Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási nyelvek és Fordítóprogramok Tanszék

Nem biztonságos kód biztonságossá transzformálása a RefactorErl segítségével

Témavezető:
Dr. Tóth Melinda, Dr. Bozó István
egyetemi docens, egyetemi adjunktus

Szerző:
Juhász Alexandra Nelli
programtervező informatikus MSc

Budapest, 2025

Absztrakt

Az Erlang egy funkcionális, magas szintű programozási nyelv, amely támogatja az elosztott, skálázható és hibatűrő programok egyszerű fejlesztését. Közkedvelt nyelv a világon, ugyanis gyakran használják kritikus rendszerek fejlesztésére, mint például banki, telekommunikációs és közlekedési szoftverekhez. Tehát elengedhetetlen, hogy a forráskódokban ne maradjanak támadható sérülékenységek. Statikus elemzési módszerekkel az ilyen pontok egyszerűen felderíthetők. A RefactorErl egy nyílt forráskódú statikus elemző eszköz, amely támogatja az Erlang fejlesztők mindennapi munkáját különféle kódelemzési és keresési funkciókkal, valamint lehetőséget biztosít refaktorálások és transzformációk megvalósítására is. A diplomamunkámban megvizsgálom az Erlang projektekben leggyakrabban előforduló biztonsági problémákat, és ezek alapján átalakítási mintákat definiálok. A célom, hogy olyan automatikus transzformációs módszereket implementáljak a RefactorErl statikus elemző eszközben, amelyek képesek az Erlang nyelvben előforduló sérülékeny kódrészleteket biztonságossá alakítani.

Tartalomjegyzék

1. fejezet

Bevezetés

Az informatika és annak folyamatos fejlődése rengeteg iparágra nagy hatással van. A közlekedésben, például a forgalomirányítási rendszerek vagy az önvezető autók esetében, elengedhetetlen a szoftverek biztonsága és védelme a balesetek elkerülése érdekében. Az egészségügyben létfontosságú a betegek adatainak tárolása és azok kezelése, hiszen súlyos következményekkel járhat, ha a rendszer egy támadás áldozata lesz. A pénzügyi szektorban például a digitális banki szolgáltatások és az online fizetési rendszerek biztonságától függ a tranzakciók megbízhatósága. A telekommunikációs iparágban a hálózatok védelme nélkülözhetetlen a szolgáltatások zökkenőmentes működéséhez. Jól látható, hogy egyre több folyamat kerül szoftverek irányítása alá, ezzel jelentősen megkönnyítve az emberek mindennapi életét.

Ahogy nő az informatika és a digitalizáció szerepe a világon, úgy nő a kibertámadások száma is, amelyek egyre komplexebbek lesznek az idő előrehaladtával. Fontos tisztázni, hogy mit is kell érteni kibertámadás alatt: olyan szándékos tevékenység, amelynek célja a számítógépes rendszerek vagy hálózatok működésének a megzavarása, hozzáférés bizalmas adatokhoz, azok módosítása, ellopása vagy végleges törlése. Ilyen tevékenységek mögött állhatnak egyéni hackerek vagy szervezett bűnözői csoportok. Emiatt a forráskódok biztonsága is egyre nagyobb hangsúlyt kap napjainkban, hiszen a támadások mértékétől függően komoly társadalmi és gazdasági károkat tudnak okozni. Már egyetlen sérülékenység is elegendő lehet ahhoz, hogy a támadók hozzáférhessenek érzékeny információkhoz, befolyásolhassák a vállalatok működését vagy akár megbéníthassák az egész rendszert. Tehát a fejlesztés során a működőképesség mellett nem elég csak a teljesítményre összpontosítani, a programkód biztonságával is foglalkozni kell.

A biztonságos szoftverfejlesztésre már létrejöttek szabályok és irányelvek, amelyek meghatározzák, hogy mely szempontokat érdemes figyelembe venni. A legelterjedtebb nemzetközi szabvány az ISO / IEC 27001 [1], melynek fő célja, hogy egy minimumkövetelményt állítson fel a megbízható működéshez. Egy keretrendszert biztosít a szervezeteknek az érzékeny adatok kezeléséhez, továbbá segít azonosítani és megoldani a különféle felmerülő kockázatokat. Említésre méltó még az OWASP (Open Web Application Security Project) [2] alapítvány kezdeményezése, amely útmutatókat és különféle irányelveket nyújt a biztonságos alkalmazások fejlesztéséhez.

Az Erlang [3, 4] egy deklaratív funkcionális programozási nyelv, amelyet eredetileg kifejezetten telekommunikációs rendszerek számára fejlesztettek ki. Elsődleges célja, hogy egyszerűen és hatékonyan lehessen vele párhuzamosan futó és elosztott rendszereket létrehozni.

A szoftverbiztonság tehát meghatározó tényező az élet minden részén, ahogy azt az előző bekezdésekben is alátámasztottam. Különösen fontos olyan funkcionális nyelvekben is, mint az Erlang, hogy időben felismerhetők és kezelhetők legyenek a potenciálisan támadható rések.

Éppen ezért a diplomamunkámban az Erlang nyelv sérülékenységeivel és azok biztonságossá transzformálásával foglalkozom.

1.1. A diplomamunkám és annak jelentősége

Szoftverfejlesztés során a programozók nem tudnak minden lehetséges sérülékenységet előre látni és lekezelni, így a később felmerülő hibákat utólag javítják ki manuálisan. A legtöbb esetben a kézi javítás rendkívül időigényes, és sokkal nagyobb a hibázási lehetőség. Tehát elengedhetetlen ezeknek a folyamatoknak az automatizálása. A munkámban bemutatott transzformációs módszerek képesek azonos módon kezelni az ugyanolyan sebezhető kódrészleteket. Ezáltal egy megbízható megoldás kapható a végén, amely minimalizálja a nem biztonságos részeket, így csökkenti a támadások esélyét.

Az Erlang programozási nyelvhez kifejlesztett RefactorErl [5] eszköz képes azonosítani a biztonsági kockázatot jelentő kódrészeket, azonban még csak a felismerésen van a hangsúly. A célom az, hogy kiegészítsem ezt a folyamatot olyan automatikus transzformációkkal, amelyek a sérülékeny kódrészeket biztonságosabbá alakítják. A

transzformációk alatt olyan kódon belüli átalakításokat értek, amelyek módosítják a kód szerkezetét és futásidejű viselkedését azért, hogy a program biztonságosabb legyen. Ilyen például egy új függvény vagy elágazás beszúrása.

1.2. A diplomamunka szerkezete

A munkám 2. fejezetében áttekintést adok arról, hogy miért is lényeges a forráskódok biztonsága, illetve az elhanyagolása milyen következményekkel járhat. Továbbá ismertetek néhány statikus elemző eszközt, és részletesen bemutatom az Erlang programozási nyelvet. A 3. fejezetben kifejtem az Erlang leggyakoribb sérülékenységeit, majd ismertetem a RefactorErl-t. Beszélek az eszköz használatáról és a potenciális sérülékenységeket ellenőrző függvényeiről. A 4. fejezetben ismertetem a problémát, elmagyarázom a lehetséges transzformációs módszereket, majd bemutatom az általam megvalósított átalakításokat. A végén még leírok néhány implementálható transzformációt, amelyek megvalósítására nem jutott idő. Az 5. fejezetben részletesen leírom a transzformációk futási eredményeit, amelyeket egy-egy példával is szemléltetek. A 6. fejezetben a dolgozatomhoz kapcsolódó további munkákat ismertetem. Végezetül a 7. fejezetben olvasható a diplomamunkám összegzése és a továbbfejlesztési lehetőségei.

2. fejezet

Irodalmi áttekintés

A következő fejezetben összefoglalom diplomamunkámhoz kapcsolódó legfontosabb háttérismereteket és a gyakran előforduló sérülékenységeket. Továbbá kitérek a statikus elemzésre és azon alapuló eszközökre, valamint részletesen bemutatom az Erlang programozási nyelvet.

2.1. Forráskód biztonsága

Egyáltalán nem könnyű feladat olyan programot írni, amely ellenálló a különböző kibertámadásokkal szemben. Nemcsak a forráskód olvashatóságára és érthetőségére kell figyelni, hanem annak minőségére is. Ha az utóbbin kevesebb hangsúly van, akkor nagymértékben megemelheti a rendszer fenntartási költségeit, illetve növelheti a fejlesztés során előforduló hibák számát. Ahhoz, hogy minél jobb minőségű legyen a kód, amely hosszú távon fenntartható is legyen, a következő szempontokat érdemes alapul venni a programmal kapcsolatban [6, 7]:

- megbízhatóság
- hibatűrés
- elérhetőség
- lehetséges sérülékenységek korai azonosítása

A biztonságos programozás lassan már alapkövetelménynek számít a szoftverfejlesztésben, ezért érdemes tisztázni, hogy mit is jelent. A fejlesztési folyamat azon részére utal, amely a forráskódban található sérülékenységek megelőzésére, felismerésére és kezelésére irányul. A legfőbb célja, hogy a kódban ne maradjanak olyan hibák, amelyek támadhatók.

Különféle jelentések és statisztikai adatok is hangsúlyozzák a forráskód biztonságának fontosságát. A Verzion 2025-ös DBIR (Data Breach Investigations Report) [8] szerint az elkövetett incidensek jelentős része kódbiztonsági problémákhoz köthető. A jelentés 2023. november 1. és 2024. október 31. közötti intervallumot elemezte, ahol összesen 22 052 vizsgált biztonsági esemény közül 12 195 mutatkozott valódi adatvédelmi esetnek. A kódbeli sérülékenységek kihasználása 20%-ra emelkedett, amely az előző évi jelentéshez képest 34%-os növekedést jelent. A ransomware[1] incidensek aránya is jelentősen megemelkedett, a 2025-ös jelentésben az összes adatvédelmi eset 44%-ában voltak jelen, szemben a tavalyi 32%-kal. Külön érdemes kiemelni, hogy az emberi tényezők továbbra is nagy mértékben befolyásolják a sebezhető részek felbukkanásának gyakoriságát a forráskódban, ugyanis becslések szerint az események 60%-ában játszottak szerepet [9]. Lehet szó akár rossz hitelesítésről és jelszókezelésről, nem megfelelő bemeneti adatok ellenőrzéséről vagy rossz rendszerkonfigurációról.

Több szabvány és irányelv is létrejött annak érdekében, hogy a vállalatok biztonságossá tegyék a szoftvereiket, és minimalizálják a forráskód sérülékenységéből adódó támadásokat. Ide köthető az ISO / IEC 27001 [1], amely a legelterjedtebb nemzetközi információbiztonsági szabvány. A tanúsítvány megszerzése rengeteg iparágban, például a pénzügyben, az egészségügyben vagy a telekommunikációban versenyelőnyt jelent, de van, ahol kötelező jelleggel elvárják, hogy meglegyen. Olyan folyamatokat és követelményeket határoz meg, amelyek alapján a szervezetek kialakíthatják, működtethetik és fejleszthetik az ISMS-üket (Information Security Management System). A szabvány előírja a biztonsági kockázatok azonosítását, azok kezelésére a megfelelő protokollok bevezetését, továbbá az érzékeny adatok megfelelő védelmét. Nemcsak a kód biztonságára fókuszál, hanem a teljes fejlesztési folyamatra is. Fontos, hogy ott is érvényesüljenek a biztonsági szempontok, mint például a megfelelő jogosultságkezelés, a verziókövetés és a fejlesztési környezetek megfelelő védelme. Egyik fő alapelve, hogy a biztonsági szabályozásokat a kockázatelemzés alapján kell meghatározni, és nem egy előre megadott sablon szerint. Így bármilyen méretű vállalatra könnyedén alkalmazható a szabvány.

Míg az ISO / IEC 27001 főleg szervezeti szinten szabályozza a biztonságot, addig az OWASP (Open Web Application Security Project) [2] fejlesztői irányelvekre és a kódot érintő különböző sérülékenységekre összpontosít. Az OWASP egy olyan nem-

[1]Olyan rosszindulatú program, amely titkosítja az áldozat adatait vagy rendszereit, és a támadó váltságdíjat követel a feloldásért vagy az ellopott adatok visszatartásáért.

zetközi nonprofit szervezet, amelynek legfőbb célja, hogy javítsa a szoftverek biztonságát különböző útmutatókkal és eszközökkel. Legismertebb kiadványa az OWASP Top Ten, amely a webalkalmazásokat érintő tíz legfontosabb biztonsági kockázatot sorolja fel, és konkrét gyakorlati tanácsokat ad azok megelőzésére és kezelésére. Egy másik ismertebb irányelve az OWASP ASVS (Application Security Verification Standard) [10], amely részletes követelményrendszert nyújt a biztonságos alkalmazások fejlesztéséhez és ellenőrzéséhez. Szintekre bontva (1-től 3-ig) fogalmazza meg a különböző biztonsági követelményeket, így a vállalatok a saját igényeikhez mérten választhatják ki az ellenőrzések mélységét.

2.2. Gyakori sérülékenységek a forráskódban

Az OWASP ASVS [10, 6, 11] szerint a legismertebb sérülékenységek egy forráskódban a következők:

- Nem megfelelő input ellenőrzés
- Hitelesítés és jelszókezelés
- Fájl vagy memóriakezelés
- Kimenet kódolása: Nem megfelelő kimenet kódolásakor a rendszer XSS (Cross-Site Scripting) támadásnak lehet kitéve, amely során a támadó rosszindulatú kódot futtathat a felhasználó böngészőjében.
- Munkamenet és jogosultság kezelés: Hibás munkamenet vagy jogosultságkezelés esetén illetéktelenek férhetnek hozzá védett erőforrásokhoz, ahol kártékony műveleteket is végrehajthatnak.
- Adatvédelem: Ha az érzékeny adatok nincsenek megfelelően titkosítva vagy biztonságosan kezelve, akkor azok könnyedén nyilvánosságra kerülhetnek, amelyek súlyos problémákat okozhatnak.
- Adatbázis biztonság: Nem megfelelő adatbázis kezelés SQL beszúrásos (SQL injection) támadáshoz vagy adatvesztéshez vezethet.
- Rendszerkonfiguráció: Alapértelmezett jelszavak, rosszul beállított jogosultságok vagy túl engedékeny rendszerbeállítások esetén biztonsági rések nyílhatnak a rendszerben, amelyen keresztül a támadó könnyen bejuthat és kárt okozhat.

A következőkben részletesebben bemutatok néhány sérülékenységet és azok következményeit, amelyek jól szemléltetik, hogy mennyire elengedhetetlen a kód meg-

felelő biztonsága.

2.2.1. Nem megfelelő input ellenőrzés

A bemeneti adatok hibás vagy hiányos ellenőrzése lehetővé teszi, hogy a támadó ártalmas kódot vagy parancsot juttasson a rendszerbe. Ez különösen sérülékennyé teszi az alkalmazást a beszúrásos (injection) támadásokkal szemben.

Az egyik legelterjedtebb formája az SQL beszúrásos (injection) támadás, amikor a támadó a bemeneti adatok között kártékony SQL parancsokat helyez el, amelyeket az alkalmazás végrehajt az adatbázisban. Ez nem kívánt műveleteket eredményezhet, például adatok módosítását, törlését, ellopását vagy akár az autentikáció megkerülését [7]. Funkcionális nyelvek esetén kevésbé gyakori támadási forma, ellenben imperatív nyelvek esetén nagyon is jellemző.

A hibás inputkezelés akár XSS (Cross-Site Scripting) támadáshoz is vezethet. Ilyenkor a támadó rosszindulatú kódrészletet juttat be egy űrlapon vagy egy beviteli mezőn keresztül, amelyet az alkalmazás megkötés nélkül megjelenít a weboldalon. A kód ekkor lefut a felhasználók böngészőjében, amely lehetővé teszi a támadónak az adatlopást vagy jogosulatlan műveletek végrehajtását.

Ezen felül a nem megfelelően kezelt bemenetek memóriahibákat is okozhatnak, amelyek a túlterheléses (DoS / DDoS)[2] támadások alapját képzik.

2.2.2. Hitelesítés és jelszókezelés

A hitelesítési folyamatok hibás vagy nem biztonságos megvalósítása, továbbá a jelszavak helytelen kezelése lehetővé teszi, hogy illetéktelen személyek férjenek hozzá védett adatokhoz vagy rendszerekhez. Tipikus hibák például:

- Gyenge jelszókövetelmény (rövid, könnyen kitalálható jelszavak),
- Általános jelszavak használatának engedélyezése (például „admin”),
- Jelszavak nem biztonságos tárolása vagy titkosítása,
- Többfaktoros hitelesítés hiánya,
- A session id-k helytelen kezelése: sikeres bejelentkezés után nem változik meg, vagy kijelentkezés után nincs megfelelően érvénytelenítve [7].

Itt a próbálgatásos (brute force) típusú támadások gyakran előfordulnak, különösen akkor, ha a rendszer nem korlátozza a bejelentkezési kísérletek számát. Ilyenkor

[2]Denial of Service / Distributed Denial of Service

a támadó manuálisan, vagy jellemzően automatizált eszközökkel sorra kipróbálja a lehetséges kulcs és jelszó párosokat addig, amíg be nem tud jelentkezni. Ezt követően a támadó hozzáférhet érzékeny adatokhoz, manipulálhatja az információkat és a szolgáltatásokat, megzavarhatja az üzleti folyamatokat, valamint további rendszerek felé is terjeszkedhet.

2.2.3. Fájl vagy memóriakezelés

Fájlok nem megfelelő kezeléséről például akkor van szó, amikor nincsenek biztonságosan létrehozva az ideiglenes állományok, nincsenek megfelelően kezelve az elérési útvonalak, vagy fájlfeltöltéskor nincsenek ellenőrizve a bemenetek.

A felsorolt hibák különféle következményekkel járhatnak. Az első kettő esetben a támadó könnyedén hozzáférhet különböző védett állományokhoz, például konfigurációs fájlokhoz, amelyekből információt szivárogtathat ki. Az ellenőrizetlen fájlfeltöltés injection támadáshoz vezethet, például amikor egy rosszindulatú script fájl felkerül a szerverre, és RCE-t (Remote Code Execution) eredményez. Ez azt jelenti, hogy a támadó képes lesz távolról is kódot futtatni.

Memóriakezelési hibák akkor jelentkeznek, ha a program helytelenül kezeli az erőforrásokat. Ilyen eset lehet például a hibás indexelés, a túlzott memóriafoglalás vagy az erőforrások kimerítése. Ezek a problémák elsősorban DoS/DDoS típusú támadásokhoz vezethetnek, de súlyosabb esetben akár RCE-t is lehetővé tehetnek.

2.3. Statikus elemzés szerepe a kódbiztonságban

A vállalatok nagy erőkkel törekednek arra, hogy az alkalmazásaik biztonságosak legyenek, és ne maradjanak bennük támadható vagy sérülékeny kódrészek, amelyek veszélybe sodorhatják a céget. Szoftverfejlesztés közben az ember könnyen követhet el olyan hibákat, amelyek csak futási időben jönnek elő. Ezek az esetek lassítják a fejlesztési folyamatot, és megbízhatatlanná teszik a rendszert.

Az ilyen jellegű problémák kezelésére fejlesztették ki az automatizált elemző szoftvereket, amelyek statikus kódelemzés segítségével képesek költséghatékonyabbá tenni a fejlesztési folyamatokat. A forráskódot még futtatás előtt átvizsgálja a program, vagy még mielőtt egy ember letesztelné a szoftver működését. Így már a fejlesztés elején ki lehet javítani a hibákat vagy a lehetséges sérülékenységeket. Tehát ezzel

a kód minősége, karbantarthatósága és biztonsága is jelentősen növekszik. Minél hamarabb találják meg a hibát, annál olcsóbb és egyszerűbb lesz a javítása [12]. Ezek közül ma már számos statikus elemző program elérhető a fejlesztői környezetekbe integrálva, amelyek azonnali visszajelzést adnak például a kódolás közben felmerülő hibákról, a kódduplikációkról, potenciális sérülékenységekről vagy az optimalizálási lehetőségekről [13].

A statikus elemző szoftverek nem tudják 100%-ban lefedni az összes hibaforrást és garantálni a kód teljes biztonságát. Emiatt célszerű dinamikus teszteléssel együtt alkalmazni, hogy még kevesebb eséllyel maradjon sérülékeny rész a rendszerben. Ilyenek lehetnek például az adatfüggő logikai problémák vagy a helytelen rendszerkonfigurációból származó hibák.

A felsoroltak ellenére a fejlesztők mégsem használják rendszeresen a statikus elemző eszközöket [14]. Gyakran olyan ártalmatlan kódrészeknél is látható figyelmeztetés, amelyek nem jelentenek valódi problémát vagy veszélyt. Így a programozók sok esetben elvesznek a számtalan hamis riasztás között, demotiváltak lesznek, és csökken az eszköz használatának gyakorisága.

2.3.1. Statikus kódelemző szoftverek

Mára már rengeteg különféle programozási nyelvekre szabott statikus elemző program létezik. Talán az egyik legismertebb ezek közül a SonarQube [15], amely egy nyílt forráskódú (open source) elemző eszköz, és jó néhány programozási nyelvet képes támogatni: például Java, Python vagy C#. Az elemzés során külön kategóriákba sorolja a problémákat, hogy melyek a hibák, a sérülékenységek és a kódminőségi figyelmeztetések. Különböző CI/CD (Continuous Integration/Continuous Delivery) folyamatokba is egyszerűen integrálható. A SonarQube a cégek körében népszerű eszköz, azonban a hibadetektáló képessége rendkívül alacsony [16]. Számos valódi hibát nem képes felismerni, valamint az eszköz által "bug"-nak jelölt kódrészek többsége valójában nem vezet tényleges hibához.

Szintén számos nyelvet támogat a CheckMarx [17] rendszer, amely kifejezetten biztonsági szempontú elemzésekre épül. A vizsgálat során talált hibákat az OWASP Top Ten és a CWE/SANS [18] kategóriák szerint sorolja be. A program részletes leírást ad a sérülékeny kódrészről, majd különféle megoldási javaslatokat kínál annak biztonságosabbá tételére.

A Clang Static Analyzer [19, 20] kifejezetten a C és C++ nyelvekhez készült nyílt forráskódú szoftver. Elemzés során a rendszer összes lehetséges futási útvonalát végigvizsgálja, hogy figyelmeztessen a potenciális veszélyforrásokra, mint például memóriaszivárgásra, null pointer hivatkozásra, inicializálatlan változók használatára, hibás struktúrára vagy logikai hibákra. Ehhez egy úgynevezett 'symbolic execution' technikát használ, amellyel szimbolikusan követi végig a változók értékeit és azok állapotát a programon belül.

A CodeChecker [21, 22] a Clang Static Analyzer és a Clang Tidy eszközökre épült, könnyen integrálható CI/CD folyamatokba, és szintén a C és C++ nyelveket támogatja. Fordítás nélkül képes elemezni az utolsó futás óta módosult fájlokat. A vizsgálat során a Clang belső absztrakt szintaxisfáját (AST) használja, vagy szöveg alapján keres, vagy szimbolikus végrehajtáson alapuló ellenőrzéseket végez [20].

A SpotBugs [23] egy Java-ra specializálódott nyílt forráskódú elemző eszköz, amely a FindBugs projekt továbbfejlesztett változata. A forráskódot vagy a lefordított bájtkódot számos előre definiált hibaminta alapján vizsgálja meg, ezáltal könnyebben tudja azonosítani az általános problémákat, mint például a null pointerek használatát, erőforrás kezelési hibákat vagy nem biztonságos kódrészeket.

