TDK-dolgozat

Mosolygó Balázs József

Vándor Norbert Rudolf

Szegedi Tudományegyetem

Informatikai Intézet

Szoftverfejlesztés Tanszék

Biztonsági sérülékenységek javításának empirikus vizsgálata nyílt forrású rendszerekben

Készítette:

Mosolygó Balázs József

III. évf. programtervező informatikus BSc

Vándor Norbert Rudolf

III. évf. programtervező informatikus BSc

Témavezetők:

Dr. Hegedűs Péter, tudományos munkatárs

Antal Gábor, tudományos segédmunkatárs

SZTE TTIK Szoftverfejlesztés Tanszék

Szeged

2020

Tartalomjegyzék

Tartalomjegyzék 1

1. Bevezetés **2**

1.1. Dolgozat felépítése 3

2. Motiváció **4**

3. Kapcsolódó munkák **6**

3.1. Adatbányászat 9

4. Megközelítés **11**

4.1. CVE Manager 11

4.2. Git Log Parser 12

4.3. CVE Miner 12

5. Eredmények **15**

5.1. CVE előfordulások 15

5.2. Eltelt idővel kapcsolatos statisztikák 16

5.3. Aktivitás alapú statisztikák 19

5.4. Átlagos fájl és sor változtatások 21

5.5. CWE-kkel kapcsolatos statisztikák 23

5.6. CVE anomáliák, és okaik 25

6. Konklúzió **29**

A. Említett CWE-k definíciói **35**

1. fejezet

Bevezetés

A szoftverfejlesztés egy rohamosan terjeszkedő iparág, a fejlődése és belső működése gyökeresen különbözik a legtöbb társától, hiszen a létrehozott termék szellemi értéket képvisel elsősorban, így nehéz objektív módon értékelni, összehasonlítani fejlesztési folyamatok, technikák hatékonyságát. Minden fejlesztőcsapat törekszik a lehető legnagyobb minőség, és biztonság elérésére, ennek ellenére azonban, gyakran előfordul, hogy kisebb-nagyobb hibák keletkeznek. Ezek észlelése és javítása azonban, csak az első lépések a hiba eltüntetésében, ugyanis a felhasználókat, legyenek azok más fejlesztők, vagy végfelhasználók, értésíteni kell arról, hogy szükséges lehet az újabb verzióra frissíteniük. A különböző projektekben alkalmazott rutinok eltérése miatt az ilyen tájkoztatás egy rendkívül komoly kihívás. A célból, hogy az információ megosztása könnyebb legyen a Mitre Vállalat 1999-ben létrehozta a CVE (Common Vulnerabilities and Exposures)[1] koncepciót, aminek lényege egy publikus adatbázis készítése volt, ahol egységes nevezési konvenciók segítségével könnyítették a sérülékenységekkel, és azok javításával kapcsolatos adatok terjedését. Egy CVE egy konkrét projekt sérülékenységét írja le, és rendel hozzá egy egyedi azonosítót, még egy CWE (Common Weakness Enumeration)[2] sérülékenységek egy csoportját jelöli. Az adatbázis habár rengeteg hasznos információt tartalmaz, mint például bizonyos hibák leírása, súlyossága és rövid leírása,nem ad minden kérdésre választ.

A különböző sérülékenységek kezelésében azonban továbbra is jelentős különbségek

[1] https://cve.mitre.org
[2] https://cwe.mitre.org

vannak, nyelveket vagy akár csak projekteket tekintve. Annak érdekében, hogy ezeket a különbségeket megvizsgálhassuk, létrehoztunk egy adatbázist, ami tartalmazza néhány nagyobb nyílt forráskódú projekt CVE-inek jelentős részét, hogy ennek segítségével alaposabb belátást nyerhessünk különböző nyelvek, illetve projektek hibákhoz, sérülékenységekhez való hozzáállásába, javítási folyamataiba. A rendelkezésre álló adatok segítségével lehetőség nyílik a hibák mélyebb elemzésére, a hozzájuk kapcsolódó statisztikák létrehozására, valamint további adatok kinyerésére, mint például a konkrét javítást végző kódrészletek, amik akár automatikus hiba előrejelző rendszerekben is felhasználhatóak. Érdemes megjegyezni, hogy egy projekt sebezhetősége nem feltétlen csak az adott alkalmazást érintheti, hanem több, azt használó termékre is kiterjedhet. A függőségek sérülékenységeinek kezelése további kérdés, hiszen amíg egy használt eszközben sérülékenységek vannak, addig az azt használóban is megjelennek.

1.1. Dolgozat felépítése

A következő fejezetben ismertetjük, hogy mi motivált minket a projekt elkezdésében, illetve milyen támpontokat tartottunk szem előtt a projekt készítése során. A 3. fejezetben említést teszünk néhány kapcsolódó munkáról, amiket feldolgoztunk.

A 4. fejezetben részletezzük, hogy hogyan készítettük elő a szükséges eszközöket és adatokat. Az 5-ben pedig ismertetjük az ezek alapján előállított statisztikákat, és következtetéseket. Végül pedig a 6. fejezetben összegezzük az elért eredményeket, és jövőbeli terveket.

2. fejezet

Motiváció

Egy CVE bejegyzés tartalmazza, a sérülékenység rövid leírárást, legalább 1 nyilvánosan elérhető referenciáját, súlyosságát, CWE besorolását, illetve az azonosítóját. A CVE-k rengeteg hasznos információt tesznek elérhetővé, és CWE-kbe csoportosítva általánosabb információ is elérhető velük kapcsolatban. Egy egyszerű keresés segítségével nem tudunk általános adatokhoz hozzájutni egy projekt vagy akár egy nyelv CVE-kkel kapcsolatos javítási tendeciáihoz, azonban erre komoly szükség lehet, ha jobban meg szeretnék érteni a komoly sérülékenységek kiváltó okait, és kiküszöbölni azokat. A konkrétabb megoldások ismeretében könnyebben javíthatóvá, elkerülhetővé, illetve detektálhatóvá tehetjük a CVE-ket, és általuk a bizonyos típusú sérülékenységeket is.

Az általunk létrehozott adatbázis lehetőséget nyújt arra, hogy megvizsgálhassuk az egyes hibákra biztosított megoldásokat, illetve az ezekkel kapcsolatos statisztikák is rendelkezésre állnak. Ezek segítségével gyorsan és megalapozottan tudunk értékes következtetéseket levonni, nem csak egy bizonyos sérülékenység kiküszöböléséről, hanem akár egy sérülékenységi csoport (CWE) általános nyelvi tulajdonságait is megtekinthetjük.

Megtudhatjuk, hogy az adott nyelv nyílt forráskódú projektjeiben mely hiba csoportok fordulnak elő, milyen gyakorisággal, és mekkora lehet a javítási költségük. Nem csak időt és energiát spórolhat meg a fejlesztőknek és tesztelőknek, hanem a teljes folyamatot költség- és időhatékonyabbá teheti. Továbbá felgyorsíthatja a termékeket függőségként használó projektek reakció idejét, ami csökkenti a visszaélés esélyét. A sérülékenységek javításához való hozzáálláson is változtathat, ha tudjuk, hogy az általunk használt termék fejlesztői gyorsan és hatékonyan javítanak hibákat, ebben az esetben ugyanis nem

feltétlen van szükség saját megoldást keresni, elegendő lehet várni a frissítésre, míg ellenkező esetben akár a legmagasabb prioritást is képezheti egy-egy felfedezett hiba helyi megoldása.

Annak érdekében, hogy ezeket az információkat cégek, és fejlesztők számára elérhetővé tegyük létrehoztunk egy adatbázist, amiben különböző nyelvek CVE javításaival kapcsolatos adatokat tárolunk.

A CVE adatain kívül az adatbázis nyelvekre és projektekre lebontható információt tartalmaz arról, hogy egy-egy hibát mennyi idő, hány kódsor- és fájlváltoztatás kiküszöbölni, illetve a konkrét javítást tartalmazó commit azonosítóját. Egy commit, pedig nem más, mint egy bizonyos változtatás egy kódbázison, amit ennek egyedi hash azonosítójával bármikor megtalálhatunk, és elemezhetünk.

3. fejezet

Kapcsolódó munkák

Amióta léteznek szoftverek, előfordulnak sérülékenységek is. A különböző hibák felismerése, és megelőzése hosszú és változatos múltra tekint vissza, amiben új fejezetet nyitott a Mitre vállalat, amikor 1999-ben bevezették a CVE fogalmát, aminek segítségével a különböző hibák könnyen kategorizálhatóva váltak ezáltal csökkentve a további projektekra való tovább terjedésük valószínűségét.

A CVE-k bevezetése habár megkönnyítette a biztonsági sérülékenységekkel kapcsolatos kommunikációt a különböző projektek köztött, ez továbbra is hatalmas kihívást jelent. Jukka Ruohonen és társai [14][15] megvizsgálták a a projektek közötti koordináció folyamatát nyílt levelezési listák segítségével, és azt találták, hogy viszonylag gyakran csúszások fordulnak elő, amik hatására a CVE-k később jelennek meg a hivatalos adatbázisokban. Különösen nagy eséllyel fordulnak elő nagyobb csúszások hétvégén. Az ilyen jelentés beli késések komoly károkat is okozhatnak, Clemens Sauerwein és társai [18] egy alternatív módját vizsgálták meg a hibákról való információ gyűjtésnek. A Twitter szociális média platformon elérhető fejlesztők aktivitását vizsgálva azt találták, hogy a biztonsági sérülékenységek negyedéről volt a hivatalos publikáció előtt szó a oldalon.

