Kauno technologijos universitetas

Informatikos fakultetas

Pažeidžiamumų programiniame kode aptikimas statiniam testavimui tobulinti

Baigiamasis magistro projektas

Lukas Arlauskas

Projekto autorius

Prof. Evaldas Vaičiukynas

Vadovas

Kaunas, 2024

Kauno technologijos universitetas

Informatikos fakultetas

Pažeidžiamumų programiniame kode aptikimas statiniam testavimui tobulinti

Baigiamasis magistro projektas

Veiklos skaitmeninimas ir sistemų architektūros (6211BX009)

Lukas Arlauskas

Projekto autorius

Prof. Evaldas Vaičiukynas

Vadovas

Doc. Vytautas Evaldas Rudžionis

Recenzentas

Kaunas, 2024

Kauno technologijos universitetas

Informatikos fakultetas

Lukas Arlauskas

Pažeidžiamumų programiniame kode aptikimas statiniam testavimui tobulinti

Akademinio sąžiningumo deklaracija

Patvirtinu, kad:

1. baigiamąjį projektą parengiau savarankiškai ir sąžiningai, nepažeisdama(s) kitų asmenų autoriaus ar kitų teisių, laikydamasi(s) Lietuvos Respublikos autorių teisių ir gretutinių teisių įstatymo nuostatų, Kauno technologijos universiteto (toliau – Universitetas) intelektinės nuosavybės valdymo ir perdavimo nuostatų bei Universiteto akademinės etikos kodekse nustatytų etikos reikalavimų;

2. baigiamajame projekte visi pateikti duomenys ir tyrimų rezultatai yra teisingi ir gauti teisėtai, nei viena šio projekto dalis nėra plagijuota nuo jokių spausdintinių ar elektroninių šaltinių, visos baigiamojo projekto tekste pateiktos citatos ir nuorodos yra nurodytos literatūros sąraše;

3. įstatymų nenumatytų piniginių sumų už baigiamąjį projektą ar jo dalis niekam nesu mokėjęs (-usi);

4. suprantu, kad išaiškėjus nesąžiningumo ar kitų asmenų teisių pažeidimo faktui, man bus taikomos akademinės nuobaudos pagal Universitete galiojančią tvarką ir būsiu pašalinta(s) iš Universiteto, o baigiamasis projektas gali būti pateiktas Akademinės etikos ir procedūrų kontrolieriaus tarnybai nagrinėjant galimą akademinės etikos pažeidimą.

Lukas Arlauskas

Patvirtinta elektroniniu būdu

Arlauskas Lukas. Pažeidžiamumų programiniame kode aptikimas statiniam testavimui tobulinti. Magistro projektas / vadovas prof. Evaldas Vaičiukynas; Kauno technologijos universitetas, Informatikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Informacijos sistemos, Informatikos mokslai.

Reikšminiai žodžiai: statinė kodo analizė, pažeidžiamumų programiniame kode aptikimas, didieji kalbos modeliai, atsitiktiniai miškai.

Kaunas, 2024. 81 p.

Santrauka

Šiame tiriamajame projekte tiriamas pažeidžiamumų programiniame kode aptikimo uždavinys naudojant natūralios kalbos apdorojimo ir atsitiktinių miškų metodus. Literatūros apžvalgos dalyje pateikiami jau egzistuojantys tyrimai pažeidžiamumų aptikimo srityje, naudojami metodai, sukurti modeliai, jiems sukurti naudojami duomenų rinkiniai ir jų problemos. Tyrimo dalyje pastebėta, kad statiniai analizatoriai kenčia nuo klaidingų taikinio klasės spėjimų gausos. Išanalizuota mašininio mokymosi metodų teikiama nauda šiai problemai spręsti.

Eksperimentams naudotas *DiverseVul* C kodo failų rinkinys. Duomenys buvo apdoroti ir programinis kodas paverstas vektorine reprezentacija pasinaudojant *CodeLlama* didžiaisiais kalbos modeliais. Eksperimento metu įvertinti keturi atsitiktinių miškų klasifikatoriai, juos palyginant su statiniu kodo analizės įrankiu *Flawfinder*.

Detekcijos tikslumui išmatuoti panaudotos tikslumo, preciziškumo, jautrumo, specifiškumo ir Cohen‘o *Kappa* įverčiai. Eksperimentiškai ištyrus sukurtų modelių kiekybinius rodiklius, nustatyta, kad geriausiai pasirodžiusi konfigūracija yra *CodeLlama* 7 milijardų hiperparametrų didysis kalbos modelis ir atsitiktinių miškų modelis, kurio saujos dydis medžiui parenkamas taikant kvadratinės šaknies iš požymių kiekio taisyklę, taikant 0.48 (nustatytą pagal lygių klaidų taisyklę) klasifikavimo ribą. Ši modelio konfigūracija pasirodė geriau už statinį analizatorių *Flawfinder*, aptikdama daugiau nei tris kartus daugiau teisingų pažeidžiamų failų. Identifikuota, kad parinkus detekcijos slenkstį tokį, kad teisingų taikinio klasės spėjimų santykis sutaptų su įrankio *Flawfinder* minėtu santykiu, pastebėtas 64% klaidingų taikinio klasės atvejų sumažėjimas, lyginant su minimu statinės analizės įrankiu.

Atsižvelgiant į atliktą empirinį tyrimą, suprojektuotas, realizuotas demonstracinis kodo spragų aptikimo įrankis. Įrankis ištestuotas rankiniu testavimu pagal scenarijus.

Arlauskas Lukas. Predicting Software Vulnerabilities from Code for Improvement of Static Testing. Master's Final Degree Project / supervisor prof. Evaldas Vaičiukynas; Faculty of Informatics, Kaunas University of Technology.

Study field and area (study field group): Information Systems, Computing.

Keywords: static code analysis, software vulnerability prediction, large language models, random forests.

Kaunas, 2024. 81 p.

Summary

In this research project, the task of detecting vulnerabilities in software code using natural language processing and random forests methods is investigated. The literature review section presents existing research in the field of vulnerability detection, the methods used, developed models, datasets used for their development, and their issues. The research part notes that static analyzers suffer from a high number of false positive predictions. The benefits of machine learning methods for addressing this problem are analyzed.

The *DiverseVul* dataset of C code files was used for experiments. The data was processed, and the source code was converted into a vector representation using the *CodeLlama* large language models. During the experiment, four random forest classifiers were evaluated and compared with the static code analysis tool *Flawfinder*.

To measure detection accuracy, metrics such as accuracy, precision, recall, specificity, and Cohen's *Kappa* were used. Quantitative indicators of the created models were experimentally investigated, and it was found that the best-performing configuration was the *CodeLlama* large language model with 7 billion parameters and a random forest model where the tree's leaf size was chosen using the square root of the number of features rule, applying a classification threshold of 0.48 (determined by the equal error rule). This model configuration outperformed the static analyzer *Flawfinder*, detecting more than three times as many correct vulnerable files. It was identified that by selecting a classification threshold such that the ratio of correct target class predictions matched the ratio mentioned by the *Flawfinder* tool, a 64% reduction in false positive cases was observed compared to the mentioned static analysis tool.

Based on the conducted empirical research, a demonstrative tool for detecting code vulnerabilities was designed and implemented. The tool was tested manually according to scenarios.

Turinys

Lentelių sąrašas ... 8
Paveikslų sąrašas ... 9
Santrumpų ir terminų sąrašas ... 11
Įvadas ... 13
1. Probleminės srities analizė ... 15
1.1. Analizės tikslas ... 15
1.2. Tyrimo objektas, sritis ir problema ... 15
1.3. Sistemų pažeidžiamumo statinio testavimo įrankių analizė ... 15
1.3.1. Statinio programų sistemų saugumo testavimo principai ... 15
1.4. Programų sistemų kodo spragų prognostikos metodų analizė ... 17
1.4.1. Programų sistemų kodo pažaidų prognostikos metodų principai ... 17
1.4.2. Programų sistemų pažaidų prognostikos tradiciniais mašininio mokymosi algoritmais metodų analizė ... 18
1.4.3. Programų sistemų pažaidų prognostikos natūralios kalbos apdorojimo algoritmais metodų analizė ... 23
1.5. Programų sistemų pažaidų prognostikos modeliams apmokyti naudojamų duomenų rinkinių analizė ... 25
1.6. Statinio programų sistemų saugumo testavimo ir mašininio mokymosi metodų susiejimo analizė ... 26
1.7. Tyrimo objekto naudotojų analizė ... 29
1.8. Esamų statinės programų saugos analizės įrankių analizė ... 30
1.8.1. Checkmarx SAST statinės programų saugos analizės įrankio analizė ... 30
1.8.2. SonarSource SonarQube statinės programų saugos analizės įrankio analizė ... 32
1.8.3. Snyk Code statinės programų saugos analizės įrankio analizė ... 32
1.8.4. Veracode Static Analysis statinės programų saugos įrankio analizė ... 33
1.8.5. DeepSource SAST įrankio analizė ... 34
1.9. Analizės išvados ... 35
2. Programų sistemų kodo spragų aptikimo algoritmas ... 37
2.1. Dalykinės srities aprašas ... 37
2.2. Mašininio mokymosi modelio apmokymo etapas ... 37
2.3. Duomenų rinkinio parengimas ... 39
2.4. Kodo vektorizavimas didžiaisiais kalbos modeliais ... 41
2.4.1. Didieji kalbos modeliai ... 41
2.4.2. LLaMA modelio architektūra ... 42
2.4.3. Vektorinių kodo reprezentacijų išgavimas ... 43
2.5. Atsitiktiniai miškai ... 43
2.5.1. Sprendimų medis ... 43
2.5.2. Ansamblinis mokymasis ... 44
2.6. Algoritmo naudojimo etapas ... 45
3. Eksperimentinis pažeidžiamumų programiniame kode aptikimo modelio tyrimas ... 48
3.1. Eksperimento planas ... 48
3.2. Eksperimento rezultatai ... 52
3.3. Sprendimo taikymo rekomendacijos ... 58

4. Pažeidžiamumų programiniame kode aptikimo įrankio eksperimentinės realizacijos projektas 59
4.1. Įrankio eksperimentinės realizacijos panaudojimo atvejų modelis 59
4.2. Kodo spragų aptikimo eksperimentinės realizacijos loginė architektūra 71
5. Programinio kodo spragų aptikimo įrankio realizacija ir testavimas 74
5.1. Sprendimo realizacijos ir veikimo aprašas 74
5.2. Testavimo modelis, duomenys, rezultatai 74
Išvados 75
Literatūros sąrašas 77
Priedai 82
1 Priedas. Common Vulnerability Scoring System (CVSS) analizė 82
2 Priedas. OWASP rizikos vertinimo metodikos analizė 86
3 Priedas. EER slenksčio radimo grafikai modelių klasifikavimo slenksčio nustatymui 89
4 Priedas. Naudotojo vadovas 91

Lentelių sąrašas

1 lentelė. Populiariausios SVDB (pagal Alqahtani) [20] ... 18
2 lentelė. NVD, CNVD ir CNNVD nacionalinių duomenų bazių lyginamoji lentelė [29] ... 19
3 lentelė. Literatūros apie SVDB naudojimą analizės apibendrinimas ... 22
4 lentelė. Programų sistemų pažaidų prognostikos natūralios kalbos apdorojimo algoritmais metodų lyginamoji analizė ... 24
5 lentelė. Duomenų rinkinių lyginamoji analizė ... 26
6 lentelė. Statinio programų sistemų saugumo testavimo ir mašininio mokymosi metodų susiejimo analizės apibendrinimas ... 28
7 lentelė. Naudotojų problemos ... 30
8 lentelė. Įrankių lyginamoji analizė ... 34
9 lentelė. Duomenų rinkinio eilutės (angl. *entry*) struktūra ... 39
10 lentelė. Duomenų pasiskirstymas pagal taikinio atributus ... 40
11 lentelė. Duomenų rinkinyje aptiktos problemos ir jų sprendimai ... 40
12 lentelė. Atliktos *cwe* atributo duomenų rinkinyje transformacijos ... 41
13 lentelė. Eksperimentiniame tyrime naudotų modelių parametrai ... 49
14 lentelė. Sumaišymų matricos pavyzdys ... 50
15 lentelė. Apibendrinti eksperimento metu tirtų modelių rezultatai, lyginant su *Flawfinder* statiniu analizatoriumi ... 53
16 lentelė. Modelių tikslumo rezultatai atskirais CWE identifikatoriais pažymėtiems failams ... 54
17 lentelė. Modelių ir įrankio jautrumo ir specifiškumo rezultatai atskirais CWE identifikatoriais pažymėtiems failams ... 55
18 lentelė. Atsitiktinių miškų modelių palyginimai esant lygioms teigiamų rezultatų atpažinimo galimybėms ... 57
19 lentelė. Įrankio *Flawfinder* eksperimento metu gauta sumaišymo matrica ... 57
20 lentelė. 7b_sqrt modelio eksperimento metu gauta sumaišymo matrica ... 57
21 lentelė. Panaudojimo atvejo „Konfigūruoti įrankį“ specifikacijos lentelė ... 61
22 lentelė. Panaudojimo atvejo „Atlikti pažaidų analizę“ specifikacijos lentelė ... 64
23 lentelė. Panaudojimo atvejo „Pateikti kodo failus“ specifikacijos lentelė ... 65
24 lentelė. Panaudojimo atvejo „Vektorizuoti kodą“ specifikacijos lentelė ... 67
25 lentelė. Panaudojimo atvejo „Taikyti parengtą modelį“ specifikacijos lentelė ... 69
26 lentelė. Panaudojimo atvejo „Eksportuoti analizės rezultatus“ specifikacijos lentelė ... 70
27 lentelė. Sistemos konfigūracijos testavimo atvejis ... 74
28 lentelė. Sistemos naudojimo testavimo atvejis ... 74
29 lentelė. CVSS specifikacijos metrikų vertės ir jų aprašai [74] ... 83
30 lentelė. CVSS bazinių metrikų įverčių skaitinės reprezentacijos [74] ... 85
31 lentelė. OWASP metodologijos rizikos vertinimo matrica [75] ... 89

Paveikslų sąrašas

1 pav. SVP modelių abstraktaus kūrimo principo veiklos diagrama [8] 17
2 pav. Trys programinio kodo analizės metodai naudoti tyrime [32] 20
3 pav. Atlikto tyrimo metodikos UML veiklos diagrama [51] 27
4 pav. Statinės programų sistemų saugumo analizės veiklos panaudojimo atvejų diagrama 29
5 pav. Checkmarx SAST projekto skenavimo progreso stebėjimo langas 30
6 pav. Checkmarx SAST įrankio skenavimo rezultatų peržiūros panelė 31
7 pav. Checkmarx SAST įrankio suteikiamas programų sistemų pažaidų makro perspektyvos grafo peržiūros langas 31
8 pav. SonarQube aptiktų rezultatų peržiūros langas [58] 32
9 pav. Snyk Code AI varikliuko demonstracija [60] 33
10 pav. Veracode Static Analysis įrankio kodo analizės pavyzdiniai rezultatai 33
11 pav. DeepSource SAST įrankio pagrindinis langas 34
12 pav. Dalykinės srities ER modelis 37
13 pav. Modelio apmokymo procesas 38
14 pav. Transformatoriaus architektūra [70] 42
15 pav. Binarinės klasifikacijos sprendimo medžio pavyzdys 44
16 pav. Algoritmo vykdymo žingsniai su apmokytu modeliu 46
17 pav. Eksperimentinio tyrimo plano veiklos diagrama 48
18 pav. ROC kreivės pavyzdys 51
19 pav. Detekcijos gerumo kreivės: ROC (kairėje) ir PRG (dešinėje) 53
20 pav. Programinio kodo pažeidžiamumų aptikimo įrankio panaudojimo atvejų diagrama 59
21 pav. Panaudojimo atvejo „Konfigūruoti įrankį“ veiklos diagrama 60
22 pav. Panaudojimo atvejo „Konfigūruoti įrankį“ sekų diagrama 61
23 pav. Panaudojimo atvejo „Atlikti pažaidų analizę“ veiklos diagrama 62
24 pav. Panaudojimo atvejo „Atlikti pažaidų analizę“ sekų diagrama 63
25 pav. Panaudojimo atvejo „Pateikti kodo failus“ veiklos diagrama 64
26 pav. Panaudojimo atvejo „Pateikti kodo failus“ sekų diagrama 65
27 pav. Panaudojimo atvejo „Vektorizuoti kodą“ veiklos diagrama 66
28 pav. Panaudojimo atvejo „Vektorizuoti kodą“ sekų diagrama 67
29 pav. Panaudojimo atvejo „Taikyti parengtą modelį“ veiklos diagrama 68
30 pav. Panaudojimo atvejo „Taikyti parengtą modelį“ sekų diagrama 69
31 pav. Panaudojimo atvejo „Eksportuoti analizės rezultatus“ veiklos diagrama 70
32 pav. Panaudojimo atvejo „Eksportuoti analizės rezultatus“ sekų diagrama 70
33 pav. Realizuoto programinės įrangos pažeidžiamumų prognostikos įrankio paketų diagrama ... 71
34 pav. Kodo spragų aptikimo įrankio sistemos komponentų diagrama 72
35 pav. Programinio kodo spragų aptikimo įrankio diegimo diagrama 73
36 pav. CVSS specifikacijos nurodomos pažaidų metrikų grupės [74] 82
37 pav. CVSS specifikacijos nurodomas metrikų vertinimo algoritmas [74] 83
38 pav. 7b_log2 modelio neteisingų priėmimų ir atmetimų santykio kreivės 90
39 pav. 7b_sqrt modelio neteisingų priėmimų ir atmetimų santykio kreivės 90
40 pav. 13b_log2 modelio neteisingų priėmimų ir atmetimų santykio kreivės 91
41 pav. 13b_log2 modelio neteisingų priėmimų ir atmetimų santykio kreivės 91
42 pav. Nesukonfigūruotos programinio kodo spragų aptikimo įrankio namų puslapis 92
43 pav. Įrankio konfigūracijos langas 92

