Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Vándor Norbert Rudolf

2021

Szegedi Tudományegyetem

Informatikai Intézet

Sorszintű sérülékenység-előrejelző modell implementálása JavaScript nyelvű rendszerekhez

Implementing a line-based vulnerability prediction model for JavaScript systems

Szakdolgozat

Készítette:

Vándor Norbert Rudolf

informatika szakos

hallgató

Témavezetõ:

Dr. Hegedűs Péter

tudományos munkatárs

Szeged

2021

Tartalomjegyzék

Feladatkiírás

A feladat egy olyan sérülékenység-előrejelző modell implementálása volt, amely JavaScript fájlokat feldolgozva megjelöli azok sérülékenységet tartalmazó sorait, és a döntést példával is alátámasztja.

Tartalmi összefoglaló

A mai világban a szoftverfejlesztés egyik legfontosabb kérdése minden kétséget kizárólag a biztonság. Ezen biztonsági kockázatok kivédésére mind a kutatók, mind a programozók többféle eszközzel és technikával álltak elő. Ilyen eszközök lehetnek például különböző statikus vagy dinamikus elemzők, de manapság kezdenek egyre jobban elterjedni a mesterséges intelligenciára épülők. Ezen belül is főként gépi tanulást használnak, amivel igen meggyőző eredmények is tudnak születni a biztonsági problémák megtalálása és előrejelzése terén. Viszont ezek az eszközök általában két ponton is lehet, hogy nem tesznek eleget a szoftverfejlesztők elvárásainak: az egyik a megmagyarázhatóság, a másik pedig a részletesség.

A dolgozat célja, hogy egy olyan modellt (és az azt használó eszközt) mutasson be, ami egyszerűen, de hatékonyan képes az esetlegesen sérülékeny forráskód előrejelzésére JavaScript alapú rendszerekben. Ez a modell mind a megmagyarázhatóságában, mind az előrejelzések részletességében igyekszik a fent említett problémákra megoldást találni: az előrejelzések nem fájl- vagy metódusszintűek, hanem egészen a kódsorokig lemennek. Ez pedig eléggé részletes ahhoz, hogy a fejlesztők azonnal meg tudják tenni a hiba javításához szükséges lépéseket. Ezen kívül a modell azt is megmondja, hogy az egyes sérülékenynek vélt sorokat miért jelezte annak: rámutat a tanítóhalmazban található, a kérdéses sorhoz legjobban hasonlító elemre.

A dolgozat eredményeiből publikáció [8] is született az ICCQ nemzetközi konferencián. [1].

[1] International Conference on Code Quality, `https://www.iccq.ru`

Bevezetés

Mivel a különböző sérülékenységek kihasználása egyre könnyebbé válik az egyre növekvő számú komplex szoftver elterjedésének köszönhetően, ezért a mai szoftverfejlesztésben kiemelt központi szerepet kap a biztonság. Éppen ezért a biztonsági problémák megoldására komoly hangsúlyt kell fektetni a szoftverfejlesztési folyamat minden fázisában. Ez a dolgozat az implementációs- és a minőségbiztosítási fázisra fókuszál.

Ahogy már korábban említésre került, manapság a sérülékenységeket egyre több esetben különböző gépi tanulási modellekkel keresik, amik viszont általában két ponton nem tesznek eleget az elvárásoknak: ez pedig a megmagyarázhatóság és a részletesség. A megmagyarázhatóság azt jelenti, hogy a modell az előrejelzése mellé magyarázatot is ad, hogy miért azt az eredményt adta. Ennek hiányában a fejlesztők nem tudják, hogy az aktuálisan sérülékenynek mondott kódrészlet valójában az-e, vagy csak egy hamis pozitív találatról van szó. Ez nagyban megnehezíti a javítási munkálatokat, mert így vagy le kell ellenőrizni, hogy a modell valóban helyesen jelzett, vagy be kell bizonyítani, hogy az a kódrészlet hibamentes. Részletesség alatt pedig azt kell érteni, hogy a modell mekkora részekre bontja a forráskódot, amikor elemzi. Ez lehet például fájl, osztály, metódus, blokk vagy akár sor is. Minél kevésbé részletes egy modell, annál jobb teljesítményt nyújt. Viszont a gyakorlati haszna nagyban függ attól, hogy mennyire tudja lokalizálni a találatát. A legtöbb használt sérülékenység-előrejelző modell fájl-, bináris- vagy osztályszinten dolgozik, de még a legrészletesebb modellek is csak metódusszintig mennek le.

Ezen kívül a sérülékeny kódsorok általában valamilyen mintát követnek, amiket viszont a gyakran használt módszerekkel nem lehet egyszerűen meghatározni. Ennek a kiküszöbölésére a Word2Vec használata megfelelőnek bizonyult, ugyanis nem csak a természetes nyelv, hanem a forráskód esetén is alkalmazható a szemantikailag hasonló kódsorok megtalálására.

Fontos még megemlíteni, hogy a legkorszerűbb gépi tanulási modellek (főleg a mélytanulás) nagy mennyiségű tanító adatot igényelnek. Egy valós helyzetben egyszerűen nincs meg a megfelelő mennyiségű adat, illetve az esetek nagy részében ez az adat kiegyensúlyozatlan, vagyis a sérülékeny kódrészek jóval kevesebbszer fordulnak elő, mint a nem sérülékenyek, ami tovább nehezíti a modell tanítását. Éppen ezért a vektoralapú megközelítésnek ebben az esetben hatalmas előnye van: csak egyszer van szükség komoly tanításra, amikor a Word2Vec modellt előállítja, magához az előrejelzéshez már egyáltalán nem használ gépi tanulási technikákat. Nem vonatkozik rá a kiegyensúlyozatlanság problémája sem, ugyanis csak és kizárólag sérülékeny kódrészekkel dolgozik. Ebből kifolyólag bármekkora lehet a tanító halmaz (bár nagyobb méretű adathalmazzal gyorsabb), és iteratívan is növelhető.

Ebben a dolgozatban a fentiek szerint leírt modell kerül bemutatásra, ami sorszinten jelzi a potenciális sérülékenységeket JavaScript rendszerekben. Azért esett a választás a JavaScriptre, mert ez az egyik legnépszerűbb ma használt nyelv, viszont kutatások tekintetében kevesebb figyelmet kap mint például a Java vagy a C/C++.

1. fejezet

Kapcsolódó munkák

Annak ellenére, hogy gépi tanulást használni szoftverkomponensek sérülékenységeinek előrejelzésére egy viszonylag új ötlet, már rengeteg kutatás készült a témában.

Fájl szint Shin és társai [13] egy empirikus esettanulmányt végeztek két nagy kódbázison. Azt vizsgálták, hogy a kód komplexitási, a kód lemorzsolódási, és a fejlesztői aktivitási metrikák hasznosak-e sérülékenység-előrejelzés szempontjából. Azt tapasztalták, hogy az általuk vizsgált fájlszintű metrikák képesek sérülékenységek előrejelzésére. A Mozilla kódbázisának ismert sérülékenységeinek 70,8%-át, és a Red Hat Linux kernelének 68,8%-át voltak képesek megtalálni.

Chowdhury és társai [3] létrehoztak egy sérülékenység-előrejelző keretrendszert, ami a CCC (Complexity, Coupling, and Cohesion) metrikát [2] használja. Négy statisztikai és gépi tanulási módszert hasonlítottak össze, és arra a következtetésre jutottak, hogy az ő esetükben a döntési fa alapú technikák jobban teljesítettek, mint a statisztikai modellek.

Morrison és társai [7] reprodukálták Zimmermann és társai [18] munkáját, és annak mintájára egy új modellt készítettek. A modelljük binárisokra, és forráskódra is fájl szinten működik, számos kód metrikát felhasználva. Azt észlelték, hogy a sérülékenység-előrejelzés nem praktikus binárisok esetén, mert túl sok ideig tart átvizsgálni őket. A forráskód metrika alapú megközelítésük 0.5 alatti pontossággal és 0.2 alatti fedéssel rendelkezett. A kutatók több algoritmust is ellenőriztek mint például, az SVM-et, a Naiv-Bayes-t, a random forest-et, és logisztikus regressziót. Az adathalmazukon a Naiv-Bayes, és a random forest teljesített a legjobban.

