztonság fontosságáról és az elektromos, elektronikus rendszerek megtervezésével, jóváhagyási folyamataival kapcsolatban bővebben az ISO/SAE 21434, Jaspar, UNECE R155 és R156-os szabványokból tájékozódhatnak a szakemberek, de ezeknek az eljárási folyamatoknak egy része publikusan nem hozzáférhető, ezáltal bővebb bemutatásukra ebben a dolgozatban nincs lehetőség.

1.5. Jelentősebb járműipari protokollok

A mai modern járművekben található ECU-k többsége -mint például a kormány, fék - biztonságkritikus rendszereknek minősülnek, mivel egy esetleges meghibásodás kárt tehet a felhasználóban [6]. A járművön belül található egységek különböző protokollok segítségével kommunikálnak egymással, mint például CAN protokollal. Ezen felül különböző diagnosztikai protokollok is felhasználásra kerülnek, mint például Unified Diagnostic Services (UDS).

1.5.1. Control Area Network

Ez egy Bosch által kifejlesztett soros, aszinkron, multicast, fél-duplex, CSMA/CA alapú kommunikációs protokoll , amelyet kifejezetten autóipari felhasználásra terveztek. A protokollt később szabványosították is, napjainkban a bővebb specifikáció az ISO 11898-as számú szabványban olvasható. A CAN az Open System Interconnection modell 7 szintjéből az adatkapcsolati és fizikai réteget valósítja meg, melyek a következőkért lehetnek felelősek [8]:

Adatkapcsolati réteg	Fizikai réteg
üzenetek filterezése üzenetek állapotkezelése hiba felismerés arbitráció üzenet keretezése szállítási időkeret beállítása üzenet validáció	bit reprezentáció és jelszint továbbítás

1.1 táblázat - Az OSI adatkapcsolati rétegének funkciója

A fenti táblázatban látható üzenetek a buszon áthaladó adathalmazok, melyek fix formátumú, de különböző hosszúságúak lehetnek egészen 64 bájtig. A CAN protokollon belül kettő formátumot különböztethetünk meg. Az első az egyszerű CAN, melyben a

keret adattartalma maximum 8 bájt, ellenben a másik típus, a CAN Flexible Data-rate – röviden CAN-FD – esetében ez az érték már maximum 64 bájt is lehet. Az utóbbi protokoll típus nagyobb sávszélességet és alacsonyabb idejű késleltetést is lehetővé tesz. [8]

Az arbitrációs mechanizmus biztosítja, hogy idő- és adatveszteség nélkül lehessen üzenetet küldeni. A CAN-buszon tetszőleges számú ECU lehet, ezért a buszon az üzenetek könnyen konfliktusba keveredhetnek (összeütközhetnek), ha egyszerre küldik őket. A bitenkénti arbitráció úgy nevezett azonosítója (identifier) segít feloldani az előbb említett problémát, mivel priorizálja az üzeneteket. Minél alacsonyabb az arbitrációs értéke az üzenetnek, az annál fontosabb. [8]

A CAN protokollnak jelentős szerepe van a járműipari kommunikáció létrehozásában, mivel olcsó és robosztus megoldást kínál. A legfőbb előnye az egyszerűségében rejlik, mivel mindamellett, hogy alacsony hálózati komplexitással rendelkezik, a vezetékes összeköttetések költséghatékony megvalósítást tesznek lehetővé, amik ráadásul nem hozzáférhetőek kívülről. [6]

A protokoll nem rendelkezik beépített információ biztonságot támogató lehetőségekkel, illetve a mai járművek nagy sávszélességet is igényelnek. Ez az 1980-as években még nem volt szempont. Protokollszinten nem rendelkezik autentikációs mechanizmussal és titkosítással sem, ezért könnyen válhat egy esetleges támadó célpontjává, bár a vezetékezés miatt fizikai hozzáférést igényel. A protokoll sebezhető számos kiberbiztonsági támadással szemben, így például a visszajátszásos támadással (replay attack), a túlterheléses támadással (Denial of Service, DoS), a lehallgatással (eavesdropping) vagy álarcos támadással (masquerading attack) szemben. Az elsőként említett támadás típusnál a támadó megfelelő CAN-üzeneteket küld újra és újra valós időben. A kiberbűnöző könnyedén tud DoS támadást magas prioritású üzenetek küldésével kivitelezni, hiszen ezáltal az alacsonyabb prioritással rendelkező üzenetek nem tudnak célba érni. [6] A lehallgatás során a titkosítás nélküli CAN-hálózati adatforgalmat rögzítik, és kiértékelik annak érdekében, hogy bizalmas információkat szerezzenek, vagy feltérképezzék az egyes entitások működését. Az álarcos támadás esetében a hackerek hitelesítési információkat lopnak el, majd ezeket használják fel a támadás során a rendszerbe történő bejelentkezésre. A támadók tehát álcázni próbálják magukat az áldozat számítógépén vagy hálózatán, többségében információlopás céljából.

1.5.2. Unified Diagnostic Services

Az Unified Diagnostic Services (UDS) egy kommunikációs protokoll, amelyet az autóiparban használnak a különböző vezérlőegységek diagnosztikai adatainak elérésére és állapotuk ellenőrzésére. Az OSI modellben az alkalmazási réteget valósítja meg.

Az UDS egy olyan szabványosított protokoll, amely a kommunikáció szintjén az adott járműkommunikációs hálózat protokollján alapszik, így akár a már korábban kifejtett CAN-re is épülhet. Lehetővé teszi az adatátvitelt a diagnosztikai egység (diagnosztikai szoftver vagy hardver) és az ECU-k között. A diagnosztikai protokoll különböző szolgáltatásokat definiál, amelyek lehetővé teszik az adatok lekérdezését, a paraméterek beállítását és az ECU-k által kibocsátott hibakódok olvasását és törlését. Az ISO 14229 az automatizált vezérlési rendszerek diagnosztikai adatainak elérését és kezelését szabályozza a járműiparban, így az UDS-re is kiterjed. A szabvány része az ISO 15031-nek, amely a járműipari diagnosztikai rendszerekre vonatkozik, és meghatározza a diagnosztikai adatok hozzáférhetőségét és a jelentések formátumát. [9]

Az UDS állapotalapú protokoll, amely meghatároz bizonyos munkameneteket és szolgáltatásokat is. A *session*, vagyis *munkamenet* egy konkrét kommunikációs folyamatot jelöl, amely több *szolgáltatást* is tartalmazhat. A *service*, vagyis a *szolgáltatás* egy adott diagnosztikai funkció. Az UDS a következő munkameneteket támogatja [9]:

- Session 0 (Default Session): Az alapértelmezett session, amely lehetővé teszi a diagnosztikai eszköz számára a rendszer alapvető információinak lekérdezését.
- Session 1 (Programming Session): Lehetővé teszi a diagnosztikai eszköz számára a rendszer programozását, beállításait és adatainak frissítését.
- Session 2 (Extended Diagnostic Session): Bővített diagnosztikai funkciókat biztosít, beleértve a rendszer részletes diagnosztikáját és hibaelhárítását.

Az UDS-ben mind a vevő, mind a beszállító definiálhat egyedi munkameneteket, amelyek a termék különböző életciklusai (például fejlesztés, gyártás) során szükséges speciális funkciók elérhetőségét biztosítják. A munkamenet tehát egy konkrét kommunikációs folyamatot jelöl, amely több service-t is tartalmazhat, míg maga a service egy adott diagnosztikai funkció. [9]

Néhány főbb szolgáltatás és azonosítója hexadecimális formában:

Service	Service ID
Diagnostic Session Control	0x10
ECU Reset	0x11
Security Access	0x27
Routine Control	0x31

1.2 táblázat - Néhány UDS szolgáltatás

Az NRC (Negative Response Code) az UDS protokoll része, amely a vezérlőegységek válaszait jelöli a diagnosztikai parancsokra. Az NRC egy negatív válaszkód, amely jelezheti, hogy a vezérlőegység nem tudta végrehajtani a diagnosztikai parancsot, vagy hogy a parancs nem volt érvényes. Az NRC segítségével a diagnosztikai eszközök és a vezérlőegységek kommunikációjának hibáit lehet kiszűrni, és megállapítani azt, hogy a vezérlőegységnek sikerült-e végrehajtani a diagnosztikai parancsot. Ha a vezérlőegység negatív választ küld, az NRC segítségével a diagnosztikai eszköz képes lesz megérteni, hogy mi okozta a hibát, és hogyan lehet javítani azt. Az NRC-k különböző számokat jelölnek, és a számok jelentése szintén az előbb említett szabványban vannak meghatározva. A szabvány szerint a különböző NRC-k jelentése a parancs érvénytelenségét, a vezérlőegység túlterheltségét, a hozzáférési jogok hiányát vagy más hibákat jelölhetnek. [9]

Kijelenthető, hogy az UDS javítja az autók diagnosztikájának hatékonyságát és időráfordítását, ami jelentős mértékben csökkenti a javítás költségeit és időigényét.

1.5.3. Következtetések

Az UDS (Unified Diagnostic Services) és a CAN (Control Area Network) két különböző protokoll, amelyeket autóipari alkalmazásokban használnak.

Az UDS egy diagnosztikai protokoll, amely lehetővé teszi a szakemberek számára, hogy ellenőrizzék és diagnosztizálják az autó vezérlőegységeit. Az UDS protokoll egy olyan általános szabvány, amelyet széles körben használnak az autógyártók és a beszállítók szakemberei egyaránt.

A CAN (Control Area Network) ezzel szemben egy kommunikációs protokoll, amelyet a járművek belső vezetékes hálózatain belül használnak. A CAN segíti az autó vezérlőegységeit abban, hogy kommunikáljanak egymással az adatok továbbításában, beleértve a jármű állapotának és működésének ellenőrzését.

2. A tesztelés terminológiája

E fejezet célja, hogy felhívjam a figyelmet a tesztelés fontosságára, bemutassam a különböző tesztelési módszertanokat és eljárásokat, különös tekintettel a fuzzing-ra, melynek alapjait és fajtáit részletekbe menően kívánom ismertetni.

2.1. Tesztelési módszertanok

A tesztelés nélkülözhetetlen eszköze a verifikációs és validációs technikák világának. Az előbbi kifejezés (verifikáció) a követelmények megfelelőségét ellenőrzi, utóbbi (validáció) pedig egy olyan strukturáltan felépített folyamatot foglal magában, melyben egy visszaigazolást kapunk a megfelelő szoftvertermék működésével kapcsolatban. Ezen tevékenységek összességét az iparban V&V tevékenységeként szoktuk rövidíteni, melyben többek között a különböző tesztelési módszertanok is szerepet kapnak, így a fuzzing tesztelés is.

A változatos tesztelési módszertanok átfogó ismerete és szakszerű alkalmazása kulcsfontosságú annak érdekében, hogy hatékonyan lehessen kivitelezni a termékben visszamaradt hibák és kockázatok felfedezését. A tesztelés minőségi mutatóként szolgál, hiszen, ha az adott program nincsen tesztek által felülvizsgálva, nem győződhetünk meg arról, hogy mennyire hatékonyan működik a lefejlesztett termék. Mivel tökéletesen lefejlesztett, hibamentes szoftver nem létezik, így a tesztelések végrehajtása is elengedhetetlen amennyiben biztonságos terméket kívánunk fejleszteni. A biztonság megteremtésének céljából számos esetben a fejlesztő cégek más vállalatok által elvégzett teszteket is igénybe vesznek, hogy nagyobb lefedettséget biztosítsanak, és az apróbb hibák is felszínre kerüljenek. Egy harmadik fél által végrehajtott, független validáció tehát tovább növeli az elérni kívánt biztonsági szintet. A szoftvertesztelésnek számos módszertana létezik, melyek különböző aspektusból közelítik meg a sérülékenységek megtalálását. A teljesség igénye nélkül a három legelterjedtebbet szeretném bemutatni a dolgozatomban, melyek a következők: fehér doboz, fekete doboz, illetve szürke doboz módszertan. [10] A szoftvertermékre tekinthetünk úgy, mint egy dobozra, a találó név innen eredeztethető. A fejezet további részében bemutatom az előbb említett három módszertant, kiemelve az adott stratégiának a gyengeségeit és erősségeit is.