Az Erlang nyelvhez is készültek különböző kódelemző alkalmazások, ezek közül az egyik legnépszerűbb a Dialyzer (Discrepancy Analyzer for Erlang) [24], amely 2007 óta az Erlang/OTP (Open Telecom Platform)[3] része. A Dialyzer statikus elemzéssel és típuskövetkeztetésen alapuló elemzéssel azonosítja a forráskódban előfordulő szoftverhibákat, például típushibákat, kivételt dobó kódrészeket, elérhetetlenné vált logikai ágakat vagy versenyhelyzeteket. Más statikus elemző eszközökhöz képest lényegesen kevesebb hamis riasztást generál. Automatikusan is tud működni, valamint forráskódon és bájtkódon is le lehet futtatni a vizsgálatot. Emellett többféle módón is használható az eszköz, például parancssorból vagy grafikus felületen keresztül.

2.4. Erlang

Az Erlang [3, 4] egy nyílt forráskódú, deklaratív funkcionális programozási nyelv, amelyet eredetileg a telekommunikációs rendszerek fejlesztésére hozott létre az Ericsson az 1980-as évek közepén. Alapvető feladata, hogy egyszerűvé tegye a párhuzamosan futtatható elosztott rendszerek készítését. Rengeteg különböző ipar-

[3] Az Erlang programozási nyelv köré épített könyvtárak és tervezési alapelvek gyűjteménye.

ágban használják [25], például telekommunikációban (Ericcson, Nokia), pénzügyben (Goldman Sachs), közösségi médiában (WhatsApp), üzenetközvetítő (message broker) rendszerekben (RabbitMQ), blockchain technológiában (Aeternity) vagy az IoT-ben (Internet of Things). Sok vállalkozás által kedvelt nyelv, mivel több olyan tulajdonsággal is rendelkezik, amelyek jelentősen elősegítik a biztonságos fejlesztést és a támadásokkal szembeni védelmet. A legfontosabb jellemzői közé tartozik a megváltoztathatatlan adatszerkezetek biztosítása, az elosztottság támogatása, a párhuzamosság programozási nyelvi szintű megvalósítása és a hibatűrő modell [4].

2.4.1. Az Erlang legfontosabb nyelvi jellemzői

Megváltoztathatatlan adatszerkezet

A funkcionális nyelvcsaládba való tartozás miatt az Erlangban az adatok immutable típusúak, vagyis létrehozás után nem módosíthatók. Ez megakadályozza az értékek véletlen megváltoztatását futás közben. Mivel az adatok változatlanok, a függvények többszöri hívása esetén ugyanarra a bemenetre mindig ugyanazt a kimenetet adják vissza. Ez a determinisztikus működés segít bizonyítani a szoftver helyességét, emellett nagy mértékben növeli a program stabilitását és kiszámíthatóságát, ami különösen lényeges a hibamentes párhuzamos futásokhoz.

Konkurens programozás

Az Erlang egyik alapvető jellemzője, hogy támogatja a konkurens programok fejlesztését, amelyek nélkülözhetetlenek a telekommunikációs rendszerek számára. A folyamatok egymástól teljesen elszigetelve, izoláltan futnak anélkül, hogy kihatással lennének a másik szálra. Saját memóriaterülettel rendelkeznek, és csakis üzenetküldéseken keresztül kommunikálnak egymással. Ez a fajta aszinkron kommunikáció biztonságosabb, mert a folyamatoknak nincs közös memóriájuk, és így kevesebb eséllyel jönnek elő a tipikus párhuzamos programozásból ismert problémák. Ilyenek például a folyamatok éheztetése (starvation), holtpont (deadlock) állapotok előidézése vagy versenyhelyzetek (race condition) kialakulása [7].

Hibatűrési modell

Az Erlang szintén kulcsfontosságú jellemzője a hibatűrés. A nyelv "Let it crash" [4] filozófiája szerint nem minden hibát kell megelőzni, hanem gondoskodni kell arról,

hogy a hibák ne terjedhessenek tovább a rendszerben. Ha egy folyamat hibát észlel, akkor leáll, és annak helyreállítását egy felügyelő (supervisor) folyamat végzi. A supervisor futás közben figyeli az alatta lévő műveleteket, és hiba esetén például újraindíthatja a problémásakat. Ez a megközelítés lehetővé teszi, hogy hiba esetén is működőképes maradjon az egész program.

Elosztott programozás

A nyelv elosztottságának köszönhetően a folyamatok akár több virtuális gépen is szétoszthatók, amelyek között az információáramlás változatlanul működik. A csomópontok között a kommunikáció transzparens, így ha kiesik egy pont, a többi ugyanúgy tud tovább működni. Ez az architektúra tehát jelentősen növeli a rendszer rendelkezésre állását (availability). Elosztott rendszerek létrehozásánál a következő szempontokra kell figyelni [7]:

- A hálózat biztonsága: Az Erlangot eredetileg zárt és védett hálózatokra tervezték, így nem tartalmaz beépített biztonságos kommunikációs réteget, amely megvédené a programot külső támadásoktól. Napjainkban ez már alapkövetelménnyé vált az elosztott programokban, azonban egy ilyen protokoll megvalósítása rendkívül időigényes és költséges lehet.
- A hálózat megbízhatósága: Hálózati kimaradás esetén fel kell készülni a lehetséges következményekre, pontosabban az információvesztésre és a csomópontok közötti kommunikáció megszakadására. Az Erlang aszinkron üzenetküldése és monitorozási képessége lehetővé teszi a fennakadások hatékony kezelését.
- Késleltetés kezelése: A hálózaton folytatott kommunikáció lassabb, mint a lokális folyamatok közötti. Az Erlangban implementált izolált folyamatmodell és aszinkron üzenetküldés segít az elosztott alkalmazásoknak, hogy stabilan tudjanak működni nagyobb késleltetés mellett is.
- Garantált hálózati topológia: Az elosztott rendszereknek el kell különülniük a hálózati csomópontok elrendezésétől és számától, mivel ezek állandóan módosulhatnak.
- Nagyméretű adathalmaz küldése a hálózaton keresztül: Az Erlang kommunikációs módszere garantálja az üzenetek helyes küldési sorrendjét. Azonban mivel ez a művelet rendkívül költséges, nagyobb üzenetek küldése esetén a rövidebbek is várakozásra kényszerülnek.

- Magas átviteli költség: Adatok küldésekor további költségeket jelenthetnek a szerializációs és adattömörítési műveletek.
- Hálózat résztvevői közötti formai különbségek kezelése: A további feleknek képesnek kell lenniük az adatcserére vonatkozó protokollok és formátumok kezelésére.

2.4.2. Shared Everything és Shared Nothing koncepciók

Számos elosztott rendszert fejlesztenek C, C++ vagy Java programozási nyelveken, amelyek a Shared Everything koncepcióra épülnek. Ez azt jelenti, hogy a rendszer összes folyamata vagy komponense közösen fér hozzá az erőforrásokhoz, például a memóriához vagy az adatbázisokhoz. Így a program különböző részei ugyanazt az adatstruktúrát olvashatják és módosíthatják, ami gyorsabb és hatékonyabb kommunikációt jelent. A teljesítményre való pozitív hatása mellett ez a megosztási módszer komoly kockázatokkal is járhat, ugyanis a párhuzamos hozzáférések miatt versenyhelyzet (race condition) alakulhat ki, amely akár teljes rendszerhibához is vezethet.

Ezzel szemben az Erlang a Shared Nothing [26, 6] elvet követi, ahol az egyes folyamatok vagy komponensek egymás mellett teljesen elszigetelten működnek, és nem osztoznak semmilyen közös erőforráson. Minden folyamat a saját adataival és memóriaterületével dolgozik, a kommunikáció pedig kizárólag üzenetküldéseken keresztül történik. Ez a fajta megközelítés nagy mértékben növeli a rendszer megbízhatóságát, mivel kizárja a közös memóriahasználatból adódó versenyhelyzeteket (race condition) és adatütközéseket.

3. fejezet

Sérülékeny kódrészletek és azok elemzései

A következő fejezetben részletesen bemutatom a leggyakoribb biztonsági problémákat az Erlang programozási nyelvben, néhányat kódrészlettel is szemléltetve. Továbbá ismertetem a RefactorErl elemző eszközt, annak használatát és az elérhető biztonsági kockázatokat azonosító függvényeket.

3.1. Erlang gyakori sérülékenységei

Annak ellenére, hogy az Erlang programozási nyelvet megbízható és hibatűrő rendszernek tervezték, itt is akadnak olyan hibák, amelyek biztonsági kockázatot jelenthetnek [27]. Az Erlangban előforduló sérülékenységeket a [7] dolgozat a következő kategóriákba sorolja:

- együttműködésből fakadó sebezhetőségek
- konkurens programozásból eredő hibák
- elosztott programozásból adódó sérülékenységek
- beszúrásos (injection) támadások
- memória túlterhelésből származó támadások

A következőkben ezeket részletesebben kifejtem.

3.1.1. Együttműködésből fakadó sebezhetőségek

Különböző programozási nyelvek együttműködési képessége azt jelenti, hogy az eltérő nyelven írt komponensek miként tudnak adatokat cserélni és egymással együtt-

működni. Az Erlang több módon képes kooperálni az idegen komponensekkel:

Porthasználat

Lehetővé teszi az Erlang virtuális gépén kívüli külső programokkal való kapcsolattartást. Az ehhez szükséges port az `open_port/2` függvényhívással nyitható meg, a megfelelő argumentumok megadásával. Segítségével a program külső folyamatokat (például C, Python vagy shell script) tud elindítani és kommunikálni velük. A kommunikáció általában bináris formában történik, így elengedhetetlen a beérkező adatok formátumának és tartalmának szigorú ellenőrzése a feldolgozás előtt.

Gyakori hiba szokott lenni, ha az üzenet formátuma nem megfelelő, ha a bemenet nincs leellenőrizve, vagy ha a kommunikációs protokoll nem egyezik meg az Erlang virtuális gépe és a külső program között. Ilyen esetekben instabillá válhat, vagy össze is omolhat az egész rendszer.

További kockázatot jelenthet, ha a porton keresztül futtatott külső program megbízhatatlan forrásból származik, vagy ha a paraméterek felhasználói bemeneten keresztül vannak megadva. Ekkor a szoftver injection típusú támadásnak lehet kitéve.

A 3.1-es példában látható kód egy olyan port nyitását szemlélteti, amelynek a paramétere a felhasználói bemenetből érkezik. Ez több szempontból is veszélyes az egész alkalmazásra nézve, hiszen nincs semmilyen ellenőrzés definiálva a parancssori argumentumra. A 3.2-es példa kimenet esetében a kód egy OS parancsot futtat le, amely eredményként kilistázza az összes fájlt és mappát.

```
1 -module(example1).
2 -export([unsafe_port/1]).
3
4 unsafe_port(Arg) ->
5   Port = open_port({spawn, Arg}, [stream, exit_status]),
6   receive
7     {Port, {data, Data}} ->
8       io:format("Adat: ~p~n", [Data])
9   end.
```

3.1. forráskód. Nem biztonságos port nyitása felhasználói bemenő paraméterrel

```
Eshell V12.2.1  (abort with ^G)
1> c(example1).
{ok, example1}
2> example1:unsafe_port("ls -a").
Adat: ".\n..\nCOPYRIGHT.txt\nDockerfile\nEmakefile\nLICENSE.txt\n"
```

3.2. forráskód. Nem biztonságos port nyitás eredménye bemenő paraméterrel

Dinamikusan betölthető driverek

Az Erlang lehetőséget ad arra, hogy külső komponensek integrálva legyenek a futtató környezetbe a dinamikusan betölthető könyvtárak által. Ilyenkor a kintről érkező kód az Erlang virtuális gépén fut le, elkerülve ezzel a távoli folyamatok közötti kommunikációs költségeket.

A dinamikus betöltés az `erl_ddll` modulon keresztül érhető el, és a következő függvények használhatók erre [7]:

- `try_load/3`
- `load/2`
- `load_drive/2`
- `reload/2`
- `reload_driver/2`

Mivel a betöltött fájlok a virtuális gépen belül futnak, így különösen fontos odafigyelni a külső fájlok és a bemeneti paraméterek ellenőrzésére. Ellenkező esetben jogosulatlan műveletek végrehajtásához és a szoftver kiszámíthatatlan viselkedéséhez vezethet.

A 3.3-as példakód egy olyan dinamikus driver betöltését szemlélteti, amely paraméteren keresztül kapja meg a betöltendő driver útvonalát. Az ellenőrizetlen argumentum miatt sérülékenynek számít.

```
-module(example2).
-export([dynamic_load/1]).

dynamic_load(Path) ->
  erl_ddll:load(Path, []).
```

3.3. forráskód. Dinamikus driverek ellenőrizetlen betöltése

NIF-ek (Native Implemented Functions)

A NIF-ek (Native Implemented Functions) általában C nyelven írt függvények, amelyek közvetlenül meghívhatók az Erlang kódjából. A `load_nif/2` függvény segítségével tölthetők be és használhatók a natív hívások úgy, mint az Erlang nyelvben írtak.

Az előző példakódhoz hasonlóan, a 3.4-es forráskódban is egy sebezhető betöltés látható, ahol az importálni kívánt függvényt ellenőrizetlen felhasználói bemeneten keresztül kell megadni.

```
1 -module(example3).
2 -export([nif_load/1]).
3
4 nif_load(Path) ->
5   erlang:load_nif(Path, 0).
```

3.4. forráskód. A NIF sérülékeny betöltése

A függvények itt is a virtuális gépen belül futnak, ezért alapvető dolog odafigyelni a biztonságos memóriakezelésre és a végtelen ciklusok mellőzésére. Máskülönben adatszivárgást és adatokhoz való jogosulatlan hozzáférést eredményezhet, vagy akár le is állhat az egész rendszer.

3.1.2. Konkurens programozásból eredő hibák

Az Erlang népszerűségét a nyelvi szinten támogatott konkurens programozásnak köszönheti. A folyamatok könnyű létrehozása és izolált futása megkönnyíti a párhuzamos rendszerek fejlesztését, viszont számos új hibalehetőséget is eredményezhet, például versenyhelyzeteket vagy inkonzisztens állapotokat.

Folyamatkezelés

Különböző független folyamatokat a `spawn/1-4` függvénnyel lehet létrehozni, majd a `link/1` segítségével a szülőfolyamathoz kapcsolni.

Mivel ezek egymástól független, különálló műveletek, előfordulhat, hogy versenyhelyzet alakul ki. A `spawn/1-4` függvénnyel létrehozott folyamat még a `link/1` meghívása előtt terminál, ami instabilitást eredményezhet. Ezt a hibát könnyedén ki lehet kerülni a `spawn_link/1-4` függvény használatával, amellyel ugyanúgy létre lehet hozni új folyamatokat és összekapcsolni azokat egymással egy atomi műveletben.

A 3.5-ös példában látható kód előidézheti a fent említett versenyhelyzetet, ahol a `link` függvény előtt terminál a `spawn` hívás.

```
1 -module(example4).
2 -export([unsafe_spawn_link/0]).
3
4 unsafe_spawn_link() ->
5   Pid = spawn(math, sqrt, [25]),
6   link(Pid),
7   Pid ! ok.
```

3.5. forráskód. A `spawn` és `link` függvények sérülékeny alkalmazása

Ütemezés állítás

Az Erlang programozási nyelv egy ütemezőt (scheduler) használ, amely párhuzamosan kezeli a futó folyamatokat. A scheduler mindenkinek egy meghatározott prioritást ad, ami befolyásolja, hogy egy művelet milyen gyakran és mennyi erőforrást kaphat a futás során. Alapból mindegyik `normal` prioritáson fut, viszont a `process_flag/2` függvény segítségével módosítható még `low`, `high` vagy `max` értékekre, amelynek a használata a 3.6-os példában látható.

```
1 process_flag(priority, PriorityLevel).
```

3.6. forráskód. A `process_flag` függvény alkalmazása

Óvatosan kell kezelni a prioritás állítását, ugyanis nem determinisztikus viselkedést és folyamat éheztetést (starvation) idézhet elő [7]. Az utóbbi általában akkor fordul elő, ha egy folyamatot `high` vagy `max` értékek egyikére állítanak be, így az alacsonyabb prioritású folyamatok nem jutnak elegendő erőforráshoz, és nem tudnak lefutni.

ETS tábla kezelés

Az Erlang nyelvbe beépített `ets` modulban található az ETS (Erlang Term Storage) adatkezelő eszköz, amely elősegíti a kulcs-érték párok tárolását különféle táblázatokban, és biztosítja ezek gyors és hatékony elérését. A táblákat folyamatok hozzák létre, és egy táblához bármelyik folyamat hozzáférhet. Az adatmódosítások,

mint például a beszúrás, a törlés és a frissítés, atomi és izolált módon történnek. Ez azt jelenti, hogy ha egy folyamat éppen szerkeszt egy táblát, akkor egy másik folyamat nem férhet hozzá a módosuló táblához [7].

Ellenben ez a megkötés már nem terjed ki a tábla bejárására. Abban az esetben, ha egy folyamat éppen olvassa a táblát, viszont egy másik ugyanazt a táblát szerkeszti, akkor inkonzisztens állapotot és versenyhelyzetet eredményezhet. Ilyenkor a lekérdezés hibás adatokat adhat vissza, de előfordulhat az is, hogy futásidejű hibát okoz.

A 3.7-es példában látható kód a fent leírt állapot kialakulását szemlélteti az ETS táblában, a 3.8-ban pedig annak futtatása és egyik lehetséges eredménye látható. Attól függ a kód kimenete, hogy melyik folyamat fut le hamarabb: az elem törlése (output: `[]`) vagy a tábla lekérdezése (output: `[{k,1}]`).

```
1  -module(example5).
2  -export([unsafe_ets/0]).
3
4  unsafe_ets() ->
5    Table = ets:new(t, [set, public]),
6    ets:insert(Table, {k, 1}),
7    spawn(
8      fun() ->
9        ets:delete(Table, k)
10     end
11   ),
12   Result = ets:lookup(Table, k),
13   io:format("Lekerdezes eredmenye: ~p~n", [Result]).
```

3.7. forráskód. Az ETS tábla egyidejű módosítása

```
1  Eshell V12.2.1  (abort with ^G)
2  1> c(example5).
3  {ok, example5}
4  2> example5:unsafe_ets().
5  Lekerdezes eredmenye: [{k,1}]
6  ok
```

3.8. forráskód. Az ETS tábla egyidejű módosításának eredménye

A fent említett problémák megelőzésére létezik ugyanebben a modulban a `safe_fixtable/2` függvény (3.9-es kódrészlet), amely ideiglenesen rögzíti a tábla állapotát a bejárás idejére, így garantáltan megbízható és konzisztens adatokat ad vissza.

```
1 safe_fixtable(Table, true | false).
```

3.9. forráskód. A `safe_fixtable` függvény alkalmazása

3.1.3. Elosztott programozásból adódó sérülékenységek

Az Erlangot eredetileg zárt és megbízható hálózatokra tervezték, ezért az alapértelmezett rendszerbeállítások nem veszik figyelembe annyira azokat a biztonsági kockázatokat, amelyek a mai nyílt hálózatokra jellemzőek. Az Erlang virtuális gépe alapszintű hozzáférés vezérlést támogat, így egy távoli csatlakozott csomópont teljes hozzáférést kap nemcsak az elosztott hálózat minden résztvevőjéhez, hanem a teljes virtuális géphez is [6].

Hálózati hozzáférés

A `net_kernel` modulon belül lévő `allow/1`, `connect_node/1` és `start/1` függvények megkönnyítik a csomópontok hálózathoz való kapcsolódását. Ellenőrizetlen vagy nem megfelelő használatuk esetén akármilyen megbízhatatlan, ismeretlen helyről származó gép is csatlakozhat a hálózathoz, ezzel teljes körű hozzáférést biztosítva mindenhez.

SSl/TLS beállítások

A közbeékelődéses (Man-in-the-Middle) támadások és adatlopások megelőzése érdekében érdemes elkerülni az alfejezetben felsorolt SSL-3.0 és TLS-1.0 protokollokra vonatkozó beállításokat, amelyek az `ssl` modulban található `listen/2`, `ssl_accpet/1-3`, `connect/2-3`, `handshake/1-3` és `handshake_continue/2-3` függvények meghívásán keresztül állíthatók. A kompatibilitás megőrzése miatt benne maradtak olyan régi és elavult beállítások is, amelyek súlyos sebezhetőségeket eredményezhetnek [6, 7]:

- `{padding_check, false}`: A padding ellenőrzésének kikapcsolásával az alkalmazás védtelen lesz POODLE (Padding Oracle On Downgraded Legacy Encryption) támadásokkal szemben.
- `{beast_mitigation, disabled}`: A beast mitigáció kikapcsolása lehetőséget nyit a BEAST (Browser Exploit Against SSL/TLS) támadásoknak, melynek célja a titkosított session cookie-k megszerzése.
- `{fallback, true}`: A fallback opció lehetővé teszi, hogy a kapcsolat egy régebbi és gyengébb TLS protokollra váltson vissza.
- `{dh, tetszőleges binárissal}`: Szükséges argumentum a Diffie-Hellman kulcscseréhez, amely már nem támogatott a TLS-1.3 verzió felett.
- `{dhfile, tetszőleges fájlnévvel}`: Szintén nem támogatott a TLS-1.3 verzió felett.

3.1.4. Beszúrásos (injection) támadások

Az injection típusú támadás elsősorban akkor fordul elő, amikor a megbízhatatlan forrásból kapott argumentumot a szoftver parancsként vagy kódként hajtja végre. Eredményezhet távoli kódfuttatást (RCE), adatlopást vagy szolgáltatásmegtagadást (DoS / DDoS). A támadások tipikusan a következőképpen történhetnek Erlangban.

OS parancsok

Az `os` modulban lévő `cmd/1-2` és `putenv/2` függvények segítségével hajthatók végre operációs rendszer szintű utasítások. Abban az esetben, ha nincs megfelelő paraméter validáció, valamint a kódban az utasítás és az argumentum szövegként van összefűzve, akkor a támadó könnyedén új parancsokat illeszthet be a `;` és `&&` jelek segítségével, vagy felülírhatja a már meglévőt a `|` jel beillesztésével (nem használható együtt a beillesztő jelekkel).

Ezek alapján a 3.10-es példában látható kód könnyen OS injection támadás áldozata lehet. A 3.11-es példában szemléltetett függvényhívás ki is használja az említett sérülékenységet.

```
1 -module(example6).
2 -export([os_injection/1]).
3
4 os_injection(Arg) ->
5   os:cmd("less " ++ Arg).
```

3.10. forráskód. Injection OS parancson keresztül

```
1 Eshell V12.2.1  (abort with ^G)
2 1> c(example6).
3 {ok, example6}
4 2> example6:os_injection("example6.txt && echo OS ; echo INJECT").
5 "Lorem ipsum.OS\nINJECT\n"
```

3.11. forráskód. Injection OS parancson keresztül eredménye

A `cmd/1-2` függvény helyett sokkal biztonságosabb az `open_port/2` hívást alkalmazni a `{spawn_executable, FileName}` paraméterrel [28], amely a parancssori argumentumokat listaként fogadja el.