A munkájuk során, Yu és társai előterjesztették a HARMLESS-t [17]: egy költséghatékony SVM eszközt, ami képes sérülékenységek előrejelzésére, illetve rá tud mutatni, hogy mely további kódrészeket érdemes vizsgálni. A futása során a HARMLESS azokat a forrás fájlokat jelöli meg, amelyek a legnagyobb valószínűséggel tartalmaznak sérülékenységeket. Ők is a Mozilla kódbázisát használták a tanulmányuk során, és három különböző tulajdonsághalmazt vizsgáltak: az egyik a metrikák, a másik a szöveg, a harmadik pedig szövegbányászasi technikák és crash dump stack trace-ek kombinációja. Az eredményeik azt mutatták, hogy a HARMLESS megtalálja a sérülékenységek 60-90%-át, a forrásfájlok mindössze 6-34%-át megvizsgálva. Emellett az előző tanulmányaikat is túlteljesítettek mind fedés mind költséghatékonyság terén [16].

Jimenez és társai [6] elkészítették a VulData7-et, egy kiegészíthető keretrendszert (és adathalmazt) valós sérülékenységekből, amiket automatikusan gyűjtöttek szoftver archívumokból. A VulData7 egy általános keretrendszer, ami sérülékenységeket tartalmaz négy kritikus nyílt forráskódú projektből, fájlszinten. Viszont a keretrendszerük rengeteg adatot tartalmaz, emiatt egy további feldolgozó lépésre van szükség, hogy az használható legyen sérülékenység előrejelzés során.

Neuhaus és társai [9] bemutatták az eszközüket, a Vulture-t, ami képes sérülékeny komponenseket megtalálni a forráskódban, a függőségeket vizsgálva. Egy komponens a megközelítésükben egy header-forrás pár (ha mindkettő elérhető, egyébként csak a forrásfájl) C/C++-ban (és egy .java fájl Java esetén). Analizálták a függőségeket és függvény hívásokat a komponensek között, és SVM-et használták az osztályozásukra. Az eszközük a sérülékeny komponensek felét azonosította a rendszerben, a megjelölt komponensek pedig kétharmada helyesen volt megjelölve.

Osztály szint Siavvas és társai [15] megvizsgálták a viszonyt a szoftvermetrikák és specifikus sérülékenység típusok között. Osztályszintű metrikákat használtak, és azt észlelték, hogy a szoftvermetrikák nem biztos, hogy elégséges indikátorok bizonyos sérülékenységi típusokhoz.

Basili és társai [1] objektum orientált dizájn metrikákat [2] használtak arra, hogy megjelöljék a hibára hajlamos osztályokat, amihez egy statisztikai megközelítést alkalmaztak. Ezzel ellentétben a mi megközelítésünk elsősorban gépi tanulás alapú.

	File level	Class level	Function level	Block \ Line level
Pascarella és társai [11]			•	
Gigerés társai [5]			•	
Basili és társai [1]		•		
Palomba és társai [10]		•		
Ferenc és társai [4]			•	
Siavvas és társai [15]		•		
Jimenez és társai [6]	•			
Neuhas és társai [9]	•	•		
Shin és társai [14]			•	
Shin és társai [13]		•		
Morrison és társai [7]	•			
Chowdhury és társai [3]	•			
Yu és társai [17]	•			

1.1. táblázat. Kapcsolódó munkák összefoglalása

Palomba és társai [10] létrehoztak egy modellt, hogy hibákat jelezzenek elő úgynevezett "code smell"-eket tartalmazó osztályokban. Több előrejelzési modellt kiértékelek: a legjobb eredményeket az egyszerű logisztikus modell érte el. Számos egyéb metrikát is megvizsgáltak, és bevezettek egy újat, amit intenzitásnak hívtak; létrehozására a JCodeOdor[1]-t használták.

Függvény szint Shin és társai. [14] létrehoztak egy empirikus módszert, hogy függvény sérülékenységeket jelezzenek elő statikus kódkomplexitási metrikák alapján. Mann–Whitney-próbát, és bináris logikai regresszió analízist használtak a tanulmányukban. Megmutatták, hogy a sérülékeny függvényeknek egyedi karakterisztikái vannak, amik segítségével megkülönböztethetőek a nem sérülékeny, de hibás függvényektől.

Giger és társai[5] metódus szintű előrejelző modelleket állítottak elő, változás- és forráskód-metrikákat használva. A modelljeik 84% pontosságot és 88%. fedést értek el.

Ferenc és társai [4] végeztek egy tanulmányt, amely során összevetettek nyolc sérülékenység előrejelző módszert, hogy megvizsgálják a teljesítményüket JavaScript függvény szinten. Létrehoztak egy adatbázist (statikus kódmetrikákat, és sérülékenységi információkat használtak az NVD[2] rendszeréből, illetve GitHubról gyűjtött javításokat) és ennek segítségével tanították be, és tesztelték az algoritmusokat. A legjobb eredményeket a KNN algoritmussal érték el (91% pontosság és 66% fedés). Ez a kutatás hasonlít a dol-

[1] `https://essere.disco.unimib.it/jcodeodor/`
[2] National Vulnerability Database

gozatban leírthoz, azonban ők más algoritmusokat és előrejelzési metrikákat használtak.

Pascarella és társai [11] reprodukálták Giger és társai [5] munkáját a metódus szintű hiba előrejelzéssel kapcsolatban. Előterjesztettek egy realisztikusabb kiértékelési stratégiát, mint amit az eredeti kutatásban használtak. A realisztikus stratégia azt mutatta, hogy a használt modellek közel véletlenszerűen működtek.

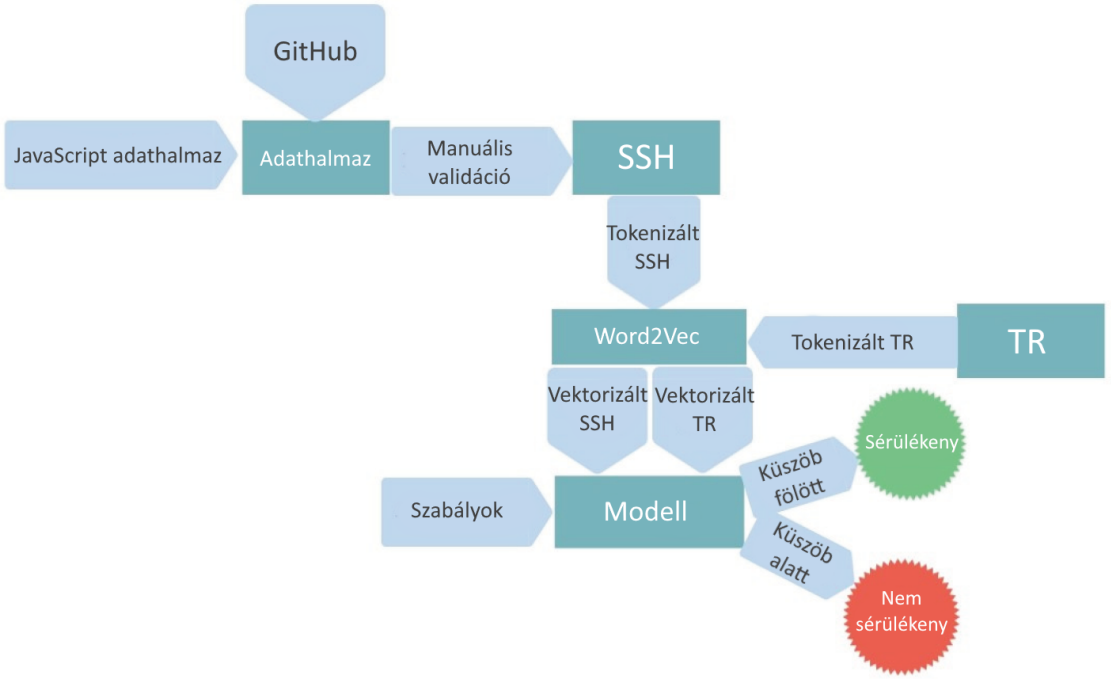
A 1.1 táblázat összefoglalja a kapcsolódó munkákat az előrejelzés szintje szerint.