Annak ellenére, hogy nagy mennyiségű CVE bejegyzés áll rendelkezésre, ezek közül nem mindegyik biztosít kellő részletességű információt. Ennek a problémának orvosolását tűzték ki célul Sarang Na és társai [12]. A naív Bayes algoritmus felhasználásával állítottak elő CWE csoportosításokat már létező CVE-khez, annak érdekében, hogy a hiányosságokat pótolják. 3 évvel később Ying Dong és csapata készített által egy kutatást [3] aminek célja az volt, hogy felfedezzék mennyi inkonzisztens információ található

az NVD[1] adatbázisa, és a különböző CVE-k leírása között. Több mint 78000 CVE megvizsgálása után arra az eredményre jutottak, hogy nem csak gyakoriak az ily módon hibás adatok, de az is előfordulhat, hogy nem jelölnek meg komoly hibákat, vagy nem hibás verziókat jelölnek meg sérülékenyként. További problémára lett figyelmes Luis Gustavo Araujo Rodriguez és csapata [13], amikor arra próbáltak fényt deríteni, hogy mennyi az átlagos késés egy hiba javítása és az NVD adatbázisába való bekerülése között. Eredményeik szerint 1-7 nap késés is előfordulhat, illetve az is megtörténhet, hogy nem hivatalos oldalakon hamarabb és pontosabban elérhetővé válik a sérülékenységgel kapcsolatos információ.

Fabio Massacci és Viet Hung Nguyen [9] több sérülékenységgel és biztonsággal foglalkozó problémát vizsgált. Összefoglalták a különböző kutatások során használt adatbázisok jellemzőit, és összehasonlították ezeket. Ezen felül különböző kísérleteket végeztek Mozilla Firefoxon, hogy megmutassák, hogy különböző adatbázisok használata különböző eredményeket hozhat. Ezzel azt is hangsúlyozzák, hogy az adathalmaz megválasztása egy kulcskérdés az ilyen területű kutatásokban.

Annak érdekében, hogy pontosabb információ legyen elérhető gyorsabban, Luis Alberto Benthin Sanguino és Rafael Uetz [17], készítettek egy eszközt aminek segítségével lehetséges könnyedén egy alkalmazáshoz CPE-t, azaz egy egységes nevezéktan szerint megalkotott nevet rendelni, hogy könnyebb legyen automatikus hiba keresést végezni a szoftveren. Eredeti terveik szerint automatikusan szerették volna létrehozni a hozzárendeléseket, azonban számos nehézség merült fel ezzel kapcsolatban, és végül nem találták praktikusnak. Zhengzi Xu és társai [22] szintén a CVE adatok automatikus kiegészítését tűzték ki célul, ők azonban a bináris szinten érzékelik automatikusan a különböző potenciális hibákat.

A CVE-k súlyossági értékét figyelembe véve lehetséges prioritást rendelni különböző hibák javításához, ehhez készített egy algoritmust Anshu Tripathi és Umesh Kumar Singh [20].

Andrew Kronser [5] tézisében arról beszél, hogy bizonyos típusú szoftverek gyakrabban szenvednek biztonsági sérülékenységektől mint mások. Ennek oka nem csak a nagyobb potenciális támadó bázis, hanem a használt eszközök széleskörűsége, ami miatt

[1] https://nvd.nist.gov/vuln

lehetséges, hogy megnő a támadási felület. A legsérülékenyebb eszközök változnak attól függően, hogy mi az aktuális leggyakoribb sérülékenység, például 2008 és 2010 között az SQL injekció volt a domináns hiba. A CVE adatbázis növekedésének köszönhetően egyre hatékonyabb adatbányászó eszközök állíthatók elő, melyek segítségével a különböző támadásokat lehetővé tevő hibák könnyebben észlelhetőek lesznek.

A hibák előfordulási trendjeit V. Mounika és társai [10] megvizsgálták, és arra jutottak, hogy a közismert biztonsági sérülékenységek előfordulási aránya idővel változik. Thanapon Bhuddtham és Pirawat Watanapongse [2] a biztonsági sérülékenységek trendjeit kihasználva T-CVE-ket készítettek, amik korai figyelmeztetésekkel látják el a biztonsági szakértőket. Hasonlóan Richard Kuhn és társai [6] is megfigyeltek bizonyos trendeket a sérülékenységekkel kapcsolatban. Ők arról írnak, hogy bár a nagyon súlyos hibák aránya csökkenő tendenciát mutat, az implementációval kapcsolatosaké viszont nő. Ebbe a csoportba tartoznak olyan programozói hibák, mint a bemeneti adatok ellenőrzésének hiánya.

Stefan Frei és társai [4] pedig egy kiterjedt vizsgálat során megállapították azokat az időpontokat, amikor az egyes sérülékenységeket felfedezték, publikálták, kihasználták, illetve kijavították. Ezen kívül létrehoztak egy eszközt, aminek segítségével mérhetővé tehető a kihasználtság és a javítás elérhetősége közötti rés. Ezzel azt is megállapították, hogy 2006 előtt a támadók sokkal gyorsabban reagáltak az egyes sérülékenységekre, mint a gyártók, manapság viszont ez általánosságban fordítva igaz.

Ezt a megállapítást támasztja alá Muhammad Shahzad és társai [19] kutatása. Ők szintén egy nagyméretű adathalmazon dolgoztak, hogy megvizsgálják a különböző sérülékenységeket és kategorizálják őket. Megerősítették, hogy a gyártók egyre gyorsabban reagálnak a sérülékenységekre; ezen belül a zárt forráskódú projektek sokkal jobban teljesítenek ilyen téren, mint a nyílt forráskódúak.

Frei és társaihoz hasonlóan Frank Li és Vern Paxson [7] is nagyméretű adathalmazt vizsgált. Ők, mint a mi kutatásunk is, CVE adatbázist használtak a vizsgálathoz, viszont ők igénybe vettek egyéb külső forrásokat is. Ezeket felhasználva megfigyelték a hibajavítások életciklusát, ezek hatását a kódbázisra, illetve hogy mekkora mértékben tudnak a fejlesztők biztonságos és megbízható javításokat végezni. Azt találták, hogy a sérülékenységek javításai jóval kisebb nyomot hagynak a kódbázison, mint az egyéb, nem

biztonsági hibáké. Ezen felül az összes sérülékenység nagyjából harmada több, mint 3 évvel a végleges kijavításuk előtt már ismert volt. Olyan esetek is előfordultak, hogy a hibajavítás nem oldotta meg teljesen a problémát, helyette akár negatív hatást is kifejtett az adott szoftverre.

3.1. Adatbányászat

Annak érdekében, hogy a különböző sérülékenységekkel kapcsolatos adathalmazt bővítsék többen fordulnak adatbányászathoz. A bányászás több forrás alapján is történhet, például Leon Wu és csapata [21] különböző problémakövető adatbázisból (Bugzilla, Debbugs stb.) gyűjtöttek ki hibajavítási információt, aminek segítségével létrehozták a BugMiner eszközt, ami képes ellenőrizni a hibajelentések redundanciáját. Ők elsősorban a jelentések pontosságára koncentráltak mi azonban inkább a fejlesztési és javítási folyamat elemzésére fektettük a hangsúlyt.

Az erős adatbányászati algoritmusok segítségével kiküszöbölhetők a CVE-k korábban tárgyalt hiányosságai, vagyis az adatok, mint például CWE besorolás néha fellépő hiánya. Su Zhang és társai [24] készítettek egy tanulmányt, aminek célja az volt, hogy megvizsgálják, hogy mekkora a valószínűsége annak, hogy egy szoftver tartalmaz egy még fel nem fedezett sérülékenységet. Az ehhez használt kódelemzés mélyrehatóan vizsgálja a forráskódokat, ami sok potenciális hibát képes megtalálni.

Sagar Samtani és társai hozzánk hasonlóan szövegbányászással közelítettek meg egy internet biztonsági problémát [16]. Ők a Shodan [2] kereső motor segítségével elérhető információt dolgozták fel, annak érdekében, hogy SCADA rendszereket[3], és ezek hibáit azonosítsák. Mi ezzel szemben nyílt forráskódú szoftvereket vizsgáltunk meg, forrásként pedig a GitHub [4] szoftver fejlesztési platformot használtuk.

Daisuke Miyamoto és társai [23] létrehoztak egy eszközt, ami képes emberi nyelven megfogalmazott leírások alapján megállapítani egy-egy hiba súlyosságát. A sérülékenységek ilyen irányú összehasonlítása sokat segíthet abban, hogy a kiberbiztonsági szakértők hatékonyabban legyenek képesek reagálni rájuk. Viszont nem kellően pontos ahhoz, hogy

[2] https://www.shodan.io/
[3] Felügyeleti Irányítást és Adatgyűjtést végző rendszer
[4] https://github.com/

teljesen elhanyagolható legyen az emberi felülvizsgálat.

Az adatbányászat által szerzett információ pontosítása értekében, Xiang Li és csapata [8] készítettek egy algoritmust, ami állításuk szerint jelentősen hatékonyabb mint a korábbiak. Syed Shariyar Murtaza és társai [11] szintén hatékonyabbá akarták tenni a sérülékenységek észlelését, ennek érdekében különböző mintázatokat kerestek a korábbi hiba előfordulásokban. Eredményeik szerint egy adott sérülékenység akár 150-szer is előfordulhat egy szoftverben. Az előforduló sérülékenység típusa pedig, előrejelezhető trendeket követ. Például egyre kevesebb az SQL injekcióval kapcsolatos sérülékenység, azonban egyre több a titkosítási hiba. A hibák további vizsgálatához szükséges lehet egy a mienkhez hasonló adatbázis, hiszen a különböző javítások, és javított hibák adatainak hosszú távú tárolása kulcsfontosságú lehet a további hiba trendek felfedezésében.