Fájlokkal kapcsolatos műveletek

Ha a bemeneti fájl nincs megfelelően leellenőrizve, akkor a támadó akár rosszindulatú kódot juttathat be a rendszerbe, vagy nagy adatmennyiségű fájl beolvasása esetén atomtúlcsorduláshoz vezethet. A következő függvények használatánál fordulhat elő, így ezeknél érdemes jobban odafigyelni a bemeneti paraméterek ellenőrzésére:

- `consult/1`: Beolvassa az Erlang kifejezéseket a megadott fájlból, és visszatér egy `tuple` típussal végrehajtás eredményétől függően.
- `path_consult/2`: A paraméterként kapott útvonalról beolvassa az Erlang kifejezéseket a megadott fájlból.
- `eval/1-2`: Erlang kifejezéseket olvas be és értékel ki, és a függvényhívás nem tér vissza semmilyen értékkel.
- `path_eval/2`: Megkeresi a megadott fájlt, majd beolvassa és kiértékeli belőle az Erlang kifejezéseket. Nincs visszatérési érték.
- `path_script/2-3`: Megegyezik a `path_eval` függvénnyel, viszont itt van visszatérési érték.
- `script/1-2`: Ugyanaz, mint az `eval` függvény, csak van visszatérési értéke.

A 3.12-es példakódban szintén egy sebezhető fájlbeolvasás látható argumentumon keresztül.

```
1 -module(example7).
2 -export([file_injection/1]).
3
4 file_injection(Arg) ->
5   file:eval(Arg).
```

3.12. forráskód. Sérülékeny file művelet

Dinamikusan fordított és betöltött programkódok

A dinamikusan fordított és betöltött kódok ugyan rugalmasságot adhatnak, viszont különösen veszélyessé válhatnak, ha nincsenek megfelelően hitelesítve, mivel a kód korlátlan hozzáféréssel futhat a virtuális gépen belül. A `compile` modulban lévő `file/1-2` és `noenv_file/2`, valamint a `code` modulban található `atomic_load/1`, `load_abs/1`, `load_binary/3` és a `load_file/1` függvényeknél érdemes nagyobb hangsúlyt fektetni az argumentumok ellenőrzésére.

A 3.13-as példában látható kód a 3.14-es példa függvényhívás alapján beleteszi a paraméterként kapott szöveges kódrészletet egy fájlba (ha még nem létezik, akkor létrehozza), amelyet azonnal le is fordít, így az egyből használható lesz. A 3.15-ös példában látható a létrejött fájl.

```
1 -module(example8).
2 -export([dynamic_injection/1]).
3
4 dynamic_injection(Arg) ->
5   File = "./test_code.erl",
6   file:write_file(File, Arg),
7   compile:file(File).
```

3.13. forráskód. Sebezhető dinamikusan fordított és betöltött programkód

```
1 Eshell V12.2.1  (abort with ^G)
2 1> c(example8).
3 {ok, example8}
4 2> example8:dynamic_injection("-module(test_code).\n-export([test/0
    ]).\ntest() -> io:format(\"Hello World!~n\").\n").
```

```
{ok,test_code}
3> test_code:test().
Hello World!
ok
```

3.14. forráskód. Sebezhető dinamikusan fordított és betöltött programkód meghívása

```
-module(test_code).
-export([test/0]).
test() -> io:format("Hello World!~n").
```

3.15. forráskód. Sebezhető dinamikusan fordított és betöltött programkód eredménye

3.1.5. Memória túlterhelésből származó támadások

Erlangban az atomok értékei egy globális, fix mérettel rendelkező atomtáblában tárolódnak, amelynek a méretét a fejlesztők tudják módosítani a virtuális gép indításakor. Az új atomok futásidőben kerülnek be a táblába, viszont a fel nem használtak eltávolítására nincs lehetőség. Amint az atomok száma meghaladja a tábla maximális értékét, a virtuális gép összeomolhat, és szolgáltatáskiesést (DoS / DDoS) eredményezhet. Az atomkimerülés elkerülése érdekében óvatosan kell kezelni a dinamikus atomképzéseket és az adatok deszerializálását, illetve megfelelően le kell ellenőrizni a következő függvények paramétereit:

- Dinamikus atomképzés: `list_to_atom/1`, `binary_to_atom/1-2`
- Deszerializáció: `binary_to_term/2` függvény a `safe` beállítás nélkül
- URL feldolgozás: `http_uri:parse/1`
- XML elemzés: `xmerl_scan:file/1`, `xmerl_scan:string/1`, `xmerl_sax_parser:file/2`, `xmerl_sax_parser:stream/2`

Atomok létrehozásához érdemes elkerülni a `list_to_atom/1` és a `binary_to_atom/1-2` függvények használatát, és inkább a `list_to_existing_atom/1` illetve a `binary_to_existing_atom/1-2` függvényeket alkalmazni [29] a forráskódban. A `binary_to_term/2` hívást ajánlatos a `safe` beállítással együtt használni [30], ezzel biztonságosabbá téve a kódot.

A 3.16-os és 3.17-es példában a nem biztonságos `list_to_atom` függvényhívást szemléltetem, ahol a megadott `Arg` lista minden eleméből új atom jön létre. Mivel az Erlang atomtáblája fix méretű, és az atomok futás közben nem törlődnek, az ilyen típusú függvényhívás különösen veszélyes, ha a paraméter a felhasználói bemenetről érkezik, és nincs limitálva a mérete. Ebben az esetben egy támadó akár több százezres méretű listát is megadhat, amely könnyen atomtúlcsorduláshoz és rendszerleálláshoz vezethet.

```
1 -module(example9).
2 -export([unsafe_atom_gen/1]).
3
4 unsafe_atom_gen(Arg) ->
5   [list_to_atom(X) || X <- Arg].
```

3.16. forráskód. A `list_to_atom` függvény sérülékeny használata

```
1 Eshell V14.2.5 (press Ctrl+G to abort, type help(). for help)
2 (refactorerl@localhost)1> example9:unsafe_atom_gen([integer_to_list
     (X) || X <- lists:seq(1, 5000000)]).
3 ['1','2','3','4','5',...,'5000000']
```

3.17. forráskód. A sérülékeny `list_to_atom` függvény meghívása

3.2. RefactorErl

A RefactorErl [5, 31] egy nyílt forráskódú statikus elemző és transzformáló eszköz, amelyet az Eötvös Lóránd Tudományegyetem fejlesztett ki azért, hogy támogassa az Erlang fejlesztők mindennapi munkáját. Elsősorban olyan forráskód transzformációkat implementáltak benne, amelyek biztonságosan alakítják át a kódot anélkül, hogy megváltozna a program viselkedése vagy jelentése. Mióta publikálásra került a RefactorErl, a mai napig folyamatosan fejlesztik, és egyre több új funkcióval bővítik az alkalmazást. Ma már nem csak kódbeli refaktorálások hajthatók végre, hanem segít a kódrészek megértésében, karbantartásában, illetve különféle elemzésekben (például biztonsági) és keresésekben is. A RefactorErl statikus elemző szoftver Windows, Linux és macOS operációs rendszereken is képes futni.

A RefactorErl az Erlang nyelven írt forráskódot egy szemantikus programgráfban [32, 33] reprezentálja, amely a következő három rétegből épül fel [34]:

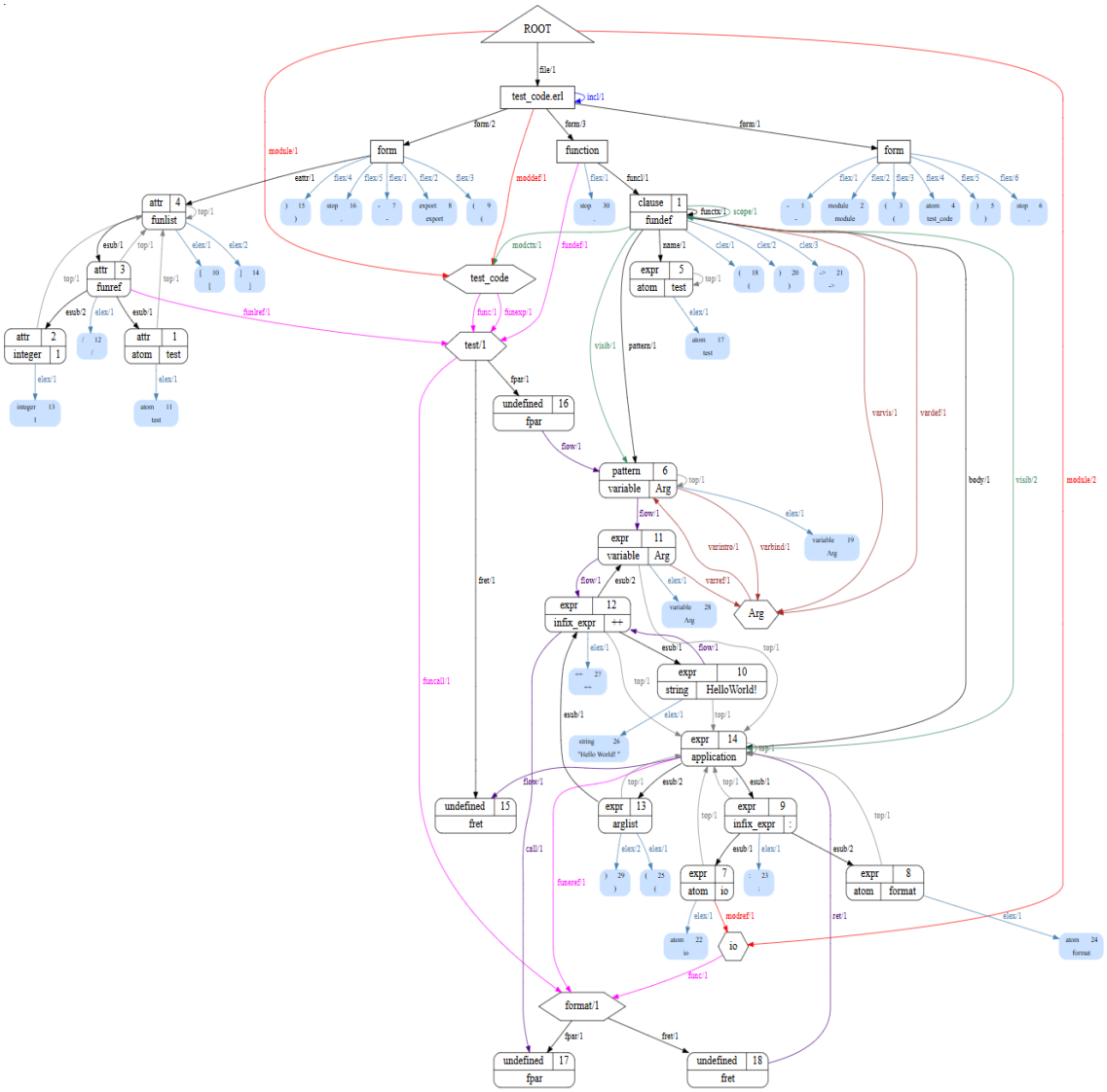
- Lexikális: Olyan tokeneket állít elő a forráskódból, amelyekből a szintaktikai elemző egy szintaxisfát épít.
- Szintaktikus: Létrehozza az elemzett Erlang modulok szintaxisfáját.
- Szemantikus: Különféle kapcsolatokkal és információkkal bővíti a szintaxisfát, mint például modul kapcsolatokkal, függvényhívási információkkal, definíciókkal vagy változók kötési és hivatkozási helyeivel.

Az elemzéseknél ez a gráf kerül bejárásra, és az abban tárolt információkat használja fel a további elemzésekre az eszköz. Teljesen új éleket és csúcsokat is hozzá lehet adni a már meglévő gráfhoz.

A 3.18-as példában látható kód összefűzi a `"Hello World! "` és a felhasználói bemenetről kapott szöveget, majd kiírja az outputra. Közvetlenül utána a 3.1. ábrán látható a szemantikus programgráf, amelyet a RefactorErl a 3.18-as kód alapján épített fel.

```
1 -module(test_code).
2 -export([test/1]).
3
4 test(Arg) ->
5   io:format("Hello World! " ++ Arg).
```

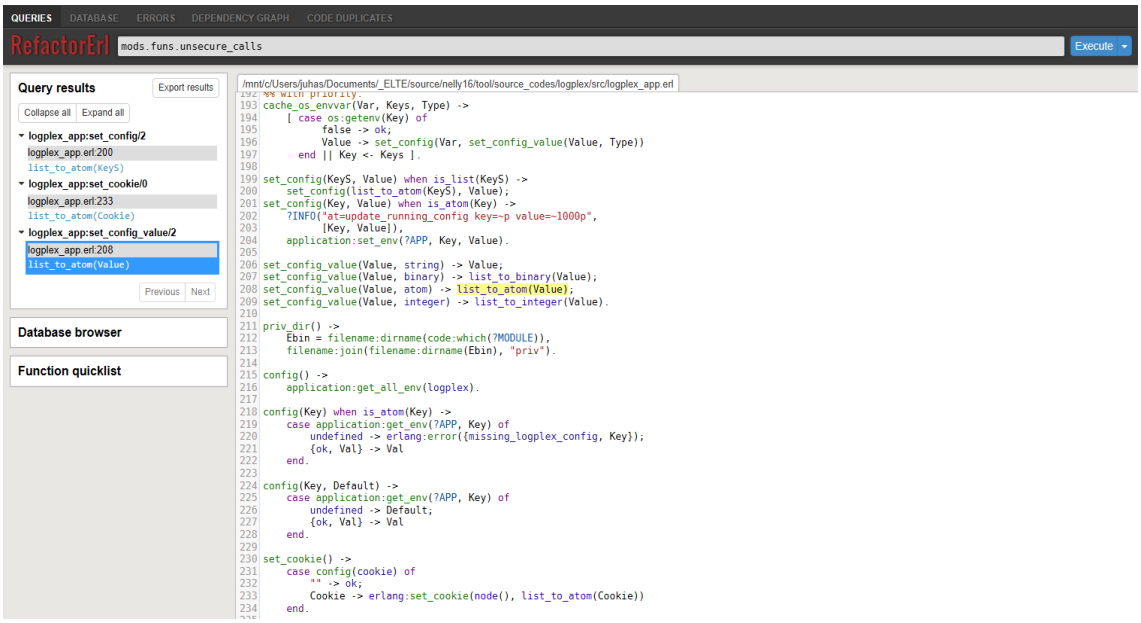
3.18. forráskód. Tesztkód az SPG gráf szemléltetésére



3.1. ábra. A tesztkód SPG gráf felépítése

3.2.1. Az eszköz használata

A RefactorErl tökéletesen integrálható különböző fejlesztői környezetekbe, mint például Visual Studio Code, Eclipse, Vim vagy Emacs [5]. Emellett könnyen használható a webes felhasználói felületről (3.2. ábra) vagy parancssorból, az eszköz saját interaktív shell-jén keresztül (3.3. ábra). Mindkét felületen ugyanazon műveletek hajthatók végre, például meg lehet tekinteni az adatbázis tartalmát vagy lekérdezéseket írni a RefactorErl szemantikus lekérdezőnyelve segítségével.



3.2. ábra. RefactorErl webes felület

```
nelly@DESKTOP-13H9LC9:/mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool$ bin/referl
Erlang/OTP 25 [erts-13.2] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [jit:ns]

Eshell V13.2  (abort with ^G)
(refactorerl@localhost)1> ri:ls().
{{ok,["/mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/source_codes/logplex/src/logplex.hrl",
      "/mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/source_codes/logplex/src/logplex_logging.hrl",
      "/mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/source_codes/logplex/src/logplex_app.erl"]},
 {error,[]}}
ok
(refactorerl@localhost)2> ri:q("mods.funs.unsecure_calls").
logplex_app:set_config/2
    list_to_atom(KeyS)
logplex_app:set_cookie/0
    list_to_atom(Cookie)
logplex_app:set_config_value/2
    list_to_atom(Value)
ok
(refactorerl@localhost)3>
```

3.3. ábra. RefactorErl shell script

3.2.2. Elérhető biztonsági ellenőrzések

A RefactorErl eszközben több olyan elemzés is elérhető, amelyek képesek felsorolni a projektben előforduló potenciálisan sérülékeny függvényhívásokat, és lehetőséget kínálnak a keresés konkrét típusokra való szűkítésére is. A 3.2. és a 3.3. ábrák bemutatják az `unsecure_calls` lekérdezés alkalmazását webes felületen és parancssorból indítva, amely a Logplex [35] projektben található összes sebezhető függvényhívást kilistázza.

A fejlesztők definiálhatnak egy fehérlistát a téves riasztások elkerülése érdekében. Az ebben szereplő függvényeket és azok paramétereit az elemzés során megbízhatónak tekinti a szoftver.

A 3.1. táblázat az elérhető lekérdezéseket mutatja be, amelyek segítségével a rendszer azonosítani tudja a potenciális sérülékenységeket a forráskódban [36, 11].

Függvény neve	**Leírása**
unsecure_calls	Felsorolja az összes lehetséges sérülékenységet
unsecure_interoperability	Azonosítja az együttműködésből eredő sebezhetőségeket
unsecure_concurrency	Felsorolja a konkurens programozásból fakadó hibalehetőségeket
unsecure_os_call	Ellenőrzi a nem biztonságos OS parancsokat
unsecure_port_creation	Azonosítja a port létrehozással kapcsolatos problémákat
unsecure_port_drivers	Felsorolja a dinamikusan betölthető könyvtárak használatából eredő sebezhetőségeket
unsecure_file_operation	Ellenőrzi az ellenőrizetlen fájlkezelési utasításokat
unstable_call	Megmutatja az atomok dinamikus létrehozásából eredő sérülékenységet
nif_calls	Azonosítja a NIF függvényekből származó sebezhetőségeket
decommissioned_crypto	Felsorolja az elavultnak számító kriptográfiai műveleteket
unsecure_compile_operations	Ellenőrzi az ismeretlen helyről származó programkódok fordítását és betöltését
unsecure_process_linkage	Felsorolja a folyamatok nem megfelelő összekapcsolásából fakadó sérülékenységeket
unsecure_prioritization	Kimutatja a folyamatok prioritás módosítása okozta hibalehetőségeket
unsecure_ets_traversal	Azonosítja az ETS táblák bejárásából eredő sebezhetőségeket
unsafe_network	Ellenőrzi a hálózati rendszerrel kapcsolatos hibalehetőségeket

unsecure_xml_usage	Azonosítja az xml feldolgozásból adódó sebezhetőségeket
unsecure_communication	Felsorolja kommunikációs beállításokból fakadó sérülékenységet

3.1. táblázat. Sérülékenységek azonosítására szolgáló függvényhívások a RefactorErl-ben

4. fejezet

Sérülékeny Erlang kódok biztonságossá transzformálásának bemutatása

Ebben a fejezetben először szemléltetem a felmerülő problémákat, és bemutatom, hogy miért szükséges ezek biztonságossá tétele. Majd áttekintést adok arról, hogy milyen transzformációkkal lehet biztonságosabbá tenni bizonyos sérülékeny Erlang kódrészleteket a RefactorErl segítségével. A végén ismertetem az általam definiált transzformációkat és néhány további átalakítási lehetőséget.

4.1. A probléma bemutatása

A sérülékeny kódrészek futás előtti felismerése jelentősen megkönnyíti a fejlesztők mindennapi munkáját, mivel a legtöbb potenciálisan támadható rés már a fejlesztés korai fázisában azonosítható és lefedhető. Azonban ezek kézi javítása gyakran újabb hibákat eredményezhet, ezért különösen fontosak az automatikus transzformációk, amelyek ezt a folyamatot megbízhatóvá teszik.

Az Erlang nyelvben több olyan függvény is előfordul, amelyek ránézésre nem jelentenek problémát, viszont bizonyos bemenetek vagy esetek mellett mégis komoly biztonsági kockázatot rejtenek. A RefactorErl eszköz segítségével azonosíthatók a projektben előforduló sebezhető hívások, így elkerülhetők az ebből adódó tipikus problémák, mint például az OS injection vagy az atomtúlcsordulás.

A következőkben néhány olyan példát mutatok be, amelyeket a RefactorErl sérülékeny hívásként azonosít.

os:cmd hívása ismeretlen bemenettel

A 4.1-es példában látható, hogy az `os:cmd` paramétere egy megbízhatatlan külső forrásból származik (`Command`), amelyet ellenőrzések nélkül egyszerűen végrehajt a kód.

Ilyen formában ez a hívás azért nagyon veszélyes, mert bármilyen rosszindulatú parancs is lefuthat a szerveren. Itt például a program hiába vár a `"cat"` utasítás után egy fájlnevet, hogy kiírja annak tartalmát, a fájl megadása után a bemenethez még könnyedén hozzáilleszthetők OS utasítások, vagy akár felül is írható az eredeti parancs.

```
1 -module(cmd_example).
2 -export([run_os/1]).
3
4 run_os(Command) ->
5   os:cmd("cat " ++ Command).
```

4.1. forráskód. `os:cmd` hívása ismeretlen bemenettel

Ebben az esetben szükségszerű bevezetni egy olyan ellenőrző függvényt, amely megvizsgálja a bemeneti paramétert, hogy vannak-e benne támadásra utaló elemek. Értelemszerűen ha talált benne ilyet, nem engedi végrehajtani a parancsot, és hibát dob. Ellenkező esetben minden további nélkül végrehajtható az eredeti utasítás, mivel a bemeneti paraméter biztonságosnak lett nyilvánítva.

Listagenerátoros list_to_atom függvény használata

A 4.2-es kódon szemléltetett részben a `list_to_atom(integer_to_list(X))` függvény a generáló lista minden elemére lefut, tehát annyi új atomot hoz létre, ahány elem van a generáló listában. Ránézésre a mérete ismeretlen, mivel az `Arg` egy bemeneti paraméter, tehát bármekkora lehet.

Az Erlang atomtáblája fix méretű, és atomok törlésére futás közben nincs lehetőség. Ezért ha nincs limitálva az atomok létrehozásának száma, a támadó atomtúlcsordulást okozhat a rendszerben.

```
1 -module(lta_example).
2 -export([run_list_gen/1]).
3
4 run_list_gen(Arg) ->
5   [list_to_atom(integer_to_list(X)) || X <- lists:seq(1, Arg)].
```

4.2. forráskód. Listagenerátoros sérülékeny `list_to_atom` függvény használata

Az említett probléma elkerülésének érdekében be kell illeszteni a kódba egy méretellenőrző függvényt, majd alkalmazni a generált listára (ebben az esetben a `lists:seq(1, Arg)` részre). Amennyiben a lista mérete meghaladja az ellenőrző függvényben megadott értéket, a kód nem engedi végrehajtani a műveletet, és hibát dob. Egyébként minden további nélkül végrehajtható az eredeti függvényhívás.

spawn és link függvények külön alkalmazása

A 4.3-as kódon bemutatott példán a `spawn` és a `link` függvényeket külön alkalmazzák. Ilyen esetben gyakran versenyhelyzet alakulhat ki, ha a `spawn` hívással létrehozott folyamat még a `link` futása előtt terminál.

```
1 -module(sl_example).
2 -export([run_spawn/0]).
3
4 run_spawn() ->
5   Pid = spawn(module, task, []),
6     link(Pid).
```

4.3. forráskód. `spawn` és `link` függvények külön alkalmazása

A versenyhelyzet kialakulásának megelőzésére a két különálló függvény helyett egy atomi hívást kell használni, vagyis ebben az esetben a `spawn_link`-et, amely paramétere megegyezik a `spawn` függvényével.

