Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Programok sebezhetőségének vizsgálata statikus elemzéssel

Szakdolgozat

Témavezetők:

Tóth Melinda, Gera Zoltán

egyetemi docens, egyetemi tanársegéd

Szerző:

Kocsis Ábel

Programtervező informatikus BSc

Budapest, 2020

SZAKDOLGOZAT / DIPLOMAMUNKA

EREDETISÉG NYILATKOZAT

Alulírott Kocsis Ábel Neptun-kód: FGSDV2

ezennel kijelentem és aláírásommal megerősítem, hogy az Eötvös Loránd Tudományegyetem Informatikai Karának, Programozási Nyelvek és Fordítóprogramok Tanszékén írt,

Programok sebezhetőségének vizsgálata statikus elemzéssel

című szakdolgozatom/diplomamunkám saját, önálló szellemi termékem; az abban hivatkozott szakirodalom felhasználása a szerzői jogok általános szabályainak megfelelően történt.

Tudomásul veszem, hogy szakdolgozat/diplomamunka esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Budapest, 2020. 05. 13.

……..

hallgató aláírása

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:

Név: Kocsis Ábel

Neptun kód: FGSDV2

Képzési adatok:

Szak: programtervező informatikus, alapképzés (BA/BSc)

Tagozat: Nappali

Belső témavezetővel rendelkezem

Témavezető neve: *Tóth Melinda; Gera Zoltán*

munkahelyének neve: ELTE-IK, PNYF

munkahelyének címe: 1117, Budapest, Pázmány Péter sétány 1/C

beosztás és iskolai végzettsége: egyetemi docens, PhD; egyetemi tanársegéd, okleveles programtervező matematikus

A szakdolgozat címe: Programok sebezhetőségének vizsgálata statikus elemzéssel

A szakdolgozat témája:

(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

Számtalan olyan esetről hallunk nap mint nap, ahol a rosszul megírt szoftver hibája következtében romlik el egy-egy eszköz, válik használhatatlanná egy termék, vagy történik akár végzetes katasztrófa. Gondoljunk csak az 1996-ban történt Ariane5 hordozórakéta balesetére, ahol egy rossz számkonvertálás döntötte romokba az expedíciót. A programozók által elkövetett hibák gyakran sebezhetővé is teszik a terméket, így egy esetleges támadás esetén kiszivároghatnak a felhasználók adatai. Az Open Web Application Security Project évről évre összegyűjti a leggyakrabb programozók által elkövetett hibákból adódó sebezhetőségeket. Célja, hogy a hackereknek minél ellenállóbb szoftvereket tudjanak a programozók a segítségével készíteni. Ezekből sokat profitálhatnak a kódolók, tesztelők, valamint tehetünk lépéseket az ehhez hasonló hibák automatikus kiszűrésére, felismerésére. Az OWASP alapelvekhez hasonlóan a "SEI-CERT Coding Standards" az igen népszerű programozási nyelvekhez (pl. C, C++, Java, Perl) ad áttekintő elemzést a programok sérülékeny pontjairól. A sérülékenységek elkerülése érdekében már számtalan statikus elemző létezik. A statikus kódanalízis egy olyan számítógép által végzett ellenőrzése a programoknak, mely a kód konkrét futása nélkül történik. Számos szoftver, plugin van már a piacon különböző programnyelvekhez az elemzés elvégzésére. Az LLVM projekt clang-tidy eszköze széleskörű eszközkészletet nyújt statikus elemzés általi ellenőrzések definiálásához. Szakdolgozatomban azzal fogok foglalkozni, hogy további ellenőrzőket (un. checkereket) implementálok majd a keretrendszerre építve és megvizsgálom hogyan egészíthető ki az elemző eszközkészlet még további elemzésekkel és tesztekkel. Elsősorban a "SEI-CERT Coding Standards"-ban megfogalmazott sérülékenységekre fókuszálok.

Budapest, 2019.11.27.

Tartalomjegyzék

1. fejezet

Bevezetés

Számtalan olyan esetről esik szó nap mint nap, ahol a rosszul megírt szoftver hibája következtében romlik el egy-egy eszköz, válik használhatatlanná egy termék, vagy történik akár végzetes katasztrófa. Egy híres példa az 1996. június 4-én történt Ariane5 hordozórakéta balesete, ahol egy nem várt érték és az ebből adódó kezeletlen kivétel döntötte romokba az expedíciót [1]. A sok évnyi munka és a befektetett körülbelül 370 millió dollár pillanatok alatt a semmibe vesztek.

A programozók által elkövetett hibák gyakran sebezhetővé is teszik a terméket, így egy esetleges támadás esetén kiszivároghatnak a felhasználók adatai. A biztonság pedig napjainkban egyre fontosabb mind a fejlesztők, mind a felhasználók számára.

Munkám során a többszálú C és C++ programok potenciális sebezhetőségének felismerésére keresek megoldást. Ehhez a SEI CERT Coding Standards [2] listáját használom kiindulásképpen. A sebezhetőségek detektálására már számos statikus elemző létezik. Ezek közül az egyik az LLVM Clang-Tidy [3], melyet kiegészítek oly módon, hogy képes legyen az említett lista további 5 sebezhetőségének automatikus felismerésére. Ezen kívül készítek egy eszközt, mely segítséget nyújt a fejlesztőnek a megírt checkerek nagy projekteken való teljesen automatikus lefuttatásában.

1.1. A sebezhetőség definiálása és lehetséges megoldások

A szoftverek sebezhetőségének vizsgálata egy gyakran félreértett feladat. „Egy szoftver biztonsága azt jelenti, hogy a szoftver az elvártnak megfelelően működik akkor is, ha éppen rosszindulatú támadásnak van kitéve" [4]. A sebezhetőség vizsgálatánál tehát nem elegendő csupán a kód klasszikus értelemben vett helyességét figyelembe venni. Például amennyiben az eszköz

egy külső input beolvasását végzi, logikusan sokszor felteszi a programozó, hogy az input megfelel a kritériumainak. A sebezhetőség vizsgálatánál azonban fontos feltenni azt a kérdést is, hogy létezik-e olyan bemenet, ami esetén a szoftver sebezhetővé válik. A sebezhetőség tehát egy olyan nem megfelelő működés, error, melyet a támadó idéz elő. Az ilyen támadási pontok észrevétele különösen is nehéz a programozó számára, hiszen a hagyományos tesztesetektől elvonatkoztatva kell gondolkodnia. Különösen nehéz lehet olyan sebezhetőségek felismerésére, amikor a program az esetek többségében jól működik, egyes esetekben viszont rosszul. Erre tipikus példa a többszálú programozás, különösen az olyan alacsonyabb szintű programozási nyelvekben, mint a C és a C++.

Szerencsére napjainkban már számtalan, sokat használt technika áll a programozók rendelkezésére a sebezhetőségek felismerésére. Ezek a technikák mind idő, mind energia, mind pénz tekintetében nagyban eltérhetnek egymástól. A hagyományos kód review ma már minden szoftver fejlesztése során alapnak számít, azonban egyes kutatások eredménye alapján könnyen előfordulhat, hogy minél több reviewer ellenőrzi a kódot, annál nagyobb valószínűséggel tartalmaz végül sebezhetőséget az [5].

Elterjedtnek számít továbbá az automatizált kódelemzés, mely során a nevéből adódóan az eszközök különböző algoritmusok alapján elemzik a programozó által írt kódot. Ezek a szoftvereket működésük alapján két nagy csoportba sorolhatók: a dinamikus és a statikus elemzők csoportjába. A fő különbség a két elemzési módszer között, hogy míg a dinamikus elemzés a program futása közben igyekszik detektálni a potenciális hibákat előre megírt tesztesetek alapján, a statikus kódelemzés a program futtatása és tesztesetek nélkül teszi ugyanezt. Ezekből a tulajdonságokból számos további jellemző következik.

1.2. A statikus elemzés, mint lehetséges megoldás

A statikus elemzés fontos előnye a dinamikus vizsgálattal szemben, hogy nincs szükség előzetes tesztek megadására [6], ezért segítségével olyan hibák detektálására is sor kerülhet, melyekre a fejlesztő egyáltalán nem gondol. Hátránya ugyanakkor a viszonylag magas hamis pozitív ráta, azaz az olyan jelzések aránya, amikor nincs valódi hibáról szó, csak az eszköz értelmezi valamilyen okból annak.

Ma már számos statikus elemző áll a programozók rendelkezésére. Ezek közül sokért csak pénzért lehet hozzáférni (pl.: Coverity [7], CodeSonar [8]), azonban rengeteg nyílt forrás-

kódú eszköz is létezik. Ezek közé tartozik C és C++ nyelvhez az Infer [9], a Cppcheck [10] vagy a Clang Static Analyzer [11]. A legtöbbjük (pl.: Clang Static Analyzer, Cppcheck) szimbolikus kiértékeléssel igyekszik fényt deríteni a potenciális futási idejű hibákra. Ez azt jelenti, hogy a kód alapján modellezi a futási idejű működést: megszorításokat és következtetéseket tesz a változók, függvények működésével kapcsolatban, ez alapján pedig kikövetkezteti, hogy lehetséges-e egy-egy hiba bekövetkezése. Ebből a működésből adódik, hogy bár nagy pontosságra törekednek az elemzők, ezt az elemzési idő kárára teszik: a legtöbb statikus kódelemző futása nagyon lassú.

1.3. A Clang-Tidy elemző

A számos statikus elemző egyike az LLVM projekten belül található Clang-Tidy [3], mely C, C++ és Objective-C nyelvek vizsgálatára készült. Ennek fontos tulajdonsága, hogy nyílt forráskódú, bárki által szabadon használható és fejleszthető. A fentebb említett elemzőkkel (Clang Static Analyzer, Cppcheck) ellentétben ez az eszköz nem szimbolikus kiértékeléssel működik, hanem az absztrakt szintaxis fa (AST) egyszerű elemzésével. Egy program, azaz karaktersorozat gépi kódra való fordításakor a szemantikus elemzés utáni szintaktikus elemzés kimenete a legtöbb fordítónál egy, az adott kódhoz tartozó absztrakt szintaxis fa. Ez egyes fordítóknál nem, vagy csak nehezen hozzáférhető, azonban az LLVM Clang fordító nagy előnye, hogy betekintést enged a fordítás ezen köztes lépésének eredményébe [12]. A Clang-Tidy statikus elemző pedig ezen AST-n végezi az elemzéshez szükséges műveleteket.

Ez az eszköz a korábbiakkal ellentétben nem képes az adatáramlást nyomon követni, azaz egy-egy memóriaszelet értékének a különböző függvényeken keresztül való vizsgálatára. A Clang-Tidy fő célja, hogy a korábban említett AST-n mintákat ismerjen fel, és problémás minták esetén figyelmeztetéseket adjon. A Clang-Tidy nem csak potenciális sebezhetőségek felismerésére hivatott, azonban számos ilyen célú elemzés implementálva van benne, például a `system()` függvény hívásának felismerése [13].

Fontos továbbá megemlíteni, hogy az egész LLVM projekt, de az ezen belüli Clang-Tidy elemző is nagy népszerűségnek örvend. Az utóbbi egy hónapban[1] a Clang-Tidy mappájában 98 fejlesztő által 946 fájlban 18177 sornyi kód került módosításra, ami a nyílt forráskódú statikus elemzők között kiemelkedőnek számít. Ez többek között annak köszönhető, hogy a

[1] 2020. 04. 06. dátumot megelőző 4 hét alatt

többi elemzővel ellentétben jól dokumentált a checkerek, azaz egy adott probléma vagy problémakör megoldására szolgáló kódrészletek fejlesztésének menete. Ezért a Clang-Tidy kitűnő indulási eszköz lehet, ha valaki érdeklődik a statikus elemzés világa iránt.

1.4. Nehézségek a Clang-Tidy eszköz fejlesztése során

Bár az eszközhöz való fejlesztés jól dokumentált és aktív közösség foglalkozik a review-zással, azaz a fejlesztett kódrészletek ellenőrzésével, mégis számos tényező nehezíti a fejlesztő dolgát a közreműködés során. A Clang-Tidy statikus elemző fejlesztése során fontos törekvés a minél alacsonyabb, lehetőleg nulla valószínűségű hamis pozitív előfordulása. Természetesen egy új checker implementálásakor szükség van tesztesetek írására, azonban egy checker elkészítése után nagy nehézséget okozhat olyan tesztek kitalálása, amelyekre mégsem a várt módon működik a megírt elemzés. Ehhez célszerű az új checkert más, nagyobb projekteken lefuttatni. Amennyiben egy nagy, népszerű projektben sok hibát talál, érdemes lehet ellenőrizni, hogy a checker nem ad-e hamis pozitív jelzéseket.

Egy projekten az Ericson által fejlesztett CodeChecker [14] segítségével futtatható egy-egy checker egyszerűen. Ahhoz azonban, hogy számos nagy projekt (pl: bitcoin [15], ffmpeg [16], stb.) elemzésre kerüljön, szükséges a projektek forrásának letöltésén kívül a függőségeik telepítése, valamint a projektek konfigurálása és buildelése is. Ez egy egyetemi védett, vagy egy kis tárhelyű magánszámítógépen nehézségekbe ütközhet.

Azért, hogy a tesztelés menetét felgyorsítsam, automatizáljam, célom egy olyan konténer, azaz virtualizált rendszer készítésének előkészítése és abban az automatizálás elvégzése, mely magától telepíti a szükséges függőségeket és futtatja a szükséges teszteket, majd szükség esetén törli azokat. Ezzel szeretném csökkenteni a tesztelő számítógépére telepítendő függőségeket, valamint megkönnyíteni a tesztelés menetét.

1.5. Többszálú programozás C és C++ nyelveken

Nagy számításigényű programoknál természetes, hogy a programozó a számítógép által biztosított erőforrás minél nagyobb részét ki szeretné használni. Emiatt nem ritka, hogy a jobb kihasználtság érdekében többszálúvá alakítják a programot. Ugyanakkor „csak a leggyakorlottabb programozóktól várható el, hogy biztosan ismerjék a C++ modelljének korlátait és a kódo-

lás során elkerüljék a sebezhetőséget" [17]. Többszálú programozás esetén ugyanis könnyedén előidézhető nemdefiniált működést, mely gyakran sebezhetőséget is jelent.

A SEI CERT Coding Standards a CERT Coordination Center által létrehozott gyűjtemény, amely a legnépszerűbb programozási nyelvek standardjeit tartalmazza, azaz iránymutatást ad a helyes kódolási gyakorlatok elsajátításához [2]. A szabályokat vizsgálva könnyen észrevehető, hogy sok szabály megszegése esetén bekövetkező sebezhetőséget már fel lehet ismerni statikus elemzőkkel. Például a null pointer dereferálását [18] számos elemző képes detektálni, például a következők: Astrée [19], CodeSonar [8], Splint [20], Coverity [7], Cppcheck [10], Klocwork [21]. Azonban hasonló szempontból megvizsgáltam a C és C++ többszálú programozáshoz kapcsolódó szabályait, ahol feltűnt, hogy jóval kevesebb, általában zárt forráskódú elemző detektálja csupán ezeket a hibákat.

Emiatt másik célom ezen C és C++ többszálú programozáshoz kapcsolódó szabályok egy részének implementálása Clang-Tidy-ben.

2. fejezet

Felhasználói dokumentáció

A bemutatni kívánt munka két nagy részre osztható. Az első részben a Clang-Tidy [3] statikus elemző eszközt bővítettem 4 checkerrel. A checker a statikus elemző egy olyan jól elkülönülő része, mely egy adott hibát, hibacsoportot, vagy sebezhetőséget ismer fel. Az általam fejlesztett 4 checker a SEI CERT Coding Standards [2] többszálú programozáshoz kapcsolható 5 szabályának ellentmondó kódrészletet képes automatikusan felismerni. Ezek a szabályok a következők:

- POS44-C. Do not use signals to terminate threads [22]
- CON54-CPP. Wrap functions that can spuriously wake up in a loop [23]
- CON36-C. Wrap functions that can spuriously wake up in a loop [24]
- CON37-C. Do not call `signal()` in a multithreaded program [25]
- CON40-C. Do not refer to an atomic variable twice in an expression [26]

A munkám másik része egy olyan tesztelési környezet kialakítása, mellyel könnyedén tesztelhető az éppen fejlesztés alatt álló új checkert nagy, nyílt forráskódú projekteken. A készített eszköz egy úgynevezett konténerben automatikusan letölti és telepíti a szükséges függőségeket, konfigurálja és buildeli a megadott projekteket, majd automatikusan lefuttatja a projektekre a Clang-Tidy elemzését. Ez tehát automatizálja egy checker fejlesztése során annak bővített tesztelését, ezzel megkönnyítve a fejlesztő munkáját.

2.1. Célcsoport

Az általam fejlesztett checkerek bármilyen szintű C vagy C++ programozónak segítséget nyújthatnak, hogy jobb és biztonságosabb programot hozzon létre. A checkerek használata különösen ajánlott, ha valaki új még a C vagy C++ nyelvek többszálú programozásának világában. A tesztkörnyezetet elsősorban olyan programozók számára fejlesztettem, akik új checkerrel bővítik a Clang-Tidy eszközt, és checkerüket szeretnék nagy projekteken lefuttatni a review megkezdése előtt. Ez a módszer egyébként nagyon ajánlott, hiszen a projekteken való futtatás során új hamis pozitív lehetőségek kerülhetnek elő, ami alapján a checker további javításra szorulhat.

2.2. Fogalomtár

Mielőtt bemutatnám az említett eszközöket, érdemes tisztázni néhány fogalmat, amit a következő fejezetekben használni fogok. A fogalmak különösen fontosak a kezdő, a programozásban még kevés tapasztalattal rendelkező felhasználók számára.

- **Atomikus változók**: angolul atomic variables. Olyan változók, melyek egyes műveletek esetén (+=, ++, stb.) többszálú biztonságot valósítanak meg manuálisan használt mutexek nélkül, így operációs rendszer hívás nélkül. Ez különösen fontos a nem blokkoló algoritmusok esetén.
- **Branch**: egy git repo esetében lehetőség van külön elágazásokat létrehozni. Ez, a jellemzően fő projekt egy adott formáját, verzióját tartalmazó rész a branch.
- **Buildelés**: egy adott projekt forrás- és konfigurációs fájljai alapján magának a projekt a felépítésének, beállításának, a fájlok lefordításásnak és linkelésének összessége a buildelés. Magyarul felépítésnek, esetleg telepítésnek lehetne fordítani, azonban a dolgozat során a buildelés szót fogom többnyire használni.
- **Builder**: az az eszköz, ami a buildelést végzi. Dolgozatom során az LLVM projekt buildelésénél a Ninja szoftvert, a checkerek futtatásánál a GNU Make eszközt használom.
- **Checker**: a statikus elemző egy olyan jól elkülönülő része, mely egy adott hibát, hibacsoportot, vagy sebezhetőséget ismer fel.
- **Dependenciák**: lásd: függőségek.

- **Error**: jellemzően egy program fordítása vagy futtatása során fellépő kritikus hibaüzenet, mely valamilyen egyértelműen rossz működést jelez.
- **Figyelmeztetés**: lásd: warning.
- **Függőségek**: egy program, projekt függőségei azok a különböző eszközök, programok, könyvtárak, melyek szükségesek a program telepítéséhez és futtatásához.
- **Hamis pozitív jelzés**: a statikus elemző által adott olyan jelzés, melyet az elemző hibának, sebezhetőségnek jelez, de valójában nem az.
- **Image**: a konténerek alapja, amire konténer épülhet. Az image buildelését kevesebbszer kell elvégezni, mint a konténer futtatását, valamint az image alapértelmezetten nem veszik el, hanem újabb és újabb konténerek alapjául szolgál.
- **Konfigurálás**: egy projekt konfigurálásán több dolog is érthető. Dolgozatomban amikor ezt a szót használom, a buildelés előtti előzetes beállító lépéseket értem.
- **Konkurens programozás**: lásd: többszálú programozás.
- **Konténer**: az informatika világában egy általában kicsi, elkülönített virtuális rendszert neveznek konténernek. Lényegében egy kis virtuális számítógép, mely jelen esetben a lokális számítógépen fut.
- **Mutex**: a többszálú programozás esetén fellépő versenyhelyzetek kiküszöbölésére szolgáló elem. Ha egy memóriaterület mutex által védve van, másképp fogalmazva mutex által zárolt, akkor csak egy adott szál férhet hozzá ehhez a területhez.
- **POSIX**: a „Portable Operating System Interface for uniX" kifejezés rövidítése, egy szabvány család. A dolgozat során amikor POSIX rendszerekről beszélek, általában Linux vagy Mac OSX operációs rendszerekre gondolok.
- **POSIX szabvány utility-jai**: Az említett rendszerekbe szabvány szerint beépített parancsok, eljárások. Ilyenek például a következőek: `ls`, `cd`.
- **Release**: egy nyílt forráskódú projekt fejlesztése során a hivatalos könyvárban sokszor nem, vagy nem teljesen ellenőrzött fájlok találhatóak. Emiatt a fejlesztők időről időre release-eket adnak ki, amik ellenőrzött, széles körű használatra szánt, stabilan működő verziói az adott szoftvernek, projektnek.
- **Review**: egy jellemzően nagyobb projekt kiegészítése, fejlesztése során a kód bekerülése előtt több programozó ellenőrzésén esik át. Ez az ellenőrzési, visszajelzési folyamat a

review.

- **Script**: (magyarosan szkript) egy előre megírt utasítássorozat. Dolgozatomban a scricpt szó használata alatt általában .sh bash fájlok futtatását értem.
- **Setup**: egy projekt, program beállításának folyamata. Ez tartalmazhatja a projekt git könyvtárának letöltését, a konfigurálás elvégzését, valamint a projekt buildelését is.
- **Szignál**: (angolul signal) egyszerű, alacsony szintű jel, mely különböző szálak vagy programok közti alapszintű kommunikációt valósít meg.
- **Többszálú programozás**: olyan program, mely működése során több szálon fut, kihasználva a rendszer által nyújtott lehetőségeket. Máshogy fogalmazva egyszerre több műveletre is képes.
- **Warning**: jellemzően egy program fordítása vagy futtatása során a fejlesztő számára adott visszajelzés, mely nem kritikus hibára (error), hanem potenciális hibalehetőségre figyelmeztet. A checkerek kimenete általában egy vagy több warning.

2.3. Szabályok magyarázata

Ha a statikus elemzés valamilyen figyelmeztetést ad az adott programra nézve, a programozónak el kell tudnia dönteni, hogy valóban sebezhetőségről van-e szó, vagy hamis pozitív jelzéssel áll szemben. Bár a checkerek úgy lettek megtervezve, hogy lehetőleg ne legyen ilyen jelzés, a statikus elemzés nem tökéletes, ezért nem szabad feltétlenül hinni a figyelmeztetésnek. Ahhoz, hogy ezt el lehessen dönteni, szükséges érteni azon programozási szabályokat, amik alapján a checkerek készültek. Ebben az alfejezetben a szabályok megértéséhez szeretnék támpontot nyújtani[1].

2.3.1. POS44-C. Do not use signals to terminate threads

A POSIX rendszerekben a programozó használhat szignálokat, azaz egyszerű jeleket a különböző processzek közti kommunikációra. Ezeknek a szignáloknak különböző jelentései lehetnek. A `SIGKILL` szignál például arra hivatott, hogy azonnal leállítsa a teljes folyamatot, alkalmazást, szoftvert bármi áron. Ehhez hasonlít a `SIGTERM` szignál abban, hogy célja egy processz leállítása, azonban ezt óvatosabban teszi, ugyanis ez egyfajta figyelmeztetés a processz számára,

[1] Kezdő felhasználó esetén a szabályok megértése elegendő, ha az adott függvény, témakör használata előtt megtörténik.

hogy hamarosan le lesz állítva, de csak azután, hogy az egy biztonságos állapotba (cancellation point) érkezik.

A `pthread_kill()` függvény segítségével lehetőség van egy adott thread, azaz egy szál leállítására signal segítségével. A szabály szerint azonban ha ennek a függvénynek kezeletlen, azaz nem elkapott (unhandled) szignállal történik a hívása, a szignál leállítja a teljes folyamatot, nem csak a különálló threadet, aminek ezt a jelet küldtük [22]. A `SIGTERM` küldése esetén ez különösen is probléma, hiszen ebben az esetben terminálásra, azaz szabályszerű leállásra szeretnénk a programot rávennni, ez azonban csak a fogadó thread esetén történik meg, a processz többi threadje gyakorlatilag figyelmeztetés nélkül áll le akkor is, ha nem értek a korábban elmített biztonságos állapotba.

2.3.2. CON54-CPP. Wrap functions that can spuriously wake up in a loop

Az `std::condition_variable` osztály `wait()`, `wait_for()`, valamint `wait_until()` függvényeit akkor használhatja a programozó, ha egy threadben várakozást szeretne elérni egy adott tulajdonság bekövetkeztéig. Ezek esetén a bizonyos tulajdonságok bekövetkeztéről egy másik thread figyelmezteti az adott szálat (`notify_one()` vagy `notify_all()` függvényekkel).

A várakozás alatt azonban az említett függvények átmenetileg elengedhetik az általuk zárolva tartott mutexet, aminek következtében a várt jelzés nélkül felébredhetnek a várakozásból [23]. Amennyiben a programozó nem végez plusz ellenőrzéseket ezen átmeneti felengedésekre vonatkozóan, mint ahogy azt a 2.1. ábrán látható kódon sem teszi, a feltétel teljesülése nélkül is felébredhet a program, ami sebezhetőséghez vezethet. A megoldásra a programozónak két lehetősége van a C++ nyelvben.

- Az említett függvényeket minden esetben egy ciklusban hívja meg, ahol a ciklus ellenőrzi az említett tulajdonságot. Egy esetleges felébredéskor így a program a ciklus feltételében ismét ellenőrzi a feltételt, és ha az nem teljesül, folytatja a várakozást.

- A függvényeknek van lehetőség lambda függvényt paraméterként megadni a feltétel ellenőrzésére. Amennyiben felébredne, ez a függvény automatikusan lefuttatásra kerül, és csak a predikátum teljesülése esetén ébred fel valójában, különben tovább várakozik.

```
1  #include <condition_variable>
2  #include <mutex>
3
4  struct Node {
5    void *node;
6    struct Node *next;
7  };
8
9  static Node list;
10 static std::mutex m;
11 static std::condition_variable condition;
12
13 void consume_list_element(std::condition_variable &condition)
14 {
15   std::unique_lock<std::mutex> lk(m);
16
17   if (list.next == nullptr) {
18     condition.wait(lk);
19   }
20
21   // Proceed when condition holds.
22 }
```

2.1. ábra. Példa a `condition.wait()` függvény nem megfelelő használatára [23]

2.3.3. CON36-C. Wrap functions that can spuriously wake up in a loop

A C nyelv `cnd_wait()` és `cnd_timedwait()` függvényét hasonlóan használhatja a programozó a C++ előző pontban említett függvényeihez, ugyanabból a célból. Ezek a függvények a `cnd_signal()` és a `cnd_broadcast()` függvényekkel lehetnek felébresztve. Az átmeneti mutex elengedés, és az emiatt való nem várt felébredés ezek esetén is igaz [24]. A fő különbség a C++ függvényekkel szemben, hogy a C függvényeknél nincs lehetőség lambda függvény paraméterként való megadására, ezért mindenképpen ciklussal kell védekezni az esetleges nem várt felébredés ellen.

2.3.4. CON37-C. Do not call `signal()` in a multithreaded program

Egy többszálú program esetén a `signal()` függvény hívása nemdefiniált működés [25]. Tehát ha a program többszálú, azaz threadeket használ valamilyen módon, tilos ezen függvény használata. A hibára egy példa a 2.2. ábrán látható.

```c
#include <signal.h>
#include <stddef.h>
#include <threads.h>

volatile sig_atomic_t flag = 0;

void handler(int signum) {
  flag = 1;
}

/* Runs until user sends SIGUSR1 */
int func(void *data) {
  while (!flag) {
    /* ... */
  }
  return 0;
}

int main(void) {
  signal(SIGUSR1, handler); /* Undefined behavior */
  thrd_t tid;

  if (thrd_success != thrd_create(&tid, func, NULL)) {
    /* Handle error */
  }
  /* ... */
  return 0;
}
```

2.2. ábra. Példa `signal()` függvény használatára többszálú programban [25]

2.3.5. CON40-C. Do not refer to an atomic variable twice in an expression

Az atomikus változók (atomic variables) többszálú biztonságot garantálnak mutexek használata, valamint operációs rendszer hívása nélkül, ami a nem blokkoló algoritmusok esetén különösen fontos. Ez azt jelenti, hogy megadott esetekben - melyekre nemsokára mutatok példát - nincs szükség a programozó által írt többszálú védelemre.

A jobb megértés érdekében a 2.3. ábrán látható C++ kódrészleten keresztül mutatom be az atomikus változók használatának okát. A program során létrejön egy globális `sumThread` változó. A program többi része során egy kellően nagy vektor töltődik fel véletlen egész értékekkel, majd ezek az elemek összeadásra kerülnek többszálúan úgy, hogy az ábrán látható `doSumThread()` függvény kerül futtatása különböző szálakon ugyan azon a vektoron, de kü-

lönböző `to` és `from` paraméterekkel. A fájl teljes kódja megtalálható a dolgozat mellékletében (A. függelék).

```
1 long sumThread = 0;
2 void doSumThread (int from, int to, vector<int> vect){
3     for (int i = from; i < to; i++)
4     {
5         sumThread += vect[i];
6     }
7 }
```

2.3. ábra. Többszálú összegzés programrészlete atomikus változók nélkül, példa a többszálú biztonságot nélkülöző programra

Az eredményt összevetve az egyszálú, lineáris összeadással sokszor különbözik az összeadás eredménye. Ez azért van, mert az 5. sorban található összeadás gyakorlatilag három műveletből áll: a `sumThread` változó olvasásából, a vektorban található érték hozzáadásából, majd az eredmény eltárolásából a `sumThread` változóba. Amennyiben azonban a program többszálú, az első és harmadik lépés között más thread olvashatja vagy írhatja a változó értékét, ami így nem megfelelő értékkel dolgozik tovább. Erre a rossz sorrendre egy példa a 2.1. táblázatban látható. Ilyen sorrend alapján a thread1 által hozzáadott +4 értéke tehát elveszett, összességében nem került hozzáadásra.

sumThread értéke	**thread1 műveletei**	**thread2 műveletei**
12	OLVAS(12)	-
12	+4 (vect[34])	OLVAS(12)
16	ÍR(16)	+6 (vect[164])
18	-	ÍR(18)

2.1. táblázat. Többszálú összeadás nem megfelelő sorrendben való kiértékelése

Mivel a C és C++ nyelvek nem garantálják a számunkra helyes sorrendű összeadást ebben az esetben, két lehetőség van az ilyen és hasonló problémák megoldására. Az egyik a mutexek használata, ami azonban bonyolult lehet a programozó számára, emiatt pedig több hibalehetőséget rejt magában (pl.: deadlock kialakulása). A másik lehetőség az atomikus változók használata.