44 pav. Įrankio konfigūracijos sėkmės pranešimas 93
45 pav. Sukonfigūruotos sistemos namų puslapis 93
46 pav. Statinės analizės rezultatų puslapis 94

Santrumpų ir terminų sąrašas

Santrumpos:

OWASP – angl. *Open Web Application Security Project*

CWE – angl. *Common Weaknesses Enumeration*

SVDB – angl. *Software vulnerabilities database*

SAST – angl. *Static application security testing*

CVE – angl. *Common Vulnerabilities and Exposures*

NIST – JAV nacionalinio standartų ir technologijos institutas

CVSS – angl. Common Vulnerability Scoring System.

NVD – angl. *National Vulnerabilities Database*

CNVD – angl. *China National Vulnerability Database*

CNNVD – angl. *China National Vulnerability Database of Information Security*

LSTM – angl. *Long short term memory* – architektūra

SCM – angl. Source control management

Terminai:

Neteisingai teigiamas identifikavimas (angl. *false positive*) – situacija, kai rezultatas identifikuojamas kaip turintis tam tikrą bruožą, kai jis jo neturi;

Programų sistemų pažaida - saugumo trūkumas ar spraga, rasta programiniame kode, kurį tada gali išnaudoti ataką atliekantis žmogus

Savirankos agregavimas (angl. *bootstrap aggregating*) – mašininio mokymosi ansamblių metodas, pagerinantis algoritmų stabilumą ir tikslumą treniruodamas kelis modelius ant skirtingų duomenų poaibių ir tada išvesdamas vidurkį iš jų spėjimų.

Transformatoriaus architektūra - neuroninių tinklų struktūra, naudojama natūralios kalbos apdorojimo užduotims, kuri pakeitė sekos apdorojimo metodus, nes naudojasi savišifravimo (angl. *self-attention*) mechanizmu. Ši architektūra leidžia efektyviai spręsti ilgos trukmės priklausomybes tarp sekos elementų be poreikio apdoroti juos nuosekliai.

Didysis kalbos modelis (angl. *large language model,* trump. *LLM*) - dirbtinio intelekto modelis, išmokytas apdoroti ir generuoti žmogaus kalbą, naudojant didžiulį tekstinių duomenų kiekį.

Atsitiktiniai miškai (angl. *random forests*) - mašininio mokymosi metodas, naudojamas klasifikavimo ir regresijos užduotims. Jie sudaryti iš daugybės sprendimų medžių, kurių kiekvienas mokomas su atsitiktine duomenų rinkinio dalimi, o galutinė prognozė gaunama balsuojant už klasifikaciją arba vidurkinant regresijos rezultatus.

Atkūrimas (angl. *recall*) arba jautrumas (angl. *sensitivity*) - klasifikavimo modelio veiklos matas, kuris rodo, kokią dalį teisingų taikinio klasės pavyzdžių modelis sugebėjo aptikti. Jis apskaičiuojamas kaip teisingai nustatytų taikinio klasės atvejų skaičiaus santykis su visais tikraisiais taikinio klasės atvejais.

Specifiškumas (angl. *specificity*) - klasifikavimo modelio veiklos matas, rodantis, kokią dalį teisingų netaikinio klasės pavyzdžių modelis sugebėjo atpažinti. Jis apskaičiuojamas kaip teisingai nustatytų netaikinio klasės atvejų skaičiaus santykis su visais tikraisiais netaikinio klasės atvejais.

Preciziškumas (angl. *precision*) - klasifikavimo modelio veiklos matas, kuris rodo, kokia dalis modelio nustatytų taikinio klasės atvejų yra teisingi. Jis apskaičiuojamas kaip teisingai nustatytų taikinio klasės atvejų skaičiaus santykis su visais modelio nustatytais taikinio klasės atvejais.

Sumaišymų matrica (angl. *confusion matrix*) - įrankis, naudojamas klasifikavimo modelio veiklai vertinti, pateikiantis teisingų ir neteisingų prognozių skaičius pagal tikrąsias ir prognozuotas klases. Ji padeda nustatyti modelio klaidų tipus ir dažnį.

ROC kreivė (angl. *receiver operating curve*) – klasifikavimo modelio grafikas, rodantis santykį tarp jautrumo ir specifiškumo esant įvairioms diskriminacijos riboms. Kuo aukščiau ir kairiau kreivė, tuo geriau modelis atskiria teigiamus ir neigiamus pavyzdžius.

PRG kreivė (angl. *precision-recall gain curve*) - klasifikavimo modelio grafikas, rodantis santykį tarp preciziškumo padidėjimo ir atkūrimo (jautrumo) padidėjimo. Ji ypač naudinga vertinant modelius su nesubalansuotais duomenimis, nes geriau atskleidžia modelio veiklą su retomis klasėmis.

Kappa įvertis - statistinis matas, naudojamas vertinti klasifikavimo modelio tikslumą, lyginant su atsitiktiniu klasifikavimu. Jis apskaičiuojamas kaip tikrojo sutapimo tarp tikrųjų ir prognozuotų klasių santykis su atsitiktinio sutapimo tikimybe, kur 1 reiškia tobulą atitikimą, o 0 - atsitiktinį atitikimą.

Stratifikuotas duomenų dalinimas – metodas, skirtas padalinti treniravimo ir testavimo duomenis taip, kad tas pats duomenų proporcingumas ir charakteristikos išlaikomi abiejuose rinkinio poaibiuose.

Hiperparametrai – konfigūruojami (neišmokstami) modelio nustatymai ar parametrai. Juos reikia apibrėžti prieš mokant modelį.

Įvadas

Darbas priklauso „Veiklos skaitmeninimo ir sistemų architektūrų“ studijų programai.

Darbo problematika ir aktualumas

Šiuolaikinei vis labiau kompiuterizuojamai visuomenei programų sistemų saugumas tampa vis aktualesnė tema. Kuriant įvairias informacines ir programų sistemas, sistemų kūrėjai susiduria su programų saugumo spragomis (angl. *software vulnerability*), kurios panaudojamos įvairioms kibernetinėms atakoms. Šias spragas aptikti svarbu dėl kelių priežasčių: saugumo defektai įmonėms kainuoja milijonus eurų dėl neveikiančios programinės įrangos, sutrikimų joje ir duomenų konfidencialumo pažeidimų [1] [2]. Be to, pažeidimai programinėje įrangoje gali neigiamai paveikti naudotojo patirtį [3]. Pažeidžiama programinė įranga reikalauja sistemų kūrėjų pastangų po programinės įrangos išleidimo. Tai reiškia, kad didelė dalis sistemų kūrimo pastangų yra išnaudojama ne naujo sistemų funkcionalumo kūrimui, bet klaidų taisymui [4]. Dėl to, kuriama programinė įranga, kuri gali aptikti šias spragas programinėje įrangoje. Vienas iš metodų yra statinis programų sistemų saugumo testavimas (angl. *static application security testing, SAST*) [5]. Tačiau SAST įrankiai turi klaidingų taikinio klasės atvejų (angl. *false positive*) tikimybę. Priklausomai nuo įrankio, ši tikimybė varijuoja iki 50% [6]. Šios klaidos papildomai gaišta sistemos kūrėjų laiko resursus, o tai atsiliepia organizacijos finansams [7] [8] [9]. Visai neseniai populiariais tapę didieji kalbos modeliai (angl. *large language models*) po truputį savo pritaikomumą randa ir kodo analizės sferoje. Šiuo metu egzistuoja modelių kurtų ne tik žmonių kalboms, bet ir programavimo kalboms analizuoti [10]. Šiuo metu, tyrimais tirti programų sistemų spragų prognostikos modeliai ir hibridiniai įrankiai leidžia manyti, kad pritaikius prognostikos metodus statinio testavimo įrankiui, gaunamas didesnis analizės rezultatyvumas [8]. Norint tai pasiekti, šiame darbe yra tiriama programų sistemų pažeidžiamumo testavimo sritis.

Darbo tikslas ir uždaviniai

Tyrimo tikslas – efektyvus kodo spragų aptikimas programiniame kode, pasiremiant moderniais natūralios kalbos apdorojimo metodais.

Tikslo pasiekimui išsikelti uždaviniai:
1. Išnagrinėti akademinę literatūrą programinio kodo pažeidžiamumų prognostikos srityje;
2. Pasirinkti programavimo kalbą, jai skirtus statinio testavimo įrankius ir duomenų rinkinius;
3. Pasiūlyti spragų programiniame kode aptikimo sprendimą, paremtą mašininiu mokymusi;
4. Eksperimentiškai ištirti pasiūlytą sprendimą ir palyginti su statinio testavimo įrankio rezultatu;
5. Suprojektuoti ir realizuoti kodo spragų aptikimo įrankį, atsižvelgiant į empirinį tyrimą;
6. Apibendrinti darbo rezultatus ir pateikti rekomendacijas.

Darbo rezultatai ir jų svarba

Pagrindinis darbo rezultatas yra pažeidžiamumų programiniame kode analizės modelis. Tyrimo rezultatai rodo, kad modelio taikymas parinkus detekcijos slenkstį pagal EER euristiką yra aktualus dėl jo gebėjimo korektiškai atpažinti tris kartus daugiau pažeidžiamų kodo failų lyginant su statiniu kodo analizatoriumi *Flawfinder*, o parinkus slenkstį tokį, kuris atpažįsta tinkamus pažeidžiamus failus tokiu pat veiksmingumu kaip *Flawfinder*, modelis patiria daugiau nei perpus mažiau klaidingų taikinio klasės spėjimų. Tai reiškia, kad sistemų kūrėjams naudojantis šiuo įrankiu reikėtų peržiūrėti

mažiau failų. Darbo metu gauti rezultatai aktualūs sistemų kūrėjams, kurie siekia modelį naudoti kaip pradinio filtravimo žingsnį arba antrinę nuomonę šalia statinio analizatoriaus bei kitiems tyrėjams.

Darbo struktūra

Magistro baigiamąjį darbą sudaro penki skyriai, literatūros sąrašas ir keturi priedai. Pirmajame skyriuje analizuojami sistemų pažeidžiamumo statinio testavimo įrankiai, programų sistemų pažaidų prognostikos metodai, programų sistemų pažeidžiamumo įrankių ir prognostikos metodų apjungimas. Taip pat atliekama esamų statinės programų saugos analizės įrankių palyginimas. Antrajame skyriuje aprašomas sukurtas programų sistemų kodo spragų aptikimo algoritmas. Trečiajame skyriuje pateikiamas pažeidžiamumų programiniame kode aptikimo modelio tyrimas, jo rezultatų analizė. Ketvirtame skyriuje pademonstruojamas pažeidžiamumų programiniame kode aptikimo įrankio eksperimentinės realizacijos projektas. Penktame skyriuje aprašomas programinio kodo spragų aptikimo įrankio realizacija ir testavimas. Pabaigoje pateikiamos išvados, literatūros sąrašas ir priedai.

1. Probleminės srities analizė

Šiame skyriuje pateikiama su problemine sritimi susijusios informacijos analizė.

1.1. Analizės tikslas

Analizės tikslas – išsiaiškinti statinio programų sistemų saugumo testavimo principus, dažniausiai naudojamus įrankius, jų privalumus, trūkumus, pažaidos prognostikos modelių principus, abiejų metodikų taikymo kartu principus, kad būtų galima nustatyti kitus prognostinio įrankio statiniam testavimui tobulinti kūrimo žingsnius.

1.2. Tyrimo objektas, sritis ir problema

Šio darbo **tyrimo sritis** yra programų sistemų pažeidžiamumo statinio testavimo įrankiai. **Tyrimo objektas** – statinis programų sistemų pažeidžiamumo testavimas. Tyrimo objekte egzistuojanti **problema** – statinės programų saugos analizės (angl. *Static Application Security Testing*, *SAST*) įrankiai šiuo metu naudoja taisyklėmis grįstas (angl. *rule-based*) metodikas, dėl kurių ryškių trūkumų analizės metu pasitaiko daug neteisingų teigiamos klasės (angl. *false positive*) atvejų. Neteisingai aptiktų teigiamos klasės atvejai reiškia, kad saugumo analizė atlikta neefektyviai [8]. Šis faktorius yra labai svarbus norint išvengti papildomo sistemų kūrėjų resursų iššvaistymo, siekiant taisyti problemą, kuri neegzistuoja. Numatomas problemos **sprendimas** – mašininiu mokymusi grįstas (angl. *learning-based*) metodas pagerinantis programų sistemų statinės analizės rezultatų tikslumą. Sukuriama jo veikimo demonstracija, sukuriant realizaciją, kurioje taikomi statinės programų saugos analizės įrankis ir sukurtas programų saugumo pažaidų prognostikos (angl. *software vulnerabilities prediction*, *SVP*) metodas.

1.3. Sistemų pažeidžiamumo statinio testavimo įrankių analizė

Programų sistemų pažaidos (angl. *Software vulnerability*) yra viena iš didžiausių saugumo problemų šiuolaikiniame pasaulyje [11]. Programų sistemų pažaida apibrėžiama kaip saugumo trūkumas ar spraga, rasta programiniame kode, kurį tada gali išnaudoti ataką atliekantis žmogus [12]. Plačiausiai paplitę testavimo metodai yra statinis sistemų pažeidžiamumo testavimas ir dinaminis sistemų pažeidžiamumo testavimas. Šioje analizėje dėmesys skiriamas statiniam programų testavimui.

1.3.1. Statinio programų sistemų saugumo testavimo principai

Statinis programų sistemų saugumo testavimas (angl. *static application security testing, SAST*) Gartner IT terminų žodyne yra apibrėžiamas kaip technologijų rinkinys, skirtas programinio kodo analizei, ieškant programavimo arba kodo dizaino saugumo spragų [13]. Statinis kodo testavimas yra dažna kodo peržiūros dalis ir šis procesas vyksta sistemos kūrimo metu. Dėl šios priežasties, sąlyginai nesunku pataisyti sistemos kūrimo metu iškilusias spragas, tai lemia mažesnius sistemos kūrimo kaštus ir laiko sąnaudas [5].

