Odkrivanje ranljivosti v programski kodi z metodami strojnega učenja

Mitja Hrovatič

Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru
Email: mitja.hrovatic@student.um.si

Uporaba programske opreme je del našega vsakdana, kjer se pogosto izpostavlja sama uporabnost programov, pri čemer je varnost zapostavljena. Razvijalci tako želijo čim hitreje dostaviti uporabnikom željene funkcionalnosti, kjer se odrine na stran zagotavljanje varnosti v samem produktu. Kot vemo, je danes vse več zlorab, vdorov in drugih varnostnih incidentov, ki se pogosto dogajajo zaradi varnostnih ranljivosti same programske opreme. V raziskavi predstavimo, kako pristopiti k odkrivanju ranljivosti v programski kodi že v fazi pisanja programske kode. V sklopu raziskave smo se osredotočili na moderni pristop z uporabo metod strojnega učenja, kjer smo razvili algoritem za odkrivanje ranljivosti v programski kodi za programsko opremo napisano v programskem jeziku Python. Pridobili smo podatke iz resničnih odprtokodnih projektov iz platforme Github in zbirk OSV. Pridobljene podatke smo procesirali z metodami obdelave naravnega jezika, kar nam je omogočilo upoštevanje konteksta v programski kodi. Napovedovanje in klasifikacijo ranljivosti smo izvedli z modeli LSTM in GRU, kjer smo pridobili rezultate evalvacije po metrikah točnosti, natančnosti, priklica in F1.

Ključne besede: Detekcija ranljivosti; Strojno učenje; Procesiranje programske kode z metodami obdelave naravnega jezika; Napovedovanje ranljivosti; Model LSTM in GRU;

1. UVOD

Danes razvijalci pri razvoju programske opreme želijo odkriti in odpraviti napake, hrošče, ranljivosti in ostale vrste pomanjkljive programske kode, v največji meri že v času razvojnega procesa. Med vsemi vrstami pomanjkljive kode, imajo varnostne pomanjkljivosti najvišjo prioriteto [1]. Z razvojem vse več programske opreme se povečuje tudi število ranljivosti, kjer je inštitut NIST, ki vodi svetovno bazo znanih ranljivosti CVE (angl. Common Vulnerabilities and Explosures), v letu 2021 glede na predhodne leto zabeležil 10% povečanje novih znanih ranljivosti in v celotni zbirki do leta 2022 beleži že več kot 175 tisoč znanih ranljivosti [2]. Raziskave [3, 4, 5] so pokazale, da lahko z uporabo orodij za varnostno analizo programske kode, pomagamo razvijalcem k učinkovitemu odpravljanju varnostnih pomanjkljivosti. Poleg tega je detekcija ranljivosti signifikantno bolj zahtevna kot odkrivanje osnovnih pomanjkljivosti v programski kodi [6]. Za odkrivanje ranljive kode se uporablja različne tehnike, ki so vezane na faze v razvoju programske opreme, in sicer najbolj uporabljene so:

- SAST (angl. Static Application Security Testing), kjer se izvaja semantična analiza programske kode za detekcijo ranljivosti,
- DAST (angl. Dynamic Application Security Testing), kjer želimo z naborom testnih primerov odkriti morebitne ranljivosti, in
- RASP (angl. Runtime Application Self-Protection), kjer poteka analiza v času izvajanja aplikacije, in sicer se analizira vnose v delujoč sistem in spremlja odzive sistema.

Vse naštete tehnike tradicionalno temeljijo na pravilih (angl. rule–based) za odkrivanje vzorcev in detekcijo ranljivosti [5]. Takšen način detekcije ranljivosti prinaša veliko napačno pozitivnih (angl. false positive) odkritih ranljivosti, saj pravila pri odločanju ne zajemajo konteksta izvorne kode [1]. Z razvojem metod umetne inteligence na podlagi učenja (angl. learning–based), so raziskave [7, 8, 9, 10, 11] pokazale, da lahko z modeli strojnega učenja bolj učinkovito detektiramo ranljivosti, poleg tega lahko z metodami obdelave naravnega jezika in predprocesiranja podatkov dosežemo, da zajamemo kontekst programske kode v katerem se nahaja potencialna ranljivost.