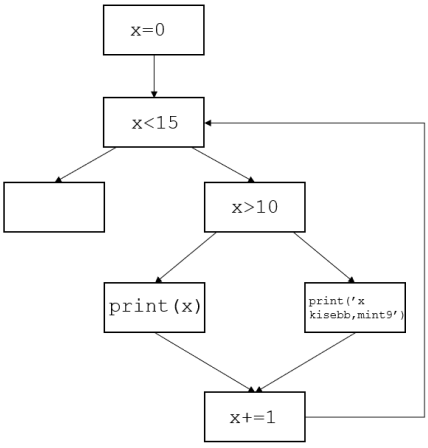
2.1.1. White-box tesztelés

A fehér doboz tesztelés eljárás során a tesztelőnek ismernie kell a forráskódot, mivel a módszertan a kód belső szerkezetének részletes vizsgálatára fekteti a hangsúlyt. Autóipari vonatkoztatásban a tesztelő tehát rálátással bír az ECU-ra és a rajta futó kódra egyaránt. A forráskód analizálása során a végrehajtandó művelet folyamán fel kell fedeznie a kódegységek között a hibára hajlamos részegységeket. [11]

Ez a tesztelési módszertan a statikus módszereket részesíti előnyben, tehát ide sorolhatjuk a manuális teszteket és a statikus kódelemzést is. Az IEEE-hez hasonló szervezetek számos iparági szabványt határoztak meg az ellenőrzések mikéntjéről, például ISO/SAE. A tesztelést nem automatizált módon hajtják végre, tesztelők és fejlesztők vesznek részt a folyamatban. Az alfejezetben megjelenő tesztelési módszertan több tesztelési eljárást is magában foglal, melyek a következők is lehetnek [10]:

- vezérlési folyamat tesztelés – control flow testing
- döntési tesztelés – branch testing

A *control flow* tesztelési eljárás egy strukturális tesztelési stratégia, mely a programvezérlési folyamatot modellvezérlési folyamatként alkalmazza. A bonyolult útvonalak helyett az egyszerűbben felfedezhető útvonalakra fekteti a hangsúlyt, tehát az utasítások végrehajtási sorrendjét tudjuk hatékonyan ellenőrizni. A technikánál a tesztelő a tesztelendő szoftver részegységeire fókuszál, és az adott részhez alkotja meg a tesztelési útvonalat. Ezt a gyakorlatban egységtesztnek - *unit test* - nevezik. A *control flow* tesztelési technikához javasolt alkalmazni gráfokat is, mely tartalmaz csomópontokat, döntési csomópontokat és éleket. A vezérlésfolyam gráfban alapvető blokkokat találhatunk, mely a program működését és logikai alapját reprezentálja [11]. Például egy egyszerű Python kódnak a vezérlésfolyam gráfja a következőképpen néz ki [12]:




```
x = 0
while x < 15:
    if x > 10:
        print(x)
    else:
        print('x kisebb,mint 9')
    x += 1
```

2.1 ábra - Egy program példa [12]

A *branch* tesztelési eljárás szorosan összefonódik az előbb említett *control flow* stratégiával, hiszen a célja az, hogy minden egyes opciót, amit a *control flow* grafikonján megtalálhatunk, tesztelni szükséges. A különböző részegységek tesztelése lehetővé teszi a hatékonyabb hibajavítást, mivel ezáltal nem az egész programot kell javítani, hanem csak magát a részkomponenst. [13]

A *white-box* eljárásnak az előnyei között megemlíthető, hogy a feleslegesnek vélt kódsorokat könnyedén eltávolíthatjuk, ezáltal a teljesítménye a programnak javulhat. A tesztelés során maximális lefedettség érhető el. Mivel tökéletesen működő biztonsági rendszer nem létezik, ezáltal teljes mértékben megfelelő tesztelés sem. Következtetésképpen a fehér dobozos eljárás is rendelkezik gyenge pontokkal. A módszertan megvalósításához tapasztalt tesztelőkre van szükség, emellett időigényes, és a megvalósítás magas költségeket eredményez. A tesztelés alatt minimális az automatizált folyamatok száma, ezért a teszt sikeressége az emberi tényezőn múlik. [10]

2.1.2. Black-box tesztelés

A *fekete dobozos* tesztelés, ellentétben a fehér dobozos teszteléssel, a forráskód ismerete nélkül történik meg, ebből is adódik az elnevezés, mivel magát a szoftvert egy fekete dobozként kezeljük. A tesztelő ebben az esetben egy támadó (hacker) szerepébe képzeli magát, és minden előzetes ismeret nélkül végzi el a teszteléseket, tehát nem lát rá sem a forráskódra, sem a dokumentációkra. A tesztelő vegyíti a manuális és automatizált tesztelési technikákat a sikeres végrehajtás érdekében. A fekete doboz tesztelési módszertant az iparban funkcionális tesztelési eljárásnak is nevezik, de ez a megnevezés

nem fedi le teljes egészében a valóságot, hiszen a módszertan ennél szélesebb spektrumban fedi le a validációs tevékenységeket. A módszertan több eljárást is tartalmaz, amelyek a teljesség igénye nélkül lehetnek a következők [10]:

- Ekvivalencia particionálás - Equivalence Partitioning
- Fuzzing

Az ekvivalencia particionálás során érvényes és érvénytelen bemeneti adatokat generálunk, melyek olyan partíciókba vannak osztva, amelyektől azonos viselkedés az elvárható. Az ilyen típusú tesztesetek létrehozásánál a tesztelők arra törekednek, hogy minden partíciót le tudjanak fedni a tesztelések során legalább egyszer. A tesztesetek számát ezáltal csökkenteni lehet, ami hatékonyabb és gyorsabb munkavégzést tesz lehetővé. [14]

Mivel a fuzzing tesztelési technika a dolgozatom fő témája, így az későbbi fejezetekben kerül bővebb kifejtésre. Magas-szinten megfogalmazva a fuzzing egy olyan automatizált vagy félautomatizált folyamat, melyben véletlenszerű bemeneteket generálva támadjuk az adott szoftverterméket, és figyeljük a visszajelzéseit, amelyekből következtetni lehet a különböző szoftverentitások funkciójára. [10]

A fekete doboz módszertan előnyei közé sorolható, hogy hatékonyan lehet nagy kód szegmenseket átvizsgálni, a tesztelő legtöbb esetben egy kiberbűnöző szerepét próbálja felvenni, mely kreatív megoldásokat és más tesztelői csoportok által felfedetlen hibákat találhat meg. A programozó és a tesztelő szerepe teljes mértékben különválik, tehát a perspektívák is jobban elkülöníthetők (felhasználó és programozó között). Ezeken felül gyorsabb teszteset fejlesztést tesz lehetővé, mint a fehér dobozos módszertan. Hátrányai között említhető, hogy csak limitált a lefedettsége a tesztelésnek, hiszen a belső forráskód elemzése elengedhetetlen a biztonságos termékfejlesztéshez. [10]

2.1.3. Grey-box tesztelés

A szürke doboz tesztelési módszertannál vegyítik a fekete doboz és fehér doboz struktúráknál jelenlévő eljárásokat. A tesztelőnek a tesztek megvalósításához ismernie kell az adatstruktúrát, illetve az algoritmusokat a tesztek megvalósításához. A Grey-box is rendelkezik több megvalósítási technikával, mint például [10]:

- Regressziós tesztelés - Regression testing
- Minta tesztelés -Pattern testing

A regressziós tesztelés alatt megbizonyosodhatunk arról, hogy az új változtatások, amiket a szoftvertermékben lefejlesztettek, nem okoz hibát. A regressziós tesztek verifikációs módszerként is megtalálhatóak, hiszen a teszteseteket gyakran kell ellenőrizni és akár manuálisan is újra futtatni, ezért időigényes, és a tesztelők számára fárasztó eljárás lehet. [15]

A minta tesztelésnél a fő cél a megfelelő architektúra és tervezésnek az igazolása [10]. Különböző architektúrájú és tervezésű programok léteznek, melyek értelmezésének segítségére lehet az úgy nevezett Unified Modeling Language (UML), amely szemléletes ábrákon mutatja be egyes forráskódok implementációját és felépítését.

2.1.4. Következtetések

Tökéletes, sérülékenységmentes szoftver nem létezik, viszont a vállalatok és fejlesztők számára a tesztelések segítséget nyújtanak a támadási felületek feltérképezésén keresztül a sérülékenységek és a hibák kijavításában. A fejezetben taglalt módszertanok –fehér doboz, fekete doboz, szürke doboz - megfelelő alkalmazása a hatékony verifikációs és validációs eljárásokhoz elengedhetetlenek. Mindegyik rendelkezik előnyökkel és hátrányokkal, ahogy az alábbi táblázatban látható:

Fehérdoboz	Szürkedoboz	Feketedoboz
teljes hozzáférés a forráskódhoz és dokumentációhoz	részleges hozzáférés a forráskódhoz és dokumentációkhoz	nincs hozzáférés a forráskódhoz
legidőigényesebb	közepes időigény	legkevésbé időigényes
leginkább manuális, de előfordul automatizáltan is	manuális és automatizált	leginkább automatizált, de előfordul manuálisan is
kisebb kódszegmensekre hatékonyabb	kisebb és nagyobb kódszegmensek vizsgálatára is hatékony	kisebb nagyobb kódszegmensekre is hatékony
szinte teljes lefedettség	limitált lefedettség	limitált lefedettség

2.1 táblázat - A tesztelési módszerek összehasonlítása

Az összes módszertan együttes használatával sem lehet garantálni a tökéletes szoftvertermék létrehozását, azonban egy kiberbűnöző számára megnehezíthetjük a kibertámadások végrehajtását. Habár az autóiparban törekednek az automatizált tesztelési

eljárások használatára, de ettől függetlenül szükséges a manuális, ember elemzés által hozzáadott érték egy-egy szoftvertermék teszteléséhez.

A fejezet további részében néhány konkrét black-box tesztelési technikát kívánok bemutatni.

2.2. Tesztelési technikák

Az ECU-k szoftveres működésük miatt könnyen kibertámadás célpontjaivá válhatnak. Ennek megakadályozására az iparban az ECU-k esetleges támadási felületeit több lépésben tesztelik a szakemberek. A folyamatok és munkák összehangolására gyakran alkalmazzák az úgy nevezett agilis módszertant vagy például *Scrum*-ot.

A szakembereknek mindig az aktuális kiberbiztonsági szabványokat és trendeket szükséges alkalmazniuk, mivel az autóipari szoftverek biztonságára vonatkozó követelmények az évek folyamán egyre jobban szigorodnak. A különböző termékfejlesztési életciklusokat, illetve azok résztevékenységeit össze kell hangolni, így például a gyártás és termékfejlesztés, vagy a tervezés, fejlesztés és tesztelés háromszögét. Már a szoftverfejlesztési folyamat során ajánlott elkezdeni a teszteléseket, hiszen a párhuzamos munkával a sérülékenységek hamarabb megtalálhatóak, ezáltal időt lehet spórolni, például nincs szükség nagyobb kód átdolgozásra. A szoftvertermékeknek az összetételét is ajánlott elemezni, hiszen a járműalkatrészek több beszállítótól is származhatnak. Az összetételt tekintve érdemes vizsgálni, hogy megjelennek-e a szoftvertermékben külső szoftverkomponensek (3rd party components), és vannak-e ismert sérülékenységek, amikre a megoldások már léteznek. Az előbb említett problémákat hatékonyan lehet szűrni automatizált és manuális tesztelések segítségével is. [16]

2.2.1. Penetrációs tesztelés

A penetrációs tesztelés az egyik leggyakrabban alkalmazott tesztelési technika, melynek végrehajtásához általában más vállalatot bíznak meg, és ha megadott idő intervallumon belül nem fedeznek fel sérülékenységet, a rendszert biztonságosnak tekintik.

A penetrációs teszt a fejlesztett termék gyártását vagy forgalmazását megelőző sérülékenységfeltáró vizsgálat, melyet tesztelők, etikus hackerek hajtanak végre az eredményeiket dokumentálva. Ez a tesztelési módszer az egyik leghatékonyabb, mivel

hackerek valós eszközeivel, eljárásaival próbálják feltérképezni a rendszert, autóipari környezetben az ECU-t is. Az eljárásnak rengeteg megközelítése létezik, de a National Institute of Standards and Technology, (NIST) által kiadott dokumentáció 4 kulcsfontosságú lépésre bontja a tesztelést [16]:

- Tervezés/Planning
- Felfedézes/Discovery
- Támadás/Attack
- Dokumentáció/Reporting

A felsorolt lépéseken kívül létezik egy kibővített, autóipar-specifikus penetrációs tesztelési lépéssorozat, melyet a Penetration Testing Execution Standard (PTES) dokumentum ír le részletesen. A szabványban megjelenik például a *támadó fa* is, mely egy a fenyegetettségi modellezési megközelítések közül. [16]

2.2.2. Vulnaribility scanning

A sebezhetőségi vizsgálat, vagy másnéven sérülékenységi szkennelés, a szoftvertermékben ellenőrzi azt, hogy az egyes szoftverentitások hogyan reagálnak bizonyos hitelesítési és engedélyezési eljárásokra. [16]

A sérülékenységi tesztek célja, hogy a lefejlesztett termék védett legyen az ismert sérülékenységekkel és fenyegetésekkel szemben. Ennek megfelelően az ilyen tesztek tárgyát képezi többek között a megfelelő biztonsági intézkedések hiányának vagy a hibák azonosításának a folyamata. A tesztelők forráskód, infrastruktúra és adatbázis segítségével térképezik fel a rendszert, különféle technikákkal megvizsgálva az egyes sérülékenységeket. Fontos megjegyezni, hogy a sebezhetőségi vizsgálatok mind szoftveres, mind hardveres komponensekre kiterjednek, amelyek különböző megközelítési eljárásokat és hibakezelési lépéseket igényelnek. Az előbb említett szoftveres és hardveres érintettség miatt kijelenthető, hogy a lefejlesztett termékek átfogó elemzése szükségszerű annak érdekében, hogy ne maradjanak szabad támadási felületek és kezeletlen kockázatok a végleges termékben, amelyeket a támadók ki tudnak használni.