Statinė kodo analizė gali būti atliekama arba rankiniu būdu, arba pasitelkiant specializuotus įrankius. Šie įrankiai tikrina kodą remdamiesi iš anksto numatytomis taisyklėmis ir naudoja keletą skirtingų metodų pažaidų paieškai vykdyti. OWASP (angl. *Open Web Application Security Project*) ir Aloraini, Nagappan, German ir Hayashi aprašo kelis tokius metodus [14] [7]:

- Struktūrų atpažintis (angl. *pattern matching*) – Viega, Bloch, Kohno, Tadayoshi ir McGraw aprašė vieną tokiu principu veikiantį C ir C++ kalbų programinio kodo analizatorių *ITS4* [15]. Šis įrankis veikia programos kodo failą paversdamas leksiniais žetonais ir tada žetonų sraute taiko struktūrų atpažintį;
- Duomenų srauto analizė (angl. *data flow analysis*) – naudojama surinkti dinaminę duomenų informaciją programinėje įrangoje;
- Kontrolinio srauto grafas (angl. *control flow graph*) – programinės įrangos veikimas reprezentuojamas kryptiniu grafu. Mazgas reprezentuoja kodo bloką, o jungimas tarp mazgų – perėjimą tarp kodo blokų;
- Sutepties analizė (angl. *taint analysis*) – bandoma identifikuoti kintamuosius „suteptus“ su naudotojo valdoma įvestimi. Rastų ir nesanitizuotų kintamųjų reikšmių perdavimas į pažeidžiamų funkcijų parametrus laikoma sistemos pažaida;
- Verčių intervalo analizė (angl. *value range analysis*) – šio metodo esmė yra panaudoti duomenų srauto analizę kintamųjų verčių apribojimams sekti;
- Simbolinis vykdymas (angl. *symbolic execution*) – naudojamos simbolinės vertės kai konkrečios vertės nėra žinomos, dėl to, išvesties vertės apskaičiuotos šiuo metodu gali būti reprezentuojamos kaip funkcijos kurių argumentas – įvestos simbolinės vertės.

OWASP šaltinyje pateiktos statinio programų sistemų saugumo testavimo metodų stiprybės ir silpnybės. Kaip minėta anksčiau, SAST įrankiai susiduria su dideliais neteisingai teigiamų identifikuojamų pažeidžiamumų kiekiais. Neteisingai teigiamai identifikuotas pažeidžiamumas šio darbo kontekste apibrėžiamas pagal NIST žodyną – įspėjimas, netinkamai indikuojantis neegzistuojančią saugumo spragą [16]. Šių atvejų mažinimui ir jų valdymui sukurta nemažai metodų. Akremi klasifikuoja šiuos metodus į septynias kategorijas [17]:

1. Mašininiu mokymusi grįsti metodai (angl. *Machine Learning based approaches*) – naudojami mašininio mokymosi algoritmai saugumo spragų nuspėjimui, neteisingai identifikuotoms spragoms nuspėti ir jas sumažinti;
2. Pagrindinių priežasčių analize grįsti metodai (angl. *Root Causes based approaches*) – pagrindinių priežasčių analizės metodas naudojamas įvykių priežastingumui nustatyti ir pritaikyti efektyvias koreguojamąsias priemones. Šis automatizavimo būdas turi savų trūkumų, pvz., netinkamai ištrinami statinės analizės įrankio (angl. *Static analysis tool*, *SAT*) generuoti pranešimai;
3. Modelio tikrinimu grįsti metodai (angl. *Model Checking based approaches*) – aprašomas sistemos elgesio savybių modelis, pagal kurį vėliau tiriamos sistemos būsenos. Tai yra naudinga ieškant neteisingai identifikuotų spragų – pranešimai tikrinami pagal iš anksto numatytą modelį;
4. Duomenų gavyba grįsti metodai (angl. *Data Mining based approaches*) – naudingi išskirti tam tikras struktūras dideliuose duomenų kiekiuose. Dažnai, duomenų gavyba grįsti metodai taikomi kartu su mašininiu mokymusi grįstais metodais;
5. Taisyklėmis grįsti metodai (angl. *Rule based approaches*) – šiais metodais siekiama taikyti žinias informacijai interpretuoti. Taisyklės gali būti tiek sukurtos žmogaus, tiek sugeneruotos mašininio mokymosi algoritmo;
6. Semantika grįsti metodai (angl. *Semantics based approaches*) – naudojant matematinę logiką, aprašomos taisyklės, apibūdinančios identifikuotus konstruktus kode ir sąryšius tarp jų;
7. Programų pjūviais grįsti metodai (angl. *Slicing based approaches*) – pagrindinė šio metodo naudojimo priežastis – vengti kompleksiškų analizių sumažinant originalų kodą iki minimalaus, išlaikant tą pačią programos elgseną. Šis darinys vadinamas pjūviu (angl. *slice*).

Lipp, Banescu ir Pretschner [18] atliko empirinį statinių C kodo analizatorių tyrimą, kuriame analizuojami šeši skirtingi statiniai analizatoriai: Flawfinder, Cppcheck, Infer, CodeChecker, CodeQL, CommSCA. Visi iš jų, išskyrus paskutinį yra atviro kodo ir prieinami nemokamai. Pagrindiniai skirtumai identifikuoti tarp statinių analizatorių:

- Iš visų analizatorių, tik *Flawfinder* pateikia rezultatus standartizuotoje CWE notacijoje. Kitų gautus rezultatus teko konvertuoti į CWE notaciją;
- Kai kuriems analizatoriams (pvz. *Cppcheck*) reikia, kad kodas būtų kompiliuojamas. Tuo tarpu, *Flawfinder* naudoja sintaksinės analizės principus, kas nereikalauja, kad programa būtų kompiliuojama;
- Skirtingi analizatoriai aptinka skirtingus pažeidžiamumus dėl naudojamų skirtingų analizės principų.

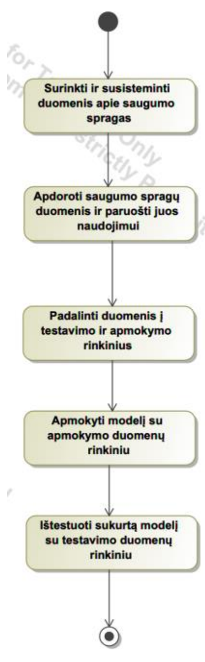
Rezultatai atskleidžia, kad nors ir komercinis statinis analizatorius *CommSCA* pasirodė geriausiai, jis vis tiek nepataikė į 47%-80% pažeidžiamumų. Tyrimo metu nustatyta, kad iš atvirai prieinamų ir nemokamų kodo analizatorių geriausiai pasirodė *CodeQL* ir *Flawfinder*. Verta pastebėti, kad Flawfinder turi mažesnę klaidingų taikinio klasės atvejų tikimybę.

1.4. Programų sistemų kodo spragų prognostikos metodų analizė

Programų sistemų kodo spragų prognostika (angl. *software vulnerability prediction, SVP*) yra taip pat žinomas metodas, skirtas atlikti statinę kodo analizę ir identifikuoti saugumo spragas [8]. SVP metodai siekia analizuoti jau turimus surinktus duomenis ir iš pasikartojančių reiškinių nuspėti prognozę ateities rezultatui.

1.4.1. Programų sistemų kodo pažaidų prognostikos metodų principai

SVP modelių bendrinis kūrimo metodas aprašytas [8]. Šis metodas iliustruotas UML veiklos diagrama (angl. *Activity diagram*) ir yra pateikiamas 1 pav.



1 pav. SVP modelių abstraktaus kūrimo principo veiklos diagrama [8]

SVP modelio kūrimas vyksta pagal procese numatytus žingsnius:

1. Modelio kūrėjai turi surinkti duomenų rinkinį (angl. *dataset*) apie duomenų spragas;
2. Modelio kūrėjai duomenis apdoroja. Dažnai pasitaiko, kad surinkti duomenys yra nevienodo / netinkamo formato arba formos. Modelio duomenys turi būti apdoroti pagal prognostinio metodo reikalavimus;
3. Padalinti duomenų rinkinį į apmokymo ir testavimo rinkinius. Dalinimo proporcijos priklauso nuo tyrėjo. Kartais, kai klasifikavimo uždaviniui nepakanka vienos ar kelių konkrečių klasių duomenų, naudojamas stratifikuotas duomenų dalinimas (angl. *stratified train-test-split*) [19]. Šio metodo esmė – dalinti duomenis proporcingai pagal klasėje esamų elementų skaičių;
4. Apmokyti modelį – modelis mokosi klasifikuoti duomenis, optimizatoriui keičiant modelio parametrų reikšmes, atsižvelgiant į modelio hiperparametrų (angl. *hyperparameters*) reikšmes;
5. Ištestuoti modelį – modelio testavimas atliekamas su testiniais duomenimis, kurių modelis nematė.

Daug SVP modelių naudoja duomenis klasifikavimui iš įvairių programų sistemų pažaidų duomenų bazių (angl. *Software vulnerability database, SVDB*) [20]. Alqahtani indikuoja šešias populiariausias SVDB pagal jų naudojamumą moksliniuose straipsniuose [20], kurios pateikiamos 1 lentelėje.

1 lentelė. Populiariausios SVDB (pagal Alqahtani) [20]

SVDB	**Naudojamumas straipsniuose**	**Prieiga**
NVD (angl. *National Vulnerabilities Database*)	31	[21]
OWASP	15	[22]
CVE (angl. *Common Vulnerabilities and Exposures*)	14	[23]
OSVDB (angl. *Open Source Vulnerabilities Database*)	10	-
CWE (angl. *Common Weakness Enumeration*)	9	[24]
SecurityFocus	7	-

Lentelėje matoma, kad daugiausiai naudotasi NVD duomenų baze. Verta paminėti, kad dvi iš šių duomenų bazių šiuo metu jau nebeteikia paslaugų, todėl į tolimesnę analizę įtrauktos nebus.

1.4.2. Programų sistemų pažaidų prognostikos tradiciniais mašininio mokymosi algoritmais metodų analizė

Zhang, Caragea ir Ou atliko empirinį NVD duomenų bazės naudojimo programinės įrangos pažaidoms aptikti tyrimą [25]. NVD duomenų bazėje saugumo pažaidų duomenys laikomi formatu <D, CPE, CVSS>, kur:

- D – duomenų rinkinys, susidedantis iš publikavimo datos, pažaidos aprašo ir išorinių išnašų apie pažaidą;
- CPE – bendroji platformos enumeracija (angl. *Common Platform Enumeration*). Tai – struktūrizuota įvardijimo schema, skirta IT sistemoms, programinei įrangai ir paketams. Ši schema naudojama tikslingai ir struktūrizuotai įvardinti programinių paketų ir programinės įrangos versijoms. CPE žodynas yra palaikomas JAV nacionalinio standartų ir technologijos

instituto (angl. trump. *NIST*) [26]. CPE yra plačiai naudojamas tokių pažaidų skenavimo įrankių kaip *NMAP* [27], o standarto tipinė eilutė susideda iš:

  - detalės (angl. *part*) – programinė įranga, operacinė sistema, aparatūrinė įranga;
  - tiekėjo (angl. *vendor)*;
  - produkto (angl. *product*);
  - versijos (angl. *version*);
  - atnaujinimo (angl. *update*);
  - varianto (angl. *edition*);
  - kalbos (angl. *language*).

- NVD duomenų bazė palaiko CVSS 2.0 ir 3.x specifikacijų bazinių metrikų vertes, kurios reprezentuoja esmines kodo spragos savybes [28]. Gilesnė CVSS standarto analizė pateikiama priede.

Tyrime pasirinkta spėjama charakteristika yra laikas iki kitos spragos (angl. *Time To Next Vulnerability, TTNV*), o charakteristikos iš kurių spėjama yra laikas, atstumas tarp skirtingų versijų (angl. *versiondiff*), programinės įrangos pavadinimas ir CVSS įvertis. Tyrimo atlikimo metu tyrėjai nustatė NVD duomenų bazės apribojimus, kurie trukdo nuspėti TTNV metriką: trūkstama informacija, pažaidos paskelbimo laikų nesutapimai, duomenų klaidos.

Forain, de Oliveira ir de Sousa savo tyrime [29] lygino tris nacionalines pažaidų duomenų bazes: minėtą NVD, CNVD (angl. *China National Vulnerability Database*) ir CNNVD (angl. *China National Vulnerability Database of Information Security*). Iškelta problematika, kad NVD duomenų bazė neturi duomenų apie saugumo pažaidas esančias programinėje ir aparatinėje įrangoje, tiekiamoje tokių Kinijos tiekėjų kaip Xiaomi ir Huawei. Apibendrintos straipsnio išvados pateikiamos lentele (žr. 2 lentelė).

2 lentelė. NVD, CNVD ir CNNVD nacionalinių duomenų bazių lyginamoji lentelė [29]

Duomenų bazė / **Lyginamasis aspektas**	**NVD**	**CNVD**	**CNNVD**
Duomenų formatas	JSON	XML	XML
Duomenų prieiga	Atvira	Atvira, tačiau nedraugiška užsieniečiams, informacija pateikiama tik kinų mandarinų kalba. Pastebimas galimas pažaidų slėpimas [30].	
Duomenų struktūra	• NVD pažaidos duomenys • NVD nuoroda (angl. *reference*) • CWE nuoroda • Bazinis CPE • Kompleksinis CPE	• CNVD pažaidos duomenys • CVE nuoroda • Produkto duomenys	• CNNVD pažaidos duomenys • CVE nuoroda • Produkto duomenys • CNNVD nuoroda
Duomenų bazės dydis	1.5 GB	146.9 MB	2.4 GB
Pažaidų skaičius	178906	99261	180567
Trūkstamų CVSS įverčių kiekis	10933	326	8466

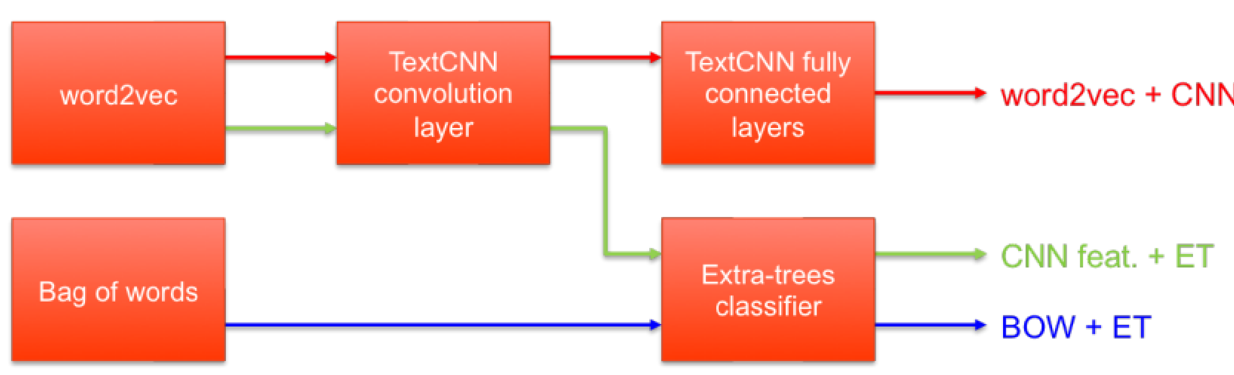
Pateikta duomenų lentelė rodo, kad CNVD ir CNNVD duomenų bazės, nors ir turi daugiau duomenų sudėjus, šie duomenys turi standartizavimo ir kitų problemų:

- Duomenų formatas ne vienodas – CNVD ir CNNVD nelaiko duomenų apie CVE nuorodas;
- CNNVD duomenų bazė vidutiniškai pažaidas su aukštu CVSS įverčiu pateikia su didesniu vėlavimu nei tas, kurios turi mažą CVSS įvertį, lyginant su NVD duomenų baze. Dėl šios priežasties, [30] autoriai įtaria, kad CNNVD (kuri yra valdoma ir palaikoma Kinijos Liaudies Respublikos Valstybės saugumo ministerijos) sąmoningai nuslepia kai kurias saugumo spragas, kurios galimai stipriai išnaudojamos Kinijos saugumo interesams ginti;

Tačiau, nepaisant šių problemų, tyrimo metu buvo nustatyta, kad CNNVD duomenų bazė laiko daugiau duomenų, lyginant su NVD duomenų baze apie Kinijos aparatūrinės ir programinės įrangos tiekėjų Huawei ir ZTE produktuose aptiktas saugumo spragas.