2. RAZISKOVALNA METODA

V sklopu raziskave smo izvedli raziskovalno metodo eksperiment, kjer smo razvili algoritem za statično analizo programske kode z metodami strojnega učenja za odkrivanje ranljivosti v programskem jeziku Python. V nadaljevanju podrobno predstavimo proces zajema podatkov, predprocesiranje, izbiro in učenje modelov strojnega učenja ter evalvacijo pridobljenih rezultatov.

2.1. Pridobivanje podatkov

Modeli stojnega učenja so močno odvisni od karakteristik in dovoljšne količine kvalitetnih podatkov, ki se uporabljajo v procesu učenja modelov. Pri razvoju programske opreme se pogosto uporablja sisteme za upravljanje in verzioniranje programske kode. Za namene raziskave smo pridobili podatke in izvorno kodo iz oblačne platforme Github, ki ponuja veliko zbirko odprtokodnih projektov.

V prvi fazi smo izvedli pridobivanje podatkov iz javno dostopnih repozitorijev, preko vmesnikov API (angl. application programming interfaces), in sicer OSV in Github API. OSV (angl. open source vulnerability database) je zbirka zapisov o znanih ranljivostih iz različnih baz ranljivosti, v standardiziranem formatu. Zbirka vsebuje trenutno dobrih 28000 zapisov ranljivosti za 16 različnih sistemov iz 6 podatkovnih zbirk. Pobudnik za izgradnjo enotne zbirke ranljivosti je korporacija Google, ki tudi skrbi za ažurnost, dopolnjevanje podatkov in pridobivanje novih zbirk, ki podprejo OSV, ter infrastrukturo, ki je potrebna za delovanje. Podatki so formulirani v formatu OSV, s katerim so avtorji želeli poenotiti in standardizirati format zapisa in predstavitve ranljivosti, in ga je možno pridobiti v obliki zapisa JSON objektov, kot je prikazano na sliki 1. Za naš raziskovalni problem so bili relevantni metapodatki "ecosystem", "aliases" in "references", kjer smo pridobili podatke o vrsti sistema, tipu ranljivosti ter povezavo do različice izvorne kode (angl. commit), v kateri se je ranljivost pojavila ter spremembe, ki so bile potrebne za odpravo ranljivosti. Slednje povezave smo v nadaljevanju uporabili za pridobivanje (angl. scraping) podatkov iz sistema Github. Za rudarjenje podatkov iz sistema Github smo uporabili knjižnico "PyDriller", ki nam je omogočila pridobivanje podatkov o repozitorijih in izvorni kodi.

Za programski jezik Python smo iz zbirke OSV pridobili 2079 zapisov s potrebnimi metapodatki za nadaljnjo uporabo, na 557 različnih repozitorijih. Za ekstrakcijo primerov ranljivosti iz programske kode pridobljene iz repozitorijev, smo uporabili sistem za verzioniranje programske kode GIT, ki nam je omogočil primerjavo sprememb (angl. "git diff") ob stanju verzije (angl. "commit") projekta. Tako smo za ranljivo programsko kodo vzeli predhodno stanje verzije, novo stanje smo obravnavali kot odpravljeno ranljivost oziroma ne ranljivo programsko kodo.

Pridobljeno programsko kodo smo razvrstili v pet kategorij, prikazano v tabeli 1, ki so bile zastopane v podatkih in smo za njih pridobili dovoljšne število podatkov. V nadaljevanju smo modele strojnega učenja učili po posamezni kategoriji, kar nam je nato pri napovedovanju omogočilo klasificiranje ranljivosti po kategorijah.

```
{
        "id": string,
        "modified": string,
        "published": string,
        "withdrawn": string,
        "aliases": [ string ],
        "related": [ string ],
        "package": {
                "ecosystem": string,
                "name": string,
                "purl": string,
        },
        "summary": string,
        "details": string,
        "affects": [ {
                "ranges": [ {
                        "type": string,
                        "repo": string,
                        "introduced": string,
                        "fixed": string
                } ],
                "versions": [ string ]
        } ],
        "references": [ {
                "type": string,
                "url": string
        } ],
        "ecosystem_specific": { see spec },
        "database_specific": { see spec },
}
```