Amennyiben atomikus változó használatára kerül sor, a többszálú biztonság a += művelet esetén garantálva van, tehát biztos nem történik meg a korábban említett rossz írási-olvasási sorrend. Ehhez egyetlen sort kell átírni a korábban említett programban, mégpedig a `sumThread` változó létrehozását a 2.4. ábra szerint.

```
1 atomic_long sumThread = ATOMIC_VAR_INIT(0);
```

2.4. ábra. Helyes többszálú összegzés programrészlete

Azonban a CERT-es szabály alapján az atomikus változók használata közben is könnyedén előfordulhat logikailag rossz, és ezért könnyedén sebezhető programot eredményező használat [26]. Amíg például a 2.5. ábrán látható programnál feltételezhető, hogy a program egy egész számnál nem kisebb számok összegével tér vissza, ez korántsem biztos, hogy így történik.

```
1  #include <stdatomic.h>
2
3  atomic_int n = ATOMIC_VAR_INIT(0);
4
5  /* Call some threads */
6
7  /* In one thread: */
8  int thread8(){
9      return n * (n + 1) / 2;
10 }
```

2.5. ábra. Atomikus változó helytelen használata

Az `n` változó kétszeri olvasási értéke ugyanis ilyen esetben változhat, tehát lehetséges, hogy a baloldali `n` változó olvasása után másik thread módosítja az `n`-t, és a jobboldali változó olvasásakor már másik értéket olvas a program. Egy program során valószínűsíthető, hogy a programozó a kód írása közben az `n`-re mint azonos értékeket tartalmazó változóra gondolt.

A szabály alapján tehát tilos az atomikus változó használata többször egy adott kifejezésen belül. Ez az adott kifejezés lehet akár függvény, akár bináris operátor. Amennyiben mégis erre van a programozónak szüksége, használhatja az `atomic_compare_exchange` függvényeket. Ezek a függvények lehetővé teszik, hogy az adott atomikus változón hosszabb időt igénybe vevő algoritmust hajtson végre, majd kizárólag abban az esetben írja felül a változót, ha ez időközben nem változott. A checker szempontjából ebből a lényeg, hogy ilyen függvény használata esetén megengedett a változóra való többszöri hivatkozás is egy adott kifejezésben.

2.4. Követelmények

A Clang-Tidy statikus elemző az LLVM projekt része. „Az LLVM projekt egy moduláris és újrafelhasználható fordító és toolchain technológiát foglal magába" [27]. A projekt kezdete egy kutatáshoz vezethető vissza, mely a University of Illinois intézményhez köthető. Abból kifolyólag, hogy a statikus elemző ezen nagy projekt része, nincs lehetőség csak az elemző külön telepítésére. Emiatt bár az általam megírt rész nem mondható kifejezetten hatalmas projektnek, az előkészületek hosszú időt vehetnek igénybe.

Az általam készített tesztelő környezet gyakorlatilag egy kis, virtuális számítógépben futó scriptsorozat, mely elvégzi a munkálatokat a programozó helyett. Ez a virtuális számítógép a konténer, és melyhez egy Docker [28] nevű segédprogramra lesz szükség.

Ebben a fejezetben áttekintem, hogy a telepítés és beállítások elvégzése előtt mire lesz szüksége a programozónak mindkét környezet beállításához. Az LLVM telepítéséhez szükséges követelményeket a hivatalos dokumentációjuk alapján gyűjtöttem össze [29]. Mivel a Docker kézileg könnyen telepíthető, ennek beállítását előzetes követelményként várom a programozótól, ugyancsak a dokumentációja szerint [30].

2.4.1. Számítógép, operációs rendszer, tárhely

Az LLVM és a Docker hivatalosan működik többféle Linux, MacOS, valamint Windows operációs rendszereken is. Szakdolgozatom során Linuxon teszteltem, azon belül egy Debian 10-es verzión, ezért a felhasználói útmutatóm elsősorban ehhez hasonló vagy megfelelő rendszerhez vonatkozik.

Az LLVM telepítéshez legalább 15-20 GB szabad tárhelyre van szükség a dokumentáció szerint, azonban az általam bemutatott telepítési útmutató alapján ez az érték akkor marad 20 GB körüli, ha kizárólag Clang-Tidy eszköz buildelésére kerül sor. Amennyiben azonban a teljes LLVM-et buildelésre kerül, már 55 GB tárhelyre lesz szükség. A buildelés gyorsabbá tétele érdekében ajánlott minél erősebb processzorú számítógép használata. A tesztelés során az általam használt számítógép egy 14 magos Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz processzorral volt felszerelve. Ezen a RAM mérete 31.4 GB volt. Bár ilyen magas gépkövetelményeket nem igényel hivatalosan az LLVM, otthoni, kevés magos számítógépen előfordulhat, hogy nagyon hosszú időbe telik a buildelése. Az alapvető ajánlott gépkövetelmény legalább 8 magos processzor és 8 GB RAM.

A Docker telepítéséhez alapvetően nem szükséges nagy tudású eszköz, az egy egyszerű, alap számítógépre is telepíthető. A hivatalos követelmények alapján [31] a Linux számítógépnek hardver szempontjából a következőkre lesz szüksége: Lunix kernel verzió legalább 3.10 verzió, 2 GB RAM, 3 GB szabad tárhely, valamint statikus IP cím.

A Docker használatához nem, azonban a tesztkörnyezethez új projekt hozzáadásához már mindenképpen POSIX rendszerre van szükség. Ezentúl fontos hangsúlyozni, hogy bár a Docker telepítéséhez az említett köveletmények szükségesek, a tesztkörnyezetet a checkerek tesztelésére lesz használva, ehhez pedig szükséges az LLVM projekt is. Ez tehát azt jelenti, hogy a tesztkörnyezet használata esetén is fennállnak az Tidy-nál felsorolt követelmények. Alapértelmezetten a tesztkörnyezet során tesztelt projektek mérete is beleszólhat a használt tárhely nagyságába, de ez kisebb projektek esetén 1 GB alatt marad.

Összességében ajánlott egy erős, legalább 8 magos számítógép használata, de optimálisabb a 16 mag. Legalább 30 GB szabad tárhely szükséges, de több nagy projekt tesztelése és az egész LLVM könyvtár buildelése esetén pedig 65 GB-ra is nőhet az igényelt hely.

2.4.2. Szükséges eszközök

Mindkét eszköz (LLVM és Docker) telepítése során vannak olyan szoftverek, eszközök, melyekre ezek támaszkodnak. Nélkülük nem telepíthető, buildelhető a két eszköz, ennélfogva nem használhatók a megírt checkereket és a tesztkörnyezetet sem. A szoftverek operációs rendszerenként eltérők lehetnek. Ezeket a 10-es verziójú Debian esetében mutatom be.

Az LLVM függőségei

Az LLVM projekt számára szükséges szoftvereket és azok verziói a 2.2. táblázatban láthatók.

Szoftver neve	Szükséges verzió	Tesztelésnél használt verzió
CMake [32]	>=3.4.3	3.13.4
GCC [33]	>=5.1.0	8.3.0
python [34]	>=2.7	2.7.6
GNU Make [35][2]	3.79, 3.79.1	4.2.1

2.2. táblázat. Szükséges szoftverek és minimális verziójuk az LLVM buildeléséhez

Ezeken kívül a fordító támaszkodik a POSIX szabvány utility-jaira, melyek a következőek: ar, bzip2, bunzip2, chmod, cat, cp, date, echo, egrep, find, grep, gzip, gunzip, install,

[3] A telepítési útmutatómban (2.5. alfejezet) a buildelésre a Ninja eszközt fogjuk használni, ezért a GNU Makere nincs szükségünk. Mivel a hivatalos dokumentáció ajánlja, ezért fontosnak tartottam ideírni.

mkdir, mv, ranlib, rm, sed, sh, tar, test, unzip, zip.

Bár a szoftver telepítéséhez hivatalosan nem szükséges, az általam bemutatott felhasználói útmutató támaszkodni fog további segédszoftverekre is, melyek neve és ajánlott verziója a 2.3. táblázatban látható.

Szoftver neve	Tesztelésnél használt verzió
git [36]	2.20.1
ninja [37]	1.8.2

2.3. táblázat. A telepítési útmutató által használt további szoftverek

A Docker függőségei és telepítése

A Docker számára szükséges szoftvereket és azok verziói a 2.4. táblázatban láthatók. Ezen függőségek megtalálhatók például a Debian csomagjai között [38].

Szoftver neve	Tesztelésnél használt verzió
apt-transport-https	1.8.2
ca-certificates	20190110
curl	7.64.0-4
gnupg-agent	2.2.12-1
software-properties-common	0.96.20.2-2

2.4. táblázat. Szükséges eszközök és minimális verziójuk a Docker telepítéséhez

Az általam létrehozott tesztkörnyezet használatba vételének bemutatása előtt megkövetelem a felhasználótól, hogy önállóan telepítse a Dockert a telepítési útmutató alapján [39]. A telepítési útmutató ugyancsak a Debian említett verziójához tartozik, azonban a honlapon könnyedén megtalálható a telepítés menete más verziókhoz is. A honlap végigvezet a telepítés minden lépésén, így kezdő programozó számára is könnyedén elvégezhető.

Ahogy korábban említettem, a tesztkörnyezethez való projekt hozzáadása és törlése műveletek elvégzéséhez POSIX rendszerre van szükség. A scriptek egészen pontosan a következő POSIX utility-kra támaszkodnak: basename, cp, echo, printf, read, sed.

2.5. Az LLVM telepítése és a tesztkörnyezet beállítása

Az LLVM projekt telepítése és a tesztkörnyezet beállítása már kevésbé magától érthetődő folyamat. Ahhoz, hogy az elkészített munka kipróbálható legyen, részben eltérek a dokumentációban megadott lépésektől. A fejezet során részletesen bemutatom az LLVM projekt letöltésének,

beállításának és buildelésének megfelelő módját, valamint bemutatom az előzetes szükséges lépéseket a tesztkörnyezet használatbavételéhez.

Az LLVM letöltése, konfigurálása és buildelése

A telepítés a 2.6. ábrán látható módon készíthető elő. Ezen parancsokat a számítógép parancssorából kell futtatni, egy tetszőleges mappából. A mappa kiválasztását oly módon kell megtenni, hogy a felhasználónak legyen ott írási joga.

```
1  $ git clone https://github.com/abelkocsis/llvm-project.git
2  $ cd llvm-project
3  $ git checkout thesis2020
4  $ mkdir build
5  $ cd build
6  $ cmake \
7  >    -G "Ninja" \
8  >    -DLLVM_ENABLE_PROJECTS="llvm;clang;clang-tools-extra" \
9  >    -DCMAKE_BUILD_TYPE=RelWithDebInfo \
10 >    -DBUILD_SHARED_LIBS=ON \
11 >    -DLLVM_TARGETS_TO_BUILD=X86 \
12 >    -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
13 >    -DLLVM_ENABLE_ASSERTIONS=ON \
14 >    ../llvm/
```

2.6. ábra. Az LLVM telepítésének előkészítése

Ezekben a sorokban letöltésre kerül az általam előkészített git repository (1. sor), majd ebbe belépve (2. sor) megtörténik a szakdolgozat branchére váltás (3. sor). Ez a branch a legutolsó LLVM release-ből indul ki[4], ezen kívül az általam írt változtatásokat tartalmazza. Ezután a `build` mappában (5. sor) a CMake előkészíti a projekt buildelését, valamint létrehozza a Ninja buildernek szükséges fájlokat (6-14. sor). A `DLLVM_ENABLE_PROJECTS` kapcsoló különösen is fontos, ugyanis ezzel beállítható, hogy a clang-tools-extra projekt is előkészítésre kerüljön, ami tartalmazza a Clang-Tidy elemzőt. A további kapcsolók használatáról a hivatalos dokumentációból [29] lehet információt kapni.

A CMake befejeztével a 2.7. ábrán látható módon ajánlott buildelni a Clang-Tidy-t. Ezt a korábban létrehozott build mappából szükséges futtatni. A 2.8. ábrán látott módon lehetőség van az egész LLVM buildelésére is.

Amennyiben hiba lépne fel valamelyik művelet során, a CMake által generált fájlok

[4] 2020. 04. 13.-án a legújabb, LLVM 10.0.0 verzió

```
1 $ ninja clang-tidy
```

2.7. ábra. A Clang-Tidy ajánlott buildelése

```
1 $ ninja
```

2.8. ábra. A teljes LLVM projekt buildelése

törlése javasolt a 2.9. ábrán látható módon. Ezután a korábbi lépések végrehajtása javasolt ismét.

```
1 $ rm CMakeCache.txt
2 $ rm -r CMakeFiles
```

2.9. ábra. A CMake által generált fájlok törlése

A telepítés végeztével a build mappából a 2.10. ábrán látható módon ellenőrizhető a Clang-Tidy. Ha a telepítés megfelelő volt, a parancs kiíratja a rendelkezésre álló Tidy verzióját.

```
1 $ ./bin/clang-tidy --version
```

2.10. ábra. A Clang-Tidy megfelelő telepítésének ellenőrzése

Amennyiben nincs ilyen fájl, a következők ellenőrzése javasolt:

- a `cmake`, valamint a `ninja` parancs hiba nélkül lefutott,
- a `cmake` parancs következő kapcsolóját a leírásnak megfelelően beállításra került: `-DLLVM_ENABLE_PROJECTS="llvm;clang;clang-tools-extra"`,
- a futtatás elindítására a megfelelő mappából került sor.

Amennyiben nem lépett fel hiba, szükséges ellenőrizni, hogy van-e korábbi `clang-tidy` parancs a jelenlegi `PATH` környezeti változóhoz hozzáadva. Ha van, ezt el kell távolítani. Ezután javasolt az említett környezeti változóhoz az imént telepített eszközök hozzáadása a 2.11. ábrán látható parancs segítségével, melyet a `build` mappából kell futtatni.

A használat előtt ellenőrizendő, hogy valóban hozzáadásra került a `PATH` változóhoz a szükséges parancs. Ezt a 2.12. ábrán látható módon lehet megtenni. Ez a parancs már tetszőleges mappából futtatható.

```
1 $ export PATH=$PATH:$PWD/bin/
```

2.11. ábra. Clang-Tidy és LLVM projekt futtatható fájljainak hozzáadása a `PATH` környezeti változóhoz

```
1 $ clang-tidy --version
```

2.12. ábra. A PATH környezeti változó megfelelő beállításának ellenőrzése

Ezután a programozó készen áll, hogy használatba vegye az elemzőt. Az összes rendelkezésre álló checkert a 2.13. ábrán látható paranccsal lehet lekérni. Az összes rendelkezésre álló checker listája az LLVM 10.0.0. verzióban a hivatalos dokumentációból elérhető [40].

```
1 $ clang-tidy -list-checks -checks=*
```

2.13. ábra. Összes checker lekérdezése

A 2.13. ábrán található parancs eredményeképp kapott listában szerepelnie kell a következő checkerek mindegyikének:

- bugprone-bad-signal-to-kill-thread
- bugprone-do-not-refer-atomic-twice
- bugprone-signal-in-multithreaded-program
- bugprone-spuriously-wake-up-functions

Amennyiben ez nem így van, szükséges ellenőrizni, hogy minden művelet az útmutatónak megfelelően lett elvégezve és a beállítást ismételten megtenni.

A tesztkörnyezetünk letöltése

A tesztkörnyezetet egyszerűen letölthető a 2.14. ábrának megfelelő módon. A tesztkörnyezet használatához továbbá szükség van az LLVM projekt buildelésére is az előző fejezetben megadott módon. A tesztkörnyezet további előkészítésére nincs szükség, annak használatát a 2.7. alfejezetben mutatom majd be.

```
$ git clone \
>  https://github.com/abelkocsis/clang-test-docker.git
```

2.14. ábra. A tesztkörnyezet letöltése

2.6. A statikus elemző használata

Miután az LLVM projekt, valamint a rá épülő Clang-Tidy statikus elemző beállításra került, hozzá is lehet látni ezek használatához. Mivel a checkerek C és C++ programok elemzésére készültek, általánosságban az ilyen fájlokon való használatát mutatom be a statikus elemzőnek. Az eszköz futási idejű üzenetei - amennyiben az inputként megadott fájl fordítható - kizárólag az elemzés eredményeképpen adódó figyelmeztetések, azaz warningok lesznek. A checkerek futtatása egy-egy C vagy C++ fájlon általában rövid ideig tart, a kimenet pedig a használt parancsorban jelenik meg.

Alapvetően a 2.15. ábrán látható parancssori használattal futtatható az elemző.

```
$ clang-tidy <fájlelérés/fájlnév>
```

2.15. ábra. A Clang-Tidy általános futtatása

Az első példát egy `test` mappában létrehozott `test1.cpp` fájlon mutatom be, melyeknek létrehozása a 2.16. ábrán, a fájl tartalma pedig a 2.17. ábrán látható. A fájl mentése után az elemző futtatása a 2.18. ábrán látható paranccsal történik. Ekkor a 2.19. ábrán látható hibaüzenet olvasható.

```
$ mkdir test
$ cd test
$ vim test1.cpp
```

2.16. ábra. Az első elemzés előkészítése

```
void a (int i){
    return;
}
```

2.17. ábra. Az első tesztfájl, a `test1.cpp` tartalma

```
1 $ clang-tidy test1.cpp
```

2.18. ábra. Elemzés futtatása a `test1.cpp` fájlon

```
1  Error while trying to load a compilation database:
2  Could not auto-detect compilation database for file
3      "test1.cpp"
4  No compilation database found in /home/abelkocsis/thesis/test
5      or any parent directory
6  fixed-compilation-database: Error while opening fixed
7      database: No such file or directory
8  json-compilation-database: Error while opening JSON database:
9      No such file or directory
10 Running without flags.
```

2.19. ábra. Error üzenet az első elemzés futtatása során

Az úgynevezett compilation database-ről a fejlesztői dokumentációban (3.6.2. alfejezet) ejtek majd szót, egyelőre megjegyzem, hogy a 2.20. ábra szerint a `--` kapcsolóval kapcsolható ki annak használata.

```
1 $ clang-tidy test1.cpp --
```

2.20. ábra. A Clang-Tidy megfelelő futtatása egy fájl esetén

Ekkor a parancs nem generál semmilyen kimenetet. Ebből látható, hogy az elemző lefutott, és nem talált a fájlban hibát. Azonban a Clang-Tidy checkereit böngészve a hivatalos dokumentációjában [40] észrevehető, hogy a misc-unused-parameters checker jelzi, ha nem használt paraméter van egy függvényben, ami a példában szereplő `a` függvénynél bekövetkezik. Az aktuálisan használt checkerek lekérhetők a 2.21. ábrán szereplő paranccsal.

```
1 $ clang-tidy -list-checks
```

2.21. ábra. Aktuálisan aktív checkerek kiíratása

Ebből kiderül, hogy az aktív checkerek között nem szerepel az említett checker. Az összes checkerrel való elemzés parancsa a 2.22. ábrán látható. Ekkor a 2.23. ábrán olvasható figyelmeztetés kerül kiírásra.

A Clang-Tidy összes checkerét egy adott fájlon tehát a 2.24. ábrán látható paranccsal lehet futtatni.

```
$ clang-tidy -checks=* test1.cpp --
```

2.22. ábra. Az összes checker engedélyezése a `test1.cpp` fájl elemzése esetén

```
    1145 warnings generated.
/home/abelkocsis/thesis/test/test1.cpp:1:13: warning:
    parameter 'i' is unused [misc-unused-parameters]
void a (int i){
            ^
             /*i*/
/home/abelkocsis/thesis/test/test1.cpp:2:9: warning:
    redundant return statement at the end of a function
    with a void return type
    [readability-redundant-control-flow]
        return;
        ^~~~~~
Suppressed 1143 warnings (1143 in non-user code).
Use -header-filter=.* to display errors from all
non-system headers. Use -system-headers to display errors
from system headers as well.
```

2.23. ábra. A Clang-Tidy figyelmeztetései a `test1.cpp` fájl esetén

```
$ clang-tidy -checks=* <dirTo/cOrCppFile> --
```

2.24. ábra. Az összes checker futtatása egy adott fájlon

Amennyiben csak a szakdolgozat során megírt checkerek futtatása a cél, a 2.25. ábra szerint lehet megtenni. Futtatható kizárólag a bugprone modul is - mely tartalmazza az általam hozzáadott négy checkert is - a 2.26. ábrának megfelelő módon.

```
$ clang-tidy \
> -checks=-*,bugprone-bad-signal-to-kill-thread,\
>bugprone-do-not-refer-atomic-twice,\
>bugprone-signal-in-multithreaded-program,\
>bugprone-spuriously-wake-up-functions \
> <dirTo/cOrCppFile> --
```

2.25. ábra. A szakdolgozat során hozzáadott checkerek futtatása

A Clang-Tidy használata során van lehetőség további kapcsolók használatára, melyet az általam hozzáadott checkerek során nem szükséges használni. A kapcsolók elérhetők a Clang-Tidy dokumentációjában [3].

```
1 $ clang-tidy -checks=-*,bugprone-* <dirTo/cOrCppFile> --
```

2.26. ábra. A bugprone modulban lévő checkerek futtatása

2.6.1. A checkerek bemutatása

A megírt checkerek működése jól elkülöníthető egymástól. Bár van lehetőség őket egyszerre futtatni, egymásról nem tudnak és nem használják (nem is használhatják) fel egymás kimeneteit. Emiatt a továbbiakban egyesével mutatom be, hogy melyiket milyen módon lehet használni.

bugprone-bad-signal-to-kill-thread

Ahogy a 2.3.1. alfejezetben is bemutattam, a POSIX rendszerekben lehetőség nyílik arra, hogy szignállal kerüljön egy-egy thread terminálásra. Ugyanakkor ha a felhasználó nem elkapott szignált küld, akkor nem csak a threadet, hanem az egész processzt leállásra kényszeríti [22]. Ez potenciális sebezhetőség, ugyanis a program kezeletlen megszakítása például megakadályozhatja egyes fájlok szabályszerű bezárását, vagy nemdefiniált működéshez vezethet.

A checker olyan `phtread_kill` függvényhívást keres, amely `SIGTERM` szingállal állítja le a thread működését. Használata a 2.27. ábrán látható. Amennyiben egy fájlban hibát talál - a többi checkerhez hasonlóan - figyelmeztetést ad a felhasználó számára. Egy ilyen figyelmeztetés a 2.28. ábrán látható fájl esetén a 2.29. ábrán található.

```
1 $ clang-tidy \
2 >  -checks=-*,bugprone-bugprone-bad-signal-to-kill-thread \
3 >  <dirTo/cOrCppFile> --
```

2.27. ábra. A bugprone-bad-signal-to-kill-thread checker futtatása

A kifejlesztett checkert elfogadták a vezető fejlesztők, és megtalálható a 10.0.0-ás LLVM projektben [41].

bugprone-do-not-refer-atomic-twice

A C és C++ programozási nyelvekben többszálú programozásnál korlátozni kell az adott közös változók elérését a konzisztencia érdekében. Ezt sokszor mutexekkel oldják meg. Ennél kényelmesebb és elegánsabb, valamint több szempontból praktikusabb megoldás a mindkét nyelvben jelenlevő atomikus változók használata, amiről bővebben a 2.3.5. alfejezetben írtam. Az ilyen típusú változók garantálják a biztonságot többszálú programok esetén.

```
1  #include <sys/stat.h>
2  #include <sys/types.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <assert.h>
8  #include <string.h>
9  #include <signal.h>
10 #include <pthread.h>
11
12 void *func(void *foo) {
13   /* Execution of thread */
14 }
15
16 int main(void) {
17   int result;
18   pthread_t thread;
19
20   if ((result = pthread_create(&thread, NULL, &func, 0)) != 0)
21   {
22     /* Handle Error */
23   }
24   //CHECK: do not send signal to terminate a trhead
25   if ((result = pthread_kill(thread, SIGTERM)) != 0) {
26     /* Handle Error */
27   }
28
29   /* This point is not reached because the process terminates
30   in pthread_kill() */
31
32   return 0;
33 }
```

2.28. ábra. A `badSignal.c` fájl tartalma [22]

Ahogy említettem a korábbi fejezetben is, ez a biztonság elveszik, ha egy kifejezésen belül kétszer használja a programozó ugyan azt a változót [26]. Bár a Clang-Tidy elemző adta lehetőségekkel ezt a szabályt nem lehet tökéletesen megoldani (ennek okáról bővebben a 3.4.1. alfejezetben írok), jelentősen szűkíthető a fel nem ismert problémák halmaza a következő megoldással: az általam írt checker jelez, ha egy bináris operátor (pl.: `=`, `+`, `-`, `*`, `/`, stb.) jobb és bal oldalán is szerepel ugyanaz az atomikus változó. A checker a 2.30. ábra szerint használható. Az `atomicTwice.c` nevű példafájlra (2.31. ábra) való futtatás eredménye a 2.32. ábrán látható.

```
$ clang-tidy \
>  -checks=-*,bugprone-bad-signal-to-kill-thread \
>  ./badSignal.c --
2 warnings generated.
/home/abelkocsis/atm/thesisExamples/badSignal.c:28:17:
warning: thread should not be terminated by raising the
'SIGTERM' signal [bugprone-bad-signal-to-kill-thread]
  if ((result = pthread_kill(thread, SIGTERM)) != 0) {
                ^
Suppressed 1 warnings (1 with check filters).
```

2.29. ábra. A bugprone-bad-signal-to-kill-thread checker futtatása a `badSignal.c` fájlra

```
$ clang-tidy \
>  -checks=-*,bugprone-do-not-refer-atomic-twice \
>  <dirTo/cOrCppFile> --
```

2.30. ábra. A bugprone-do-not-refer-atomic-twice checker futtatása

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void) {
  bool temp_flag = flag || flag;
}
```

2.31. ábra. Az `atomicTwice.c` fájl tartalma

```
$ clang-tidy \
>  -checks=-*,bugprone-do-not-refer-atomic-twice \
>  ./atomicTwice.c --
1 warning generated.
/home/abelkocsis/atm/thesisExamples/atomicTwice.c:7:28:
warning: Do not refer to 'flag' atomic variable twice
in an expression [bugprone-do-not-refer-atomic-twice]
  bool temp_flag = flag || flag;
                           ^
```

2.32. ábra. A bugprone-do-not-refer-atomic-twice checker futtatása az `atomicTwice.c` fájlra

A checker dolgozatom írása alatt[5] review alatt áll [42].

[5] 2020. 04. 13-án

bugprone-signal-in-multithreaded-program

A `signal()` függvény meghívása egy többszálú programban nemdefiniált működés [25], ahogy azt a 2.3.4. alfejezetben is említettem. A checker megkeresi a `signal()` hívásokat egy olyan programban, amit többszálúként határoz meg. A checker alapértelmezetten többszálúnak határoz meg egy programot, ha `std::thread` létrehozás történik benne, vagy a következő függvények bármelyike meghívásra kerül: `thrd_create()`, `std::thread()`, `boost::thread()`, `pthread_t()`. A checker futtatása a 2.33. ábrán látható. A `signalMulti.c` fájl tartalma a 2.34. ábrán, az elemzésének eredménye a 2.35. ábrán látható.

```
1 $ clang-tidy \
2 > -checks=-*,bugprone-signal-in-multithreaded-program \
3 > <dirTo/cOrCppFile> --
```

2.33. ábra. A bugprone-signal-in-multithreaded-program checker futtatása

Ugyanakkor természetesen egy program nem csak akkor többszálú, ha valamelyik feljebbi függvényhívás megtörténik benne. Használhat a programozó egyéb, akár saját thread könyvárat is. Ilyen esetben lehetősége van plusz paraméterként megadni a saját thread hívó függvénye nevét a checkernek paraméterként, ahogy az a 2.36. ábrán is látható.

A checker dolgozatom írása közben[6] review alatt áll [43].

bugprone-spuriously-wake-up-functions

Ahogy a 2.3.2. és 2.3.3. alfejezetekben is bemutattam, az `std::condition_variable` osztály `wait()`, `wait_for()`, és `wait_until()` függvényei (C++), valamint a `cnd_wait()` és `cnd_timedwait()` (C) függvények átmenetileg elengedhetik az általuk birtokolt mutexet [23, 24]. Emiatt ezen függvényeket hívása csak olyan kódból helyes, amelyek valamilyen módon a meghatározott feltétellel védve vannak. Az általam készített checker ellenőrzi, hogy az említett függvényhívások ciklussal (`while`, `dowhile` vagy `for`) vagy lambda függvény paraméterrel védve vannak-e. Mivel a szükséges „felébredési" feltétel más és más lehet, az nem kerül ellenőrzésre, hogy amennyiben ciklusban helyezkedik el a függvény, a ciklus feltétele helyes-e, hanem a checker feltételezi, hogy az. A checkert a 2.37. ábrán látható módon lehet meghívni. A 2.38. ábrán látható fájlon való elemzésének az eredménye a 2.39. ábrán látható.

A checker átment a review-n és elfogadásra került [44], a jelenlegi llvm-project könyvtárban szerepel, és a következő release-ben minden bizonnyal benne lesz.

[6] 2020. 04. 13-án

```
1  #include <signal.h>
2  #include <stddef.h>
3  #include <threads.h>
4
5  volatile sig_atomic_t flag = 0;
6  int thrd_create(thrd_t *__thr, thrd_start_t __func,
7      void *__arg) { return 0; };
8
9  void handler(int signum) { flag = 1; }
10
11 /* Runs until user sends SIGUSR1 */
12 int func(void *data) {
13   while (!flag) {
14     /* ... */
15   }
16   return 0;
17 }
18
19 int main(void) {
20   signal(SIGUSR1, handler); /* Undefined behavior */
21   thrd_t tid;
22
23   if (thrd_success != thrd_create(&tid, func, NULL)) {
24     /* Handle error */
25   }
26   /* ... */
27   return 0;
28 }
```

2.34. ábra. A `signalMulti.c` fájl tartalma [25]

```
1  $ clang-tidy
2  >  -checks=-*,bugprone-signal-in-multithreaded-program  \
3  >  ./signalMulti.c --
4  1 warning generated.
5  /home/abelkocsis/atm/thesisExamples/signalMulti.c:19:3:
6  warning: signal function should not be called in a
7  multithreaded program
8  [bugprone-signal-in-multithreaded-program]
9    signal(SIGUSR1, handler); /* Undefined behavior */
10   ^
```

2.35. ábra. A bugprone-signal-in-multithreaded-program checker futtatása a `signalMulti.c` fájlra

```
$ clang-tidy \
> -checks=-*,bugprone-signal-in-multithreaded-program \
> -config='{CheckOptions: \
> [{key: bugprone-signal-in-multithreaded-program.ThreadList,\
> value: "thrd_create"}]}' \
> test2.cpp --
```

2.36. ábra. A bugprone-signal-in-multithreaded-program checker futtatása opcionális kapcsolóval

```
$ clang-tidy \
-checks=-*,bugprone-spuriously-wake-up-functions \
<dirTo/cOrCppFile> --
```

2.37. ábra. A bugprone-spuriously-wake-up-functions checker futtatása

```
1  #include <condition_variable>
2  #include <mutex>
3
4  struct Node {
5    void *node;
6    struct Node *next;
7  };
8
9  static Node list;
10 static std::mutex m;
11 static std::condition_variable condition;
12
13 void consume_list_element(std::condition_variable &condition)
14 {
15   std::unique_lock<std::mutex> lk(m);
16
17   if (list.next == nullptr) {
18     condition.wait(lk);
19   }
20
21         // CHECK: The thread may wake up too early
22   // Proceed when condition holds.
23 }
24
25 int main()
26 {
27         std::condition_variable c;
28         consume_list_element(c);
29 }
```

2.38. ábra. Az `spWakeUpFunctions.cc` fájl tartalma

```
1  $ clang-tidy \
2  >  -checks=-*,bugprone-spuriously-wake-up-functions \
3  >  ./spWakeUpFunction.cc --
4  1 warning generated.
5  /home/abelkocsis/atm/thesisExamples/spWakeUpFunction.cc:17:15:
6  warning: 'wait' should be placed inside a while statement or
7  used with a conditional parameter
8  [bugprone-spuriously-wake-up-functions]
9      condition.wait(lk);
10                ^
```

2.39. ábra. A bugprone-spuriously-wake-up-functions checker futtatása az `spWakeUpFunctions.cc` fájlra

2.7. A tesztkörnyezet használata

A tesztkörnyezet nagy projektek telepítésére és Tidy checkerekkel való tesztelésére hoztem létre. Célom volt, hogy ezeket minél inkább automatizálva tegye meg, kezdve a dependenciák telepítésével, folytatva a projektek beállításával és elemzésével, és befejezve akár a projektek teljes törlésével. Egy konténer használata alapvetően két nagy lépésből áll: az image buildeléséből, amelyre ideálisan csak egyszer, de mindenesetre ritkán van szükség; illetve a konténer futtatásából, amire már gyakrabban sor kerül.

2.7.1. Az image buildelése

Az image a leendő tesztkörnyezet, konténer kiindulási alapja. A továbbiakban felsorolt műveletek futtatására a korábban letöltött `clang-test-docker` mappából van szükség. Az első lépés az image buildelése annak megfelelően, hogy mire szeretné a programozó használni. Amennyiben azt szeretné, hogy később mindenre (mind beállításra, mint tesztelésre) képes legyen, vagy még nem biztos ebben, használhatja az alap, 2.40. ábrán látható parancsot a számítógép parancssorából futtatva. Ekkor az előre konfigurált projektek összes függőségét feltelepíti erre az image-re automatikusan a rendszer, ami jelentős időbe telhet, azonban csak egyszer kell végrehajtani.