Autóipari szempontból a különböző hálózati protokollok analizálása szolgálhat példának egy ilyen tesztelés végrehajtásához, így például a CAN vagy FlexRay protokollok. A sérülékenységi szkennelés lehet automatizált és manuális is.[16]

2.2.3. Fuzzing

A fuzzing egy olyan automatizált teszteljárás, amely során a vizsgálandó rendszert nagy számú véletlenszerűen generált bemenettel támadják meg, és figyelik az egyes bemenetekre adott válaszreakciókat. A betáplált értékek származhatnak egy előre elkészített értékekkel rendelkező adatbázisból vagy teljesen véletlenszerű bemeneteket generáló entitásból is. [17]

A fuzzerek eredete egészen 1988-ig nyúl vissza, mikor Barton Miller egyetemi tanár 1200 baud-os telefonvonalon csatlakozott az egyetemi számítógépéhez egy viharos estén. A vihar zajt okozott a vonalon, és a UNIX parancsok mindkét végén érvénytelen vagy rossz bemenetet kaptak, ezáltal a rendszer újra és újra összeomlott. Az előbb említett probléma gyakran megjelent a mindennapokban, ezért Barton kidolgozott egy programozási gyakorlatot a diákjai számára, amely később fuzzer néven vált ismertté. Mivel a történetben kifejtett jelenségek a mai napig relevánsak számunkra, így napi szükség van a fuzzerek alkalmazására a különböző programokon. [17]

A működését három fő lépésre bonthatjuk:

- A bevitelnek az előkészítése
- Az bemenet eljuttatása a célrendszerre
- A rendszer megfigyelése

2.2 ábra - A fuzzerek működése [17]

A bevitel előkészítése folyamán egy adatbázisból beolvasva vagy legenerálva összegyűjtik a szoftvertermékbe küldeni kívánt bemeneteket, majd azokat egymás utáni vagy véletlenszerű sorrendben eljuttatják a célrendszerre. Ennél a lépésnél lehetséges kibertámadást is végrehajtani, például egy Denial of Service támadást, amikor a rendszer nem tudja kezelni megfelelően azt a nagymennyiségű adatot, ami beérkezik. Ez végül egyes funkciók elvesztéséhez vagy akár a szoftver teljes leállásához is vezethet. Ha nem

lépett fel probléma a tesztelés során, akkor a rendszert megfigyelik és a későbbiekben a naplózott elemek kiértékelésre kerülnek. [16]

A tesztelési folyamat során több alapfogalmat is szükséges megkülönböztetni:

- Attack surface,
- Trust boundary,
- Input source = input space.

Az *attack surface*, vagyis támadási felület a kód azon részét jelenti, amit a támadó el tud érni. A biztonsági szakemberek számára az elsődleges feladat a támadási felület kiterjedtségének a meghatározása. Vannak olyan belső részek, amiket képtelenség külső adatokkal, bemenetekkel módosítani, ezáltal az előbb említett részeket sérülékenységi szempontból vizsgálni tesztelés alatt nem szükséges. A belső kódot, amit módosítani nem lehet, természetesen más tesztelési technikákkal szükséges szintén ellenőrizni. [18]

A *trust boundary*, vagyis megbízhatósági határ, egy erőforrásokhoz fűződő engedélyek halmaza, melyben az adatok egyik megbízhatósági szintről kerülnek a másikra. Például felhasználói módból átlépni egy operációs rendszer kernelébe szintén egy megbízhatósági határ átlépése. A processzor megbízik a kernelben, ezáltal szinte az összes funkcióját lehetséges elérni, viszont felhasználói módban ennek a töredéke használható. A kiberbiztonsági szakemberek a megbízhatósági határoknál is a sérülékenységet keresik, és ez alapján optimalizálják a bemeneteket a fuzzing tesztelés során. [18]

Az *input source* vagy *input space* kifejezések nagyon hasonlóak, hiszen mindkettőnek a jelentése arra irányul, hogy az adatok hogyan lesznek leképezve és kézbesítve. Az *input space*, másnéven beviteli tér végtelen, az minden egyes lehetséges permutáció összesége. Különböző feltételek segítségével lekorlátozhatjuk ennek a végtelen térnek a méretét, amivel egy tesztelési célokra felhasználható halmazt kapunk. Továbbá a múltban felfedezett hibák regressziós használata is segíthet a sérülékenységek feltérképezésében. [18]

A fuzzer típusok között megkülönböztetünk még generációs (generation-based) illetve mutációs alapú (mutation-based) fuzzereket. A generációs alapú megoldás fehér dobozos eljárást tesz nekünk lehetővé, mely alkalmazza az ismert protokollok struktúráját és felépítését a tesztesetek felépítéséhez. A mutációs alapú megoldás fekete dobozos

módszer, mely teljesen véletlenszerűen készíti el a bemeneteket. A generációs alapú teszteléssel kiterjedtebb lefedettség érhető el, illetve több teszteset lesz sikeres, mint a mutációs alapúnál, viszont nagyobb költségekkel jár. A két módszer ötvözése használható szürke dobozos tesztelési eljárásként. A későbbi fejezetekben látható, hogy amíg az Információs Technológia világában a fuzzing egy teljes mértékben bevett szokás, addig az autóipari szoftverekben még mindig ritkaságnak számít. Ennek az oka abból eredeztethető, hogy az IT világában alkalmazott fuzzing eszközök nem feltétlenül alkalmazhatók autóipari környezetben, illetve a kiberbiztonság új területnek számít az autóipari szegmensben. [1]

A tesztek végrehajtása során fokozott óvatossággal kell eljárni, ha "élő" rendszert tesztelünk, mivel a folyamat során az összeomolhat, vagy esetleg adat kompromitállódás is történhet. A cégek, gyártók ezért úgynevezett tükörrendszereket szoktak felállítani, amiket veszélyek nélkül lehet tesztelni, hiszen az az eredeti másolata. Ezzel a módszerrel a támadó perspektívájából rengeteg sérülékenységet lehet felfedezni. [19]

A gyakorlatban megkülönböztetünk még intelligens és nem intelligens fuzzereket [18]. A dolgozatom fő témája az intelligens témakörrel foglalkozik, melyekről a következő fejezetben lesz részletesen szó.

2.2.4. Következtetések

A modern járművek egyre több funkcióval képesek kielégíteni a vásárlói igényeket, amely kiberbiztonsági szempontból a támadási felületek számának növekedését is eredményezi. Mivel az egyes interfészek, komponensek könnyen kibertámadások célpontjaivá válhatnak, így kiemelten fontos azok átfogó biztonsági tesztelésének a végrehajtása, amely során a sérülékenységeket feltérképezik, és amelynek eredményeképp biztonsági intézkedéseket vezetnek be a sérülékenységek javítására.

A teljesség igénye nélkül három tesztelési technikát vizsgáltam meg. Ezeket a lentebb található táblázatban egységes szempontok szerint összehasonlítottam, és arra a következtetésre jutottam, hogy mivel minden tesztelési technikának megvannak a maga előnyei és hátrányai, így az átfogó tesztelés végrehajtásához érdemes minél több technikát alkalmazni a termékfejlesztési életciklus során.

	Penetrációs Teszt	Vulnerability Scanning	Fuzzer
Cél	Célzott eszközön, adott hibák megtalálása, felfedezése	Az összes sérülékenység, hiba azonosítása	Rendszer sérülékenységeinek megtalálása véletlenszerű adatokkal
Eljárás módja	Mindenképp szükséges manuális tesztelés	Teljes mértékben lehet automatizált	Automatizált
Idő	Több nap is lehet, de megszabott időkeret van általában	Általában több óra	Változó időtartam, a tesztelőktől függően

2.2 táblázat - A módszerek összehasonlítása

3. Adaptív fuzzing

A fejezet fő témája az adaptív fuzzing részletes ismertetése, amely egy olyan, általában fekete doboz módszertant alkalmazó tesztelési eljárás, amelynek célja a biztonsági rések felfedezése, az esetleges hibák megtalálása véletlenszerű bemenetek felhasználásával. A tesztelési eljárást végrehajtó SW-t fuzzernek nevezzük, amelynek többféle típusát különíthetjük el: egyszerű és okos (smart). Ez utóbbit a szakirodalomban gyakran illetik intelligens vagy adaptív fuzzer elnevezéssel is. Dolgozatom további részében az adaptív fuzzerek kifejezést használom. A fő különbség az egyszerű és az adaptív között az, hogy az adaptív esetében a szakemberek előre meghatározott modelleket és heurisztikát is alkalmaznak annak érdekében, hogy hatékonyabban találjanak sérülékenységeket. [20] Jelen fejezetben az adaptív fuzzerek részletes ismertetésén túl bemutatásra kerülnek olyan különböző keretrendszerek, melyek támogatják az okosított verzióját a fuzzing-nak, illetve az IT és az autóipar világában használt eljárások is.

3.1. Adaptív fuzzing technika

Az adaptív fuzzing lehetővé teszi, hogy az automatizált tesztelőprogramok hatékonyabban fedezzék fel a hibákat, miközben minimalizálják a hamis pozitív eredmények számát. [21]

Az okos fuzzerek használata számos területen alkalmazható, beleértve az adatvédelmet, információbiztonságot és a SW-es minőségellenőrzést is. A technika alapelve, hogy a biztonsági szakember valamilyen "okos" megközelítéssel felruházza az egyszerű fuzzert, amely ezáltal több hibát és nagyobb arányú sebezhetőséget talál, mint az elődje. A teljesség igénye nélkül néhány példa, amely az adaptív fuzzing megközelítés tárgykörébe tartozik: olyan heurisztikák alkalmazása, melyek a bemenetet előzetesen modellezik; olyan algoritmusok használata, amelyek megtalálják a bemeneteken fellelhető mintázatokat, és ezeket felhasználja a további bemenetek generálásához. Már csak a fenti példákat figyelembe véve is kijelenthető, hogy a fuzzer képes tanulni a kimenetről, és esetlegesen hatékonyabb bemeneti értékeket generál. Az előbb említett tanulási folyamatot gépi algoritmusok segítségével lehet elérni. Az okos fuzzing technika alkalmazásának több előnye is van az elődjeihez képestt, amelyeket az alábbi látható táblázatban összegeztem. [20]

	Egyszerű fuzzer	Okos fuzzer
Idő	több	kevesebb
Kód lefedettség	kisebb	nagyobb
Hibakeresés	kevésbé hatékony	hatékony
Hatékonyság	rosszabb	jobb
Biztonság	rosszabb	jobb

3.1 táblázat – Fuzzerek összehasonlitása [20]

A fent látható táblázat első szempontja az időhatékonyságra helyezi a fő hangsúlyt, mivel a tesztelési folyamat során ez rendkívül fontos tényező. Az okos fuzzerrel időt lehet megtakarítani az egyszerűvel szemben, mivel kevesebb tesztelési ciklusra van szüksége, hogy elérje ugyanazt az eredményt. A kód lefedettség szintén nagyobb az adaptív esetében, hiszen protokoll-specifikus információkat is meg lehet tanítani az adaptív fuzzernek, ezáltal pontosabb tesztelés végezhető el. [22]

A hibakeresés szempontjából is ajánlott az okos fuzzerek alkalmazása, mivel a tesztelés során elemzi és modellezi az adatbemeneteket, ezáltal több olyan hibát is képes felfedezni, amit egy egyszerű fuzzer képtelen lenne. A tesztelési folyamat során a hatékonyság is fontos szempont. Mivel az okos változat képes kiválasztani azokat az adatbemeneteket, amelyek a legnagyobb valószínűséggel fognak hibát okozni, így nagyobb hatékonyságot biztosít, mint az egyszerű adaptáció. A biztonsági rések felfedezésére is jobb az előbb felsorolt tényezők alapján az adaptív fuzzer. Végeredményként az adaptív fuzzerek nagyobb esélyt adnak a biztonsági rések felfedezésére, mint az egyszerű fuzzerek, ezáltal lehetőséget biztosítanak a sérülékenységek javítására, és közvetett módon így biztonságosabb programokat, alkalmazásokat és rendszereket eredményeznek, mint az egyszerű fuzzerek.. [23]

Az adaptív megoldási technikának is vannak hátrányai, mivel egyszerre csak egy adott protokollra lehet tesztmintákat generálni, azzal párhuzamosan nem alkalmasak más protokollok tesztelésére. Az előbb említett ok miatt elmondható, hogy az adaptív fuzzing-nak speciális, célzott tervezése van, nem egy rugalmas tesztelési eljárás, nem képesek lefedni a rendszer minden egyes lehetséges mintáját, ezért kiegészítő teszteléseket szükséges elvégezni mellé. [23]

Összeségében elmondható, hogy mind az egyszerű, mind az adaptív technika hatékony. Egy okos fuzzer megalkotására több időt kell allokálni, komolyabb szakértelmi ismeretekre van szükség a létrehozásnál, viszont a későbbiekben célravezetőbb automatizált tesztelési folyamat érhető el a segítségével.