SLIKA 1: OSV shema

2.2. Predprocesiranje

Surove podatke smo v nadaljevanju prečistili, faktorizirali in jih pripravili za učenje modelov strojnega učenja. Pridobljeno izvorno kodo smo obravnavali kot tekst oziroma naravni jezik, katerega smo želeli obravnavati v kontekstu, zato smo predstavitev programske kode postavili v obliko žetonov (angl. tokens). Programski jezik Python uporablja interpreter v izvajalnem okolju, kjer se ob izvedbi interpretacije izvede leksikalna, sintaktična in semantična analiza programske kode. Tako

TABELA 1: Število pridobljenih podatkov po kategorijah ranljivosti

Vrsta ranljivosti	Število repozitorijev	Št. stanj verzije (angl. commits)	LOC – Lines of code
SQL injection	326	828	93571
Command injection	44	94	22433
XSS	87	354	36051
Remote code execution	39	52	14846
XSRF	61	139	27433

že sam programski jezik vsebuje zbirko žetonov, ki jih potrebuje za interpretacijo jezika.

Za izluščitev programskih konstruktov, kot so programski stavki ("if", "for", "class", "def", idr.), operatorji, spremenljivke, komentarji, smo uporabili vgrajen Python tokenizer. Da smo pridobljene žetone postavili v kontekst, smo uporabili metode NLP (angl. natural language processing), kjer smo uporabili model Word2Vector, s katerim smo zgradili vektorje, ki predstavljajo kontekstno povezane konstrukte. V nadaljevanju smo vektorje uporabili pri klasifikaciji ranljive in ne ranljive kode v modelu LSTM. Kot ranljivo ali ne ranljivo kodo smo nato obravnavali celotno kontekstno podobno območje, ki je bilo vezano na izbrani žeton.

2.3. Model Word2Vector

Programski konstrukti v programske jeziku so definirane oblike zapisa (kot so pogoji, zanke, metode, objekti, idr.), in jih uporabljamo za vzpostavitev logičnega postopka. Sami programski konstrukti, kot posamezni stavki ali samostojni ukazi, ne predstavljajo ranljivosti, če jih postavimo v širši kontekst same logike pa zaporedje izvajanja konstruktov lahko privede do ranljivosti. Za predstavitev programskih konstruktov v logičnem kontekstu, smo uporabili model za obdelavo naravnega jezika Word2Vector. Model deluje tako, da za vsak unikaten programski konstruktu, zgradi vektor za preostale programske konstrukte, kjer se izračuna podobnost, glede na mesto konstrukta v besedilu. Za učenje modela potrebujemo večjo količino teksta, da zgradimo korpus, ki predstavlja bazo znanja o širšem področju izbranega problema. V našem primeru smo pridobili večje repozitorije iz platforme Github ("tensorflow", "numpy", "ansible", "keras", idr.), ki vsebujejo programski jezik Python, in imajo skupaj nekaj več kot 73 milijonov žetonov. V nadaljevanju smo naučili model na pridobljenem korpusu žetonov, kar nam v nadaljevanju zagotovi odkrivanje kontekstno podobnih vzorcev programske kode, in nam nato omogoča klasifikacijo programske kode, kot ranljivo ali ne ranljivo z modelom LSTM in GRU.