4. fejezet

Megközelítés

Az adatbázisunk létrehozásához nyílt forráskódú projekteket elemeztünk. A projekteket a GHTorrent[1] adatbázisa segítségével gyűjtöttük össze. A GHTorrent célja, hogy a GitHub Rest API-on keresztül elérhető adatokat, azaz az összes a felhasználók által alapvetően megtekinthető adatot, egy skálázható, offline adatbázisba gyűjtsék. Az adatbázis 2020 januárjában 18 TB adatot tárolt, mi ehhez a Google által készített BigQuery[2] eszköz segítségével fértünk hozzá.

A CVE információk kinyeréséhez létrehoztunk egy programot, ami átvizsgálja a projektek GitHub oldalán levő commitokat, és megnézi, hogy említenek-e egy adott CVE-t, majd ezeket az említő commitokat és a hozzájuk tartozó adatokat elmenti az adatbázisunkba. A módszerünk legfőbb előnye, hogy nincs szükség mélyreható kódelemzésre.

A módszer részletes leírásához bemutatjuk az általunk létrehozott programot. Maga a program három részből áll: a CVE Manager nevű modulból, a Git Log Parser nevű modulból, és a statisztikák feldolgozásáért felelő fő eszközből. Az elkülönítésük azért történt, mert a modulok önmagukban is használhatóak, nem közvetlen részei a fő eszköznek.

4.1. CVE Manager

A CVE Manager feladata az, hogy a Mitre Vállalat oldaláról letöltse az összes CVE adatait és ezeket kezelje. A letöltés után az adatokat feldolgozza, csoportosítja, és egy helyi

[1] https://ghtorrent.org/
[2] https://cloud.google.com/bigquery/

PostgreSQL[3] adatbázisban tárolja. A kezeléshez hozzátartozik az adatok lekérdezése is. Ezt többféle tulajdonság alapján is meg lehet tenni, például CVE-knél azonosító, súlyosság, publikáció éve alapján, CWE-k esetében pedig azonosító vagy hozzá tartozó CVE alapján. A legfőbb feladata, a megfelelő előkészületek elvégzése, a CVE Miner helyes futásának érdekében.

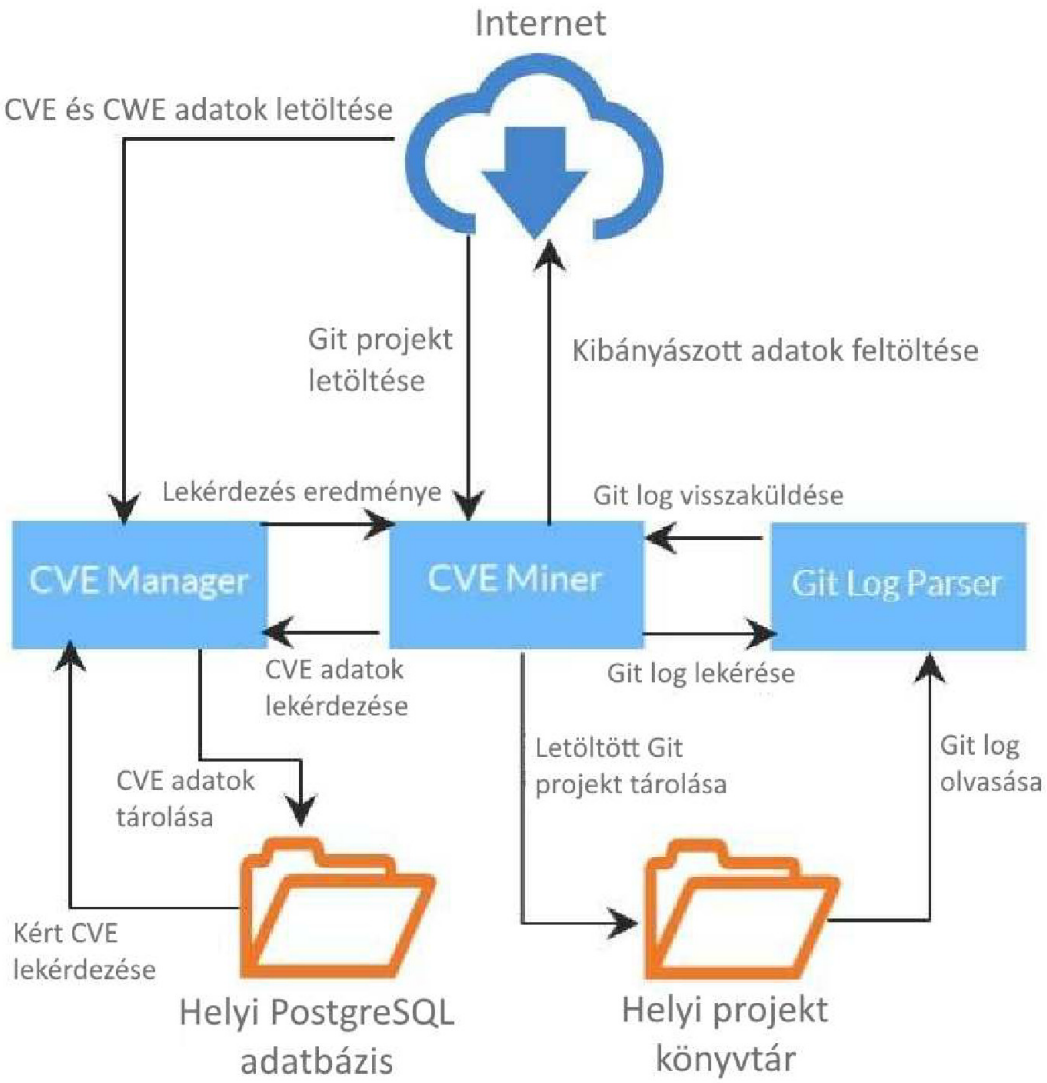
4.2. Git Log Parser

A Git Log Parser feladata a projektek commitjainak a feldolgozása. Ehhez egy Python modult használ, amivel szimulálni tudja a felhasználói parancsokat, ezzel részlegesen megkerülve a GitHub API[4] limitációit. Először is egy elérési útvonalat vár, amely lehet egyetlen projekt helyi címe, vagy egy könyvtár ami több projektet is tartalmaz. Minden esetben szükséges, hogy a projektek `git clone` segítségével legyenek letöltve, hogy a később kiadott parancsok működőképesek legyenek. A commit adatgyűjtési folyamat megkezdésekor a megadott, vagy a könyvtárban elsőként szereplő projekt mappájába navigál, majd a `git log` parancs eredményeit feldolgozva kinyeri a projekt összes commitját és azok meta-adatait. Ezeket egy listába rendezi, majd kiegészíti a megváltozott kódsorok és fájlok számával. A módosítási adatok kinyeréséhez szükséges kapcsolatba lépni a GitHub API-val ami jelentősen limitálja az eszközt. Végül az összes adatot beleírja egy ideiglenes JSON fájlba, amit a fő program kezel.

4.3. CVE Miner

A projektünk fő eszköze a CVE Miner, ami felhasználja az előző két alprogramot, illetve további funkciókkal egészíti ki őket. Az alkalmazás működését a a 4.1 ábra illusztrálja.

[3] https://www.postgresql.org/about/
[4] https://developer.github.com/v3/



4.1. ábra. Az alkalmazás működése

A program működéséhez néhány előzetes beállítás szükséges. Itt meg kell adnunk az utat, ahova a CVE Manager letölti a szükséges adatokat, illetve a helyi adatbázishoz tartozó hozzáférési adatokat. A CVE Manager ezek után előkészíti a helyi adatbázist a későbbi használatra.

A program működése során a parancssorban meg kell adnunk a feldolgozni kívánt projekt helyi vagy online elérési útját, de akár egy JSON fájlt is megadhatunk, ami több projektet tartalmaz. Ha a projekt nem lokálisan elérhető, akkor a CVE Miner letölti. Ezek után a CVE Miner elindítja a Git Log Parsert, ami legenerál egy ideiglenes JSON fájlt. Ezt a program beolvassa, és átnézi a commitokhoz tartozó üzenetet, azt figyelve, hogy van-e CVE említés az üzenetekben. Az alkalmazás egy-egy CVE első és utolsó említését tárolja

közvetlenül, hiszen ezek tartalmazzák a legtöbb információt, a köztes említések adatait utólag számítjuk ki ezek alapján. A program a CVE Manageren keresztül lekérdezi a talált CVE-hez tartozó fontosabb adatokat. Ilyenek például a publikálás dátuma, a hiba súlyossága, illetve az, hogy melyik CWE csoportba tartozik.

A következő lépés a statisztikák készítése. Minden feldolgozott projektről készül egy JSON fájl, ami többféle statisztikát tartalmaz. Az egyik csoport a CVE-kre lebontott statisztikák. Ide tartoznak a korábban kibányászott adatok mint a súlyosság, illetve az első és utolsó említés között eltelt idő és a commitok száma. A másik csoport a projektre vonatkozó statisztikák, amik létrehozásához a NumPy[5] Python csomagot használtuk. Ide tartoznak különböző korrelációk (idő és súlyosság, megváltozott kódsorok száma és súlyosság, illetve megváltozott fájlok száma és súlyosság közötti korrelációk) és átlagok (első és utolsó említések között eltelt átlagos idő, megváltozott kódsorok és fájlok átlagos száma). Ezek után, amennyiben a megfelelő kapcsolót megadtuk, az adatok felkerülnek egy Airtable [6] adatbázisba. Ha nem adjuk meg, akkor csak helyi JSON fájlok készülnek róluk. A harmadik csoportba tartozó statisztikák az adatbázisbeli adatokból jönnek létre. Ezek az egyes nyelvekre lebontott statisztikák. Itt ugyanazokat a korrelációkat és átlagokat számolja a program mint a projektek esetében, de kiegészíti őket szórással és mediánnal is. Ha esetleg valamiért az Airtable nem megfelelő, akkor egy másik kapcsoló megadásával lehetőség van létrehozni egy helyi PostgreSQL adatbázist, ahova a program minden adatot feltölt.