```
1 $ docker build -t test-clang .
```

2.40. ábra. A teszt image alapértelmezett buildelése

A buildeléskor van lehetőség több opció közül választani, amik később befolyásolják a konténer futtatását is. Ezeket a 2.41. ábra szerint, buildelési argumentumként (build arguments) lehet megadni.

```
1 $ docker build -t <kontenerNev> .  \
2 >    --build-arg <builArg1Nev>=<buildArg1Ertek> \
3 >    --build-arg <builArg2Nev>=<buildArg2Ertek> \
4     ...
```

2.41. ábra. Argumentumok használata image buildelésénél

A rendelkezésre álló argumentumok és azok funkciói a 2.5. táblázatban találhatók.

Paraméter neve	**Paraméter lehetséges értéke**	**Paraméter alapértelmezett értéke**	**Paraméter leírása**
setup	TRUE / FALSE	TRUE	Elvégzi a szükséges konfigurálásokat és telepítéseket a projekten
analyze	TRUE / FALSE	TRUE	Elvégzi az elemző futtatását a már beállított és telepített projekteken
projects	projektek nevei vesszővel elválasztva, szóköz nélkül	all	Beállítja, hogy a műveletek (beállítás, elemzés) melyik projekteken menjenek végbe.
checkers	checkerek nevei vesszővel elválasztva, szóköz nélkül	all	Beállítja, hogy az elemzés során mely checkereket futtassa a projekteken
delete	TRUE / FALSE	FALSE	TRUE esetén a műveletek elvégzése után törli a projektek mappáit.

2.5. táblázat. A konténer buildelésénél rendelkezésre álló argumentumok és funkcióik

Jelenleg korlátozott számú projekt tesztelésére van lehetőség, azonban ahogy később bemutatom majd, ezen projektek számát van lehetőség egyszerűen bővíteni. Alapból a következő projektek kiválasztás megengedett: bitcoin [15], curl [45], ffmpeg [16], memcached [46], nginx [47], postgres [48], redis [49], tmux [50], openssl [51], git [36], vim [52], cpp-taskflow [53], enkiTS [54], RaftLib [55], FiberTaskingLib [56]. Fontos tudni, hogy a projektekre kizárólag pontosan az itt megadott, azaz a git könyvtárukban található neveikkel lehet hivatkozni a tesztkörnyezet használata közben. Tehát a „FiberTaskingLib" helyett helytelen a „fiberTaskingLib" vagy a „fibertaskinglib" név használata, ezekben az esetekben errort jelez a környezet.

A fentiek alapján amennyiben egy olyan konténer alap image-ét szeretné a programozó létrehozni, mely telepíti, konfigurálja, és teszteli a bitcoin és curl projekteket a bugprone-bad-signal-to-kill-thread checker alapján, erre a 2.42. ábra szerint van lehetősége.

```
1 $ docker build -t test-bit-and-curl . \
2 >    --build-arg checkers=bugprone-bad-signal-to-kill-thread \
3 >    --build-arg projects=bitcoin,curl
```

2.42. ábra. Példa egy image buildelésére

Ebben az esetben a test-bit-and-curl nevű konténerben a CodeChecker, a bitcoin, va-

lamint a curl projektek függőségei lesznek telepítve. Fontos megemlíteni, hogy a tesztelést a CodeChecker [14] eszköz futtatja majd, tehát a Codechecker függőségei minden esetben telepítve lesznek. Ezen kívül a kiválasztott projektek függőségei kerülnek telepítésre.

Ahogy később bemutatom majd, egyes paraméterek a konténer futtatásakor módosíthatók. A `checkers`, a `delete`, a `run`, valamint a `setup` paramétereket kizárólag akkor érdemes beállítani ebben a lépésben, ha a futtatásokkor az adott értéket szeretné a felhasználó alapértelmezettnek. A `projects` változóra azonban már fontos odafigyelni. Adott konténerhez ugyanis később már nem lehetséges utólag új projektet hozzáadni, csak az image teljes újrabuildelésével. Ezért ezt kötelező ezen lépésnél, azaz az image buildelésénél megfelelően megadni.

2.7.2. A tesztkörnyezet futtatása

A konténer használata előtt két fontos lépésre van szükség. Egyrészt meghatározni, szükség esetén létrehozni egy mappát, melybe a konténer a projekteket letöltheti, majd buildelheti. Itt fontos megjegyezni, hogy a projektek függőségei bár a konténerben találhatóak, maguk a felépített projektek lokálisan a felhasználó számítógépén, úgynevezett hozzácsatolt mappákban lesznek elhelyezve. Ez gyakorlatilag azt jelenti, hogy a telepített projektek a konténer futásának befejeztével nem tűnnek el, mivel a számítógépen vannak (nem a konténerben), azaz a következő futtatáskor nem kell ismét beállítani őket. A futtatás előtt buildelni kell továbbá az LLVM projektet a 2.5. alfejezetben leírt módon. Bár az előző fejezetben opcionális volt az `llvm-project/build` mappába való buildelése a projektnek, a tesztkörnyezet működtetéséhez kötelező így buildelni. Példáim során a teszt mappa elérési útját a `$testingRootDir` változóban, míg a buildelt LLVM főmappájának elérését (`llvm-project`) az `$llvm` változóban tárolom.

A konténert alapvetően a 2.43. ábrán látható módon van lehetőség futtatni. Ezzel a paranccsal a korábbi buildelésnek megfelelő feladatokat végzi el automatikusan a környezet. Ez gyakorlatilag azt jelenti, hogy ha az image buildelésénél a beállítást és az elemzést is engedélyezve lett az összes projektre, a kiadott parancs után ezek végre is hajtódnak.

```
1 $ docker run \
2 >     -v <localDirToBuildProjects>:/testDir \
3 >     -v <localDirToLlvmProject>:/llvm-project \
4 >     <kontenerNev>
```

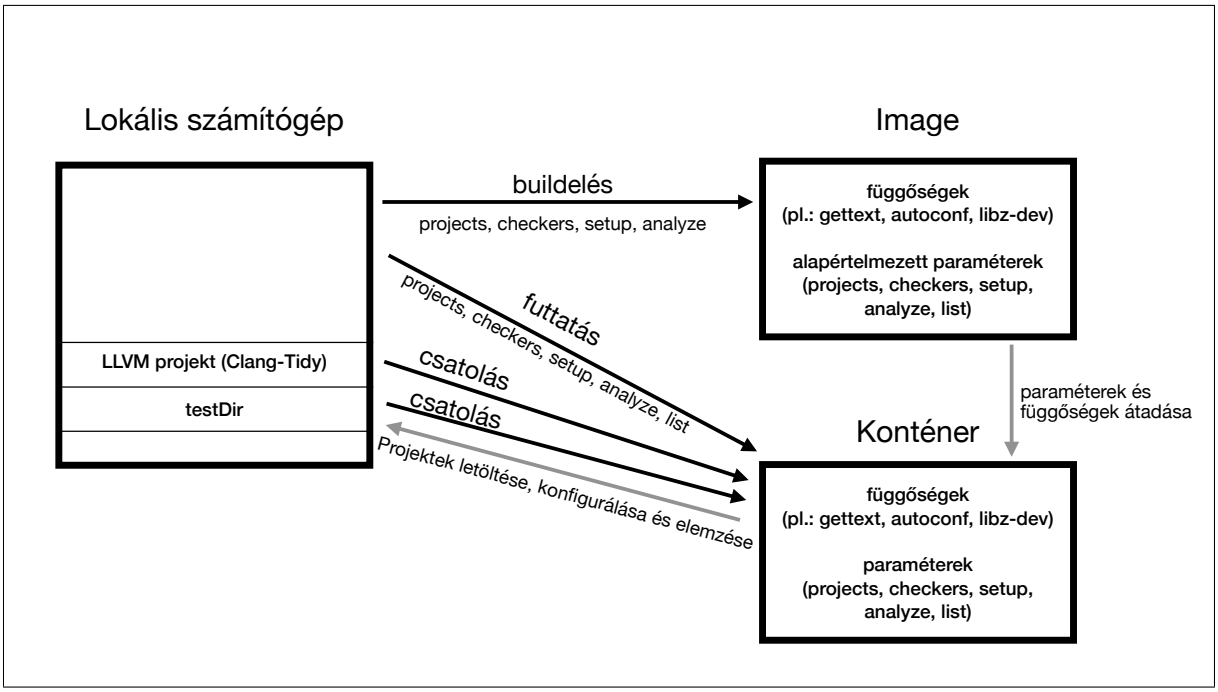
2.43. ábra. A tesztkonténer alapvető futtatása

A továbbiakban a test-clang nevű konténeren folytatom a bemutatást. A futtatásnál megadhatók hasonló argumentumok mint a buildelésnél, a 2.44. ábrán látható módon. A korábbi buildelési argumentumok (build-args) módosítására a megegyező nevű környezeti változók (environmental variables) segítségével van lehetőség, ahogy az ábrán is látható.

```
1 $ docker run test-clang \
2 >     -e "<arg1>=<ertek1>" \
3 >     -e "<arg2>=<ertek2>" \
4       ...
```

2.44. ábra. Környezeti változók beállítása a tesztkörnyezet futtatásánál

A lokális számítógép, valamint a korábban említett image és konténer leegyszerűsített kapcsolata a 2.45. ábrán látható. A fekete nyíllal jelölt műveletek (image buildelése, konténer futtatása, mappák csatolása) manuálisan elvégzendő műveletek, míg a szürkék (paraméterek és függőségek átadása, valamint a futtatási lehetőségek) automatikusak. A konténer minden egyes futtatás után törlődik, azonban az ábrából jól látszik, hogy a projektek és hozzájuk tartozó függőségek hogyan használhatók fel újra és újra.



2.45. ábra. A lokális számítógép, a buildelt image, valamint a konténer leegyszerűsített kapcsolata

A tesztkörnyezet futása, működése jól elkülöníthető részekre osztható, melyek a következők:

- a projektek kilistázása,
- a projektek beállítása és konfigurálása,
- a projektek tesztelése,
- a projektek mappáinak törlése,
- projekt hozzáadása, valamint
- projekt törlése.

Az alábbiakban ezen funkciókat mutatom be.

Projektek kilistázása

Ahogy korábban említettem, amennyiben a buildeléskor egy projektet nem ad hozzá a programozó a `projects` paraméterhez, azt a beállítás vagy tesztelés futtatásakor már nem lehet utólag hozzáadni, mivel - a függőségei hiányában - nagy valószínűséggel sikertelen lenne a konfigurálása vagy a buildelése. Ugyanakkor könnyen elfeledkezhet arról a programozó, hogy az eredetileg buildelt image-ben milyen projekteket engedélyezett.

Ezen praktikussági okból a következő paranccsal lekérhetők az engedélyezett - ezért elemzésre alkalmas - projektek a 2.46. ábra szerint.

```
1 $ docker run -e "list=TRUE" test-clang
```

2.46. ábra. Projektek kilistázása adott tesztkonténer esetén

Ebben a kivételes esetben a fentebb említett mappák (felépített llvm, valamint projektek mappájának) megadása nem kötelező. Ezzel a kapcsolóval azonban automatikusan kikapcsolásra kerül minden más funkció: ha ez a változó igaz, semmi más nem végezhető az adott futtatás során. Fontos ugyanakkor, hogy a listázás engedélyezésekor ne adjon meg a programozó a `projects` paraméternek új, futási idejű értéket, ugyanis ekkor ez kerül kiíratásra, nem a buildelésénél beállított érték.

Projektek mappáinak törlése

A konténeren keresztül lehetőség van a projektek teljes mappáinak automatikus törlésére a 2.47. ábrán látható paranccsal. A parancs hatására a kiválasztott projektek, melyek mappái a lokális számítógépen találhatók, törlődnek. Természetesen ha egyszerre beállítás vagy elemzés is beállításra kerül a törléssel együtt, a törlésre csak az egyéb műveletek után kerül sor.

```
$ docker run -e "delete=TRUE" \
> -v $testingRootDir:/testDir test-clang
```

2.47. ábra. Projektek mappáinak törlése az elemzés befejeztével

Ebben az esetben is tetszőlegesen módosíthatók a törlésre szánt projektek. A 2.48. ábra parancsával semmi mást nem történik, csak a bitcoin projekt mappája törlésre kerül. Amennyiben egy projekt mappája törlődik, természetesen később azt a `setup` paraméterrel van lehetőség ismételten letölteni és beállítani.

```
$ docker run -e "delete=TRUE" -e "projects=bitcoin" \
>    -e "setup=FALSE" \
>    -e "analyze=FALSE" \
>    -v $testingRootDir:/testDir test-clang
```

2.48. ábra. Példa egyetlen projekt mappájának törlésére a konténeren keresztül

Projektek beállítása és konfigurálása

A projektek beállítását engedélyezve a környezet a kiválasztott könyvtárba letölti a kiválasztott projektet, amennyiben még nincs letöltve. Amennyiben megtalálja, frissíti azt. Emiatt a setupot nem csak egyszer érdemes engedélyezni, hanem időről időre egyébként is a projektek frissítése céljából. A letöltés után automatikusan rááll a környezet a legutóbbi tagre, ami általában[7] a legfrissebb stabilan működő verzió.

Ezután a setup elvégzi a szükséges konfigurálásait a kiválasztott projekteknek valamint a CodeChecker projektnek, amely majd az elemző futtatását végzi. Végső lépésként beállítja a CodeCheckert az elemzéshez, valamint lefuttatja a CodeChecker `log` parancsát, melyet minden projekt esetén egyszer kell. A parancs során a CodeChecker buildeli a projektet, valamint menti a buildelési parancsokat, melyeken alapján később a Clang-Tidy az elemzést végzi.

Amennyiben a programozó csak a beállítást szeretné elvégezni, futtathatja a 2.49. ábrán látható parancsot. Természetesen a korábban említett változók (pl.: projects) rendelkezésre állnak, és tetszőlegesen kiegészíthető velük a parancs.

[7] Az általam tesztelt projektek mindegyikénél

```
$ docker run \
>     -e "setup=TRUE" \
>     -e "analyze=FALSE" \
>     -v $testingRootDir:/testDir \
>     -v $llvm:/llvm-project \
>     test-clang
```

2.49. ábra. Projektek letöltésének és konfigurálásának futtatása

Projektek elemzése

A projektek elemzése az egész környezet legfontosabb művelete, a korábbi lehetőségek inkább csak ennek a megkönnyítését, egyszerűbbé tételét hivatottak végezni. A futtatás elején a környezet automatikusan beállítódik a CodeChecker működésének megfelelően, majd megkezdődik az elemzés. Az elemzés átmeneti fájljait a felhasználó által megadott `$testingRootDir/reports/projektnév` mappába menti a működés során a program. Az elemzés végső, felhasználóbarát kimenete HTML formátumban megtekinthető, mégpedig a `$testingRootDir/reports/<projektnév>/html` mappában található `index.html` fájl megnyitásával.

A 2.50. ábrán a RaftLib projekt összes Clang-Tidy checkerrel való elemzése utáni kimenete látható.

Go To Statistics

	File	Severity	Checker name	Message	Bug path length
1	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 23	L	llvm-include-order	#includes are not sorted properly	2
2	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 34	L	cppcoreguidelines-pro-type-member-init	constructor does not initialize these fields: argv, argc	1
3	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 34	L	performance-unnecessary-value-param	the const qualified parameter 'n' is copied for each invocation; consider making it a reference	2
4	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 48	S	cppcoreguidelines-owning-memory	deleting a pointer through a type that is not marked 'gsl::owner<>'; consider using a smart pointer instead	2
5	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 54	L	cppcoreguidelines-avoid-c-arrays	do not declare C-style arrays, use std::array<> instead	1
6	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 54	S	cppcoreguidelines-avoid-magic-numbers	81 is a magic number; consider replacing it with a named constant	1
7	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 55	L	cppcoreguidelines-pro-bounds-array-to-pointer-decay	do not implicitly decay an array into a pointer; consider using gsl::array_view or an explicit cast instead	1
8	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 55	S	cppcoreguidelines-avoid-magic-numbers	81 is a magic number; consider replacing it with a named constant	1
9	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 56	L	cppcoreguidelines-pro-bounds-array-to-pointer-decay	do not implicitly decay an array into a pointer; consider using gsl::array_view or an explicit cast instead	1
10	/testDir/RaftLib/git-dep/cmdargs/lib/command_arguments.cpp @ Line 64	L	cppcoreguidelines-pro-bounds-array-to-	do not implicitly decay an array into a pointer; consider using gsl::array_view or an explicit cast instead	1

2.50. ábra. A RaftLib projekt elemzésének eredménye (részlet)

2.7.3. Projekt hozzáadása

Új image buildelése előtt van lehetőség új projekt hozzáadására a környezethez. Ezután a hozzáadott projekt is a már előre beállítottakhoz hasonlóan fog működni: automatikusan konfigurálásra, valamint elemzésre kerül. Az egyszer hozzáadott projekt ezután a konténer törlésével sem veszik el a listából, azaz új image buildelésekor nem szükséges az újbóli hozzáadás, az a többi projekttel azonos módon használható.

Fontos, hogy a felhasználó ne keverje össze a projekt elemzéshez való hozzáadását, valamint a projekt tesztkörnyezethez való hozzáadását. Az image buildelésénél, majd a konténer futtatásánál beállítható a `projects` változó, azonban kizárólag az előre megadott projektekkel. Ezeket a 2.7.2. alfejezetben soroltam fel. Amennyiben teljesen új projektet szeretne hozzáadni, először az ebben az alfejezetben leírt műveletekre lesz szüksége.

A legtöbb projekt rendelkezik valamilyen konfiguráló, beállító paranccsal vagy parancsokkal. Azaz általános nem várják el, hogy a programozó telepítés előtt minden beállítást tökéletesen megértsen és megtegyen, hanem scriptek segítségével elérik, hogy a projekt beállítsa önmagát. Ehhez persze a programozónak futtatnia kell az előre megadott scripteket.

Ahhoz, hogy helyesen új projektet tudjon hozzáadni a felhasználó a tesztkörnyezethez, először szükséges megértnie, hogyan működik a projektek automatikus konfigurálása. Egy-egy projekt esetén a konténerben futó script a következőket teszi:

- telepíti a szükséges függőségeket,
- letölti az előre megadott git linkről a projektet,
- a mappába lépve törli a korábbi buildelést, ha van,
- ha van `CMakeLists.txt` fájl, futtatja a `cmake` parancsot,
- ha van `autogen.sh` fájl, futtatja azt,
- ha van `buildconf` fájl, futtatja azt,
- ha van `configure` fájl, futtatja azt.

Fontos megjegyezni, hogy egyik projektnél sincs szükség az összes fenti lépés elvégzésére: a különböző projektet különböző módon állítják be magukat. Van azonban olyan, ahol több lépéses a konfiguráció, ezért nem elegendő egy konfiguráló lépés elvégzése.

Ezek alapján könnyen kitalálható, hogy egy projekt hozzáadásánál milyen adatokat szükséges megadni az adott projektről. A tesztkörnyezethez jelenleg csak olyan projekt hozzáadására van lehetőség, mely a GNU Make [35] eszközzel buildelhető. A `clang-test-docker` főmappában található `add_project.sh` script végigvezeti a felhasználót egy projekt hozzáadásának lépésein. A script különböző kérdéseket tesz fel, mely során a következő lépéseket kell végrehajtani:

1. git link megadása, mely után a script automatikusan felismeri a projekt nevét, amivel később lehet hivatkozni a projektre;
2. projekt függőségeinek megadása, melyre az alábbi három módon van lehetőség (a három lehetőségből pontosan egyféleképpen):
   (a) szöveges fájl megadása (.txt), melyben a függőségek sortöréssel elválasztva vannak felsorolva. Fontos megjegyezni, hogy csak olyan függőségeket adhatók itt meg, melyek telepíthetőek az apt Debian Package Management [57] eszközzel;
   (b) függőségek megadása egy sorban, szóközzel elválasztva. Ennél a lépésnél is érvényes a 2.(a) pontban megadott kitétel a függőségekre;
   (c) amennyiben a függőségeket nem lehet az említett csomagkezelővel egyszerűen telepíteni, megadható egy ezt, azaz a függőségek telepítését elvégző speciális beállító script (.sh) elérése is;
3. amennyiben a projekt a `configure` paranccsal lesz inicializálva, de szükséges plusz argumentumokat megadni ennek, meg lehet tenni az argumentumok sortörés nélküli begépelésével;
4. amennyiben a projekt inicializálása nem a korábban felsorolt fájlok valamelyikével működik, megadható egy speciális konfiguráló script (.sh). Ez csak haladó programozóknak ajánlott. A scriptet majd a projekt főkönyvtárából futtatja a környezet.

Speciális script (2(c). pont) használata esetén ügyelni kell rá, hogy minden függőség telepítve legyen, valamint a script futása közben ne legyen szükség felhasználói interakcióra. Ez a gyakorlatban azt jelenti, hogy például az `apt-get install` használata önmagában nem megfelelő, kizárólag a `-yqq` kapcsolóval együtt.

A beállítások elvégzésével a projekt adatai elmentődnek. Ezután használható a hozzáadott projekt is. Az elemzés működése az újonnan hozzáadott projektre csak abban az esetben

garantált, ha minden konfigurálási és függőségi adat megfelelően lett megadva, és a projekt a make paranccsal buildel, amit a főkönyvtárából kell futtatni.

Példa: A Secure Reliable Transport (SRT) projekt hozzáadása a környezethez

A következőkben egy példán keresztül bemutatom, miként lehet új projektet hozzáadni a tesztkörnyezethez. Első lépésként választani kell egy projektet, aminek a buildelése a `make` paranccsal történik, és megfelelően dokumentált a Linuxon való telepítése. A példaprojekt a Secure Reliable Transport (SRT) [58].

A megfelelő mappába lépés után, futtatni kell az `add_project.sh` fájlt a 2.51. ábrán látható módon. Amennyiben szükséges, a futtatási jogot be kell beállítani. Ezután a git linkjét a projektnek[8] be kell másolni a 2.52 ábrán latható módon, ezután ismét entert ütni. A script ekkor automatikusan ellenőrzi a link elérhetőségét, valamint a linkből kikövetkezteti a projekt nevét.

```
1 $ ./add_project.sh
2 Adding project to test docker...
3 Please, paste the git link of the project here:
```

2.51. ábra. Projekt hozzáadása a tesztkörnyezethez

```
1 Adding project to test docker...
2 Please, paste the git link of the project here: \
3 https://github.com/Haivision/srt.git
4 Repository checked.
5 srt project was saved.
```

2.52. ábra. Projekt linkjének megadása

A projekt függőségeinek megadásakor a github linken található leírásból lehet kiindulni. A példa során a függőségek nem txt fájlként kerülnek megadásra, ezért az első, 2.53. ábrán is látható kéréskor entert kell ütni, majd szóközzel elválasztva beírni a szükséges függőségeket. Ezután ismét entert kell ütni.

A következő kérdésnél meg kell vizsgálni, hogy a konfiguráláshoz, buildeléshez milyen további műveletek szükségesek a projekt beállítása során. Ott az található, hogy teljes buildeléshez a 2.54. ábrán látható parancsokat kell futtatni. Az `apt-get update` és `upgrade`

[8] A jelenlegi verzióban kizárólag HTTPS-en keresztül van lehetőségünk a projektek letöltésére, SSH-n keresztül nincs.

```
1 Dependency file (.txt) destination:
2 Dependencies separated by space: tclsh pkg-config cmake \
3 libssl-dev build-essential
```

2.53. ábra. Projekt függőségeinek megadása

parancsait a környezet mindig automatikusan elvégzi, a dependenciák pedig az imént kerültek megadásra. Mivel a konfigurálás a `configure` paranccsal történik, ezt a konténerben futó szkript automatikusan felismeri majd, ezért nem szükséges megadni. A `make` parancs futtatása már a CodeCheckerre feladata, tehát a hozzáadásnál nincs több teendő. A további kérdéseinél az éppen futó `add_project.sh` szkriptnek entert kell ütni.

```
1 sudo apt-get update
2 sudo apt-get upgrade
3 sudo apt-get install tclsh pkg-config cmake libssl-dev \
4     build-essential
5 ./configure
6 make
```

2.54. ábra. Az SRT projekt konfigurálása és buildelése

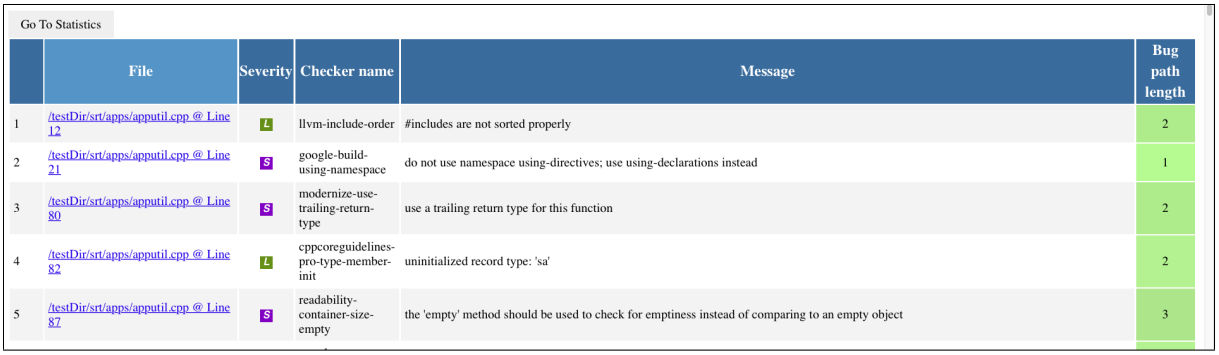
A script befejeződése után bemutatom a projekt használatát. Ehhez egy speciális konténer létrehozása javasolt, majd ennek futtatása, hogy megbizonyosodjon a felhasználó az új projekt helyes működéséről a 2.55. ábrán látható módon. A parancsok futtatása után sikeresek a tesztek, az eredmény HTML fájl a 2.56. ábrán látható. Tehát a projekt hozzáadása működött, és nem lépett fel hiba közben.

```
1 $ docker build -t test-srt . --build-arg projects=srt
2 $ docker run -v $testingRootDir:/testDir \
3 >   -v $llvm:/llvm-project test-srt
```

2.55. ábra. Az SRT projekt tesztelésének kipróbálása a tesztkörnyezetben

2.7.4. Projekt törlése

Amennyiben rosszul került hozzáadásra egy projekt a környezethez, lehetőség van annak végleges törlésére. Fontos azonban megjegyezni, hogy ezzel a projekt használhatatlanná válik



Go To Statistics

	File	Severity	Checker name	Message	Bug path length
1	/testDir/srt/apps/apputil.cpp @ Line 12	L	llvm-include-order	#includes are not sorted properly	2
2	/testDir/srt/apps/apputil.cpp @ Line 21	S	google-build-using-namespace	do not use namespace using-directives; use using-declarations instead	1
3	/testDir/srt/apps/apputil.cpp @ Line 80	S	modernize-use-trailing-return-type	use a trailing return type for this function	2
4	/testDir/srt/apps/apputil.cpp @ Line 82	L	cppcoreguidelines-pro-type-member-init	uninitialized record type: 'sa'	2
5	/testDir/srt/apps/apputil.cpp @ Line 87	S	readability-container-size-empty	the 'empty' method should be used to check for emptiness instead of comparing to an empty object	3

2.56. ábra. Az SRT projekt elemzésének eredménye (részlet)

minden később buildelt image számára! Ez tehát nem összekeverendő a projekt mappájának törlésével, mely során a mappa törlődik, de a projekt később újra beállítható.

Egy projekt törlésére nagyon egyszerűen van lehetőség. Ehhez a `clang-test-docker` főmappába lépve, futtatni kell a `delete_project.sh` scriptet. Ezután a projekt pontos nevét megadva a projekt minden adata törlődik. Törlésre kerül tehát a projekt git linkje, a függőségeit leíró fájlok, valamint a konfigurálását leíró fájl, ha van. Az imént hozzáadott SRT projekt törlésére a 2.57. ábra szerint van lehetőség. A script futtatása közben az ábrán is látható visszajelzések kerülnek kiírásra.

```
1 $ ./delete_project.sh
2 Deleting project on test docker...
3 Please, type the name of project you would like to delete: srt
4 Deleting...
5 srt deleted succesfully!
```

2.57. ábra. Az SRT projekt végleges törlése

Amennyiben nem létező projektet kísérel meg a programozó törölni, a script a 2.58. ábrán látható hibajelzést adja. Ebben az esetben természetesen semmi sem kerül törlése.

```
1 $ ./delete_project.sh
2 Deleting project on test docker...
3 Please, type the name of project you would like to
4 delete: srtt
5 TestEnv Error: srtt is not a valid project name
```

2.58. ábra. Az SRT projekt végleges törlésének sikertelensége

Projekt törlésénél a projekt mappája nem, csupán az őt leíró, tesztkörnyezetben lévő fájlok törlődnek. A projekt törlése csupán új image buildelésekor lép életbe.

2.7.5. Futási idejű üzenetek

A tesztkörnyezet használata közben akár az image buildeléséről, akár projekt hozzáadásáról, akár a tesztek futtatásáról van szó, folyamatosan visszajelzést kap a felhasználó a standard outputon, azaz gyakorlatilag a parancssorban. Az üzenetek alapvetően két nagy csoportba oszthatók: a tesztkörnyezet által generált üzenetekre, valamint a tesztkörnyezet által futtatott parancsok által generáltakra. A bemutatás során elsősorban az előbbivel foglalkozok, de több helyen szót ejtek az utóbbiról is. A tesztkörnyezet által adott visszajelzések minden esetben `TestEnv` előtaggal kezdődnek (pl.: 2.59. ábra), ezzel segítve a kezdő programozót a környezet használatában. Fontos kiemelni, hogy sok esetben a hibaüzeneten kívül segítség is megjelenik a hibaüzenethez kapcsolódóan az outputon. Mivel ezek a segítségek a további táblázatban egyébként is szerepelni fognak, ezekre részletesen itt nem térek ki.