Yosifova, Tasheva ir Trifonov [31] atliko lyginamąją mašininio mokymosi algoritmų taikymo programų sistemų pažaidoms klasifikuoti analizę. Tyrime naudojami duomenys iš CVE (angl. *Common Vulnerabilities and Exposures*) duomenų bazės ir bandoma atpažinti CVE pažaidos tipą iš apibūdinimo. Lyginamoji analizė tarp tiesinių atraminių vektorių, daugianarinio naivaus Bajeso klasifikatoriaus ir atsitiktinių miškų klasifikatoriaus atskleidė, kad gauti preciziškumo (angl. *precision*), atkūrimo (angl. *recall*) ir F1 įverčio rezultatai yra nuoseklūs.

Tuo tarpu Harer'is et. al. [32] tyrime analizavo automatizuoto programų sistemų pažaidų aptikimo galimybes, pasinaudojant mašininio mokymo metodus. Duomenų rinkiniui sudaryti buvo panaudotos atviro kodo C ir C++ programavimo kalbomis rašytos programos, o pažaidos gaunamos iš CVE duomenų bazės. Tyrime diskutuojamas skirtumas tarp jau sukompiliuotų programų ir programinio kodo. Pasak autorių, sukompiliuotos programos, tegu ir gauna kompiliatoriaus reikšmingų bruožų išgavimo privalumą, tačiau šie bruožai dažnai abstrahuoja tam tikras kodo savybes, kurios yra naudingos programų sistemų pažaidų atpažinimui. Programinio kodo bruožų gavybai panaudotas specifinis C/C++ lekseris, o gauti žetonai paverčiami vektorine reprezentacija, pasinaudojant *word2vec* algoritmu. Programinio kodo analizėje naudoti metodai iliustruojami 2 pav.



2 pav. Trys programinio kodo analizės metodai naudoti tyrime [32]

Pagal pateiktą iliustraciją, autoriai naudojo tris modelius:

- Baziniam variantui pasirinktas žodžių krepšelio ir atsitiktinių medžių metodas;

- Žodžių vektorius, kuriuo apmokomas *TextCNN* sakinių atpažinimo modelio [33] konvoliucinis sluoksnis ir išeiga naudojama kaip įvestis atsitiktinių medžių metodui;
- Žodžių vektorius, kuriuo apmokomas visas *TextCNN* sakinių atpažinimo modelis.

Tyrimas parodė, kad ploto po ROC kreive (angl. *Area under ROC curve, AUC ROC*) buvo didžiausias antrojo ir trečiojo modelio atvejais – 0.87. Tai reiškia, kad modeliai klasifikavimą atlieka pakankamai gerai. Taip pat tyrimo metu prieita prie išvados, kad programinio kodo analize grįsti modeliai pažeidžiamo kodo atpažinimo užduotį atliko geriau, nei kompiliuotų failų analizės pagrindu kurti modeliai.

Natūralios kalbos apdorojimas (angl. *Natural Language Processing, NLP*) yra apibrėžiamas kaip „algoritmų naudojimas natūralios kalbos savybių atpažinčiai, tam, kad kompiuterių sistemos suprastų žmonių užrašytą tekstą“. NLP apima kompiuterinių sistemų apmokymą, kaip išgauti semantinius duomenis iš parašyto teksto, versti iš vienos kalbos į kitą ir atpažinti spausdintinį ar rašytinį tekstą [34]. NLP metodai gali būti naudojami programinio kodo analizei. Pavyzdžiui, Ziems‘as ir Wu [35] naudoja NLP metodus spragų atpažinčiai C ir C++ kode. Kodo spragų duomenys imamas iš SARD (angl. *Software Assurance Reference Dataset*) duomenų rinkinio, kuriame pateikiami C/C++ kodas ir CWE žyma spragos identifikavinui. Tyrime lyginamos kelios giliojo mokymosi architektūros programinio kodo atpažinčiai:

- LSTM (angl. *Long short term memory*) architektūra [36]. Šios architektūros modeliai naudoja skirtingą formulę nei standartiniai rekurentiniai neuroniniai tinklai (angl. *recurrent neural networks, RNN*), kas padeda išvengti sprogstančių arba išnykstančių gradientų (angl. *exploding/vanishing gradient*) problemos;

- Dvikryptis LSTM (angl. *Bidirectional LSTM*). Ši architektūra sprendžia svarbios informacijos ilgose sekose pasimetimą;

- BERT (angl. *Bidirectional Encoder Representations from Transformers*). BERT modeliai reikalauja fiksuoto ilgio sekos. Tyrėjai pradžioje apmokė BERT modelį su angliškos Vikipedijos duomenų rinkiniu kaip BERT bazę.

Tyrimo metu atlikta šių modelių lyginamoji analizė. Kadangi BERT modelis reikalauja fiksuoto ilgio sekos, duomenų rinkinys skaidytas į fiksuoto ilgio žetonų rinkinius, kurie perduodami į modelį. Tiksliausiai atpažinimą atliko BERT ir dvikrypčio LSTM modelių ansamblis, pasiektas tikslumas 93.49 proc.

Kito tyrimo metu atlikta mašininio mokymosi technikų taikymo programinės įrangos spragų atpažinčiai lyginamoji analizė [37]. Tyrime naudoti duomenys iš NVD duomenų bazės. Analizuoti šie mašininio mokymosi metodai:

- Tiesioginio duomenų perdavimo atgalinės propagacijos neuroninis tinklas (angl. *feed-forward back propagation neural network, FFBPNN*). Vienasluoksnis FFBPNN turi sigmoidinės funkcijos sluoksnį ir išeigos tiesinį sluoksnį;
- Kaskadinio duomenų perdavimo atgalinės propagacijos neuroninis tinklas (angl. *cascade-forward back propagation neural network, CFBPNN*). Skirtumas tarp CFBPNN ir FFBPNN yra tas, kad CFBPNN atveju įeigos vertės perskaičiuojamos po kiekvieno paslėpto sluoksnio atgalinio svorių propagavimo į įvesties sluoksnį;

- Adaptyvioji neraiškiosios logikos sistema (angl. *Adaptive Neuro Fuzzy Inference System, ANFIS*). Tai technika kuri sukombinuoja neraiškiosios logikos ir dirbtinių neuroninių tinklų privalumus. ANFIS yra neraiškios logikos sistema, kurios parametrai koreguojami neuroadaptyviai, t.y. panašiu principu kaip tai daro dirbtiniai neuroniniai tinklai;
- Daugiasluoksnis perceptronas (angl, *multilayered perceptron*, *MLP*). Šio metodo principas panašus kaip FFBPNN, tik atgalinei propagacijai naudojamas gradientinio nuolydžio (angl. *gradient descent*) metodas;
- Atraminių vektorių mašina (angl. *Support Vector Machine, SVM*). Klasifikavimo metodas, kuris sugeneruoja n-dimensinę hiperplokštumą, kuri padalina duomenų rinkinį į dvi grupes;
- Saviranko agregavimas (angl. *bootstrap aggregating,* trump. *bagging*). Remiamasi idėja, kad prognozuojantis modelis gali būti stabilesnis, pakartotinai naudojant jau egzistuojančius pavyzdžius (angl. *bootstrapping*). Patrauklus ansamblių metodas, kai reikia sumažinti variaciją;
- M5rules. "Skaidyk ir valdyk" pagrindo algoritmas, skirtas generuoti sprendimų sąrašams.
- M5P. Algoritmas skirtas tiems atvejams, kai reikia sumažinti standartinį nuokrypį.

Autorių nuomone, dėl R kvadrato ir koreliacijos koeficiento artimo 1, geriausiai pasirodė ANFIS, saviranko agregavimo ir FFBPNN metodai.

Iki šiol analizuotos literatūros apibendrinimas pateikiamas lentele (3 lentelė).

3 lentelė. Literatūros apie SVDB naudojimą analizės apibendrinimas

Šaltinis	Naudota SVDB	Tyrimo tikslas	Tyrime naudota analizės metodologija
[25]	NVD	Duomenų kasybos metodų empirinė analizė, bandymas nuspėti laiką iki kitos pažaidos mašininio mokymosi metodais	WEKA programinės įrangos mašininio mokymosi algoritmų implementacijos: • tiesinė regresija; • mažiausių vidutinių kvadratų metodas; • daugiasluoksnis perceptronas; • RBF tinklas; • SMO regresija; • Gauso procesai
[29]	NVD, CNVD, CNNVD	Metodologijos sukūrimas NVD, CNVD ir CNNVD SVDB lyginimui	Duomenų struktūravimas, analizė ir vizualizavimas
[31]	CVE	Lyginamoji mašininio mokymosi algoritmų taikymo programų sistemų pažaidoms klasifikuoti iš tekstinio apibūdinimo	• atraminių vektorių mašina; • naivusis Bajeso klasifikatorius; • atsitiktiniai miškai
[32]	CVE	Automatizuoto programų sistemų pažaidų aptikimo galimybių tyrimas iš programinio kodo ir kompiliuotų failų, panaudojant mašininio mokymo metodus	• lekseris + CNN; • lekseris + CNN sluoksnis + atsitiktiniai miškai; • žodžių krepšelis (angl. Bag of Words) + atsitiktiniai miškai
[35]	SARD, NVD	Ištirti natūralios kalbos apdorojimo metodų taikymo galimybes programinės įrangos spragų atpažinimui	• LSTM architektūra; • Dvikryptis LSTM; • BERT modelis

[37]	Iš NVD surinkti autorių pačių keli duomenų rinkiniai	Ištirti įvairių mašininio mokymosi metodų pritaikymą programinės įrangos spragų atpažinimui.	• Tiesioginio duomenų perdavimo atgalinės propagacijos neuroninis tinklas; • Kaskadinio duomenų perdavimo atgalinės propagacijos neuroninis tinklas; • Adaptyvioji neraiškiosios logikos sistema; • Daugiasluoksnis perceptronas; • Atraminių vektorių mašina; • Savirankos agregavimas; • M5rules; • M5P;

1.4.3. Programų sistemų pažaidų prognostikos natūralios kalbos apdorojimo algoritmais metodų analizė

Khare et. al. savo tyrime lygino tris didžiuosius kalbos modelius: OpenAI GPT-4, CodeLlama-13B ir CodeLlama-7B [38]. Modeliai lyginti su trimis duomenų rinkiniais, iš kurių du buvo sintetiniai (OWASP ir Juliet), o vienas realus (CVE-fixes). Naudojamas įvesties inžinerijos (angl. *prompt engineering*) metodas, kuriuo perduodamos specifinės instrukcijos, pvz., kuriuos pažeidžiamumus aptikti, kodo aprašas. Geriausi rezultatai pasiekti su sintetiniu rinkiniu perduodant instrukcijas naudojantis modeliu GPT-4: Preciziškumas siekia 0.87, atkūrimas – 0.92, F1 įvertis – 0.89, Su realiu duomenų rinkiniu perdavus instrukcijas, geriausiai pasirodė CodeLlama-7B (preciziškumas 0.49, atkūrimas 0.98, F1 įvertis 0.65).

Noever‘is tyrė GPT-4 pažeidžiamo kodo atpažinimo ir taisymo galimybes. Tyrime skenuojami failai prieš ir po sutvarkymo „Snyk“ statiniu kodo analizatoriumi [39]. Rezultatai atskleidė, kad GPT-4 identifikavo beveik 4 kartus daugiau pažeidžiamų vietų lyginant su statiniu analizatoriumi, o pritaikius siūlomus pakeitimus, pažeidžiamumų sumažinta 90 %.

Thapa et. al. tyrė įvairių transformatorių pagrindu sukurtų kalbos modelių taikymą programinės įrangos pažeidžiamumų aptikimui, naudojant tiek RNN pagrindu sukurtus modelius (BiLSTM ir BiGRU), tiek transformatorių pagrindu sukurtus modelius (BERT, DistilBERT, RoBERTa, CodeBERT, GPT-2 ir Megatron variantai) [40]. Rezultatai parodė, kad transformatorių pagrindu sukurti modeliai, ypač GPT-2 Large ir GPT-2 XL, lenkia RNN pagrindu sukurtus modelius binarinio ir kategorinio klasifikavimo užduotyse tokiuose duomenų rinkiniuose kaip VulDeePecker ir SeVC. Pastebėtina, kad GPT-2 Large pasiekė 95,51% F1 balą buferio klaidoms, žymiai aukštesnį nei BiLSTM 86,60%, o resursų valdymo klaidų kategorijoje šie modeliai pasiekė 96,84% ir 95% atitinkamai binarinėje klasifikacjoje. GPT-2 XL pasiekė 95,94% F1 balą kategorinėje klasifikacijoje. Tyrimas taip pat nagrinėja didelių modelių patobulinimo iššūkius ir rekomenduoja naudoti tokias platformas kaip HuggingFace, integruotas su DeepSpeed, efektyviam mokymui.

Taip pat, tirtas buvo ir distiliuotų kalbos modelių efektyvumas [41]. Modelių žinių distiliavimas (angl. *knowledge distillation*) yra procesas, kuriuo stengiamasi perkelti didesnio modelio žinias arba sukauptą informaciją į mažesnį ir efektyvesnį modelį. Tikimasi, kad mažesnio modelio veiksmingumas išliks daug maž toks pat su mažesniais skaičiavimo resursų reikalavimais. Lyginami iš anksto apmokyti kontekstualizuoti modeliai ir jų distiliuotos versijos su nekontekstualizuotais modeliais. Geriausią rezultatą išgaunantis modelis panaudojamas karkaso sudarymui, kuris

lyginamas su kitomis kodo analizės sistemomis. Rezultatai parodė, kad distiliuotas CodeBERT modelis pasiekė geriausią balansą tarp tikslumo ir efektyvumo. Jo pagrindu sukurtas pažeidžiamumų atpažinimo karkasas pasiekė 40% preciziškumą, 65% atkūrimą.

Fu ir Tantithamthavorn‘as sukūrė LineVul architektūrą, kurioje naudojamas CodeBERT transformatoriaus architektūra grindžiamas modelio variantas, apdorojus duomenis baitų poros lygio užkodavimu [42]. Tiriamas modelio gebėjimas atpažinti pažeidžiamumo statusą funkcijos mastu ir eilutės mastu. Modelis atlieka binarinę klasifikaciją ir tiria koduotes ar rasti pažeidimai. Rezultatai parodė 91% F1 įvertį ir 65% bendras tikslumas žinomiausioms 10 pažaidoms.

Analizės apibendrinimas pateikiamas 4 lentelėje.