[5]https://numpy.org/
[6]https://airtable.com/product

5. fejezet

Eredmények

Ebben a fejezetben ismertetjük a kutatásunk eredményeit. Az alábbiakban közöltek nem jellemzik a nyelveket általánosságban, hiszen ahhoz túl kicsit a vizsgált projektek száma, hanem inkább a népszerűbb GitHub projektekről vonhatunk le általánosabb következtetéseket.

5.1. CVE előfordulások

Annak érdekében, hogy a későbbi statisztikákat a megfelelő kontextusba tudjuk helyezni elengedhetetlen, hogy tudjuk, pontosan hány projektről beszélünk nyelvenként. Minden nyelvhez 500 projektet néztünk át mint ahogy ezt a 5.1 láthatjuk is.

Nyelv	**Átnézett projektek száma**	**CVE említést tartalmazó projektek száma**	**CVE említést tartalmazó projektek aránya**	**CVE említések száma**
BitBake	500	29	5.8%	1028
C	500	51	10.2%	1476
C++	500	54	10.8%	709
Go	500	50	10%	245
Java	500	44	8.8%	210
JavaScript	500	21	4.2%	115
PHP	500	61	12.2%	133
Python	500	92	18.4%	317
Ruby	500	86	17.2%	1505
Scheme	500	8	1.6%	2180

5.1. táblázat. A CVE előfordulásokkal kapcsolatos adatok

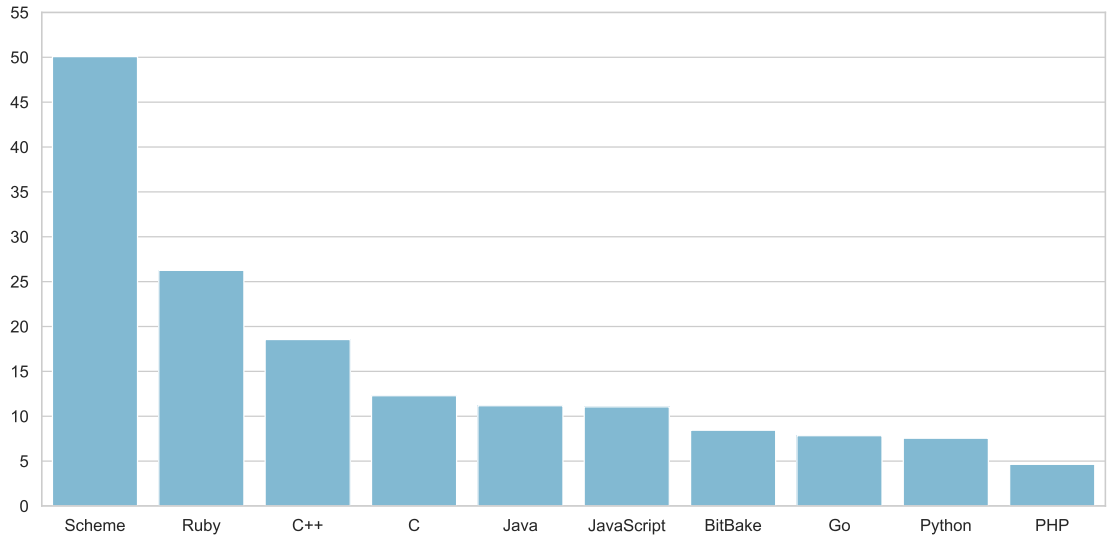
Jól látható, hogy komoly eltérések vannak a projektenkénti CVE-k számának terén, Scheme-ben például mindössze 8 projektben találtunk említést, de így is több darabot, mint minden más nyelv esetében. Ennek oka, hogy itt elsősorban csomag kezelő alkalmazásokat ellenőriztünk, amelyeknek rendkívül sok külső sérülékenységre kell reagálniuk. A legnagyobb CVE előfordulási aránnyal a Python rendelkezik, itt a projektek 18 százalékában találtunk legalább 1 említést. Ez mögött számos ok állhat, ezek közül a legvalószínűbb, hogy a Python egy fiatalabb, feltörekvőben lévő nyelv, ebből kifolyólag a fejlesztési folyamatnak szervesebb részévé válhatnak a modern megközelítések, mint például a CVE-k kezelése, és rögzítése.

5.2. Eltelt idővel kapcsolatos statisztikák

Átlagos idő egy hiba első és utolsó említése között Ez a statisztika több dolgot is megmutathat egy adott programozási nyelven fejlesztő közösség tendenciáival kapcsolatban. A legkézenfekvőbb jelentésük, hogy mennyi idő egy hibát átlagosan kijavítani. Ez nem pontos statisztika, hiszen, arra nincs mód, hogy megtudjuk mikor vettek észre egy adott hibát és kezdtek el dolgozni a javításán, viszont az arányok mutatására így is alkalmas. Előfordulhat, hogy egy CVE többször is meg van említve, ennek leggyakoribb oka, hogy a javító kód később lett a központi fejlesztési ágra bevéve. Ezt az esetet úgy vettük mintha a hiba a merge pillanatáig nem lenne javítva, hiszen hiába áll rendelkezésre egy potenciális megoldás, ha az éles verziónak nem része.

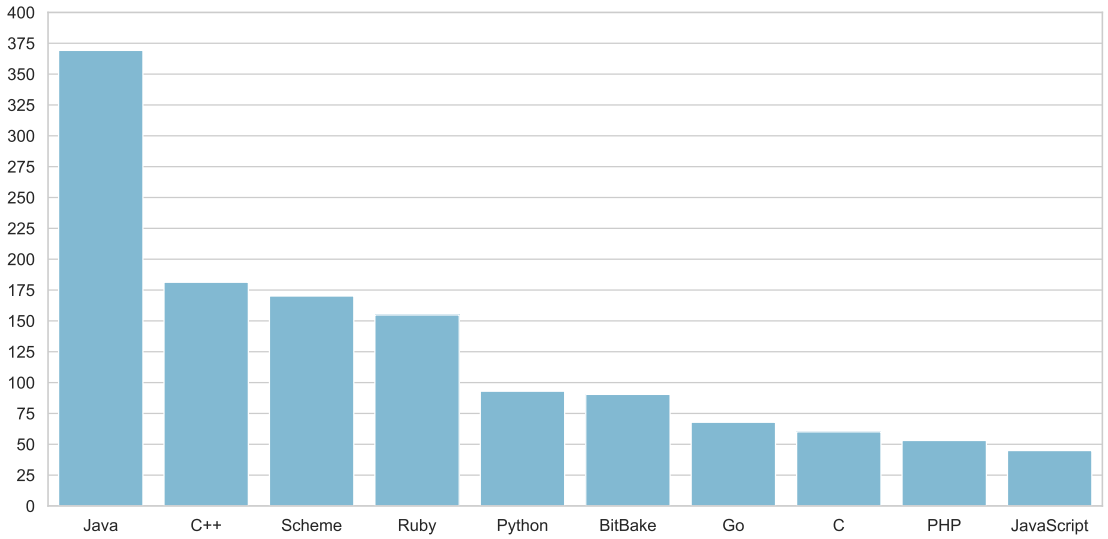
Egy másik leolvasható érték, hogy milyen gyakorisággal van szükség később visszatérni egy-egy hiba javításához. A merge-ek általában gyorsan történnek, 1-2 hétnél nem tartanak tovább, hiszen ekkorra már a hiba közismert, és kihasználható. Előfordulhat, hogy akár hónapokkal később a fejlesztők felfedeznek egy jobb megoldást a problémára, és lecserélik a korábbi implementációt. Ritka esetekben az is megtörténhet, hogy a probléma kiküszöbölése során további hibákat hoznak létre, amik megoldásához újra foglalkozni kell a korábban írt kóddal.

Az ezzel kapcsolatos eredményeink az 5.1. ábrán láthatóak.



5.1. ábra. Az átlagos eltelt idő egy CVE első és utolsó említése között napokban

Átlagos eltelt idő egy CVE publikációja és javítása között A legtöbb CVE javítás átterjedő hibákból ered, azaz egy kisebb projekt egy nagyobbat használ függőségként, vagy fordítva, ekkor a függőség hibája javításra kerül a használóban is, legtöbb esetben egy egyszerű verziófrissítés keretein belül. Ezekben az esetekben a javítás csak a CVE publikációja után lehetséges. A forrás projektekben nagyon ritkán fordul elő, hogy a hivatalos publikáció után említenék a hibát, hiszen senki nem publikálna egy komoly sérülékenységet, amit nem tudnak javítani. A függő projektekben természetesen szinte minden CVE javítás a publikáció után történik. Előfordul azonban, hogy a tudatos verziófrissítés korábban történik meg, ami annak a következménye, hogy a nagyobb nyitott forrású projekteknek léteznek saját önálló módjai arra, hogy a komoly sérülékenységekről és azoknak javításáról értesítse a felhasználóikat.