2. fejezet

Megközelítés

A dolgozat elkészítéséhez 824 valós életből vett és megerősített JavaScript sérülékenység javítása került átvizsgálásra. Ezek megtalálhatóak a Ferenc Rudolf és társai [4] által publikált adathalmazban. A javítások sorokra lebontva manuálisan is ellenőrzésre kerültek: a folyamat során a változtatások közül csak azok a sorok lettek megtartva, amelyek valóban hozzájárultak a sérülékenység javításához.

Ezek a validált adatok egy adathalmazban, az úgynevezett Sérülékeny Sorok Halmazában (*SSH*) tárolódnak. A halmazról elmondható, hogy a korábban említett ellenőrzés miatt lényegében nem található hamis pozitív adat. Ez azért is fontos, mert a megközelítés alapja az, hogy az SSH soraihoz hasonlítjuk a Tesztelendő Rendszer (*TR*) sorait. Ez úgy történik meg, hogy mind az SSH, mind a TR soraiból vektorreprezentáció készül, majd páronként véve a vektorokat a modell kiszámolja a kettő közötti koszinusz távolságot. A távolságokból csak a legkisebbek maradnak meg (vagyis azok a sorpárok, amik a legközelebb vannak egymáshoz, tehát a legjobban hasonlítanak). A távolságból és néhány további szabály alkalmazásából a modell minden sorhoz számol egy valószínűséget, ami azt mondja meg, hogy mennyi az esélye, hogy az adott sor sérülékeny. Ez úgy valósul meg, hogy a kiszámolt valószínűséget összeveti egy küszöbértékkel, és ha a a küszöbérték fölött van, akkor sérülékenynek, egyébként pedig nem sérülékenynek jelöli meg. Minden egyes sérülékeny sorhoz az az SSH-beli sor is biztosítva van, amelyik a legjobban hasonlít rá. A megközelítés vizualizációja a 2.1 ábrán látható.



2.1. ábra. A folyamat leírása

2.1. Hasonlóságkeresés és a predikciós algoritmus

2.1.1. Forráskód feldolgozás

Azért, hogy a bemutatott módszert alkalmazni lehessen, először is vektorokat kell a forráskód soraihoz rendelni. Ehhez először a JavaScript forráskódokat tokenizálni kell, majd abból Word2Vec modellt készíteni a 2.1 táblázatban felhasznált tokenek segítségével. Néhány token vektorának vizualizációja látható az 2.2 ábrán.

A forráskód lexikális feldolgozása (vagyis a tokenek készítése) egy JsTokens[1] nevű nyílt forráskódú node modul segítségével történt. A JavaScript szintaxis tokenjein kívül néhány más token is hozzáadásra került. Ezek a tokenek különböző literál értékek, olyanok, amik fontosabb jelentést hordozhatnak. Ezek a literálok a következők: "HTML", "SQL", "HTTP", "GET", "POST", "Port" string literálok, illetve a 0, -1, 1 numerikus literálok. A literálok azonosítása egyszerű módon történik: ha az említett literál szerepel a kódban, akkor le lesz cserélve a neki megfelelő tokenre, egyébként nem (Például a "GET"-ből GETStringLiteral lesz). A HTML és SQL esetében a folyamat egy kicsit más, ugyanis egyik sem szerepel konkrétan a forráskódban. Az azonosításhoz speciális karak-

[1] `https://www.npmjs.com/package/js-tokens`

StringLiteral	IdentifierName	synchronized	debugger
transient	implements	1	await
SQLStringLiteral	protected	Infinity	NaN
if	byte	false	this
GETStringLiteral	yield	double	public
function	try	return	catch
HTMLStringLiteral	interface	null	volatile
native	in	0	true
RegularExpressionLiteral	enum	var	abstract
default	with	else	-1
HTTPSStringLiteral	goto	boolean	class
typeof	switch	case	break
POSTStringLiteral	while	package	undefined
delete	static	for	short
PortStringLiteral	long	throw	new
float	export	int	throws
HTTPStringLiteral	TemplateHead	instanceof	let
void	Invalid	super	arguments
NoSubstitutionTemplate	TemplateMiddle	continue	const
finally	eval	private	do
NumericLiteral	TemplateTail	import	char
extends	final		

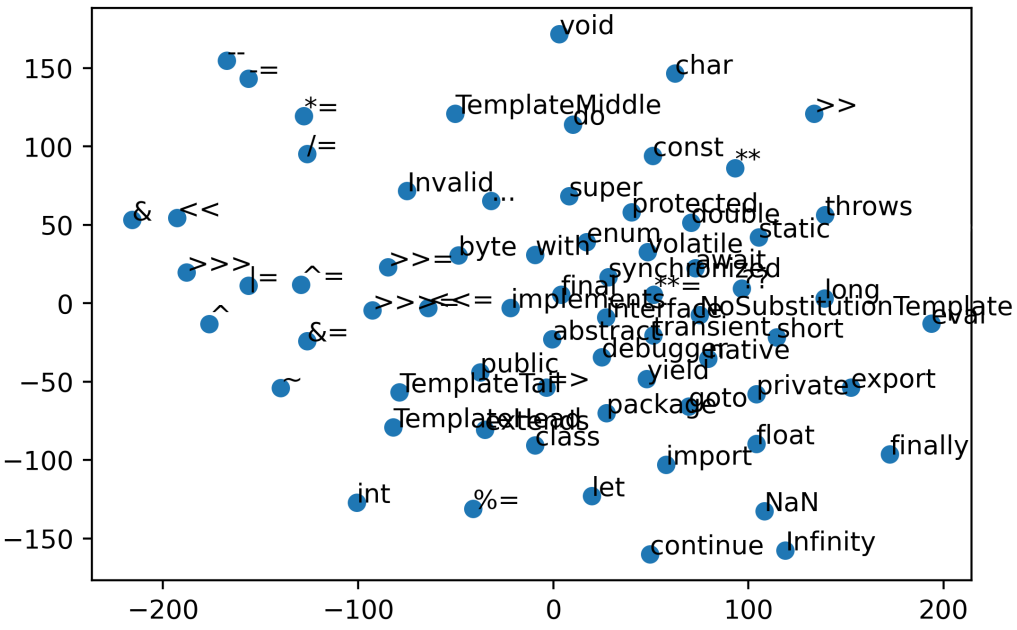
2.1. táblázat. A tokenek

tereket kell keresni, például HTML esetében a "<" és ">" karaktereket, az SQL esetében a "SELECT" és a "FROM" szavakat. Ezek a speciális értékek azért kerülnek a tokenek közé, mert, mint ahogy fentebb említve lett, fontos jelentést hordoznak: általában biztonsági funkcionalitáshoz kapcsolódnak. Éppen ezért sokkal kifejezőbb Word2Vec eredményre lehet számítani, mint nélkülük.

Az imént szemléltetett modell az, ami a legjobbnak bizonyult. De ez előtt több lehetőség is kipróbálásra került. Az első modellben minden elem tokenizálva lett, ami azt eredményezte, hogy túl általános lett, majdnem minden sor be lett jelölve. Második próbálkozásként a nyelv kulcsszavai nem lettek tokenizálva, ami valamennyivel javította az eredményeket. És végül a harmadik verzió az, amit fentebb említésre került.

A Word2Vec modell által használt corpushoz 150.000 JavaScript fájl lett felhasználva, amiket Raychev és társai biztosítottak. [12] A modell előállítása a Gensim[2] nevű Python könyvtárral történt.