4 lentelė. Programų sistemų pažaidų prognostikos natūralios kalbos apdorojimo algoritmais metodų lyginamoji analizė

Šaltinis	Modelis	Tyrimo metodika	Pasiekti rezultatai
[38]	GPT-4 CodeLlama-13B CodeLlama-7B	Lyginami modelių rezultatai įvairiems sintetiniams ir realiems duomenų rinkiniams: OWASP (sintetinis), Juliet (sintetinis), CVE-fixes (realus). Modeliams perduodamos specifinės instrukcijos kaip analizuoti duomenis.	Geriausi rezultatai pasiekti su sintetiniu rinkiniu perduodant instrukcijas naudojantis modeliu GPT-4: Preciziškumas siekia 0.87, atkūrimas – 0.92, F1 įvertis – 0.89, Su realiu duomenų rinkiniu perdavus instrukcijas, geriausiai pasirodė CodeLlama-7B (preciziškumas 0.49, atkūrimas 0.98, F1 įvertis 0.65)
[39]	GPT-4	Modeliui duodama užduotis sutvarkyti pažeidžiamas kodo vietas. Skenuojami failai prieš ir po sutvarkymo „Snyk“ statiniu kodo analizatoriumi	GPT-4 identifikavo beveik 4 kartus daugiau pažeidžiamų vietų lyginant su statiniu analizatoriumi. Pritaikius siūlomus pakeitimus, pažeidžiamumų sumažinta 90 %.
[40]	Abipusis LSTM Abipusis GRU Įvairūs BERT pagrindo modeliai GPT-2	Apdoroti SeVC arba VulDeePecker duomenys naudojami apmokyti modelius. Lyginami rezultatai binarzinei ir kategorinei klasifikacijai.	GPT-2 Large ir GPT-2 XL, lenkia RNN pagrindu sukurtus modelius binarinio ir kategorinio klasifikavimo užduotyse tokiuose duomenų rinkiniuose kaip VulDeePecker ir SeVC. Pastebėtina, kad GPT-2 Large pasiekė 95,51% F1 balą buferio klaidoms, žymiai aukštesnį nei BiLSTM 86,60%, o resursų valdymo klaidų kategorijoje šie modeliai pasiekė 96,84% ir 95% atitinkamai binarinėje klasifikacjoje. GPT-2 XL pasiekė 95,94% F1 balą kategorinėje klasifikacijoje.
[41]	Iš anksto apmokyti kontekstualizuoti modeliai Nekontekstualizuoti modeliai Distiliuotos BERT modelių versijos	Lyginami iš anksto apmokyti kontekstualizuoti modeliai ir jų distiliuotos versijos su nekontekstualizuotais modeliais. Geriausią rezultatą išgaunantis modelis panaudojamas karkaso sudarymui, kuris lyginamas su kitomis kodo analizės sistemomis.	Distiliuotas CodeBERT modelis pasiekė geriausią balansą tarp tikslumo ir efektyvumo. Jo pagrindu sukurtas pažeidžiamumų atpažinimo karkasas pasiekė 40% preciziškumą, 65% atkūrimą.
[42]	LineVul	Naudojamas CodeBERT transformatoriaus architektūra grindžiamas modelio variantas, apdorojus duomenis baitų poros lygio užkodavimu. Tiriamas modelio	Pasiektas 0.91 F1 įvertis, 0.97 preciziškumas ir 0.86 atkūrimas funkcijos mastu. Bendras tikslumas – 0.65 top 10 pažaidų tipams.

		gebėjimas atpažinti pažeidžiamumo statusą funkcijos mastu ir eilutės mastu	

Tačiau nors ir dauguma analizuotų sprendimų naudoja arba iš anksto parengtus duomenų rinkinius, arba ima duomenis iš egzistuojančių SVDB ir autoriai pateikia tyrimuose palyginti puikius rezultatus, šie rezultatai dažnai realaus pasaulio scenarijuose atlieka reikiamas užduotis labai prastai [43]. Identifikuotos SVP metodų problemos:

- Duomenų šaltiniai ir jų anotacijos yra nerealistiškos. Modeliai apmokyti ant paprastų ir sintetinių kodo pavyzdžių yra apriboti atpažinti spragas būtent paprastuose kodo pavyzdžiuose. Šalia to, bet koks modelis, kuris naudoja statinio analizatoriaus anotuotus duomenis, kentės nuo tos pačios neteisingai pažymimų tariamai egzistuojančių spragų problemos ir rezultatai nebus objektyvūs;
- Nerealus pažeidžiamo kodo ir neutralaus kodo pasiskirstymas. Realiame scenarijuje neutralaus kodo yra žymiai daugiau nei pažeidžiamo;
- Žetonų pagrindu kurti modeliai kenčia nuo sintaksinės analizės trūkumo. Šio tipo modeliai veikia su prielaida, kad tarp žetonų tėra leksinė priklausomybė, todėl semantinė priklausomybė prarandama. Tuo tarpu grafų pagrindu kurti modeliai geba atpažinti duomenų priklausomybę ir lengviau atpažįsta pažeidžiamas kodo eilutes, kurios neseka viena po kitos;
- Dabartiniai modeliai yra „trapūs“. Nors modeliai išmoksta atskirti pažeidžiamą kodą nuo neutralaus, modelių apmokymo paradigma neteikia ypatingo dėmesio atskirties tarp pažeidžiamų ir neutralių kodo pavyzdžių didinimo;
- Dabartinių modelių vertinimas yra ribotas. Visi egzistuojantys sprendimai skelbia savo vertinimą, kuris gautas naudojant jų kūrėjų pačių susikurtą duomenų rinkinį. Viskas, ką galima sužinoti iš tokio vertinimo yra tai, kaip gerai sukurtas modelis tinka atpažinti spragas sukurtame duomenų rinkinyje.

1.5. Programų sistemų pažaidų prognostikos modeliams apmokyti naudojamų duomenų rinkinių analizė

Programų sistemų pažaidų prognostikos metodai yra duomenimis grįstas procesas, kuriuo stengiamasi iš jau išanalizuotų duomenų nuspręsti apie naujus ir nematytus duomenis ar jie pažeidžiami, ar ne. Dėl šios priežasties yra svarbu naudoti tinkamus duomenis apmokyti dirbtinio intelekto modeliams [44] Šiame poskyryje aptariami skirtingi viešai pasiekiami duomenų rinkiniai būtent šiai užduočiai atlikti.

SATE IV yra C/C++ ir Java kodo rinkinys su žinomomis klaidomis, susidedančiomis iš virš 120 tūkstančių kodo funkcijų [45]. SATE IV turi sintetinius kodo pavyzdžius kurie yra sužymėti pagal CWE pažaidų vertinimo metodiką. Pašalinus dublikatus, SATE IV turi 55 procentų pažeidžiamų kodo funkcijų ir 45 procentus nepažeidžiamų kodo funkcijų [43]. Naudojant tik šį duomenų rinkinį, dėl nerealistiško pažeidžiamo/nepažeidžiamo kodo balanso iškyla persimokymo galimybė ir dėl savo sintetinės prigimties tegali atpažinti paprastus kodo pažeidžiamumus. Dėl šios priežasties, šį duomenų rinkinį reikėtų pildyti realaus kodo pavyzdžiais.

Rebecca L. Russell et. al. tyrime naudoja savo sukurtą duomenų rinkinį *Draper VDISC* [46] (duomenų rinkinys pasiekiamas [47]), kuris susideda iš *SATE IV* duomenų rinkinio, papildyto viešų *Github* ir *Debian* distribucijos repozitorijų kodo spragomis, gautomis iš trijų skirtingų statinių kodo analizatorių ir pašalinus viena kitą dublikuojančias funkcijas siekiant išvengti modelio persimokymo.

Autoriai teigia, kad pastarieji du šaltiniai buvo pasirinkti dėl to, kad *Debian* repozitorija turi geros kokybės kodo, o *Github* viešose repozitorijose galima rasti žemesnės kokybės, taigi ir labiau pažeidžiamo, kodo. Kita vertus, *Draper VDISC* šios ataskaitos rašymo metu nebuvo jau atnaujintas virš dvejus metus, taigi gali neapimti naujesnių pažeidžiamumų. Trečia, dėl duomenų rinkinio prigimties (duomenys buvo surinkti iš trijų analizatorių), tai bet koks modelis, kuris apmokomas *Draper VDISC* rinkiniu, mokosi ne klaidų atpažinimo, o SAST įrankių emuliavimo [48].

Chakraborty et. al. savo tyrime sukūrė naują duomenų rinkinį *ReVeal* [43]. Duomenų rinkinys [49] sudarytas iš dviejų programų (*Chromium* ir *Debian*) surinktų pažeidžiamumų iš *BugZilla* ir *Debian* security tracker pažeidžiamumų sekimo įrankių. Pažeidžiamumai atrinkti tie, kurie jau pataisyti, taigi užfiksuotos teisingai identifikuotos spragos (angl. true positives). Pažeidžiamumai žymimi versijavimo sistemos baze, t.y. pažymima ir versija kuri pažeidžiama, ir versija kuri yra „švari" (neturinti pažeidžiamumo). Kaip duomenų rinkinio dalis paimtas ir kitas FFMPEG+Qemu rinkinys.

Chen‘as et. al. pasiūlė savo pažeidžiamo kodo duomenų rinkinį DiverseVul [50]. Duomenų rinkinys Sudarytas iš 18945 C/C++ pažeidžiamų funkcijų (apie 150 CWE tipų) ir 330492 nepažeidžiamų funkcijų. Apima 295 projektus. Pažeidžiamumai atrinkti tokie, kurie jau yra patvirtinti su siekiu pagerinti giliojo apmokymo pritaikymo pažeidžiamumų atpažinimui galimybes.

Šių analizuotų duomenų rinkinių lyginamoji analizė pateikiama 5 lentelėje.

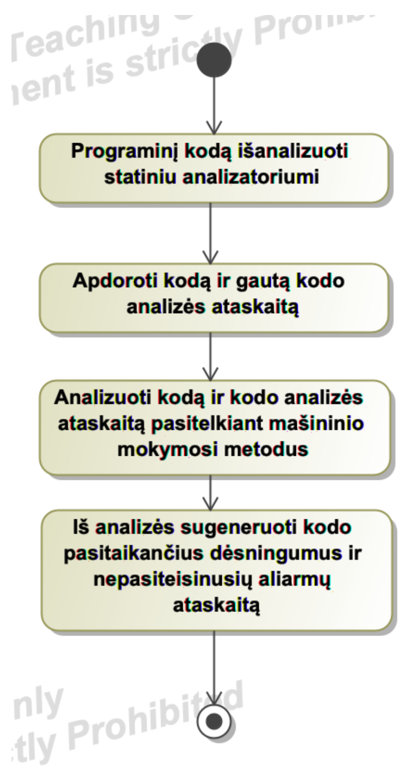
5 lentelė. Duomenų rinkinių lyginamoji analizė

	SATE IV	***Draper VDISC***	***ReVeal***	***DiverseVul***
Dydis	12 tūkstančių kodo pavyzdžių	1.2 milijono pažeidžiamumų	18 tūkstančių kodo pavyzdžių	350 tūkstančių kodo pavyzdžių
Kalba	C/C++ ir Java	C/C++ ir Java	C/C++	C/C++
Programos	- (Sintetinis)	Github viešų repozitorijų kodas ir Debian repozitorija	Chromium, Debian	295 projektų kodas
Rinkinio prigimtis	Sintetinis	Realus	Realus	Realus
Rinkinio vienetų granuliarumas	Funkcija	Funkcija	Funkcija	Funkcija
Pažeidžiamų / nepažeidžiamų kodo vietų santykis	Reliatyviai nerealistiškas	Realistiškas	Labai realistiškas	Labai realistiškas
Žymėjimas	Aiškus	Aiškus	Aiškus	Labai aiškus
Pasiekiamumas	Viešas	Viešas	Viešas	Viešas
Palaikymas	Reguliariai atnaujinamas	Neatnaujintas 2 metus	Neatnaujintas 3 metus	Sukurtas prieš pusę metų

1.6. Statinio programų sistemų saugumo testavimo ir mašininio mokymosi metodų susiejimo analizė

Kaip minėta 1.3.1 poskyryje, mašininio mokymosi metodai gali būti naudojami statinės kodo analizės tobulinimui. Šiame poskyryje analizuojama, kaip statinio programų sistemų saugumo testavimo metodai gali būti sujungiami su mašininio mokymosi metodais siekiant didesnio saugumo analizės efektyvumo.

Tokie mašininio mokymosi metodai kaip Bajeso naivusis klasifikatorius arba LSTM architektūra gali būti naudojami neteisingai identifikuotų spragų atpažinimui [51]. Tyrime teigiama, jog statinės analizės įrankiai remiasi aproksimacijomis ir tam tikromis prielaidomis, kurios leidžia įrankį pritaikyti didelėms sistemoms. Tyrime analizuotas Java kodas, o jo metodikai vizualizuoti pasitelkiama UML veiklos diagrama, pateikiama 3 pav.



3 pav. Atlikto tyrimo metodikos UML veiklos diagrama [51]

Šioje diagramoje matyti tyrėjų atlikti žingsniai, kurie detalizuojami žemiau:

- Programinis kodas įkeliamas į SAST įrankį. SAST įrankis grąžina kodo statinės analizės ataskaitą;
- Kodas apdorojamas jį redukuojant. Šis žingsnis yra svarbus, kadangi kodo segmentai, kurie nėra labai svarbūs analizei gali modeliui suteikti triukšmą. Modelis taip gali persimokyti. Šiame specifiniame tyrime buvo naudotos dvi kodo redukavimo technikos:
    - Analizuoti tik metodą, kuriame yra pažeidžiamo kodo eilutė. Tyrėjai perspėja, kad ši technika turi trūkumų: kartais kodo pažaidos priežastys gali apimti kelis metodus;
    - Egzistuojant duotam fiksuotam taškui programoje, jos kodas pjaustomas, kol gaunama minimizuota programos versija, kuri atlieka tą patį funkcionalumą duotame taške. Autoriai teigia, kad šitoks metodas teoriškai generuoja programos atgalinį pjūvį, kuris padengia visas spragai aktualias kodo eilutes. Šiam uždaviniui autoriai pasitelkė karkasą WALA.
- Modeliai apmokomi. Šiam dalykui pasitelkti du modeliai: naiviuoju Bajeso spėjimu remtas modelis ir LSTM architektūros modelis. Pirmasis yra lengviau interpretuojamas, o antrasis gali išmokti sudėtingesnes struktūras;
- Modelis sugeneruoja pasitaikančių kodo dėsningumų ir neteisingai identifikuotų spragų ataskaitą.

Pasitelkus specifinės SAST programinės įrangos generuojamas ataskaitas.ir juos išanalizavus, buvo pastebėta, kad LSTM modelio tikslumas buvo didžiausias - 89.6 procento (precizija siekė 97.3 proc.), naudojant tik metodo kodą, o naudojant atgalinių pjūvių metodą su LSTM pasiektas 85 procentų tikslumas (preciziškumas - 97 proc.). Tačiau, verta pastebėti ir tai, kad LSTM naudojant pažeidžiamos funkcijos izoliavimo metodą 18.7 procentų neegzistuojančių spragų klasifikuojami kaip teisingos pažaidos, o naudojant atgalinių pjūvių metodą, 21.8 procento tariamai neteisingai identifikuotų atvejų buvo iš tikrųjų kodo pažaidos.

Kitas korėjiečių mokslininkų tyrimas neteisingai identifikuotų spragų mažinimui naudoja atraminių vektorių mašinos metodą [52]. Šis tyrimas taip pat tyrė Java atviro kodo projektus ir naudojo iš esmės panašią metodiką kaip aukščiau aprašytas tyrimas. Pagrindiniai skirtumai: duomenų apdorojimui naudojamas abstrakčiojo sintaksės medžio aptrumpinimo (angl. *AST pruning*) ir požymių vektorių gavybos metodus. Iš įvairių projektų, tokių kaip maven-core ir jenkins-core pasinaudojant SPARROW statiniu analizatoriumi buvo išgauta 190 teisingai identifikuotų spragų ir 75 neteisingai identifikuotos spragos. Iš šių, modelis teisingai klasifikavo 172 teisingai identifikuotas spragas ir 58 neteisingai identifikuotas spragas. Tai reiškia, kad modelio klasifikavimo tikslumas siekė 86.79 proc. o neteisingai teigiamos klasės atvejų sumažinta 37.33 proc., netinkamai išmetant 3.16 proc. teisingai identifikuotų teigiamos klasės atvejų.

Dar vienas tyrimas naudojo mašininio mokymosi modelių ansamblį kelių statinės analizės įrankių įspėjimų vertinimui [53] . Šiame tyrime naudotas Juliet testavimo įrankių rinkinys buvo panaudotas rinkinio sudarymui. Iš viso išanalizuota virš 280 tūkst. aliarmų. Kadangi naudoti SAST įrankiai (Clang, Cppcheck ir Frama-C) generuoja skirtingo formato ataskaitas, tyrimo metodikoje siūloma kurti suvienodintą ataskaitą. Ansambliui panaudotas AdaBoost klasifikatorius. Klasifikavimo validacijai buvo panaudota kryžminė patikra. Ansamblio tikslumas – 80.5 proc, preciziškumas – 67.8 proc., atmintis – 95.8 proc. Tai reiškia, kad atmetus suklasifikuotus neteisingai identifikuotus spragų atvejus, teisingai identifikuotų atvejų būtų prarandama mažiau nei 5 procentai.

Konvoliuciniai neuroniniai tinklai taip pat gali būti naudojami neteisingai identifikuotų spragų analizei atlikti [54]. Tyrime analizuotas C ir C++ kodas, kuris tirtas pagal šešias dažniausiai pasitaikančias spragas. Tyrimo metodika apima lekserio naudojimą kodo vertimui į žetonus, tada naudojant word2vec algoritmą, žetonai verčiami į vektorių (panašiai, kaip tai darė ir [32]). Duomenys validuoti pasinaudojant kryžmine patikra. Šio metodo rezultatai geriausiu atveju - vidutiniai: vidutinė atmintis – 51.09 proc., o preciziškumas siekė 79.72 proc.

Analizės rezultatai apibendrinami literatūros lyginamosios analizės lentele (žr. 6 lentelė).