2.4. Model LSTM in model GRU

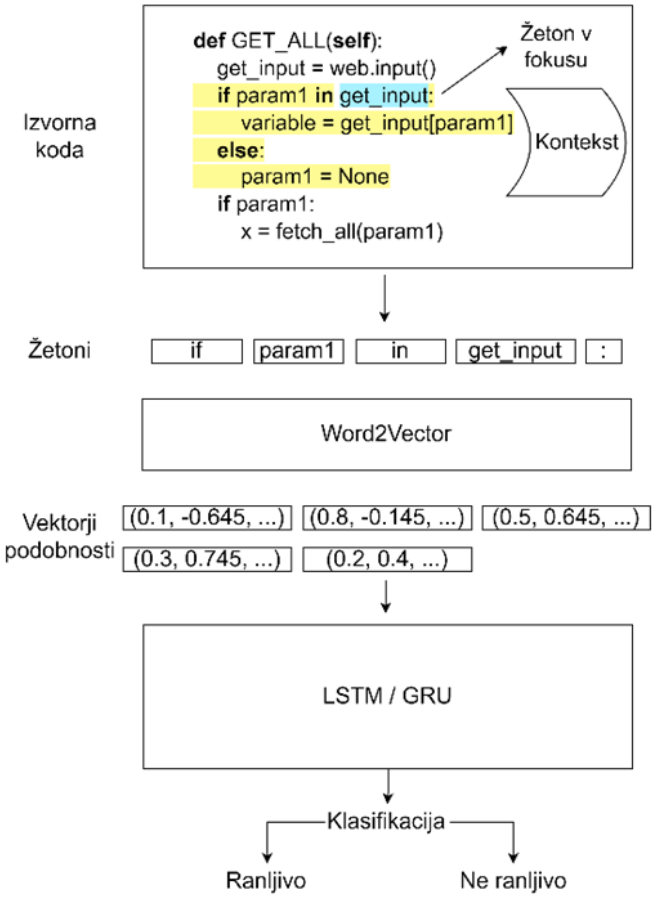
Za napovedni model smo izbrali model LSTM in model GRU. V raziskavah sta se izkazala kot najbolj primerna za reševanje problemov po pristopu z metodami obdelave naravnega jezika. Spadata v skupino modelov globokega učenja (angl. deep learning), kjer se uporabljajo več nivojske nevronske mreže. Za napovedovanje se uporabljajo uteži in več nivojev (angl. layers) nevronov. Razlika med modeloma je v številu vhodov in izhodov za gradnjo napovednega modela, in sicer model GRU je enostavnejši, oba pa prejmeta enake vhodne parametre. Za učenje modelov smo podatkovno množico razdelili po kategorijah ranljivosti, in posamezno kategorijo na 70% učne množice, 15% testne množice in 15% validacijske množice. Podatki so sestavljeni iz dveh klasifikacijskih razredov, in sicer ranljivi in ne ranljivi razred. Razreda sta v podatkih neuravnoteženo zastopana, saj imamo veliko več primerov ne ranljive kode, kot ranljive kode. Zato smo izračunali in postavili uteži, z knjižnico "sklearn", ki smo jih nato uporabili v hiperparametrih modelov. Privzeto smo določili parametre obeh modelov, in sicer število nevronov na 100, velikost svežnja (angl. batch size) na 128, osip (angl. dropout) na do 20%. Na učni in testni množici smo nato izvedli izdelavo in optimizacijo modela. Na validacijski množici smo izvedli evalvacijo naučenih modelov.

3. REZULTATI IN EVALVACIJA

Za napovedovanje ranljivosti smo vzpostavili proces, ki je sestavljen iz treh osnovnih korakov, kot je prikazano na sliki 2, in sicer:

1. Pridobivanje izvorne kode,
2. Obdelava izvorne kode z modelom Word2Vector za predstavitev, v vektorski obliki,
3. Napovedovanje ranljivosti z naučenima modeloma LSTM in GRU.

Za ocenjevanje uspešnosti izbrane metode in modelov, smo za pridobljene rezultate napovednih modelov izračunali metrike F1, točnost, natančnost in priklic. Primere ranljive kode smo obravnavali kot pozitivne (angl. positive) in primere čiste ne ranljive kode kot negativne (angl. negative). Pri napovedovanju je tako lahko bila čista koda napačno klasificirana kot ranljiva (angl. FP – false positive), in obratno ranljiva koda kot čista (angl. FN – false negative), resnično ranljiva koda je bila obravnavana kot pravilno pozitivna (angl. TP – true positive), in resnično čista ne ranljiva koda, kot pravilno negativna (angl. TN – true negative).



SLIKA 2: Proces napovedi ranljivosti

Z izračunom metrike **točnosti**, tako pridobimo razmerje pravilno napovedanih primerov v primerjavi z vsemi napovedanimi primeri.

Z metriko **natančnosti** smo želeli pokazati, koliko je pravilno klasificiranih primerov znotraj pozitivnega razred (ranljiva koda), glede na vse pravilno pozitivno ranljivo in napačno pozitivno čisto ne ranljivo kodo.