[2] `https://radimrehurek.com/gensim/`



2.2. ábra. Néhány vektor vizualizációja

A Gensim egy nyílt forráskódú Python modul, amit szövegek különböző vektorokká történő átalakítására terveztek. Ezen belül a már korábban említett Word2Vec modellt használtam, amit elsősorban természetes nyelvfeldolgozásnál használnak, hogy a különböző szavakat vektorként ábrázolják úgy, hogy a vektorok alapján egyszerűen lehessen mérni a szavak szemantikai hasonlóságát. Jelen esetben nem természetes nyelvet dolgozunk fel, de mint ahogy a dolgozatból kiderül, ilyen módon is használható.

A modell előállításához először egy JavaScript kulcsszavakat tartalmazó fájl kerül beolvasásra, aminek a tartalmát a fentebb említett tokenizáló modul kapja meg. Ezek után a Raychev és társai által biztosított JavaScript fájlok kerülnek tokenizálásra. A folyamat a fentebb szemléltetett módon történik, több szálon párhuzamosítva. A szálak által generált fájlokat ezután a szkript összefűzi, majd létrehoz belőlük egy LineSentence objektumot, aminek az a funkciója, hogy soronként beolvassa a fájlt, és később jelzi, hogy minden sor külön mondatnak felel meg. Végül a LineSentence objektum átadásra kerül a Word2Vec modellt létrehozó metódusnak. További paraméterként megadtam, hogy minden szót vegyen figyelembe (különben az 5-nél kevesebbszer előforduló szavakat kihagyná), illetve, hogy párhuzamosítsa a műveletet. A vektorok hosszát az alapértelmezett 100-nak hagy-

tam.

A tokenek vektorizálása után minden JavaScript sorhoz el kell készíteni egy aggregált vektort, ugyanis a bemutatott modell a sorokat vektorokhoz hasonlítja. Ehhez egy igen egyszerű módszert használtam: átlagoltam az adott sorban található tokenek vektorait. Ennek a hátránya, hogy elfedi a tokenek sorrendjét, viszont éppen ez az előnye is, ugyanis így össze lehet hasonlítani olyan sorokat is, amikben ugyanazok a tokenek vannak, csak más sorrendben. Viszont a legjobb aggregáló algoritmus megtalálása további vizsgálódást igényel, amire ebben a dolgozatban nem kerül sor.

2.1.2. A sérülékenység valószínűségének számítása

Ahhoz, hogy meg lehessen jósolni, hogy a rendszer melyik sora sérülékeny, először az SSH minden sorához létre kell hozni az aggregált vektort, amiket aztán majd az analízis során a modell felhasznál. Ezután a TR-ben is ki kell számolni az aggregált vektorokat, amiket a modell összehasonlít az SSH minden sorával, majd kiszámolja a koszinusz távolságokat. A legrövidebb távolság megtalálása után a valószínűség a következő formula szerint számolódik:

$$P_{tav}(sor_{kod}) = 1 - min_e(cos(\vec{v}(sor_{kod}), \vec{v}(sor_{SSH_e})))$$

Ezt a valószínűséget viszont nem lehet közvetlenül használni, ugyanis bizonyos nyelvi szintaktikai elemek általában külön sorba íródnak (például: *else*, *}*), és így garantáltan sérülékenynek lesznek jelölve. Hasonló problémát okoznak a gyakori nyelvi minták, mint a *var myVar;* vagy a *return value;*. Csak szimplán a távolságot használva ezek is mindig meg lennének jelölve. Ezért szükség volt bizonyos szabályok bevezetésére: (i) egyszavas sorok kizárása és (ii) komplexitás szabály. Az (i) szabály a nevéből adódóan azt mondja, hogy amennyiben egy olyan sort kell vizsgálni, ami csak egy tokenből áll, akkor ahhoz nem kell vektort keresni, hanem egyszerűen nem sérülékenynek lesz jelölve. A (ii) szabály azt mondja, hogy két sor hasonlósága "többet ér", hogy ha a vizsgálandó sor komplexebb, vagyis ilyenkor a valószínűségét meg kell növelni. Ez azért szükséges, mert a komplexebb sorok nagyon specifikus SSH-beli sorokhoz hasonlítanak, ezért sokkal biztosabbak lehetünk a sérülékenységükben. A komplexitás meghatározásához a következő

formula használatos:

$$P_{komplex}(sor_{kod}) = 1 - \frac{1}{kulonbozo(sor_{kod})}$$

Ahol a kulonbozo() függvény azt mondja meg, hogy hány különböző token található az adott sorban.

Ha mindkét valószínűség, $P_{tav}(sor_{kod})$ és $P_{komplex}(sor_{kod})$ is megvan, akkor már csak az átlagot kell számolni, hogy kiderüljön a sérülékenység valószínűsége:

$$P_{seulekeny}(sor_{kod}) = \frac{P_{tav}(sor_{kod}) + P_{komplex}(sor_{kod})}{2}.$$

2.1.3. További szabályok és módszerek

A fent említett szabályokon kívül egy másikat is kipróbáltunk: ez arra volt hivatott, hogy megvizsgálja a sorok bizonyos távolságon belüli szomszédait, és azok sérülékenységének valószínűsége alapján módosítsa az aktuális sor sérülékenységének valószínűségét. Távolság alatt a vizsgált környezet méretét értjük, aminek meghatározására több módszert is alkalmaztunk: statikusat és adaptívat. Statikus méret esetén mindig előre meghatározott távolságot néztünk (1), adaptív esetén pedig az aktuális sort tartalmazó blokk egészét vizsgáltuk. Ezt a blokkot úgy határoztuk meg, hogy az adott sor körül kiszámítottunk egy sérülékeny szakaszt, amelyet a hozzá legközelebbi nem sérülékenynek számító sorok határolnak. Mindkét esetben a vizsgált területre eső sérülékenynek jelölt sorok számát vettük figyelembe úgy, hogy a kiszámolt értéket átadtuk a komplexitási szabálynál bemutatott aggregáló függvénynek (2.1.2 rész). Ezeknek a szabályoknak vizsgáltuk egy iteratív változatát is, ami több iteráción keresztül számolta a környezetben levő sérülékeny sorokat. Minden iterációban kiszámoltuk, hogy mi lenne az adott sorra a sérülékenység valószínűsége, majd ezt használtuk a következő iterációban. Végül egyik szabályt sem alkalmaztuk, mivel közel véletlenszerű eredményeket adtak.

Az előző részben (2.1.2) említett módszer úgy számolta a valószínűséget, hogy a szabályok eredményét átlagolta. Ennek finomításaként megvizsgáltam egy másik, regresszión alapuló módszert. Ennek lényege, hogy összevetjük a ténylegesen sérülékeny sorok szabályok által adott eredményeit (itt a szomszédsági szabályt is figyelembe vettük) a

ténylegesen nem sérülékeny sorok eredményeivel, és megpróbálunk arra következtetni, hogy az eredményekhez súlyozott átlag számításakor milyen együtthatók tartozzanak. Mivel az adathalmaz kiegyensúlyozatlan, tehát nem sérülékeny sorból lényegesen több van, mint sérülékenyből, ezért alul-mintavételezést kellett alkalmazni, vagyis csak annyi nem sérülékeny sor lett véletlenszerűen kiválasztva, mint amennyi sérülékeny.