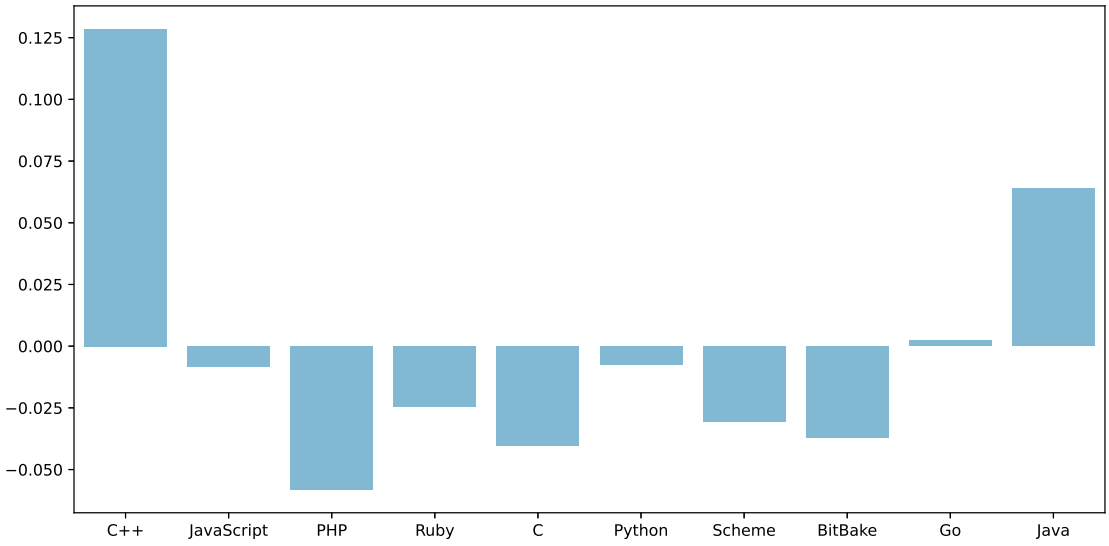
5.2. ábra. Az átlagos eltelt idő egy CVE publikációja és utolsó említése között napokban

Az 5.2. ábrán megtekinthető a publikációtól eltelt átlagos idő napokban nyelvekre bontva. Az itt látható értékek lényegesen nagyobbak mint az 5.1. ábrás láthatóak, ennek elsődleges oka, hogy egy CVE gyanús sérülékenységet csak a publikáció után neveznek hivatalosan annak. A második legvalószínűbb indok, a fent említett probléma, miszerint a függő programok csak a publikáció után tudják javítani az így felmerülő hibákat.

Korreláció az eltelt idő egy a probléma súlyossága között Javításhoz szükséges idő, és a hibák súlyossága közötti korreláció azt mutatja, hogy a nagyobb hiba súlyosság[1] hosszasabb fejlesztési időt eredményez-e.

A pozitív összefüggés azt mutatja, hogy a súlyosabb hibákat több idő javítani, a negatív pedig ennek az ellentétét, tehát, hogy súlyosabb problémákat gyorsabban küszöbölik ki.

[1]CVSS severity score, https://www.first.org/cvss/v3.1/specification-document



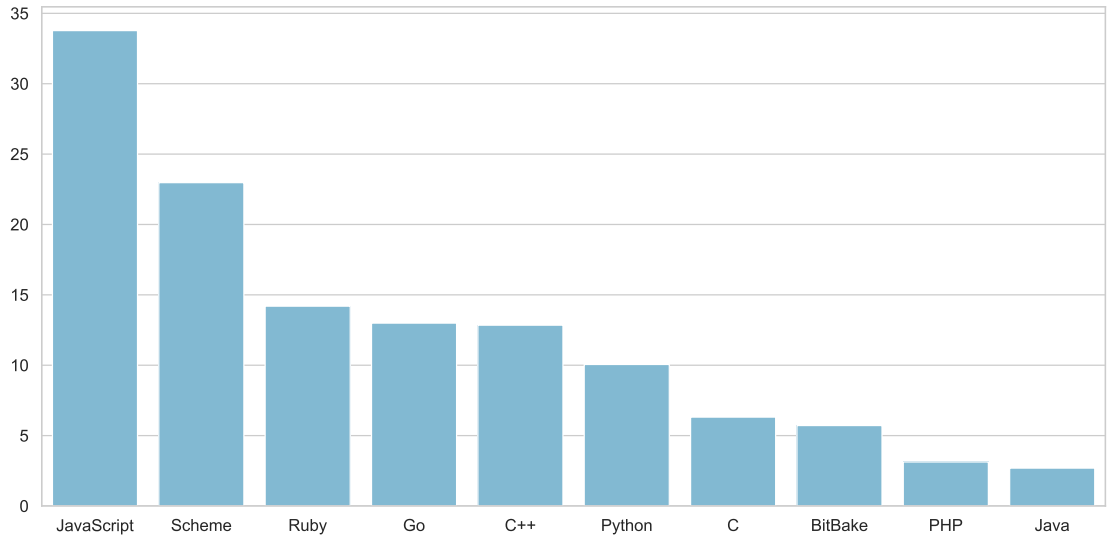
5.3. ábra. Korreláció az eltelt idő egy a probléma súlyossága között

Az 5.3. ábrán látható, hogy a legtöbb esetben teljesen elhanyagolható a korreláció, tehát a hiba komolysága nem áll összefüggésben a javításhoz szükséges idővel. Mivel a szoftver hibákról általánosan tudjuk, hogy legtöbbször kisebb kódrészletekre korlátozódnak, így ez a várható eredmény, hiszen a debugolási folyamat legidőigényesebb része a hiba helyének meghatározása.

A C++ az egyetlen nyelv, amelyben a mostani adathalmazunk alapján található összefüggés a két érték között. Az érték még itt is csak egy gyenge kapcsolatot jelez, de ez azonban már alap lehet egy mélyebbre ható kutatásra.

5.3. Aktivitás alapú statisztikák

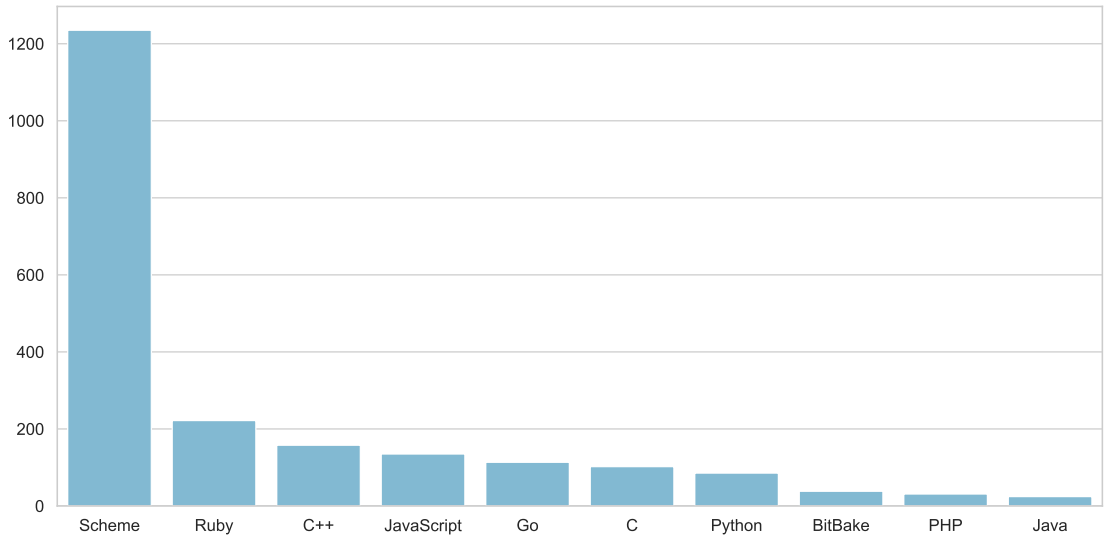
Aktív fejlesztők miközben egy CVE javítás folyamatban van A fejlesztők száma egy hasznos kiegészítő statisztika. Jellemzi, az átlagos aktivitást a vizsgált nyelvekben, ami segíthet jobban kontextusba helyezni a korábbi statisztikákat. Például, ha egy CVE jelenléti idejében kevesen aktívak, és a javítás sokáig tart, egy kevésbé menedzselt projektről beszélünk, azonban ha kevés fejlesztő gyorsan javít átlagosan, az egy gyors és hatékony közösségről árulkodik.



5.4. ábra. Az aktív fejlesztők átlagos száma

Az 5.4. ábra további kontextust biztosít a korábbi statisztikákhoz, illetve indoklásként szolgál az eddigi eredményekhez. Az általunk vizsgált adatok alapján például Java-ban átlagosan majdnem kétszer annyi ideig tartott egy hiba javítása a publikációjától számítva (lásd 5.2), mint C++-ban, de átlagosan sokkal kevesebben is dolgoztak rajta. További példaként megfigyelhető a JavaScript, ami mindkét idő alapú statisztikában jobb eredményeket mutat, mint a legtöbb vizsgált nyelv, ami valószínűleg a kiemelkedő számú aktív fejlesztőnek köszönhető.

CVE javítása alatt keletkező commitok száma A commitok száma hasonló információkkal láthat el mint a fejlesztők száma, azonban kicsit más fókusszal, ugyanis egyetlen fejlesztő készíthet több commitot is, ennek eredményeként a commitok, sokkal inkább a fejlesztés sebességét jellemzik. Ha egy nyelvben hosszú időn keresztül aktívak a CVE-k és átlagosan kevés fejlesztő dolgozik rajta, alacsony commit számmal, az arra enged következtetni, hogy a vizsgált projektek nem feltétlenül aktívak. Természetesen aktívabb projektekkel rendelkező nyelvben is előfordulhatnak hasonló statisztikák, hiszen ez a statisztika nem veszi figyelembe a commitok méretét. Ha egy projektben kevesebb, de nagy méretű és hatású commit készül, nem nevezhető kevésbé aktívnak mint egy olyan, amiben sok de marginális változtatásokat tartalmazó commit található.