A diplomamunkámban a bemutatott példákhoz hasonló sérülékeny függvények biztonságosabbá alakításával foglalkozom. Olyan automatikus transzformációkat implementálok, amelyek a sebezhető hívásokat megbízhatóbbá alakítják.

4.2. Transzformációs módszerek

Az átalakítások módosítják a kód szerkezetét és futásidejű viselkedését, hogy a program biztonságosabban tudjon működni. Néhány függvénycserés transzformáció szükség szerint a paramétereket is módosítja. Ez okból a transzformációk alkalmazásáról mindig a felhasználónak kell eldöntenie, hogy szeretné-e azt.

A megvalósítható módszereket több csoportra bontottam, amelyeket a későbbiekben részletesen kifejtek [27]:

- Bemeneti adat validációjának beszúrása: transzformáció során egy ellenőrző-függvényt illesztek bele a kódba, amelyet alkalmazok a megbízhatatlan bemeneti paraméterre.
- Nem megbízható függvények cseréje: átalakításkor a sérülékeny függvényhívást a biztonságos párjára cserélem, és szükség szerint módosítom a paramétereket is.
- Atomi összekapcsolás bevezetése: a két különálló lépésben végrehajtott sérülékeny műveletet egyetlen biztonságos atomi hívássá alakítom.

4.2.1. Bemeneti adat validációjának beszúrása

A paraméterek validációja az egyik legösszetettebb téma, hiszen rengeteg különféle típusú adat érkezhet felhasználói bemenetről: például fájlnév, elérési útvonal, szöveg (tartalmazhat akár OS parancsot vagy forráskódot), szám, lista és még sok más. Leggyakrabban az ellenőrizetlen argumentumok miatt válik sérülékennyé egy kódrész, amely során a rendszer számtalan támadásnak lehet kitéve.

Annak érdekében, hogy elkerülhetők legyenek az ebből eredő beszúrásos (injection), túlterheléses (DoS / DDoS) és egyéb támadások, egy olyan transzformációt futtatok le a kódon, amely beilleszt egy ellenőrző műveletet és alkalmazza a nem megbízható paraméterre. Abban az esetben, ha az argumentum nem megy át az ellenőrzésen, akkor a rendszer hibát fog dobni. Viszont, ha teljesül a feltétel, akkor folytatódik tovább a kiértékelés.

Ebben az alfejezetben említett `check_input` és `size_check` ellenőrző függvényeket a transzformáció során automatikusan beleteszem a fájlba. A generált függvények törzse tetszőlegesen módosítható, így a felhasználó a saját igényei szerint alakíthatja az ellenőrzést.

A következőkben az `os:cmd`-t használom ennek bemutatására.

$$\frac{\texttt{os:cmd}(X)}{\textbf{case } G(X) \textbf{ of } \{true \ \rightarrow H(),\ false \ \rightarrow \ \texttt{os:cmd}(X)\} \textbf{ end}}$$

Ebben az esetben az `X` bemeneti adat `string` típusú. A vonal alatti komponensben az átalakítást követő állapotot mutatom, ahol bevezetésre kerül egy új `G` ellenőrző függvény, amelyet az `os:cmd` esetében annak paraméterére alkalmazok. Amennyiben az argumentumban nincsenek támadásra utaló részek (`false` értéket ad vissza `G`), úgy az eredeti hívás hajtódik végre. Ellenkező esetben a `H` művelet fut le, amely például kiírja a hibát, ami miatt nem fut le az eredeti hívás.

A következő példákban kódrészletekkel is szemléltetem az `os:cmd`-re alkalmazható transzformációt. A 4.4-es példában az átalakítás előtti állapot látható, ahol meghívom az `os:cmd`-t az `Expr1` ismeretlen paraméterrel. A 4.5-ös példában pedig már a módosított kód olvasható, ahol az újonnan beszúrt `check_input` függvényt alkalmazom a nem biztonságos `Expr1` argumentumra, amely megvizsgálja, hogy az argumentum tartalmazza-e a megadott speciális karakterek valamelyikét. Ha nem szerepel benne ilyen, akkor végrehajtom az `os:cmd` hívást, egyébként a `throw(Error)` rész fut le.

```
1 do_command(Expr1) ->
2   os:cmd(Expr1).
```

4.4. forráskód. Sérülékeny `os:cmd` hívás transzformáció előtt

```
1 do_command(Expr1) ->
2   case check_input(Expr1) of
3     true -> throw(Error);
4     false -> os:cmd(Expr1)
5   end.
6
7 check_input(Pattern) ->
8   lists:any(fun(E) -> string:find(Pattern, E) =/= nomatch end, [";"
      , "&&", "|"]).
```

4.5. forráskód. Sérülékeny `os:cmd` hívás transzformáció után

Ennél a részfejezetnél külön kiemelem az erőforrások limitálását, például az atomok létrehozásánál, az ETS táblák méretének beállításánál, vagy bizonyos folyama-

toknál. Szintén akkor a legkritikusabb a helyzet, ha a paraméter felhasználói bemenetről érkezik, ugyanis könnyedén túlterheléses (DoS / DDoS) támadás következhet be. Ez ellen egy hasonló átalakítást vezettem be, mint ami a 4.5-ös példában látható. Az átalakítás során szintén egy új blokkal és egy ellenőrző függvénnyel egészül ki a kód, amelyet nem minden esetben a sérülékeny hívás argumentumára alkalmazok. A továbbiakban ezt a `list_to_atom` függvényen keresztül szemléltetem.

$$\frac{[\texttt{list\_to\_atom}(X) \mid\mid X \leftarrow Xs]}{\begin{array}{c}\textbf{case } G(Xs) \textbf{ of } \{true \rightarrow [\texttt{list\_to\_atom}(X) \mid\mid X \leftarrow Xs], \\ false \rightarrow H()\} \textbf{ end}\end{array}} \quad \begin{array}{l}\text{WHEN} \\ var(X)\end{array}$$

A formálisan leírt ábrában a `list_to_atom` bemeneti `X` paramétere egy olyan elem, amely értéke az `Xs` listából származik. Az átalakítás szinte ugyanaz, mint az `os:cmd` alfejezetnél. Bekerül egy új `G` ellenőrző művelet, viszont nem `list_to_atom` argumentumára alkalmazom, hanem arra, amiből érkezik a paraméter (ebben az esetben az `Xs`-ből). Az eredménytől függően az eredeti kifejezés, vagy a `H` függvény (például hiba kiírás) fut le.

A 4.6-os és 4.7-es példában láthatók a kódok, amelyekkel szemléltetem az átalakítás előtti és utáni állapotokat. Beszúrok egy `size_check` ellenőrző függvényt, amelyet alkalmazok a bizonytalan `Expr1` paraméterre, mivel az generálja le a listát. Az eredménytől függően végrehajtom az eredeti sérülékenynek vélt, de már biztonságosnak ítélt részt, vagy a `throw(Error)` kifejezést.

```
1 gen_atom(Expr1) ->
2   [list_to_atom(Expr2) || Expr2 <- Expr1].
```

4.6. forráskód. Erőforrás korlátozó transzformáció előtt

```
1 gen_atom(Expr1) ->
2   case size_check(Expr1) of
3     true -> [list_to_atom(Expr2) || Expr2 <- Expr1];
4     false -> throw(Error)
5   end.
6
7 size_check(Pattern) ->
8     length(Pattern) < 5000000.
```

4.7. forráskód. Erőforrás korlátozó transzformáció után

4.2.2. Nem megbízható függvények cseréje

Az EEF (Erlang Ecosystem Foundation) Security WG [27] számos sérülékeny függvényre kínál egy sokkal megbízhatóbb és biztonságosabb megoldást:

- `list_to_atom/1` helyett `list_to_existing_atom/1`
- `binary_to_atom/1-2` helyett `binary_to_existing_atom/1-2`
- `os:cmd/1-2` helyett `open_port/2`: Első argumentumnak `{spawn, Command}` helyett `{spawn_executable, FileName}`
- `crypto:sign/4-5` helyett `public_key:sign/3-4`
- `crypto:verify/5-6` helyett `public_key:verify/4-5`
- `http_uri:parse/1` helyett `uri_string:parse/1`

A függvények kicserélésén alapuló transzformáció formális módon a következőképpen írható le.

$$\frac{UnsafeFun(X)}{SafeFun(Y)} \quad \begin{array}{l}\text{WHEN } UnsafeFun \in \\ (list\_to\_atom,\ binary\_to\_atom,\ os:cmd,\\ crypto:sign,\ crypto:verify,\ http\_uri:parse)\end{array}$$

Az `UnsafeFun` a WHEN részben megadott függvények közül bármelyik lehet, így a bemeneti paramétere (`X`) is ennek megfelelően változik. A már átalakított `SafeFun`, az `UnsafeFun` felsorolásbeli párja. Bizonyos esetekben (például a `list_to_atom/1` átalakításánál) a paraméterezés nem változik meg, vagyis a biztonságosabb változat ugyanolyan értékű paraméterekkel meghívható. Ilyenkor az `Y` bemeneti paraméter

megegyezik az `X`-el. Más függvények esetén a transzformáció során már szükséges a paraméterek módosítása a következőképpen:

- `os:cmd/1-2`: A bemeneti stringet fel kell bontani a futtatható állományra (`Filename`) és az argumentumok listájára (`{args, Args}`). Az átalakított függvény így néz ki: `open_port({spawn_executable, FileName},[{args, Args}])`.
- `crypto:sign/4-5`: A bemeneti paramétereket a transzformáció során a `public_key:sign/3-4` függvénynek megfelelő sorrendbe kell beilleszteni és módosítani. A `crypto:sign(Algorithm, DigestType, Msg, Key, Options)` hívás átalakítás után: `public_key:sign(Msg, DigestType, Key, Options)`.
- `crypto:verify/5-6`: A paramétereket itt is a `public_key:verify/4-5` függvény által elvárt módon kell alkalmazni. A `crypto:verify(Algorithm, DigestType, Msg, Signature, Key, Options)` hívás transzformáció után: `public_key:verify(Msg, DigestType, Signature, Key, Options)`.

A 4.8-as példakódban a `binary_to_atom` sebezhető művelet látható, majd a 4.9-es példában az erre lefuttatott transzformáció utáni eredmény található. Ebben a példában a sérülékeny `binary_to_atom` függvényt kicseréltem a biztonságosabb `binary_to_existing_atom` hívásra ugyanolyan értékű paraméterekkel alkalmazva.

```
1 func() ->
2   binary_to_atom(Expr1, Expr2).
```

4.8. forráskód. Sérülékeny függvény az átalakítás előtt

```
1 func() ->
2   binary_to_existing_atom(Expr1, Expr2).
```

4.9. forráskód. Sérülékeny függvény az átalakítás után

Valamint ide sorolom még a következő hasonló esetet is. A szerializációhoz használt `binary_to_term` függvénynél ajánlatos bevezetni egy plusz `[safe]` paramétert, amelyre ugyanúgy alkalmazható a fent említett transzformáció. Az átalakítás során a régi függvényt kicserélem egy újra, amely ugyanazzal a névvel és plusz egy paraméterrel rendelkezik.

$$\frac{\texttt{binary\_to\_term}(X)}{\texttt{binary\_to\_term}(X,\ [safe])}$$

A fenti részben látható formálisan az átalakítás, ahol a végén a `binary_to_term` műveletbe belekerül a plusz `[safe]` paraméter. Erre a 4.10-es és 4.11-es forráskódban mutatok példát.

```
1 deserialization() ->
2   binary_to_term(Expr1).
```

4.10. forráskód. A `binary_to_term` átalakítás előtt

```
1 deserialization() ->
2   binary_to_term(Expr1, [safe]).
```

4.11. forráskód. A `binary_to_term` átalakítás után a `[safe]` paraméterrel

4.2.3. Atomi összekapcsolás bevezetése

A folyamatok létrehozása és összekapcsolása során könnyedén versenyhelyzet alakulhat ki, ha a két művelet nem egy lépésben történik. A `spawn` és a `link` függvények külön meghívása ezt eredményezheti, így ezt a hibát is szükséges lekezelni egy transzformációval.

$$\frac{\begin{array}{c} Pid = \texttt{spawn}(Fun), \\ \texttt{link}(Pid) \end{array}}{Pid = \texttt{spawn\_link}(Fun)} \quad \text{WHEN } fun(Fun)$$

A formális példában a `spawn/1` és a `link` függvényeket szemléltetem, de a transzformáció a többi `spawn/2-4` függvényre is egyaránt alkalmazható. Alapvető megkötése, hogy a `spawn`-hoz tartozzon `link` hívás, ugyanis csak ebben az esetben hajtható végre az átalakítás.

A 4.12-es és 4.13-as példában látható kódban szemléltetem a `spawn/1` függvény transzformációs folyamatát, ahol a különálló `spawn` és `link` függvények helyett egy atomi `spawn_link` hívás kerül beszúrásra a `spawn`-al megegyező paraméterekkel.

```
1 spawn_link_func() ->
2   Pid = spawn(fun() -> Expr_body1 end),
3   link(Pid).
```

4.12. forráskód. A `spawn` és `link` függvények az átalakítás előtt

```
1 spawn_link_func() ->
2   Pid = spawn_link(fun() -> Expr_body1 end).
```

4.13. forráskód. A spawn és link függvények az átalakítás után

4.3. Implementált transzformációk bemutatása

Ebben az alfejezetben részletesen ismertetem az automatikus transzformációs megoldásaimat. Az átalakítások jelenleg csak bizonyos függvényekre alkalmazhatók, viszont a transzformációk mögötti alapelv egyszerűen kibővíthető bármely más sérülékeny függvényre. A fejezetben bemutatott folyamatok, mint például információ lekérdezése vagy új node létrehozása, mind az SPG-ben (Semantic Program Graph) hajtódnak végre.

A transzformációs műveletek meghívásához két dolgot kell megadni bemenetként: a sérülékeny függvényhívás moduljának nevét, illetve a hívás kezdetét és végét skaláris értékekként. A skaláris értékek megállapításához első körben {sor, oszlop} formátumban kell meghatározni a nem biztonságos függvény helyét, majd meg kell hívni a fájlra a 4.14-es forráskód részleten látható függvényt:

```
1 reflib_token_gen:lc2pos(file:read_file(File), {Row, Column}).
```

4.14. forráskód. Sor és oszlop páros skalárissá alakítása

A RefactorErl transzformációs keretrendszerében a transzformációknak egy előre definiált `prepare` callback függvényt kell megvalósítaniuk, amely elvégzi a szükséges ellenőrzéseket, összeszedi a megfelelő adatokat, és visszatér egy szintaxisfa transzformáló függvény sorozattal, amelyet a keretrendszer végrehajt. Ezután helyreállítja a szemantikus réteg konzisztenciáját. Tehát amikor meghívom a transzformációt, lényegében a `prepare` függvény fut le.

Az átalakítások lépéseit a következő módon általánosítom:

1. A transzformáció bemenetként megkapja a sérülékeny függvény modulját és a pozícióját skaláris értékekkel megadva.
2. A paraméterek alapján megkeresi a beadott függvényhez tartozó `application` típusú gráfcsúcsot.
3. Leellenőrzi, hogy a megadott függvényhez létezik-e implementált átalakítás.
4. Ha igen, akkor megkeresi a sérülékeny kifejezést, amelyre alkalmazni kell a transzformációt.
5. Végezetül elvégzi a függvényhívásnak megfelelő transzformációt a megtalált elemre.

Első körben ismertetem a `get_application_node_from_arg` függvényt, amelyet az általánosított lépésekben megfogalmazott első két pont végrehajtására definiáltam. Ezt minden transzformációs folyamat elején alkalmazom.

A bemeneti paramétere (`Args`) egy modulnévből és egy `tuple` típusból áll, amely a függvényhívás skaláris értékekkel megadott pozícióját tartalmazza.

A bemeneti adatok alapján lekérem a gráf megfelelő csúcsát, majd azonosítom a kapott csomópont típusát. Több típust különítek el, mivel előfordulhat, hogy a felhasználó a transzformáció meghívásakor nem a függvény teljes pozíciótartományát adja meg, hanem annak csak egy részét. Ilyen esetben nem az `application` típusú node-ot találja meg az argumentumokat feldolgozó függvény. Tehát a megtalált csúcsot típus alapján kell elkülönítenem, mivel minden esetben eltérő lekérdezésekkel jutok el a keresett pontig. A típusok a következők:

- `application`: Megtaláltam amit kerestem, így egyszerűen visszatérek az értékével.
- `infix_expr`: Lekérdezem a node szülőjét (`get_parent`), majd leellenőrzöm, hogy `application` típust kaptam-e. Ha igen, visszatérek az értékével, ellenkező esetben hibát dobok. Erre azért van szükség, hogy a beadott modul minősítővel ellátott függvény (például az `os:cmd` hívásban a modul minősítő az `os`) esetén is meg tudjam találni a függvényhívást.
- `list_comp`: Lekérdezem a node-hoz tartozó `hexpr` típusú ágat (`get_clause`), majd annak tartalmát (`get_body`). Ha az így kapott pont `application` típusú, visszatérek az értékével. Ha `infix_expr` típusú, alkalmazom rá a már ennél a típusnál említett folyamatot. Egyéb esetben hibát dobok.

- `atom`: Lekérdezem a csomópont szülőjét, majd `application` vagy `infix_expr` típustól függően végrehajtom a már említett típushoz tartozó folyamatot. Egyéb esetben hibát dobok.
- `cons`: Mivel a vizsgált érték egy lista konstruktor, ahhoz, hogy megtaláljam a lista elemét, lekérdezem a node-hoz tartozó `{cons_e, back}` ágat. Ezután leellenőrzöm, hogy `application` típust kaptam-e. Ha igen, visszatérek az értékével, ellenkező esetben hibát dobok.
- egyéb(`_`): Ebben az esetben hibával térek vissza.

A `get_application_node_from_arg` egyszerűsített változata az 1. algoritmusban látható.

1. algoritmus Beadott értékek alapján application típusú gráfcsúcs azonosítása

Function get_application_node_from_arg(*Args*)

```
1:  App ← get_expression_range(Args)
2:  switch (get_type(App))
3:      case application :
4:          App
5:      case infix_expr :
6:          handle_infix_expr_type
7:      case list_comp :
8:          handle_list_comp_type
9:      case atom :
10:         handle_atom_type
11:     case cons :
12:         handle_cons_type
13:     case _ :
14:         throw(bad_range)
15: end switch
```

4.3.1. Bemeneti paraméter validáció az os:cmd függvényekre

Az `os:cmd` minden esetben string típusú argumentummal hívható meg. Az átalakítás során egy olyan bemeneti paraméter ellenőrző műveletet szúrok be a forráskódba, amely megvizsgálja, hogy bizonyos karakterek vagy kifejezések szerepelnek-e a megadott argumentumban (például az "OS parancsok" alfejezetben említett karakterek). A transzformáció a `ri:transform_oscmd_input(...)` futtatásával történik.

A 2. algoritmusban található `prepare` műveletnek átadom paraméterül a modul nevét és a sérülékeny hívás helyét skalárisan megadva (4.14-es példa alapján), amelyre alkalmazni fogom a transzformációt (például: `oscmd_t1,{59,61}`).

A bemeneti paramétereket feldolgozom a `get_application_node_from_arg` függvény segítségével, majd az eredményül kapott `application` típusú node-ot (`App`) leellenőrzöm, hogy biztosan a `cmd` hívásra alkalmazták-e a transzformációt. Lekérdezem az `os:cmd` paraméterét (`Arg`), amely ha nem `string` típusú, akkor meghívom a `cmd_input_sanitize`-t az `App` és `Arg` paraméterekkel.

2. algoritmus Bemeneti paraméterek feldolgozása és az `os:cmd` függvény azonosítása

Function prepare($Args$)

```
1:  App ← get_application_node_from_arg(Args)
2:  Function ← get_function(App)
3:  switch (get_name(Function))
4:      case cmd :
5:          Arg ← get_first_child(get_second_child(App))
6:          switch (get_type(Arg))
7:              case string :
8:                  throw(already_safe)
9:              case _ :
10:                 cmd_input_sanitize(App, Arg)
11:         end switch
12:     case _ :
13:         throw(no_transformation)
14: end switch
```

A `cmd_input_sanitize` függvényben még az új node-ok kialakítása előtt elvégzem a szükséges lekérdezéseket, mivel közben ezekre már nincs lehetőség: lekérem a módosítandó fájl csúcsát a gráfból, illetve az `App` szülőjét, amelyre majd a node-ok kicserélésénél szükségem lesz. Továbbá leellenőrzöm, hogy létezik-e már az új beszúrandó függvény a fájlban. Ezt követően rátérek a csúcspontok létrehozására:

1. Lemásolom az `Arg` node-ot, majd elkészítem rá az új függvény alkalmazását (`case` kifejezés feje).

2. Elkészítem a `true` ágat, és hozzá illesztem az ide tartozó hibadobást.

3. Létrehozom a `false` ágat, és összekapcsolom a lemásolt `App` ponttal (eredeti függvényhívás).

4. Előállítom a `case` kifejezést az előző három lépésben elkészített pontokkal.

5. Végezetül elkészítem az új függvényhívást és a törzsét (`check_input`).

Miután létrehoztam az új csúcsokat, kicserélem az eredeti `os:cmd` hívást az újonnan előállított `case` kifejezéssel, illetve beszúrom a fájl végére az előállított új `check_input` ellenőrző függvényt, amennyiben még nem létezik (ezt még a csúcslétrehozások előtt kérdeztem le).

4.3.2. Bemeneti paraméter validáció az list_to_atom függvényekre

Az átalakítás során ebben az esetben is egy új ellenőrző műveletet szúrok be, amely korlátozza a `list_to_atom` függvény lefutásának számát, és nem engedi, hogy beteljen az atomtábla. Ez a `ri:transform_lta_input(...)` hívással valósul meg.

A megbízhatatlan elem felismerése összetettebb a többi átalakításhoz képest, ugyanis a `list_to_atom` vizsgálandó argumentuma nem mindig egyértelmű. Függ a `list_to_atom` hívástól, amelyek a következőképpen alakulhatnak:

- Vizsgált műveleten belüli list comprehension-ből származó atomgenerálás: Az ellenőrzést nem közvetlen a `list_to_atom` paraméterére kell alkalmazni, hanem a lista előállításának a forrására. A 4.15-ös példában a `lists:seq(1, 5000000)` kifejezés készíti el a listát, így a transzformáció erre alkalmazná a megkötést.
- Külső függvényhívásból származó atomgenerálás: A módosítást egy külső függvény azon részén kell végrehajtani, amely meghívja a `list_to_atom`-ot tartalmazó függvényt, és meghatározza, hogy az hányszor fusson le. A 4.16-os példában a transzformáció az `outer_fun_call`-ban az `Arg` értékre alkalmazná az ellenőrzést, mivel ez mondja meg, hogy hányszor hívódik meg a `lta_func` függvény.
- Rekurzív hívás: Ebben az esetben az ellenőrzést nem a rekurzív részen belül kell végezni, hanem egy másik részben, ahol a rekurzív függvény meghívásra kerül. A 4.17-es példa esetén a transzformáció az `outer_call`-ban alkalmazná

a megkötést a `lists:seq(1, 5000000)` részre, mivel a rekurzív függvény ezzel a generált listával van meghívva. A transzformáció jelenlegi implementációja nem jól kezelni a rekurzív függvényhívásokat, így a dolgozatban ezt nem fejtem ki.

```
1 list_comp_gen() ->
2     [list_to_atom(integer_to_list(X)) || X <- lists:seq(1, 5000000)
        ].
```

4.15. forráskód. Vizsgált függvényen belüli list comprehension-ből származó atomgenerálás

```
1 lta_func(X) ->
2    list_to_atom(integer_to_list(X) ++ "example").
3
4 outer_fun_call(Arg) ->
5    [lta_func(X) || X <- Arg].
```

4.16. forráskód. Külső függvényhívásból származó atomgenerálás

```
1 lta_recursive([X | Xs]) ->
2    list_to_atom(integer_to_list(X)),
3    lta_recursive(Xs);
4 lta_recursive([]) ->
5     ok.
6
7 outer_call() ->
8   lta_recursive(lists:seq(1, 5000000)).
```

4.17. forráskód. Sérülékeny rekurzív függvényhívás

A transzformáció elindításakor a `prepare` függvény legelején feldolgozom a bemeneti paramétereket a `get_application_node_from_arg` segítségével, így megkapom a `list_to_atom` híváshoz tartozó `application` típusú node-ot (`LtaApp`). Leellenőrzöm, hogy a megfelelő függvényre alkalmazom-e a transzformációt, majd lekérem az `LtaApp` argumentumát (`Arg`). Ezután ebből az `Arg` paraméterből indulok ki, amelyet a típusától függően eltérő módokon vizsgálok.