```
1 TestEnv: Installing dependencies for codechecker project...
```

2.59. ábra. Részlet a tesztkörnyezet futási idejű visszajelzéseiből

Az üzenetek általában információt adnak az éppen zajló folyamatról. Mivel a buildelés, konfigurálás, valamint tesztelés műveletei egyarán hosszú időt vehetnek igénybe, ezért a felhasználó számára fontos jelezni, hogy éppen hol tart a folyamat.

A következőkben áttekintem, hogy a korábban említett lépések, tevékenységek közben milyen üzenetekkel találkozhat a felhasználó. Ezen kívül bemutatom, hogy egyes visszajelzéseket tapasztalva van-e teendője a programozónak, és ha igen, akkor mi az.

Futási idejű visszajelzések: az image buildelése

Az image buildelését elindítva a programozó visszajelzést kap ennek lépéseiről. A környezet visszajelzései alapvetően három csoportba oszthatók. Az első a tájékoztatás, mely során információt jelennek meg a folyamat menetéről; a második a figyelmeztetés, ami esetleges hibára utalhat; valamint a harmadik a hiba, ami fontos problémára hívja fel a figyelmet. Utolsó esetben az image buildelése sikertelen.

Az image buildelése során egyetlen fajta, a környezettől származó tájékoztató üzenet olvasható, mégpedig `TestEnv: Installing dependencies for <p> project...` szöveggel. Ez azt jelenti, hogy a p nevű projekt függőségeinek telepítése folyamatban van. Ezenkívül a docker folyamatos visszajelzést ad a buildelés lépésiről, ami sikeres esetben a `Successfully`

`built` és a `Successfully tagged` kimenetekkel tér vissza, melyek után a konténer neve és azonosítója olvasható.

A figyelmeztető- és hibajelzések a 2.6. táblázatban olvashatók. Hibajelzések esetén a konténer felépítése sikertelen. Az errorok kódja az az érték, amivel a hibásan végetérő program visszatér. Ez kizárólag továbbfejlesztésnél fontos.

Üzenet	**Visszajelzés típusa**	**Kód**	**Üzenet leírása**	**Ajánlott ellenőrizni**
TestEnv Warning: ./requirements/ <p>_debian.txt not exist!	Warning	-	A projektnek nem talált függőségeket leíró fájlját.	projektnek van-e függősége
TestEnv Error: You must add at least one project!	Error	1	Egyetlen projekt sem került megadásra	projects környezeti változó
TestEnv Error: project_links.txt file is not found!	Error	2	A projektek linkjeit tartalmazó fájl nem található.	-
TestEnv Error: requirements folder is not found!	Error	3	A projektek függőségeit tartalmazó mappa nem található.	-
TestEnv Error: <p> is not a valid project name	Error	4	A megadott <p> projektnév nem létezik.	projekt megadott neve
TestEnv Error: Installing dependencies failed. Please, try again.	Error	5	A környezet alapvető függősségeinek telepítése sikertelen.	internetkapcsolat
TestEnv Error: Installing dependencies failed. Check your custom dependency file for <p>.	Error	6	A <p> projekt speciális beállító fájljának futtatása sikertelen.	projekt beállítását végző speciális script
TestEnv Error: Installing dependencies failed. Make sure that you used the right dependency name when setting up <p>	Error	7	A <p> projekt függőségeinek telepítése sikertelen.	függőségek telepíthetőek az apt-get install paranccsal

2.6. táblázat. Az image buildelése során fellépő figyelmeztető üzenetek

Futási idejű visszajelzések: a projektek konfigurálása

A projektek konfigurálásánál folyamatosan az outputra íródnak a különböző műveletek kimenetei. Ilyenek például a `git clone`, a `configure`, vagy a CodeChecker által végzett `make`

parancsok kimenetei. Ahogy korábban, itt is találkozhat a programozó a tesztkörnyezet által adott speciális, `TestEnv` kezdetű visszajelzésekkel. Az általános, tájékoztatást elősegítő üzenetek a 2.7. táblázatban olvashatók. A figyelmeztető, valamint hibát jelző üzenetek pedig a 2.8. táblázatban találhatók. Hiba esetén a működés automatikusan leáll, és a várt művelet sikertelen. Egyes hibák könnyen kijavíthatóak (pl.: projekt nevének helytelen megadása), azonban egyesek az image teljes újrabuildelését, esetleg a tesztkörnyezet újbóli beállítását követelhetik. A `TestEnv Error: project_links.txt file is not found!` error esetén például ajánlott a teljes környezet törlése, majd ismételt letöltése, míg ha egy projekt függőségeivel vagy beállításával van probléma, ajánlott a projekt korábban leírt törlése, majd újbóli, most már megfelelő hozzáadása.

Üzenet	**Üzenet leírása**
TestEnv: Setting up projects...	A projektek beállításának (letöltésének majd konfigurálsáának) kezdése
TestEnv: Checking <p> directory...	A <p> projekt mappájának ellenőrzése (letöltése vagy frissítése)
TestEnv: Configuring <p>...	A <p> projekt konfigurálása
TestEnv: Special config file is found	A projekt speciális scripttel lesz konfigurálva
TestEnv: CMakeLists.txt is found	A projekt CMake paranccsal lesz konfigurálva
TestEnv: autogen.sh is found	A projekt autogen.sh scripttel lesz konfigurálva
TestEnv: buildconf is found	A projekt builconf paranccsal lesz konfigurálva
TestEnv: configure file is found	A projekt configure paranccsal lesz konfigurálva
TestEnv: configure argument file is found	A projekt configure parancsa speciális argumentummal lesz futtatva
TestEnv: Configuring up CodeChecker...	CodeChecker konfigurálása
TestEnv: Running CodeChecker log on <p>...	A CodeChecker log parancsának futtatása a <p> projekten

2.7. táblázat. A projektek beállítása során fellépő tájékoztató üzenetek

Futási idejű visszajelzések: a projektek elemzése

A projektek tesztelése közben is számos üzenet olvasható a standard outputon. Ilyen például a projektek buildelésének menete, melynek egy részlete a 2.60. ábrán látható, vagy a checkerek futásáról való visszajelzés, ami pedig a 2.61. ábrán tekinthető meg.

Ezen kimeneteken túl ismételten a tesztelő környezet is ad visszajelzést. A tájékoztató visszajelzések a 2.10. táblázatban olvashatóak. A 2.8. táblázatban található első négy

Üzenet	**Üzenet leírása**	**Ajánlott ellenőrizni**
TestEnv Warning: setup argument is neither TRUE nor FALSE. Default TRUE value is applied.	A setup környezeti változó beállítása nem megfelelő. Alkalmazott érték: TRUE.	setup környezeti változó
TestEnv Warning: analyze argument is neither TRUE nor FALSE. Default TRUE value is applied.	Az analyze környezeti változó beállítása nem megfelelő. Alkalmazott érték: TRUE.	analyze környezeti változó
TestEnv Warning: delete argument is neither TRUE nor FALSE. Default FALSE value is applied.	A delete környezeti változó beállítása nem megfelelő. Alkalmazott érték: FALSE.	delete környezeti változó
TestEnv Warning: list argument is neither TRUE nor FALSE. Default FALSE value is applied.	A list környezeti változó beállítása nem megfelelő. Alkalmazott érték: FALSE.	list környezeti változó
TestEnv Warning: there is no git link for <p>	A <p> projekthez nem található alapértelmezett git link.	projekt letöltését valóban speciális beállító script végzi

2.8. táblázat. A projektek buildelése során fellépő figyelmeztető üzenetek

```
1  TestEnv: Running CodeChecker log on curl...
2  [INFO 2020-04-28 16:53] - Starting build ...
3  Making all in lib
4  make[1]: Entering directory '/testDir/curl/lib'
5  make  all-am
6  make[2]: Entering directory '/testDir/curl/lib'
7    CC       vauth/libcurl_la-cleartext.lo
8    CC       vauth/libcurl_la-cram.lo
9    CC       vauth/libcurl_la-krb5_sspi.lo
10   CC       vauth/libcurl_la-digest.lo
11   CC       vauth/libcurl_la-digest_sspi.lo
12   CC       vauth/libcurl_la-krb5_gssapi.lo
```

2.60. ábra. A curl projekt buildelésének kimenete (részlet)

figyelmeztetést a tesztelés során is megkapható. Az errort, azaz hibát jelző üzeneteket, valamint a szükséges teendők teszt futtatása esetén megegyeznek a 2.9. táblázatban szereplő első három üzenettel. Itt továbbra is igaz, hogy a megoldáshoz lehetséges, hogy újra kell buildelni az image-et. A korábbiakon túl egy fontos error üzenetről kell még szót ejteni. A `estEnv Error: invalid checker name: <c>` üzenet esetén a `checker` környezeti változóban megadott `c` checker nem létezik. Ebben az esetben ellenőrizni kell, hogy helyesen lett-e megadva a neve, valamint hogy létezik-e ilyen checker!

Üzenet	**Kód**	**Üzenet leírása**	**Ajánlott ellenőrizni**
TestEnv Error: <p> is not a valid project name	3	A megadott <p> projektnév nem létezik.	<p> gépelése, <p> hozzá lett adva az image buildelése során
TestEnv Error: /llvm-project is not found!	15	Nem található csatolt llvm-project mappa.	llvm-project megfelelő csatolása
TestEnv Error: /testDir is not found!	15	Nem található csatolt testDir mappa.	tesztelő mappa megfelelő csatolása
TestEnv Error: cloning <p> is failed.	5	A <p> projekt letöltése a megadott címről sikertelen.	internetkapcsolat, megfelelő git link
TestEnv Error: <p>_setup.sh failed.	5	A <p> projekt konfigurálása a speciális script segítségével sikertelen.	script helyessége, függőségek
TestEnv Error: CMake failed during configuring <p>.	7	A projekt konfigurálása a CMake segítségével sikertelen.	függőségek
TestEnv Error: autogen.sh failed during configuring <p>.	8	A projekt konfigurálása autogen.sh segítségével sikertelen.	függőségek
TestEnv Error: buildconf failed during configuring <p>.	9	A projekt konfigurálása buildconf script segítségével sikertelen.	függőségek
TestEnv Error: configure failed during configuring <p> with additional arguments.	10	A projekt konfigurálása configure script segítségével, speciális argumentumokkal sikertelen.	függőségek, configure argumentumai
TestEnv Error: configure failed during configuring <p>.	11	A projekt konfigurálása configure script segítségével sikertelen.	függőségek
TestEnv Error: Configuring CodeChecker failed.	4	A CodeChecker konfigurálása sikertelen.	-
estEnv Error: clang or clang-tidy binaries are not found.	13	A Clang vagy Clang-Tidy futtatható fájljai nem találhatóak.	llvm-project megfelelő csatolása, llvm-project megfelelő buildelése

2.9. táblázat. A projektek buildelése során fellépő hibaüzenetek

Üzenet	**Üzenet leírása**
TestEnv: Checking checker names...	A környezet a megadott checkerek létezését ellenőrzi
TestEnv: Setting up environment...	A környezet megfelelő beállítása a CodeChecker számára
TestEnv: Running analyzis...	Elemzés futtatása
TestEnv: Running CodeChecker analyze on <p>...	<p> projekt elemzése

2.10. táblázat. Az projektek elemzése során fellépő tájékoztató üzenetek

```
TestEnv: Setting up environment...
TestEnv: Running analyzis...
TestEnv: Running CodeChecker analyze on curl...
[INFO 2020-04-28 16:55] - '--enable-all' was supplied for
this analysis.
[INFO 2020-04-28 16:55] - Starting static analysis ...
[INFO 2020-04-28 16:55] - [1/331] clang-tidy analyzed
.c successfully.
[INFO 2020-04-28 16:55] - [2/331] clang-tidy analyzed
nss.c successfully.
[INFO 2020-04-28 16:55] - [3/331] clang-tidy analyzed
digest_sspi.c successfully.
[INFO 2020-04-28 16:55] - [4/331] clang-tidy analyzed
spnego_sspi.c successfully.
[INFO 2020-04-28 16:55] - [5/331] clang-tidy analyzed
ntlm.c successfully.
[INFO 2020-04-28 16:55] - [6/331] clang-tidy analyzed
krb5_gssapi.c successfully.
```

2.61. ábra. A curl projekt elemzésének kimenete (részlet)

Egyéb futási idejű üzenetek

A konténer futtatása során néhány további visszajelzéssel is találkozhat a programozó, melyek többféle futtatási módra is jellemzőek. Ezek a 2.11. táblázatban találhatók. Az első oszlopban az olvasható, hogy melyik környezeti változó értéke igaz, azaz melyik futtatási módnál lehet találkozni az adott üzenetekkel.

Projekt hozzáadásánál (`add_project.sh`) és törlésénél (`delete_project.sh`) keletkező futási idejű üzenetek a használat alapját képezik. Mivel ezeket bemutattam a 2.7.3. és 2.7.4. fejezetekben, ezekre itt nem térek ki.

2.7.6. Egyéb futtatással kapcsolatos tudnivalók

A futtatások során fontos tudnia a felhasználónak néhány egyéb, a korábbiakhoz kapcsolódó tulajdonságról. A checkerek önálló futtatása egy-egy fájlon (2.6. alfejezet) gyors, maximum néhány másodperces feladat, ezért ott a futási idejű megszakításról nem volt értelme beszélni. Ezzel szemben a tesztkörnyezettel kapcsolatban már nem ez a helyzet. Fontos tudni, hogy a tesztkörnyezet minden lehetséges művelete közben elméletileg van lehetőség a művelet megszakítására a szokásos módon (POSIX rendszerekben Ctrl + C billentyűkombináció). Erre azért van szükség, mivel a legtöbb folyamat (konfigurálás, letöltés, tesztek futtatása) hosszú

Üzenet előfordulása	**Üzenet**	**Üzenet leírása**
setup, analyze	TestEnv: Setting up permissions...	A projekthez tartozó jogosultságok beállítása
setup, analyze, delete	TestEnv: Done!	A műveletek sikeresen végrehajtódtak
list	TestEnv: The container has been set up to test the following projects:	Projektek kilistázása
delete	TestEnv: Deleting projects...	Projektek mappáinak törlése
delete	TestEnv: Deleting <p>...	<p> projekt mappájának törlése

2.11. táblázat. Egyéb futási idejű visszajelzések

időt vehet igénybe, ezért a felhasználónak bármikor engedélyezni szeretném a műveletek megszakítását. Ilyen megszakítás esetén az adott műveletek nem fejeződnek be sikeresen, azonban hagyhatnak maguk után fájlokat. Éppen ezért a megszakítás úgy tekintendő, mintha error következett volna be: letöltés esetén újból el kell végezni azt, konfigurálásnál és a tesztek futtatásánál pedig hasonlóan. Az esetlegesen fennmaradt átmeneti fájlok (például egy feléig lefutott elemzés átmeneti fájljai) a következő azonos futtatásnál automatikusan törlődnek, ezért ezekkel nem szükséges a felhasználónak foglalkoznia.

2.7.7. Hibakezelés

A tesztkörnyezet működéséből adódó hibakezelésre a 2.7.5. alfejezetben bővebben kitértem, ahol a hibákat futási idejű hibaüzenetekkel párosítottam. Van azonban néhány, a környezet kódolásától független működés, melyet fontos ismernie a felhasználónak.

Függőségek sikertelen telepítése

Előfordulhat, hogy az image buildelése során a függőségek nem kerülnek telepítésre, és a buildelés után a standard kimeneten a 2.62. ábrán látható visszajelzés olvasható. Ebben az esetben a függőségek telepítése nem futott le újból, az image buildelése során ugyanis a docker a cacheben tárolt adatokra támaszkodott.

Amennyiben ezután nem megfelelő működést tapasztal a felhasználó, használja a 2.63. ábrán is látható `--no-cache` kapcsolót az image buildelése során. Ebben az esetben letiltja a cache használatát, és biztosan lefutnak a várt parancsok.

```
1 Step 18/19 : RUN bash install_deps.sh $projects
2  ---> Using cache
3  ---> a1f0ff11dce9
4 Step 19/19 : CMD ["bash", "-c", "./start.sh ${setup}
5 ${analyze} ${checkers} ${delete} ${list} ${projects}" ]
6  ---> Using cache
7  ---> 17be6f4c8b88
8 Successfully built 17be6f4c8b88
9 Successfully tagged test-clang:latest
```

2.62. ábra. Cache használata az image buildelése során

```
1 $ docker build -t test-clang . \
2 > --build-arg projects=FiberTaskingLib
3 > --no-cache
```

2.63. ábra. Cache használatának letiltása az image buildelése során

Buildelés vagy elemzés leállítása sikertelen

Egyes esetekben az tapasztalható, hogy a konténer futását nem lehet leállítani a szokásos Ctrl+C billentyűkombinációval. Ezt a tulajdonság sajnos a konténer működéséből adódik. Ennek a megoldására - amennyiben mindenképpen le szeretné állítani a programozó a futást - nyitnia kell egy új terminált. Abból futtatnia szükséges a 2.64. ábrán látható parancsot, ahol ellenőrizheti a konténer azonosítóját (ID) az első sor parancsával. Ezután a második sorban található paranccsal leállásra kényszerítheti a futó konténert. Ebben az esetben természetesen a konténerben végzett művelet sikertelen.

```
1 $ docker ps
2 $ docker stop <id>
```

2.64. ábra. A futó konténer kényszerített leállítása

Projekt konfigurálása vagy buildelése sikertelen

Ha a projekt konfigurálása vagy buildelése hibába torkollik, először minden esetben ellenőrizni kell, hogy telepítetve lettek-e a megfelelő függőségek, azaz az image buildelésére megfelelően került sor. Ha kézzel lettek hozzáadva a függőségek, ellenőrizni kell ezek megfelelő hozzáadását is.

Vannak azonban egyes esetek, amikor egy új release kiadásával változik a letöltött projekt konfigurálásának menete. Elképzelhető például, hogy egy projekt az egyik release esetén még beállítható a `cmake` paranccsal, a következőben viszont már nem.

Az ilyen esetek elkerülése érdekében, ha a függőségekkel kapcsolatban nem található hiba, a projektet törölni kell teljesen a tesztkörnyezetből (`delete_project.sh`), majd hozzáadni ismét (`add_project.sh`). Hozzáadás előtt azonban ellenőrizni kell, hogy az újonnan megadott függőségek és a konfigurálás módja működik az új release esetén.

Ismét egy nem túl gyakori eset, hogy a projekt `cmake` és `configure` paranccsal is konfigurálható, azonban a használt rendszeren kizárólag az egyik működik. Ebben az esetben szükséges megadni speciális beállító scriptet a projekthez, amiből a megfelelő, működő parancsot kell futtatni.

3. fejezet

Fejlesztői dokumentáció

A fejlesztői dokumentáció során bemutatom a megírt checkerek és a tesztkörnyezet belső felépítésének tervét, a működés tervét, valamint a megvalósítást és annak okait. A fejlesztői dokumentáció segítséget nyújthat a programban lévő kisebb módosítások elvégzéséhez, illetve a checkerek vagy a konténer további fejlesztéséhez. Ahogy az a korábbi fejezetekből kiderült, az elkészített munka két nagy csoportra bontható: a Clang-Tidy statikus elemzőhöz tartozó checkerekre és a létrehozott tesztekörnyezetre, mely a tesztelést hivatott elősegíteni.

3.1. Fogalomtár

A felhasználói dokumentáció épít a fejlesztői dokumentációban megismert fogalmakra (2.2. alfejezet). Ezentúl használni fogom a következő kifejezéseket is:

- **AST / Absztrakt szintaxis fa:** a fordítási lépések közül a szemantikus elemzést követő szintaktikus elemzés eredménye. A dolgozatomban ez gyakorlatilag az LLVM Clang fordító által generált absztrakt szintaxis fát jelenti.
- **Checker:** a Clang-Tidy statikus elemző azon része, mely egy hiba, sebezhetőség felismerését végzi. A dolgozatomban ez egy-egy C++ és a hozzá tartozó header fájlt jelenti.
- **Matcher:** olyan program vagy parancs, ami egy adott adathalmazban megkeresi egy adott mintára illeszkedő elemeket. Munkám során az AST csúcsaira fogok matchereket írni, amik megtalálják az adott tulajdonságú csúcsot vagy csúcsokat.

3.2. Specifikáció és követelmények

A fejezet során részletesen áttekintem, hogy mit várok el a leendő fejlesztésektől, valamint ehhez milyen eszközökre, szoftverekre lesz szükségem. A specifikáció során kitérek mind a funkcionális, mind a nem funkcionális követelményekre.

3.2.1. Célcsoport

A célcsoport a checkerek esetén C++ programozók, egészen a kezdőktől a seniorokig, hiszen a statikus elemzés, ellenőrzés, minden generáció minden szintjének segíthet jobbá, biztonságosabbá tenni a programját. A tesztkörnyezetet elsősorban Clang-Tidy fejlesztők számára hoztam létre, akiknél feltételeztem a C++ ismeretének és a parancsor használatának alapvető ismeretét. A működtetés, használat gyakorisága eltérő lehet: a mindennapos többszöri használattól egészen a havonta egyszeri használatig.

3.2.2. A checkerek specifikációja

A kitűzött feladat, hogy a következő öt programozási szabálynak nem megfelelő C vagy C++ nyelven írt kód automatikusan felismerésre kerüljön a Clang-Tidy [3], és az abba fejlesztett új checkerek segítségével:

- POS44-C. Do not use signals to terminate threads [22]
- CON54-CPP. Wrap functions that can spuriously wake up in a loop [23]
- CON36-C. Wrap functions that can spuriously wake up in a loop [24]
- CON37-C. Do not call `signal()` in a multithreaded program [25]
- CON40-C. Do not refer to an atomic variable twice in an expression [26]

A megírt checkerek az LLVM Clang-Tidy statikus elemző részét fogják képezni. Tehát a végső programban is a megfelelő kapcsolókkal ellátott elemző kerül majd használatra.

Meghívandó parancs paraméterekkel:

```
clang-tidy -checks=-*,<letrehozottCheckerek> <inputFájl>
```

Bemenet: vizsgálni kíván C vagy C++ fájl.

Kimenet: a korábbi szabályoknak nem megfelelő fájl esetén a megadott hibajelzés (warning), egyébként semmi.

A specifikáció meghatározásánál fontos leszögezni, hogy a lehető legjobb megoldásra törekszem, azonban a Clang-Tidy eszköz korlátait figyelembe véve. Előfordulhatnak ugyanis olyan problémák, amikhez az adatáramlás vizsgálatára lenne szükség (data flow elemzés), amire a Tidy nem képes. Éppen ezért nem feltétlenül kell tökéletes megoldásra törekedni, csupán a lehetséges hibalehetőségek halmazát szeretném csökkenteni.

3.2.3. A tesztkörnyezet specifikációja

A munka során egy olyan rendszert szeretnénk létrehozni, mely automatikusan teszteli az LLVM Clang-Tidy [3] statikus elemző checkereit nagy, nyílt forráskódú, git könyvtárral rendelkező, előre megadott projekteken a CodeChecker [14] eszköz segítségével. Ehhez elvárom, hogy egyes projekteket automatikusan letöltsön és konfiguráljon, automatikusan buildeljen, valamint legyen lehetőség a projektek elemzésére. Ehhez természetesen a projekteknek különböző függőségekre van szükségük, melyek telepítését ugyancsak elvárom a leendő eszköztől. Mindezeket úgy szeretném végrehajtatni, hogy a felhasználó által a lehető legkevesebb energiabefektetésre legyen szükség. Szeretném, hogy egyes projekteket a felhasználó bármilyen hozzáértés nélkül tesztelhessen. Haladó felhasználók számára azonban szeretném lehetővé tenni, hogy tudjon projektet hozzáadni az elemzéshez, valamint szükség szerint törölni azokat.

Az eszköztől elvárt funkcionális követelmények tehát összefoglalva a következők:

- előre megadott projektek függőségeinek automatikus telepítése,
- előre megadott projektek automatikus letöltése és konfigurálása,
- előre megadott projektek automatikus statikus elemzése a Clang-Tidy eszközzel,
- új projekt, és annak függőségeinek és beállításainak hozzáadása a környezethez,
- projekt teljes törlése a környezetből,
- amennyiben bármely művelet sikertelen (konfigurálás, buildelés, elemzés) a tesztkörnyezet álljon le és adjon hibajelzést.

Nem funkcionális követelményként elvárom, hogy a környezet legalább olyan gyorsan működjön, mintha a a felsorolt műveleteket a programozó kézzel hívná meg. Ezeken kívül megkövetelem, hogy a környezet mérete lényegesen ne haladja meg a projektek és függőségeik által feltétlenül szükséges méretet. Bár a felhasználó számítógépén a függőségeknek valamilyen formában meg kell jelenniük, szeretném, ha ezek minél kevésbé szemetelnék össze a számítógépét, és bármikor egyszerűen törölhetők lennének.

3.2.4. Rendszer és hardver követelmények

A rendszer és hardver követelmények megegyeznek 2.4. alfejezetben leírtakkal. A fejlesztéshez ezeken kívül kizárólag egy szövegszerkesztő van szükség. A 2.5. alfejezet alapján beállított és buildelt llvm-re, valamint a letöltött tesztkörnyezetre hivatkozni fogok egyes magyarázatok és tervezések során, ezért fejlesztés előtt mindenképpen ajánlatos részletesen elvégezni a felhasználói dokumentációban leírtakat.

3.2.5. Külső követelmények

A statikus elemző, valamint a checkerek és a tesztkörnyezet nem használnak személyes adatokat. Mivel az elemzés lokálisan vagy a konténerben kerülnek futtatásra, a használat során semmilyen adat vagy kódrészlet nem kerül az elemzőt vagy a környezetet fejlesztők birtokába.

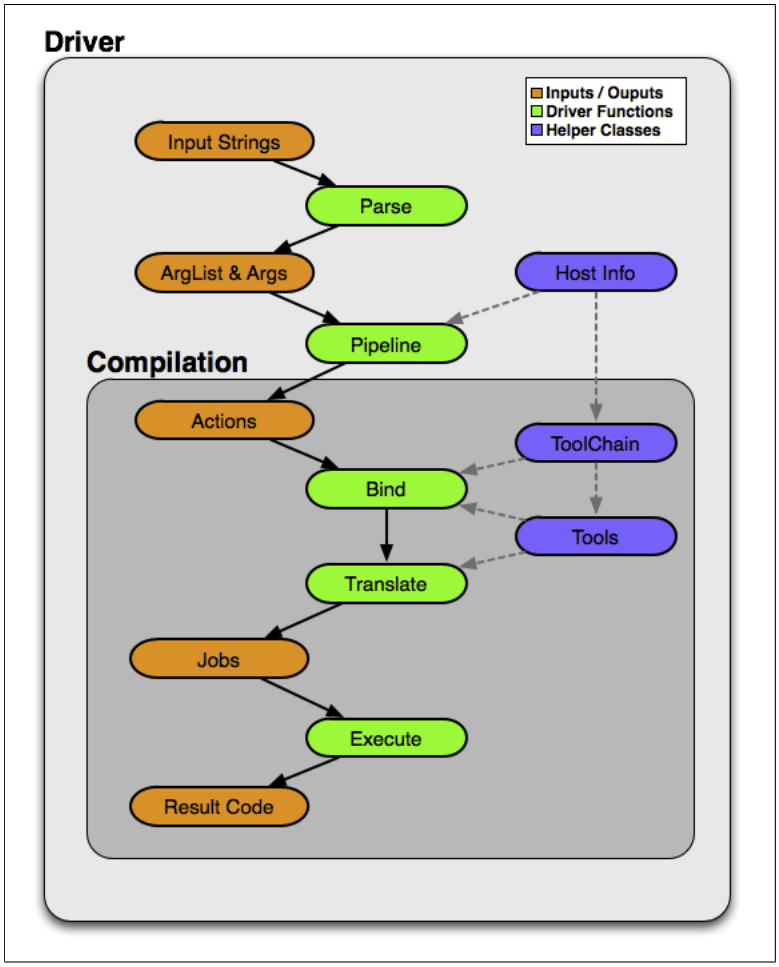
3.3. Szoftver architektúra

Az eszközök tervezését a lehető legtávolabbról kezdem. Az architektúrális tervezés során megvizsgálom, hogy a leendő checkereket és a tesztkörnyezetet milyen rendszerbe szeretném beilleszteni, milyen rendszer szolgál alapjául. Ezentúl meghatározom a nagy rendszer általam létrehozni kívánt komponenseit, és az ezek közti kapcsolatokat.

3.3.1. A checkerek architektúrája

Ahhoz, hogy saját checkert fejlesszen valaki Clang-Tidy-ban, először meg kell értenie magának az LLVM Clang fordítónak és a Tidy-nak az alapjait. Ezért szeretnék először egy rövid összefoglalót nyújtani ezekről a Clang [12] és a Clang-Tidy [59] dokumentációja alapján. A Clang egyszerűsített architektúradiagramja a 3.1. ábrán látható.

„A Clang projekt egy nyelvi frontend és eszköz infrastruktúra a C nyelvi család részére (C, C++, Objective C/C++, OpenCL, CUDA, és RenderScript) az LLVM projektben" [12]. A fordító használatának számos előnye van, a végzett statikus elemzés szempontjából a legfontosabb azonban az általa nyújtott hatékony diagnosztika. Ez azt jelenti, hogy a Clang a fordítás során hozzáférhetővé és ami még fontosabb, a felhasználó, programozó számára is érthetővé teszi az adott fordítási lépés kimeneteit [60]. Ez a szintaktikus elemzés kimeneteként kapott absztrakt szintaxis fánál (AST) is nagyban kihasználható.



3.1. ábra. A Clang fordító egyszerűsített architektúradiagramja [12]

A Clang-Tidy elemző a nevéből is adódóan a Clang fordítóra épülve működik. Legfontosabb előnye abban rejlik, hogy lehetővé teszi egyszerű elemzések elvégzését C és C++ kódokon. A Clang-Tidy checkerei a preprocesszor szinten vagy az AST szinten tudnak bekapcsolódni az elemzésbe. Bár legtöbbször az előbbit fogom használni, előfordulhat, hogy utóbbira is lesz szükségem.

Miután bemutattam a Clang és a Clang-Tidy alapjait, ejtek néhány szót a fejlesztés során talán legtöbbet használt eszközről, a clang-query-ről. Ezen eszköz segítségével a programozó egyszerűen betekintést nyerhet egy C vagy C++ program Clang által felépített absztrakt szintaxis fájába. Mivel a fejlesztés során az AST csúcsaira kell matchereket írni, az AST ismerete nélkülözhetetlen egy probléma vizsgálatánál. A következőkben a clang-query elemzése tekinthető meg a 3.2. programkódon. Egy beállítással és egy alapvető matcher írásával (3.3. ábra) könnyedén megtekinthető a fájl teljes AST-je, mely a 3.4. ábrán látható. A clang-query eszköz a telepítési útmutató során (2.5. alfejezet) automatikusan telepítésre került a felhasználó számítógépére.