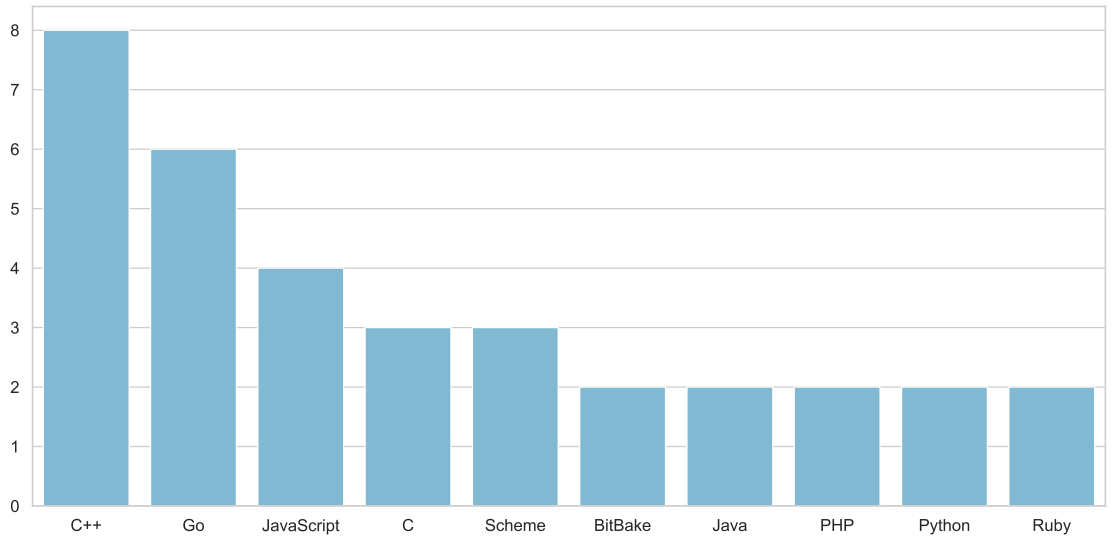
5.5. ábra. Átlagos commit szám

Az 5.5. ábrán látható, hogy a mintákban jelentős eltérés tapasztalható az elkészült commitok száma között. A Scheme projektjeiben például rengeteg commit készül, és viszonylag sok az aktív fejlesztő mialatt egy CVE aktív, ennek ellenére aránylag lassú a válaszidő a sérülékenységekre.

5.4. Átlagos fájl és sor változtatások

Az átlagos fájl és sor változtatások azt mutatják, hogy egy átlagos CVE mekkora hatást gyakorolt az adott nyelvek projektjeire. Ezt rengeteg faktor befolyásolja, nem csak nyelvi, de projekt szinten is. Különböző fejlesztési konvenciók komolyan módosíthatják az itt mutatott átlagokat.

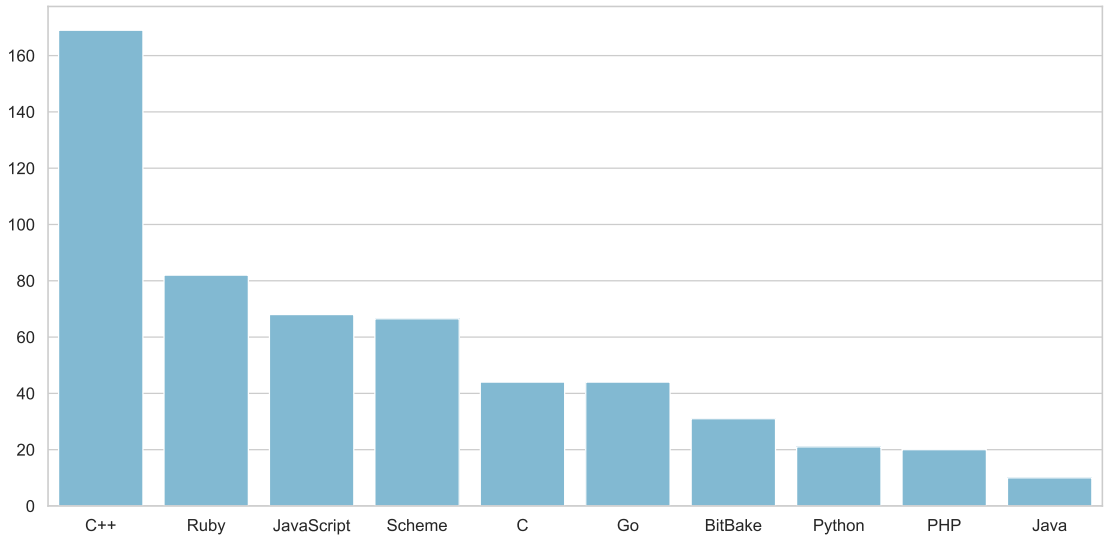
Átlagos fájl módosítások commitonként nyelvekre bontva A fájl módosítások elsősorban a hibák kiterjedtségét jellemzik ideális esetben. Ha egy sérülékenység kiküszöböléséhez több fájl módosítására is szükség van, feltehető, hogy nagyobb részét érintette az adott projektnek. Természetesen ez nem minden esetben mondható el, hiszen erre nagy befolyása lehet az adott termék belső struktúrájának is. Ha a projekt összességében kevés fájlból áll, az átlagos módosítások száma valószínűleg kevesebb lesz, mint egy jobban szegmentáltnak.



5.6. ábra. Átlagos fájl módosítások

Érdemes külön kiemelni itt a Go-t, ami az általunk vizsgált esetekben valóban a hibák kiterjedtsége miatt igényel ennyi fájl változtatást. A Go estében magában a nyelvben előfordulnak CVE-k, amik hatására a fejlesztők kénytelenek újabb nyelvi verziókra frissíteni a projektjeiket, ami rengeteg fájl módosításával járhat. C++-ban viszont a kiemelkedő szám nem feltétlenül a hibák típusából ered, hanem a vizsgált projektek szokásaiból. Gyakran előfordul, hogy a CVE-ket csak egy-egy nagyobb kiadás alkalmával nevezik meg, így egy commit nagy mennyiségű fájlváltozással jár.

Átlagos sor módosítások commitonként nyelvekre bontva A sor módosításokat az eltávolított, és hozzáadott sorok összegeként értelmeztük, ami ahhoz vezet, hogy ha jelentős mennyiségű létező kód módosul, az átlag lényegesen jobban romlik, mintha ugyanannyi új sort adtak volna hozzá. Ennek oka az, hogy a git egy sor módosítását néha úgy érzékeli mintha kitöröltük volna az egész sort, és újra írtuk volna a módosításokkal kiegészítve, tehát összesen két sort változtattunk.



5.7. ábra. Átlagos sor módosítások

Az itteni adatok tovább árnyalják a korábbi eredményeket, hiszen említettük, hogy előfordulhat, hogy kevés de hosszú commit készül egy projekt fejlesztése során. Az 5.7. ábrán láthatóak ezek az adatok. Itt ismét kiemelkedő értékekkel rendelkezik a C++, aminek okait korábban már tárgyaltuk. A Go-val kapcsolatos állításainkat is jól szemlélteti ez az eredmény, hiszen az új nyelvi verzióra való frissítés általában kevesebb sor- és több fájlmódosítást igényel.

5.5. CWE-kkel kapcsolatos statisztikák

A CWE-k csoportosítások a CVE-k számára, általánosabban foglalják össze a hiba típusát, például memória szivárgás, SQL injekció, stb. Ezek segítségével lehetőségünk van arra, hogy lényegében átkonvertáltjuk a specifikus hibajavításokról létrehozott adatainkat, és hibatípusok javítási adataira általánosítsuk őket. Ezek segítségével magasabb szintű megállapításokat tehetünk az egyes nyelvek gyakori hibatípusaira vonatkozóan.

A leggyakoribb CWE-k nyelvenként Egy nyelv leggyakoribb hibáinak megismerése hasznára lehet mind fejlesztőknek mind szoftver cégeknek a hibák és sérülékenységek felkutatásába fektetett erőforrások optimálisabb elosztásában, illetve az elkerülésükben. Habár az adathalmazunk nem elég nagy ahhoz, hogy indikatív legyen a nyelvekről álta-

lánosságban, arra több mint alkalmas, hogy betekintést nyerjünk a különböző nyelvekre jellemző tendenciákba.

Language	CWE Group	Percentage
BitBake	CWE-119	19.41%
	CWE-125	13.32%
	CWE-20	11.67%
Go	CWE-400	17.55%
	CWE-20	9.30%
	CWE-295	8.16%
Python	CWE-20	17.35%
	CWE-200	9.46%
	CWE-79	5.99%
Java	CWE-502	13.87%
	CWE-119	11.48%
	CWE-20	8.61%
	CWE-200	5.74%
	CWE-79	5.62%
	CWE-400	5.26%
JavaScript	CWE-400	14.78%
	CWE-20	12.17%
	CWE-200	9.56%
	CWE-79	6.95%
	CWE-119	6.08%

Language	CWE Group	Percentage
C	CWE-119	17.68%
	CWE-125	15.78%
	CWE-200	7.18%
	CWE-20	8.00%
C++	CWE-119	55.93%
	CWE-20	6.35%
	CWE-200	6.21%
Ruby	CWE-119	13.51%
	CWE-20	11.98%
	CWE-79	8.18%
	CWE-22	6.39%
Scheme	CWE-119	23.86%
	CWE-20	7.69%
	CWE-416	6.81%
	CWE-125	6.17%
PHP	CWE-79	18.79%
	CWE-352	11.28%
	CWE-20	11.27%
	CWE-77	6.76%
	CWE-89	6.67%