3.2. Adaptív fuzzing az iparágakban

Az adaptív fuzzerek alkalmazása számos iparágban hatékony automatizált tesztelési eljárást biztosít. A legtöbb ma ismert ipari szektorban, ágazatban hasznát lehet venni az adaptív fuzzereknek. A dolgozatomban az IT és az autóiparban használatos megközelítésére helyezem a hangsúlyt.

3.2.1. Adaptív fuzzing az IT világában

Az *Information Technology* (IT) vagy *Információs Technológia* a számítógépes rendszerekkel és technológiákkal kapcsolatos tevékenységek összesége, melyet számos területen alkalmaznak. A mai modern társadalomnak már elengedhetetlen velejárója. Az IT a szoftverek, hardverek és hálózatok használatával biztosítja a hatékonyabb adatkezelést, viszont számos biztonsági kockázatot, sérülékenységet rejt magában. A fuzzerek és az IT között szoros kapcsolat áll fenn az előbb említett problémák miatt, melyeket a szakemberek biztonsági tesztelésekkel próbálnak kiszűrni, illetve javítani. Az okos fuzzerek lehetővé teszik, hogy automatizált módon teszteljék a protokollokat, alkalmazásokat, rendszereket. A szakembereknek segítséget nyújt a rendszerek megértésében, és ezáltal az esetleges biztonsági rések kiküszöbölésében. Az Információs Technológia világában számos adaptív fuzzer áll rendelkezésre, melyet adott tesztelési feladathoz is lehet akár igazítani. A következő alfejezetekben bemutatok néhány, az IT világában alkalmazott fuzzert.

3.2.1.1 American Fuzzy Lop

Az American Fuzzy Lop, röviden AFL egy nyílt forráskódú fuzzer eszköz, amely dinamikus bináris tesztelést alkalmaz, illetve a tesztvektorokat figyeli, hogy melyik vezet a legtöbb elágazáshoz. Linux operációs rendszeren alkalmazható. Az eszköz képes az alkalmazásokat automatizáltan tesztelni anélkül, hogy a forráskódot meg kellene változtatni. A tesztelési folyamat alatt az eszköz akár több ezer tesztesetet is képes elvégezni, képes tanulni a tesztelés során, és ezáltal optimalizálni a tesztelési folyamatot olyan módon, hogy a lehető legtöbb sérülékenységet és hibát felfedezze. [25]

Több változata is létezik az AFL-nek, például rendelkezik kifejezetten *smart* változattal, mely egy továbbfejlesztett változata a sima AFL-nek. Az előbb említett eljárás gépi tanuláson alapuló algoritmust alkalmaz a fuzzing tesztekhez, amely így lehetővé teszi, hogy adaptív módon változtassa a tesztek generálásának stratégiáját. Az AFLSmart képes az előző fuzzing tesztesetek eredményeiből tanulni és elemezni őket, így javítva a tesztelés hatékonyságát és hibafelfedezési rátáját. [23]

Emellett az AFLSmart a sima AFL-el szemben tartalmaz egy kifejezetten adatgyűjtésre és elemzésre tervezett modult, hogy további javításokat, finomhangolásokat lehessen végrehajtani a tesztelés során. [25]

3.2.1.2 Peach

Az IT iparágban egy gyakran használt fuzzer a Peach, amely egy olyan zárt forráskódú keretrendszer, melyet eredetileg a protokollok és interfészek tesztelésére fejlesztettek ki. XML alapú konfigurációs fájlok használatával támogatja a bemenetek létrehozását protokollok alapján. [21]

Fontos kiemelni, hogy a szoftvercsomag ingyenes annak ellenére, hogy zárt forráskódú, viszont MIT engedélyköteles. A Peach moduláris architektúrája lehetővé teszi a tesztelők számára, hogy saját eszközeiket és tesztpéldáikat alkalmazzák, továbbá könnyen integrálható más fuzzerekkel is. Az előbb említett American Fuzzy Lop-hoz hasonlóan a Peach fuzzer is rendelkezik egy intelligens változattal, ezáltal a felhasználók számára lehetővé teszi a hatékonyabb tesztelést, adatgenerálást, adatok validálását, illetve a hibák automatikus azonosítását. Az okos módszer alapja a generatív fuzzing, mely biztosítja a programnak, hogy tanuljon az adatok struktúrájából, és ezek segítségével alkossa meg azokat az új adatszerkezeteket, melyek legenerálásra kerülnek. A Peach generatív fuzzingon túlmenően mutációs alapúra is képes. [26] Több mint 15 különböző protokollt támogat, mint például TCP/IP-t vagy a http-t, illetve képes azonosítani a protokollhibákat is [27]. Automatikusan generálja az új teszteseteket a hibák reprodukálására és javítására egyaránt. A Peach folyamatosan tanul és alkalmazkodik a célprogramhoz, kiváló eszköz lehet a hibák analízisére a tesztelők számára.

3.2.1.3 Sulley

A fent említett két megoldáson kívül, specifikusabban hálózati kommunikációra és protokollok tesztelésére fejlesztették ki a Sulley fuzzert. A Sulley egy Python programnyelven írt keretrendszer, mely lehetővé teszi az előbb említett funkciók

tesztelését. Könnyen használható és egyszerűen bővíthető, így a felhasználók testre szabhatják a tesztelési környezetüket az egyéni igényeikhez és céljaikhoz. A Sulley segítségével lehetőség van alkalmazások által használt protokollok, például TCP vagy SMTP tesztelésére. [21]

Előnye abban rejlik, hogy könnyen alkalmazható, a szélesen elterjedt Python programozási nyelvnek köszönhetően. A tesztek automatizálhatóak? ennek okán a folyamat meggyorsítható. Mindezek mellett alkalmazható Windows-os környezetben. Hátránya a Python sajátosságában rejlik, mivel a többi nyelvhez képest a Python lassúnak számít, ezáltal a tesztek lefutásának az ideje is időigényesebb. [28]

3.2.2. Adaptív fuzzing az autóiparban

A járművekben egyre több elektronikus vezérlőegység, vagyis ECU található, melyek különböző funkciókat látnak el. Az előbb említett elektronikus vezérlőegységekben futó szoftverek biztonságának ellenőrzése kiemelt fontossággal bír, mivel biztonságkritikus rendszerként tekintünk rájuk. [6]

Az autóiparban használt okos fuzzereknek magasabb szintű biztonsági szintet kell elérniük, mivel az alkalmazott szoftverek biztonságos és hatékony működése létfontosságú.

Az előző fejezetben bemutatott fuzzereket az autóiparban is alkalmazzák, de a legtöbb vállalat a változatos fejlesztési környezetek miatt saját fejlesztésű adaptív megoldásokat használ. Ebből következik, hogy az adaptív fuzzerek-nek az alkalmazása az autóiparban nehezebben kivitelezhető, mint az IT világában. Egy alkalmazás futtatása sokkal több időt vehet igénybe, mint az IT területen. Az egyes programok, alkalmazások speciális eszközöket igényelnek, tehát általában a tesztelés sokkal specifikusabb, illetve az aktuális biztonsági előírásokat és szabványokat be kell tartani.

Különböző gyártók által használt protokollok és interfészek jelentős eltérést mutathatnak, ezáltal az adaptív változatnak képesnek kell lennie ezekhez alkalmazkodni. Az autóipari adaptív fuzzernek rendelkeznie kell megfelelő sebességgel és skálázhatósággal a tesztesetek végrehajtásához. A gyártók akár saját egyedi tesztkörnyezetet is létrehoznak a tesztesetek implementálásához és végrehajtásához, melyet kombinálnak az adaptív fuzzerrel, ezáltal biztosítva a termék megfelelő minőségét és biztonságát.

Az autóipari alkalmazásokhoz használt adaptív fuzzerekről nincsen sok információ megosztva, mivel általában zárt forráskódúak, számos szellemi magántulajdont tartalmaznak. Az előző fejezetben tárgyalt nyílt forráskódú megoldások alkalmazhatóak autóipari területen is, illetve léteznek még más keretrendszerek, melyek alkalmazhatóak tesztelésekre, mint például a CanFuzz, CANoe. Fontos megjegyezni azonban, hogy az előbb említett példák nem biztosítanak okos megoldási lehetőséget.

Az AFL és a Peach okos változatát is lehet használni autóipari vonatkoztatásban, de kiegészítő eszközként meg lehet említeni az EffCAN-t, a VulFuzz-t illetve a CaringCaribou-t is. Természetesen az adaptív változat integrálásával az autógyártó cégek a tesztelési folyamat automatizálása mellett a költséghatékonyságot is megcélozzák, ennek megfelelően próbálják megtalálni a számukra tökéletes eszközöket. A korábban kifejtettek lehetővé teszik a különböző fuzzing tesztesetek implementálást és alkalmazását például járművezérlő szoftvereken vagy akár szórakoztató rendszereken is.

Az EffCAN hatékony alkalmazásához szükséges az ECU szétszerelése, hogy vezérlőfolyamat grafikonokat (control flow grafikon) lehessen megalkotni a hatékony fuzzing tesztelés eléréséhez. Az eszköz szétszerelése viszont komoly feladat és kihívás, mely igényel bizonyos szintű szaktudást. A kódrégiók illetve a hardver-specifikus driverek/illesztőprogramok szétválasztása a szoftver összetevőkből komoly kódanalízist igényel, mely szintén szakértelmet vár el a tesztelőtől. A tesztelőnek az adott ECU-ról és annak architektúrájáról nagy tudással kell rendelkeznie, különben az EffCAN nem alkalmazható hatékonyan. [3]A megközelítés javíthatja a kód lefedettséget, és jobb megoldást kínál, mint a véletlen mutációs stratégia. [1]

A vulFuzz egy olyan keretrendszer, amely kifejezetten az autonóm járművek sebezhetőségi réseire specializálódott. Legtöbb esetben a fuzzing tesztelések folyamán, a kódlefedettséget célozzák meg a tesztelők. Egy tanulmány alapján a vulFuzz több eseményt képes észlelni, mint az AFL, ugyanolyan kódlefedettség mellett. Azt viszont fontos megemlíteni, hogy az AFL nem támogatja a Python programozási nyelvet. [3]

A CaringCaribou egy autóipariban igen elterjedt keretrendszer, amely Python programozási nyelven íródott, és amellyel különböző biztonsági teszteléseket lehet végrehajtani. [29] A CaringCaribou egy nagyobb kutatómunkának a része, melynek a neve HEAVENS (Healing Vulnerabilities to Enhance Software Security and Safety). Ennek elsődleges célja, hogy minél több biztonsági rést lehessen felfedezni autóipari rendszerekben, továbbá másodlagos célként módszertanokat és eszközöket is kínál a

biztonsági értékelésekhez. [30]A CaringCaribou kettő fuzzing modullal rendelkezik. Az egyik, az eredeti fuzzer modul, egy egyszerű megoldást kínál a CAN hálózatok teszteléséhez, több módszerrel is kisegíti a tesztelőt. Például ilyen funkció a véletlenszerű adatgenerálás, vagy az ismétlés (replay), mely visszajátssza az előző fuzzing naplózott fájlt. Az autoFuzz modul már okosabb megoldást biztosít, mivel képes azonosítani azt, hogy melyik naplózott fájl váltott ki speciális eseményt. [31]

Az előbb említett módszerek közül az EffCAN és a VulFuzz kiváló lehetőséget nyújt a fuzzing tesztelések végrehajtásához, viszont a probléma az, hogy ezeknek az intelligens változata egyelőre nem érhető el a fejlesztés hiánya miatt. A jövőben elképzelhető, hogy a fejlesztések előre haladtával az okos változat is meg fog jelenni.