Részletesebben kifejtve a regresszió a következő lépések alapján történt: először meghatározásra került, hogy pontosan melyik szabályok alkalmazása során keletkező eredményeket fogom vizsgálni. Ez, mint ahogy korábban említésre került, azt az esetet jelenti, amikor mindhárom szabályt figyelembe vettem. Ezután a szkript beolvasta az összes szükséges fájlt, beleértve a tényleges sérülékenységeket tartalmazó patch-ek előkészített fájljait is. Itt előkészítés alatt azt kell érteni, hogy minden lehetséges patch-ből egy egyszerű szöveges fájlba lettek kigyűjtve a sérülékeny sorok. A beolvasás után az adatokat egy pandas[3] DataFrame-ben tároltam az egyszerű kezelés végett. A DataFrame-ben minden sorról tároltam a három szabály által adott eredményt, illetve utólag hozzáadtam egy oszlopot. Ebbe az oszlopba 1-es került, ha a sor benne volt valamelyik patch-ben (vagyis valóban sérülékeny), különben 0-ás. Ezután az 1-est tartalmazó sorokat kiszűrtem, a 0-ás értéket tartalmazók közül pedig véletlenszerűen választottam úgy, hogy a két osztály aránya 1:1 legyen. Magát a regressziót az sklearn[4] csomag segítségével csináltam. A csomaggal gyorsan és egyszerűen lehet különböző statisztikai vizsgálatokat csinálni, elég csak a megfelelő osztály fit() metódusát hívni. Itt lineáris, illetve logisztikus regresszió is kipróbálásra került, de a jelen adathalmaz eredményeinél lényeges eltérés nem mutatkozott. Az eredmények végül egy JSON fájlban kerültek tárolásra.

Az új módszer végül nem felelt meg az eredeti célnak, viszont megmutatta, hogy a szomszédsági szabály nem hasznos a valószínűség-számítás szempontjából.

És éppen a fent említett eredmények miatt vetettük el a szomszédsági szabályt, és választottuk végül a komplexitási szabályt.

[3] `https://pandas.pydata.org`
[4] `https://scikit-learn.org/stable/`

2.1.4. Előrejelzés meghatározása

A fent kiszámolt valószínűséget már lehet arra használni, hogy az értéke szerint bizonyos kódrészek áttekintését priorizálni lehessen. Viszont a cél az volt, hogy egy olyan modell jöjjön létre, ami azokat a sorokat jelöli meg, amiknél a sérülékenység valószínűsége magas. Ezért meg kell határozni egy küszöbértéket, és azokat a sorokat megjelölni, amik valószínűsége ezen érték fölött van. Ezt a küszöbértéket empirikus módon határoztuk meg a következő módon: az adathalmazt $train$, dev, és $test$ halmazokra bontottuk (Részletesen a 3.0.1 részben). A dev halmazt felhasználva megkerestük a legjobb küszöbértéket úgy, hogy minden sorhoz meghatároztuk a $P_{serulekeny}(sor_{kod})$ értéket, és megnéztük a teljesítményértékeit (lásd 3.0.1 rész). A vizsgált küszöbértékek 0.85 és 0.99 között mozogtak 0.01 lépésközzel. Az optimalizálandó teljesítményérték függvényében tudjuk kiválasztani a megfelelő küszöbértéket a predikciókhoz.

2.2. Adathalmaz és manuális ellenőrzés

Az SSH egy publikusan elérhető JavaScript adathalmazból [4] és további adatbányászat eredményéből készült. Az adathalmaz függvényszinten tartalmazta a sérülékenységeket, ezért ki kellett gyűjteni a javító patch-eket a git commit hash értékeik alapján. A patch-eken kívül GitHub-on adatbányászást is végeztünk; több mint 500 JavaScript projekt commit-jai lettek átvizsgálva. Ehhez heurisztikus megközelítést alkalmaztunk: azokat a commit-okat néztük, amelyek szövege tartalmazta a "CVE" [5] kifejezést. Ezután a tényleges javításokat elkülönítettük a tesztektől és a dokumentáció frissítéseitől, és új patch-eket hoztunk létre. Végül az összes patch (vagyis mind az adathalmaz, mind az adatbányászat eredményeit) unióját vettük, és manuálisan filtereztük és megtisztítottuk őket. A megtisztítás azt jelenti, hogy a patch-ekből csak azok a sorok maradtak meg, amiknél a manuális ellenőrzés során meg tudtuk erősíteni, hogy sérülékenységhez tartoznak. Erre azért volt szükség, hogy minimalizálni lehessen a hamis pozitívok esélyét az SSH-ban, ezzel is javítva a teljesítményt. A 2.2 táblázat mutatja a létrehozott adathalmaz statisztikáit a manuális ellenőrzés után.

[5] `https://cve.mitre.org/cve/`

Projektek száma	91
Fájlok száma	122
Függvények száma	443
Patch-ek száma ellenőrzés előtt	614
Patch-ek száma ellenőrzés után	186
Fennmaradó patch-ek aránya	30%
Sérülékeny sorok száma	893
Projektenként átlagos sorok száma	9.8

2.2. táblázat. A felhasznált adatok statisztikái

Line_number	Chance_of_vuln	Line_content
0	0.8405269382091669	'use strict';
3	0.9127906976744187	Resolver: CallbackResolver,
6	0.9432030764397065	} = require('internal/dns/utils');

2.3. táblázat. Részlet egy konkrét csv fájlból

2.3. Az eredmények előállítása

A fent leírt folyamat alapján minden Word2Vec modell, minden szabály és minden vizsgált küszöbérték kombinációjára létrejött egy csv fájl a következő mappaszerkezetben: $modellnev/fajlnev/kuszobertek/szabaly.csv$ A csv fájlok tartalmára a 2.3 táblázat ad példát, ami egy konkrét fájlból mutat részletet. A fájl a végül használt modellben történő feldolgozás eredményét mutatja az egyik fájlon, 80%-os küszöbértékkel minden szabályt felhasználva. A Line_number oszlop mutatja, hogy a vizsgált fájl hanyadik soráról van szó, a Chance_of_vuln azt, hogy mennyi a sérülékenység valószínűsége, a Line_content oszlopban maga a sor található, míg a Reason_for_pred oszlop az ehhez legjobban hasonló sor tokenizált változatát mutatja. A fejléc hossza miatt a táblázat két részletben látható (folytatás a 2.4 táblázatban).

Abban az esetben, ha regresszióval szeretnénk foglalkozni, akkor egy speciális, 0%-os küszöbbel rendelkező fájlt kell beolvasnia a szkriptnek. Ezek a csv-k kicsit máshogy

Reason_for_pred
IdentifierName . IdentifierName (StringLiteral , StringLiteral)
IdentifierName : IdentifierName ,
, IdentifierName = IdentifierName (StringLiteral) ;

2.4. táblázat. Részlet egy konkrét csv fájlból (folytatás)

Line_number	Chance_of_vuln	Line_content
0	0.6258307894070944	'use strict';
1	0.4258664051691691	const {
3	0.5555555555555556	Resolver: CallbackResolver,

2.5. táblázat. Regressziós csv fájl részlete

Reason_for_pred
IdentifierName . IdentifierName (StringLiteral , StringLiteral)
class IdentifierName {
IdentifierName : IdentifierName ,

2.6. táblázat. Regressziós csv fájl részlete (folytatás)

néznek ki, mint a fentebb elmondottak; a felépítésüket a 2.5 táblázat mutatja. Itt a 2.3 táblázatban szereplő fájlnak látható részlete. Itt három új oszlop látható: a distance mutatja az aktuális és a Reason_for_pred oszlopban levő sor koszinusz-távolságát, a surrounding oszlop mutatja a szomszédsági szabály értékét, a complexity pedig a komplexitás szabályét. A fejléc hossza miatt a táblázat 3 részletben látható (folytatás a 2.6 és a 2.7 tánlázatokban).