Z metriko **priklica** smo ugotovili, kakšno je razmerje med pravilno klasificirano ranljivo kodo, glede na pravilno in napačno klasificirano ranljivo kodo.

Z metriko **F1** smo izračunali harmonično povprečje, ki je izpeljano iz metrik natančnosti in priklica. Z metriko lahko tako pridobimo podatek o napačno pozitivnih in napačno negativnih primerih. V idealnih pogojih želimo doseči vrednost napačnih primerov 0 oziroma 100% F1 metrike, kar pomeni, da bi bili vsi primeri pravilno klasificirani kot pravilno ranljivi ali ne ranljivi. V tej raziskavi smo predvideli, da bi za učinkovito uporabo v praksi zadostovala 85% uspešnost.

$$\text{Točnost} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\text{Natančnost} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Priklic} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1–score} = \frac{2 * \text{Natančnost} * \text{Priklic}}{\text{Natančnost} + \text{Priklic}}$$

Pri napovedovanju ranljivosti smo naučili LSTM in GRU modela po posameznih kategorijah ranljivosti, ki smo jih pridobili iz podatkov. Tako smo pridobili rezultate evalvacije v tabeli 2.

3.1. SQL injection

Za napovedovanje ranljivosti z vrivanjem SQL stavkov, smo pridobili največje število primerov za učenje ranljivosti, in sicer 93571 vrstic kode (LOC) v katerih so zastopane te oblike ranljivosti. Primer kako izgleda ranljivost je prikazan na sliki 3. Za napovedovanje te vrste ranljivosti smo dosegli z modelom LSTM točnost 91,7%, z modelom GRU pa točnost 90,9%. Uspešnost modelov pri napačnih klasifikacijah, smo po metriki F1 dosegli z modelom LSTM 80,5%, z modelom GRU pa 77,1%.

```
@dataclass()
class Visitor:
    ip_address: str
    user_agent: str
    referrer: str
    full_path: str
    visit_time: pytz

    def on_save(self):
        connection = get_connection()
        cursor = connection.cursor()
        cursor.execute(
            f"insert into visitors (ip_address, user_agent, referrer, full_path, visit_time)
values ('{self.ip_address}', '{self.user_agent}', '{self.referrer}', '{self.full_path}',
'{self.visit_time}');")
        connection.commit()
        connection.close()
        return 0
```

SLIKA 3: Primer ranljivosti z vrivanjem SQL stavkov

3.2. Command injection

Za ranljivost vrivanja ukazov na nivoju operacijskega sistema, smo pridobili 22433 vrstic kode (LOC), kar je relativno majhen vzorec za posploševanje. Na sliki 4 je prikazan primer ranljivosti vrivanja ukazov. Rezultati metrik so pokazali, da je točnost modela LSTM 97,4%, modela GRU pa 97,5%. Dosežena je bila tudi zelo nizka vrednost napačno klasificiranih primerov, in sicer po metriki F1 za model LSTM 90,5%, za model GRU pa 91,7%.

TABELA 2: Rezultati evalvacije napovednih modelov LSTM in GRU po kategorijah ranljivosti

Vrsta ranljivosti	Točnost (LSTM/GRU)	Natančnost (LSTM/GRU)	Priklic (LSTM/GRU)	F1 (LSTM/GRU)
SQL injection	91,7% / 90,9%	82,2% / 80,1%	78,8% / 74,1%	80,5% / 77,1%
Command injection	97,4% / 97,5%	94,2% / 95,9%	86,8% / 87,4%	90,5% / 91,7%
XSS	96,9% / 97,6%	92,1% / 92,7%	79,8% / 80,7%	85,9% / 86,7%
Remote code execution	98,5% / 98,5%	96,5% / 97,0%	81,3% / 79,2%	88,1% / 88,1%
XSRF	97,3% / 97,5%	93,0% / 91,4%	84,4% / 83,8%	89,7% / 87,6%
Skupaj povprečje:	96,4% / 96,4%	91,6% / 91,4%	82,2% / 81,0%	86,9% / 86,2%

```
p = subprocess.Popen("ping -c1 {}".format(self.remoteIP), shell=True)
p.wait()
time.sleep(0.1)
```