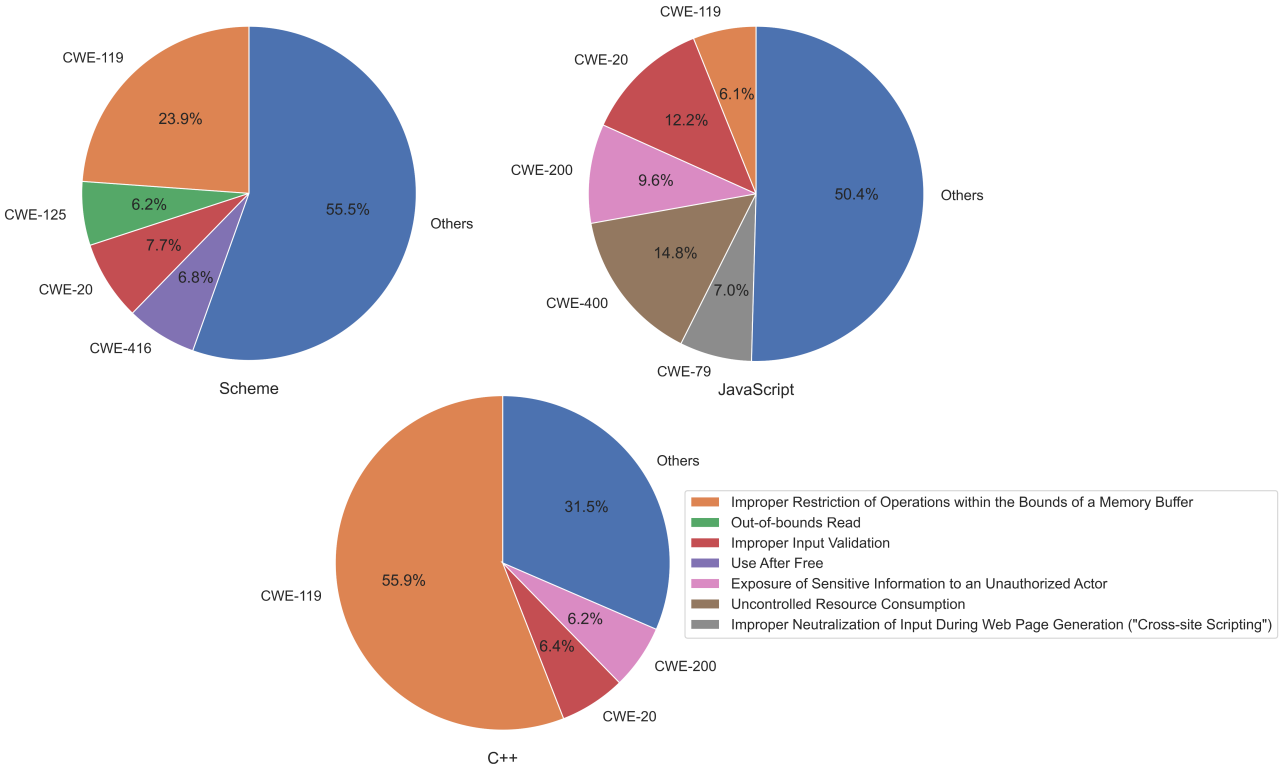
5.2. táblázat. Most Common CWEs per languages

A 5.2 táblázatban láthatóak a leggyakoribb CWE-k nyelvenként. Itt csak azokat tüntettük fel, amik több mint 5 százalékos prevalenciával fordulnak elő, hiszen azok a sérülékenységi osztályok, amik ennél ritkábban fordulnak elő, nem nevezhetőek jellemzőnek. Az általunk vizsgált adathalmaz azt mutatja, hogy a legtöbb nyelvnek van 1-2 kiemelkedő hiba típusa. Az előfordulás aránya változékony, C++ esetében például a CWE-119[2] messze a legjellemzőbb, viszont például Java-ban a leggyakoribb hiba típusok nem nyomják el teljesen a többit, maximum 14 százalékos jelenléttel rendelkeznek.

A sérülékenységek eloszlásában történő nyelvek közötti differencia mértéke lehet annak, hogy mennyire nehéz elkerülni őket. C++-ban közismert hibaforrás a memória nem megfelelő kezelése, ezt igazolja túlnyomó dominanciája az ezzel kapcsolatos hibáknak.

[2]Improper Restriction of Operations within the Bounds of a Memory Buffer

A tény, hogy a hibatípus jellemzősége ennyire közismert, és mégis ilyen sokszor fordul elő, arra enged következtetni, hogy aránytalanul nehéz elkerülni. A többi nyelvben nem látható ennyire kiemelkedő hiba, de ez bizonyos szempontból rosszabb is lehet, hiszen így nehezebb meghatározni, hogy mire a legfontosabb odafigyelni.



5.8. ábra. CWE vizualizáció (C++, JavaScript, Scheme)

A kördiagramok segítségével (5.8. ábra) vizualizáltuk a leggyakoribb CWE-ket C++-ban, JavaScript-ben, és Scheme-ben. Láthatjuk, hogy a C++ kivételével minden esetben, az előforduló CWE-k felét teszik ki a feltüntetettek.

5.6. CVE anomáliák, és okaik

Egy CVE mint korábban említettük, egy konkrét projekt hibája, és így az elvárt viselkedésük általánosan meg kellene egyezzen egy átlagos hibáéval. Csak 1 projektben kellene említve lennie, maximum 1-2 commit üzenetben kellene szerepelnie. Az utóbbi elvárás azért merülhet fel, mert nyílt forráskódú projektekről beszélünk, tehát ha egy részlegesen javított hiba túl hamar szivárog ki, könnyedén előfordulhatnak visszaélések. A CVE-k

előfordulások többsége ezeknek megfelelően viselkedik, így a kivételeket anomáliának nevezzük.

Az alábbiakban következő megállapításokat az anomáliák manuális vizsgálata során gyűjtött adatokra alapozzuk. Az ellenőrzés során a publikusan elérhető commit üzenetek tartalmára hagyatkozunk, ami nem tette lehetővé a teljes körű vizsgálatot, hiszen néhány, az eszközünk által megtalált commit privát láthatóságú volt, így nem fértünk hozzá további információhoz.

Többször említett CVE-k Több korábban említett statisztika is az ilyen esetekre vonatkozik, hiszen értékes információ szűrhető ki egy-egy hiba aktív időszakából, azonban egy CVE többszöri említése nem szokványos, ahogy ezt az előző bekezdésben említettük. Ennek ellenére gyakran előfordul, hogy többször említenek adott hibákat, erre a leggyakoribb indok, hogy a javítást előzőleges vizsgálatnak vetnek alá mielőtt az élő verzióba is beveszik. Az újbóli említések okai legtöbbször biztonsági szempontból jelentéktelenek, később teszteket készítenek a hiba újbóli előfordulásának elkerülésére, vagy régebbi verziókban is implementálják a javításokat. Ugyan ebbe a kategóriába sorolhatóak olyan esetek is ahol egy függőségben található sérülékenység ideiglenes kiküszöbölése érdekében átmeneti megoldásokat készítenek, amiket amikor a függőségben elkészültek a javítással kivesznek. Egy ilyen esetre a Tomcat[3] commitjai között találtunk példát. Az első üzenet amit az eszközünk észrevett már egy korábbi "workaround" (kerülő megoldás) javításához tartozik[4]. A második, és utolsó az eszközünk által felfedezett commmi során egy végső javítás kerül implementációra, ami a Java virtuális gép által biztosított megoldásokat implementálja[5]. Előfordul azonban olyan eset is, hogy egy hiba javítására készített módosítás egyéb funkciók hibás működését hozza magával, lényegében további hibákhoz vezet. Egy hiba köztes változtatások is során újra megjelenhet, ilyen eseményre példa a curl[6] esetében a CVE-2016-5419[7]. A hibát először 2016 augusztusában javították[8], 1 évvel az első commit után újból javításra szorul, ugyanis egy köztes verzió során ismét

[3]http://tomcat.apache.org/
[4]https://github.com/apache/tomcat/commit/30af3f
[5]https://github.com/apache/tomcat/commit/b4e948
[6]https://curl.se/
[7]https://nvd.nist.gov/vuln/detail/CVE-2016-5419
[8]https://github.com/curl/curl/commit/247d890

felbukkan a probléma[9].

Több projektben megemlített CVE-k Többször is említettük, hogy egy CVE egy adott termék hibája, ennek ellenére előfordul, hogy egy hiba több különböző projektben is meg van említve. A leggyakoribb indok erre a jelenségre a projektek egymástól való függése. Ha egy kódbázisban egy másik ettől függetlenül fejlesztett projekt funkcióit használják, azt függésnek nevezzük. A nyílt forráskódú fejlesztés során egyre többször fordul elő, hogy akár több száz más projekttel kerül egy szoftver függési viszonyban, közvetlenül vagy közvetetten. Az ilyen tömeges függés mellékhatásaként a különböző sérülékenységek képesek elterjedni, és egyszerre több projektre is hatással lenni. A legtöbb esetben a hiba javításra kerül a felhasznált projektben, így csak annak frissítésére van szükség, ekkor fordul elő, hogy az új verzió használatának indoklásaként említik az így kiküszöbölt CVE-t.

Hasonlóan viselkednek a forkok (elágaztatások), illetve a nyelvi hibák. A lényegi különbség elágaztatott programok esetén, hogy a fejlesztők tudatosan használják egy jelentős részét egy másik kódbázisnak, így felkészültebbek a potenciális sérülékenységekre. A nyelvi hibák különlegesek abból a szempontból, hogy garantáltan minden azon a nyelven készített projektben javításra szorulnak, ezzel egy konkrét összehasonlítási lehetőséget biztosítva a különböző projektek között, mind sebesség mind dokumentáltság terén.

Mint láthatjuk a többszöri említéseket közös alapok, vagy függési viszonyok okozzák. Fontos észrevétel azonban, hogy nem feltétlenül említik a CVE-ket név szerint. Gyakran előfordul, hogy egy projektben egy használt modul frissítése során említenek egy CVE-t, azonban ezt a modul fejlesztői a javítás véglegesítése során nem teszik meg. Ennek eredményeként a CVE-k eredetének megállapítása nem minden esetben triviális a jelenlegi eszközünk eredményei alapján.

Rendhagyó CVE javítások Rendhagyónak olyan javításokat nevezünk, amelyek egyszerre több CVE említéssel rendelkeznek, vagy az átlagosnál jelentősen több sormódosítással bírnak. A két feltétel gyakran egyszerre fordul elő, ezek az esetek nem jeleznek meglepő viselkedést, egyszerre több hiba javítása természetesen az átlagosnál több sort

[9]https://github.com/curl/curl/commit/33cfcfd

igényel. Az ilyenek legtöbbször egy újabb verzió kiadásához történő előkészületek vagy egy korábbi kiadás komolyabb hibáinak utólagos javításának részei.