A transzformáció során a lekérdezéseknek két fő célja van. Megtalálni azt a kifejezést, amelyre alkalmazni fogom az ellenőrző függvényt. Vagyis azt a megbízhatatlan

listát, amely meghatározza a létrehozandó atomok számát. Valamint megkeresni azt a sérülékeny részt, amelyet a `case` kifejezés `true` ágába teszek bele. Tehát amit nem engedek lefuttatni addig, amíg nem győződtem meg arról, hogy biztonságos környezetben fogom végrehajtani.

A lehetséges `list_to_atom` hívások alapján a következő implementált csoportokra bontottam a keresést:

Függvényen belüli list comprehension

Ebben az esetben mindig arra törekszem, hogy megtaláljam a lista generálásának a forrását. Ez jöhet külső paraméterből, függvényből, vagy lehet akár konkrét érték is. A lényegén nem változtat, hogy egy lista, amin végig iterál a `list_to_atom` függvényhívás. A `case` kifejezés `true` ágába ebben az esetben a teljes list comprehensiont teszem bele, vagyis a szögletes zárójelek közötti részt.

Például, ha a 4.15-ös kódra ráfuttatom a transzformációt, a 4.18-as kódon található eredményt kapom. Látható, hogy a `size_check` ellenőrző függvényt a `lists:seq(1, 5000000)` kifejezésre alkalmaztam, és a teljes list comprehension belekerült a `case` kifejezéshez `true` ágába.

```
1 list_comp_gen() ->
2     case size_check(lists:seq(1, 5000000)) of
3         true -> [list_to_atom(integer_to_list(X)) || X <- lists:seq
              (1, 5000000)];
4         false -> throw("Variable criteria not met")
5     end.
6
7 size_check(X) -> length(X) < 5000000.
```

4.18. forráskód. Függvényen belüli list comprehension transzformáció után

Atomgenerálás külső függvényhívásból

Amikor alkalmazom a transzformációt, a `list_to_atom` függvényhívás pozícióját kell megadnom. Mivel ebben az esetben nem adott függvényen belül lesz az ellenőrizendő rész, ezért meg kell keresnem a `list_to_atom`-ot tartalmazó függvény külső hívásait. Olyan függvényt kell találnom (ha szükséges, iteratívan ismételve), ahol list comprehension segítségével hívják meg a `list_to_atom`-ot tartalmazó, vagy

ahhoz elvezető függvényt. Tehát ezen a hívási lánc mentén kell az argumentumok függőségeit végigkövetni, majd meghatározni a iterációs listát.

Ezt úgy tehetem meg, ha először megkeresem a `variable` típusú változót, ami a feltehetőleg külső értékből származik. Majd erre alkalmazom a `get_flow_deps` függvényt (3. algoritmusban szemléltetem), ahol lekérdezem a bemenet közvetlen eredetét (`Flow`) a `reflib_dataflow:flow_back` függvénnyel, majd megnézem, hogy honnan jön az értéke (`Deps`) a `reflib_dataflow:deps` hívással. Ha a `Deps` nem üres, megtaláltam az ellenőrizendő kifejezést tartalmazó függvényt, és a `Deps` érték alapján megkeresem az iterációs listát és a hozzá tartozó list comprehension kifejezést, amelyekre alkalmazható a transzformáció. (Ebben az esetben a `get_flow_deps` legelső meghívása nem adhatja vissza azt az értéket, amikor a `Deps` nem üres, mert még nem léptem ki a kiinduló függvényből.) Ha a `Deps` üres, akkor a `Flow` csúccsal haladok tovább, amely biztosan `pattern` típusú. Erre ismét alkalmazom a `reflib_dataflow:flow_back` hívást, majd lekérdezem az eredményül kapott csúcshoz tartozó `{call, back}` ágat. Ez visszaadja, hogy milyen más függvényekből[1] hívták meg a `list_to_atom`-ot tartalmazó függvényt.

Ez a folyamat iteratívan alkalmazható. Amennyiben megtaláltam a megfelelő külső függvényt, megkeresem benne az iterációs listát, és ebben a függvényben alkalmazom rá az ellenőrzést. A `case true` ágába azt a teljes kifejezést teszem bele (például list comprehension, vagy egyszerű függvényhívás), amely tartalmazza a függvényhívást (vagy függvényhívási láncot), amellyel eljutok a `list_to_atom`-ig.

3. algoritmus `list_to_atom` függvényhez `variable` típusú paraméter származásának vizsgálata

Function get_flow_deps(Arg)

1: $Flow \leftarrow reflib\_dataflow : flow\_back()$
2: $Deps \leftarrow reflib\_dataflow : deps(Flow)$
3: **switch** ($length(Deps)$)
4: **case** 1 :
5: $\{1,\ Deps\}$
6: **case** 0 :
7: $\{0,\ Flow\}$
8: **end switch**

[1] A transzformáció jelenlegi állapotában nem tudja jól kezelni azt az esetet, ha több függvényhívást is talál, így ez egyelőre nem elérhető. Tehát csak az az eset értelmezett, amikor kereken egy `{call, back}` ágat találok.

Például lefuttatva az átalakítást a 4.16-os kódon, a 4.19-es példán látható kódot kapnám eredményül. Megfigyelhető, hogy a `list_to_atom` hívást tartalmazó `lta_func` függvény nem változott, viszont az azt meghívó `outer_fun_call` igen. Ebbe került bele az ellenőrző függvény alkalmazva a lista forrására (`Arg`), és a `case` kifejezés `true` ágába a list comprehension.

```
1  lta_func(X) ->
2     list_to_atom(integer_to_list(X) ++ "example").
3
4  outer_fun_call(Arg) ->
5      case size_check(Arg) of
6          true -> [lta_func(X) || X <- Arg];
7          false -> throw("Variable criteria not met")
8      end.
9
10 size_check(X) -> length(X) < 5000000.
```

4.19. forráskód. Külső függvényhívásból származó atomgenerálás a transzformáció után

Amennyiben a fentiek alapján megtaláltam az iterációs lista kifejezést (`UntrustedArg`), meghívom rá a `list_to_atom_sanitize` függvényt az ellenőrizendő kifejezéshez tartozó `application` típusú node-al (`App`), és magával `UntrustedArg`-al. Az új csúcspontok létrehozása előtt lekérem a módosítandó fájl node-ját a gráfból és az `App` szülőjét, továbbá leellenőrzöm, hogy létezik-e már az ellenőrző függvény, amit be szeretnék illeszteni a fájlba. Az új node-ok kialakítása lényegében megegyezik az `os:cmd` részben kifejtett lépésekkel:

1. Lemásolom az `UntrustedArg` node-ot, majd elkészítem vele a `case` kifejezés fejét.
2. Létrehozom a `true` ágat, és összeillesztem a lemásolt `App` ponttal, amely az eredetileg sebezhetőnek vélt kifejezés (például list comprehension).
3. Elkészítem a `false` ágat, és beillesztem hozzá az ide tartozó hibadobást.
4. Előállítom a `case` kifejezést az előző három pontban létrehozott node-okkal.
5. Végezetül elkészítem az új függvényhívást és a törzsét (`size_check`).

Ezután kicserélem az eredeti `list_to_atom` hívást az újonnan előállított `case` kifejezéssel, valamint beillesztem a fájl végére az előállított új `size_check` ellenőrző függvényt, ha nem létezne.

Jelenleg az implementáció most csak a listagenerátoros iterációkat kezeli, de a bemutatott módszer könnyedén átírható magasabbrendű függvények kezelésére. Mint például a 4.20-as kódban, ha a list comprehension helyén egy `lists:map` vagy más ismert magasabbrendű listafüggvény állna. A `size_check` függvény argumentuma ebben az esetben a magasabbrendű hívás lista paramétere lesz, a `case` kifejezés `true` ágába pedig a magasabbrendű függvényhívás kerül. A transzformáció lefutása után 4.21-es példában szemléltetett kód jönne létre.

```
1 high_order_atom_gen() ->
2    lists:map(fun(X) -> list_to_atom(integer_to_list(X)) end, lists:
        seq(1, 5000000)).
```

4.20. forráskód. Atomgenerálás magsabbrendű függvénnyel

```
1 high_order_atom_gen() ->
2     case size_check(lists:seq(1, 5000000)) of
3         true -> lists:map(fun(X) -> list_to_atom(integer_to_list(X)
              ) end, lists:seq(1, 5000000));
4         false -> throw("Variable criteria not met")
5     end.
6
7 size_check(X) -> length(X) < 5000000.
```

4.21. forráskód. Atomgenerálás magsabbrendű függvénnyel transzformáció után

4.3.3. Függvények cseréje

Az alábbiak mentén bemutatom a sérülékeny függvények biztonságos formára való transzformálásának működését, amely a `ri:transform_atom_exh(...)` meghívásával hajtható végre. Jelenleg a következő függvényátalakításokat valósítottam meg:

- `binary_to_atom` kicserélése `binary_to_existing_atom`-ra
- `binary_to_term` hívásba a `safe` opció bevezetése
- `list_to_atom` átalakítása `list_to_existing_atom`-ra

A felsorolt transzformációk működésének alapja nagyon hasonló, ezért csak a `binary_to_atom` hívás `binary_to_existing_atom` függvényre cserélését fogom bemutatni.

A `prepare` művelet bemeneti adatait feldolgozom és lekérem belőlük az `application` típusú pontot (`App`) a `get_application_node_from_arg` függvény segítségével. Lekérdezem az `App`-hoz tartozó függvény nevét, ami alapján eldöntöm, hogy melyik transzformációt kell végrehajtani.

Tegyük fel, hogy az `App` függvény neve `binary_to_atom`, tehát ebben az esetben meghívom a `transform_binary_to_existing_atom`-ot az `App` paraméterrel, amelyet a 4. algoritmusban szemléltetek.

A `transform_binary_to_existing_atom` függvény elején lekérem az `App` szülőjét (`AppParent`) és argumentumát (`Arg`), majd létrehozom az új csomópontokat a `create` illetve a `construct` műveletek segítségével. Lemásolom az `Arg` értékét, létrehozom az új `binary_to_existing_atom` függvényt mint `atom` típust, majd ezeket összeillesztem egy `application` típusban. Végezetül a `replace` segítségével kicserélem a régi függvény `application` típusú node-ját az újonnan előállítottra, majd jelzem a keretrendszernek, hogy a változtatásokat el kell menteni a `transform_touch` hívással.

4. algoritmus A `binary_to_atom` függvény transzformációja `binary_to_existing_atom` függvényre

Function transform_binary_to_existing_atom(*App*)

1: $[\{\_,\ AppParent\}] \leftarrow get\_parent(App)$
2: $Args \leftarrow get\_second\_child(App)$
3: **fun() →**
4: $\quad \{\_,\ NewArgs\} \leftarrow lists\_keyfind(Args,\ 1,\ copy\_of(Args))$
5: $\quad NewAtom \leftarrow construct(atom,\ binary\_to\_existing\_atom)$
6: $\quad NewApp \leftarrow create(application,\ [NewAtom,\ NewArgs])$
7: $\quad replace(AppParent,\ App,\ NewApp)$
8: $\quad transform\_touch(NewArgs)$
9: **end**

4.3.4. Atomi összekapcsolás

A következőkben ismertetem azt a transzformációs folyamatot, amellyel a `spawn` és a `link` műveletekből egy `spawn_link` hívást készítek a `spawn`-al megegyező pa-

raméterekkel. Az átalakítás a `ri:transform_spawn_link_err_handle(...)` alkalmazásával érhető el.

A `prepare` legelején meghívom a `get_application_node_from_arg` függvényt a megadott bemeneti paraméterrel, amelyből megkapom az átalakítandó hívás `application` típusú node-ját (`App`). Leellenőrzöm, hogy biztosan a `spawn` hívásra futtassam le a transzformációt, majd meghívom az 5. algoritmusban is látható `transform_spawn_to_spawn_link` függvényt az `App` paraméterrel.

A `transform_spawn_to_spawn_link` elején lekérem az `App` szülőjét (`SpawnAppParent`) és argumentumát (`SpawnArgs`), majd megkeresem a `spawn`-hoz tartozó `link` node-ját, ha az létezik. Ehhez a `reflib_dataflow:reach` függvényt használom, amelynek az `App` a bemeneti paramétere. A kapott lista elemeire kétszer egymás után alkalmazom a `parent` függvényt, így megkapom, hogy mivel van összekapcsolva a `spawn` hívás. Ha a lista nem üres, és a megmaradt node (`LinkApp`) tényleg a `link` függvény, akkor alkalmazható a transzformáció. Mivel már megbizonyosodtam a `link` létezéséről, ennek a `LinkApp` pontnak is lekérdezem a szülőjét (`LinkAppParent`).

Először lemásolom a `SpawnArgs` node-ját, létrehozom a `spawn_link` függvény atomját, majd ezeket összeillesztem egy `application` típusú pontban (`SpawnLinkApp`). Így megkapom a `spawn_link` függvényt a `spawn` paramétereivel (lemásolt, de az értékük ugyanaz). Kicserélem a régi `App`-ot az új `SpawnLinkApp`-ra, valamint törlöm a régi `LinkApp`-ot (üresre cserélem), és végül elmentem a változtatásokat.

5. algoritmus A `spawn` és `link` függvények átalakítása

Function transform_spawn_to_spawn_link(*App*)

1: $[\{\_,\ SpawnAppParent\}] \leftarrow get\_parent(App)$
2: $ReachList \leftarrow get\_dataflow\_reaches(App)$
3: $ParentsList \leftarrow lists\_map(\textbf{fun(E)} \rightarrow get\_parent(get\_parent(E))\ \textbf{end},\ ReachList)$
4: $LinkApp \leftarrow lists\_filter(\textbf{fun(E)} \rightarrow E \neq []\ \textbf{end},\ ParentsList)$
5: **if** $LinkApp = []$ **then**
6: $\quad throw(link\_fun\_not\_found)$
7: **end if**
8: $LinkFunction \leftarrow get\_function(LinkApp)$
9: **if** $get\_name(LinkFunction) \neq link$ **then**
10: $\quad throw(link\_fun\_not\_found)$
11: **end if**
12: $SpawnArgs \leftarrow get\_second\_child(App)$
13: $[\{\_,\ LinkAppParent\}] \leftarrow get\_parent(LinkApp)$
14: **fun() →**
15: $\quad \{\_,\ SpawnLinkArgs\} \leftarrow lists\_keyfind(SpawnArgs,\ 1,\ copy\_of(SpawnArgs))$
16: $\quad SpawnLinkAtom \leftarrow construct(atom,\ spawn\_link)$
17: $\quad SpawnLinkApp \leftarrow create(application,\ [SpawnLinkAtom,\ SpawnLinkArgs])$
18: $\quad replace(SpawnAppParent,\ App,\ SpawnLinkApp)$
19: $\quad replace(LinkAppParent,\ LinkApp,\ [])$
20: $\quad transform\_touch(SpawnLinkArgs)$
21: **end**

4.4. További implementálható transzformációk

Az "Erlang gyakori sérülékenységei" alfejezetben bemutattam a nyelv legjellemzőbb sebezhetőségeit. Az ott említett esetek a "Transzformációs módszerek" alfejezet elején ismertetett három fő csoport valamelyikébe besorolhatók. Mivel az egy kategóriába tartozó hívások transzformációs lépései nagyon hasonlóak, így az általam implementált módszerek bármelyik sérülékeny függvényre könnyen átalakíthatók.

A következőkben néhány megvalósítható transzformációt szemléltetek formális leírásokkal.

Porthasználat

Portok nyitása az `open_port/2` függvény segítségével történik. A hívás első paramétere egy `tuple`, amely egy atomból és a futtatandó program nevéből áll. Második paramétere egy lista, amely a futás beállításait tartalmazza.

Amennyiben a paraméterei felhasználói bemenetről érkeznek, úgy szükséges ellenőrizni a program elérhetőségét és tartalmát, illetve a beállításlistát. A bemeneti adat validációs módszerrel egy ellenőrző függvényt kell beilleszteni a fájlba, és alkalmazni az ismeretlen bemeneti argumentumokra még az `open_port` végrehajtása előtt. Ez a validáció dönti el, hogy a megadott paraméterekkel biztonságosan meghívható-e a függvény.

$$\frac{\texttt{open\_port}(Name, Sett)}{\textbf{case}\ G(Name, Sett)\ \textbf{of}\ \{true\ \rightarrow H(),\ false\ \rightarrow\ \texttt{open\_port}(Name, Sett)\}\ \textbf{end}}$$

Az `open_port` függvény transzformációja során beszúrásra kerül egy új ellenőrző függvény (`G`), amely megvizsgálja a beérkezett argumentumokat. Amennyiben a program nem biztonságos forrásból származik, vagy ha a paraméterek bármelyike nem megengedett részeket tartalmaz (a validációs függvény `true` értékkel tér vissza), a program nem engedi végrehajtani a portnyitást. Ha viszont nem szerepel benne veszélyes rész, úgy biztonságosan végrehajtható az eredeti függvényhívás.

Az `open_port` hívásra megadható egy másik biztonságosabbá tevő átalakítás is, amelyet a függvénycserés módszerbe sorolható. Ha a függvény első paraméterében `spawn` szerepel, érdemes helyette a `spawn_executable` értéket használni. A transzformáció alatt a paramétereket is át kell alakítani, mivel a `spawn`-hoz tartozó rész a futtatandó programot tartalmazza, de a `spawn_executable` mellé egy futtatható program kerül és a futtatandó parancsokat az `open_port` második paraméterében megadva a következő módon várja: `{args, Args}`.

$$\frac{\texttt{open\_port}(\{spawn, Command\},\ [])}{\texttt{open\_port}(\{spawn\_executable, Filename\},\ [\{args, Args\}])}$$

Fájlokkal kapcsolatos műveletek

A `file:consult/1` függvény egy paraméterül kapott fájlnévből beolvassa a benne szereplő adatokat. Sikeres betöltés esetén visszatér egy Erlang termmel.

A függvény biztonságos használatához fájlbeolvasás előtt érdemes leellenőrizni az ismeretlen fájl méretét. Ha a fájl túl nagy, akkor túl sok atomot tartalmazhat, ami az atomtábla kimerüléséhez vezethet. Erre a bemeneti adat validációs módszer alkalmazható, amely beszúrja a fájlba az ellenőrző függvényt és alkalmazza a bemenetre.

$$\frac{\texttt{file:consult}(X)}{\textbf{case}\ G(X)\ \textbf{of}\ \{true\ \rightarrow \texttt{file:consult}(X),\ false\ \rightarrow\ H()\}\ \textbf{end}}$$

Átalakítás után a `G` ellenőrző függvény bekerül a kódba, amelyet alkalmazok a `file:consult` argumentumára. Ha a fájl mérete a meghatározott értéken belül van (`true` értéket ad vissza a `G`), akkor végrehajtom az eredeti fájlbeolvasást. Ha viszont meghaladja a maximális értéket, hibára fog futni.

Az os:cmd hívás open_port függvényre cserélése

Ahogy már a "Nem megbízható függvények cseréje" részen belül a módosuló paraméterek felsorolásában említettem, az `os:cmd` bemeneti string paraméterét fel kell bontani a futtatható állományra és az argumentumok listájára. Az így keletkezett adatokat kell beilleszteni a függvénycserés transzformáció során az `open_port` hívásba.

Abban az esetben, ha az `os:cmd` függvényt két paraméterrel hívják meg (`os:cmd(Command, Options)`), akkor a `Command` argumentumot ugyanúgy fel kell bontani, és az `Options` paraméter elhagyható. Az egyszerűség kedvéért a formális leírásban `os:cmd/1`-et használom.

$$\frac{\texttt{os:cmd}(Command)}{\texttt{open\_port}(\{spawn\_executable, FileName\}, [\{args, Args\}])}$$

A spawn és monitor függvények atomi összekapcsolása

A `spawn` és `monitor` hívások külön alkalmazása versenyhelyzetet idézhet elő, ha a `spawn` még a `monitor` futása előtt terminál.

Hasonlóan, mint a `spawn` és a `link` függvények esetében, itt is az atomi összekapcsolásos módszerre lesz szükség, amely a két függvényhívást egy `spawn_monitor` függvénnyé alakítja.

A `spawn` és a `spawn_monitor` argumentumai megegyeznek, így a transzformáció során elég lemásolni a `spawn` paraméterét, és azt belehelyezni a `spawn_monitor`-ba, majd törölni a `monitor` hívást. Viszont a függvény visszatérési értéke változik, ezért az eredeti `Pid` értéket ki kell cserélni egy `tuple` típusra.

$$\frac{\begin{array}{c} Pid = \texttt{spawn}(Fun), \\ Ref = \texttt{monitor}(process,\ Pid) \end{array}}{\{Pid,\ Ref\} = \texttt{spawn\_monitor}(Fun)} \quad \text{WHEN } fun(Fun)$$

5. fejezet

A transzformációk futási eredményei

A transzformációk működésének teszteléséhez több, GitHub-on elérhető nyílt forrású Erlang projektet választottam ki: Miner [37], Lager [38], Mochiweb [39], Erlware Commons [40] és MongooseIM [41]. Ezek között vannak kisebb méretű, néhány modulból álló alkalmazások, valamint nagyobb, összetettebb rendszerek is.

Minden transzformációs esetnél bemutatok és elmagyarázok egy példát valamelyik vizsgált projektből, amely szemlélteti az átalakítás működését. A futtatások további eredményei az A. függelékben olvashatók.

A következő általános lépéseket végeztem el a tesztelés során:

1. Megkeresem a sérülékeny függvényhívásokat a 3.1-es táblázat lekérdezései segítségével.
2. Az eredményül kapott hívások kezdetének és végének a pozícióját átalakítom skaláris formára a 4.14 példában megtekinthető függvényhívással.
3. Lefuttatom a megfelelő transzformációs műveletet a kapott értékekkel.
4. Leellenőrzöm az átalakítás eredményét.