SLIKA 4: Primer ranljivosti z vrivanjem ukazov

3.3. XSS (Cross–site scripting)

Za ranljivost vrivanje programske kode, ki se izvede na strani odjemalca, smo pridobili 36051 vrstic kode (LOC). Primer ranljivosti je prikazan na sliki 5. Za to vrsto ranljivosti smo dosegli točnost z modelom LSTM 96,9%, z modelom GRU pa 97,6%. Pri napovedovanju manj napačnih primerov je bil pri tej vrsti ranljivosti bolj uspešen model GRU, in sicer po metriki F1 z rezultatom 86,7%. Model LSTM je dosegel rezultat 85,9%.

```
@view_config(route_name='search', renderer='templates/search.pt')
def search(request):
    return {'query': request.params.get('q', '')}
    # http://localhost:6543/search?q=%3Cscript%3Ealert(123)%3C/script%3E
```

SLIKA 5: Primer ranljivosti z vrivanjem programske kode na strani odjemalca

3.4. Remote code execution

Za ranljivost izvajanja ukazov na oddaljeni napravi, smo pridobili 14846 vrstic kode (LOC). Primer ranljivosti je prikazan na sliki 6, kjer vidimo, da se uporablja parameter "shell=True", ki omogoči izvajanje ukazov na korenskem nivoju (angl. root). Za to vrsto ranljivosti smo dosegli klasifikacijsko točnost za model LSTM in model GRU 98,5%. Rezultat klasifikacija napačno klasificiranih primerov po metriki F1 smo pri dobili za oba modela vrednost 88,1%.

```
def get_allconfiguredwifi():
    """
    nmcli con | grep 802-11-wireless
    """
    ps = subprocess.Popen('nmcli -t -f NAME,TYPE conn | grep 802-11-wireless', shell=True,stdout=subprocess.PIPE).communicate()[0]
    wifirows = ps.split('\n')
    wifi = []
    for row in wifirows:
        name = row.split(':')
        print(name)
        wifi.append(name[0])
    return wifi
```