```
1 int main(){
2     int a = 12;
3     long b = 24;
4     int c = a + b;
5     return 0;
6 }
```

3.2. ábra. A `test.cpp` példafájl

```
1 $ clang-query test.cpp --
2 clang-query> set output dump
3 clang-query> m translationUnitDecl()
```

3.3. ábra. A Clang-query futtatása, és alapvető matcher írása

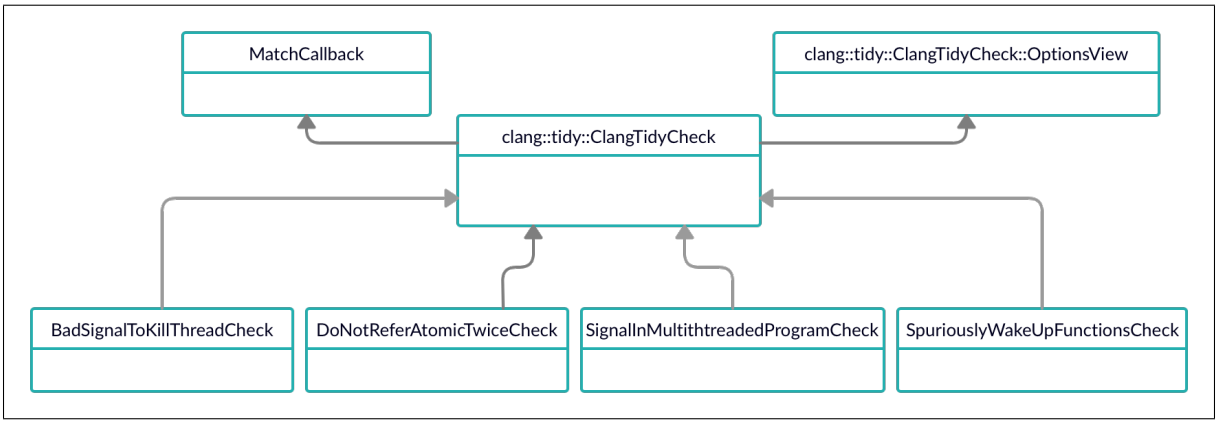
```
clang-query> set output dump
clang-query> m translationUnitDecl()

Match #1:

Binding for "root":
TranslationUnitDecl 0x185b018 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x185b8f0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x185b5b0 '__int128'
|-TypedefDecl 0x185b960 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x185b5d0 'unsigned __int128'
|-TypedefDecl 0x185bcc8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstantString
_tag'
| `-RecordType 0x185ba50 '__NSConstantString_tag'
|   `-CXXRecord 0x185b9b8 '__NSConstantString_tag'
|-TypedefDecl 0x185bd60 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x185bd20 'char *'
|   `-BuiltinType 0x185b0b0 'char'
|-TypedefDecl 0x189c168 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag [1]'
| `-ConstantArrayType 0x189c110 '__va_list_tag [1]' 1
|   `-RecordType 0x185be50 '__va_list_tag'
|     `-CXXRecord 0x185bdb8 '__va_list_tag'
`-FunctionDecl 0x189c210 </home/abelkocsis/thesis/test/test.cpp:1:1, line:6:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x189c630 <col:11, line:6:1>
    |-DeclStmt 0x189c3c0 <line:2:2, col:12>
    | `-VarDecl 0x189c338 <col:2, col:10> col:6 used a 'int' cinit
    |   `-IntegerLiteral 0x189c3a0 <col:10> 'int' 12
    |-DeclStmt 0x189c490 <line:3:2, col:13>
    | `-VarDecl 0x189c3f0 <col:2, col:11> col:7 used b 'long' cinit
    |   `-ImplicitCastExpr 0x189c478 <col:11> 'long' <IntegralCast>
    |     `-IntegerLiteral 0x189c458 <col:11> 'int' 24
    |-DeclStmt 0x189c5e8 <line:4:2, col:15>
    | `-VarDecl 0x189c4c0 <col:2, col:14> col:6 c 'int' cinit
    |   `-ImplicitCastExpr 0x189c5d0 <col:10, col:14> 'int' <IntegralCast>
    |     `-BinaryOperator 0x189c5b0 <col:10, col:14> 'long' '+'
    |       |-ImplicitCastExpr 0x189c598 <col:10> 'long' <IntegralCast>
    |       | `-ImplicitCastExpr 0x189c568 <col:10> 'int' <LValueToRValue>
    |       |   `-DeclRefExpr 0x189c528 <col:10> 'int' lvalue Var 0x189c338 'a' 'int'
    |       `-ImplicitCastExpr 0x189c580 <col:14> 'long' <LValueToRValue>
    |         `-DeclRefExpr 0x189c548 <col:14> 'long' lvalue Var 0x189c3f0 'b' 'long'
    `-ReturnStmt 0x189c620 <line:5:2, col:9>
      `-IntegerLiteral 0x189c600 <col:9> 'int' 0

1 match.
clang-query> _
```

3.4. ábra. A `test.cpp` programhoz tartozó absztrakt szintaxis fa

A tervezett Clang-Tidy checkerek tehát a következő algoritmus alapján fognak nagy vonalakban, az architektúra szintjén működni:

- a Clang fordító elkezdi a fájl fordítását, eljut az AST szintjéig,
- ekkor a Tidy az AST minden csúcsára megpróbál matchelni, azaz ellenőrzi, hogy az adott csúcs megfelel-e a matcherben megfogalmazott kitételeknek,
- amennyiben a matchelés sikeres volt, meghívja az adott checker `check()` függvényét, mely általában további ellenőrzéseket végez,
- amennyiben a további ellenőrzések alapján úgy kívánja, a checker figyelmeztetést jelezhet a felhasználó számára.

Az algoritmus alól kivételt képezhet, ha szükség van a preprocesszor fázisban végbemenő tulajdonságokra, amit a konkrét checkerek tervezésénél mutatok be. Ekkor a preprocesszor művelet ellenőrzésére speciális függvény használatára van lehetőség, majd minden a fenti algoritmus alapján folytatódik. Az általam fejlesztett checkerek a bugprone modul részét képezik. A további tervezésben a checkerek egy-egy komponenst (cpp és a hozzá tartozó header fájlt) formálnak majd, ezek architektúra terve a 3.5. ábrán látható. Az ábrán nem jelöltem a további felmenő osztályokat, valamint az osztályok függvényeit. Előbbi ugyanis nem lesz fontos számomra, utóbbiról pedig az egyes komponensek esetén fogok szót ejteni.



3.5. ábra. A checkerek architektúrájának terve

Fontos ezentúl megjegyezni, hogy minden checker fejlesztése során el kell majd végezni az alábbi műveleteket is:

- checker header fájljában rövid dokumentáció írása komment formájában,
- `ReleaseNotes.txt` fájlban a végzett módosítások rövid leírása,

- checkerek dokumentálása,
- checkerek aliasainak létrehozása, és azok dokumentálása.

A checkerek aliasainak létrehozására azért van szükség, mert jelenleg a SEI CERT lista problémáit oldom meg, aminek külön modulja van a Clang-Tidy-n belül.

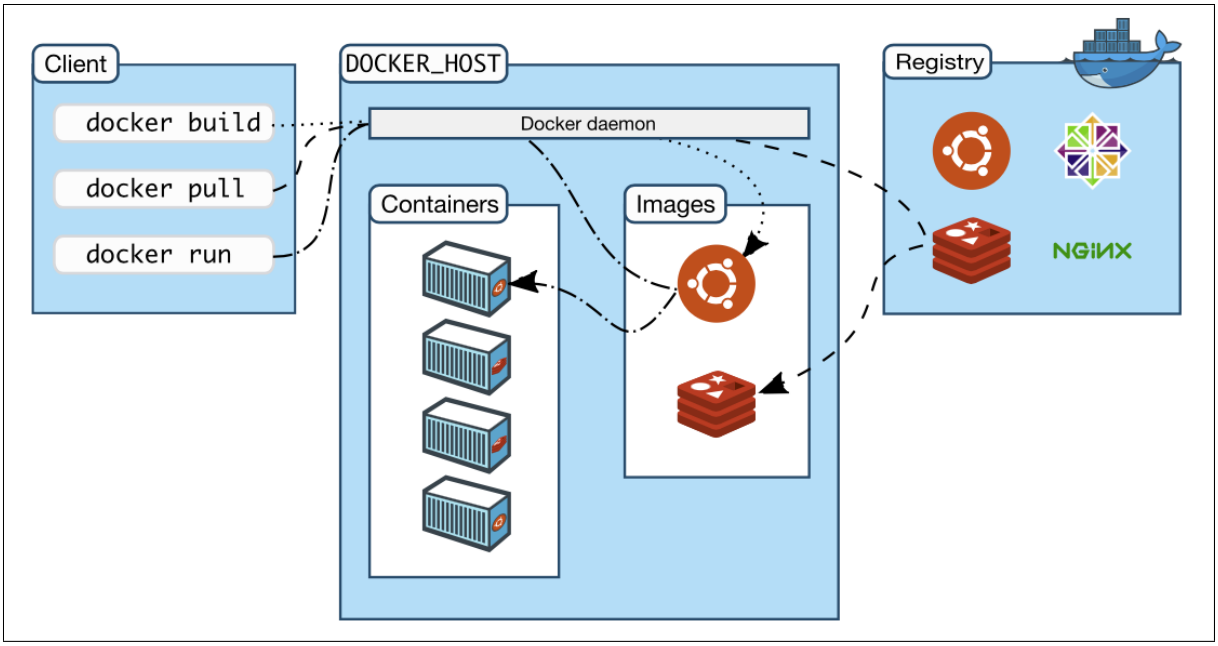
3.3.2. A tesztkörnyezet architektúrája

A tesztkörnyezet esetén a specifikációban leírt függőségek telepítése és a projektek konfigurálása és buildelése nagyon időigényes folyamat. Könnyen átgondolható, hogy ha egy checker fejlesztése közben az újabb és újabb verzióit le szeretné futtatni a programozó egy nagyobb projekten többször, akkor ha a tesztkörnyezet minden alkalommal letöltene, telepítene minden függőséget, majd letöltene és buildelne minden projektet, majd ezeket mind törölné, és legközelebbi futtatás esetén elölről kezdené ezeket a műveleteket, az nagyon sok időbe és energiába kerülne. Ez nagyon pazarló és felesleges megoldás lenne. Megoldás lehet, hogy csak bizonyos parancs kiadása után törlődjenek ezek a szoftverek. Ekkor viszont fennáll a lehetősége, hogy a felhasználó elfelejti ezeknek a törlését, és a projektek a rendszerére telepítve maradnak. Ezek mellett problémát okozhat, ha a számítógéphez nem rendelkezik a programozó rendszergazdai hozzáféréssel, ugyanis így sok projekt és szoftver telepítésére nincs is alapvetően lehetősége.

Ezen problémákat hivatott kiküszöbölni az az ötlet, miszerint a környezet fusson egy konténerben. A konténer virtuális számítógép, ahová tetszőlegesen telepíthetők a projektek függőségei, azok bár fizikailag természetesen foglalják a helyet, bármikor egyszerűen törölhetők, és nem szükséges ehhez semmilyen rendszergazdai jogosultság.

A használni tervezett Docker [28] konténer általános architektúrája a 3.6. ábrán látható. A Docker telepítése a dokumentáció [39] leírása alapján könnyen elvégezhető többféle platformra. A Docker használatának általános lépéseire ezen fejezetben nem térek ki. A két legfontossabb műveletre azonban igen, ami pedig az image buildelése, valamint a konténer futtatása. Ezek között a különbség, hogy az image felépítésére általában egyszer, mindenesetre kevésszer van szükség, és felépítés után alapvetően nem veszik el - bár felülírható vagy törölhető -, míg a konténer futás után automatikusan törlődik. A konténer alapja az adott image: minden, amit az image-ben telepítésre vagy beállításra kerül automatikusan öröklődik az abból indított konténerbe. A kérdés tehát, hogy a specifikációban végzett műveletek hol legyenek elvégezve: a felhasználó lokális számítógépén, az image buildelésénél, vagy a konténer futtatásánál.

A tesztelő rendszer felépítése során a következőket szeretném:



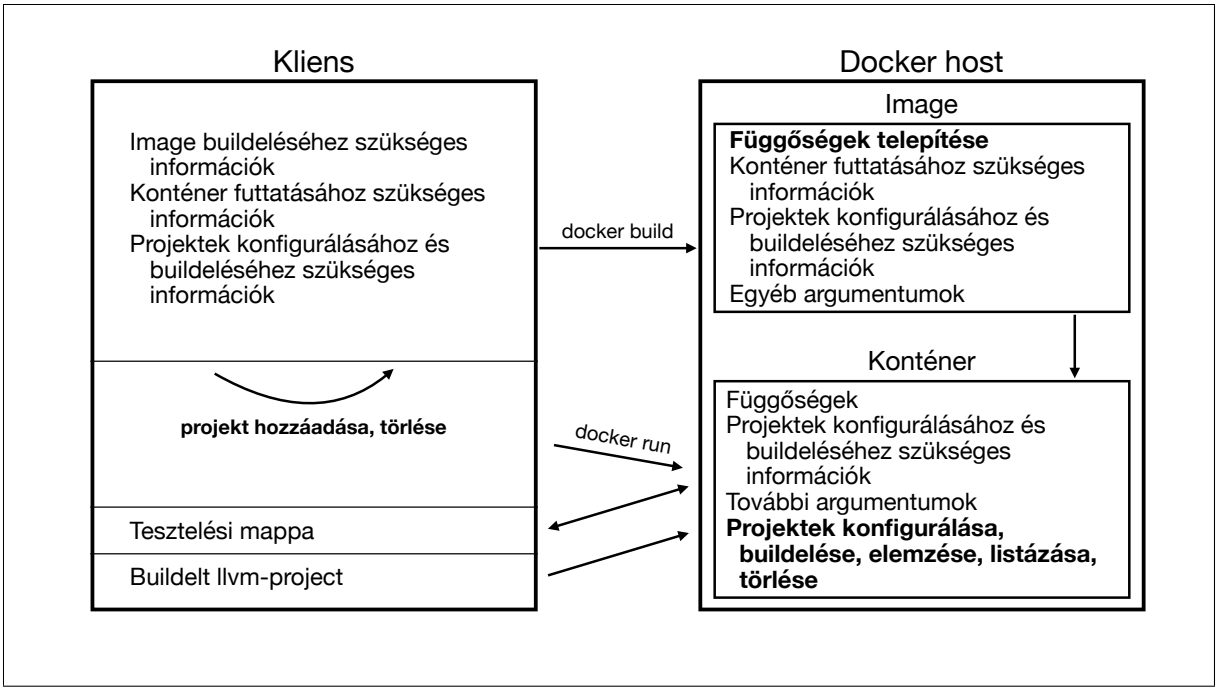
3.6. ábra. A Docker konténer általános architektúrája [28]

- mivel a függőségek minden művelet során (projektek konfigurálása, buildelése, tesztelése) szükségesek, azokat minél kevesebbszer kelljen telepíteni,
- az egy alkalommal telepített függőségeket mind a konfigurálás, mind az elemzés fel tudja használni,
- a projektek letöltésére és konfigurálására ne legyen minden elemzéskor szükség,
- a projektek letöltését és konfigurálását legyen lehetőség önállóan (elemzéstől függetlenül) futtatni,
- a teszteket legyen lehetőség önállóan futtatni (konfigurálástól és letöltéstől függetlenül),
- a tesztelés során a felhasználó által buildelt llvm-project kerüljön használatra, hiszen vélhetően ebben található a fejlesztő checkere is.

Ezen tulajdonságokból kiindulva a tesztelési környezet architektúrájának megtervezése során a 3.7. ábrán látható döntéseket hoztam. Az ábra egyszerűbbé tételéhez nem jelöltem a docker daemont, ami a kommunikációt biztosítja a kliens és a docker host, valamint a docker host és a registry között. Az ábrán a végzett műveletek félkövér betűtípussal láthatók, míg a tárolt adatok sima betűvel.

A tervezés során tehát a következő döntéseket hoztam:

- a függőségek a konténerek alapját képező image-ben lesznek telepítve, ezzel biztosítva az újrafelhasználhatóságot,



3.7. ábra. A tesztelési környezet architektúrája

- a projektek letöltése, konfigurálása, buildelése, elemzése, listázása és törlése ugyanazon konténeren belül kerülnek majd egymástól függetlenül futtatásra,
- a konténer - a korábban említett tulajdonságokból adódóan - az image-ből megörökli a már telepített függőségeket,
- a konténer a projektek letöltését, konfigurálását és buildelését egy, a kliensen található mappába végzi. Ezzel biztosítja, hogy a konténer megszűnése után is megmaradjon a letöltött projekt, valamint egy teljesen új konténer használata során is használhassa a felhasználó,
- bár az ábrából nem derül ki, a konténer a tesztelés eredményeit a kliensen található tesztelési mappába menti,
- az elemzés során a kliensen található előzőleg buildelt LLVM projektet használja a konténer.

Az előforduló alrendszerek kommunikációjára alapvetően a Docker által nyújtott lehetőségeket használom. A kliens szerverén elhelyezkedő különböző adatokat tartalmazó fájlok (szükséges információk) fel lesznak másolva az image-re, majd onnan automatikusan a konténerekre. A tesztelési mappák akár több, korábban buildelt nagy projektet is tartalmazhatnak, valamint előnyös, ha futtatás után az elvégzett műveletek nem vesznek el. Ezért ez a mappa

csatolva lesz majd a konténerhez. Hasonlóan lesz csatlakoztatva a már felhasználó által buildelt llvm-project mappa is.

Az image, valamint a konténer működésénél is látható „egyéb argumentumok" felirat. Ezeket később, a komponensek tervezésénél specifikálom majd. Egyelőre megjegyzendő, hogy ezen argumentumokat az image a build parancs futtatásakor kaphatja meg, a konténer pedig vagy az image-től örököli, vagy a run parancs futtatása közben kapja a felhasználótól. Az itt meghatározott műveletek alkotják a rendszer egy-egy komponensét.

3.4. Komponens terv

Az előző fejezet során bemutattam a checkerek, valamint tesztkörnyezet alapvető architektúrájának jellemzőit. A következőkben ezen architektúrák komponensinek megtervezését végezem el. Kezdve a komponensek nagy felépítésétől egészen az apróbb függvényekig.

3.4.1. A checkerek komponens terve

Ezen rész komponensei maguk az egyes checkerek lesznek. A neveik legyenek a következők: bugprone-bad-signal-to-kill-thread, bugprone-do-not-refer-atomic-twice, bugprone-signal-in-multithreaded-program és bugprone-spuriously-wake-up-functions. A bugprone előtag mindegyikben a modul nevére utal, amiben szerepel majd.

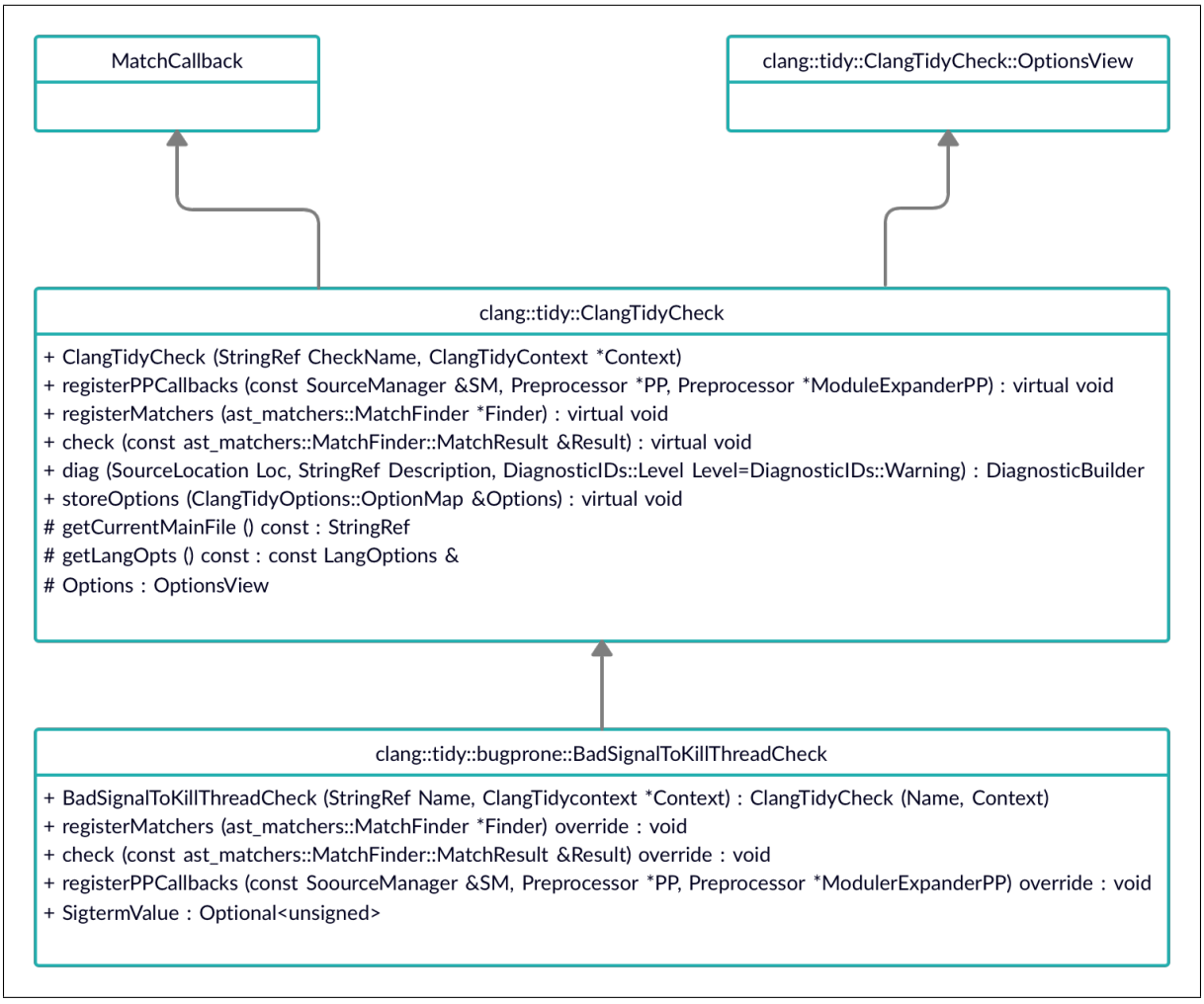
Általánosságban szinte minden checker rendelkezik két alapvető függvénnyel. Ezek a `registerMatchers()` és a `check()` függvények. Előbbi az AST-re írt matcherek regisztrálását végzi majd, utóbbi pedig a matcher találata esetén hívódik meg. Ezek a függvények a tervezendő checkerekben is megtalálhatók lesznek.

A matcherek algoritmusának tervét érdemes a matcherek későbbi implementálásának figyelembevételével létrehozni. A Clangban létező matcherek megtalálhatók a hivatalos dokumentációjában [61]. Ahogy látható, ezek a függvények egy speciális funkcionális nyelvet képeznek, melyek során van lehetőség az AST csúcsainak matchelésére, a csúcsok alapvető bejárása, valamint konvertálására is. Ezen funkcionális nyelvből kifolyólag a matcherek algoritmusát speciálisan, ezen függvények és az AST ismerete alapján mutatom be egy erre alkalmas Psceudo kód segítségével. A továbbiakban egyesével tekintem át a checkerektől várt működést, az osztálydiagramjaik tervét, valamint egyéb jellemzőiket.

bugprone-bad-signal-to-kill-thread

A bugprone-bad-signal-to-kill-thread checker írása során a POS44-C [22] szabályra keresem a megoldást. A szabály részletesebb leírása a 2.3.1. alfejezetben olvasható. Ahogy említettem, a `SIGTERM` szignál hívása különösen is nagy kockázatot jelent, mivel a programozó szabályos terminálásra szólítja fel az adott threadet, azonban ezen kívül az egész processz szabálytalan megszakításra kerül. Emiatt a szabály során ezen `SIGTERM` szignál használatára ad a checker figyelmeztetést, amikor az a `pthread_kill()` függvénnyel lett meghívva. Felmerülhet a kérdés, hogy nem kellene-e az ugyancsak a teljes processzt leállító `SIGKILL` használata esetén is figyelmeztetést adni. A terv írása közben azért dönöttem a `SIGKILL` figyelmen kívül hagyása mellett, mert ezen szingál pontosan arra való, hogy minden biztonsági ellenőrzést figyelmen kívül hagyva azonnal megszakítson minden folyamatot. Tehát ennek használatakor feltételezhető, hogy a programozó minden áron mindent le szeretne állítani.

A checker tervének UML diagramja a 3.8. ábrán látható.



3.8. ábra. A BadSignalToKillThreadCheck osztály UML diagramja

A szabály alapján az AST-n felül ellenőrizni kell a `define` kulcsszóval definiált signal értékeket is. Ezt preprocesszor visszahívással lehet megtenni, amit a `registerPPCallbacks()` függvény tesz majd meg. A `SigtermValue` változó a `SIGTERM` szignál értékét tárolja, amennyiben van ilyen.

Fontos, hogy a matcher megírása során minél inkább szűk legyen a potenciális felismerés. Az adott checker esetén a `pthread_kill` függvényeket kell megkeresni, melyek `SIGTERM` szignállal lettek meghívva. Működő megoldás lenne tehát, ha minden függvényhívásra történne matchelés, majd ezután a checker ellenőrizné, hogy ez a függvényhívás `phtread_kill` volt-e, és milyen szignállal lett meghívva, stb. Konkrét esetben a fejlesztett matcher működésése az 2. algoritmuson látható. Amennyiben a minta illeszkedik, az osztály `check()` függvénye meghívásra kerül, mely a 1. algoritmus szerint működik.

Algoritmus 1. *BadSignalToKillThreadCheck* :: *registerMatchers*()

```
1:  for az AST minden c csúcsára do
2:      matchelés( callExpression( összesTeljesül(
3:          függvényhívás( melynekADeklarálásának(
4:              aNeve( pthread_kill valamilyen osztályban ) ) ),
5:          argumentumSzáma( egyenlő(2) ),
6:          argumentuma( elsőSzámúArgumentum,
7:              egySzámLiterál().megjegyez(integer-literal néven) )
8:      )).megjegyez(thread-kill néven) )
9:      if a c csúcs megfelel a kitételeknek then
10:         BadSignalToKillThreadCheck::check meghívása
11:     end if
12: end for
```

A checkertől ezentúl azt várom, hogy a jelzés a `pthread_kill()` függvénynél jelenjen meg. Bár a Clang-Tidy-ban van lehetőség automatikus javítást beállítani, ebben az esetben ezt nem tettem meg. Ennek az oka, a programozóra bízom, milyen más szignállal szeretné a szálat leállítani.

bugprone-do-not-refer-atomic-twice

A szabály bővebb leírása a 2.3.5. alfejezetben olvasható, az ide vonatkozó SEI CERT szabály pedig a CON40-C azonosítót viseli [26]. A szabály alapján felállított problémát vizsgálva azt találtam, hogy a probléma teljes, tökéletes megoldásához nyomon kell követni egy-egy atomikus változó módosításainak menetét. Ugyanis ha egy függvényben a programozó kiolvassa az értéket, majd ismét így tenne, feltételezhető, hogy ugyan arra a változóra gondolt. Eközben azonban az érték változhatott másik thread működése során, ami logikailag rossz programot

Algoritmus 2. *BadSignalToKillThreadCheck* :: *check*()

```
 1: procedure IsSigterm(KulcsÉrték)
 2:     if KulcsÉrték→szignálNeve = "SIGTERM" then return True
 3:     elsereturn False
 4:     end if
 5: end procedure
 6: procedure TryExpandAsInteger(makró_iterátor)
 7:     if makró_iterátor→kiterjeszthető egész számnak then return kiterjesztés utáni egész szám
 8:     elsereturn semmi
 9:     end if
10: end procedure
11: SigtermMacro ← semmi
12: for minden p preprocesszor makróra do
13:     if IsSigterm(p) then
14:         SigtermMacro ← p
15:     end if
16: end for
17: if nincs SigtermMacro VAGY SigtermMakro nem kiterjeszthető számként then return
18: end if
19: *MatchedExpr ← matchelt_thread-kill
20: *MatchedLiteral ← matchelt_integer-literal
21: if MatchedLiteral→érték = SigtermÉrték then
22:     Figyelmeztetés kiírása
23: end if
```

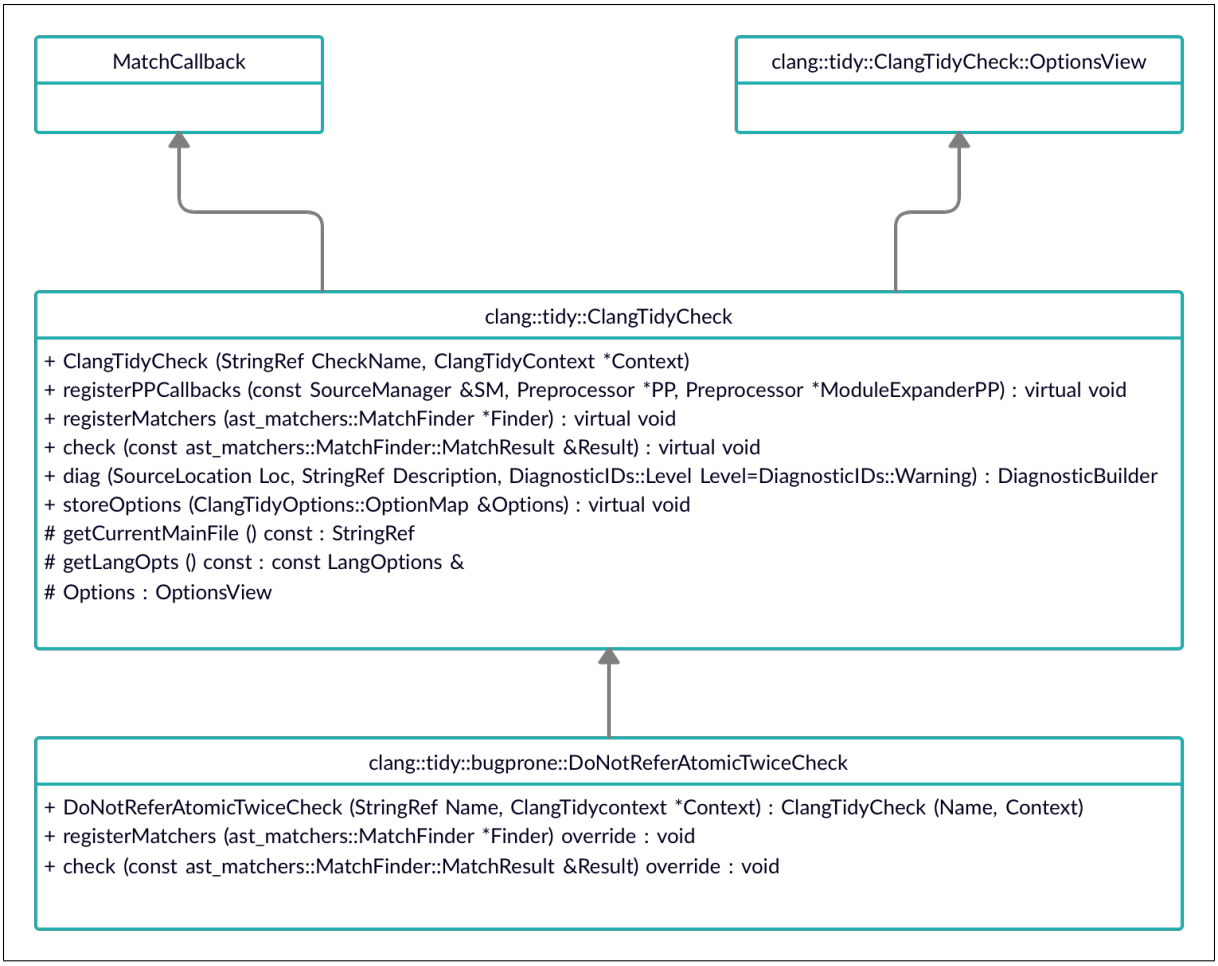
eredményez. Az atomikus változó „nyomonkövetéséhez", azaz az adatáramlás modellezésre ugyanakkor a Clang-Tidy segítségével nincs lehetőség, ezért tökéletesen biztosan nem lehet megoldani a problémát.

Nagyban szűkíteni lehet azonban a hibalehetőséget, mégpedig a bináris operátorok ellenőrzésével. Amennyiben egy bináris operátor (pl.: =) két oldalán ugyan az az atomikus változó áll, az logikailag hibás programot eredményez, ahogyan azt korábban is bemutattam. A matchelés során tehát olyan bináris operátor megkeresése a célom, melynek bal és jobb oldalán is szerepel ugyanaz az atomikus változó.

Nem lenne azonban szerencsés az összes bináris operátorra matchelni, majd ezután vizsgálni őket: először túl sok eredményt adna, és csak fokozatosan szűkülne a kör. Ehelyett először a checker keresse meg a változókat, azonnal ellenőrizze, hogy atomikus-e, és ha igen, csak ebben az esetben folytassa kitételek ellenőrzését.

A bugprone-do-not-refer-atomic-twice checker tervének UML diagramja a 3.9. ábrán látható. A matcher működésének terve a 3. algoritmuson látható.

Az előző checkerrel ellentétben itt a matcher megírásának algoritmusa több helyen magyarázatra szorul. Azt már említettem, hogy miért egy változóreferenciával kezdődik a matcher megírása. A változó atomic ellenőrzése (3. sor), illetve a deklarálásának megjegyzése (4. sor) magáért beszél, vannak azonban nem ennyire triviális részek is a matcherben.



3.9. ábra. A DoNotReferAtomicTwiceCheck osztály UML diagramja

Algoritmus 3. *DoNotReferAtomicTwiceCheck* :: *registerMatchers*()