5.1. Bemeneti paraméter validációs transzformációk

os:cmd függvények

Az 5.1-es táblázatban látható a sebezhető `os:cmd` függvényekre futtatott transzformációk eredménye alkalmazásokra bontva: a második oszlopban a sérülékenységek

száma olvasható, a harmadikban pedig az átalakítás ideje milliszekundumban megadva. Az `unsecure_os_call` függvény a Miner, Mochiweb és az Erlware Commons projektekben talált ilyen hívásokat. A többiben biztonságosan alkalmazzák ezt a függvényt, így ott nem volt szükség a forráskód módosítására.

Alkalmazás neve	**Felismert sérülékenységek száma**	**Átlagos futási idő**
Miner	*5*	*28670 ms*
Lager	-	-
Mochiweb	*1*	*4021 ms*
Erlware Commons	*2*	*1765 ms*
MongooseIM	-	-

5.1. táblázat. Input validációs transzformáció eredményei az `os:cmd` hívásra

Az 5.1-es kód a Mochiweb projekt `mochiweb_util.erl` fájljából vett 122-123. sorokat jeleníti meg a transzformáció előtti állapotban. A példán az `os:cmd` argumentuma egy megbízhatatlan külső paraméterből származik.

```
1 cmd(Argv) ->
2     os:cmd(cmd_string(Argv)).
```

5.1. forráskód. Mochiweb kódrész az `os:cmd` transzformáció előtt

Az `os:cmd` hívás a 123. sorban található, megadom az elejét és végét {sor, oszlop} formátumban ({123, 8}, {123, 11}), majd átalakítom skaláris értékekre. A modul nevével és a skaláris értékekkel alkalmazom a transzformációt, amely az 5.2-es forráskód első sorában olvasható. Az átalakítást követően a második és harmadik sorban visszajelzést kapok arról, hogy pontosan melyik fájl módosult, és hogy minden rendben volt-e a folyamat során.

```
ri:transform_oscmd_input(mochiweb_util,{3925,3928}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/mochiweb/src/mochiweb_util.erl
ok
```

5.2. forráskód. `os:cmd` transzformáció futása a Mochiweb projekten

Az 5.3-as kód a transzformáció utáni állapotot szemlélteti. A példán jól látható, hogy az eredeti hívás argumentumára (`cmd_string(Argv)`) alkalmaztam az újonnan hozzáadott `check_input` validáló függvényt, amely a fájl végére lett beszúrva. Amiatt alkalmaztam az ellenőrzést a teljes paraméterére, mivel kiszámíthatatlan, hogy egy másik függvény (mint ebben a példában a `cmd_string`) hogyan módosítja az eredeti bemenetet.

```
cmd(Argv) ->
    case check_input(cmd_string(Argv)) of
        true -> throw("Variable criteria not met");
        false -> os:cmd(cmd_string(Argv))
    end.
...
check_input(X) ->
    lists:any(fun(E) -> string:find(X, E) =/= nomatch end,
        [";", "&&", "|"]).
```

5.3. forráskód. Mochiweb kódrész az `os:cmd` transzformáció után

list_to_atom függvények

Az 5.2-es táblában olvasható a nem biztonságos `list_to_atom` függvényekre alkalmazott transzformációk eredménye. Ezeket a hívásokat az `unstable_call` függvény segítségével kaptam meg. A táblázat első sorában a vizsgált projektek, a második sorában a felismert sebezhető `list_to_atom` hívások száma, a harmadikban pedig az átlagos futási idő olvasható milliszekundumban.

Alkalmazás neve	Felismert sérülékenységek száma	Átlagos futási idő
Miner	*2*	*28751 ms*
Lager	*3*	*2182 ms*
Mochiweb	*2*	*2125 ms*
Erlware Commons	-	-
MongooseIM	*29*	*54201 ms*

5.2. táblázat. Input validációs transzformáció eredményei a `list_to_atom` hívásra

Az 5.4-es kódrészlet a Miner projekt `poc/miner_poc_grpc_client_statem.erl` fájlból származó 678-695. sorokat szemlélteti. Összetettebb példa, viszont megfigyelhető, hogy a `list_to_atom` függvényt egy list comprehension konstrukción belül alkalmazzák. Ez azt jelenti, hogy az atomgeneráló függvényt annyiszor hívják meg, ahány elemet tartalmaz a lista generálásának forrása (`Vars` változó).

```
_ ->
    %% retrieve some config from the returned validator
    Req2 = build_config_req(FilteredKeys),
    case send_grpc_unary_req(ValIP, ValGRPCPort, Req2, 'config') of
        {ok, #gateway_config_resp_v1_pb{result = Vars},
            _Req2Details} ->
            [
                begin
                    {Name, Value} = blockchain_txn_vars_v1:from_var
                        (Var),
                    application:set_env(miner, list_to_atom(Name),
                        Value)
                end
             || #blockchain_var_v1_pb{} = Var <- Vars
            ],
            ok;
        {error, Reason, _Details} ->
            {error, Reason};
```

```
16          {error, Reason} ->
17              {error, Reason}
18      end
```

5.4. forráskód. Miner kódrész a `list_to_atom` transzformáció előtt

A 686. sorban látható `list_to_atom` hívásnak meghatározom az elejét és végét ({686,56}, {686,68}), majd átváltom skaláris értékekre. Az 5.5-ös kód első sorában található a transzformáció alkalmazása az átalakítandó modul nevével és a sérülékeny hívás skaláris értékeivel. A következő két sorban az átalakítás visszajelzése látható, miszerint sikeresen módosult a `miner_poc_grpc_client_statem.erl` fájl.

```
1 ri:transform_lta_input(miner_poc_grpc_client_statem,{27163,27175}).
2 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
      source_codes/miner/src/poc/miner_poc_grpc_client_statem.erl
3 ok
```

5.5. forráskód. `list_to_atom` transzformáció futása a Miner projekten

Az átalakítást követően kialakult kódrészlet az 5.6-os példában olvasható. Látható, hogy a transzformáció beszúrta a fájl végére a `size_check` ellenőrző függvényt. Ebben az esetben a vizsgálatot (az átalakítás előtti kódrészletnél kifejtett okokból) a lista előállításának forrására alkalmazom, vagyis a `Vars` változóra.

```
1  _ ->
2      %% retrieve some config from the returned validator
3      Req2 = build_config_req(FilteredKeys),
4      case send_grpc_unary_req(ValIP, ValGRPCPort, Req2, 'config') of
5          {ok, #gateway_config_resp_v1_pb{result = Vars},
              _Req2Details} ->
6              case size_check(Vars) of
7                  true ->
8                      [
9                      begin
10                         {Name, Value} = blockchain_txn_vars_v1:
                               from_var(
11                                   Var),
12                         application:set_env(miner, list_to_atom(
                               Name),
13                              Value)
14                     end
```

```
15                      || #blockchain_var_v1_pb{} = Var<-Vars
16                      ];
17                  false -> throw("Variable criteria not met")
18              end,
19              ok;
20          {error, Reason, _Details} ->
21              {error, Reason};
22          {error, Reason} ->
23              {error, Reason}
24      end
25 ...
26 size_check(X) -> length(X) < 5000000.
```

5.6. forráskód. Miner kódrész a `list_to_atom` transzformáció után

5.2. Sebezhető függvényhívások kicserélésén alapuló transzformációk

Az 5.3-as táblában található a kicseréléses átalakítások eredménye. A sérülékenységek megtalálására itt is az `unstable_call` függvényt futtattam a projekten, amely az én esetemben a következő releváns függvényeket adja eredményül: `binary_to_term`, `binary_to_atom` és `list_to_atom`. A második oszlopban a felderített sérülékeny függvény neve és mellette zárójelben a darabszáma látható. Az utolsó oszlopban pedig az azonos transzformációk átlagos futási ideje olvasható milliszekundumban.

Alkalmazás neve	**Sérülékeny függvények**	**Átlagos futási idő**
Miner	*list_to_atom(2),* *binary_to_atom(2),* *binary_to_term(14)*	*28960 ms,* *30738 ms,* *27149 ms*
Lager	*list_to_atom(3)*	*3288 ms*
Mochiweb	*list_to_atom(2)*	*3650 ms*
Erlware Commons	-	-

MongooseIM	*list_to_atom(29),* *binary_to_atom(18),* *binary_to_term(8)*	*48534 ms,* *48808 ms,* *47843 ms*

5.3. táblázat. A függvénycserés transzformációk eredményi

Az 5.7-es kód a Miner projekt `miner_lora.erl` fájljának 759-765. sorait mutatja az átalakítás előtt. A példában két helyen is látható `binary_to_atom` hívás, viszont most csak az elsőre fogom alkalmazni a transzformációt.

```
1 {ok, Line} ->
2     CP = binary:compile_pattern([<<$,>>, <<$\n>>]),
3     [ShortCode, Lat, Long, Country, Region, GeoZone] = binary:split
          (Line, CP, [global, trim]),
4     Rec = #country{short_code=ShortCode, lat=Lat, long=Long, name=
          Country,
5                    region=binary_to_atom(Region, utf8), geo_zone=
                        binary_to_atom(GeoZone, utf8)},
6     ets:insert(?COUNTRY_FREQ_DATA, {ShortCode, Rec}),
7     csv_rows_to_ets(F);
```

5.7. forráskód. Miner kódrész függvénycserés transzformáció előtt

A 763. sorban lévő `binary_to_atom` függvényre az 5.8-as kód első sorában olvasható módon, a megfelelő modul nevével és skaláris értékekkel futtatom a transzformációt. Miután befejeződött a folyamat, megkapom a második és a harmadik sorban a visszajelzést az átalakított fájlról és annak sikerességéről.

```
1 ri:transform_fun_replacement(miner_lora,{34887,34901}).
2 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
      source_codes/miner/src/miner_lora.erl
3 ok
```

5.8. forráskód. Függvénycserés transzformáció futása a Miner projekten

Az átalakítás utáni állapotot az 5.9-es példa szemlélteti. Látható, hogy az eredetileg `binary_to_atom` hívást a transzformáció lecserélte a `binary_to_existing_atom` függvényre. Ennél az átalakításnál a paraméterek változtatására nem volt szükség, így az új függvénybe ugyanazok a paraméterek kerültek (le lettek másolva).

```
{ok, Line} ->
    CP = binary:compile_pattern([<<$,>>, <<$\n>>]),
    [ShortCode, Lat, Long, Country, Region, GeoZone] = binary:split
        (Line, CP, [global, trim]),
    Rec = #country{short_code=ShortCode, lat=Lat, long=Long, name=
        Country,
                   region=binary_to_existing_atom(Region, utf8),
                geo_zone=binary_to_atom(GeoZone, utf8)},
    ets:insert(?COUNTRY_FREQ_DATA, {ShortCode, Rec}),
    csv_rows_to_ets(F);
```

5.9. forráskód. Miner kódrész a függvénycserés transzformáció után

5.3. Atomi összekapcsolás bevezetése

A fejezet elején felsorolt, valamint több GitHub-on elérhető nyílt forrású Erlang projekt átvizsgálása során nem találtam olyan példát, ahol a `spawn` és `link` hívások `spawn_link` függvénnyé transzformálása tesztelhető lett volna. Az elemzett forráskódokban a folyamatok létrehozása minden esetben a már biztonságos `spawn_link` függvénnyel történik.

A transzformáció működésének bemutatásához ezért egy általam készített `spl_test.erl` tesztkódon szemléltetem az átalakítást. A kiinduló kódrészlet az 5.10-es példán olvasható. Az átalakítandó rész a `run` függvény első két sorában látható.

```
run() ->
    Pid = spawn(spl_test, task, []),
    link(Pid),
    Pid ! {start, <<"test data">>},
    timer:sleep(1000),
    Pid ! stop,
    ok.

task() ->
    receive
        {start, Data} ->
            io:format("Got data: ~p~n", [Data]),
            task();
        stop ->
```

```
15              io:format("Stopping.~n"),
16              ok
17      end.
```

5.10. forráskód. Példakód hibakezeléses transzformáció előtt

A `spawn` hívásnak lekérdezem a skaláris értékeit, majd a modul nevével együtt alkalmazom a transzformációt, amely az 5.11-es példa első sorában látható.

```
1 ri:transform_spawn_link_err_handle(spl_test,{73,77}).
2 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
      spl_test.erl
3 ok
```

5.11. forráskód. Hibakezeléses transzformáció futása a példakódon

Az átalakítás után az 5.12-es kódban szemléltetett rész jön létre. A példában már nincsen különálló `spawn` és `link` függvény, csak egyetlen atomi `spawn_link` hívás. Általánosan a `spawn_link` argumentumai megegyeznek a `spawn` függvényével, tehát ebben a transzformációban a `spawn`-hoz tartozó paraméterek kerültek az új `spawn_link`-be.

```
1  run() ->
2      Pid = spawn_link(spl_test, task, []),
3      Pid ! {start, <<"test data">>},
4      timer:sleep(1000),
5      Pid ! stop,
6      ok.
7
8  task() ->
9      receive
10         {start, Data} ->
11             io:format("Got data: ~p~n", [Data]),
12             task();
13         stop ->
14             io:format("Stopping.~n"),
15             ok
16     end.
```

5.12. forráskód. Példakód a hibakezeléses transzformáció után

6. fejezet

Kapcsolódó munkák

Mostanra már számos statikus elemző eszköz készült különféle programozási nyelvekre specializálva, amelyek segítenek felismerni a sérülékeny kódrészeket, mint például a Clang Static Analyzer [19], a SpotBugs [23], a CodeChecker [21], a CheckMarx [17] vagy a SonarQube [15].

Specifikusan az Erlang nyelvhez készült sebezhető hívásokat felismerő eszközök közül a RefactoErl mellett a PEST (Primitive Erlang Security Tool) [42] is képes hasonló vizsgálatokat végezni. A különbség a két rendszer között az, hogy különböző módszertanokat használnak [6]. A RefactorErl egy szemantikus programgráf segítségével végzi az elemzéseket, a PEST pedig egyszerű szöveges keresésen alapul.

A statikus elemző programok a hibás, sebezhető kódrészek felismerésére fókuszálnak, és nem mindegyik ad tanácsot azok javítására, vagy ha igen, akkor manuális javítást igényélnek. Például a Clang Static Analyzer és a SpotBugs eszközök csak megvizsgálják a kódot, és kiírják a sérülékenységeket. A CodeChecker, a CheckMarx és a SonarQube eszközök pedig már konkrét ötletekkel is ellátnak, hogy a rendszer biztonságosabb és hibamentes legyen. Emiatt jöttek létre az automatikus kódtranszformáló eszközök, amelyek nemcsak azonosítják és tanácsot adnak, hanem módosítják is a problémás kódrészeket. A következőkben néhány ilyen alkalmazást és transzformálási módszert mutatok be, és összevetem a diplomamunkámban megvalósítottakkal.

Wrangler

A Wrangler [43, 44] egy nyílt forráskódú eszköz, amelyet kifejezetten Erlang nyelvhez fejlesztett ki a Kenti Egyetem. Beépített szabályok alapján különböző

strukturális transzformációkat tud végrehajtani, mint például a függvények átnevezését, a kódduplikáció megszüntetését vagy a függvények összevonását. Több fejlesztői környezetbe is integrálható (például Emacs vagy Eclipse), valamint parancssorból is használható.

A program célja a kód minőségének és karbantarthatóságának javítása. Ezzel szemben az általam megvalósított megoldás elsősorban a biztonsági kockázatok kezelésére fókuszál.

GenProg

A GenProg [45] generikus programozást alkalmazva képes a programhibák automatikus javítására. A rendszer a meglévő tesztekre támaszkodik úgy, hogy a pozitív tesztesetek a kívánt működést, míg a negatív tesztesetek a hibás viselkedést definiálják. Az algoritmus a program absztrakt szintaxisfájában (AST) dolgozik, és ott hajtja végre a transzformációkat és a különböző műveleteket.

Ezt a módszert a [46] cikk szerint tizenhat különböző C nyelven írt szoftveren tesztelték (összesen 1,25 millió sornyi kód), amelynek a sikerességi aránya átlagosan 77%. Nyolcféle hibatípust javítottak ki, amelyek a következők:

- végtelen ciklus
- szegmentálási hiba
- távoli heap buffer túlcsordulás beszúráshoz (injection)
- távoli heap buffer túlcsordulás változók felülírásához
- nem túlcsordulásos szolgáltatásmegtagadás
- lokális stack buffer túlcsordulás
- szám túlcsordulás
- string sebezhetőség

Az eszköz genetikus algoritmusokon alapuló módszert használ, főleg az általános programhibák javítására, míg az én munkámban szemantikus programgráfon alapuló transzformációkat alkalmazok, a sérülékeny részekre koncentrálva.

FixMiner

A FixMiner [47] egy automatikus mintabányászati módszer, amely nyílt forráskódú projektek hibajavítási commitjaiból azonosítja az ismétlődő javítási mintákat. A változtatásokat több szinten is elemzi és csoportosítja, majd ezekből általánosított

sablonokat hoz létre. Az előállított mintákat automatizált programjavító rendszerek (APR) bemeneteként lehet használni.

A módszer tesztelésére megalkották a PARFixMiner-t [48], ami egy APR prototípus. A FixMinerrel több ezer nyílt forráskódú projektet kiértékeltek, majd az elkészült mintákat beintegrálták a PARFixMiner eszközbe. A sablonokkal a Defects4J benchmark [49] 26 hibáját sikeresen kijavították, illetve a generált javítások 81%-a átment a tesztkészleten.

Mivel a FixMiner módszere a commitokból megtanult mintákra épül, ezért előfordulhat, hogy a javítások eredményei nem determinisztikusak lesznek. Az általam implementált szabályalapú módszer viszont mindig megbízható és előre megjósolható eredményt ad.

7. fejezet

Összegzés

Az informatika rohamos fejlődésének hatására egyre több folyamat kerül szoftveres irányítás alá, amely jelentősen megkönnyíti az emberek mindennapi életét és a különböző iparágak működését. Ezzel párhuzamosan a kibertámadások száma és súlyossága is folyamatosan növekszik, így a forráskódok biztonsága napjainkra már kiemelt fontosságúvá vált.

A szoftverfejlesztés során a programozók nem tudnak minden potenciális sérülékenységet előre lefedni, így számos ilyen hiba csak utólag derül ki. Ezek manuális javítása rendkívül időigényes, és nagyobb eséllyel keletkezhetnek újabb hibák. Ebből kifolyólag elengedhetetlen, hogy a sebezhető részek időben felismerhetők és automatikusan kezelhetők legyenek, különösen a funkcionális nyelvekben, mint az Erlang.

Így diplomamunkám célja az volt, hogy az Erlang nyelvben írt forráskódokban a RefactorErl eszköz által felismert sérülékenységekre olyan automatikus transzformációkat definiáljak, amelyek segítségével biztonságosabbá tudom tenni a sebezhető részeket.

A munkám elején áttekintést adtam a forráskód biztonságának elméleti alapjairól, majd bemutattam az Erlang nyelvre jellemző sérülékenységeket. Röviden ismertettem a statikus elemző szoftvereket, amelyek közül külön kiemeltem a RefactorErl eszközt. Emellett összefoglaltam azokat a függvényeket, amelyek segítségével azonosíthatók a különböző sebezhető kódrészek egy projektben.

Példákon keresztül szemléltettem a téma jelentőségét, majd a transzformációs módszereket a következő három nagy csoportra bontva mutattam be: bemeneti adat validációjának beszúrása, nem megbízható függvények cseréje, illetve atomi összekapcsolás bevezetése. Ezeket formális leírásokkal és algoritmusokkal is szemléltettem.

Részletesen kifejtettem az általam implementált transzformációkat, majd további lehetséges átalakításokat fogalmaztam meg. Végezetül több projektre is lefuttattam a transzformációkat, melyek eredményeit táblázatokban foglaltam össze.

7.1. Továbbfejlesztési lehetőségek

A munkámban szemléltetett átalakítási módszerek főként a transzformációk alapelvét és működési módját mutatják be, tehát a téma még számos bővítési és továbbfejlesztési lehetőséget tartogat magában.

További transzformációk implementálhatók, például azokat az átalakításokat, amelyeket a "További implementálható transzformációk" alfejezetben kifejtettem. A `list_to_atom` transzformáció tovább okosítható, hogy a rekurzív hívásokat, magasabbrendű atomgenerálást és a több függvényhívásos eseteket is tudja kezelni és átalakítani. A bemeneti adat validáció átalakításai módosíthatók úgy, hogy a transzformáció meghívásakor a felhasználó által megadott feltétel kerüljön be az ellenőrző függvény törzsébe. Valamint kibővíthető a meglévő összes transzformáció úgy, hogy {sor, oszlop} formátummal is meglehessen hívni azokat, ne kelljen külön skaláris értékekre átváltani.

Ezen felül megvalósítható egy olyan bővítmény is bizonyos fejlesztői környezetekhez, amely gombnyomásra képes végrehajtani ezeket a transzformációkat a forráskódon a környezeten belül. A plugin felismerné a sebezhető részeket, majd a felhasználói jóváhagyás után automatikusan biztonságos változatra alakítaná át azokat.

A. függelék

Transzformációk futási eredményei

A következő mellékletben a tesztelt projektek futási eredményeit foglalom össze transzformációkra bontva. A vizsgált projektek a következők: Miner [37], Lager [38], Mochiweb [39], Erlware Commons [40] és MongooseIM [41].

A részfejezetekben szereplő első példakódban az átalakítás előtti állapotot mutatom be, feltüntetve a fájl nevét és a releváns sorok sorszámát az első sorban. A második példakód az alkalmazott transzformációk futását szemlélteti, fájlok szerint elkülönítve. Végezetül a harmadik példában az átalakítást követő állapot látható.

Amennyiben egy projekten belül egy transzformáció nagyobb számban fordul elő, úgy abból a típusból öt példát mutatok be a függelékben.