SLIKA 6: Primer ranljivosti izvajanja ukazov na oddaljeni napravi

3.5. XSRF (Cross-site request forgery)

XSRF ranljivost se izkorišča na nivoju HTTP protokola in delovanju brskalnikov, ki avtomatsko pripenjajo zahtevkom piškotke, kjer se želi prestrezati in ponarejati uporabniške seje, avtorizacijske žetone in podobno, z namenom izvajanja uporabniških akcij in pridobitvijo popolnega dostopa nad uporabniškim računom. Za preprečevanje takšne ranljivosti, poleg vhodnih parametrov dodajamo unikaten XSRF žeton, ki se nato validira na zalednemu sistemu. Primer ranljivosti je prikazan na sliki 7. Za to vrsto ranljivosti smo pridobili 27433 vrstic kode (LOC). Rezultat klasifikacijske točnosti za to vrsto ranljivosti je za model LSTM 97,3%, za model GRU pa 97,5%. Pri napovedovanju napačno klasificiranih primerov smo po metriki F1 pridobili rezultate za model LSTM 89,7%, za model GRU pa 87,6%.

```
GET /email/change?email=pwned@evil-user.net HTTP/1.1
Host: vulnerable-website.com
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

SLIKA 7: Primer ranljivosti s ponarejanjem zahtevkov

4. DISKUSIJA

V raziskavi smo se osredotočili na detekcijo in klasifikacijo ranljivosti z uporabo metod za statično analizo izvorne kode na podlagi metod strojnega učenja. Za večjo zunanjo veljavnost eksperimenta smo v raziskavi uporabljali podatke pridobljene iz platforme Github, kjer so tudi javno dostopni repozitoriji aplikacij in druge programske opreme, ki se uporablja v resničnih primerih, kar nam omogoča učenje modelov strojnega učenja na resničnih primerih. Slabost tako pridobljenih primerov, je problem kakovosti le-teh, saj je od posameznih razvijalcev odvisna kakovost napisane programske kode, ne glede nato ali je ranljiva ali ne. Tako smo na podlagi pridobljenih podatkov naslovili prvo raziskovalno vprašanje.

4.1. RV1: Kakšen vektor napadov odkrivamo s statično analizo programske kode temelječo na strojnem učenju?

Z zbranimi podatki smo pridobili pet različnih kategorij ranljivosti, ki so najbolj zastopane in izpostavljene v programskem jeziku Python, in sicer:

- vrivanje SQL stavkov (angl. SQL injection),
- vrivanje ukazov za izvajanje v operacijskem sistemu (angl. Command injection),
- vrivanje programske kode, ki se izvede na strani odjemalca (angl. XSS – Cross Site Scripting),
- izvajanja ukazov na oddaljeni napravi (angl. remote code execution),
- ponarejanje zahtevkov (angl. XSFR - Cross Site Request Forgery).

Vse naštete kategorije so široko odkrite in naslovljene v zbirkah CVE in pri organizaciji OWASP, kjer so predstavljeni tudi ukrepi za posredovanje in preprečevanje tovrstnih ranljivosti. Kot opazimo je vektor napada povezan predvsem z različnimi načini vrivanja programske kode, ki jih največ zasledimo v spletnih aplikacijah, ki so tudi najbolj dostopne napadalcem.

4.2. RV2: Kako učinkoviti so modeli strojnega učenja pri detekciji ranljivosti programske kode?

V raziskavi smo za napovedovanje in klasifikacijo uporabili dva modela strojnega učenja, in sicer model LSTM in model GRU. Izvedli smo učenje in evalvacijo izbranih modelov, kjer smo prišli do rezultatov, ki kažejo na klasifikacijsko točnost pri detekciji ranljivosti v povprečju 96,4%, kar kaže na visoko pravilnost. Po primerjavi rezultatov s podatki, se je izkazalo, da je metrika točnosti za področje napovedovanja ranljivosti zavajajoča, saj imamo ne uravnoteženo število primerov ranljive in ne ranljive kode, kar se pozna pri izračunu, kjer lahko dosežemo zelo visoko točnost, čeprav napovemo zelo malo pravilno ranljivih primerov, kjer imamo po drugi strani zelo veliko število pravilno napovedanih ne ranljivih primerov. Zato se je pomembno osredotočiti tudi na metriko F1, ki nam pove koliko je uspešnost modelov pri napačno klasificiranih primerih. Izkazalo se je, da smo za metriko F1 pridobili rezultate v razponu med 77% in 91%, kar pripisujemo predvsem količini podatkov, ki smo jih pridobili za učenje modelov. Drastično se izraža razlika po količini podatkov med "SQL injection" kategorijo, za katero imamo največ podatkov in najslabši rezultat po metriki F1, ter "Command injection" kategorijo, za katero smo imeli skoraj petkrat manj podatkov in najboljši rezultat po metriki F1.

Primerjava napovedi med modeloma LSTM in GRU, se je izkazalo, da ni drastičnih razlik, vendar smo opazili, da model LSTM potrebuje večjo količino podatkov, da doseže boljše rezultate točnosti in F1. Ravno obratno velja za model GRU, pri katerem se je izkazalo, da je bolj učinkovit pri napovedovanju že pri manjši količini podatkov.

4.3. RV3: Kakšne rezultate dosega algoritem s povezovalnimi pravili (angl. rule–based) glede na strojno učenje temelječi statični analizi programske kode?