A csv fájlok beolvasása után azok feldolgozása következik. Ebben a fázisban számítjuk ki a 3.1 táblázatban található teljesítménymérőket (részletesebben a 3.0.1 részben) a következő pszeudokódban látható módokon:

```
file_lines = sorok_száma(eredeti fájl)


flagged_lines = sorok_száma(csv fájl)


vuln_lines = 0
for minden patch-re
do vuln_lines += sorok_száma(patch)
```

distance	surrounding	complexity
0.877492368221283	2	2
0.7775992155075073	1	2
1.0	1	3

2.7. táblázat. Regressziós csv fájl részlete (folytatás)

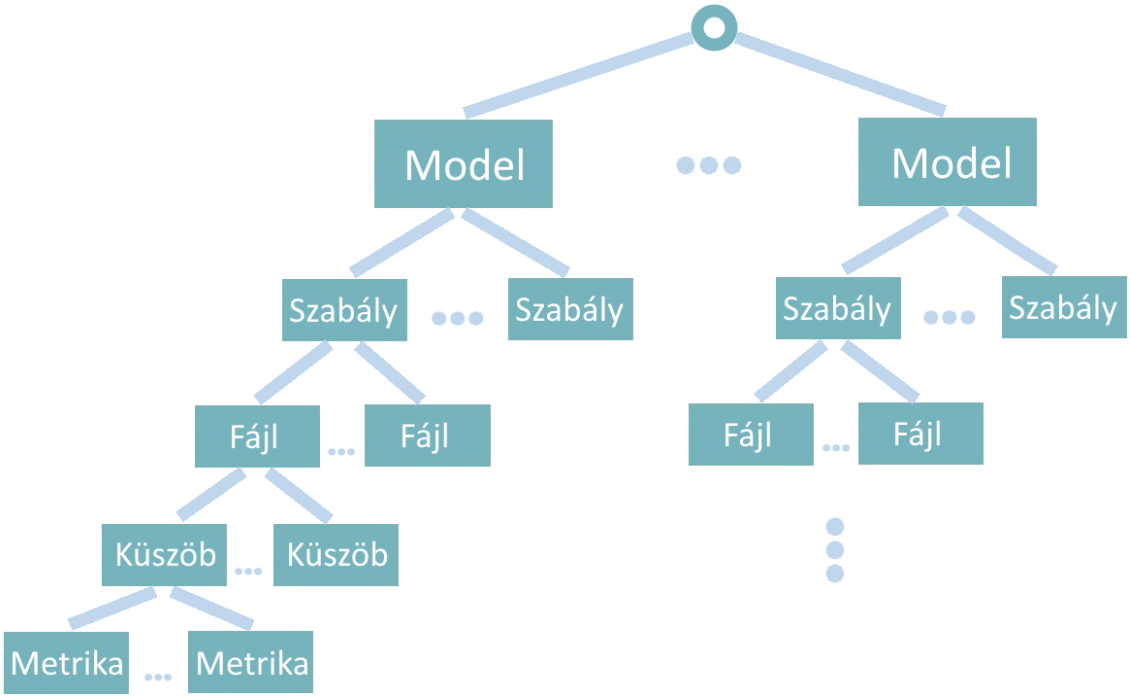
```
flagged_vuln_lines = 0
for minden patch-re
do for a patch minden sorára
do if patch sora benne van a csv-ben
then flagged_vuln_lines++

%_flagged = flagged_vuln_lines * 100 / file_lines

%_is_vuln = 0
if flagged_lines > 0
then %_is_vuln = flagged_vuln_lines * 100 / flagged_lines

%_vuln_flagged = flagged_vuln_lines * 100 / vuln_lines
```

Ezek a részeredmények a 2.3.1 részben részletesebben leírt módon tárolódnak, illetve mindegyikről készül egy JSON fájl is arra az esetre, ha külön szeretnénk őket vizsgálni.

Következő lépésként ezek a részeredmények egyesítve lettek egy közös fájlban. Ebből a fájlból le lehet olvasni minden fontos adatot, illetve megkönnyíti a különböző modellek, szabályok, és küszöbértékek összehasonlítását. A fájl felépítésére példát a 2.8 táblázat tartalmaz.

			Küszöbök		
			85%	86%	...
Modellnév	Szabály1	Összes sor			
		Megjelölt sorok			
		Sérülékeny sorok			
		Megjelölt sérülékeny sorok			
		Megjelöltek %-a			
		Sérülékenyek %-a			
		Megjelölt sérülékenyek %-a			
	Szabály2	...			
...					

2.8. táblázat. Példa az összesítő fájl felépítésére



2.3. ábra. Az adattároló felépítése

2.3.1. Az eredményt előállító szkriptről részletesebben

Az legtöbb (főleg az összegző típusú) eredményt, beleértve a 2.3 rész táblázatát és a 2.1.3 részben leírtakat, egyetlen szkript állította elő, illetve dolgozta fel. A szkript egy parancssori Python alkalmazás, melynek különböző argumentumokat is meg lehet adni. Be lehet állítani, hogy konkrétan melyik modell eredményét szeretnénk megjeleníteni, szeretnénk-e aggregálni az eredményeket, szeretnénk-e regressziót végezni, és azt is, hogy szeretnénk-e újra feldolgozni a patch-eket. Ha nem adunk meg argumentumot, akkor csak összegzi az összes csv fájl eredményét. Ehhez Python dictionary-k és listák segítségével létrehoz egy adattárolót (lényegében egy gráfot) a 2.3 ábrának megfelelően.

Ez annyiban tér el a 2.8 táblázattól, hogy a "Küszöbök" helyett a feldolgozott fájl neve van, és ez minden fájlra meg van ismételve.

Ha az aggregációt választjuk, akkor a 2.8 táblázatnak megfelelő eredmény születik, vagyis átlagolva jeleníti meg az összes fájl eredményét. A regresszió kapcsoló kiválasztása esetén a 2.1.3 részben leírtak szerint elvégzi a regressziót. Ilyenkor nem keletkezik összesítő táblázat, csak a megfelelő JSON fájl. A patch-ek újrafeldolgozását akkor érdemes választani, ha az adathalmaz új sérülékeny sorokat tartalmazó patch-el bővült.

Ilyenkor a 2.1.3 részben leírtaknak megfelelően a szkript kigyűjti a sérülékeny sorokat és egy szöveges fájlban tárolja őket.

A keresztvalidáció eredményeinek a feldolgozása úgy valósult meg, hogy az azt futtató szkript minden futás után az eredményeket egy külön mappába tette és számozással látta el. Ezek után csak futtatni kellett az eredményt előállító szkriptet, és meg kellett neki adni a futások mappáit, amiket külön modellként értelmezve dolgoz fel és jelenít meg.

3. fejezet

Eredmények

3.0.1. Esettanulmány előkészítése és teljesítménymérés

A javasolt sérülékenység-előrejelzési módszer tényleges teljesítményének értékeléséhez esettanulmányt készítettünk. A 2.2 részben leírt adathalmaz patch-eit két halmazra osztottuk 90%-10% arányban. Az előrejelzéshez szükséges küszöbérték meghatározásához a 10%-os dev halmazt használtuk (lásd: 2.1 rész). Az adatok fennmaradó 90%-ánál tízszeres keresztvalidációt alkalmaztunk, vagyis véletlenszerűen felosztottuk tízszer 80%-20% arányban $train$, $test$ halmazokra. Minden iterációban a $train$ halmaz sérülékeny sorait használtuk SSH-ként, és a predikciós módszert alkalmaztuk a JavaScript-projekt minden sorára a $test$ halmaz javításaival. Kiértékeltük az előrejelzések pontosságát az egyes iterációkban, és kiszámoltuk azok átlagát.

Mivel a javasolt módszerünk nem klasszikus gépi tanulási modell, a teljesítményét kifejezhetjük a helyesen kategorizált sérülékeny kódsorokkal (pontosság), a helyesen sebezhetőnek jelölt sorok számának az összes sérülékeny sorral való összevetésével (fedés), és az összes sérülékenynek jelzett sorok számának a TR összes sorával való összevetésével (hatékonyság). Az összegyűjtött teljesítménymérők teljes listáját a 3.1 táblázat tartalmazza.