A.1. Miner

A.1.1. os:cmd bemenet ellenőrző transzformációk

```
1 %--- cli/mochiweb_util.erl | 431-448. sor ---
2 get_log_errors(ErrorCount, ScanRange)->
3     {ok, BaseDir} = application:get_env(lager, log_root),
4     LogPath = lists:concat([BaseDir, "/console.log"]),
5
6     TxnErrors = os:cmd(          "tail -n" ++ ScanRange ++ " " ++
          LogPath ++ " | sort -rn |"
7                                  "grep -m" ++ ErrorCount ++ " -E \"
                                       error.*blockchain_txn|
                                       blockchain_txn.*error\""
8                                  ),
9
```

```
10      POCErrors = os:cmd(          "tail -n" ++ ScanRange ++ " " ++
            LogPath ++ " | sort -rn |"
11                                   "grep -m" ++ ErrorCount ++ " -E \"
                                        error.*miner_poc|miner_poc.*
                                        error|error.*blockchain_poc|
                                        blockchain_poc.*error\""
12                                   ),
13
14      GenErrors = os:cmd(          "tail -n" ++ ScanRange  ++ " " ++
            LogPath ++ " | sort -rn |"
15                                   "grep -m" ++ ErrorCount ++ " -E \"
                                        error\" | "
16                                   "grep -Ev \"blockchain_txn|
                                        miner_poc|blockchain_poc|
                                        absorbing\""
17                                   ),
18
19      {[POCErrors], [TxnErrors], [GenErrors]}.
20
21  %--- miner_gateway_port.erl | 145-152. sor ---
22  cleanup_port(#state{port = Port} = State) ->
23      erlang:demonitor(State#state.monitor, [flush]),
24      case erlang:port_info(Port) of
25          undefined -> true;
26          _Result -> erlang:port_close(Port)
27      end,
28      os:cmd(io_lib:format("kill -9 ~p", [State#state.os_pid])),
29      ok.
30
31  %--- miner_mux_port.erl | 91-98. sor ---
32  cleanup_port(#state{port = Port} = State) ->
33      erlang:demonitor(State#state.monitor, [flush]),
34      case erlang:port_info(Port) of
35          undefined -> true;
36          _Result -> erlang:port_close(Port)
37      end,
38      os:cmd(io_lib:format("kill -9 ~p", [State#state.os_pid])),
39      ok.
```

A.1. forráskód. Miner `os:cmd` transzformáció előtt

```
%--- cli/mochiweb_util.erl ---
ri:transform_oscmd_input(miner_cli_info,{15051,15054}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/miner/src/cli/miner_cli_info.erl
ok

ri:transform_oscmd_input(miner_cli_info,{14773,14776}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/miner/src/cli/miner_cli_info.erl
ok

ri:transform_oscmd_input(miner_cli_info,{14529,14532}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/miner/src/cli/miner_cli_info.erl
ok

%--- miner_gateway_port.erl ---
ri:transform_oscmd_input(miner_gateway_port,{5024,5027}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/miner/src/miner_gateway_port.erl
ok

%--- miner_mux_port.erl ---
ri:transform_oscmd_input(miner_mux_port,{3045,3048}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/miner/src/miner_mux_port.erl
ok
```

A.2. forráskód. `os:cmd` transzformáció futása a Miner projekten

```
%--- cli/mochiweb_util.erl ---
get_log_errors(ErrorCount, ScanRange)->
    {ok, BaseDir} = application:get_env(lager, log_root),
    LogPath = lists:concat([BaseDir, "/console.log"]),

    TxnErrors = case check_input("tail -n" ++
                ScanRange ++ " " ++ LogPath ++ " | sort -rn |"
                "grep -m" ++
                ErrorCount ++
                " -E \"error.*blockchain_txn|blockchain_txn.*error
                    \""
```

```
11                  ) of
12              true -> throw("Variable criteria not met");
13              false ->
14                  os:cmd("tail -n" ++ ScanRange ++ " " ++
15                      LogPath ++ " | sort -rn |"
16                      "grep -m" ++
17                      ErrorCount ++
18                      " -E \"error.*blockchain_txn|blockchain_txn.*
                            error\""
19                      )
20          end,
21
22      POCErrors = case check_input("tail -n" ++
23                  ScanRange ++ " " ++ LogPath ++ " | sort -rn |"
24                  "grep -m" ++
25                  ErrorCount ++
26                  " -E \"error.*miner_poc|miner_poc.*error|error.*
                        blockchain_poc|blockchain_poc.*error\""
27                  ) of
28              true -> throw("Variable criteria not met");
29              false ->
30                  os:cmd("tail -n" ++ ScanRange ++ " " ++
31                      LogPath ++ " | sort -rn |"
32                      "grep -m" ++
33                      ErrorCount ++
34                      " -E \"error.*miner_poc|miner_poc.*error|error
                            .*blockchain_poc|blockchain_poc.*error\""
35                      )
36          end,
37
38      GenErrors = case check_input("tail -n" ++
39                  ScanRange ++ " " ++ LogPath ++ " | sort -rn |"
40                  "grep -m" ++ ErrorCount ++ " -E \"error\" | "
41                  "grep -Ev \"blockchain_txn|miner_poc|blockchain_poc
                        |absorbing\""
42                  ) of
43              true -> throw("Variable criteria not met");
44              false ->
45                  os:cmd("tail -n" ++ ScanRange ++ " " ++
46                      LogPath ++ " | sort -rn |"
47                      "grep -m" ++ ErrorCount ++ " -E \"error\" | "
```

```
48                      "grep -Ev \"blockchain_txn|miner_poc|
                           blockchain_poc|absorbing\""
49                      )
50          end,
51
52      {[POCErrors], [TxnErrors], [GenErrors]}.
53  ...
54  check_input(X) ->
55      lists:any(fun(E) -> string:find(X, E) =/= nomatch end,
56          [";", "&&", "|"]).
57
58  %--- miner_gateway_port.erl ---
59  cleanup_port(#state{port = Port} = State) ->
60      erlang:demonitor(State#state.monitor, [flush]),
61      case erlang:port_info(Port) of
62          undefined -> true;
63          _Result -> erlang:port_close(Port)
64      end,
65      case check_input(io_lib:format("kill -9 ~p", [State#state.
           os_pid])) of
66          true -> throw("Variable criteria not met");
67          false -> os:cmd(io_lib:format("kill -9 ~p", [State#state.
               os_pid]))
68      end,
69      ok.
70  ...
71  check_input(X) ->
72      lists:any(fun(E) -> string:find(X, E) =/= nomatch end,
73          [";", "&&", "|"]).
74
75  %--- miner_mux_port.erl ---
76  cleanup_port(#state{port = Port} = State) ->
77      erlang:demonitor(State#state.monitor, [flush]),
78      case erlang:port_info(Port) of
79          undefined -> true;
80          _Result -> erlang:port_close(Port)
81      end,
82      case check_input(io_lib:format("kill -9 ~p", [State#state.
           os_pid])) of
83          true -> throw("Variable criteria not met");
84          false -> os:cmd(io_lib:format("kill -9 ~p", [State#state.
```

```
            os_pid]))
85      end,
86      ok.
87  ...
88  check_input(X) ->
89      lists:any(fun(E) -> string:find(X, E) =/= nomatch end,
90          [";", "&&", "|"]).
```

A.3. forráskód. Miner `os:cmd` transzformáció utáni eredménye

A.1.2. Függvénycserés transzformációk

```
1   %--- miner.erl | 202-212. sor ---
2   case gen_server:call(Mod, Group, 60000) of
3       undefined -> #{};
4       Pid ->
5           try libp2p_group_relcast:queues(Pid) of
6               {_ModState, Inbound, Outbound} ->
7                   O = maps:map(
8                       fun(_, V) ->
9                           [erlang:binary_to_term(Value) || Value <- V
                                ]
10                      end,
11                      Outbound
12                  ),
13
14  %--- miner_lora.erl | 759-765. sor ---
15  {ok, Line} ->
16      CP = binary:compile_pattern([<<$,>>, <<$\n>>]),
17      [ShortCode, Lat, Long, Country, Region, GeoZone] = binary:split
            (Line, CP, [global, trim]),
18      Rec = #country{short_code=ShortCode, lat=Lat, long=Long, name=
            Country,
19                     region=binary_to_atom(Region, utf8), geo_zone=
                           binary_to_atom(GeoZone, utf8)},
20      ets:insert(?COUNTRY_FREQ_DATA, {ShortCode, Rec}),
21      csv_rows_to_ets(F);
22
23  %--- miner_restart_sup.erl | 173-177. sor ---
24  check_for_region_override(undefined)->
25      case os:getenv("REGION_OVERRIDE") of
```

```
26          false -> undefined;
27          Region -> list_to_atom(Region)
28      end;
29
30  %--- poc/miner_poc_grpc_client_statem.erl | 682-689. sor ---
31  {ok, #gateway_config_resp_v1_pb{result = Vars}, _Req2Details} ->
32      [
33          begin
34              {Name, Value} = blockchain_txn_vars_v1:from_var(Var),
35              application:set_env(miner, list_to_atom(Name), Value)
36          end
37       || #blockchain_var_v1_pb{} = Var <- Vars
38      ],
```

A.4. forráskód. Miner függvénycserés transzformáció előtt

```
1  %--- miner.erl ---
2  ri:transform_fun_replacement(miner,{6122,6136}).
3  modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/miner/src/miner.erl
4  ok
5
6  %--- miner_lora.erl ---
7  ri:transform_fun_replacement(miner_lora,{34887,34901}).
8  modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/miner/src/miner_lora.erl
9  ok
10
11 ri:transform_fun_replacement(miner_lora,{34955,34968}).
12 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/miner/src/miner_lora.erl
13 ok
14
15 %--- miner_restart_sup.erl ---
16 ri:transform_fun_replacement(miner_restart_sup,{6139,6151}).
17 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/miner/src/miner_restart_sup.erl
18 ok
19
20 %--- poc/miner_poc_grpc_client_statem.erl ---
21 ri:transform_fun_replacement(miner_poc_grpc_client_statem,{
```

```
   27163,27175}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
   source_codes/miner/src/poc/miner_poc_grpc_client_statem.erl
ok
```

A.5. forráskód. Függvénycserés transzformáció futása a Miner projekten

```
%--- miner.erl ---
case gen_server:call(Mod, Group, 60000) of
    undefined -> #{};
    Pid ->
        try libp2p_group_relcast:queues(Pid) of
            {_ModState, Inbound, Outbound} ->
                O = maps:map(
                    fun(_, V) ->
                            [erlang:binary_to_term(Value, [safe])
                               ||
                            Value <- V]
                    end,
                    Outbound
                ),

%--- miner_lora.erl ---
{ok, Line} ->
    CP = binary:compile_pattern([<<$,>>, <<$\n>>]),
    [ShortCode, Lat, Long, Country, Region, GeoZone] = binary:split
       (Line, CP, [global, trim]),
    Rec = #country{short_code=ShortCode, lat=Lat, long=Long, name=
       Country,
                   region=binary_to_existing_atom(Region, utf8),
                geo_zone=binary_to_existing_atom(GeoZone, utf8)},
    ets:insert(?COUNTRY_FREQ_DATA, {ShortCode, Rec}),
    csv_rows_to_ets(F);

%--- miner_restart_sup.erl ---
check_for_region_override(undefined)->
    case os:getenv("REGION_OVERRIDE") of
        false -> undefined;
        Region -> list_to_existing_atom(Region)
end;

```

```
%--- poc/miner_poc_grpc_client_statem.erl ---
{ok, #gateway_config_resp_v1_pb{result = Vars}, _Req2Details} ->
    [
        begin
            {Name, Value} = blockchain_txn_vars_v1:from_var(Var),
            application:set_env(miner, list_to_existing_atom(
                Name), Value)
        end
     || #blockchain_var_v1_pb{} = Var <- Vars
    ],
```

A.6. forráskód. Miner függvénycserés transzformáció után

A.1.3. list_to_atom input ellenőrző transzformációk

```
%--- miner_restart_sup.erl | 173-177. sor ---
check_for_region_override(undefined)->
    case os:getenv("REGION_OVERRIDE") of
        false -> undefined;
        Region -> list_to_atom(Region)
    end;

%--- poc/miner_poc_grpc_client_statem.erl | 678-695. sor ---
_ ->
    %% retrieve some config from the returned validator
    Req2 = build_config_req(FilteredKeys),
    case send_grpc_unary_req(ValIP, ValGRPCPort, Req2, 'config') of
        {ok, #gateway_config_resp_v1_pb{result = Vars},
            _Req2Details} ->
            [
                begin
                    {Name, Value} = blockchain_txn_vars_v1:from_var
                        (Var),
                    application:set_env(miner, list_to_atom(Name),
                        Value)
                end
             || #blockchain_var_v1_pb{} = Var <- Vars
            ],
            ok;
        {error, Reason, _Details} ->
            {error, Reason};
```

```
24          {error, Reason} ->
25              {error, Reason}
26      end
```

A.7. forráskód. Miner `list_to_atom` transzformáció előtt

```
1  %--- miner_restart_sup.erl ---
2  ri:transform_lta_input(miner_restart_sup,{6139,6151}).
3  No transformation is needed because there is no multiple atom
       generation.
4  deny
5
6  %--- poc/miner_poc_grpc_client_statem.erl ---
7  ri:transform_lta_input(miner_poc_grpc_client_statem,{27163,27175}).
8  modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/miner/src/poc/miner_poc_grpc_client_statem.erl
9  ok
```

A.8. forráskód. `list_to_atom` transzformáció futása a Miner projekten

```
1  %--- poc/miner_poc_grpc_client_statem.erl ---
2  _ ->
3      %% retrieve some config from the returned validator
4      Req2 = build_config_req(FilteredKeys),
5      case send_grpc_unary_req(ValIP, ValGRPCPort, Req2, 'config') of
6          {ok, #gateway_config_resp_v1_pb{result = Vars},
               _Req2Details} ->
7              case size_check(Vars
8                      ) of
9                  true ->
10                     [
11                     begin
12                         {Name, Value} = blockchain_txn_vars_v1:
                               from_var(
13                                   Var),
14                         application:set_env(miner, list_to_atom(
                               Name),
15                             Value)
16                     end
17                     || #blockchain_var_v1_pb{} = Var<-Vars
18                     ];
19                 false -> throw("Variable criteria not met")
```

```
20              end,
21              ok;
22          {error, Reason, _Details} ->
23              {error, Reason};
24          {error, Reason} ->
25              {error, Reason}
26      end
27  ...
28  size_check(X) -> length(X) < 5000000.
```

A.9. forráskód. Miner `list_to_atom` transzformáció után

A.2. Lager

A.2.1. Függvénycserés transzformációk

```
1   %--- lager_stdlib.erl | 485-495. sor ---
2   pp_arguments(PF, As, I) ->
3       case {As, io_lib:printable_list(As)} of
4           {[Int | T], true} ->
5               L = integer_to_list(Int),
6               Ll = length(L),
7               A = list_to_atom(lists:duplicate(Ll, $a)),
8               S0 = binary_to_list(iolist_to_binary(PF([A | T], I+1))
                    ,
9               brackets_to_parens([$[,L,string:sub_string(S0, 2+Ll)]);
10          _ ->
11              brackets_to_parens(PF(As, I+1))
12      end.
13
14  %--- lager_util.erl | 585-586. sor ---
15  make_internal_sink_name(Sink) ->
16      list_to_atom(atom_to_list(Sink) ++ "_lager_event").
17
18  %--- lager_transform.erl | 284-285. sor ---
19  make_varname(Prefix, CallAnno) ->
20      list_to_atom(Prefix ++ atom_to_list(get(module)) ++
          integer_to_list(erl_anno:line(CallAnno))).
```

A.10. forráskód. Lager függvénycserés transzformáció előtt

```
%--- lager_stdlib.erl ---
ri:transform_fun_replacement(lager_stdlib,{17135,17146}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/lager/src/lager_stdlib.erl
ok

%--- lager_util.erl ---
ri:transform_fun_replacement(lager_util,{22623,22635}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/lager/src/lager_util.erl
ok

%--- lager_transform.erl ---
ri:transform_fun_replacement(lager_transform,{13162,13174}).
modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/lager/src/lager_transform.erl
ok
```

A.11. forráskód. Függvénycserés transzformáció futása a Lager projekten

```
%--- lager_stdlib.erl ---
pp_arguments(PF, As, I) ->
    case {As, io_lib:printable_list(As)} of
        {[Int | T], true} ->
            L = integer_to_list(Int),
            Ll = length(L),
            A = list_to_existing_atom(lists:duplicate(Ll, $a)),
            S0 = binary_to_list(iolist_to_binary(PF([A | T], I+1))
                ),
            brackets_to_parens([$[,L,string:sub_string(S0, 2+Ll)]);
        _ ->
            brackets_to_parens(PF(As, I+1))
    end.

%--- lager_util.erl ---
make_internal_sink_name(Sink) ->
    list_to_existing_atom(atom_to_list(Sink) ++ "_lager_event").

%--- lager_transform.erl ---
make_varname(Prefix, CallAnno) ->
    list_to_existing_atom(Prefix ++
```

```
21          atom_to_list(get(module)) ++ integer_to_list(erl_anno:line(
                CallAnno))).
```

A.12. forráskód. Lager függvénycserés transzformáció után

A.2.2. list_to_atom input ellenőrző transzformációk

```
1  %--- lager_stdlib.erl | 485-495. sor ---
2  pp_arguments(PF, As, I) ->
3      case {As, io_lib:printable_list(As)} of
4          {[Int | T], true} ->
5              L = integer_to_list(Int),
6              Ll = length(L),
7              A = list_to_atom(lists:duplicate(Ll, $a)),
8              S0 = binary_to_list(iolist_to_binary(PF([A | T], I+1)))
                   ,
9              brackets_to_parens([$[,L,string:sub_string(S0, 2+Ll)]);
10         _ ->
11             brackets_to_parens(PF(As, I+1))
12     end.
13
14 %--- lager_util.erl | 585-586. sor ---
15 make_internal_sink_name(lager) ->
16     ?DEFAULT_SINK;
17 make_internal_sink_name(Sink) ->
18     list_to_atom(atom_to_list(Sink) ++ "_lager_event").
19
20 %--- lager_transform.erl | 284-285. sor ---
21 make_varname(Prefix, CallAnno) ->
22     list_to_atom(Prefix ++ atom_to_list(get(module)) ++
          integer_to_list(erl_anno:line(CallAnno))).
```

A.13. forráskód. Lager `list_to_atom` transzformáció előtt

```
1  %--- lager_stdlib.erl ---
2  ri:transform_lta_input(lager_stdlib,{17135,17146}).
3  No transformation for this case.
4  deny
5
6  %--- lager_util.erl ---
7  ri:transform_lta_input(lager_util,{22623,22635}).
```

```
8 No transformation is needed because there is no multiple atom
    generation.
9 deny
10
11 %--- lager_transform.erl ---
12 ri:transform_lta_input(lager_transform,{13162,13174}).
13 No transformation is needed because there is no multiple atom
    generation.
14 deny
```

A.14. forráskód. `list_to_atom` transzformáció futása a Lager projekten

A.3. Mochiweb

A.3.1. os:cmd bemenet ellenőrző transzformációk

```
1 %--- mochiweb_util.erl | 122-123. sor ---
2 cmd(Argv) ->
3     os:cmd(cmd_string(Argv)).
```

A.15. forráskód. Mochiweb `os:cmd` transzformáció előtt

```
1 %--- mochiweb_util.erl ---
2 ri:transform_oscmd_input(mochiweb_util,{3925,3928}).
3 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/mochiweb/src/mochiweb_util.erl
4 ok
```

A.16. forráskód. `os:cmd` transzformáció futása a Mochiweb projekten

```
1 %--- mochiweb_util.erl ---
2 cmd(Argv) ->
3     case check_input(cmd_string(Argv)) of
4         true -> throw("Variable criteria not met");
5         false -> os:cmd(cmd_string(Argv))
6     end.
7 ...
8 check_input(X) ->
9     lists:any(fun(E) -> string:find(X, E) =/= nomatch end,
10         [";", "&&", "|"]).
```

A.17. forráskód. Mochiweb `os:cmd` transzformáció után

A.3.2. Függvénycserés transzformációk

```
1 %--- mochiglobal.erl | 66-67. sor ---
2 key_to_module(K) ->
3     list_to_atom("mochiglobal:" ++ atom_to_list(K)).
4
5 %--- mochiweb_socket_server.erl | 89-91. sor ---
6 parse_options([{name, L} | Rest], State) when is_list(L) ->
7     Name = {local, list_to_atom(L)},
8     parse_options(Rest, State#mochiweb_socket_server{name=Name});
```

A.18. forráskód. Mochiweb függvénycserés transzformáció előtt

```
1 %--- mochiglobal.erl ---
2 ri:transform_fun_replacement(mochiglobal,{2354,2366}).
3 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/mochiweb/src/mochiglobal.erl
4 ok
5
6 %--- mochiweb_socket_server.erl ---
7 ri:transform_fun_replacement(mochiweb_socket_server,{2859,2871}).
8 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/mochiweb/src/mochiweb_socket_server.erl
9 ok
```

A.19. forráskód. Függvénycserés transzformáció futása a Mochiweb projekten

```
1 %--- mochiglobal.erl ---
2 key_to_module(K) ->
3     list_to_existing_atom("mochiglobal:" ++ atom_to_list(K)).
4
5 %--- mochiweb_socket_server.erl ---
6 parse_options([{name, L} | Rest], State) when is_list(L) ->
7     Name = {local, list_to_existing_atom(L)},
8     parse_options(Rest, State#mochiweb_socket_server{name=Name});
```

A.20. forráskód. Mochiweb függvénycserés transzformáció után

A.3.3. list_to_atom input ellenőrző transzformációk

```
1 %--- mochiglobal.erl | 66-67. sor ---
2 key_to_module(K) ->
```

```
3       list_to_atom("mochiglobal:" ++ atom_to_list(K)).
4
5   %--- mochiweb_socket_server.erl | 89-91. sor ---
6   parse_options([{name, L} | Rest], State) when is_list(L) ->
7       Name = {local, list_to_atom(L)},
8       parse_options(Rest, State#mochiweb_socket_server{name=Name});
```

A.21. forráskód. Mochiweb `list_to_atom` transzformáció előtt

```
1   %--- mochiglobal.erl ---
2   ri:transform_lta_input(mochiglobal,{2354,2366}).
3   No transformation is needed because there is no multiple atom
        generation.
4   deny
5
6   %--- mochiweb_socket_server.erl ---
7   ri:transform_lta_input(mochiweb_socket_server,{2859,2871}).
8   This is a recursive function, there is no transformation for that.
9   deny
```

A.22. forráskód. `list_to_atom` transzformáció futása a Mochiweb projekten

A.4. Erlware commons

A.4.1. os:cmd bemenet ellenőrző transzformációk

```
1   %--- ec_git_vsn.erl | 83-107. sor ---
2   get_patch_count(RawRef) ->
3       Ref = re:replace(RawRef, "\\s", "", [global]),
4       Cmd = io_lib:format("git rev-list --count ~ts..HEAD",
5                           [Ref]),
6       case os:cmd(Cmd) of
7           "fatal: " ++ _ ->
8               0;
9           Count ->
10              Count
11      end.
12
13  -spec parse_tags(t()|string()) -> {string()|undefined, ec_semver:
        version_string()}.
```

```
14 parse_tags({}) ->
15     parse_tags("");
16 parse_tags(Pattern) ->
17     Cmd = io_lib:format("git describe --abbrev=0 --tags --match \"~
           ts*\"", [Pattern]),
18     Tag = os:cmd(Cmd),
19     case Tag of
20         "fatal: " ++ _ ->
21             {undefined, ""};
22         _ ->
23             Vsn  = string:slice(Tag, string:length(Pattern)),
24             Vsn1 = string:trim(string:trim(Vsn, leading, "v"),
                   trailing, "\n"),
25             {Tag, Vsn1}
26     end.
```

A.23. forráskód. Erlware Commons `os:cmd` transzformáció előtt

```
1 %--- ec_git_vsn.erl ---
2 ri:transform_oscmd_input(ec_git_vsn,{3149,3152}).
3 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/erlware/src/ec_git_vsn.erl
4 ok
5
6 ri:transform_oscmd_input(ec_git_vsn,{2806,2809}).
7 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
    source_codes/erlware/src/ec_git_vsn.erl
8 ok
```