6 lentelė. Statinio programų sistemų saugumo testavimo ir mašininio mokymosi metodų susiejimo analizės apibendrinimas

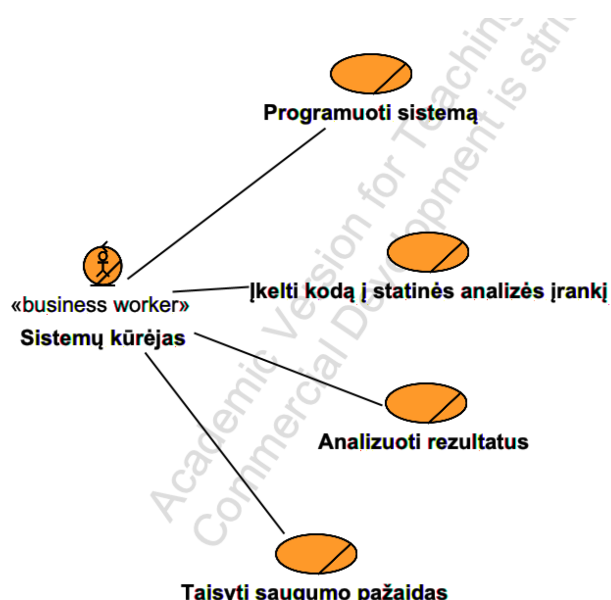
Šaltinis	Analizuoto kodo kalba	Tyrimo metu naudoti mašininio mokymosi metodai	Tyrimo rezultatai
[51]	Java	Bajeso naivusis klasifikatorius ir LSTM	LSTM pasiekė geriausių rezultatų: tikslumas 89.6 procento precizija 97.3 proc. Dalis rezultatų (18.7 proc.) klasifikuojami kaip teisingos pažaidos
[52]	Java	Atraminių vektorių mašina	Klasifikavimo tikslumas - 86.79 proc. Neteisingai identifikuotų spragų

			sumažinta 37.33 proc., neteisingai pašalinant 3.16 proc. teisingų spragų
[53]	C/C++	Mašininio mokymosi modelių ansamblis	Ansamblio tikslumas – 80.5 proc, preciziškumas – 67.8 proc., atmintis – 95.8 proc.
[54]	C/C++	Konvoliucinis neuroninis tinklas	Atmintis – 51.09 proc., preciziškumas - 79.72 proc.

Iš pateiktų šaltinių analizės galima daryti prielaidą, kad statinės programų sistemų analizės įrankių ir mašininio mokymosi metodų susietumo sprendimų paieška yra aktuali, o iš rezultatų galima spręsti, kad yra galimybė pateikti geresnį sprendimą. Konvoliucinių neuroninių tinklų taikymas šioje užduotyje pasirodė prasčiausiai. Verta pastebėti, kad dažniausiai esant netobulam modelio diskriminavimui tarp klasių, kai mažinamas neteisingai identifikuotų atvejų skaičius, atsiranda rizika, kad išmesime tam tikrą iš tikro naudingą ir vertingą statinės analizės įrankio rezultatų kiekį [17].

1.7. Tyrimo objekto naudotojų analizė

Statinės programų sistemų saugumo analizėje dalyvauja sistemos kūrėjas, kuris iš esmės yra programuotojas arba testuotojas. Statinės programų sistemų saugumo testavimo veiklos panaudojimo atvejų diagrama pateikiama 4 pav.



4 pav. Statinės programų sistemų saugumo analizės veiklos panaudojimo atvejų diagrama

Šioje diagramoje pateikiamas aktorius yra programų sistemų arba informacinių sistemų kūrėjas, kuris programuoja sistemą, sistemos kodą pateikia analizei, analizuoja rezultatus ir pagal juos taiso programų sistemų pažaidas.

Analizuojant programinės įrangos kodą ir ieškant jame saugumo spragų gali kilti problemų. Šios naudotojo problemos pateikiamos žemiau (žr. 7 lentelė).

7 lentelė. Naudotojų problemos

Problema	**Kaip ji sprendžiama dabar**
Statinės programų sistemų saugumo analizės įrankiai dažnai neteisingai identifikuoja neegzistuojančias spragas	Naudojami keli SAST įrankiai ir jų rezultatai lyginami.
Statinės programų sistemų analizės įrankių rezultatų interpretavimas gali būti intensyvi užduotis žmogiškųjų resursų atžvilgiu	Egzistuojantys metodai nenumato analizės rezultatų interpretuojamumo lengvinimo.

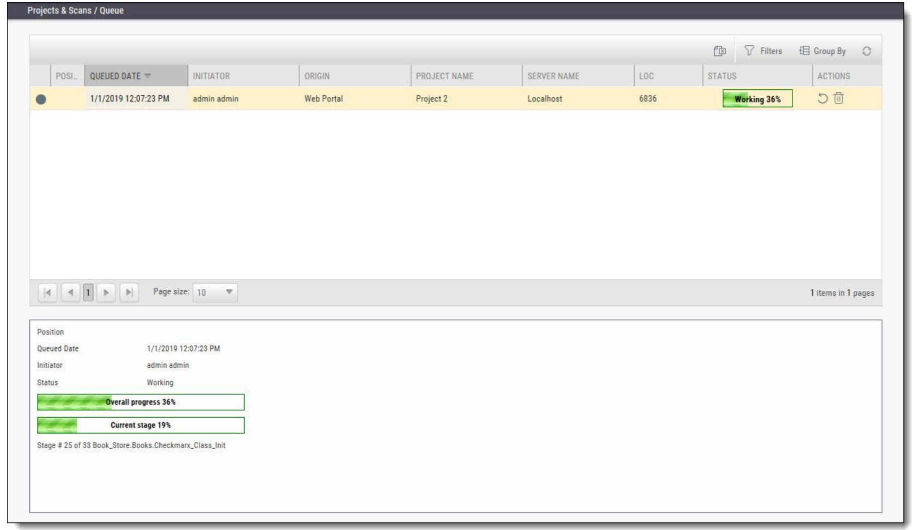
Tyrimo metu kuriamas programų sistemų spragų prognostikos algoritmas galėtų veikti kaip pirminis filtravimo žingsnis, taip pat pateikti standartizuotą rezultatų aprašą, taip palengvinant šių rezultatų interpretavimą.

1.8. Esamų statinės programų saugos analizės įrankių analizė

Šiame skyriuje yra atliekama tokių statinės programų saugos analizės įrankių tokių kaip „Checkmarx SAST“ analizė ir esamų sistemų palyginimas.

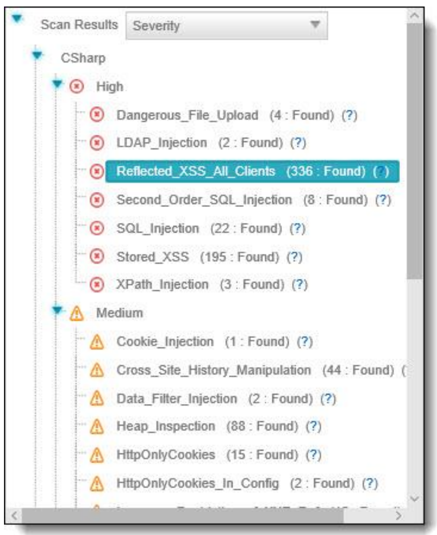
1.8.1. Checkmarx SAST statinės programų saugos analizės įrankio analizė

Įmonė „Checkmarx“ siūlo savo SAST sprendimą „Checkmarx SAST“ [55] (dokumentacija prieinama [56]). Šis įrankis integruojasi su beveik visomis plačiai žinomomis integruotomis kūrimo aplinkomis (angl. *Integrated Development Environment, IDE*), programinio kodo valdymo platformomis (angl. *Source Code Management, SCM*), nuolatinės integracijos (angl. *Continuous Integration, CI*) serveriais.Checkmarx SAST leidžia stebėti vykdomo skenavimo progresą.



5 pav. Checkmarx SAST projekto skenavimo progreso stebėjimo langas

Kodo analizę galima atlikti lokaliuose projektuose, esančiuose toje pačioje mašinoje, kaip įrankis, arba kodą galima iš kodo versijavimo (angl. *source control*) sistemų, tokių kaip TFS (angl. trump. *Team Foundation Server*), SVN (angl. trump. *Subversion*), *GIT* arba *PerForce*. Analizuojant kodą, galima išimti tam tikrus aplankus iš analizės apimties, galima nurodyti kodo analizės periodiškumą ir pan. 6 pav. pateikiamas skenavimo rezultatų pavyzdys.

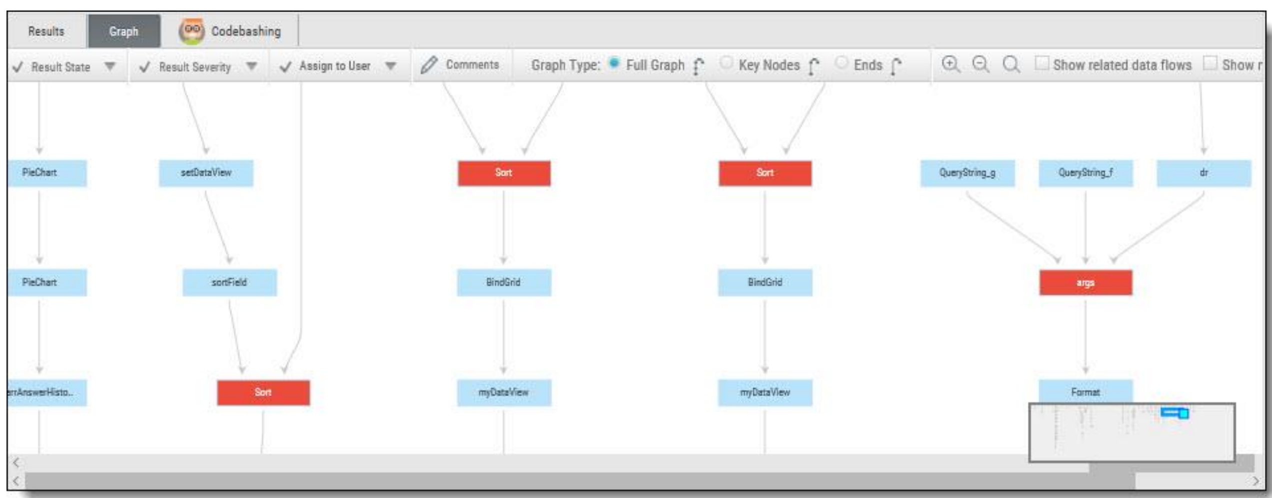


6 pav. Checkmarx SAST įrankio skenavimo rezultatų peržiūros panelė

Kodo analizės rezultatai pateikiami atskiroje panelėje. Kiekvieną rezultatą galima peržiūrėti ir pamatyti, su kuria kodo dalimi konkreti pažaida susijusi. Šias pažaidas galima įvertinti pagal jų rimtumą remiantis šiais standartais:

- OWASP Top 10 2017;
- OWASP Top 10 2013;
- PCI DSS v3.2;
- FISMA 2014;
- NIST SP 800-53;
- OWASP Mobile 10 2016;
- ASD STIG 4 10;

Taip pat yra galimybė apsirašyti savo vertinimą, tačiau tai reikalauja aprašyto vertinimo metodo integracijos su Checkmarx SAST programine įranga. Rezultatų tarpusavio susietumą galima atvaizduoti grafu (7 pav.).

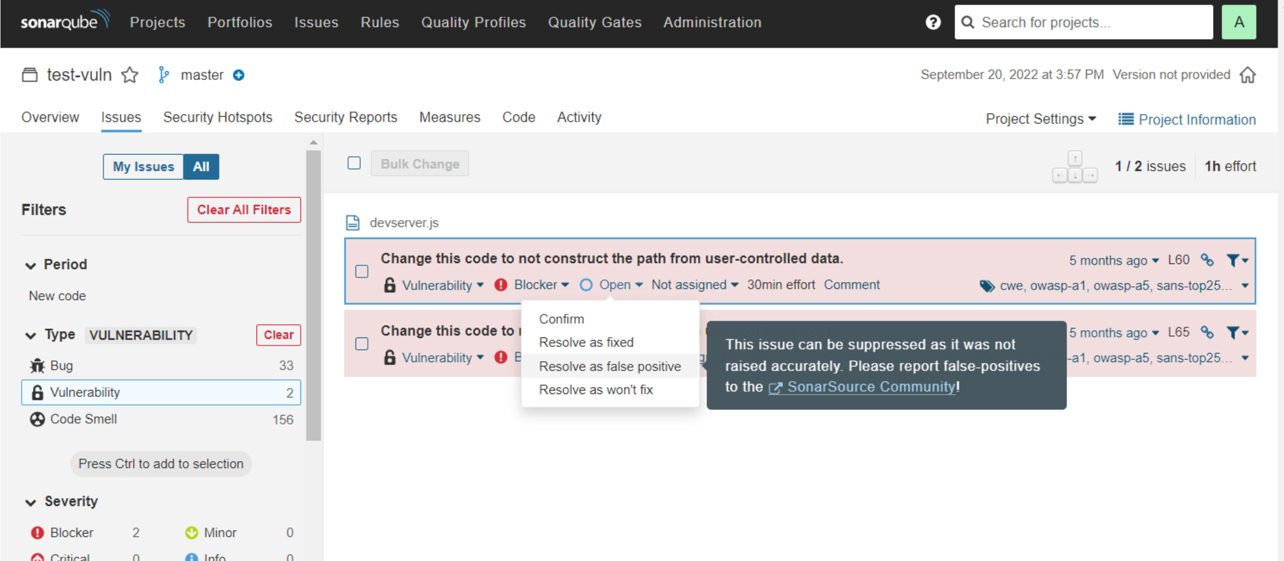


7 pav. Checkmarx SAST įrankio suteikiamas programų sistemų pažaidų makro perspektyvos grafo peržiūros langas

Šiame lange galima matyti kaip programos funkcijos, jų komponentai yra susiję su pažaidomis. Raudonai pažymėtos grafo viršūnės laikomos optimaliausiomis taisymui.

1.8.2. SonarSource SonarQube statinės programų saugos analizės įrankio analizė

SonarSource SonarQube yra statinės programų saugos analizės įrankis [57] (dokumentacija pasiekiama [58]), kuris yra pasiekiamas kaip IDE įskiepis (SonarLint), programinė įranga diegiama savuose serveriuose (angl. *On Premises*) (*SonarQube*) ir kaip programinė įranga kaip paslauga (angl. *Software as a Service, SaaS*). Sistema gali atlikti kodo analizę, pateikti pažaidų ir galimų kodo dizaino klaidų analizę (8 pav.), palaiko GIT ir SVN versijavimo sistemas, tačiau pažaidos skirstomos pagal vidinį standartą, o ne pagal aukščiau minimus standartus.

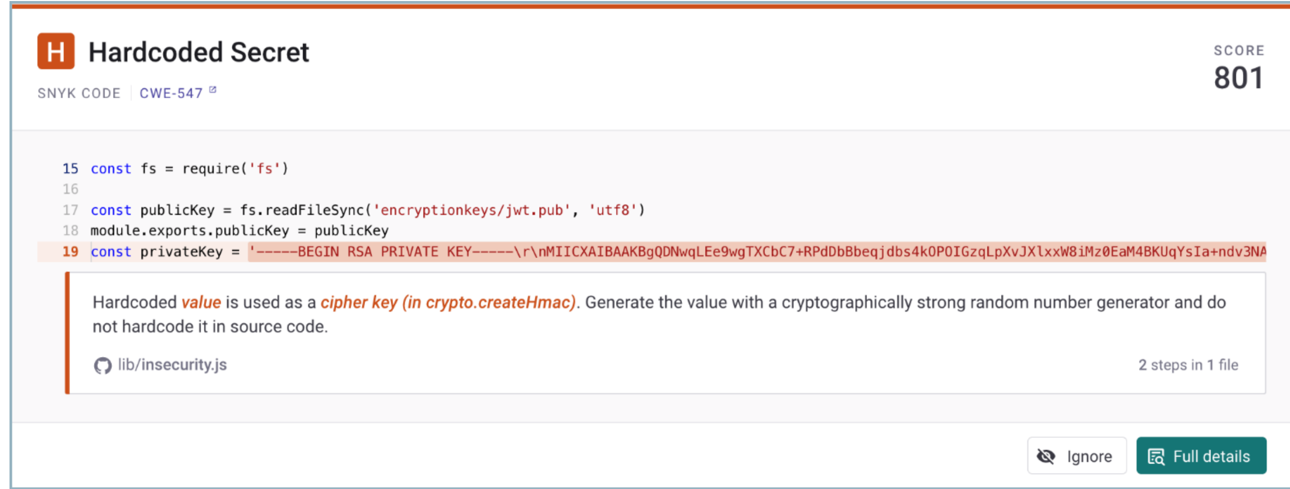


8 pav. SonarQube aptiktų rezultatų peržiūros langas [58]

Pateiktoje iliustracijoje matoma, kad šoninėje panelėje galima išskirstyti kodo trūkumus pagal tipą, rimtumą ir kitus aspektus. Taip pat, galima specifinį trūkumą pažymėti kaip neteisingai identifikuotą arba kaip netaisytiną.