```
 1: for az AST minden c csúcsára do
 2:     hivatkozásAmireTeljesül(
 3:         típusa( konvertálva (atomicTípus()) ),
 4:         ugrás( változóDeklarációjához().megjegyez("atomic" néven) ),
 5:         vanFelmenője( binárisOperátorAmireTeljesül(
 6:             vanJobbOldalán( valaholBenne( hivatkozásAminek(
 7:                 változóDeklarációja( megegyezik( "atomic"-kal ) ). megjegyez( "rhs" néven ),
 8:             ) )
 9:             nincsMatchelésHa( vanBenneValahol( atomicFüggvény() ) )
10:         )),
11:         nincsMatchelésHa( megegyezik( "rhs"-sel ) )
12:     )
13:     if a c csúcs megfelel a kitételeknek then
14:         DoNotReferAtomicTwice::check meghívása
15:     end if
16: end for
```

A `vanFelmenője()` (5. sor) függvény adott csúcstól az AST-ben elindul a fán „fölfelé", és megnézi, van-e őse egy-egy csúcsnak. Ennek megértéséhez fontos tudni, hogy az operátorok esetén egy változó referenciájának minden esetben felmenője maga az operátor. Ezután a

`vanJobbOldalán()` függvénnyel (6. sor) ellenőrzi, hogy az operátor másik oldalán is történik-e hivatkozás az adott változóra. A `nincsMathcelésHa( vanBenneValahol( atomicFüggvény() ) )` részre (9. sor) azért van szükség, mert egyes atomikus függvényekkel el lehet érni, hogy az amúgy nem biztonságos művelet mégis az legyen. Ilyen például az `atomic_compare_exchange_weak()`. Ilyen függvényt találva a checker feltételezi, hogy a felhasználó megfelelően használta ezt, és nem ad figyelmeztetést. Az utolsó magyarázatra szoruló rész az, hogy miért nem történik matchelés a változóra, ha az megegyezik `rhs`-sel (11. sor). A válasz abban rejlik, hogy ezen sor nélkül mind az első hivatkozás (2. sor), mind az `rhs`-ként elmentett hivatkozás (7. sor) megegyezhetne, azaz a műveletek során önmagát találná meg. Ezzel a kiegészítéssel azonban biztosítva van, hogy a `hivatkozásAmireTeljesül()` hívás (2. sor) a bináris operátor bal oldalán, míg a másik operátor (7. sor) a jobb oldalán helyezkedik el.

Amennyiben a matchelés sikeres volt, az osztály `check()` függvénye meghívásra kerül. Ennek működése a 4. algoritmuson látható. A figyelmeztetést a második változóhasználatra szeretném kapni, hiszen ott történt a logikailag hibás, azaz második hívás.

Algoritmus 4. *DoNotReferAtomicTwiceCheck* :: *check*()

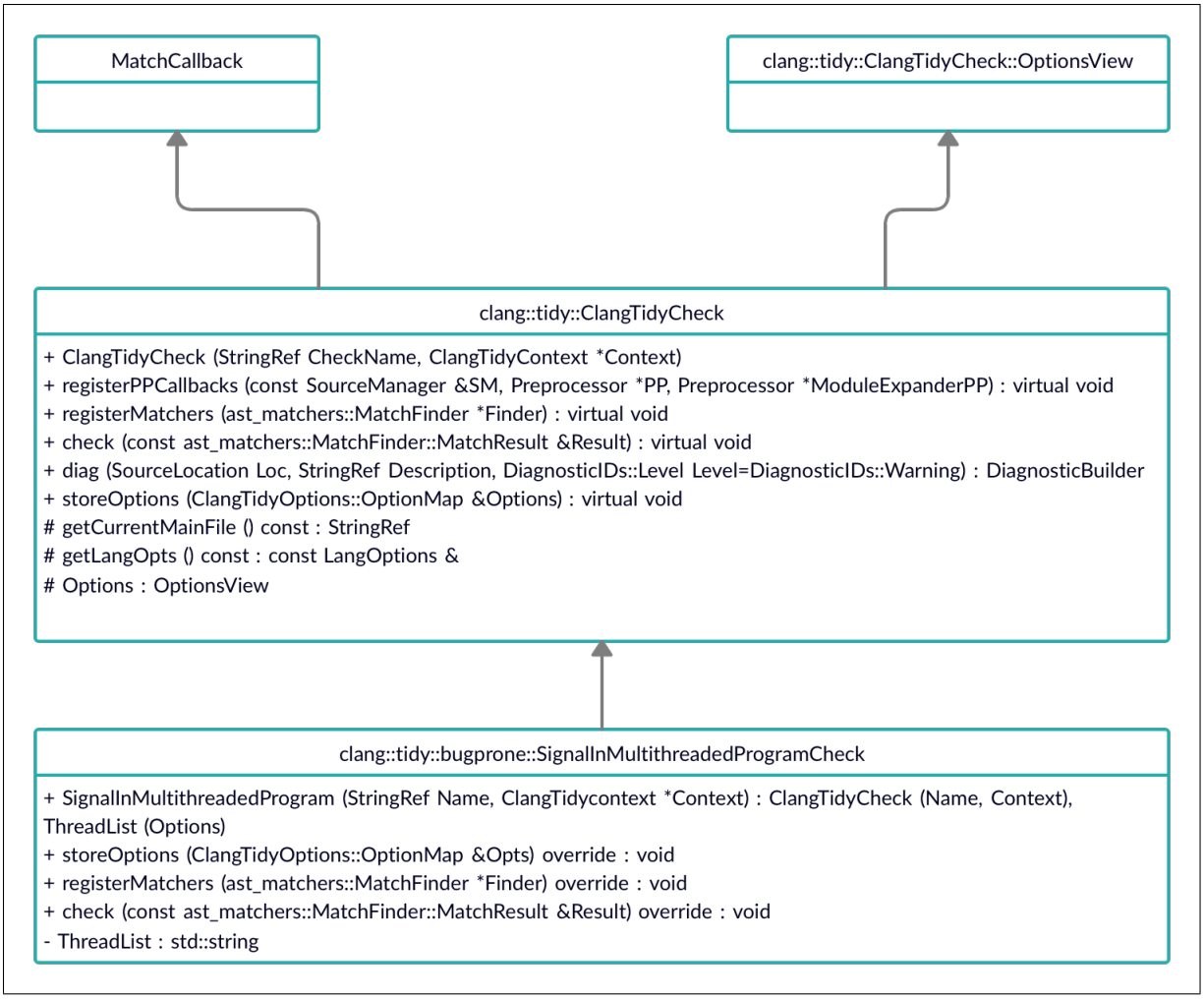
1: MatchedVar←matchelt_atomic
2: MatchedRef←matchelt_rhs
3: **if** nem talált MatchedVar-t VAGY nem talált MatchedRef-et **then return**
4: **end if**
5: Figyelmeztetés kiírása

bugprone-signal-in-multithreaded-program

A bugprone-signal-in-multithreaded-program checker a 2.3.4. alfejezetben bővebben is bemutatott CON37-C [25] szabálynak ellentmondó programot igyekszik majd felismerni. A checker tervének UML diagramja a 3.10. ábrán látható.

A checkerrel a `signal()` függvény hívását kell felismerni, és figyelmeztetést adni, ha a program többszálú. A checker írása közben feltételezem, hogy egy függvény többszálú, ha tartalmaz

- `std::thread` változó létrehozást, vagy
- `ThreadList` függvényhívást, vagy
- `thrd_create` függvényhívást, vagy
- `std::thread` függvényhívást, vagy



3.10. ábra. A SignalInMultithreadedProgramCheck osztály UML diagramja

- `boost::thread` függvényhívást, vagy
- `pthread_t` függvényhívást.

Ezentúl szeretném, ha a felhasználó a saját thread függvényének nevét átadhatná a checkernek paraméterként. A matcher regisztrálása előtt létrehoztam majd egy saját `hasAnyListedName()` függvényt, mely a `hasAnyName()` függvény paraméterének átadja a paraméterben kapott függvények neveit. A `hasAnyName()` függvény egy adott rekord esetén megvizsgálja, hogy a rekord neve megegyezik-e bármelyik névvel a listában. Ezenkívül betöltöm a `ThreadList` változóba a Tidy meghívásánál megadott paramétereket. Ezután regisztrálom a matchert az 5. algoritmus szerint.

A `threadCall()` segédmatcher használatának nincs külön jelentősége, viszont megnöveli a program olvashatóságát. A `threadCall()`-on belül (2-11. sor) a program ellenőrzi, hogy történik-e említett függvényhívás. Az `implicitKonverziótFigyelmenKívülHagyva()`

Algoritmus 5. *SignalInMultithreadedProgramCheck* :: *registerMatcher*

```
 1: for az AST minden c csúcsára do
 2:     threadCall ← bármelyikTeljesül(
 3:         vanLeszármazottjaAmi( függvényhívás(implicitKonverziótFigyelmenKívülHagyva(
 4:             vanLeszármazottjaAmi( hivatkozás( deklarációja( függvénydeklaráció(
 5:                 hasAnyListedName( ThreadList )
 6:             ) ) ) )
 7:         ) ) ),
 8:         vanLeszármazottjaAmi( változóDeklaráció( típusa( rekordDeklaráció(
 9:             neve( "std::thread" )
10:         ) ) ) )
11:     )
12:     matcherRegisztálás(
13:         függvényhívás( implicitKonvertálástFigyelmenKívülHagyva( vanLeszármazott( hivatkozás( deklará-
    ciója( függvénydeklaráció( összesTeljesül(
14:             neve( "signal" ),
15:             praméterszáma( 2 ),
16:             paramétere( 0. paraméter, típusa( egészSzám ) )
17:         ) ) ) ) ) ) ),
18:         vanFelmenője( összetettStatement( threadCall ) )
19:     ).megjegyez( "signal" néven )
20:     if a c csúcs megfelel a kitételeknek then
21:         DoNotReferAtomicTwice::check meghívása
22:     end if
23: end for
```

(3. sor) függvényre a felépített AST sajátosságai miatt van szükség. A `bármelyikTeljesül()` (2. sor) függvény második paraméterére (8. sor) azért kell külön, mivel a matchelni kívánt legtöbb thread hívás bár függvényhívás, az `std::thread` másképp működik, változóként kell keresni az AST-ben.

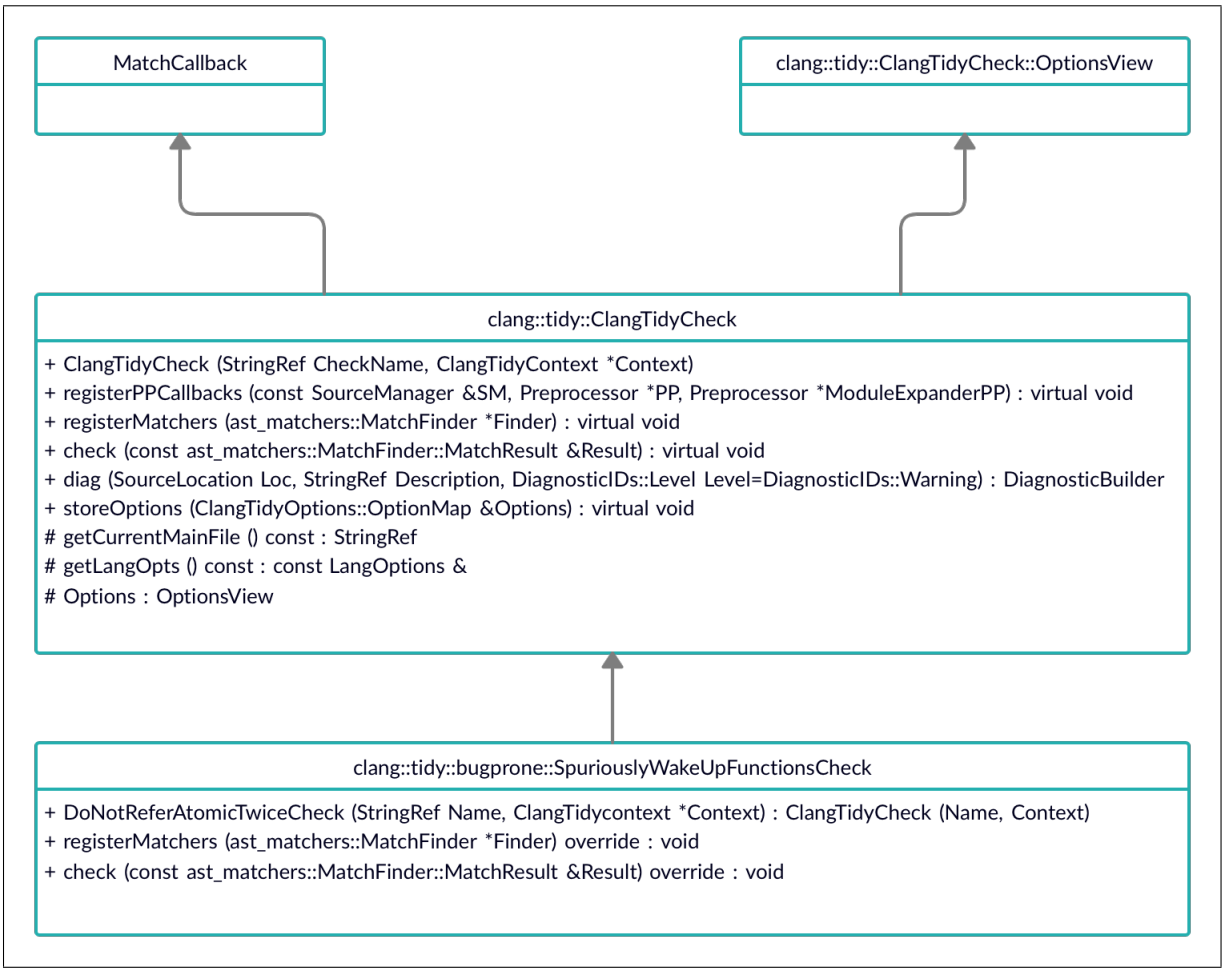
A `threadCall()` megírása bár megjelenésileg följebb található az algoritmusban (1-11. sor), fontos megemlíteni, hogy a regisztrált matcher először a `signal()` függvényt keresi meg (13-17. sor), majd ha talál ilyet, csak abban az esetben ellenőrzi, hogy többszálú programmal van-e dolga (18. sor).

Amennyiben a matchelés végbemegy, az osztály `check()` függvénye meghívásra kerül. Ebben a `check()` függvényben nincs szükség további ellenőrzésre, a `signal()` függvény hívásának helyén figyelmeztetést ad.

bugprone-spuriously-wake-up-functions

A bugprone-spuriously-wake-up-functions checker létrehozása során két hasonló SEI CERT szabály megoldásán dolgozok, mégpedig a CON54-CPP [23] és a CON36-C [24] azonosító problémákon, melyekről bővebben a 2.3.2. és a 2.3.3. alfejezetekben lehet olvasni. A checker

tervének UML diagramja a 3.11. ábrán látható.



3.11. ábra. A SpuriouslyWakeUpFunctionsCheck osztály UML diagramja

A matcher írásakor segédmatchereket fogok használni. A `hasUniqueLock()` megvizsgálja, hogy az adott csúcs leszármazottjai között van-e `std::unique_lock<mutex>` típusú változó. A `hasWaitDescendantCPP()` megvizsgálja, hogy a csúcs leszármazottai között van-e a következő függvények közül valamelyik: `std::condition_variable::wait()`, `std::condition_variable::wait_for()`, `std::condition_variable::wait_until()`, valamint ezek nem rendelkeznek a feltételt tartalmazó lambda függvény paraméterrel. A `hasWaitDescendantC()` matcher hasonlóan működik, azonban a `cnd_wait()` és a `cnd_timedwait()` függvényekre, melyek esetén nincs lehetőség az említett lambda paraméter megadására.

A szükséges segédmatcherek definíciója után a matcher logikája a 6. algoritmus szerint működik.

A C és C++ függvények különböző matchelésére azért van szükség, mert a C-ben használt `cnd_wait()` és `cnd_timedwait()` - mint ahogy az a SEI CERT oldalán található pél-

Algoritmus 6. *SpuriouslyWakeUpFunctionsCheck* :: *registerMatcher*()

```
 1: for az AST minden c csúcsára do
 2:     if elemzendő program nyelve C++ then
 3:         matcherRegisztálás( haElágazás( mindTeljesül(
 4:             hasWaitDescendantCPP,
 5:             nincsMatchelésHa(
 6:                 bármelyikTeljesül(
 7:                     vanLeszármazottja( haElágazás( hasWaitDescendantCPP ) ),
 8:                     vanLeszármazottja( whileCiklus( hasWaitDescendantCPP ) ),
 9:                     vanLeszármazottja( forCiklus( hasWaitDescendantCPP ) ),
10:                     vanLeszármazottja( doWhileCiklus( hasWaitDescendantCPP ) )
11:                 )
12:             )
13:         ) ) )
14:     else
15:         matcherRegisztálás( haElágazás( mindTeljesül(
16:             hasWaitDescendantC,
17:             nincsMatchelésHa(
18:                 bármelyikTeljesül(
19:                     vanLeszármazottja( haElágazás( hasWaitDescendantC ) ),
20:                     vanLeszármazottja( whileCiklus( hasWaitDescendantC ) ),
21:                     vanLeszármazottja( forCiklus( hasWaitDescendantC ) ),
22:                     vanLeszármazottja( doWhileCiklus( hasWaitDescendantC ) ),
23:                     vanSzülője( whileCiklus() )
24:                     vanSzülője( összetettStatement( vanSzülője( whileCiklus() ) ) )
25:                     vanSzülője( forCiklus() )
26:                     vanSzülője( összetettStatement( vanSzülője( forCiklus() ) ) )
27:                     vanSzülője( doWhileCiklus() )
28:                     vanSzülője( összetettStatement( vanSzülője( doWhileCiklus() ) ) )
29:                 )
30:             )
31:         ) ) )
32:     end if
33:     if a c csúcs megfelel a kitételeknek then
34:         SpuriouslyWakeUpFunctionsCheck::check meghívása
35:     end if
36: end for
```

dából [25] is látszik -, gyakran `if` elágazáson belül kerülnek meghívásra, ezért úgymond azon túl is meg kell vizsgálni, található-e `while`, `dowhile`, vagy `for` ciklus. Az említett ciklusok megtalálásakor a checker feltételezi, hogy a programozó a ciklusban a megfelelő feltételt ellenőrzi. Mivel ez a feltétel nem általánosítható, ezért döntöttem ezen feltételezés mellett.

Amennyiben a matchelés sikeres volt, az osztály `check()` függvénye meghívásra kerül. Mivel további ellenőrzésre nincs szükség, ez a `hasWaitDescendantC()` vagy `hasWaitDescendantCPP()` által megtalált függvénynél szeretném figyelmeztetni a felhasználót a hibás használatra.

3.4.2. A tesztkörnyezet komponens terve

Ahogy a környezet architektúrális felépítéséből is kiderült, alapvetően 4 nagy komponensre bontható a környezet működése. Ezek közül néhány aztán tovább osztható kisebb részekre. Ezek a komponensek a következők:

- projekt hozzáadása a környezethez,
- projekt törlése a környezetből,
- image buildelése és függőségek telepítése,
- konténer futtatása:
  - projektek beállítása,
  - projektek elemzése,
  - projektek kilistázása,
  - projektek mappáinak törlése.

Az egyes komponensek egy-egy különböző feladat elvégzéséért felelnek. Ezeket a feladatokat, a szükséges bemeneteket, és az elvárt működéseket mutatom be a következő alfejezetekben. Minden komponenstől elvárom, hogy a feladatának elvégzésén kívül visszajelzést nyújtson a felhasználó számára a tevékenységéről, valamint detektálja az észrevehető hibákat, és ilyenek esetén adjon hibajelzést.

Projekt hozzáadása a környezethez

Ahogy azt a specifikációban is megfogalmaztam, azt szeretném, hogy a tesztkörnyezet használható legyen több, előre beállított projekt elemzésére. Ezen kívül, ha haladó programozó használja a környezetet, könnyen igénye nyílhat új, saját projektet hozzáadni a környezethez. Ehhez nyújt segítséget az éppen taglalt komponens.

A komponens elsődleges célja tehát, hogy egy nyílt forráskódú projektet hozzáadjon a tesztkörnyezethez, és annak letöltését, konfigurálását, buildelését, valamint tesztelését lehetővé tegye. Emiatt a komponensnek képesnek kell lennie elmenteni az adott projekt következő adatait a tesztkörnyezet megfelelő fájljaiba:

- a projekt neve,
- a projekt git linkje,

- a projekt függőségeinek nevei (ha van),
- a projekt speciális konfiguráló scriptje (ha van),
- a projekt `configure` parancs hívása során használandó argumentumait (ha van).

Azért ezen tulajdonságokat választottam elmentésre, mivel az általam tesztelt projektek beállítása során legtöbbször ezen adatokkal elvégezhető volt a projekt konfigurálása. A komponenstől elvárom, hogy a git linket ellenőrizze, azaz ha nincs ilyen link vagy nem hozzáférhető, adjon hibajelzést és a mentés ne menjen végbe. A rendszer architektúrájából adódik, hogy az újonnan hozzáadott projekt nem lesz használható az image újrabuildelése nélkül.

Projekt törlése a környezetből

Amennyiben a programozó új projekt hozzáadása közben nem triviális hibát vét (pl.: rosszul sorolja fel a függőségeit a projektnek), a projekt hozzáadásánál ez nem derül ki, azonban a projekt mégsem lesz konfigurálható, ennélfogva tesztelhető sem. Ilyenkor a programozónak lehetőséget kell nyújtani, hogy a korábban, rosszul hozzáadott projektet törölje, majd ismét hozzáadja. Ez a komponens a projekt adatainak törlését hivatott elvégezni.

A működésétől annyit várok el, hogy miután a felhasználó megadta a projekt nevét, törölje annak összes adatát a környezetből. Ezek az adatok természetesen megegyeznek a projekt hozzáadása részben taglaltakkal. Előnyös továbbá, ha a komponens felismeri, ha a megadott projekt nem létezik, ezért ebben az esetben hibát szeretnék kapni.

Image buildelése és függőségek telepítése

Ahogy az architektúrális tervben is említettem, az image buildelése során az egyik legfontosabb feladat a projektek függőségeinek telepítése. Ez a telepítés alapértelmezetten folyhat az előre megadott függőségek listája alapján, melyeket lehet telepíteni az apt Debian Package Managementtel [57], vagy egy ugyancsak előre megadott szkript alapján. Ahogy korábban lefektettem, a környezetnek alkalmasnak kell lennie előre megadott projektek tesztelésére. Azonban minél több projekt kerül tesztelésre, annál hosszabb időbe telik. Emiatt a tesztkörnyezettől jogosan várható el, hogy legyen lehetőség - az előre beállított projektek közül - a projektek kiválasztására. Az architektúrából adódóan ha az image buildelése során csak <p> projekt függőségei kerülnek telepítésre, az utána használt konténerben is csak ezek lesznek telepítve, amit fontos itt is megjegyezni.

Az image buildelése során tehát elvárható, hogy adott projektek függőségei telepítve legyenek, valamint a telepítés során fellépő hiba esetén a buildelés hibát jelezzen és sikertelen legyen. A buildeléstől elvárom továbbá, hogy valamilyen módon lehetőséget nyújtson a következők beállítására:

- mely projektek függőségei legyenek telepítve, ezáltal később melyik projektek legyenek beállíthatók és elemezhetők,
- a konténer futtatása során később alapértelmezetten megtörténjen-e a projektek letöltése, konfigurálása és buildelése,
- a konténer futtatása során később alapértelmezetten megtörténjen-e a projektek elemzése,
- a konténer futtatása során később alapértelmezetten engedélyezett checkerek neveinek megadása
- a konténer futtatása után alapértelmezetten törlődjenek-e a projektek mappái.

Ezekre azért van szükség, hogy a későbbi konténer futtatás alatt ne legyen mindig kötelező például a projektek beállításának kikapcsolása, hanem legyen mód ezt alapértelmezetté tenni. Az image buildelésétől továbbá megkövetelem, hogy minden argumentum opcionális legyen, tehát legyen egy-egy alapértelmezett értéke.

Konténer futtatása

A konténer futtatása során a következő műveleteket szeretném lehetővé tenni az előre megadott projekteken:

1. projektek letöltése,
2. projektek beállítása a legfrissebb stabil verzióra,
3. projektek konfigurálása különböző, automatikusan felismert parancsokkal,
4. projektek konfigurálása előre megadott script segítségével,
5. CodeChecker környezetének beállítása,
6. projektek buildelése a CodeChecker segítségével,
7. elemzés futtatása a megadott checkerekkel,
8. elemzés elmentése HTML fájlként,
9. projektek mappáinak automatikus törlése,

10. engedélyezett projektek kilistázása.

A működés során elvárom, hogy a projektek listája szűkíthető legyen az image listájához képest, valamint a checkerek tetszőlegesen kiválaszthatók legyenek. Ezenkívül szeretném, hogy a felhasználónak ne kelljen minden alkalommal az összes műveletet futtatnia, hanem választhasson az alábbiak közül: setup (ami a projektek beállítását hivatott végezni, azaz a következő műveleteket: 1,2,3,4,5,6), analyze (ami az elemzést futtatja a 5,7,8 műveletek alapján), list (10) és delete (9). Így lehetővé téve a felhasználó számára, hogy pontosan azokat a műveleteket végezze el, amikre szüksége van. Felesleges például minden alkalommal elvégeznie a projektek konfigurálását és buildelését, azt elegendő egyszer a legelső futtatáskor.

Szeretném, ha ezeket a műveleteket a komponens automatikusan, az előre megadott adatok (pl.: git link) alapján végezné el. Természetesen itt is elvárom, hogy a futás közben visszajelzést nyújtson a felhasználó számára, valamint ha valamelyik művelet sikertelen, a hibaüzenetet kiírva azonnal leálljon. Külön megkövetelem az inputok ellenőrzését (projektek, checkerek, stb.). Mivel a konténer alapértelmezetten root joggal bír, az ezzel nem rendelkező felhasználó számára a docker által írt fájlok (teszt fájlok, projektek buildelése) előfordulhat, hogy nem lennének módosíthatók és törölhetők egy átlagos felhasználó számára. Ráadásképp emiatt szeretném, hogy a futás végén beállítsa a szkript a megfelelő jogosultságokat a felhasználónak.

3.5. Implementálási terv

Míg a checkerek fejlesztése viszonylag kötött folyamat, hiszen az ezt fejlesztő közösség elvárja a beépített LLVM könyvtár használatát és a megszokott matcher implementációt, a tesztkörnyezet fejlesztésénél már kevésbé van megkötve a fejlesztő keze. Ennek során tehát különösen is át kellett gondolnom, hogy milyen implementálási stratégiákat alkalmazok.

3.5.1. A tesztkörnyezet implementálási terve

A tesztkörnyezet specifikációja során kiemeltem, hogy szeretném ezt a környezetet minél inkább kis méretűnek tartani. Ez persze nem azt jelenti, hogy megspórolható a projektek által kötelezően előírt függőségek telepítése, sem azt, hogy a projektek teljes buildelése nélkül lehetne azokon statikus elemzést futtatni a Clang-Tidy eszközzel. Más döntéseknél azonban nagyban figyelembe tudom venni ezt a szempontot, és így is kell, hogy tegyek.

Az alap image kiválasztása

A docker konténer fejlesztésekor első teendőnek kell lennie, hogy kiválasszam, milyen image-ből szeretném a saját image-et származtatni („baseImage"). Ez korántsem egy egyszerű kérdés. Esetemben mindenképpen figyelembe kell venni, hogy az image minél kisebb legyen. Ugyanakkor vannak olyan tulajdonságok, melyekre szeretnék építeni, ezért mindenképpen egy stabil, elegendő utlity-vel rendelkező Linux image-ből szeretnék származtatni. A választásom végül a `debian:stable-slim` image-re esett, ezen ugyanis megfelelően működik a legtöbb projekt esetén referált apt Debian Package Manager [57], viszont jóval kisebb, mint más Linux verziók. Csak egy gyors összehasonlítás: a `debian:stable-slim` image mindössze 69.2 MB önmagában, míg az alap `debian` image 114 MB.

Ez persze nem azt jelenti, hogy a leendő image mindössze ekkora lesz. Erre ugyanis rájönnek még a különböző függőségek, amik nagyban megnövelik majd ezt az értéket.

Scriptek nyelvének kiválasztása

Ahhoz, hogy a korábban felsorolt műveletek, funkciók végrehajtásra kerüljenek, természetesen valamilyen nyelven programot kell majd írnom. A programozási nyelv kiválasztásánál ismét az egyszerűség volt az elsődleges szempont. A programokat bash scriptben fogom írni, hogy az image-re kizárólag ezek miatt ne kelljen semmilyen új környezetet (pl.: python) feltelepíteni, és így továbbra is alacsonyan tartsam a környezet alapértelmezett méretét.

Adatábrázolás

Ahogy azt az architektúrális tervezésnél is bemutattam, valamint a komponensek tervezésnél is szó esett róla, a projektekről számon kell tudni tartani bizonyos tulajdonságokat. Ilyen tulajdonságok, hogy honnan kell letölteni, mi a neve, valamint hogyan kell konfigurálni. Az adattárolás módjánál ismét az egyszerűséget és a méretet veszem figyelembe. Fontos továbbá az is, hogy a projektek száma nem ugorhat meg nagyon, azaz szó sincs arról, hogy a környezetnek egyszer több ezer projektet kellene elemeznie. Ilyen okból tehát nincs szükség nagy adatbázis létrehozására sem.

Az adatokat jól elkülönítve fogom tárolni, megkönnyítve ezzel a fejlesztő, valamint az ezek módosítását végző scriptek működését (projekt hozzáadása, törlése). Az imént felsorolt okokból alapvetően kétféle módon fogom tárolni a projektek adatait: `txt` szövegfájlban és `sh` bash szkriptben. A projektek neveit, valamint git linkjüket egy txt fájlban tárolom majd. A projektek függőségeit, valamint a `configure` parancs argumentumaival ugyan így teszek,

de ezeket már projektenként külön-külön fájlban. Ezentúl a speciális beállító és konfiguráló szkripteket ugyancsak külön-külön fájlban tárolom majd, amit a környezet futtathat.

3.6. Megvalósítás

A tervek felépítése után itt az idő, hogy bemutassam a megoldást részleteibe menően. A megoldások bemutatása során feltételezem, hogy az olvasó elolvasta és elvégezte a telepítési útmutatóban (2.5. alfejezet), valamint a használati útmutatóban (2.6. és 2.7 alfejezetek) leírtakat. A bemutatás során hivatkozok az ott megjelenő fájlokra és mappákra. A fájlok megtalálhatók továbbá a dolgozat mellékletében is (A. függelék).

3.6.1. A checkerek megvalósítása

Az elkészített checkerek forrásfájljai megtalálhatók az `llvm-project/clang-tools-extra/clang-tidy/bugprone/` mappában. Minden checkerhez két fájl tartozik, mégpedig egy C++ és az ahhoz tartozó header fájl. A következő néven kell ezeket keresni:

- BadSignalToKillThreadCheck
- DoNotReferAtomicTwiceCheck
- SignalInMultithreadedProgramCheck
- SpuriouslyWakeUpFunctionsCheck

A checkerek implementálása során a legnagyobb feladatot a matcherek helyes megírása jelentette. A matcherek írására és menet közbeni tesztelésére a clang-query segítségével volt lehetőségem, amit már korábban is említettem (3.3.1. alfejezet). A checkerek fejlesztése elején támaszkodtam az `llvm-project/clang-tools-extra/clang-tidy` mappában található `add_new_check.py` scriptre, mely a szükséges fájlok alapjainak generálásáért, valamint ezen fájlok megfelelő mappába való elhelyezéséért felelt.

Checkerek dokumentációja

Ahogy azt az architektúrális tervezésnél is megjegyeztem, egy checker fontos része annak dokumentálása. Ennek a dokumentálásnak egyrészt részét képezi a header fájl elején való rövid

leírás komment formájában és az `llvm-project/clang-tools-extra/docs` mappában található `ReleaseNotes.rst` megfelelő módosítása is. Ezek a módosítások elsősorban a fejlesztők munkáját segítik.

A dokumentálás másik fontos lépése a dokumentációs fájlok megírása, melyek az `llvm-project/clang-tools-extra/docs/clang-tidy/checks` mappában találhatók. Ebből a dokumentációból ugyanis a hivatalos release kiadása után generálódik egy HTML kimenet, ami utána minden felhasználó számára könnyedén elérhető. Az általam írt dokumentációs fájlok a következők:

- `bugprone-bad-signal-to-kill-thread.rst`,
- `bugprone-do-not-refer-atomic-twice.rst`,
- `bugprone-signal-in-multithreaded-program.rst`,
- `bugprone-spuriously-wake-up-functions.rst`.

Ezek közül kettő már a hivatalos honlapon is megtalálható [62, 63].

Aliasok beállítása

Ahogy az architektúra tervezésénél is említettem, a checkereim a SEI CERT [2] szabályaira adnak teljes vagy részleges megoldást, ezért a Clang-Tidy cert moduljához is hozzá kell őket adni. Ezt természetesen nem bemásolással kell megtenni, hanem a megfelelő aliasok létrehozásával, valamint ezek dokumentálásával. Ehhez elsősorban következő fájlokban kellett módosítást végrehajtanom (főmappának jelen kivételes esetben az `llvm-project/clang-tools-extra` mappa tekintendő):

- `clang-tidy/cert/CERTTidyModule.cpp`,
- `docs/ReleaseNotes.rst`,
- `docs/clang-tidy/chekers/list.rst`,
- `docs/clang-tidy/chekers/cert-con36-c.rst`,
- `docs/clang-tidy/chekers/cert-con37-c.rst`,
- `docs/clang-tidy/chekers/cert-con40-c.rst`,
- `docs/clang-tidy/chekers/cert-con54-cpp.rst`,
- `docs/clang-tidy/chekers/cert-pos44-c.rst`.

3.6.2. A tesztkörnyezet megvalósítása

A tesztkörnyezet megvalósításának bemutatását először az adattárolást végző fájlokkal mutatom be. Ezután az architektúrális tervezésben is megjelent komponenseken keresztül megyek majd tovább. Ezen komponensek általában egy-egy, esetleg pár futtatható fájlhoz kötődnek. Tekintve, hogy a fájlok összmérete nincs 1 MB, ezen fájlok minden alkalommal a buildelt image-re másolódnak.

Ahogy az implementálási tervnél (3.5.1. alfejezet) is említettem, az adatok tárolását txt és sh fájlokban tárolom. A különböző fájlok, azok elhelyezkedése, valamint leírása a 3.1. táblázatban olvashatók.

Fájl neve	**Fájl elhelyezkedése**	**Fájl tartalma**
project_links.txt	clang-test-docker	Projektek nevei és git linkjei projektenként új sorban, szóközzel elválasztva
<p>_custom_deps_debian.sh	clang-test-docker/ requirements/	<p> projekt függőségeit telepítő script
<p>_debian.txt	clang-test-docker/ requirements/	<p> projekt függőségeit tartalmazza sortöréssel elválasztva
<p>_setup.sh	clang-test-docker/ setup_files/	<p> projekt telepítését végző script
<p>_config_args.txt	clang-test-docker/ setup_files/	<p> projekt `configure` parancsnál használandó argumentumai egy sorban

3.1. táblázat. A tesztkörnyezet adatait tároló fájlok

Projekt hozzáadása a környezethez

A projekt hozzáadása a projekt főmappájában található `add_project.sh` szkripttel történik. A szkript nem vár semmilyen argumentumot, működéséről pedig a standard outputon tudatja a felhasználót. A működés során a 2.7.5. alfejezetben látható futási üzeneteket adja, ahol interaktívan kommunikál a felhasználóval. A felhasználó általi bemenetet kezelve az imént említett fájlokban megfelelően eltárolja a megadott tulajdonságokat. Fontos kiemelni, hogy ezen tulajdonságokat akár a függőségeket tartalmazó txt fájlról, akár a beállítást végző scriptről van szó, minden esetben átmásolja a megfelelő helyre. Azaz a felhasználó a projekt hozzáadását követően ezeket a (saját) fájlokat nyugodtan törölheti.

Az implementációból két kisebb részt emelnék ki, amik kevésbé triviálisak. Az egyik, hogy a projekt linkje a 3.12. ábra szerinti módon kerül ellenőrzésre. Ennek során a `wget`

letöltés nélkül megvizsgálja, hogy a link létezik-e, azaz le tudná-e tölteni. Amennyiben nem, a `$?` változó, ami a legutóbbi parancs sikerességéről tárol információt, nullától különböző értéket vesz fel, és emiatt az eljárás hibát dob.