Tekom raziskave smo prišli do ugotovitev, da je ključnega pomena pri napovedovanju ranljivosti, znižanje števila napačno klasificiranih primerov, saj so raziskave [5] pokazale, da je za razvijalce sprejemljiv rezultat pod 20%, da še lahko učinkovito upravljajo z ranljivostmi. Z modeloma LSTM in GRU, smo v povprečju dosegali rezultat po metriki F1 86,5%, kar pomeni približno 14% primerov, ki so statistično napačno klasificirani. Raziskave [4, 12], ki temeljijo na pristopu s povezovalnimi pravili, so pokazale, da dosegajo glede na rezultate med 20% in 40% stopnjo primerov, ki so statistično napačno klasificirani.

5. ZAKLJUČEK

V raziskavi smo naslovili problem odkrivanja ranljivosti v izvorni kodi z metodami strojnega učenja. Na področju podatkovnega rudarjenja in strojnega učenja, poznamo vrsto različnih metod in modelov za obdelavo, učenje in napovedovanje, ki se uporabljajo in so prilagojene za reševanje različnih vrst problemov. V prvi fazi raziskave smo se tako osredotočili na pridobivanje podatkov in problem odkrivanja znanja in povezav med podatki. Podatke smo pridobili iz javno dostopnih zbirk podatkov OSV in odprtokodnih repozitorijev na platformi Github. Nato smo uporabili metode obdelave naravnega jezika z modelom Word2Vector, s katerim smo pripravili podatke za učenje modelov LSTM ter GRU in napovedovanje ranljivosti. Zbrane podatke smo razdelili na pet kategorij ranljivosti in izvedli učenje modelov LSTM in GRU. Izvedli smo evalvacijo, ki je pokazala spodbudne rezultate pri napovedovanju znanih ranljivosti. Pri učenju modelov smo se osredotočili na zniževanje napačno klasificiranih primerov, kar je pomembno predvsem s stališča razvijalcev. Prišli smo do ugotovitev, da so metode strojnega učenja perspektivne na področju napovedovanju ranljivosti, vendar pa je omejitev takšnih metod ravno pomanjkanje podatkov, katerih za redke ali nove ranljivosti praktično ni.

LITERATURA

[1] Christakis, M. and Bird, C. (2016) What developers want and need from program analysis: An empirical study. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA ASE 2016 332–343. Association for Computing Machinery. Accessed: 16.3.2022.

[2] Glyder, J., Threatt, A. K., Franks, R., Adams, L., and Stoker, G. (2021) Some analysis of common vulnerabilities and exposures (cve) data from the national vulnerability database (nvd). *Proceedings of the Conference on Information Systems Applied Research ISSN* 1508. Accessed: 14.3.2022.

[3] Smith, J., Do, L. N. Q., and Murphy-Hill, E. (2020) Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. *Proceedings of the Sixteenth USENIX Conference on Usable Privacy and Security*. USENIX Association, USA. Accessed: 14.3.2022.

[4] Croft, R., Newlands, D., Chen, Z., and Babar, M. A. (2021) An empirical study of rule-based and learning-based approaches for static application security testing. *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, New York, NY, USA ESEM '21. Association for Computing Machinery. Accessed: 8.3.2022.

[5] Kumar, R. and Goyal, R. (2020) Modeling continuous security: A conceptual model for automated devsecops using open-source software over cloud (adoc). *Computers and Security*, **97**, 101967. Accessed: 5.1.2022.

[6] Morrison, P. J., Pandita, R., Xiao, X., Chillarege, R., and Williams, L. (2018) Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering*, **23**, 1383–1421.

[7] Kronjee, J., Hommersom, A., and Vranken, H. (2018) Discovering software vulnerabilities using data-flow analysis and machine learning. *Proceedings of the 13th International Conference on Availability, Reliability and Security*, New York, NY, USA ARES 2018. Association for Computing Machinery. Accessed: 16.3.2022.

[8] Kaur, G., Malik, Y., Samuel, H., and Jaafar, F. (2018) Detecting blind cross-site scripting attacks using machine learning. *Proceedings of the 2018 International Conference on Signal Processing and Machine Learning*, New York, NY, USA SPML '18 22–25. Association for Computing Machinery. Accessed: 16.3.2022.

[9] Ghaffarian, S. M. and Shahriari, H. R. (2017) Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, **50**. Accessed: 18.3.2022.

[10] Bilgin, Z., Ersoy, M. A., Soykan, E. U., Tomur, E., Çomak, P., and Karaçay, L. (2020) Vulnerability prediction from source code using machine learning. *IEEE Access*, **8**, 150672–150684. Accessed: 18.3.2022.

[11] Hovsepyan, A., Scandariato, R., Joosen, W., and Walden, J. (2012) Software vulnerability prediction using text analysis techniques. *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, New York, NY, USA MetriSec '12 7–10. Association for Computing Machinery. Accessed: 15.3.2022.

[12] Jimenez, M., Le Traon, Y., and Papadakis, M. (2018) [engineering paper] enabling the continuous analysis of security vulnerabilities with vuldata7. *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep., pp. 56–61. Accessed: 19.3.2022.