1.8.3. Snyk Code statinės programų saugos analizės įrankio analizė

Snyk įmonė siūlo SAST įrankį Snyk Code [59] (dokumentacija pasiekiama [60]). Šis įrankis pasiekiamas kaip IDE įskiepis arba SCM įskiepis. Taip pat, Snyk Code galima valdyti pasinaudojant interneto naršykle. Įrankis gali peržiūrėti ir analizuoti naujai atsiradusias spragas kodo įkėlimo į versijavimo sistemą arba kodo rašymo metu. Įrankis turi integruotą dirbtinio intelekto varikliuką (9 pav.), kuris atpažįsta kodo klaidas, parametrų įrašymą sunkiai keičiamu būdu (angl. *hardcoding*), stebi programos srautų kontrolę, taip apsaugodamas nuo tokių klaidų kaip nulinės rodyklės atkodavimas (angl. *null-dereferencing*).



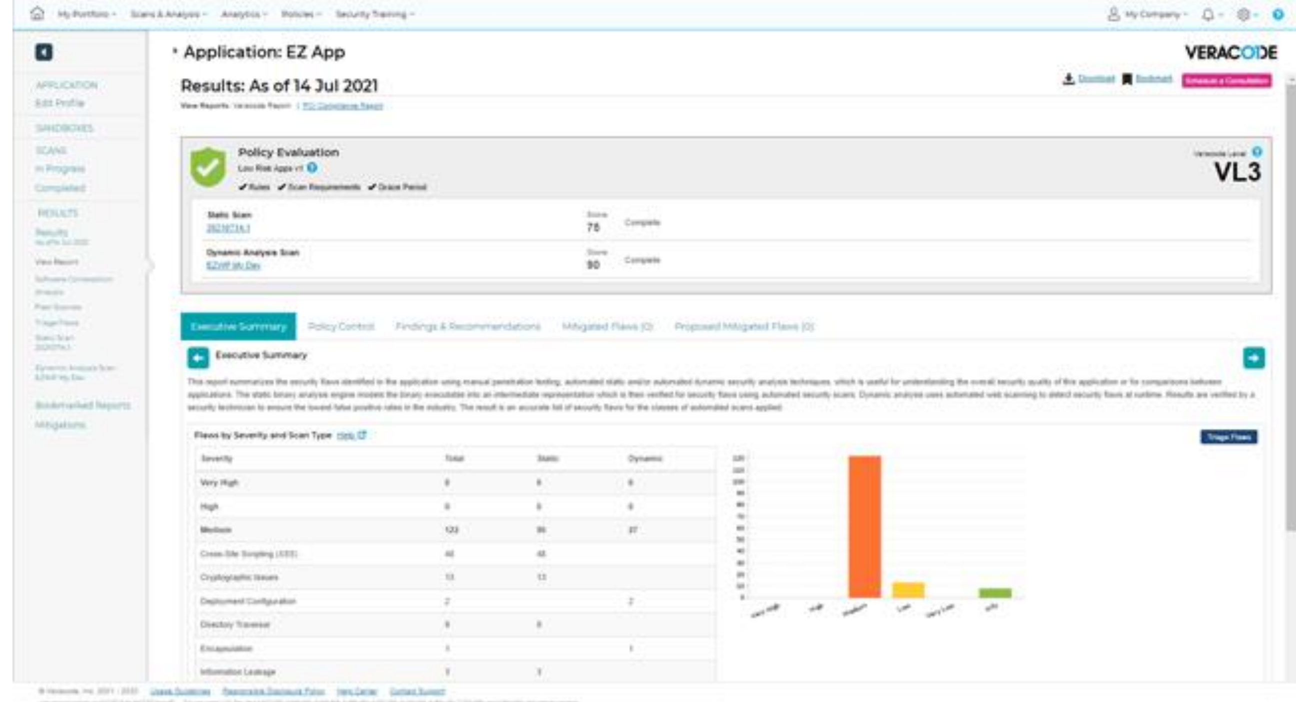
9 pav. Snyk Code AI varikliuko demonstracija [60]

Pateiktoje iliustracijoje galima matyti, kad Snyk Code įrankis palaiko CWE enumeraciją, o norint ištaisyti klaidą, suteikiama pagalba ir instrukcijos, kaip tą padaryti. Snyk Code įrankis palaiko GIT repozitorijas ir gali būti integruojamas į Jira.

1.8.4. Veracode Static Analysis statinės programų saugos įrankio analizė

Įmonė Veracode siūlo savo sprendimą statinei kodo analizei vykdyti [61] (dokumentacija pasiekiama [62]).

Šis įrankis pasiekiamas kaip platforma arba SCM įrankis ir naudoti jį galima pagal tam tikras darbo metodikas. Jeigu naudotojas turi tiesioginę prieigą prie kodo, reikalaujama įkelti zip failą su programiniu kodu kuris neviršytų 2 GB apimties. Žemiau matyti pavyzdiniai rezultatai (žr. 10 pav.).

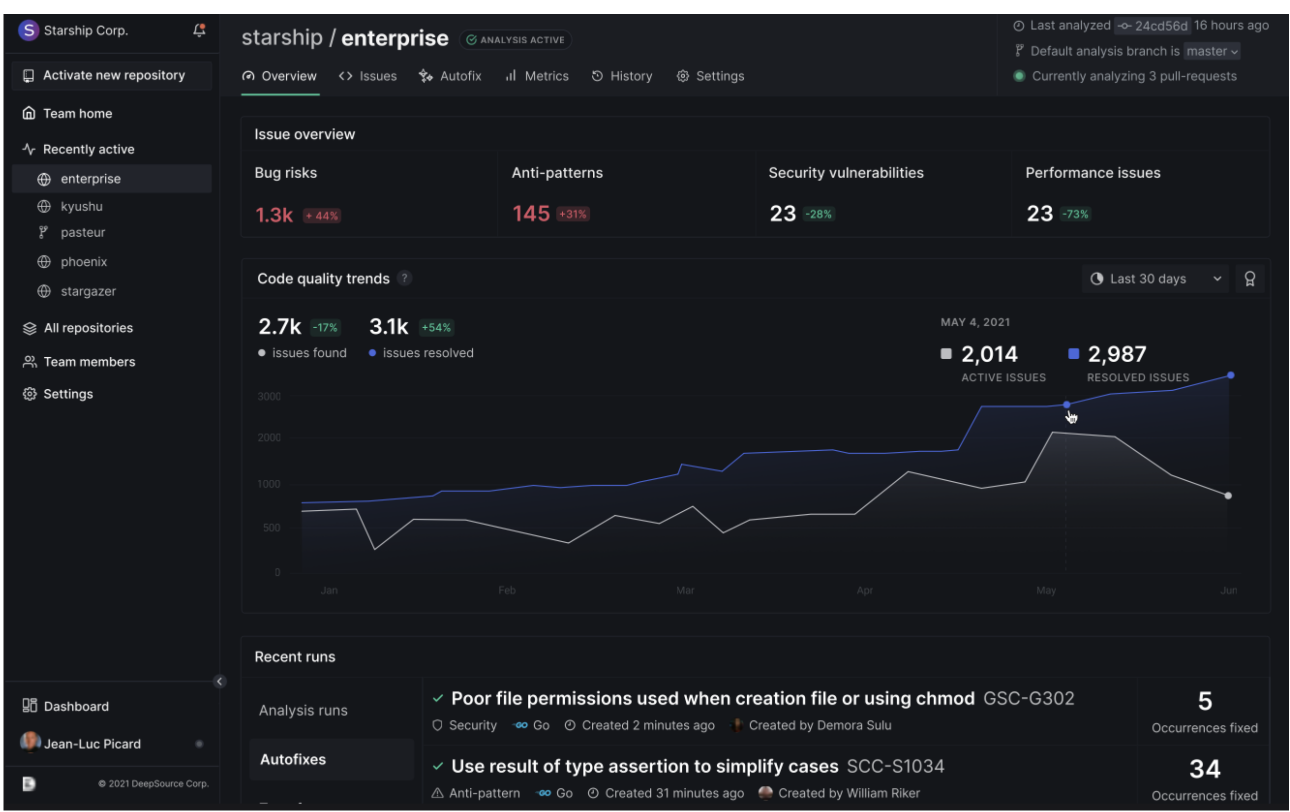


10 pav. Veracode Static Analysis įrankio kodo analizės pavyzdiniai rezultatai

Šioje iliustracijoje matome, kad sistema leidžia rikiuoti kodo pažaidas pagal atributus, matomas CWE identifikatorius. Rengiant ataskaitą, Veracode įrankis pateikia ataskaitą pagal savo aprašytą rizikų vertinimo metodiką.

1.8.5. DeepSource SAST įrankio analizė

DeepSource siūlo savo SAST įrankį [63] (dokumentacija prieinama [64]). Įrankio darbinė panelė pateikiama 11 pav.:



11 pav. DeepSource SAST įrankio pagrindinis langas

Šis įrankis suteikia galimybę kodo analizę atlikti kiekvieno kodo įkėlimo metu į versijavimo sistemą. Palaikoma GIT sistema, tačiau jokių kitokių integracijų įrankis nesiūlo. Galima naudoti įrankio API.

Esamų statinės programų saugos analizės įrankių palyginimas pateikiamas 8 lentelėje.

8 lentelė. Įrankių lyginamoji analizė

Kriterijus	„Checkmarx SAST“	„SonarQube“	„Snyk Code“	„Veracode“	„DeepSource SAST“
Svetainė ar kompiuterinė programa	Programa	Svetainė	Svetainė	Svetainė	Svetainė
Platforma	Visos	Visos	Visos	Visos	Visos
Palaikomos kalbos (skliausteliuose pateikiamos didžiausios palaikomos)	23 (Java, JS, TS, C#, PHP, Python, kt.)	20 (Java, JS, TS, C#, PHP, Python, kt.)	10 (Java, JS, TS, C#, PHP, Python, kt.)	23 (Java, JS, TS, C#, PHP, Python, Go, kt.)	7 (Java, JS, TS, C#, PHP, Python, Ruby, Go)

Pasiekiamas kaip IDE įskiepis	+	+	+	+	-
Pažaidų reitingavimas pagal tarptautinius pripažintus standartus	+	-	+	+	+
Naudoja dirbtinio intelekto metodus	+/-	-	+	-	-
Galima skenuoti periodiškai	+	-	-	-	+
GIT palaikymas	+	+	+	-	+
SVN palaikymas	+	+	-	-	-
TFS palaikymas	+	-	-	-	-
Jira integracija	+	-	+	+	-
Kaina	Viešai neprieinama informacija	$150 metams (1 žmogui)	$98 mėnesiui (1 žmogui)	Viešai neprieinama informacija	Viešai neprieinama informacija
Bandomoji versija	Netiekiama	Reikalaujama užklausa	Galima nemokama versija	Netiekiama	Galima įrankį išbandyti

Atlikus analizę buvo nuspręsta, kad galingiausias įrankis yra „Checkmarx SAST“ dėl savo suteikiamų pažangių kodo analizės ir sistemų palaikymo galimybių bei galimybės lyginti pažaidas pagal įvairius standartus, tačiau labiausiai prieinamas individualiam kūrėjui yra „SonarQube“ įrankis. Svarbiausi kriterijai šiuo metu aktualūs tyrimo sričiai yra pažaidų reitingavimas pagal tarptautinius standartus, dirbtinio intelekto naudojimas atpažinčiai ir įvairių SCM integracijos. Checkmarx SAST negavo pilno pliuso už dirbtinio intelekto metodus, nes norint naudoti šį funkcionalumą, reikia apmokyti užklausų rinkinį [65]. Plačiausią kalbų palaikymą suteikia Checkmarx SAST ir Veracode. Verta paminėti, kad Checkmarx SAST palaiko pagrinde informacinių ir internetinių sistemų kalbas, o Veracode palaiko ir truputį mažiau populiarių ir senesnių kalbų, kaip *COBOL, VB.NET*. Verta pastebėti, kad bendrai kalbant, rinkoje nėra labai daug įrankių kurie darytų kodo statinę analizę dirbtinio intelekto metodais. Veracode pagrindinis trūkumas yra tai, kad vietoj to, kad naudotų tarptautiniu lygiu pripažintus standartus, naudoja savo kurtą uždaro kodo metodologiją pažaidos sudėtingumo vertinimui, o DeepSource SAST nesuteikia intuityvaus sąrankos proceso per grafinę naudotojo sąsają.

1.9. Analizės išvados

Atlikus analizę prieita prie šių išvadų:

1. Statinės kodo analizės metodų analizės metu įvardyti pagrindiniai statinės kodo analizės principai. Išsiaiškinta, kad mokslininkai bando mažinti spragų identifikavimo klaidas įvairiais būdais, vienas iš jų yra analizuoti kodą ir ataskaitas mašininio mokymosi metodais;
2. Atlikta programų sistemų pažaidų prognostikos analizė atskleidė, kad bandyta atlikti analizę pasitelkiant duomenis iš įvairių pažaidų duomenų bazių, tačiau dažniausiai naudota yra NVD. Pastebėta tendencija imti subrinkinį iš NVD kaip duomenų rinkinį. Pastaraisiais mėnesiais pradėti taikyti didžiųjų kalbos modelių architektūromis grindžiami sprendimai stipriai nugali tradicinius

įrankius. Tačiau tai, kad pasirinkti duomenų rinkiniai neatspindi realių scenarijų, sukelia įvairių problemų, kaip pvz., ryškiai neatitinkantis tikslumas;

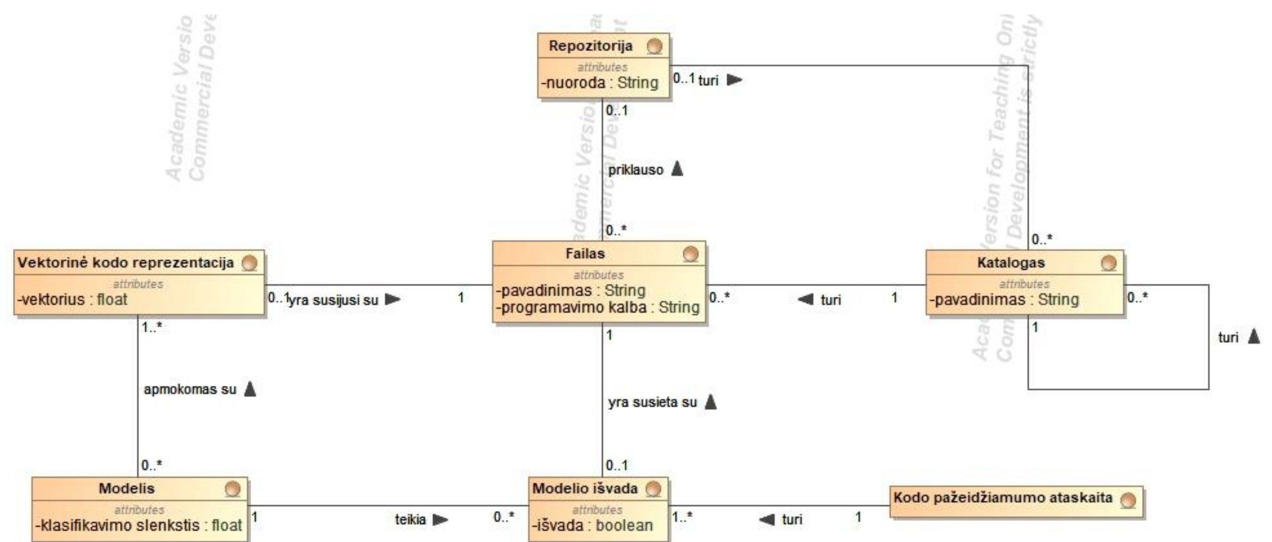
3. Atlikta statinės kodo analizės ir programų sistemų pažaidų prognostikos susietumo galimybių analizė atskleidė, kad statinės programų sistemų analizės įrankių ir mašininio mokymosi metodų susietumo sprendimų paieška yra aktuali, o iš rezultatų galima spręsti, kad yra vietos pateikti geresnį sprendimą. Kai mažinamas neteisingai identifikuotų spragų skaičius, atsiranda rizika, kad išmesime tam tikrą iš tikro naudingą ir vertingą teisingą spragos atvejį;
4. Atlikta egzistuojančių sprendimų analizė parodė, kad svarbiausi kriterijai šiuo metu aktualūs tyrimo sričiai yra pažaidų reitingavimas pagal tarptautinius standartus, dirbtinio intelekto naudojimas atpažinčiai ir įvairių SCM integracijos. Pastebėta, kad rinkoje nėra labai daug įrankių kurie darytų spragų identifikavimą dirbtinio intelekto metodais;
5. Tolimesnei darbo eigai pasirinktas duomenų rinkinys DiverseVul dėl savo realistiškumo, naujumo, standartizuoto pažaidų tipų žymėjimo CWE notacijoje ir funkcijų įvairovės. Dėl sąlyginai žemo klaidingų taikinio klasės atvejų kiekio ir naudojamo nekompiluojamo statinės analizės metodo lyginamąjai eksperimento analizei pasirinktas *Flawfinder* statinis analizatorius.