A járműiparban a biztonsági tesztelések végrehajtása egyre komplexebb feladat, a tesztelőknek rengeteg kihívással kell szembenézniük, mint például a rendszer összetettsége és mérete vagy a bemenet/kimenet fluktuációja. [3]

3.3. Következtetések

Általánosságban elmondható, hogy az adaptív fuzzerek alkalmazása a tesztelési folyamatok során kifizetődő mind az IT, mind az autóipar számára. Az elkövetkezendő időkben is fontos szempont lesz a hatékonyabb, és ezáltal nagyobb lefedettséget elérő tesztelések létrehozása. Az intelligens fuzzerek világa a jövőben elképzelhetően még ennél is nagyobb szerepet fog kapni az egyes ipari szektorokban.

A dolgozatom következő fejezetében bemutatásra kerül egy az UDS protokollal kapcsolatos elvi teszteset felépítése, illetve egy szolgáltatás elméleti tesztesetének reprezentációja.

4. Adaptív fuzzing az UDS-en belül

Az alábbi fejezetben bemutatom az általam választott két UDS szolgáltatás elméleti teszteseteinek a felépítését adaptív fuzzer felhasználásával. A teljes megértés érdekében a fejezet tárgyát képzi a Security Access-hez való hozzáférés módjának ismertetése. az NRC kódokból tanuló adaptív fuzzer felépítésének a kifejtése, valamint kutatásom továbbfejlesztési lehetőségeinek és következtetéseinek a levonása a jövőbeli eredményes és hatékony felhasználáshoz.

4.1. Adaptív fuzzing elméleti lépései

Az adaptív fuzzer tervezésénél elsődleges elképzelésem az volt, hogy egy olyan öntanító fuzzert hozzak létre, amely a gyűjtött és kapott adatok alapján képes automatikus módon finomhangolni a tesztelési folyamatot.

A UDS egy eseményvezérelt protokoll, amely kizárólag a kiküldött üzenetekre válaszol. A diagnosztikai üzenetek segítségével különböző műveletek hajtódnak végre, mint például a memóriaírás vagy -olvasás vagy a vezérlőegység kalibrálása. A diagnosztikai kéréseket az úgynevezett Tester küldi, melyekre az ECU válaszol. A vezérlőegység sikeres végrehajtás esetén pozitív, míg sikertelen válasz esetén olyan negatív választ küld vissza a Tester-nek, amely tartalmazza az úgynevezett Negative Response Code-ot (NRC-t). Az NRC egy olyan hibakód, amely segít azonosítani a Tester-nek, hogy miért volt sikertelen a végrehajtás. Részletesebb kifejtéssel az olvasó 0 fejezetben találkozhatott.

A tesztesetek implementálása előtt érdemes modellezni az előzetesen összegyűjtött adatokat a hatékonyabb felhasználás érdekében. Ez hozzájárul ahhoz is, hogy az ECU által elérhető UDS szolgáltatásokat és a negatív válaszok során jelentkező NRC-ket rendszerezve lehessen áttekinteni, amely egy fuzzer tervezéséhez elengedhetetlen. Az adatok tárolására célszerű egy olyan adatbázist létrehozni, amely adott bemenethez a hozzátartozó kimenetet rendeli. Jelen esetben ez azt jelenti, hogy az adott UDS szolgáltatással kell párosítani a megfelelő pozitív és negatív válaszkódokat (NRC).

A hatékony adaptív változatok létrehozásához szükséges egy előre definiált stratégiát megalkotni, melynek főbb pontjai mentén a tesztelés hatékonyan elvégezhető.

Az általam alkalmazott stratégia a fuzzing alapelveit vette figyelembe, melyek a következők:

A tervezett algoritmus lépései:

1. Az adaptív fuzzer az előre elkészített, bemeneti és kimeneti párokat tartalmazó adatbázis alapján megtanulja, hogy melyik UDS szolgáltatáshoz melyik lehetséges válaszok tartozhatnak.
2. A fuzzer különböző típusú teszteseteket generál, melyeket az ECU felé továbbít.
3. A fuzzer (Tester) által küldött diagnosztikai kérésnek megfelelően az ECU lefuttatja az adott UDS szolgáltatást.
4. Az adaptív fuzzer a megtanult adatok alapján kiértékeli az ECU válaszát, és összeveti a lehetséges válaszok halmazával.
5. Az algoritmus különbséget tesz a sikeres (Passed) és sikertelen (Failed) végkimenetelű tesztek között.
6. A fuzzer az NRC kódból nyert információkat felhasználja további tesztadatok generálásához.
7. A tesztelési folyamat alatt született eredményeket egy szakember kiértékeli, és az esetlegesen felmerülő hibákat javítja, mely tovább pontosítja az algoritmus működését.

Fontos megemlíteni, hogy a negatív válaszkódokat két kategóriába lehet sorolni, az általános, illetve gyártóspecifikus negatív válaszkódok csoportjába. Az előbbi az UDS protokoll általános hibáira utal, míg az utóbbi pedig a gyártó által definiált konkrét hibára mutat.

A következő példákban, melyek bemutatásra kerülnek, általános megközelítés kerül alkalmazásra, nem cég specifikus esetleírások.

4.1.1. Teszteset - Adaptív fuzzer felépítése NRC-k segítségével

Az alfejezetben egy lehetséges elméleti példát mutatok be az UDS protokollon belüli negatív válaszkódokból való adaptív fuzzer felépítéséhez. Fontos megemlíteni, hogy a teljesség igénye nélkül, pontokba szedve részletezem a fuzzer felépítést annak

érdekében, hogy a későbbiekben a teszteset gondolatmenete átlátható és közérthető legyen.

Adaptív megoldást olyan algoritmusra lehetne építeni, mely képes adatgyűjtésre és NRC-k analizálására. A lépések egy ilyen fuzzer működéséhez a következők lehetnének:

1. Az UDS-ből negatív válaszkódok gyűjtése.
2. A kódok szétbontása két kategóriába: általános és gyártóspecifikus kódok.
3. Tanítóhalmaz elkészítése, mely tartalmazza az előbb említett hibakódokat.
4. Az algoritmus létrehozása, mely a tanulási folyamata során képes az adathalmazból megérteni és kategorizálni a hibákat.
5. A tanítási folyamat végén az algoritmus képes lesz a hibák értelmezésére, melynek segítségével a későbbiekben lehet javítani a fuzzer működését.

Kiemelt figyelmet kell fordítani a 4. pontban leírtakra, mely az adaptív fuzzerek létrehozásának sarkalatos pontja. Az algoritmus az adathalmaz által összegyűjtött információk alapján tanul és alkalmazza a gépi tanulás módszereit azért, hogy automatikusan kitalálja a tesztelendő rendszer működését, így képes megérteni a rendszer kommunikációs protokollját, az adatcsomagok szerkezetét és azokat a visszajelzéseket, amelyeket a rendszer ad az adatok feldolgozásakor. Az algoritmus a tanulás során kialakít egy belső működési modellt, amely képes előre jelezni a rendszer reakcióit a különböző adatkombinációkra, és így hatékonyabban tudja felderíteni a hibákat és biztonsági problémákat a rendszerben.

Ezt követően a fuzzer a szintén kiemelendő 5. pontnak megfelelően működik tovább. Az itt leírt értelmezésre való képesség az adathalmazból történő tanulás és a gépi tanulás módszereinek alkalmazása által jön létre. Az algoritmus képes felfedezni az adathalmazban található hibákat, megfigyelni azt, hogy milyen adatkombinációk eredményezik őket, majd ezt az egészet megjegyezni, mely által a fuzzer folyamatosan tanítja önmagát. Így a folyamat végére egy olyan modell fog létrejönni, amely képes előre jelezni azt, hogy a rendszer milyen válaszokat fog küldeni különböző adatkombinációk esetében. Az algoritmus tehát képes értelmezni a hibák jelzéseit azáltal, hogy felismeri az adathalmazban található mintákat, amelyekből a hibák előfordulása megjósolható.

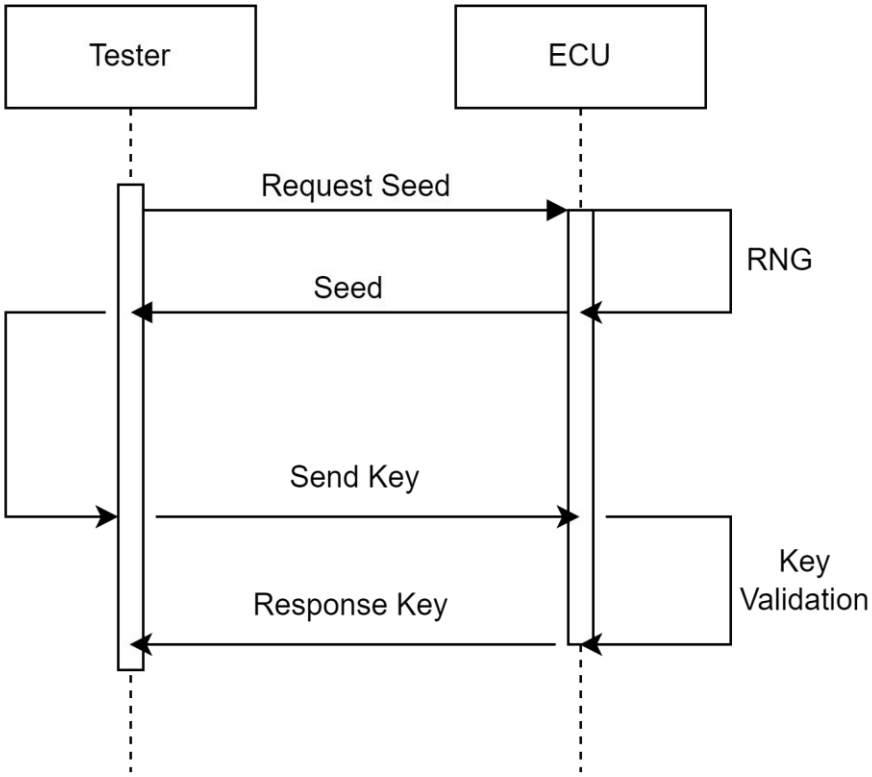
Egy NRC-kből tanuló fuzzer lefejlesztése szakértői tudást igényel, viszont a sikeres megvalósítása a jövőben nagyban hozzájárulhat a hatékonyabb tesztelés eléréséhez.

4.1.2. Példa - Security Access hozzáférés adaptív fuzzing segítségével

A Security Access (biztonsági hozzáférés) az egyik olyan UDS szolgáltatás, amely lehetővé teszi, hogy bizonyos funkciók csak autentikáció után legyen elérhetők az ECU-ban. Security Access-t használják arra, hogy a szoftver frissítéshez és a kulcscseréhez tartozó UDS funkciókat csak autentikáció után lehessen elérni.

A Security Access szolgáltatás négy fő részből áll:

1. Request Seed (teszter kéri): Ezzel kérünk hozzáférést az ECU-tól bizonyos funkciókhoz. Emellett ezzel kérünk az autentikációhoz egy véletlenszerű számot. A kérésben szerepel az, hogy melyik szinthez (security level) szeretnénk hozzáférést kérni.
2. Send Seed (szerver küldi) : Az ECU megvizsgálja az előre definiált feltételeket, hogy lehetséges a biztonsági hozzáférés, azaz a körülmények adottak, akkor elküld egy véletlen számot.
3. Send Key: A teszter a kapott véletlen számot a közös szimmetrikus kulccsal titkosítja, majd válaszüzenetben ezt küldi vissza az ECU-nak.
4. Response Key: Az ECU a korábban kiküldött véletlenszámot titkosítja a szimmetrikus kulccsal, és ezt összeveti a tesztertől kapott kulccsal. Ha a kapott kulcs és a számított kulcs megegyezik, a biztonsági szint feloldásra kerül.



4.1 ábra Seed-Key kihívás

Összefoglalva, a Security Access UDS szolgáltatáson keresztül tudja bizonyítani a teszter, hogy a szimmetrikus kulcs birtokában van.

Ebben a példában az okos fuzzer segítségével a biztonsági funkció(0x27) korlátozására, kulcsproblémákra és kulcskezelésekre helyezem a hangsúlyt, mely kiberbiztonsági szempontból kiemelt szereppel bír. Az alkalmazása lehetővé teszi a diagnosztikai szoftverek számára, hogy különböző kulcsokat és paramétereket próbáljanak ki, és így felfedezzék a hibákat és sérülékenységeket, amelyeket a rendszerben ki lehet használni.