A statisztikákat azonban elővigyázatosan kell értelmezni, mivel jelentésük kissé eltér a klasszikus gépi tanulási osztályozási rendszertől. A hatékonyság-fedés párt tartjuk a legkifejezőbb teljesítménymérőnek. Például egy 10%-60% arány azt jelentené, hogy leszűkíthetjük a vizsgálandó területet a forráskód 10%-ára, ahol a sérülékenységek 60%-át

file_lines	Sorok száma egy iterációban
flagged_lines	Sérülékenynek jelölt sorok száma
vuln_lines	Ténylegesen sérülékeny sorok száma
flagged_vuln_lines	Helyesen sérülékenynek jelölt sorok száma
%_flagged	file_lines megjelölt sorainak aránya
%_is_vuln	file_lines sérülékeny sorainak aránya
%_vuln_flagged	file_lines Helyesen sérülékenynek jelölt sorainak aránya

3.1. táblázat. A metrikák és rövid leírásuk

megtalálhatnánk.

	file_lines	flagged_lines	vuln_lines	flagged_vuln_lines
ANR	4639.4	2084.2	12.2	10.1
ACR	4639.4	576.8	12.2	7.2
MiNR	15139	6195	38	18
MiCR	11044	2326	26	17
MaNR	2686	1046	7	7
MaCR	2614	228	6	6

3.2. táblázat. 10-szeres keresztvalidáció eredményei komplexitás metrikával és nélküle

	%_flagged	%_is_vuln	%_vuln_flagged	flagged_ratio
ANR	44.49	0.48	82.11	1.84
ACR	12.43	1.24	59.01	4.74
MiNR	40.92	0.29	47.36	1.15
MiCR	21.06	0.73	65.38	3.1
MaNR	38.94	0.66	100.0	2.56
MaCR	8.72	2.63	100.0	11.46

3.3. táblázat. 10-szeres keresztvalidáció eredményei komplexitás metrikával és nélküle (folytatás)

3.0.2. A módszer előrejelzési teljesítménye

Módszerünk jelenlegi prediktív teljesítményének bemutatásához tízszeres keresztvalidációt hajtottunk végre az adathalmazon, amelynek eredményeit a 3.2 és a 3.3 táblázatok tartalmazzák. A táblázatok tartalmazzák azokat az eredményeket, amikor nem alkalmaztunk további szabályokat (NR posztfix) az eredmények szűrésére, csak a megfelelő vektorok Word2Vec távolságát, illetve azt az esetet is, amikor a komplexitási szabályt (lásd: 2.1.2 rész) alkalmaztuk (CR posztfix). Nemcsak a tíz eredmény átlagát mutatjuk be, hanem

azok minimumait és maximumait is. Ezek azok a szélső értékek, amelyekben a módszer a legjobban, illetve a legrosszabbul teljesített. Ezeket az eseteket úgy határoztuk meg, hogy kiszámítottuk a $\%\_vuln\_flagged/\%\_flagged$ arányt minden iterációban, és megkerestük a minimális és a maximális értékeket. Ez az arány a 3.3 táblázat $flagged\_ratio$ oszlopában látható. A minimum sor azt az esetet mutatja, amikor a prediktív teljesítmény a legalacsonyabb volt, vagyis a sérülékeny sorok közül sok észrevétlen maradt, de a vizsgált kód viszonylag nagy része meg lett jelölve. Az itt látható értékek a két esetre eltérő természetűek, a komplexitási szabály nélkül (*MiNR*) a módszer az átlagosnál kevesebb sort jelöl meg, ugyanakkor hiányzik a sérülékeny sorok nagy része is. Másrészt, amikor a komplexitási szabályt alkalmazzuk (*MiCR*), akkor a sérülékenységeknek az átlagosnál nagyobb százalékát találja meg, de a sorok nagyobb részét is jelöli sebezhetőnek. A maximum mező megmutatja azt az esetet, amikor a modell a legjobban teljesített: itt mindkét módszernek sikerült megtalálni az összes sérülékeny sort. A különbség közöttük a hamis pozitívok száma, ami ha nem alkalmazzuk a komplexitási szabályt (*MaNR*) jelentősen magasabb marad, mint amikor alkalmazzuk azt (*MaCR*).

Az átlagos eredmények nem annyira egyértelműek, mint az eddigiek, hiszen ezek azt mutatják, hogy a komplexitási szabály (*ACR*) csak a sérülékenységek 60%-át találja meg, míg a szabály nélkül (*ANR*) 82%-ukat sikerült megtalálni. Ennek ellenére a komplexitási szabály javít a teljesítményen, hiszen nélküle az eszköz a sorok 44,49%-át megjelöli, vele viszont mindössze 12,43%-át, tehát azért cserébe, hogy 20%-kal kevesebb sort talál meg, majdnem negyedére csökken a megjelölt sorok száma. Az eszköz használhatósága szempontjából pedig az egyik legfontosabb kérdés, hogy nem jelöl-e meg túl sok sort, így ezt határozott javulásnak nevezhetjük. A szélsőséges esetekben még látványosabbak a szabály pozitív hatásai. A legrosszabb esetben szinte feleannyi sort jelölt meg, és mégis 20%-kal több sérülékenységet talált meg, a legjobb esetben pedig akárcsak a szabály nélküli verzió képes volt megtalálni az összes sérülékenységet, de 30%-kal kevesebb sort jelölt meg. Egyértelműen kijelenthetjük, hogy a komplexitási szabály szükséges ahhoz, hogy a lehető legjobb előrejelzésekre legyük képesek.

A statisztikák habár ígéretesek, nem hagyhatjuk figyelmen kívül a jelentős szórásukat, különösen a szélsőséges esetekben, aminek oka valószínűleg az általunk használt adathalmaz elégtelen mérete, ami csökkenti az esélyét annak, hogy találunk a sérülékeny sorhoz

megfelelő közelségű példát a TR-ben. Az adatok hiánya okozhatja a magas hamis-pozitív rátát, hiszen kisebb elfogadási küszöböt kell használnunk miatta, ami kevésbé biztos előjelzésekhez vezet.

3.1. Egy példa az eredményekből, és a hozzá tartozó bizonyíték

A módszer működésének demonstrálásához választottunk egy konkrét példát az esettanulmányi eredményeink közül a hozzá tartozó magyarázattal együtt. A 3.1 ábra mutatja a teszt halmazbeli sort, amit mi helyesen sérülékenynek jelöltünk. Alatta látható a referenciaként szolgáló javított sor. A 3.2 ábrán látható a helyesen jelölt sor magyarázata. A javítását is mutatjuk, ami egy patch-ben található.

```
-       tail = normalizeString(path.slice(rootEnd), !isAbsolute, '\\');
+       tail = normalizeString(path.slice(rootEnd), !isAbsolute, '\\', isPathSeparator);
```

3.1. ábra. Példa egy jól előrejelzett sorra és a tényleges javítására

```
-     related = path.parse(path.join(__dirname, '/../assets', relative))
+     related = decodeURIComponent(path.join(__dirname, '/../assets/styles.css'))
```

3.2. ábra. A bizonyíték az előrejelzésre és a tényleges javítása

Ahogy láthatjuk, mindkét példában a sérülékenység elérési útvonalakkal kapcsolatos[1]. Annak ellenére, hogy valamelyest hasonlítanak egymásra, például van egy *path* művelet egy függvényhíváson belül, ami tartalmaz egy relatív útvonalat kifejező szöveges literált, nehezen lehetne őket eddig ismert másolás detektálókkal kiszűrni. Ettől függetlenül, az eszközünk egy tokenizálót futtat a kódon, aminek eredményét a 3.1 és a 3.2 listázásban láthatunk.

```
1 IdentifierName = IdentifierName ( IdentifierName . IdentifierName
      ( IdentifierName ) , ! IdentifierName , StringLiteral ) ;
```

3.1. Listing. Példa egy előrejelzett sor tokenizált változatára

[1] https://owasp.org/www-community/attacks/Path_Traversal

```
1 IdentifierName = IdentifierName . IdentifierName ( IdentifierName
      . IdentifierName ( IdentifierName , StringLiteral ,
      IdentifierName ) ) ;
```

3.2. Listing. Példa a sor bizonyítékának tokenizált változatára

A tokenizáció után még láthatóak különbségek, de a Word2Vec vektorjaik átlaga hasonló lesz, hiszen rengeteg hasonló típus található bennük. Ennek köszönhetően a találati valószínűség magas lesz($P_{tav}(sor_{kod}) = 0.965$). Figyelembe véve a tényt, hogy a sorok nem triviálisak, sok különböző típus található bennük, a komplexitás értékük is magas lesz ($P_{komplex}(sor_{kod}) = 0.889$). A kettő átlaga nagyobb lesz mint 0,92, tehát a módszerünk helyesen sérülékenynek fogja jelölni a 3.1 ábrán látható sort, amit a 3.2 ábrán látható magyarázattal lát el, és támaszt alá.