```
1 wget $link -q --spider -o /dev/null
2 if [ $? -ne 0 ]; then
3     echo "Invalid repository!"
4     exit 1
5 else
6     printf "Repository checked."
7 fi
```

3.12. ábra. Az `add_project.sh` script git link ellenőrzése

A másik említésre méltó művelet, hogy a felhasználó részéről nincs szükség a projekt nevének megadására. Mivel a projekt nevének pontosan egyeznie kell a git linken található névvel, ahogy azt a felhasználói dokumentációban említettem, ezért félő, hogy a programozó rosszul adja hozzá, és a tesztelés során használhatatlanná válik. Éppen ezért a 3.13. ábrán látható módon a script képes a linkből kikövetkeztetni a projekt nevét, és eltárolni azt.

```
1 basename=$(basename $link)
2 name=${basename%.*}
```

3.13. ábra. Az `add_project.sh` script névkikövetkeztetése

Projekt törlése a környezetből

Egy projekt törlése ugyancsak a `test-clang-docker` főmappában található `delete_project.sh` scripttel történik a felhasználói dokumentációban bemutatott módon. A script beolvassa a projekt nevét, ellenőrzi, hogy van-e ilyen projekt. Ha nincs, hibát dob és terminál, ha pedig van, törli a projekthez tartozó összes fájlt, valamint törli a `project_links.txt` fájlból az erre a projektre vonatkozó sort.

Image buildelése és függőségek telepítése

Az image buildelése alapvetően a főmappában található `Dockerfile` futtatásával történik. Ebben az image az implementálás tervezésénél (3.5.1. alfejezet) említett okból a `debian:stable-slim` image-ből származik le. Ezután kiolvassa az argumentumokat, és el-

tárolja környezeti változóként, hogy azt futtatásnál szükség esetén lehessen módosítani. A lehetséges argumentumokról és azok használatáról a felhasználói dokumentációban lehet olvasni (2.7. alfejezet). Az alapértelmezett argumentumértékeket úgy választottam meg, ahogy egy kezdő felhasználó számára a leginkább javasolnám az első futtatást: a projektek beállítása és elemzése menjen végbe a projektek mappáinak törlése nélkül.

Ezután frissítésre kerül az apt Debian Package Manager, valamint átmásolódik a teljes `test-clang-tools` könyvtár az image-re, majd a `install_deps.sh` script segítségével települnek a szükséges függőségek. A futtatás végén beállítódik, hogy egy konténer indítása esetén mit tegyen alapértelmezetten: jelen esetben futtassa a `start.sh` scriptet, és adja át neki az összes argumentumot.

Az `install-deps.sh` scriptről ejtenék még néhány szót. A script feladata, hogy telepítse a projektek függőségeit. A projekteket argumentumként kapja meg annak megfelelően, ahogy az image buildelése során megadásra került: vesszővel elválasztva. Az argumentum feldolgozása után ellenőrzi, hogy lett projekt hozzáadva (enélkül ugyanis a telepítésnek nincs értelme), valamint hogy a szükséges adatfájlok (`project_links.txt` fájl, `requirements` mappa) rendelkezésre állnak.

Amennyiben az összes projekt engedélyezve van (`projects=all`), az összes projektet kiolvassa a megfelelő fájlokból. Amennyiben nincs, ellenőrzi, hogy megfelelő projektnevek kerültek-e használatra a `check_project_args.sh` scripttel. Ezután telepíti a projektek függőségeit a megfelelő módon: ha van `<p>_custom_deps_debian.sh`, akkor azzal, ha van `<p>_debian.txt` fájl akkor pedig a hagyományos módon. Ha hibát észlel, leáll, ha pedig egy projektnek nem talál semmilyen függőséget, figyelmeztetést ad. Több futási idejű üzenet a 2.7.5. alfejezetben olvasható.

Konténer futtatása

A konténer futtatásakor futó script megírása a legösszetetteb feladat volt a tesztkörnyezet implementálása során. Ennek a fájlnak, ami a `test-clang-tools` könyvtárban `start.sh` néven található ugyanis képesnek kell lennie számos művelet elvégzésére. Ilyen műveletek az engedélyezett projektek listázása, a projektek beállítása (azaz letöltése, konfigurálása és buildelése), a projektek elemzése, valamint a projektek mappáinak törlése.

A script működése során az újrafelhasználhatóság és az áttekinthetőség érdekében saját függvényeket használtam. A legfontosabb függvények és rövid leírásuk a 3.2. táblázatban olvashatók. A script során a konténer futtatásakor átadott argumentumokon kívül három fontos

változó van. Az egyik a `data`, mely egy kétdimenziós tömb, és a kiválasztott projektek neveit, valamint a hozzájuk tartozó git linket tárolja. A másik kettő a `projects` és a `checkers`, amik a megadott projektek és a checkerek neveit tárolják, azonban már nem szövegként hanem tömbként.

Függvény neve	**Függvény argumentuma**	**Függvény leírása**
list_projects	-	Konténerben engedélyezett projektek kilistázása
get_all_projects	-	Összes projekt és azok linkjeinek kiolvasása a data tömbbe
get_chosen_projects_link	-	Az argumentumként átadott projektek és azok linkjeinek kiolvasása a data tömbbe
update_repo	<p>	<p> projekt könyvtárának letöltése vagy frissítése, és beállítása a legfrissebb tagre
configure_project	<p>	<p> projekt konfigurálása automatikusan
codechecker_config	-	CodeChecker projekt konfigurálása
codechecker_log	<p>	CodeChecker log parancsával a <p> projekt buildelése
setup_checkers	-	Megadott checkerek alapján a CodeChecker argumentumának beállítása
codechecker_analyze	<p>	CodeChecker elemzésének futtatása a <p> projekten
delete_projects	-	Összes projekt mappájának törlése
check_checkers	-	Argumentumként megadott checkerek ellenőrzése
check_args	-	Konténer futtatásakor csatolt mappák ellenőrzése

3.2. táblázat. A `start.sh` script fontosabb függvényei

A `start.sh` script során számos futási idejű üzenettel találkozik a felhasználó, melyekről a 2.7.3. alfejezetben lehet bővebben olvasni. A program ugyanakkor számos kommenttel rendelkezik, segítve ezzel a leendő fejlesztőt az eligazodásban. A program teljes működésének logikája a 7. algoritmus alapján történik.

A működés teljes megértéséhez fontos szót ejteni a CodeChecker [14] projektről. Ennek fő célja, hogy egy nagy projekt összes C és C++ fájlját elemezze a Clang Static Analyzer vagy a Clang-Tidy checkereivel. Ehhez először a projektet konfigurálni kell. Ezután a CodeChecker megfelelő beállítása után a nagy projekt buildelését a `CodeChecker log` paranccsal kell elvégezni. Ez buildeli a teljes projektet, a szkriptemben a `make` paranccsal. Emiatt a teszt-

Algoritmus 7. *start.sh*

```
Argumentumok megfelelő beállítása
if list then
    list_projects() return
end if
if projects == "all" then
    get_all_projects()
else
    check_projects_arg.sh meghívása
    get_chosen_projects_link()
end if
if setup then
    check_args
    for p in projects do
        update_repo(p)
    end for
    CodeChecker törlése a projektek közül
    for p in projects do
        configure_project(p)
    end for
    codechecker_config
    for p in projects do
        codechecker_log(p)
    end for
end if
if analyze then
    check_args()
    if not checkers == "all" then
        check_checkers()
    end if
    CodeChecker beállítása
    setup_checkers()
    for p in projects do
        codechecker_analyze(p)
    end for
end if
if delete then
    delete_projects()
else
    for p in projects do
        Jogosultságok beállítása p projekthez
    end for
end if
```

környezet csak olyan projektekre futtatható, melyek buildelhetők így.

A buildelés során a CodeChecker létrehoz egy `compilation.json` fájlt is a projekt főmappájában. Ez a fájl tartalmazza a projektben található összes C és C++ fájlról a következőket: melyik mappában található, milyen paranccsal és kapcsolókkal kell fordítani, valamint mi a fájl neve. Egy részlet a bitcoin projekthez tartozó `compilation.json` fájlból a 3.14. ábrán látható. Ezen fájl alapján a CodeChecker képes elemezni a projekteket a `CodeChecker analyze` paranccsal és a megfelelő kapcsolókkal. Végül a `CodeChecker parse` parancs segítségével lehet felhasználóbarát HTML kimenetet generálni a hibákból, ahogy az a `start.sh` scrip futtatása során is történik.

```
1  [{
2  "directory": "/testDir/bitcoin/src",
3  "command": "/usr/bin/g++ -std=c++11 -DHAVE_CONFIG_H -I.
4  -I../src/config -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=2
5  -I. -DBOOST_SP_USE_STD_ATOMIC -DBOOST_AC_USE_STD_ATOMIC
6  -pthread -I/usr/include -I./leveldb/include
7  -I./leveldb/helpers/memenv -I./secp256k1/include
8  -I./univalue/include -pthread -DHAVE_BUILD_INFO
9  -D__STDC_FORMAT_MACROS -fstack-reuse=none -Wstack-protector
10 -fstack-protector-all -Wall -Wextra -Wformat -Wvla
11 -Wswitch -Wredundant-decls -Wunused-variable -Wdate-time
12 -Wno-unused-parameter -Wno-implicit-fallthrough -fPIE
13 -g -O2 -MT libbitcoin_server_a-shutdown.o -MD -MP -MF
14 .deps/libbitcoin_server_a-shutdown.Tpo -c -o
15 libbitcoin_server_a-shutdown.o shutdown.cpp",
16 "file": "shutdown.cpp"
17 },
```

3.14. ábra. A bitcoin projekthez tartozó `compilation.json` fájl tartalma (részlet)

3.7. Tesztelés

Mind a checkerek, mind a tesztkörnyezet elkészítése után fontos rész az elkészült eszköz tesztelése. A checkerek esetén elsősorban azért, mert a rosszul elkészült checker jobb esetben sosem ad hibajelzést, rosszabb esetben viszont sok hamis pozitív jelzést ad. A programozó pedig sok hamis pozitívnál előbb-utóbb abbahagyja az eszköz használatát, ami még nagyobb sebezhetőséghez vezet.

A tesztkörnyezet tesztelése ugyancsak nagyon fontos. Mivel itt nagyobb projektek

elemzéséről van szó, létfontosságú, hogy megbizonyosodjak eszközöm helyességéről, ugyanis kezdő programozó a környezet által végzett sok művelet közül könnyen lehet, hogy keveset ért, ezért nem biztos, hogy képes lenne az eszköz hibáit észrevenni és kijavítani.

3.7.1. A checkerek tesztelése

Az LLVM könyvtár saját tesztelési környezetet biztosít a különböző projektjei számára. Nevezetesen a Clang-Tidy projekt tesztfájljai megtalálhatók az `/llvm-project/clang-tools-extra/test/clang-tidy/checkers` mappában[1]. A projekt beállításánál (`llvm-project/build` mappában) korábban említettem (2.5. alfejezet), hogy a `ninja clang-tidy` paranccsal buildelhető a projekt. A tesztesetek futtatására a 3.15. ábrán látható parancsok kiadásával van lehetőség. Ekkor a környezet architektúratesztet futtat az egyes checkerek tesztelésével, ezáltal tesztelve a komponenseit is.

```
1 ### clang-tools-extra tesztelése
2 ### (mely tartalmazza a Clang-Tidyt):
3 $ ninja check-clang-tools
4 ### egész llvm-project tesztelése:
5 $ ninja check-all
```

3.15. ábra. A Clang-Tidy tesztjeinek futtatása

A tesztkörnyezet nagyban megkönnyíti a tesztelés menetét. A különböző checkerekre külön-külön tesztfájlok vonatkoznak (egyesekre több is), melyek a chekerek megfelelő működését hivatottak ellenőrizni. A checkerek tesztelése során el kellett érni, hogy a tesztfájlok fordíthatók legyenek az llvm környezeten belül. Ehhez sok függvényt forward deklarálni kellett, melyre bővebben nem térek ki.

bugprone-bad-signal-to-kill-thread

A futtatott tesztfájl az `llvm-project/clang-tools-extra/test/clang-tidy/` mappában található `bugprone-bad-signal-to-kill-thread.cpp` fájl. A tesztesetek során a korábban említett függvények meghívása kerül ellenőrzésre különböző szignálokkal. A tesztesetek jellemzői és a tesztelés eredménye a 3.3. táblázatban találhatók.

[1] Pár korábbi checker, mint az általam fejlesztett bugprone-bad-signal-to-kill-thread checker is az egyel feljebbi `test` mappában helyezkednek el. Erre a tesztelés ugyan úgy működik.

Kód sora	Vizsgált függvény	Meghívott szignál	Várt eredmény	Kapott eredmény	Tesztelés eredménye
23	`pthread_kill()`	SIGTERM	figyelmeztetés	figyelmeztetés	PASS
31	`pthread_kill()`	SIGINT	nincs figyelmeztetés	nincs figyelmeztetés	PASS
33	`pthread_kill()`	0xF	figyelmeztetés	figyelmeztetés	PASS

3.3. táblázat. A bugprone-bad-signal-to-kill-thread checker tesztelésének eredménye

bugprone-do-not-refer-atomic-twice

A futtatott tesztfájl az `llvm-project/clang-tools-extra/test/clang-tidy/checkers/` mappában található `bugprone-do-not-refer-atomic-twice.cpp` fájl. A tesztelés során különböző módon deklarált különböző típusú atomikus változók kerülnek tesztelésre. Amennyiben egy operátor két oldalán ugyanaz az atomikus változó szerepel, a többszálú biztonság már nem áll fent, ezért a checker hibajelzést ad. Az előre megírt tesztfájl futtatása a 3.16. ábrán látható módon futtatható. A tesztek eredményei a 3.4. táblázatban találhatók.

```
1 $ cd clang-tools-extra/test/clang-tidy/checkers
2 $ clang-tidy -checkers=-*,bugprone-do-not-refer-atomic-twice\
3 >    bugprone-do-not-refer-atomic-twice.cpp
```

3.16. ábra. A bugprone-do-not-refer-atomic-twice checker futtatása

A tesztesetek a következő szempontok figyelembevételével lettek összeállítva:

- az atomikus változókat többféle módon is lehet deklarálni, mégpedig például az int típus esetén a 3.17. ábrán látható módokon,
- a checkernek különböző atomikus változókra működnie kell (pl.: `atomic_bool`, `atomic_int`, stb.),
- amennyiben egy bináris operátor két oldalán ugyanaz az atomikus változó áll, figyelmeztetést kell kapni,
- egyéb esetekben (egyszerű értékadás, unáris operátorok) nem szabad figyelmeztetést kapni.

A fenti esetek több kombinációja lefedésre került a tesztesetek által. Az int típus példáján keresztül ellenőrzésre kerültek a többfajta deklaráció felismerései (pl.: 1-3. teszteset). A `bool` típusú `b` változóval ellenőrzésre került, hogy a checker nem csak egész számokra műkö-

Teszt sorszáma	Kód sora	Várt figyelmeztetés	Tapasztalt figyelmeztetés	Teszt eredménye
1	13	igen	igen	PASS
2	15	igen	igen	PASS
3	17	igen	igen	PASS
4	19	igen	igen	PASS
5	24	igen	igen	PASS
6	29	igen	igen	PASS
7	34	igen	igen	PASS
8	39	igen	igen	PASS
9	44	nem	nem	PASS
10	45	nem	nem	PASS
11	46	nem	nem	PASS
12	47	nem	nem	PASS
13	51	nem	nem	PASS
14	55	nem	nem	PASS
15	59	nem	nem	PASS
16	63	nem	nem	PASS
17	67	nem	nem	PASS
18	68	nem	nem	PASS
19	69	nem	nem	PASS
20	70	nem	nem	PASS
21	74	nem	nem	PASS
22	75	nem	nem	PASS
23	76	nem	nem	PASS

3.4. táblázat. A bugprone-do-not-refer-atomic-twice checker tesztelésének eredménye

```
1 atomic_int n;
2 _Atomic int n2;
3 _Atomic(int) n3;
```

3.17. ábra. Ugyanazon tulajdonságú int atomikus változó három különböző deklarációs módja

dik. A különböző bináris operátorok tesztelésével pedig ellenőrizve lett, hogy megfelelő hívás esetén nem történik hamis pozitív jelzést.

bugprone-signal-in-multithreaded-program

A tesztelés a különböző checker argumentumok miatt alapból négy fájlt tartalmaz, melyek a korábban említett mappában a következők:

1. `bugprone-signal-in-multithreaded-program-noconfig-std::thread.cpp`
2. `bugprone-signal-in-multithreaded-program-config-std::thread.cpp`

3. `bugprone-signal-in-multithreaded-program-noconfig-thrd_create.cpp`
4. `bugprone-signal-in-multithreaded-program-config-thrd_create.cpp`

Mivel a Clang-Tidy tesztelése esetén alapértelmezettként nincs lehetőség figyelmeztetéssel nem rendelkező fájl tesztelésére, a projekt főmappájában létrehozott `temp` mappában található további négy fájllal tesztelhető az esetleges hamis pozitív jelzés:

6. `bugprone-signal-in-multithreaded-program-no_signal.cpp`
7. `bugprone-signal-in-multithreaded-program-no_thread.cpp`
8. `bugprone-signal-in-multithreaded-program-badconfig-std::thread.cpp`
9. `bugprone-signal-in-multithreaded-program-badconfig-thrd_create.cpp`

A tesztek során ellenőrzésre kerül, hogy a különböző thread hívások esetén a checker konfigurációja nélkül (1. és 3. teszt), valamint azt megfelelően beállítva (2. és 4. teszt) adódik-e hibajelzés. Ezután ellenőrzésre kerül, hogy amennyiben nincs szignál vagy thread hívás a programban, nem történik jelzés (5. és 6. teszt). Legutoljára tesztelt, hogy a checker konfigurálása működik, ugyanis rossz argumentumok esetén nem történik jelzés (7. és 8. teszt). A tesztek eredménye a 3.5. táblázatban látható.

Teszt sorszáma	**Várt figyelmeztetés**	**Tapasztalt figyelmeztetés**	**Teszt eredménye**
1	igen	igen	PASS
2	igen	igen	PASS
3	igen	igen	PASS
4	igen	igen	PASS
5	nem	nem	PASS
6	nem	nem	PASS
7	nem	nem	PASS
8	nem	nem	PASS

3.5. táblázat. A bugprone-signal-in-multithreaded-program checker tesztelésének eredménye

bugprone-spuriously-wake-up-functions

Ez a checker másképp működik C és C++ programokra, ezért a tesztelésre két külön tesztfájlt használok. A C fájl esetén (`bugprone-spuriously-wake-up-functions.c`) a következő függvényhívásokat kell ellenőrizni: `cnd_wait()`, `cnd_timedwait()`. A függvényhívásokra akkor történik hibajelzést, ha nem `while`, `dowhile`, vagy `for` ciklusban szerepelnek. Ennek

számos kombinációja lehetséges, ezért számos tesztesetet ki kell próbálni. A tesztek eredménye a 3.6. táblázatban látható.

A C++-ra vonatkozó tesztesetek a `bugprone-spuriously-wake-up-functions.cpp` fájlban találhatók. Itt további lehetőség, hogy a függvényeket el lehet látni lambda függvény paraméterrel, tehát ezt is ellenőrizni kell, és csak akkor hibát várni, ha a függvénynek nincs a feltétel ellenőrzése lambda függvényként átadva, valamint `while`, `dowhile`, és `for` ciklussal sincs védve. A C++ tesztekre vonatkozó eredmények a 3.7. táblázatban láthatók.

Egyéb tesztek

Ahogy bemutattam, a `ninja test-all` parancs futtatása során az egész rendszer, valamint az egyes modulok, a checkerek megbízhatóságáról is visszajelzést lehet kapni. Ezentúl a tesztesetek kézzel való futtatása során látható, hogy ezek stabilan és megfelelően működnek.

Szeretnék azonban visszajelzést kapni arról is, hogy az új checkereim nem növelik meg jelentősen a Clang-Tidy elemzésének idejét. Mivel a Tidy-ban számos checker van, és az elemző egyik előnye a szimbolikus elemzést végző Clang Static Analyzerrel [11] szemben, hogy jóval gyorsabb, természetesen ezt a gyorsaságot nem előnyös néhány új checker kedvéért elrontani. Bár a megfelelő sebességet a checkerek algoritmusából sejteni lehet, szeretnék erről konkrétabban is megbizonyosodni.

Ehhez futtatom az elkészített tesztkörnyezetet. A futtatás során a bitcoin [15] projekt elemzését vizsgálom. Ehhez véletlenszerűen kiválasztok négy checkert, legyenek a következők:

- android-cloexec-accept
- bugprone-signed-char-misuse
- cert-dcl50-cpp
- cert-msc32-c

Az ezen checkerek elemzésének gyorsaságát pedig összevetem az általam létrehozott négy checker gyorsaságával. A 3.18. ábrán látható parancsokat egymás után futtatva a Codechecker az elemzés végén kiírja, hogy mennyi időbe telt az elemzés. Fontos, hogy a két parancsot ugyanabban a környezetben futtattam, hiszen a gyorsaság nagyban függ a számítógép tulajdonságaitól.

Az elemzés során azt találtam, hogy a véletlenszerűen kiválasztott 4 checker elemzési ideje a CodeChecker visszajelzése alapján 149 másodperc volt, míg az általam megírt checkereké 152 másodperc. Természetesen ez a gyorsaság számítógépenként eltérő lehet, de ebből

Teszt sorszáma	**Kód sora**	**Várt figyelmeztetés**	**Tapasztalt figyelmeztetés**	**Teszt eredménye**
1	27	igen	igen	PASS
2	32	igen	igen	PASS
3	36	igen	igen	PASS
4	39	nem	nem	PASS
5	43	nem	nem	PASS
6	46	nem	nem	PASS
7	49	igen	igen	PASS
8	53	igen	igen	PASS
9	56	nem	nem	PASS
10	59	nem	nem	PASS
11	62	nem	nem	PASS
12	66	nem	nem	PASS
13	70	nem	nem	PASS
14	74	nem	nem	PASS
15	77	nem	nem	PASS
16	80	nem	nem	PASS
17	84	nem	nem	PASS
18	87	nem	nem	PASS
19	90	nem	nem	PASS
20	93	nem	nem	PASS
21	96	igen	igen	PASS
22	101	igen	igen	PASS
23	105	igen	igen	PASS
24	108	nem	nem	PASS
25	112	nem	nem	PASS
26	115	nem	nem	PASS
27	118	igen	igen	PASS
28	122	igen	igen	PASS
29	125	nem	nem	PASS
30	128	nem	nem	PASS
31	131	nem	nem	PASS
32	135	nem	nem	PASS
33	139	nem	nem	PASS
34	143	nem	nem	PASS
35	146	nem	nem	PASS
36	149	nem	nem	PASS
37	153	nem	nem	PASS
38	156	nem	nem	PASS
39	159	nem	nem	PASS
40	162	nem	nem	PASS

3.6. táblázat. A bugprone-spuriously-wake-up-functions C fájlra vonatkozó tesztelésének eredménye

Teszt sorszáma	**Kód sora**	**Várt figyelmeztetés**	**Tapasztalt figyelmeztetés**	**Teszt eredménye**
1	122	igen	igen	PASS
2	127	nem	nem	PASS
3	131	nem	nem	PASS
4	135	nem	nem	PASS
5	140	nem	nem	PASS
6	146	nem	nem	PASS
7	152	nem	nem	PASS
8	158	igen	igen	PASS
9	162	nem	nem	PASS
10	165	nem	nem	PASS
11	168	nem	nem	PASS
12	171	nem	nem	PASS
13	176	igen	igen	PASS
14	180	nem	nem	PASS
15	183	nem	nem	PASS
16	186	nem	nem	PASS
17	189	nem	nem	PASS

3.7. táblázat. A bugprone-spuriously-wake-up-functions C++ fájlra vonatkozó tesztelésének eredménye

látható, hogy nagyságrendileg a négy checkerem azonos idő alatt elemez, mint másik négy checker. Ebből pedig le lehet vonni azt a következtetést, hogy a fejlesztett checkerek nem növelik meg jelentősen a Clang-Tidy elemzési idejét.

3.7.2. A tesztkörnyezet tesztelése

Mivel a programkódot ismerem, ezért úgynevezett fehérdoboz tesztelést végzek, azaz a program kódjának ismeretében tesztelem annak egyes funkcióit. Ezeknél a teszteknél külön odafigyelek, hogy a specifikációban meghatározott alapkövetelményeket teszteljem. Szempont ezentúl a futás gyorsasága, valamint megbízhatósága is, azaz ki fogom próbálni az eszköz gyorsaságát, és hogy nem várt bemenetekre mit eredményez.

Mivel a környezet egy virtuális számítógépen (konténerben) fut, és bash szkriptekből áll, ezért a checkerekhez hasonló tesztelő nem áll rendelkezésre. Emiatt a tesztelés során mindig meghatározom az elvárt működést vagy kimenetet, és ezt vizsgálom.

A tesztelés futtatását egy teljesen új, üres mappában kezdem, valamint egy teljesen új tesztelési mappát használok. A teszteket logikusnak tűnne funkcionalitás szerint csoportosítani. Ezt azonban nem igazán lehet megtenni, ugyanis például az image megfelelő buildelése nem