Az UDS Security Access protokoll lehetővé teszi az ECU számára, hogy az autentikációhoz kötött UDS szolgáltatásokat engedélyezze vagy letiltsa a biztonsági kulcsok használatával. Az intelligens fuzzer algoritmus különböző biztonsági funkciókat tesztelhet, beleértve a helyes és helytelen kulcsokat, kulcshossz korlátozásokat és érvénytelen kulcsformátumokat. Az NRC kódok segítségével az algoritmus hatékonyan dönthet arról, hogy tovább lép-e a következő tesztesetre, vagy folytatja-e a funkciók tesztelését.

Az előző tesztesetből adódhat a következő, melyben az esetleges kulcsproblémákra helyezzük a hangsúlyt, amely lehetnek a következők:

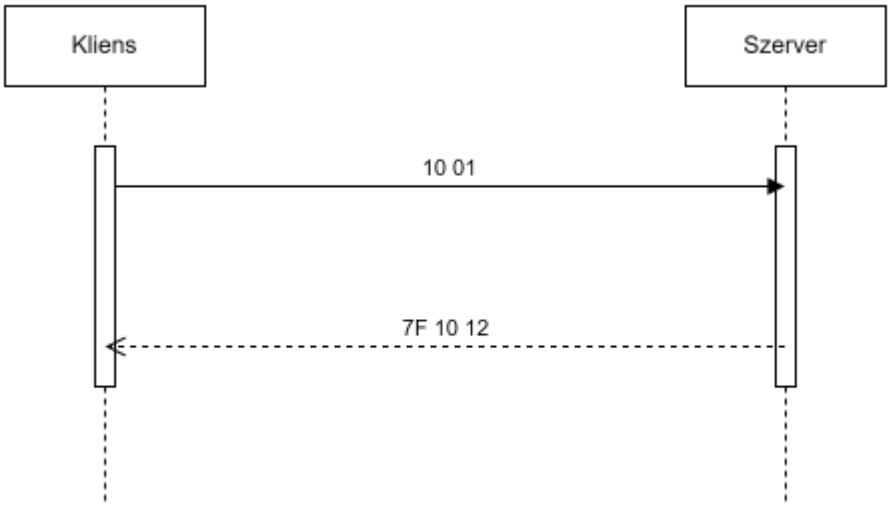
- **Nem megfelelő kulcshossz használata**: a teszteset olyan kulcsokat generálhat, amelyek nem felelnek meg az ECU által meghatározott kulcs hosszoknak, hogy megnézze, hogyan reagál a rendszer, és milyen NRC kódokat ad vissza.
- **Helytelen biztonsági kulcs megadása**: a teszteset különböző érvénytelen kulcsokat generálhat, hogy ellenőrizze, hogyan reagál a rendszer, és milyen NRC kódokat ad vissza.
- **Érvénytelen kulcsformátum használata**: a teszteset olyan kulcsokat generálhat, amelyek nem felelnek meg az ECU által meghatározott kulcsformátumnak, hogy megnézze, hogyan reagál a rendszer, és milyen NRC kódokat ad vissza.
- **Helyes kulcsok használata**: a teszteset helyes kulcsokat generálhat, és megnézheti, hogyan reagál a rendszer, és milyen NRC kódokat ad vissza.
- **Különböző kulcsok kombinálása**: a teszteset különböző kulcsok kombinációit használhatja, hogy megnézze, hogyan reagál a rendszer, és milyen NRC kódokat ad vissza.

Az előbb felsorolt példákon túl még rengeteg megvalósítási és tesztelési ötlet is felmerülhet, melynek csak a tesztelői csapat képzelete és kreativitása szab határt. További feladatokat adhat speciális felhasználói kérés.

4.2. Tesztesethez szükséges tanítóhalmaz

Az alábbi fejezetben bemutatom a tesztesetek implementálásához szükséges tanítóhalmazokat, melyek szükségesek lesznek a későbbiekben a konkrét tesztesethez.

Az UDS protokoll CAN hálózaton keresztül valósul meg - részletesebben 1.5.1 fejezetben található információ -, ezért a fuzzernek meg kell tanulnia, hol jön létre a kommunikáció, és a továbbiakban ott kell majd próbálkoznia. Ha egy átlagos fuzzerrel végeznénk el a tesztelést, nem lenne képes meghatározni automatikusan a tanítóhalmazból hol jön létre a kommunikáció, folyamatos szkennelést hajtana végre 0x0000-0xFFFF között az összes munkamenetet végig próbálva a CAN hálózaton.

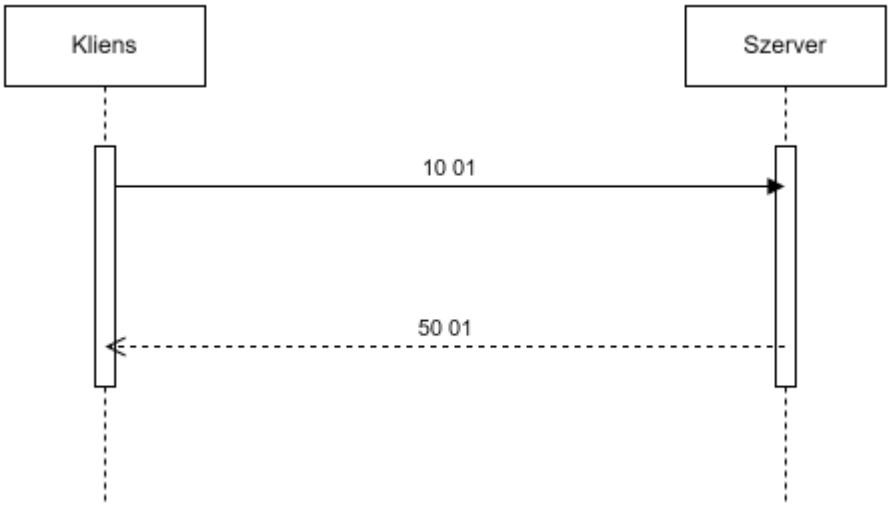


4.1 ábra - CAN kommunikáció

Amikor a tesztelő, vagy kliens a szervernek, ami jelen esetben az ECU, adatot küld az UDS protokoll segítségével a szolgáltatás egy azonosítóként fog megjelenni – röviden SID -, mely után további paraméterek következhetnek. Az elküldés után két lehetséges kimenet van, az első, hogy pozitív választ kapunk, a másik, hogy negatívat. A kommunikációban, hibakeresésnél hatékonyan meg lehet találni a 0x7F miatt, ugyanis ez a SID előtt megtalálható lesz, mint egy újabb azonosító, mely hibát jelez. Az UDS egyik gyengeségének szokták említeni, hogy a negatív válaszkódjai sokat sejtetnek egy esetleg támadó számára, például a szerver által visszaküldött válaszban a SID után megjelenik maga a negatív válaszkód is, melynek segítségével könnyedén rákereshetünk a konkrét hibára. A fenti képen látható a könnyebb megértés érdekében egy példát a negatív válaszkódra.

A 0x7F megjelenik a SID előtt, a 0x12 hexadecimális érték pedig egy hibakódra utal, mely ebben az esetben a sub-functionNotSupported, aminek a jelentése, hogy a művelet nem fog végrehajtódni (jelen esetben munkamenet váltás lenne a fő cél), mert a szerver nem támogatja a specifikus paramétereket. A 0x12-es negatív válaszkódnál a kliens egy ismert szolgáltatás azonosítót küldött, viszont az megjelenő alfunkció mellett a szerver számára ismeretlen.

Ebben az esetben a pozitív válaszkódok is fontos szereppel bírnak, a felismerésük a tesztelők számára nem jelenthet komoly kihívást, mivel a SID értékéhez 0x40-es értéket adunk. Tehát például, ha a 0x10 0x01 kerül elküldésre a szerver által, a válasz 0x50 0x10 lesz, melyet a következő képen is látható:



4.2 ábra - Pozitív válaszkód példa

A fuzzerünknek szükséges a tanítóhalmazból felvennie a SecurityAccess-hez szükséges alap algoritmust. Ahhoz, hogy a szinteken tudjunk váltani, szükséges tudnia a fuzzernek az alapvetést, hiszen enélkül nem tud majd tovább lépni. Fontos kiemelni viszont, hogy az UDS protokollon belül nincs minden service különböző biztonsági szintekhez kötve, ezért ez az algoritmus betanítása lehet opcionálisan választható.

Az egyes servicek és munkamenetek különböző elvárásokat támaszthatnak a tesztelő felé, melyeket szükséges betanítania a fuzzernek, de a kommunikáció megvalósításáért felelős Control Area Network azonosító és az UDS protokoll alapjai minden esetben szükségesek.

Az üzenet felépítése, például egy kérés az alábbi táblázatban látható:

CAN-ID	Adathossz számláló	Funkció	Alfunkció	Paraméterek			
0x755	0x06	0x85	0x01	0x02	0x02	0x02	0x02

4.1 táblázat - Egy minta üzenet

Fontos megemlíteni, hogy az UDS protokollon belül a bájtok számát is meg kell határozni, melyek elküldésre kerülnek, ez a táblázatban az adathossz számlálót jelenti.

Összefoglalva, az okos fuzzerünk szürke dobozos módszertant követ, mivel van tanult információnk – jelen esetben a protokollról – és ezáltal lesz a tesztelés végrehajtva. A tanítóhalmaz tartalmazza a CAN-ID-t (vagyis a kommunikáció helyszíne), az UDS protokoll felépítését és a SecurityAccess algoritmusának az alapjait, mely opcionális.

4.3. TesterPresent (0x3E) elméleti teszteset bemutatása

Ebben az alfejezetben implementálásra kerül egy okos fuzzer segítségével a TesterPresent funkció tesztelése az UDS protokollon belül.

A szolgáltatást azért fontos bemutatni, mivel fontos szerepet játszik az autódiagnosztikai rendszerek biztonságos használatában. Összefoglalva, a funkcióval a tesztelő megbizonyosodhat arról, hogy előzetes azonosítást követően tudott kapcsolódni a járműhöz, ezzel segít elkerülni a nem engedélyezett eszközök használatát és az esetleges támadásokat.

Az UDS protokollban a TesterPresent üzenet küldése fontos funkció, mert a diagnosztikai egységeknek lehetőséget ad arra, hogy azonosítsák a teszter jelenlétét, és észleljék az esetleges kommunikációs hibákat. A teszter jelenlétének meghatározása különösen akkor fontos, ha a diagnosztikai egység hosszabb ideig nem kap adatot a tesztertől, mert ebben az esetben a diagnosztikai egység automatikusan áttér a passzív állapotba, és várakozik a teszter jelzésére. A TesterPresent funkció a protokoll stabil és megbízható működéséhez járul hozzá.

Ezenkívül a TesterPresent szolgáltatás lehetővé teszi a teszter és a diagnosztikai egység közötti időzítési és kommunikációs paraméterek finomhangolását is, amelyek javítják a diagnosztikai műveletek hatékonyságát és biztonságát. Az eléréséhez nincs szükség SecurityAccess hitelesítésre, mivel egy olyan alapszolgáltatás, amelyet a diagnosztikai egységnek mindig el kell érnie.

Fontos ezenfelül megemlíteni azt is, hogy a TesterPresent-nek az összes munkamenetben elérhetőnek kell lennie. A teszteset implementálásához szükséges az összes munkamenetben futtatni a fuzzert, de ezt az információt képes lesz a tesztelő hozzáadni a tanítóhalmazhoz.

A TesterPresent SID-je 0x3E értéknek felel meg a szabvány szerint, az alfunkció értéke kizárólag 0x00 és 0x80 lehet. A TesterPresent-en belül az alfunkciókon felül nincsenek újabb paraméterek. Az előző alfejezetben tárgyalt kommunikáció létrehozásánál a TesterPresent pozitív válasza 0x7E, a negatív válasza pedig 0x7F 0x3E NRC. Fontos kiemelni, hogy a 0x80 alfunkció esetén a kérése esetén pozitív válasz nem várható, mivel az alfunkció azonosítójának a hetedik bitje vezérlőbitként funkcionál (Suppress Positive Response Message Indication Bit), 1-es érték esetén visszatartja a pozitív válasz üzenetek küldését, míg 0 esetén átengedi.