3.2. Az eredmények összefoglalása

Ahogy az előző szakaszban demonstráltuk, mind a tízszeres keresztvalidációval, mind a konkrét példával, a módszerünk képes értékes eredményeket produkálni, nem csak abban az értelemben, hogy valós sérülékenységeket észlel, hanem abban is, hogy pontos előrejelzéseket készít. A komplexitási szabály hatékonynak bizonyult a hibásan megjelölt sorok számának csökkentésében, míg a helyesek közül nem vetett el sokat. Az adathalmazunk kiegyensúlyozatlanságának ellenére a módszer megtalálja a legtöbb sérülékeny sort, miközben kevesebb mint 13%-át jelöli meg az elérhető soroknak. Elméletileg ez azt is jelentheti, hogy elegendő lenne a kódunk 10%-át újra átnézni, és akár a sérülékenységek 60%-át megtalálhatnánk ezt a módszert használva. Hasonló értékeket láthatunk korábbi munkákban [13] fájl szinten, és úgy gondoljuk, hogy ugyanezt az arányt elérni sor szinten komoly eredmény. Ennek ellenére a módszerünk további finomításokra szorul, hogy egy valós környezetben használható legyen.

Tapasztalataink alapján a módszer egymástól viszonylag független sorokat jelöl meg, ami nehezíti a helyesség ellenőrzését. Továbbá a megjelölt sorok száma továbbra is túl magas a valóban sérülékeny sorok számához képest. Ezt a problémát új, szofisztikáltabb szabályok bevezetésével, illetve a mostani szabályaink számításának finomhangolásával tervezzük enyhíteni. Például a komplexitási szabály esetében a függvénnyel, amit a végső

eredmény létrehozására használunk, egyelőre nem kísérleteztünk, így feltehetőleg van lehetőség javulást elérni.

4. fejezet

Konklúzió

Rengeteg kutatás folyik sérülékenység-előrejelző modellek témájában, hiszen ezek rendkívül keresettek. A túlnyomó többsége a legkorszerűbb eszközöknek gépi tanulást alkalmaz ilyen modellek létrehozására. Ebben a dolgozatban egy új módszert mutattunk be, ami két szempontból múlja felül az eddigieket: megmagyarázhatóság és részletesség.

A mi alacsony hardverigényű megoldásunk manuálisan kiértékelt sérülékenység javításokra, és a forrás sorok Word2Vec vektorainak hasonlóságára épül. Létrehoztunk egy referencia halmazt manuálisan kiértékelt sérülékeny kódsorokból, és terveztünk egy algoritmust, hogy megtaláljuk, egy vizsgált rendszerben azokat a kódsorokat, amelyek legjobban hasonlítanak a kódbázisban találhatóakra. Amikor a hasonlóság az analizált sor és a referencia halmazbeli sor egy előre meghatározott küszöb fölött van és több más szabálynak is eleget tesz, a sort sérülékenynek jelöljük. A módszer egyszerű, de rengeteg kedvező tulajdonsága van, amelyekkel többségével más korszerű megközelítések nem rendelkeznek.

Először is a módszerünk nem hagyatkozik költséges gépi/mély tanulási módszerekre, amelyeknek hatalmas tanuló halmazokra van szüksége, hogy értékes eredményeket tudjanak produkálni. Az egyetlen lépés, amikor lényeges modell készítés történik, a Word2Vec vektorok létrehozása, amelyet elég egyszer végrehajtani. Másodszor, az előrejelzési módszer eredményei sor szintűek függvény- vagy fájlszint helyett. Ez pedig lehetővé teszi, hogy a fejlesztők gyorsabban döntést tudjanak hozni az előrejelzés helyességével kapcsolatban, ezáltal a lehető leggyorsabban reagálhassanak a megtalált sérülékenységekre. Harmadszor, hogy tovább segítsük a döntés meghozását az eredménnyel kapcsolatban, a

módszerünk visszatér egy magyarázattal. Ez a leghasonlóbb sor a referencia halmazból, ami a fejlesztők segítségére lehet az adott sérülékenység javításának során. Negyedszer, egyszerű javítani a modell teljesítményét használat közben. Ehhez elég az újonnan sérülékenynek érzékelt sorokat hozzáadni a referencia halmazhoz, ezen felül nincs szükség tanítási lépésekre.

A módszerünk messze nem hibátlan, viszont egy ígéretes lépés egy könnyen értelmezhető sérülékenység-előrejelző modell felé. Demonstráltuk a használhatóságát a 91 projekt 186 valós és ellenőrzött JavaScript sérülékenységén végzett esettanulmánnyal, amely során a módszer képes volt átlagosan 60%-át felismerni az ismert sérülékenységeknek, miközben a kódbázis mindössze 10%-át jelölte meg. Néhány esetben pedig a sorok 8,72%-át megjelölve az összes ismert hibát megtalálta. Ezek az eredmények már önmagukban meggyőzőek, de természetesen van lehetőség további fejlesztésre. A jövőbeli terveink része további szabályokat definiálni, illetve a Word2Vec vektorokat más módon aggregálni, például a Doc2Vec módszer alkalmazásával.

Függelék

A dolgozatban bemutatott program forráskódja megtalálható a következő helyen: `https://github.com/vandorn99/vulnerability-prediction`

Nyilatkozat

Alulírott Vándor Norbert Rudolf programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2021. június 9.

Vándor Norbert Rudolf

aláírás

Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezetőmnek Dr. Hegedűs Péternek, és Antal Gábornak, akik szakmai támogatásukkal segítették a projekt sikerét.

Ez a szakdolgozat a SETIT projekt (2018-1.2.1-NKP-2018-00004)[1] keretében készült.

[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] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[2] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[3] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[4] Rudolf Ferenc, Péter Hegedűs, Péter Gyimesi, Gábor Antal, Dénes Bán, and Tibor Gyimóthy. Challenging Machine Learning Algorithms in Predicting Vulnerable JavaScript Functions. In *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 8–14. IEEE Press, 2019.

[5] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 171–180. IEEE, 2012.

[6] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. Enabling the Continous Analysis of Security Vulnerabilities with VulData7. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 56–61, 2018.

[7] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie A. Williams. Challenges with applying vulnerability prediction models. In *HotSoS*, 2015.

[8] Balázs Mosolygó, Norbert Vándor, Gábor Antal, Péter Hegedűs, and Rudolf Ferenc. Towards a prototype based explainable javascript vulnerability prediction model. In *2021 International Conference on Code Quality (ICCQ)*, pages 15–25, 2021.

[9] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 529–540, 01 2007.

[10] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto. Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 244–255, 2016.

[11] L. Pascarella, F. Palomba, and A. Bacchelli. Re-evaluating method-level bug prediction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 592–601, 2018.

[12] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. *ACM SIGPLAN Notices*, 51(1):761–774, 2016.

[13] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, November 2011.

[14] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317. ACM, 2008.

[15] Miltiadis Siavvas, Dionisis Kehagias, and Dimitrios Tzovaras. A preliminary study on the relationship among software metrics and specific vulnerability types. In *2017 International Conference on Computational Science and Computational Intelligence – Symposium on Software Engineering (CSCI-ISSE)*, 12 2017.

[16] C Theisen, R Krishna, and L Williams. Strengthening the evidence that attack surfaces can be approximated with stack traces. In *North Carolina State University*

Department of Computer Science TR2015-10, submitted to International Conference on Software Testing, Verification, and Validation (ICST) 2016, 2015.

[17] Zhe Yu, Christopher Theisen, Hyunwoo Sohn, Laurie Williams, and Tim Menzies. Cost-aware vulnerability prediction: the HARMLESS approach. *CoRR*, abs/1803.06545, 2018.

[18] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference onSoftware Testing, Verification and Validation (ICST)*, pages 421–428. IEEE, 2010.