Olyan esetekben, amikor viszonylag kevés sorból több hibát javítanak, az általunk vizsgált összes alkalommal egy használt modul verziójának frissítése során került sor. A nagy sormódosítási igényt, várható módon, a javítások nehézsége okozta, ezek az esetek azonban olyan ritkák, hogy az általunk vizsgált kódhalmazban csak néhány példa található. Ezek közül egy a Hue[10] esetében fordul elő, itt habár 2 CVE-t javítanak egyszerre, a kettőhöz együttesen szükséges módosítások száma jelentősen meghaladja az átlagot[11].

[10] https://gethue.com/

[11] https://github.com/cloudera/hue/commit/bf22469

6. fejezet

Konklúzió

Jelen dolgozat és kutatás célja az volt, hogy megvizsgáljuk, hogy vannak-e nyelvekre jellemző minták CVE-kkel és CWE-kkel kapcsolatban. Ennek eléréséhez készítettünk 3 eszközt, amelyek együttes működtetésével létrehoztunk egy adathalmazt a népszerűbb nyílt forráskódú projektek ezekkel kapcsolatos adatairól. Az adatok közül több százat manuálisan ellenőriztünk, és mind helyesnek bizonyult. Ezek után részben kézzel, részben automatizált módszerekkel statisztikákat készítettünk, amelyek betekintést engednek a sérülékenységek ellen folytatott küzdelembe.

Eredményeink többek közt azt mutatják, hogy általánosan aktívabb közösségek nem feltétlenül hatékonyabbak a hibák kiküszöbölésében. A CVE-k súlyossága és a javításukhoz szükséges idő között nincs releváns korreláció. Illetve, az általunk vizsgált minta alapján bizonyos nyelveknek (pl. C++) vannak erősen jellemző hibatípusai, míg másoknak nincsenek egyértelműen kiugró hibacsoportok.

Ezek az információk, a limitált adatmennyiségünk ellenére, hasznosak lehetnek projektmenedzsereknek, és fejlesztőknek egyaránt, hiszen informálhatják döntésüket azzal kapcsolatban, hogy melyek azok a potenciális hibaforrások, amelyekre a leginkább érdemes figyelmet fordítani, vagy megelőzésükre célzott szakképzéseket indítani. Felhasználhatóak továbbá különböző hiba előrejelzésekkel kapcsolat gépi tanulási feladatokhoz is.

A prezentált eredményekből készült egy elfogadott, bemutatott nemzetközi folyóiratcikk is.[1]

Köszönetnyilvánítás

Ez a TDK munka a SETIT projekt (2018-1.2.1-NKP-2018-00004)[1] keretében készült. Szeretnénk továbbá köszönetet mondani témavezetőinknek Dr. Hegedűs Péternek és Antal Gábornak, akik szakmai támogatásukkal segítették a projekt sikerét.

[1] A 2018-1.2.1-NKP-2018-00004 számú projekt a Nemzeti Kutatási és Innovációs Alapból biztosított támogatással, a "Nemzeti Kiválósági Program: 2018-1.2.1-NKP" pályázati program finanszírozásában valósult meg.

Irodalomjegyzék

[1] Gábor Antal, Balázs Mosolygó, Norbert Vándor, and Péter Hegedűs. A data-mining based study of security vulnerability types and their mitigation in different languages. In *International Conference on Computational Science and Its Applications*, pages 1019–1034. Springer, 2020.

[2] Thanapon Bhuddtham and Pirawat Watanapongse. Time-related vulnerability lookahead extension to the cve. In *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–6. IEEE, 2016.

[3] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 869–885, 2019.

[4] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense*, LSAD '06, page 131–138, New York, NY, USA, 2006. Association for Computing Machinery.

[5] Andrew Kronser et al. Common vulnerabilities and exposures: Analyzing the development of computer security threats. 2020.

[6] D. Kuhn, Mohammad Raunak, and Raghu Kacker. An analysis of vulnerability trends, 2008-2016. pages 587–588, 07 2017.

[7] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[8] Xiang Li, Jinfu Chen, Zhechao Lin, Lin Zhang, Zibin Wang, Minmin Zhou, and Wanggen Xie. A mining approach to obtain the software vulnerability characteristics. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 296–301. IEEE, 2017.

[9] Fabio Massacci and Viet Hung Nguyen. Which is the right source for vulnerability studies? an empirical analysis on mozilla firefox. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, New York, NY, USA, 2010. Association for Computing Machinery.

[10] V. Mounika, X. Yuan, and K. Bandaru. Analyzing cve database using unsupervised topic modelling. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 72–77, 2019.

[11] Syed Shariyar Murtaza, Wael Khreich, Abdelwahab Hamou-Lhadj, and Ayse Basar Bener. Mining trends and patterns of software vulnerabilities. *Journal of Systems and Software*, 117:218–228, 2016.

[12] Sarang Na, Taeeun Kim, and Hwankuk Kim. A study on the classification of common vulnerabilities and exposures using naïve bayes. In *International Conference on Broadband and Wireless Computing, Communication and Applications*, pages 657–662. Springer, 2016.

[13] Luis Gustavo Araujo Rodriguez, Julia Selvatici Trazzi, Victor Fossaluza, Rodrigo Campiolo, and Daniel Macêdo Batista. Analysis of vulnerability disclosure delays from the national vulnerability database. In *Anais do I Workshop de Segurança Cibernética em Dispositivos Conectados*. SBC, 2018.

[14] Jukka Ruohonen, Sami Hyrynsalmi, Sampsa Rauti, and Ville Leppänen. Mining social networks of open source cve coordination. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, pages 176–188, 2017.

[15] Jukka Ruohonen, Sampsa Rauti, Sami Hyrynsalmi, and Ville Leppänen. A case study on software vulnerability coordination. *Information and Software Technology*, 103:239–257, 2018.

[16] Sagar Samtani, Shuo Yu, Hongyi Zhu, Mark Patton, John Matherly, and HsinChun Chen. Identifying supervisory control and data acquisition (scada) devices and their vulnerabilities on the internet of things (iot): a text mining approach. *IEEE Intelligent Systems*, 2018.

[17] Luis Alberto Benthin Sanguino and Rafael Uetz. Software vulnerability analysis using cpe and cve. *arXiv preprint arXiv:1705.05347*, 2017.

[18] Clemens Sauerwein, Christian Sillaber, Michael M Huber, Andrea Mussmann, and Ruth Breu. The tweet advantage: An empirical analysis of 0-day vulnerability information shared on twitter. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 201–215. Springer, 2018.

[19] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781, 2012.

[20] Anshu Tripathi and Umesh Kumar Singh. Evaluation of severity index of vulnerability categories. *International Journal of Information and Computer Security*, 5(4):275–289, 2013.

[21] Leon Li Wu, Boyi Xie, Gail E Kaiser, and Rebecca Passonneau. Bugminer: Software reliability analysis via data mining of bug reports. 2011.

[22] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE, 2017.

[23] Yasuhiro Yamamoto, Daisuke Miyamoto, and Masaya Nakayama. Text-mining approach for estimating vulnerability score. In *2015 4th International Workshop on*

Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), pages 67–73. IEEE, 2015.

[24] Su Zhang, Doina Caragea, and Xinming Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *International conference on database and expert systems applications*, pages 217–231. Springer, 2011.

A. függelék

Említett CWE-k definíciói

CWE-20 Improper Input Validation

A termék bemenetet vagy adatot kap, amit nem vagy helytelenül validál.

CWE-22 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

A szoftver egy elérési útvonalat vár, amiben nem semlegesíti kellően a különleges elemeket, így előfordulhat, hogy korlátozott hozzáférésű dokumentumokhoz enged hozzáférést.

CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection')

A szoftver előkészít egy parancsot vagy egy részét, amihez külsőleg befolyásolt részleteket használ, de ezeket nem, vagy nem megfelelően semlegesíti, ennek hatására a továbbadott parancs módosulhat.

CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

A szoftver nem, vagy nem megfelelően semlegesít felhasználó által befolyásolt bemeneteket, amik később más felhasználóknak lesznek kiszolgálva weboldalon.

CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

A szoftver előkészít egy SQL parancsot vagy egy részét, amihez külsőleg befolyásolt részleteket használ, de ezeket nem, vagy nem megfelelően semlegesíti, ennek hatására a továbbadott SQL parancs módosulhat.

CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer

A szoftver műveleteket hajt végre egy memória pufferben, de képes a korlátokon kívül is írni, olvasni.

CWE-125 Out-of-bounds Read

A szoftver adatokat olvas a használt puffer vége után, vagy kezdete előtt.

CWE-200 Exposure of Sensitive Information to an Unauthorized Actor

A termék bizalmas információhoz enged hozzáférést, nem jogosult felhasználóknak.

CWE-295 Improper Certificate Validation

A szoftver nem, vagy nem megfelelően validálja a tanúsítványt.

CWE-352 Cross-Site Request Forgery (CSRF)

A webalkalmazás nem ellenőrzi kellően, hogy egy helyes kérés valóban a felhasználótól származik-e.

CWE-400 Uncontrolled Resource Consumption

A szoftver nem korlátozza kellően a limitált erőforrások foglalását, és karban tartását, ami lehetővé teszi egy külső hatás számára, hogy befolyásolja a felhasznált nyersanyag mennyiséget, ami ezek elfogyásához vezethet.

CWE-416 Use After Free

A program már felszabadított memóriát használ.

CWE-502 Deserialization of Untrusted Data

Az alkalmazás deszerializál nem megbízható adatot anélkül, hogy ellenőrizné, hogy érvényes eredményhez vezet-e.