A teszteset implementálásnál tehát az előző alfejezetben összefoglalt tanítóhalmazon felül érdemes megtanítani a fuzzernek azt a funkciót, hogy amennyiben a hetedik bitünk bekapcsolt állapotban van, átugorjuk, mivel választ nem fogunk kapni a szervertől. A tesztesetben az összes alfunkciót (kivéve a 0x80-asat) tesztelni kell, 0x00-tól egészen 0xFF értékig. Tehát, ha egy feltétel rendszert határoznánk meg kódolási alapon a következőképpen jelenhetne meg a fuzzerben:

```
for $subfunction(0,0xFF)
response = send(3E $subfunction != 80)
```

A vizsgálat fő célja, azon esetek felderítése, ahol a rendszer nem válaszol - például összeomlás miatt -, vagy nem a szabványnak megfelelő válaszokkal szolgál. Az előbb kifejtésre került, hogy a TesterPresent esetében milyen válaszkódok lehetnének elfogadhatóak, kódban a következőképpen lehet felírni:

```
if response == NULL OR (response[1] != 7E AND response[1] != 7F)
f”Finding a bug: {response}
```

A feltétel rendszer alapján, ha a válasz értéke „NULL” vagyis nem érkezik válasz, vagy a válasz első bitje nem egyenlő 0x7E vagy 0x7F értékkel, akkor a fuzzerünk hibára talált, ami lehet a rendszer teljes összeomlása, vagy egy hibás válaszüzenet is. Utóbbi esetben érdemes tovább vizsgálódni, hogy vajon milyen választ is kaptunk, tartalmaz-e szenzitív információt.

A fuzzer egyik lehetséges metódusának a felépítése a következőképpen nézhet ki:

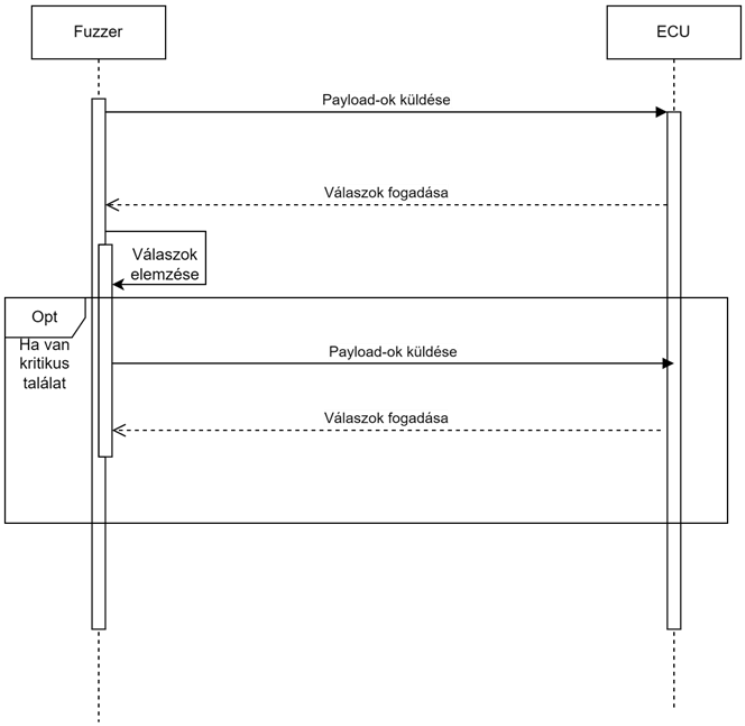
```
fuzzer_for_testerpresent()
    for $subfunction(0,0xFF)
    response = send(3E $subfunction != 80)
     if response == NULL OR (response[1] != 7E AND response[1] != 7F)
     f”Finding a bug: {response}
for $session_id(0,0xFF)
     send($10 $session_id)
     fuzzer_for_testerpresent()
```

A TesterPresent tesztelése azért is szükséges és fontos, mert ezáltal tudunk megbizonyosodni arról, hogy a rendszerünk még válaszképes, illetve a rendszerünk nem omlott össze.

A másik elképzelés a következő lépésekből állna össze:

1. A teszter készülék által generált payload-okat küld az autódiagnosztikai rendszernek a diagnosztikai porton keresztül.
2. Az autódiagnosztikai rendszer válaszol a payload-okra, és visszaküldi az azonosító adatokat és hibakódokat.
3. A teszter készülék elemzi a válaszokat, és ha talál kritikus hibát, akkor további payload-okat küld az észlelt hibák kihasználásával.
4. A folyamat ismétlődik, amíg a tesztelő készülék és az autódiagnosztikai rendszer közötti kapcsolat megszakad.

A folyamat megalkotásához segítségünkre lehet egy szekvencia diagram felvázolása is:



4.3 ábra - Válasz elemzése

A fent látható szekvencia diagramon az esetleges lépéseket tüntettem fel, amelyek az előbb kifejtésre kerültek, kivéve a negyedik pontot. Az ábrán az üzenetküldési folyamat megvalósítását szeretném bemutatni, a könnyebb átláthatóság és megértés érdekében.

4.4. ControlDTCSetting (0x85) elméleti teszteset bemutatása

A Control DTC Settings szolgáltatás az UDS protokoll része, amely lehetővé teszi az ECU (Electronic Control Unit) számára, hogy beállítsa a hibakódok kezelési beállításait, például törölheti vagy módosíthatja a hibakódokat, valamint beállíthatja a hibakódok jelentési prioritását.

A Control DTC Settings szolgáltatás hibás használata vagy kihasználása biztonsági kockázatokat rejthet magában, hiszen lehetőség van arra, hogy egy támadó hozzáférjen az ECU-hoz, és megváltoztassa a rendszer működését. Ha egy kiberbűnöző hozzáfér az ECU-hoz, akkor a Control DTC Settings szolgáltatás segítségével hibás információkat küldhet a diagnosztikai eszköznek, ami zavarokat okozhat a jármű működésében. Ezen kívül a támadó kihasználhatja a szolgáltatás hiányosságait és

manipulálhatja a DTC beállításokat, például kikapcsolhatja a hibakódokat, amelyek segítségével a diagnosztikai eszközök képesek lennének a hibákat észlelni és jelenteni.

A szolgáltatás a 0x85-ös hexadecimális értékkel definiálható a protokollon belül, az előző alfejezetekben részletesen tárgyalt módszerrel állítható, hogy a pozitív válasz esetén a 0xC5 értéket fogjuk látni, negatív válasz esetén pedig a 0x7F értéket.

Az okos fuzzer megalkotásához ebben az esetben is elengedhetetlen az UDS protokoll ismerete, a CAN hálózat azonosítója. Ezen felül ennél a tesztesetnél szükségünk lesz a Security Access eléréséhez szükséges algoritmus betáplálására is, hiszen a Control DTC Settings funkció különböző biztonsági szinteken érhető el. A tesztelés hatékonyságához érdemes az összes elérhető biztonsági szinten a fuzzert alkalmazni.

Az előző fejezetben tárgyalt TesterPresent-tel ellentétben, a Control DTC Settings rendelkezik további paraméterekkel, ezáltal a véletlenszerűen generált értékek is megnőnek ennél a tesztesetnél.

A cél nem változik ebben az esetben sem, fel nem fedett hibákra, hibás válaszüzenetekre és a rendszer összeomlására fókuszálnak a tesztelők.

A Control DTC Settings esetében is kell az összes alfunkciót tesztelni 0x00-tól egészen 0xFF értékig, tehát itt is felírható a következő kód:

```
for $subfunction(0,0xFF)
response = send(85 $subfunction)
```

Hasonlóan az előző tesztesethez, megvizsgáljuk, hogy a válaszunk, ne 0xC5 és ne 0x7F értéket vegyen fel, ugyanis, ha ezeket kapjuk válasznak, a rendszer a teszteset szempontjából megfelelően működik. Ha nem érkezik válasz, vagy teljesen véletlenszerű reakció érkezik, akkor problémát talált a fuzzer:

```
if response == NULL OR (response[1] != C5 AND response[1] != 7F)
f”Finding a bug: {response}
```

A Control DTC Settings teszteléséhez szükség van további paraméterek megadására is, ezért a fuzzerünknek további értékeket kell majd generálnia. A paraméterek intervalluma 0x00 hexadecimális értéktől egészen 0xFFFFFF-ig terjedhet, tehát decimális értékben a paraméterek összesített száma meghaladhatja a 16 millió féle

kombinációt. A példa kódban egy *for* ciklus segítségével egyelőre csak a lehetséges értékeket adom meg. Természetesen, ha kiegészítjük a paraméterekkel, a válaszrendszer megmarad, tehát itt is megadható az előbb felvázolt vizsgálat:

```
for $additional_params(0x00,0xFFFFFF)
response = send(85 $subfunction)
if response == NULL OR (response[1] != C5 AND response[1] != 7F)
f”Finding a bug: {response}
```

A hatékonyabb használat érdekében érdemes egy metódusba kiszervezni az elkészített funkciókat, mely a későbbiekben hozzájárul a hatékonyabb és egyszerűbb teszteléshez a különböző biztonsági szinteken.

```
fuzzer_for_control_dtc_settings()
     for $subfunction(0,0xFF)
          response = send(85 $subfunction)
          if response == NULL OR (response[1] != C5 AND response[1] !=
7F)
          f”Finding a bug: {response}
     for $additional_params(0x00,0xFFFFFF)
          response = send(85 $subfunction)
          if response == NULL OR (response[1] != C5 AND response[1] !=
7F)
          f”Finding a bug: {response}
```

A metódus használata a következő módon valósulhatna meg a különböző biztonsági szinteken:

```
for $session_id(0,0xFF)
send($10 $session_id)
fuzzer_for_control_dtc_settings()
SecurityAccess(Level1)
fuzzer_for_control_dtc_settings()
SecurityAccess(Level2)
.
.
.
.
SecurityAccess(LevelN)
```

A fuzzerünk jelen esetben hasonlíthat egy sérülékenységi szkenneléshez is, de fontos kiemelni, hogy a bemeneteket ettől függetlenül véletlenszerűen képezzük, hogy figyeljük a szoftver reakcióját.

4.5. Továbbfejlesztési lehetőségek

Természetesen az előbb feltüntetett példákon túl az UDS protokoll rengeteg más szolgáltatással is rendelkezik, melyeket érdemes tesztelni okos fuzzer segítségével. Továbbá a munkafolyamatok váltására és tesztelésére is lehetséges okos fuzzing módszertant kidolgozni a jövőben, hiszen kiberbiztonsági szempontból rendkívül fontos szolgáltalás.A továbbiakban ezeknek a felépítését hasonlóan végiggondolva fel lehetne vázolni p*szeudokód* és *control flow* grafikon segítségével is. A módszer segítséget nyújthat, a tesztesetek implementálása szekvenciadiagram bemutatásával is. A hatékony okos fuzzer megalkotásához szükséges az elméleti felvázolás, mielőtt a gyakorlati megvalósítást megkezdi a tesztelő.

A fejlesztés, implementálás következő fázisában érdemes egy, az előbbi fejezetekben tárgyalt fuzzert kiválasztani, és annak segítségével megvalósítani a lehetséges teszteseteket. Mindegyik fuzzernek megvan a maga sajátossága, a jövőben elképzelhető a kombinált használatuk is, a nagyobb hatékonyság elérésének érdekében. Saját keretrendszert is létre lehet hozni, de maga a folyamat hosszadalmas, rengeteg akadályt és kihívást rejt magában. A fuzzer megalkotása, felépítése komoly szakmai hátteret igényel, hiszen az UDS protokoll alapos ismerete mellett az eszköz korlátait és képességeit is behatóan ismerni kell. A legcélszerűbb egy olyan platform létrehozása, mely az összes operációs rendszert támogatja, hiszen a hatékony és gyorsabb tesztelés megvalósításához hozzájárul.

Az UDS protokollról elmondható, hogy szabványosított, ezáltal biztosak lehetünk abban, hogy adott kérésre adott válasz érkezik majd. Ha azonban NRC kódokra építenénk fel a fuzzerünket, a naplófájlokból tanulva, minden esetben meg kell tanítani az adott ECU-hoz tartozó gyártóspecifikus kódokat. Azon az elméleti útvonalon elindulva, hogy a fuzzerünk a folyamat során az NRC kódokból tanul, és ennek megfelelően képes reagálni, hogy érdemes-e tovább fuzzolni az adott részegységet vagy nem. Természetesen nem feltétlenül kell az NRC kódokból való gépi tanulást alkalmazni, elég, ha az előző fejezetekben tárgyalt tanítóhalmazt alkalmazzák a tesztelők.

A tesztelés megvalósításához, a fuzzer működéséhez szükség van megfelelő számú véletlenszerűen gyártott bemenetre. Az okos fuzzer működésénél ezt a funkciót ketté kell választani olyan esetekre, hogy az egyik tesztelési folyamat alatt csak olyan bemeneteket gyárt a tesztelő véletlenszerűen, amiről feltételezhető, hogy az ECU-nak is

megfelelő, elfogadja a formátumot, a másik tesztelésnél viszont mindent, formátumfüggetlenül. Utóbbi teszteset azért fontos, mert elképzelhető, hogy nem várt működés lép fel, például lehet egy szimbólum sorozat által elérhető, hogy az ECU összeomoljon és újra kelljen indítani.