A.24. forráskód. `os:cmd` transzformáció futása az Erlware Commons projekten

```
1 %--- ec_git_vsn.erl ---
2 get_patch_count(RawRef) ->
3     Ref = re:replace(RawRef, "\\s", "", [global]),
4     Cmd = io_lib:format("git rev-list --count ~ts..HEAD",
5                         [Ref]),
6     case case check_input(Cmd) of
7             true -> throw("Variable criteria not met");
8             false -> os:cmd(Cmd)
9         end of
10         "fatal: " ++ _ ->
11             0;
```

```
12          Count ->
13              Count
14      end.
15
16  -spec parse_tags(t()|string()) -> {string()|undefined, ec_semver:
        version_string()}.
17  parse_tags({}) ->
18      parse_tags("");
19  parse_tags(Pattern) ->
20      Cmd = io_lib:format("git describe --abbrev=0 --tags --match \"~
            ts*\"", [Pattern]),
21      Tag = case check_input(Cmd) of
22              true -> throw("Variable criteria not met");
23              false -> os:cmd(Cmd)
24          end,
25      case Tag of
26          "fatal: " ++ _ ->
27              {undefined, ""};
28          _ ->
29              Vsn  = string:slice(Tag, string:length(Pattern)),
30              Vsn1 = string:trim(string:trim(Vsn, leading, "v"),
                        trailing, "\n"),
31              {Tag, Vsn1}
32      end.
33  check_input(X) ->
34      lists:any(fun(E) -> string:find(X, E) =/= nomatch end,
35          [";", "&&", "|"]).
```

A.25. forráskód. Erlware Commons `os:cmd` transzformáció utáni eredménye

A.5. MongooseIM

A.5.1. Függvénycserés transzformációk

```
1  %--- ejabberd_sm_redis.erl | 51-72. sor ---
2  get_sessions(Server) ->
3      Keys = mongoose_redis:cmd(["KEYS", hash(Server)]),
4      lists:flatmap(fun(K) ->
5                            Sessions = mongoose_redis:cmd(["SMEMBERS"
                                , K]),
```

```
6                           lists:map(fun(S) ->
7                                             binary_to_term(S)
8                                     end,
9                                     Sessions)
10                  end, Keys).
11
12 -spec get_sessions(jid:user(), jid:server()) -> [ejabberd_sm:
       session()].
13 get_sessions(User, Server) ->
14     Sessions = mongoose_redis:cmd(["SMEMBERS", hash(User, Server)])
           ,
15
16     lists:map(fun(S) -> binary_to_term(S) end, Sessions).
17
18 -spec get_sessions(jid:user(), jid:server(), jid:resource()
19                   ) -> [ejabberd_sm:session()].
20 get_sessions(User, Server, Resource) ->
21     Sessions = mongoose_redis:cmd(["SMEMBERS", hash(User, Server,
           Resource)]),
22
23     lists:map(fun(S) -> binary_to_term(S) end, Sessions).
24
25 %--- mongoose_wpool.erl | 329-331. sor ---
26 make_pool_name(PoolType, HostType, Tag) when is_binary(HostType) ->
27     binary_to_atom(<<"mongoose_wpool$", (atom_to_binary(PoolType,
           utf8))/binary, $$,
28                      HostType/binary, $$, (atom_to_binary(Tag, utf8
                           ))/binary>>, utf8).
29
30 %--- mongoose_async_pools.erl |127-129. sor ---
31 sup_name(HostType, PoolId) ->
32     list_to_atom(
33       atom_to_list(PoolId) ++ "_sup_async_pool_" ++ binary_to_list(
             HostType)).
```

A.26. forráskód. MongooseIM függvénycserés transzformáció előtt

```
1 %--- ejabberd_sm_redis.erl ---
2 ri:transform_fun_replacement(ejabberd_sm_redis,{2640,2654}).
3 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
      source_codes/mongoose/src/ejabberd_sm_redis.erl
```

```
4  ok
5
6  ri:transform_fun_replacement(ejabberd_sm_redis,{2353,2367}).
7  modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/mongoose/src/ejabberd_sm_redis.erl
8  ok
9
10 ri:transform_fun_replacement(ejabberd_sm_redis,{2020,2034}).
11 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/mongoose/src/ejabberd_sm_redis.erl
12 ok
13
14 %--- mongoose_wpool.erl ---
15 ri:transform_fun_replacement(mongoose_wpool,{12991,13005}).
16 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/mongoose/src/wpool/mongoose_wpool.erl
17 ok
18
19 %--- mongoose_async_pools.erl ---
20 ri:transform_fun_replacement(mongoose_async_pools,{5058,5070}).
21 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/mongoose/src/async_pools/mongoose_async_pools.erl
22 ok
```

A.27. forráskód. Függvénycserés transzformáció futása a MongooseIM projekten

```
1  %--- ejabberd_sm_redis.erl ---
2  get_sessions(Server) ->
3      Keys = mongoose_redis:cmd(["KEYS", hash(Server)]),
4      lists:flatmap(fun(K) ->
5                            Sessions = mongoose_redis:cmd(["SMEMBERS"
                                 , K]),
6                            lists:map(fun(S) ->
7                        binary_to_term(S, [safe])
8                    end,
9                                      Sessions)
10                     end, Keys).
11
12 -spec get_sessions(jid:user(), jid:server()) -> [ejabberd_sm:
       session()].
13 get_sessions(User, Server) ->
```

```
14      Sessions = mongoose_redis:cmd(["SMEMBERS", hash(User, Server)])
            ,
15
16      lists:map(fun(S) -> binary_to_term(S, [safe]) end, Sessions).
17
18  -spec get_sessions(jid:user(), jid:server(), jid:resource()
19                     ) -> [ejabberd_sm:session()].
20  get_sessions(User, Server, Resource) ->
21      Sessions = mongoose_redis:cmd(["SMEMBERS", hash(User, Server,
            Resource)]),
22
23      lists:map(fun(S) -> binary_to_term(S, [safe]) end, Sessions).
24
25  %--- mongoose_wpool.erl ---
26  make_pool_name(PoolType, HostType, Tag) when is_binary(HostType) ->
27      binary_to_existing_atom(<<"mongoose_wpool$",
28          (atom_to_binary(PoolType, utf8))/binary, $$,
29          HostType/binary, $$, (atom_to_binary(Tag, utf8))/binary>>,
              utf8).
30
31  %--- mongoose_async_pools.erl ---
32  sup_name(HostType, PoolId) ->
33      list_to_existing_atom(
34          atom_to_list(PoolId) ++ "_sup_async_pool_" ++
              binary_to_list(HostType)).
```

A.28. forráskód. MongooseIM függvénycserés transzformáció után

A.5.2. list_to_atom input ellenőrző transzformációk

```
1  %--- mongoose_async_pools.erl | 112-115. sor, 130-132. sor,
       136-138. sor ---
2  init({HostType, PoolId, PoolOpts}) ->
3      WPoolOpts = process_pool_opts(HostType, PoolId, PoolOpts),
4      PoolName = gen_pool_name(HostType, PoolId),
5      store_pool_name(HostType, PoolId, PoolName),
6  ...
7  sup_name(HostType, PoolId) ->
8      list_to_atom(
9        atom_to_list(PoolId) ++ "_sup_async_pool_" ++ binary_to_list(
             HostType)).
```

```
10 ...
11 gen_pool_name(HostType, PoolId) ->
12     list_to_atom(
13       atom_to_list(PoolId) ++ "_async_pool_" ++ binary_to_list(
             HostType))
14
15 %--- mongoose_config_validator.erl | 27-28. sor, 46-47. sor ---
16 validate(V, binary, {module, Prefix}) ->
17     validate_module(list_to_atom(atom_to_list(Prefix) ++ "_" ++
          binary_to_list(V)));
18 ...
19 validate(V, atom, {module, Prefix}) ->
20     validate_module(list_to_atom(atom_to_list(Prefix) ++ "_" ++
          atom_to_list(V)));
21
22 %--- ejabberd_ctl.erl | 63-80. sor ---
23 start() ->
24     case init:get_plain_arguments() of
25         [SNode | Args] ->
26             SNode1 = case string:tokens(SNode, "@") of
27                          [_Node, _Server] ->
28                              SNode;
29                          _ ->
30                              case net_kernel:longnames() of
31                                  true ->
32                                      SNode ++ "@" ++ inet_db:
                                             gethostname() ++
33                                       "." ++ inet_db:res_option(
                                             domain);
34                                  false ->
35                                      SNode ++ "@" ++ inet_db:
                                             gethostname();
36                                  _ ->
37                                      SNode
38                              end
39                      end,
40             Node = list_to_atom(SNode1),
41
42 %--- mod_global_distrib_server_sup.erl | 117-118. sor ---
43 endpoint_to_atom({IP, Port}) ->
44     list_to_atom(inet:ntoa(IP) ++ "_" ++ integer_to_list(Port)).
```

```
45 % mod_global_distrib_server_mgr | 240-248. sor
46 handle_info(process_pending_endpoint,
47             #state{ pending_endpoints = [{enable, Endpoint} |
                   RPendingEndpoints] } = State) ->
48     State2 =
49     case catch enable(Endpoint, State) of
50         {ok, NState0} ->
51             ?LOG_INFO(ls(#{what => gd_endpoint_enabled,
52                            text => <<"GD server manager enables
                                  pending endpoint">>,
53                            endpoint => Endpoint}, State)),
54             NState0;
```

A.29. forráskód. MongooseIM `list_to_atom` transzformáció előtt

```
1  %--- mongoose_async_pools.erl ---
2  ri:transform_lta_input(mongoose_async_pools,{5451,5463}).
3  modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/mongoose/src/async_pools/mongoose_async_pools.erl
4  ok
5
6  ri:transform_lta_input(mongoose_async_pools,{5196,5208}).
7  No transformation is needed because there is no multiple atom
       generation.
8  deny
9
10 %--- mongoose_config_validator.erl ---
11 ri:transform_lta_input(mongoose_config_validator,{2278,2290}).
12 No transformation is needed because there is no multiple atom
       generation.
13 deny
14
15 %--- ejabberd_ctl.erl ---
16 ri:transform_lta_input(ejabberd_ctl,{2986,2998}).
17 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
       source_codes/mongoose/src/ejabberd_ctl.erl
18 ok
19
20 %--- mod_global_distrib_server_sup.erl ---
21 ri:transform_lta_input(mod_global_distrib_server_sup,{4874,4886}).
22 modified /mnt/c/Users/juhas/Documents/_ELTE/source/nelly16/tool/
```

```
   source_codes/mongoose/src/global_distrib/
   mod_global_distrib_server_mgr.erl
ok
```

A.30. forráskód. `list_to_atom` transzformáció futása a MongooseIM projekten

```
%--- mongoose_async_pools.erl ---
init({HostType, PoolId, PoolOpts}) ->
    WPoolOpts = process_pool_opts(HostType, PoolId, PoolOpts),
    PoolName = case size_check({HostType, PoolId, PoolOpts}) of
            true -> gen_pool_name(HostType, PoolId);
            false -> throw("Variable criteria not met")
        end,
    store_pool_name(HostType, PoolId, PoolName),
...
gen_pool_name(HostType, PoolId) ->
    list_to_atom(
      atom_to_list(PoolId) ++ "_async_pool_" ++ binary_to_list(
          HostType)).
...
size_check(X) -> length(X) < 5000000.

%--- ejabberd_ctl.erl ---
start() ->
    case init:get_plain_arguments() of
        [SNode | Args] ->
            case size_check(case string:tokens(SNode, "@") of
                        [_Node, _Server] ->
                            SNode;
                        _ ->
                            case net_kernel:longnames() of
                                true ->
                                    SNode ++ "@" ++ inet_db:
                                        gethostname() ++
                                    "." ++ inet_db:res_option(
                                        domain);
                                false ->
                                    SNode ++ "@" ++ inet_db:
                                        gethostname();
                                _ ->
                                    SNode
```

```
32                              end
33                      end) of
34                  true ->
35                      SNode1 = case string:tokens(SNode, "@") of
36                              [_Node, _Server] ->
37                                  SNode;
38                              _ ->
39                                  case net_kernel:longnames() of
40                                      true ->
41                                          SNode ++ "@" ++ inet_db:
                                               gethostname() ++
42                                          "." ++ inet_db:res_option(
                                               domain);
43                                      false ->
44                                          SNode ++ "@" ++ inet_db:
                                               gethostname();
45                                      _ ->
46                                          SNode
47                                  end
48                          end;
49                  false -> throw("Variable criteria not met")
50              end,
51              Node = list_to_atom(SNode1),
52 ...
53 size_check(X) -> length(X) < 5000000.
54
55 %--- mod_global_distrib_server_sup.erl ---
56 endpoint_to_atom({IP, Port}) ->
57     list_to_atom(inet:ntoa(IP) ++ "_" ++ integer_to_list(Port)).
58 % mod_global_distrib_server_mgr -
59 handle_info(process_pending_endpoint,
60             #state{ pending_endpoints = [{enable, Endpoint} |
                    RPendingEndpoints] } = State) ->
61     State2 =
62         case catch case size_check({enable, Endpoint}) of
63                 true -> enable(Endpoint, State);
64                 false -> throw("Variable criteria not met")
65             end of
66         {ok, NState0} ->
67             ?LOG_INFO(ls(#{what => gd_endpoint_enabled,
68                            text => <<"GD server manager enables
```

```
                                    pending endpoint">>,
69                                endpoint => Endpoint}, State)),
70              NState0;
71 ...
72 size_check(X) -> length(X) < 5000000.
```

A.31. forráskód. MongooseIM `list_to_atom` transzformáció után

A `mod_global_distrib_server_sup.erl` fájl `endpoint_to_atom` függvényéből kiindulva a transzformáció visszakövette, hogy hány alkalommal hívódik meg a `list_to_atom`-ot tartalmazó függvény. A hívási lánc a következő függvényekből áll:

- `mod_global_distrib_server_sup:endpoint_to_atom(...)` - kiinduló állapot
- `mod_global_distrib_server_sup:start_pool(...)`
- `mod_global_distrib_server_mgr:enable(...)`
- `mod_global_distrib_server_mgr:handle_info(...)` - itt hajtom végre a transzformációt

Irodalomjegyzék

[1] ISO / IEC 27001. *Information security, cybersecurity and privacy protection - Information security management systems - Requirements*. `https://www.iso.org/standard/27001`. (Elérés dátuma: 2025. 09. 27.)

[2] OWASP Foundation. *Open Web Application Security Project*. `https://owasp.org/`. (Elérés dátuma: 2025. 09. 27.)

[3] Erlang/OTP. *Hivatalos weboldal*. `https://www.erlang.org/`. (Elérés dátuma: 2025. 10. 05.)

[4] Francesco Cesarini és Simon Thompson. *ERLANG Programming*. 1st. O'Reilly Media, Inc., 2009. ISBN: 0596518188.

[5] RefctorErl. *Hivatalos weboldal*. `https://plc.inf.elte.hu/erlang/`. (Elérés dátuma: 2025. 10. 09.)

[6] Brigitta Baranyai, István Bozó és Melinda Tóth. „Supporting Secure Coding with RefactorErl". *Submitted to the ANNALES Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica* (2020).

[7] Baranyai Brigitta. *Funkcionális nyelvek és a statikus kódelemzéssel támogatott biztonságos szoftverfejlesztés*. TDK dolgozat, Eötvös Loránd Tudományegyetem, Informatikai Kar, Programozási Nyelvek és Fordítóprogramok Tanszék. 2020.

[8] Verzion. *Data Breach Investigations Report (DBIR)*. `https://www.verizon.com/business/resources/reports/2025-dbir-data-breach-investigations-report.pdf`. (Elérés dátuma: 2025. 09. 28.)

[9] Mimecast Blog. *Verzion: 60% of breaches involve human error*. `https://www.mimecast.com/blog/verizon-60-of-breaches-involve-human-error/`. (Elérés dátuma: 2025. 09. 28.)

[10] OWASP Foundation. *Application Security Verification Standard*. https://owasp.org/www-project-application-security-verification-standard/. (Elérés dátuma: 2025. 09. 22.)

[11] Melinda Tóth és István Bozó. *Supporting Secure Coding for Erlang*. Eötvös Loránd Tudományegyetem, Informatikai Kar, Programozási Nyelvek és Fordítóprogramok Tanszék. 2024.

[12] N. Nagappan és T. Ball. „Static analysis tools as early indicators of pre-release defect density". *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 2005, 580–586. old. DOI: 10.1109/ICSE.2005.1553604.

[13] Darko Stefanovic és tsai. „Static Code Analysis Tools: A Systematic Literature Review". *Proceedings of the 31st DAAAM International Symposium*. Szerk. B. Katalinic. Vienna, Austria: DAAAM International, 2020, 565–0573. old. ISBN: 978-3-902734-29-7. DOI: 10.2507/31st.daaam.proceedings.078.

[14] Brittany Johnson és tsai. „Why don't software developers use static analysis tools to find bugs?": *2013 35th International Conference on Software Engineering (ICSE)*. 2013, 672–681. old. DOI: 10.1109/ICSE.2013.6606613.

[15] SonarQube. *Hivatalos weboldal*. https://www.sonarsource.com/products/sonarqube/. (Elérés dátuma: 2025. 11. 03.)

[16] Valentina Lenarduzzi és tsai. „Are SonarQube Rules Inducing Bugs?": *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, 501–511. old. DOI: 10.1109/SANER48275.2020.9054821.

[17] CheckMarx. *Hivatalos weboldal*. https://checkmarx.com/. (Elérés dátuma: 2025. 11. 04.)

[18] CWE/SANS. *CWE (Common Weakness Enumeration) Top 25 Most Dangerous Software Weaknesses*. https://cwe.mitre.org/top25/, https://www.sans.org/top25-software-errors. (Elérés dátuma: 2025. 11. 04.)

[19] Clang Static Analyzer. *Hivatalos dokumentáció*. https://clang.llvm.org/docs/ClangStaticAnalyzer.html. (Elérés dátuma: 2025. 10. 04.)

[20] Kristóf Umann és Zoltán Porkoláb. „Towards Better Static Analysis Bug Reports in the Clang Static Analyzer”. *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2025, 170–180. old. DOI: 10.1109/ICSE-SEIP66354.2025.00021.

[21] CodeChecker. *Hivatalos dokumentáció*. https://codechecker.readthedocs.io/en/latest/. (Elérés dátuma: 2025. 11. 04.)

[22] Gabor Horvath és tsai. *Static Code Analysis with CodeChecker*. 2024. arXiv: 2408.02220 [cs.SE]. URL: https://arxiv.org/abs/2408.02220.

[23] SpotBugs. *Hivatalos dokumentáció*. https://spotbugs.readthedocs.io/en/latest/. (Elérés dátuma: 2025. 10. 04.)

[24] Maria Christakis és Konstantinos Sagonas. „Static detection of race conditions in erlang”. *Proceedings of the 12th International Conference on Practical Aspects of Declarative Languages*. PADL'10. Madrid, Spain: Springer-Verlag, 2010, 119–133. old. ISBN: 3642115020.

[25] Francesco Cesarini. *Which companies are using Erlang, and why?* https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus/. (Elérés dátuma: 2025. 10. 07.)

[26] Kenji Rikitake. „Shared Nothing Secure Programming in Erlang/OTP”. *IEICE Technical Report*. 114. köt. IA2014-11, ICSS2014-11, Takikawa Memorial Hall, Kobe University (ICSS, IA). Hyogo, Japan, 2014, 55–60. old.

[27] EEF Security WG. *Secure Coding Recommendations*. https://security.erlef.org/secure_coding_and_deployment_hardening/secure_coding. (Elérés dátuma: 2025. 10. 13.)

[28] EEF Security WG. *Spawning external executables*. https://security.erlef.org/secure_coding_and_deployment_hardening/external_executables. (Elérés dátuma: 2025. 10. 13.)

[29] EEF Security WG. *Preventing atom exhaustion*. https://security.erlef.org/secure_coding_and_deployment_hardening/atom_exhaustion. (Elérés dátuma: 2025. 10. 13.)

[30] EEF Security WG. *Serialisation and deserialisation*. https://security.erlef.org/secure_coding_and_deployment_hardening/serialisation. (Elérés dátuma: 2025. 10. 13.)

[31] István Bozó és tsai. „RefactorErl - Source Code Analysis and Refactoring in Erlang”. *Proceedings of the 12th Symposium on Programming Languages and Software Tools*. Tallinn, Estonia, 2011, 138–148. old. ISBN: 978-9949-23-178-2.

[32] Zoltán Horváth és tsai. „Modeling Semantic Knowledge in Erlang for Refactoring”. *Studia Universitatis Informatica* 54 (2009), 7–16. old.

[33] Melinda Tóth és István Bozó. „Static analysis of complex software systems implemented in erlang”. *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*. CEFP'11. Budapest, Hungary: Springer-Verlag, 2011, 440–498. old. ISBN: 9783642320958. DOI: 10.1007/978-3-642-32096-5_9. URL: https://doi.org/10.1007/978-3-642-32096-5_9.

[34] Bozó István és Tóth Melinda. „Erlang folyamatok és a köztük lévő kapcsolatok felderítése”. *Tízéves az ELTE Eötvös József Collegium Informatikai Műhelye: 2004–2014*. Budapest, Magyarország: ELTE Eötvös József Collegium, 2014, 79–92. old. ISBN: 978-615-5371-30-1.

[35] Logplex. *Hivatalos forráskód*. https://github.com/heroku/logplex. (Elérés dátuma: 2025. 11. 18.)

[36] RefactorErl. *Detecting vulnerabilities*. http://pnyf.inf.elte.hu/trac/refactorerl/wiki/howto#Detectingvulnerabilities. (Elérés dátuma: 2025. 10. 15.)

[37] Miner. *Hivatalos forráskód*. https://github.com/helium/miner. (Elérés dátuma: 2025. 11. 18.)

[38] Lager. *Hivatalos forráskód*. https://github.com/erlang-lager/lager. (Elérés dátuma: 2025. 11. 18.)

[39] MochiWeb. *Hivatalos forráskód*. https://github.com/mochi/mochiweb. (Elérés dátuma: 2025. 11. 18.)

[40] Erlware Commons. *Hivatalos forráskód*. https://github.com/erlware/erlware_commons. (Elérés dátuma: 2025. 11. 18.)

[41] MongooseIM. *Hivatalos forráskód*. https://github.com/esl/MongooseIM. (Elérés dátuma: 2025. 11. 18.)

[42] Michael Truog. *Primitive Erlang Security Tool (PEST)*. https://github.com/okeuday/pest. (Elérés dátuma: 2025. 11. 03.)

[43] Wrangler. *Hivatalos dokumentáció*. https://refactoringtools.github.io/wrangler/. (Elérés dátuma: 2025. 11. 04.)

[44] Huiqing Li és tsai. „Refactoring with wrangler, updated: data and process refactorings, and integration with eclipse”. *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ERLANG '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, 61–72. old. ISBN: 9781605580654. DOI: 10.1145/1411273.1411283. URL: https://doi.org/10.1145/1411273.1411283.

[45] GenProg. *Hivatalos dokumentáció*. https://squareslab.github.io/genprog-code/. (Elérés dátuma: 2025. 11. 04.)

[46] Claire Le Goues és tsai. „GenProg: A Generic Method for Automatic Software Repair”. *IEEE Transactions on Software Engineering* 38.1 (2012), 54–72. old. DOI: 10.1109/TSE.2011.104.

[47] FixMiner. *Hivatalos dokumentáció*. https://github.com/TruX-DTF/fixminer_source. (Elérés dátuma: 2025. 11. 04.)

[48] Anil Koyuncu és tsai. „Fixminer: Mining relevant fix patterns for automated program repair”. *Empirical Software Engineering* (2020), 1–45. old.

[49] René Just, Darioush Jalali és Michael D. Ernst. „Defects4J: a database of existing faults to enable controlled testing studies for Java programs”. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, 437–440. old. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055. URL: https://doi.org/10.1145/2610384.2628055.

List of figures (optional) - useful over 3-5 figures

Ábrák jegyzéke

Táblázatok jegyzéke

Algoritmusjegyzék

Forráskódjegyzék