```
$ docker build . -t clang-test-bitcoin \
>  --build-arg projects=bitcoin
$
$ docker run -e "analyze=FALSE" \
>  -v $testDir:/testDir -v $llvm:/llvm-project \
>  clang-test-bitcoin
$
$ docker run -e "setup=FALSE" \
>  -v $testDir:/testDir -v $llvm:/llvm-project \
>  -e "checkers=android-cloexec-accept,\
>bugprone-signed-char-misuse,\
>cert-dcl50-cpp,\
>cert-msc32-c" \
>  clang-test-bitcoin
$
$ docker run -e "setup=FALSE" \
>  -v $testDir:/testDir -v $llvm:/llvm-project \
>  -e "checkers=bugprone-bad-signal-to-kill-thread,\
>bugprone-do-not-refer-atomic-twice,\
>bugprone-signal-in-multithreaded-program,\
>bugprone-spuriously-wake-up-functions" \
>  clang-test-bitcoin
```

3.18. ábra. A checkerek gyorsaságának vizsgálata

ellenőrizhető máshogy, csak a visszajelző üzenetekkel (ami nem túl megbízható, hiszen attól, hogy nem ír ki hibát, még nem biztos a helyes működése), vagy a rá épülő konténer megfelelő futásával. Éppen ezért a tesztelésnél más logikát követek, amivel igyekszem minél több tesztesetet lefedni. A tesztelés során a buildelt `llvm-project` mappa elérését az `$llvm`, a tesztelési mappa elérését a `$testDir` változókban tárolom. A tesztek futtatása a projektek a 3.8. táblázatban látható verzióján történt. Amennyiben a projektből újabb release jön ki, előfordulhat, hogy a projekt függőségeit ismételten meg kell adni a 2.7.7. alfejezetnek megfelelően.

Buildelési argumentumok ellenőrzése

A művelet során ellenőrzöm, hogy a buildelési argumentumok megfelelően működnek. Azaz pontosan azok a projektek lesznek telepítve, amelyek megadásra kerültek, valamint az alapértelmezett értékek megfelelően működnek. A tesztek során ezen részben mindig buildelem az image-et is és futtatom a konténert is, utóbbit az alapértelmezett módon. A tesztelés várt eredménye a konténer futtatása utáni állapotra vonatkozik. A tesztek eredménye a 3.9. táblázatban látható, a futtatott parancsok pedig a projekt főmappájában található `test_env_tests.sh`

Projekt	Release név
bitcoin	v0.20.0rc1
codechecker	v6.11.1
curl	curl-7_70_0
ffmpeg	n2.8.16
memcached	1.6.6
nginx	release-1.18.0
postgres	REL9_5_22
redis	6.0.1
tmux	3.1b
openssl	openssl-3.0.0-alpha1
git	v2.26.2
vim	v8.2.0750
cpp-taskflow	v2.4.0
enkiTS	v1.8
RaftLib	a9
FiberTaskingLib	- (master)

3.8. táblázat. Projektek tesztelt verziója

fájlban érhetők el.

Ezzel ellenőriztem azt is, hogy alapvetően a projektek letöltése, konfigurálása és buildelése is sikeres, hiszen ezek nélkül egy tesztelés sem járt volna sikerrel, illetve a környezet hibát jelzett volna, ami nem történt.

Futtatási argumentumok ellenőrzése

A következő tesztek során ellenőrzöm, hogy a futtatás során valóban meg tudom változtatni az alapértelmezett, jól működő argumentumokat. Az előzőhöz hasonló teszteket végzek, azonban most egy, előre felépített image-re építve dolgozok. Ez az image rendelkezzen az eddigiekhez hasonló `test-clang` névvel, az alapértelmezett argumentumokkal, kivéve a projects változóját, ez legyen beállítva a cpp-taskflow és a postgres projekt elemzésére. A tesztek leírása és eredménye a 3.10. táblázatban található.

Projekt hozzáadásának tesztelése

A projekt hozzáadását úgy tesztelhetem, hogy elvégzem egy konkrét projekt esetén. Legyen ez a felhasználói dokumentációban is bemutatott SRT [58] projekt. A hozzáadást elvégzem a 2.7.3. alfejezetben leírt módon, majd ellenőrzöm a 3.11. táblázatban szereplő kitételeket.

Sor-szám	projects argumentum	Egyéb argumentumok	Várt eredmény	Teszt eredménye
1	FiberTaskingLib		Image sikeresen buildel	PASS
1	FiberTaskingLib		Konténer futása hiba nélkül megy végbe	PASS
1	FiberTaskingLib		A testDir kizárólag a codechecker valamint a FiberTaskingLib projektek mappái, és egy report mappa találhatók	PASS
1	FiberTaskingLib		A testDir/resports/ FiberTaskingLib/html mappában sikeres elemzés található	PASS
1	FiberTaskingLib		Az elemzés eredménye többféle típusú checker által adott hibát is talál	PASS
2	enkiTS	delete=TRUE	A konténer futtatása után a testDir mappában nincs enkiTS mappa	PASS
2	enkiTS	delete=TRUE	A testDir/resports/ enkiTS/html mappában sikeres elemzés található	PASS
3	FiberTaskingLib	setup=FALSE, checkers=cert-err58-cpp, delete=TRUE	A FiberTaskingLib konfigurálásáról nem jelenik meg üzenet	PASS
3	FiberTaskingLib	setup=FALSE, checkers=cert-err58-cpp, delete=TRUE	Az elemzés után a testDir mappában nem található FiberTaskingLib mappa	PASS
3	FiberTaskingLib	setup=FALSE, checkers=cert-err58-cpp, delete=TRUE	Az elemzés eredményében kizárólag a cert-err58-cpp checker eredményei láthatók	PASS
4	tmux	analyze=FALSE	A testDir mappában megtalálható a tmux mappája	PASS
4	tmux	analyze=FALSE	A testDir/resports/tmux mappa nem létezik	PASS
4	tmux	analyze=FALSE	A testDir/tmux mappa tartalmaz etc mappát	PASS

3.9. táblázat. A tesztkörnyezet buildelési argumentumainak tesztelése

Projekt törlésének tesztelése

A projekt törlését hasonlóan tesztelhetem a hozzáadáshoz. Ezt ugyancsak SRT projekt esetében vizsgálom. Ahogy korábban bemutattam, a projekt hozzáadásakor a következő módosítások

Sor-szám	projects argumentum	Egyéb argumentumok	Várt eredmény	Teszt eredménye
5	postgres		Kizárólag a postgres projekt kerül letöltésre és elemzésre	PASS
6	cpp-taskflow	delete=TRUE	A cpp-taskflow projekt elemzésre kerül, majd törlődik	PASS
7		list=TRUE	A postgres és cpp-taskflow projekt kiírásra kerül	PASS
8	postgres	setup=FALSE, checkers=cert-env33-c	A postgres projekt elemzésre kerül a korábbi konfigurálással, a megadott checker szerint	PASS
9	postgres,cpp-taskflow	checkers=cert-env33-c, delete=TRUE	A két projekt ismét beállításra és elemzésre kerül, majd mappáik törlődnek	PASS

3.10. táblázat. A tesztkörnyezet futtatási argumentumainak tesztelése

Történés	Várt eredmény	Teszt eredménye
A projekt git linkje hozzáadásra került a környezethez	A név és a link megjelenik a project_links.txt fájlban	PASS
A projektnek került megadásra függősége	A függőségek mentésre kerülnek a requirements/ srt_debian.txt fájlban	PASS
A projektnek nem került speciális beállító fájlja megadásra	Nincs srt_setup.sh nevű fájl a setup_files mappában	PASS
A hozzáadás sikeres	A projektet lehetséges automatikusan elemezni	PASS

3.11. táblázat. Projekt hozzáadásának tesztelése

történtek:

- `reuqirements/srt_debian.txt` létrejött,
- `project_links.txt`-ben az srt sora hozzáadásra került.

Törlöm a projektet, majd ellenőrzöm ezen fájlokat. Ezeket ellenőrizve azt találtam, hogy mindkét fájlban elvégezte a megfelelő műveletet. Ha ezután elemezni szeretném az srt projektet, hibát kapok, ami ismét megfelel a várakozásaimnak.

Hibafelismerés tesztelése

A hibaüzenetekről részletesen a 2.7.5. alfejezetben lehet olvasni, azonban a tesztelés során is szeretném ellenőrizni, hogy ezek rendben működnek. Emiatt a 3.12. táblázat alapján kifejezetten ezek előidézésére írt, nem megfelelő használatot alkalmazok, és a kimenetet vizsgálom. Itt a tesztet akkor minősítem sikeresnek, ha a leírt működés esetén error üzenetet kapok, és a működés leáll az adott ponton.

Sorszám	Történés	Teszt eredménye
10	Futtatásnál a projects argumentumba nem létező név írása	PASS
11	Futtatásnál a checker argumentumba nem létező checker írása	PASS
12	Futtatásnál nem történik testDir mappa megadása	PASS
13	Futtatásnál nem történik llvm-project mappa megadása	PASS
-	Az SRT projekt hozzáadásánál nem kerülnek megadásra a függőségek, majd elemzés indítása	PASS
14	Image buildelésénél a project argumentumbba nem létező név írása	PASS

3.12. táblázat. Hibaüzenetek tesztelése

Egyéb tesztek

Bemutattam tehát, hogy a tesztkörnyezet megfelelően működik számos tesztestre. Most szeretném megvizsgálni, hogy a futási idővel mi a helyzet. A tesztelés során a git projektet állítom be és elemzem kézzel, igyekezve, hogy minél gyorsabb legyek, majd ezután a környezet segítségével teszem meg ugyanezt. Ezzel szeretném ellenőrizni, hogy sikerült-e elérni célomat, azaz a tesztkörnyezet legalább olyan gyors-e, mintha manuálisan futtatnám a parancsokat. A futtatást természetesen ugyanazon a számítógépen, ugyanolyan internetkapcsolat esetén végzem.

Ekkor azt tapasztaltam, hogy a kézzel futtatott (de előre összekészített) parancsok körülbelül 15 perc alatt kerültek végrehajtásra. Ehhez persze hozzá kell tenni, hogy ez nem gyakorlott programozó esetén jóval több időt vehet igénybe, arról nem is beszélve, hogy egy projekt első konfigurálása sok programozónak gondot okozhat. Sajnos a CodeChecker használata sem teljesen magától érthetődő, és néhányszor végig kell olvasnia az embernek a használati útmutatót, mire biztos lehet a működésében.

Ezzel szemben a tesztkörnyezet ugyanezen projekt ugyanilyen tesztelését mindössze 10,5 perc alatt hajtotta végre (a buildelés indításától az elemzés futtatásának befejeztéig). Ez

összességében azt jelenti, hogy nem csupán egy nagy segítséget nyújtó eszközt sikerült létrehozni, hanem időgazdálkodás szempontjából is sikerült nagy előrelépést elérni. A méretet tekintve pedig megemlítendő a tervezésnél leírtak fontossága (3.5.1. alfejezet): ezt az időspórolást körülbelül plusz 60 MB tárhelyért cserébe kapja a programozó.

Fontos megjegyezni, hogy a git projekt a nagy projektek között is nagynak számít. Ez azt jelenti tehát, hogy valószínűleg nem lesz olyan projekt aminek az automatikus beállítása és tesztelése lényegesen több időt venne igénybe ennél. A tesztelés során lefuttattam a környezetet az összes rendelkezésre álló projektre, ami összesen (ismét buildeléssel, konfigurálással és teszteléssel együtt) 2 órát vett igénybe. Ez természetesen soknak tűnhet, azonban ezt úgy kaptam, hogy az összes checker engedélyezve volt.

A valódi használat során általában kizárólag a saját checkerét fogja a programozó futtatni a projekteken, valamint nem lesz szüksége azok beállítására minden alkalommal. További tesztelések és azok hozzávetőleges ideje a 3.13. táblázatban láthatók. Fontos megjegyezni, hogy a tesztelés a követelményeknél említett (2.4. fejezet) számítógépen futott. Ez tehát azt jelenti, hogy ez bár nagyságrendileg fontos lehet mindenkinek (melyik folyamat relatív mennyi ideig tart), azonban gyengébb számítógépnél ennél jóval hosszabb időre kell számítani.

A fentebb említett teszteseteken kívül az összes alapértelmezetten rendelkezésre álló projekt működése ki lett próbálva egyesével a korábban felsorolt verziók esetén.

Tesztelés leírása	**Tesztelés ideje**
Összes projekt esetén image buildelése, projektek beállítása (letöltése, konfigurálása, buildelése), projektek elemzése az összes checkerrel	2 óra 8 perc
Összes projekt esetén image buildelése (függőségeik telepítése)	7,5 perc
Összes projekt beállítása (letöltése, konfigurálása, buildelése)	38 perc
Összes projekt összes checkerrel való elemzése	1 óra 35 perc
Összes projekt bugprone-spuriously-wake-up-functions checkerrel való elemzése	13,5 perc
Egy projekt (git) esetén image buildelése, projekt beállítása (letöltése, konfigurálása, buildelése), projekt elemzése az összes checkerrel	10,5 perc
Egy projekt esetén (git) projekt elemzése összes checkerrel	3 perc
Egy projekt esetén (git) projekt elemzése a bugprone-spuriously-wake-up-functions checkerrel	35 másodperc

3.13. táblázat. A tesztkörnyezet futási idejének tesztelése

3.8. Fejlesztési lehetőségek

A Clang-Tidy statikus elemző fejlesztése talán egy véget nem érő folyamat. Ahogy új lehetőségek jelennek meg a C és C++ nyelvekben új és új kihívásokra kell választ találniuk az elemző fejlesztőinek. Az általam létrehozott checkerek egy része már átment a review-n, ezért először azt lehetne gondolni, hogy nem szorul további javításokra. Ugyanakkor fontos tudni magának a Clang-Tidy-nak a korlátairól: mivel szimbolikus kiértékelésre nem képes, ezért egyes problémákat (pl.: atomikus változókkal kapcsolatos szabály) egyszerűen nem lehetséges maradéktalanul megoldani vele. Összességében tehát úgy gondolom, hasznos lenne egy másik, szimbolikus kiértékeléssel működő statikus elemzőre implementálni ezen szabályok felismerését, és összehasonlítani az általam megírt checkerekkel mind gyorsaság, mind hatékonyság szempontjából.

A tesztkörnyezet fejlesztése kevésbé egyértelmű feladat. Úgy gondolom, hogy mindig lesz „csak még egy" projekt, amit szeretne a programozó az elemzéshez hozzáadni, azonban ehhez egy kis plusszal ki kell egészíteni az eszközt. Elsősorban ez a kis plusz lehet az, hogy automatikusan legyen képes más konfigurálási beállításokat elvégezni, vagy a CodeChecker buildelését a GNU Make-től különböző programmal megtenni. Ezentúl fontos fejlesztési lehetőség lehetne, hogy a Clang-Tidy checkereinek legyen lehetőség plusz argumentumot megadni, vagy a projektek release-ét kiválasztani.

Érdekes vizsgálat lenne egy olyan eszköz létrehozása is, mely elkezdi konfigurálni és buildelni a projekteket, majd ha hibát tapasztal, az error üzenet alapján automatikusan megkeresi és letölti a szükséges függőséget, majd ismét megpróbálkozik a buildeléssel. Ehhez persze már komolyabb vizsgálatokra és internetes fórumok automatikus feldolgozására is szükség lenne, ami miatt ez inkább egy nagyobb kutatási témának, mint egy szakdolgozatnak tűnik.

4. fejezet

Összegzés

Szakdolgozatom során egyrészt sikeresen bővítettem az LLVM Clang-Tidy [3] statikus elemzőt négy olyan checkerrel, melyek a SEI CERT Coding Standards [2] öt, konkurens programozással kapcsolatos szabályának ellentmondó kódot ismerik fel automatikusan. Ezekkel kapcsolatban fontos megjegyezni, hogy a Clang-Tidy működéséből adódóan nem lehetséges velük minden sebezhetőséget tökéletesen felismerni (pl.: atomikus változók esetén az adatáramlás vizsgálatára nincs lehetőség), azonban törekedtem a lehetőleg nulla hamis pozitív jelzésre. Ez más szavakkal azt jelenti, hogy a checkereim valószínűleg nem vélnek hibának olyan programrészletet, ami igazabból nem az.

A checkerek fejlesztése során szükségem volt a Clang fordító [12] absztrakt szintaxisfájának ismeretére, az LLVM projekt [27] beépített függvényeinek alapvető használatára, valamint az ugyancsak Clang könyvtárban megtalálható AST matcherek stabil felhasználására. Ezentúl a checkerek fejlesztésekor bővítenem kellett a hozzájuk tartozó dokumentációt is, ezzel biztosítva a jövőbeni felhasználók könnyű eligazodását.

A checkerek egy része már elfogadásra került, és megtalálható az LLVM projektben, egy része pedig review alatt áll. Ami közös bennük, hogy mindegyik teljes mértékben működik, és az elvárásaimnak megfelelő eredményeket generál a megírt tesztesetekre, valamint nagyobb projekteken lefuttatva sem adnak hamis pozitív jelzést, ami mindenképpen fontos előnyük.

Munkám második részében a Clang-Tidy-ban fejlesztett checkerek nehezen tesztelhetőségére kerestem a választ a CodeChecker [14] projekt segítségével. A fejlesztés során létrehoztam egy olyan tesztkörnyezetet - gyakorlatilag egy konténerben futó szkriptsorozatot -, mely képes nagy méretű projektek teljesen automatizált elemzésére a megadott checkerek alapján.

A környezet lehetővé teszi nem csak a projektek automatikus elemzését - amire a

CodeChecker ad lehetőséget -, hanem az előkészítő műveletek teljes automatizálását is. Ez azt jelenti, hogy az említett konténerre - pontosabban az ennek alapját képező image-re - a projektek függőségei automatikusan települnek. Ezután futtatáskor a projekt automatikusan letöltésre, majd konfigurálásra és buildelésre kerül. Ez nagy segítséget jelent kezdő programozók számára, hiszen nem kell megérteniük a függőségek működését és a konfigurálás menetét. Ezenkívül a haladó programozók dolgát is megkönnyíti, hiszen az automatizálásnak köszönhetően sokkal gyorsabban van lehetőségük az említett műveletek elvégzésére.

A műveletek végzésének helyét (lokális számítógép, image buildelése, konténer futtatása) úgy határoztam meg, hogy a lehető legjobban teljesüljön az újrafelhasználhatóság elve. Ezentúl a programozó számos beállítás közül választhat, és lehetősége van a megfelelő beállítások kiválasztásával egy tényleg hatékony, gyors és stabil tesztkörnyezet használatára.

Ahogy a tesztelés során bemutattam, a környezet stabilan működik mind kisebb projektek pár checkerrel való elemzése, mind az összes projekt összes checkerrel való elemzése során. A CodeChecker által generált HTML kimenet, valamint a környezet által adott futás közbeni visszajelzések pedig felhasználóbaráttá és egyszerűen kezelhetővé teszik a programot.

Irodalomjegyzék

[1] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, March 1997.

[2] Carnegie Mellon University Software Engineering Institute. SEI CERT Coding Standards. `https://llvm.org`. [Elérés dátuma: 2020.04.13].

[3] LLVM Developer Group. Extra Clang Tools 11 documentation - Clang-Tidy. `https://clang.llvm.org/extra/clang-tidy/`. [Elérés dátuma: 2020.04.23].

[4] B. Potter and G. McGraw. Software security testing. *IEEE Security Privacy*, 2(5):81–85, Sep. 2004.

[5] Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, SSE 2014, page 37–44, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Michael Ernst. Static and dynamic analysis: synergy and duality. `https://homes.cs.washington.edu/~mernst/pubs/staticdynamic-paste2004-slides-dist.pdf`. [Elérés dátuma: 2020.03.11].

[7] Tanmay Sinha Ali Almossawi, Kelvin Lim. Analysis Tool Evaluation: Coverity Prevent. `https://www.cs.cmu.edu/~aldrich/courses/654-sp07/tools/cure-coverity-06.pdf`. [Elérés dátuma: 2020.04.22].

[8] Inc. GrammaTech. CodeSonar - Static Analysis SAST Software for Secure SDLC. `https://www.grammatech.com/products/codesonar`. [Elérés dátuma: 2020.04.23].

[9] Facebook. Facebook Infer documentation. https://fbinfer.com/docs/getting-started. [Elérés dátuma: 2020.04.23].

[10] Cppcheck team. Cppcheck manual. http://cppcheck.sourceforge.net/manual.pdf. [Elérés dátuma: 2020.04.23].

[11] LLVM Developer Group. Clang Static Analyzer. https://clang-analyzer.llvm.org. [Elérés dátuma: 2020.04.23].

[12] The Clang Team. Clang: a C language family frontend for LLVM. https://clang.llvm.org/index.html. [Elérés dátuma: 2020.04.23].

[13] Carnegie Mellon University Software Engineering Institute. SEI CERT Coding Standards, ENV33-C. Do not call system(). https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177. [Elérés dátuma: 2020.05.02].

[14] Ericson. CodeChecker. https://codechecker.readthedocs.io/en/latest/. [Elérés dátuma: 2020.04.23].

[15] Bitcoin Core developers. Bitcoin Core. https://bitcoincore.org. [Elérés dátuma: 2020.04.29].

[16] FFmpeg developers. FFmpeg. https://ffmpeg.org. [Elérés dátuma: 2020.04.29].

[17] Stephen Turner. Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby. http://gauss.ececs.uc.edu/Courses/c6056/pdf/131731.pdf. [Elérés dátuma: 2020.04.11].

[18] Carnegie Mellon University Software Engineering Institute. SEI CERT Coding Standards, EXP34-C. Do not dereference null pointers. https://wiki.sei.cmu.edu/confluence/display/c/EXP34-C.+Do+not+dereference+null+pointers. [Elérés dátuma: 2020.04.27].

[19] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 21–30, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[20] University of Virginia Inexpensive Program Analysis Group. Splint - Secure Programming Lint. http://splint.org. [Elérés dátuma: 2020.05.03].

[21] Inc. Perforce Software. Klockwork - Static Code Analysis for C, C++, C#, and Java. https://www.perforce.com/products/klocwork. [Elérés dátuma: 2020.04.23].

[22] Carnegie Mellon University Software Engineering Institute. SEI CERT Coding Standards, POS44-C. Do not use signals to terminate threads. https://wiki.sei.cmu.edu/confluence/display/c/POS44-C.+Do+not+use+signals+to+terminate+threads. [Elérés dátuma: 2020.04.13].

[23] Carnegie Mellon University Software Engineering Institute. SEI CERT Coding Standards, CON54-CPP. Wrap functions that can spuriously wake up in a loop. https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON54-CPP.+Wrap+functions+that+can+spuriously+wake+up+in+a+loop. [Elérés dátuma: 2020.04.13].

[24] Carnegie Mellon University Software Engineering Institute. SEI CERT Coding Standards, CON36-C. Wrap functions that can spuriously wake up in a loop. https://wiki.sei.cmu.edu/confluence/display/c/CON36-C.+Wrap+functions+that+can+spuriously+wake+up+in+a+loop. [Elérés dátuma: 2020.04.13].

[25] Carnegie Mellon University Software Engineering Institute. SEI CERT Coding Standards, CON37-C. Do not call signal() in a multithreaded program. https://wiki.sei.cmu.edu/confluence/display/c/CON37-C.+Do+not+call+signal%28%29+in+a+multithreaded+program. [Elérés dátuma: 2020.04.13].

[26] Carnegie Mellon University Software Engineering Institute. SEI CERT Coding Standards, CON40-C. Do not refer to an atomic variable twice in an expression. https://wiki.sei.cmu.edu/confluence/display/c/CON40-C.+Do+not+refer+to+an+atomic+variable+twice+in+an+expression. [Elérés dátuma: 2020.04.13].

[27] LLVM Developer Group. The LLVM Compiler Infrastructure. https://wiki.sei.cmu.edu/confluence/display/seccode. [Elérés dátuma: 2020.04.13].

[28] Docker Inc. Docker. https://www.docker.com. [Elérés dátuma: 2020.04.29].

[29] LLVM Developer Group. Getting Started with the LLVM System. https://llvm.org/docs/GettingStarted.html. [Elérés dátuma: 2020.04.13].

[30] Docker Inc. Docker Documentation. https://docs.docker.com. [Elérés dátuma: 2020.04.24].

[31] Docker Inc. Docker - UCP System requirements. https://docs.docker.com/datacenter/ucp/1.1/installation/system-requirements/. [Elérés dátuma: 2020.04.24].

[32] CMake developers. CMake. https://cmake.org. [Elérés dátuma: 2020.04.29].

[33] Inc Free Software Foundation. GCC, the GNU Compiler Collection. https://gcc.gnu.org. [Elérés dátuma: 2020.04.29].

[34] Python Software Foundation. Python. https://www.python.org. [Elérés dátuma: 2020.04.29].

[35] Inc. Free Software Foundation. GNU Make. https://www.gnu.org/software/make/. [Elérés dátuma: 2020.04.29].

[36] Software Freedom Conservancy. Git. https://git-scm.com. [Elérés dátuma: 2020.04.29].

[37] Ninja build developers. Ninja. https://ninja-build.org. [Elérés dátuma: 2020.04.29].

[38] Inc. Software in the Public Interest. Debian - packages. https://www.debian.org/distrib/packages. [Elérés dátuma: 2020.04.29].

[39] Docker Inc. Install Docker Engine on Debian. https://docs.docker.com/engine/install/debian/#install-using-the-repository. [Elérés dátuma: 2020.04.24].

[40] LLVM Developer Group. Extra Clang Tools 11 documentation, Clang-Tidy - Clang-Tidy Checks. https://clang.llvm.org/extra/clang-tidy/checks/list.html. [Elérés dátuma: 2020.04.13].

[41] LLVM Developer Group. LLVM Phabricator - [clang-tidy] Adding misc-signal-terminated-thread check. https://reviews.llvm.org/D69181. [Elérés dátuma: 2020.04.29].

[42] LLVM Developer Group. LLVM Phabricator - [clang-tidy] Add do-not-refer-atomic-twice check. https://reviews.llvm.org/D77493. [Elérés dátuma: 2020.04.29].

[43] LLVM Developer Group. LLVM Phabricator - [clang-tidy] Add signal-in-multithreaded-program check. https://reviews.llvm.org/D75229. [Elérés dátuma: 2020.04.29].

[44] LLVM Developer Group. LLVM Phabricator - [clang-tidy] Add spuriously-wake-up-functions check. https://reviews.llvm.org/D70876. [Elérés dátuma: 2020.04.29].

[45] curl developers. curl. https://curl.haxx.se. [Elérés dátuma: 2020.04.29].

[46] Dormando. Memcached. https://memcached.org. [Elérés dátuma: 2020.04.29].

[47] Inc. F5. NGINX. https://www.nginx.com. [Elérés dátuma: 2020.04.29].

[48] The PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org. [Elérés dátuma: 2020.04.29].

[49] redislabs. Redis. https://redis.io. [Elérés dátuma: 2020.04.29].

[50] tmux developer group. tmux. https://github.com/tmux/tmux/wiki. [Elérés dátuma: 2020.04.29].

[51] OpenSSL Software Foundation. OpenSSL. https://www.openssl.org. [Elérés dátuma: 2020.04.29].

[52] vim developers. vim. https://www.vim.org. [Elérés dátuma: 2020.04.29].

[53] cpp-taskflow developers. cpp-taskflow. https://github.com/cpp-taskflow/cpp-taskflow. [Elérés dátuma: 2020.04.29].

[54] Doug Binks. enkiTS. https://github.com/dougbinks/enkiTS. [Elérés dátuma: 2020.04.29].

[55] RaftLib developers. RaftLib. http://www.raftlib.io. [Elérés dátuma: 2020.04.29].

[56] RichieSams. FiberTaskingLib. https://github.com/RichieSams/FiberTaskingLib. [Elérés dátuma: 2020.04.29].

[57] Osamu Aoki. Debian package management. https://www.debian.org/doc/manuals/debian-reference/ch02.en.html. [Elérés dátuma: 2020.04.29].

[58] Haivision. Secure ReliableTransport (SRT). https://github.com/Haivision/srt. [Elérés dátuma: 2020.04.29].

[59] The Clang Team. Extra Clang Tools 11 documentation - Getting Involved. https://clang.llvm.org/extra/clang-tidy/Contributing.html. [Elérés dátuma: 2020.04.23].

[60] Clang Developer Group. Clang - Expressive Diagnostics. https://clang.llvm.org/diagnostics.html. [Elérés dátuma: 2020.04.30].

[61] Clang Developer Group. Clang - AST MAtcher reference. https://clang.llvm.org/docs/LibASTMatchersReference.html. [Elérés dátuma: 2020.04.30].

[62] Clang Developer Group. Extra Clang Tools 10 documentation - CLANG-TIDY - BUGPRONE-BAD-SIGNAL-TO-KILL-THREAD. https://releases.llvm.org/10.0.0/tools/clang/tools/extra/docs/clang-tidy/checks/bugprone-bad-signal-to-kill-thread.html. [Elérés dátuma: 2020.04.30].

[63] Clang Developer Group. Extra Clang Tools 11 documentation - CLANG-TIDY - BUGPRONE-SPURIOUSLY-WAKE-UP-FUNCTIONS. https://clang.llvm.org/extra/clang-tidy/checks/bugprone-spuriously-wake-up-functions.html. [Elérés dátuma: 2020.05.05].

A. függelék

Melékletek

A dolgozat melléklete öt almappát tartalmaz.

A `checkers` mappa a Clang-Tidy statikus elemzőhöz elkészített checkerek forráskódjait, míg a `checker_tests` ezen checkerek tesztjeit, a `checker_docs` mappa pedig a dokumentációs fájljaikat tartalmazza. A checkerek ilyen formában nem használhatók, kizárólag az LLVM projekt részeként, ezért kipróbálásukhoz a dolgozatban említett beállítások elvégzése szükséges.

A melléklet tartalmaz ezentúl egy `clang-test-docker` mappát, mely a Clang-Tidy elemzőhöz készített tesztkörnyezet forráskódját tartalmazza. Ez a Docker megfelelő telepítése után használható.

A mellékletben található `atomic` almappa az atomikus változók bemutatásánál használt példafájl kódját tartalmazza.