A fuzzolás során pedig képes lenne egy grafikont is létrehozni, hogy milyen hívásokat hajtott végre az összeállítás időpontjában. A grafikon létrehozásához lehetne használni már egy meglévő, autóiparban is használatos fuzzing eszközt, az EffCan-t. Az előbb említett fuzzer segítségével a teljes bejárt útvonal felvázolható, ezáltal sokkal átláthatóbb és követhetőbb a tesztelési folyamat. [3]

A grafikon mellett a fuzzer képes lehetne részletes hibajelentések készítésére is, mivel ez is nagyban növelné a hatékonyságot és az átláthatóságot is. Az eredményekből lehetne egy adatbázist készíteni, melyet szintén fel lehetne használni a későbbi tesztelések folyamán.

4.6. Következtetések

Az UDS protokoll számos további szolgáltatással is rendelkezik, melyet a jövőben érdemes adaptív fuzzing tesztelési technikával ellenőrizni. A korábbi tesztelési eljárásokkal szemben, az okos fuzzing megoldással időt takarít meg a tesztelői csapat, hiszen a protokoll ismeretében csak az adott intervallumban fog a fuzzing megvalósulni. A TesterPresent szolgáltatásnál, ha alfunkciókat tesztelünk az okos fuzzer - mivel protokoll ismerettel rendelkezik – segítségével, akkor véges számú esetben fog lefutni. Egy nem intelligens fuzzer tekintetében ez a szám (ami a szükséges idő és a tesztesetek) a többszöröse is lehet, hiszen nem tudunk ilyen szintű tanítóhalmazt átadni a tesztelői eszközünknek. Az okos fuzzing ezáltal pontosabb és célzottabb tesztelési módszert nyújt, ezzel lehetővé teszi a biztonsági rések és hibák gyorsabb felderítését, így javítva az autók biztonságát és megbízhatóságát.

Az előbb említett időtakarékosság és intervallum fedés miatt, nagyobb hatékonyságot lehet elérni az okos megoldással, mivel az adatmezők típusát és struktúráját jobban képes felismerni. Számszerűsítve az adaptív és "buta" változat különbségei, konkrét értékei az adott tesztesetek alapján változhatnak, és azok függhetnek a használt adatok mennyiségétől és minőségétől is. Azonban általánosságban elmondható, hogy az okos megoldás egyrészt nagyobb valószínűséggel talál biztonsági rést, másrészt hatékonyabb tesztlefedettséget biztosít.

Az okos fuzzer a protokoll ismereten felül rendelkezhet a bemeneti adatstruktúra felépítésével, ezáltal is időt spórolhat a tesztelői csapat, hiszen így csak érvényes bemenettel fog történni a tesztelés. Természetesen tesztesettől függően történhet ez. Elképzelhető, hogy olyan tesztelésre is szükség van, amikor érvénytelen, teljes mértékben véletlenszerű értékek kerülnek legenerálásra.

Az okos változat, mivel hatékonyabb tesztelést kínál, csökkentheti a fejlesztési költségeket, hiszen a rendellenességek és hibák időben történő észrevétele lehetővé teszi a korai javítást, ezáltal költségspórolás érhető el.

5. Összegzés

Szakdolgozatomban betekintést nyújtottam a járműipari kommunikációval kapcsolatos technológiákba és fenyegetettségekbe. Megismerkedtem különböző tesztelési módszertanokkal és technikákkal, mely a későbbi munkám során mindenképpen hasznosnak fog bizonyulni.

A feladat elvégzéséhez kutatómunkát végeztem mind hazai és nemzetközi szabványokat és szakirodalmat felkutatva, melyek segítségével hatékonyan tudom értelmezni már ezen dokumentumokat. Emellett a Budapesti Műszaki és Gazdaságtudományi Egyetem üzemmérnök informatikus szakán elsajátított tárgyakat is hatékonyan tudtam alkalmazni, különös tekintettel a Kódolás és IT biztonság, Hálózatok alapjai és üzemeltetése, Alkalmazott mesterséges intelligencia, Algoritmusok és gráfok illetve a Gyakorlati hálózatbiztonság tárgyakra.

A dolgozat központjában az adaptív fuzzing technika került áttekintésre, különböző iparági területekre kitekintve, különböző nyílt- és zártforráskódú keretrendszerekkel bemutatva. A feladat során elmélyültem az előbb említett technika rejtelmeiben, egy esetleges elvi példán keresztül bemutattam egy autóipari okos fuzzer lehetséges felépítését. A jövőben a további funkciók elméleti kidolgozásán túl, szeretném, ha a rendszer megvalósításra kerülne, mivel az elméleti tesztesetek adaptálása mellett a gyakorlati megvalósítás elmaradt.

6. Irodalomjegyzék

[1] X. Z. Z. Y. Y. J. J. W. M. W. W. F. Feng Lou, „Cybersecurity Testing for Automotive Domain: A Survey," *Sensors,* %1. kötet22(23), %1. szám9211, pp. 1-27, 2022.

[2] R. H. R. O. James Graham, Cyber Security Essentials, Boca Raton: Auerbach Publications, 2011, pp. 5-10.

[3] M. Z. M. S. Lama J.Moukhal, „Vulnerability-Oriented Fuzz Testing for Connected Autonomous Vehicle Systems," *IEEE TRANSACTIONS ON RELIABILITY,* %1. kötet70, %1. szám4, pp. 1-16, 2021.

[4] A. Drozhzhin, „Kapersky daily," 6 Augusztus 2015. [Online]. Available: https://www.kaspersky.com/blog/blackhat-jeep-cherokee-hack-explained/9493/. [Hozzáférés dátuma: 06 04 2023].

[5] O. Solon, „The Guardian," 10 Szeptember 2016. [Online]. Available: https://www.theguardian.com/technology/2016/sep/20/tesla-model-s-chinese-hack-remote-control-brakes. [Hozzáférés dátuma: 6 Április 2023].

[6] R. S. Shiho Kim, „Chapter 4 - In-Vehicle Communication and Cyber Security," in *Automotive Cyber Security,* Singapore, Springer, 2021.

[7] Z. S. F. A. Victor Ocheri, „A Survey of Automotive Networking Applications and Protocols," in *Connected Vehicle Systems,* Boca Raton, 2017.

[8] BOSCH, „CAN Specification - Version 2.0," Robert Bosch GmbH, Stuttgart, 1991.

[9] „Road vehicles - Unified diagnostic services, ISO-14229-1," ISO copyright office, Geneva, 2013.

[10] F. K. Mohd.Ehmer Khan, „A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *International Journal of Advanced Computer Science Applications,* %1. kötet3, %1. szám6, pp. 12-15, 2012.

[11] M. B. David Liu. [Online]. Available: https://www.cs.toronto.edu/~david/course-notes/csc110-111/15-graphs/07-control-flow-graphs.html. [Hozzáférés dátuma: 23 02 2023].

[12] H. Mössenböck. [Online]. Available: https://www.researchgate.net/figure/Control-flow-graphs-of-an-IF-and-a-WHILE-statement-with-instructions-in-SSA-form_fig1_220404723. [Hozzáférés dátuma: 23 02 2023].

[13] K. T. W. Myint Myiztu Aung, „Improving Branch Coverage for White-box Testing," in *ICCTA,* Alexandria, 2017.

[14] H. Wu, „An effective equivalence partitioning method to design the test case of the WEB application," in *International Conference on Systems and Informatics,* Yantai, 2012.

[15] „javatpoint," [Online]. Available: https://www.javatpoint.com/regression-testing. [Hozzáférés dátuma: 23 02 2023].

[16] K. T. A. V. K. J. K. Todor Tagarev, „Automotive Cybersecurity Testing: Survey of Testbeds and Methods," in *Digital Transformation, Cyber Security and Resilience of Modern Societies* , Cham, Springer, 2021.

[17] R. G. M. B. G. F. C. H. Andreas Zeller, „The Fuzzing Book," [Online]. Available: https://www.fuzzingbook.org/.

[18] J. D. C. M. Ari Takanen, „Chapter 2 - Software Vulnerability Analysis," in *Fuzzing for Software Security Testing and Quality Assurance*, Norwood, ARTECH HOUSE, 2008.

[19] J. D. C. M. Ari Takanen, „Chapter 1 - Introduction," in *Fuzzing for Software Security and Quality Assurance*, Norwood, ARTECH HOUSE, INC, 2008.

[20] M. I. U. H. B. Lucas McDonald, „ResearchGate," December 2021. [Online]. Available: https://www.researchgate.net/publication/356980212_Survey_of_Software_Fuzzing_Techniques. [Hozzáférés dátuma: 23 Február 2023].

[21] J. D. C. M. Ari Takanen, „Chapter 5 - Builiding and Classifying Fuzzers," in *Fuzzing for Software Security and Quality Assurance*, Norwood, ARTECH HOUSE, INC, 2008.

[22] „Testfully - A guide to fuzz testing," Testfully Pty Ltd., 29 Július 2021. [Online]. Available: https://testfully.io/blog/fuzz-testing/. [Hozzáférés dátuma: 17 04 2023].

[23] M. B. E. S. R. C. R. Van-Thuan Pham, „Smart Greybox Fuzzing," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, %1. kötetXX, %1. számX, pp. 1-17, 2019.

[24] C. B. R. G. L. M. Sofia Bekrar, „Finding Software Vulnerabilities by Smart Fuzzing," in *IEEE International Conference on Software Testing, Verification and Validation*, Berlin, 2011.

[25] D. M. H. E. M. H. Andrea Fioraldi, „AFL++: Combining Incremental Steps of Fuzzing Research," in *USENIX Workshop on Offensive Technologies*, 2020.

[26] „Peach Fuzzer," [Online]. Available: https://peachtech.gitlab.io/peach-fuzzer-community/. [Hozzáférés dátuma: 07 03 2023].

[27] „Peach Fuzzer - Publishers," [Online]. Available: https://peachtech.gitlab.io/peach-fuzzer-community/v3/Publisher.html. [Hozzáférés dátuma: 07 03 2023].

[28] „GitHub - Sulley," [Online]. Available: https://github.com/OpenRCE/sulley. [Hozzáférés dátuma: 07 03 2023].

[29] „GitHub - CaringCaribou," [Online]. Available: https://github.com/timower/caringcaribou. [Hozzáférés dátuma: 07 03 2023].

[30] „Vinnova," 24 02 2020. [Online]. Available: https://www.vinnova.se/en/p/heavens-healing-vulnerabilities-to-enhance-software-security-and-safety/. [Hozzáférés dátuma: 07 03 2023].

[31] „GitHub - Autofuzz," [Online]. Available: https://github.com/timower/caringcaribou/blob/autoFuzz/documentation/autofuzz.md. [Hozzáférés dátuma: 07 03 2023].

[32] D. X. Du Qingfent, „An improved algorithm for basis path testing," in *International Conference on Business Magagement and Electronic Information*, Guanzhou, 2011.

[33] Q. C. P. Z. Shunhui Ji, „Neural Network Based Test Case Generation for Data-Flow Oriented Testing," in *International Conference On Artificial Intelligence Testing*, Newark, 2019.

[34] V. Nanda, „tutorialspoint," [Online]. Available: https://www.tutorialspoint.com/what-is-loop-testing-methodology-with-examples. [Hozzáférés dátuma: 23 02 2023].

7. Táblázatok és ábrák jegyzéke

Függelék

Rövidítések

AAA: Authentication, Accounting, Authorization

AFL: American Fuzzy Loop

CAN: Control Area Network

CAN-FD: CAN Flexible Data

CIA: Confidentiality, Integrity, Availability

DoS: Denial of Service

ECU: Electrical Control Unit

HEAVENS: Healing Vulnerabilities to Enhance Software Security and Safety

HTTP: Hypertext Transfer Protocol

IEEE: Institute of Electrical and Electronics Engineers

IoT: Internet of Things

ISO: International Organization for Standardization

IT: Information Technology

LIN: Local Interconnect Network

MOST: Media Oriented Systems Transport

NIAG: NATO Industrial Advisory Group

NIST: National Institute of Standards and Technology

NRC: Negative Response Code

OSI: Open Systems Interconnection

PTES: Penetration Testing Execution Standard

SAE: Society for Automotive Engineers

SID: Service Identifier

SMTP: Simple Mail Transfer Protocol

TARA: Threat and Risk Analysis

TCP: Transmission Control Protocol

UDS: Unified Diagnostic Service

UML: Unified Modeling Language

XML: Extensible Markup Language