2. Programų sistemų kodo spragų aptikimo algoritmas

Šiame skyriuje aprašomas siūlomas sprendimas. Algoritmas susideda iš modelio apmokymo ir jo naudojimo dalių.

2.1. Dalykinės srities aprašas

Dalykinės srities aprašas pateikiamas esybių-ryšių diagrama (žr. 12 pav.).

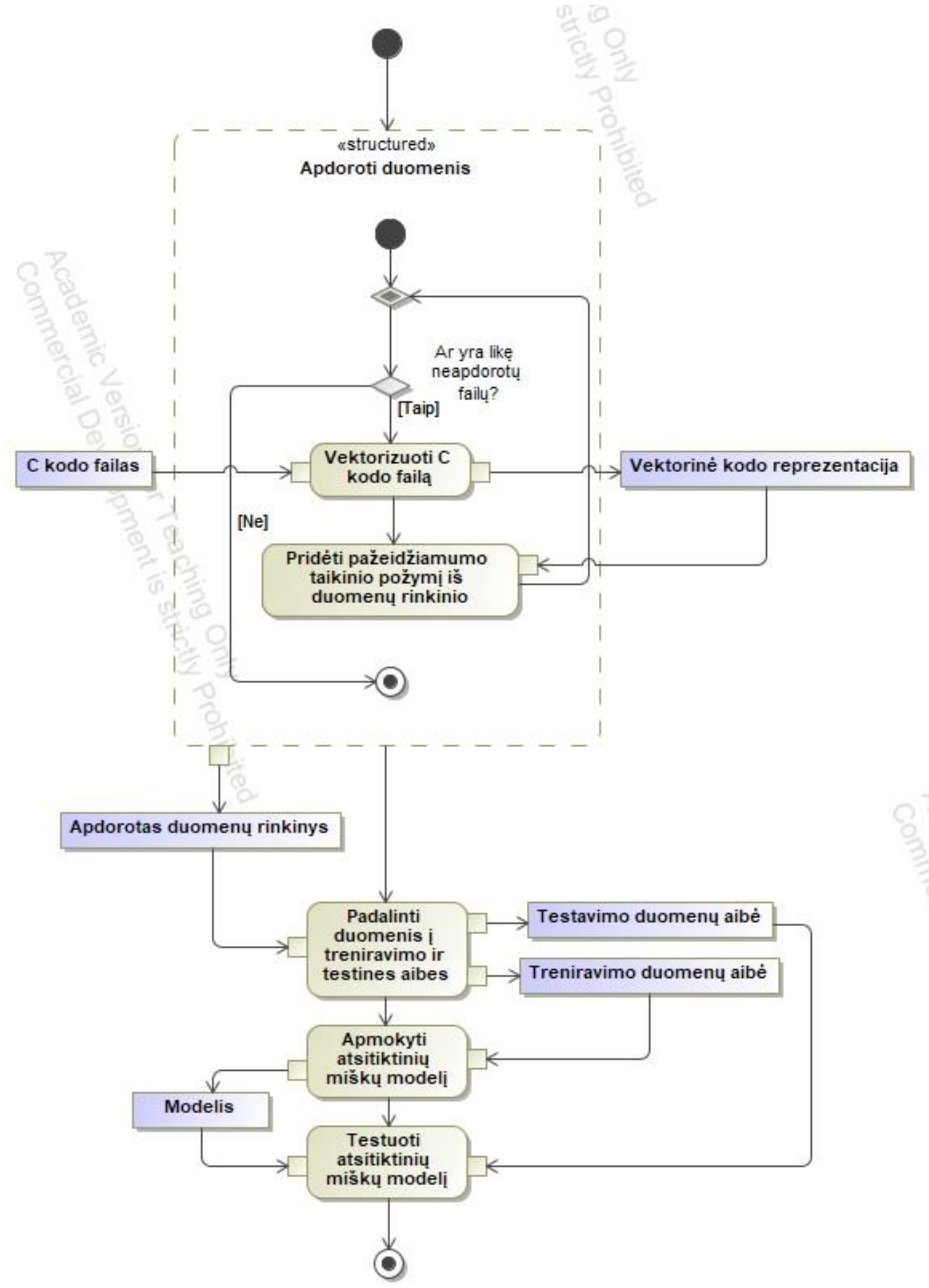


12 pav. Dalykinės srities ER modelis

Failas gali priklausyti repozitorijai. Failas būtinai priklauso katalogui. Repozitorija gali susidėti iš failų ir katalogų. Failų katalogai gali turėti vaikinius katalogus. Šie failai gali būti panaudoti modelio apmokymui. Apmokytas modelis naudodamasis vektorine kodo reprezentacija, teikia galutines modelio išvadas, kurios yra susietos su konkrečiu failu. Iš šių išvadų yra suformuojama ataskaitą.

2.2. Mašininio mokymosi modelio apmokymo etapas

Atsitiktinių miškų modelio apmokymo procesas pateikiamas UML veiklos diagrama (žr. 13 pav.).



13 pav. Modelio apmokymo procesas

Žingsniai toliau detalizuojami:

1. Apdoroti duomenis. DiverseVul rinkinys susideda iš virš 330000 .c failų. Kiekvienam iš jų taikomi šie apdorojimo žingsniai:

   - Vektorizuoti C failo kodą. Šis žingsnis vykdomas pasinaudojant Meta įmonės dirbtinio intelekto modelio CodeLlama 2 variantu, turinčiu 7 milijardus hiperparametrų. Šiame žingsnyje, kodas verčiamas į vektorinę kodo reprezentaciją, siekiant standartizuoti įvesties ilgį mašininio modelio apmokymui.
   - Pridėti pažeidžiamumo taikinio požymį iš duomenų rinkinio. Originaliame duomenų rinkinyje specifikuojamas taikinio (angl. target) atributas kiekvienam failui, kurio reikšmė yra viena iš dviejų: 0, kuri reiškia, kad faile pažeidžiamumų nėra ir 1, kuri reiškia, kad faile pažeidžiamumų rasta.

2. Padalinti duomenis į testavimo ir testinę aibes. Apdorotas duomenų rinkinys, padalijamas į treniravimo ir testavimo dalis 80%/20% proporcingumu. Dalijimas vykdomas stratifikuotai pagal *CWE* atributą, t.y. išlaikant proporcijas tarp šių klasių:

   - CWE-0: Sveiki failai.
   - CWE-X: Duomenų rinkinyje failo taikinio atributas pažymėtas kaip pažeidžiamas, bet CWE identifikatorius nenurodytas.
   - CWE-119: CWE-119 identifikatoriumi pažymėti failai.
   - CWE-125: CWE-125 identifikatoriumi pažymėti failai.
   - CWE-20: CWE-20 identifikatoriumi pažymėti failai.
   - CWE-787: CWE-787 identifikatoriumi pažymėti failai.
   - CWE-Other: dirbtinai sudaryta klasė, kurią sudaro kitais, nepatenkančiais į aukščiau išvardytas klases, pažeidimais pažymėti failai.

3. Apmokyti atsitiktinių miškų modelį. Apmokyti modeliai, jų specifikacija ir pasiekti rezultatai detalizuojami 3 skyriuje;
4. Testuoti apmokytą modelį. Tikrinamas modelio veiksmingumas su nematytais duomenimis.

Naudojami darbo metodai detalizuojami kituose šio skyriaus poskyriuose.

2.3. Duomenų rinkinio parengimas

Spragų aptikimo programiniame kode modeliui apmokyti pasitelktas duomenų rinkinys DiverseVul (straipsnis pasiekiamas [50], rinkinys pasiekiamas [66]). Duomenų rinkinio *JSON* failo struktūra detalizuojama 9 lentelėje.

9 lentelė. Duomenų rinkinio eilutės (angl. *entry*) struktūra

Atributas	**Atributo reikšmė**
Func	Failo kodas
Target	Taikinio atributas
Cwe	Pažeidžiamumų sąrašas pagal CWE notaciją
Project	Projektas, kuriam failas priklauso
Commit_id	Failo pakeitimo versijavimo sistemoje identifikatorius

Hash	Failo šifras
Size	Simbolių kiekis faile
Message	Specialios pastabos nuo autorių apie failą

Lentelėje matomi atributai tegu ir suteikia gilesnės informacijos apie pateikiamą kodą, tačiau dauguma laukų, kaip kodo priklausymas konkrečiam projektui ar failo šifras yra neaktuali kodo analizei informacija ir jos galima atsisakyti apdorojant duomenų rinkinį. Paanalizavus rinkinį, pastebėta, jog pažeidžiamumų tipų duomenų rinkinyje numatyta 155. Detalesnė duomenų rinkinio sandara pagal pažeidžiamumo statusą ir CWE pažeidžiamumo tipą pateikiama.

10 lentelė. Duomenų pasiskirstymas pagal taikinio atributus

Target **atributas**	***CWE*** **atributas**	**Procentinė failų kiekio nuo viso duomenų rinkinio dalis**
0	Netaikoma	94.5%
1	CWE-119	0.4%
1	CWE-125	0.47%
1	CWE-787	0.47%
1	CWE-20	0.34%
1	Kitos reikšmės	3.82%

Verta pastebėti, kad rinkinys labai netolygiai pasiskirstęs (t.y. labai didelė dalis reikšmių turi *target* 0 atributą). Pastebima dar kitų duomenų rinkinio problemų. Jos pasiūlytos išspręsti 11 lentelėje.

11 lentelė. Duomenų rinkinyje aptiktos problemos ir jų sprendimai

Problema	**Sprendimas**
Kai kurių failų *target* atributas pažymėtas 1, tačiau pažeidžiamumų pagal CWE notaciją nepateikta	Sukurti atskirą atributą „cwe“ ir pažymėti failus kategorija CWE_X, *target* atributą paliekant kaip taikinio (t.y. 1)
Platus CWE kategorijų spektras	Išskirti keturias didžiausias pažeidžiamumų klases (atributas „cwe“), likusias sugrupuoti į CWE_Other klasę (atributas „cwe“)
Kai kurių failų *target* atributas pažymėtas 0, tačiau pažeidžiamumai pagal CWE notaciją visvien pateikiami	Pažymėti failus kaip netaikinio klasės (t.y., *target* atributas 0)
Dėl skirtingo ir plataus pažeidžiamumų pobūdžio, kyla pavojus dėl perduodamų duomenų klasių nevienodumo padalinant duomenis pagal *target* atributą	Atlikti stratifikuotą duomenų padalijimą ne pagal *target* atributą, o pagal naujai sukurtą *cwe* atributą
Modeliams apmokyti reikia standartinio ilgio įvesties	Vektorizuoti kodo failus, gaunant vienodo ilgio skaičių vektorių

Kadangi atliekamos *cwe* atributo transformacijos, kaip matoma iš lentelės, žemiau lentelėje pateikiamos atliktos šio atributo transformacijos (žr. 13 lentelė). Transformacijos buvo atliekamos, atsižvelgiant į *target* ir *cwe* reikšmę. Siekiant išvengti duomenų dublikavimo, užtikrinama, kad esant keliems aptiktiems *cwe* pažeidžiamumams, transformuojama pagal pirmą, atitinkantį kriterijus.

12 lentelė. Atliktos *cwe* atributo duomenų rinkinyje transformacijos

Target **atributo reikšmė**	***Cwe*** **atributo pirminiame duomenų rinkinyje reikšmė**	**Naujai priskirta *cwe* atributo reikšmė**
0	Tuščia	CWE-0
0	Netuščia	CWE-X
1	CWE-119	CWE-119
1	CWE-20	CWE-20
1	CWE-120	CWE-120
1	CWE-787	CWE-787
1	Kitos bet kokios reikšmės	CWE-Other

Žemiau detalizuojami žingsnių apibendrinimas, kurie buvo atlikti su duomenų rinkinio *JSON* failu:

1. Iš duomenų rinkinio išrinkti atributus *func, target* ir *cwe*:
2. *Func* atributo reikšmes išsaugoti kaip atskirus C failus, užvadinant juos eiliškumo tvarka, pradedant nuo 0. Šia eiliškumo tvarka išsaugoti *target* ir *cwe* atributus į atskirą *csv* failą;
3. Gauti kodo failų vektorines reprezentacijas, pasinaudojant *CodeLlama* 2 didžiuoju kalbos modeliu (žr. 2.4 poskyrį);
4. Atlikti *cwe* atributo transformaciją pagal 12 lentelėje numatytas korekcijas;
5. Gautas kodo failų vektorines reprezentacijas sujungti pagal eiliškumo tvarką su *target* ir *cwe* atributais.

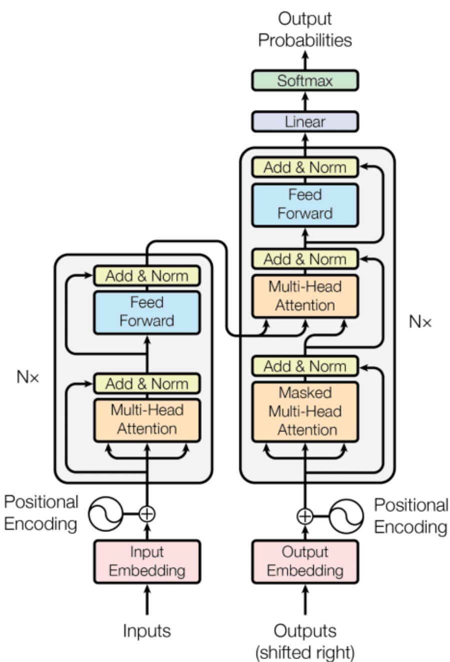
2.4. Kodo vektorizavimas didžiaisiais kalbos modeliais

Siūlomam artefaktui pateikiami duomenys, apdoroti *CodeLlama* 2 didžiuoju kalbos modeliu, išgaunant kodo vektorius (angl. *code embedding*). Kodo vektorius – vektorinė kodo reprezentacija, kuri enkapsuliuoja sintaksinę ir semantinę informaciją. Šie kodo vektoriai naudingi kodo paieškai [67] ir automatinei kodo peržiūrai [68].

2.4.1. Didieji kalbos modeliai

Didysis kalbos modelis (angl. *large language model*) yra pažangus mašininio mokymosi modelis, sukurti suprasti ir generuoti žmonių kalbą. Šie modeliai naudoja transformatorių architektūras aukštos raiškos teksto apdorojimui ir produkavimui [69].

Transformatoriaus (angl. *transformer*) [70] architektūros naudojimas paplito tarp moderniausių natūraliosios kalbos apdorojimo modelių, kaip GPT, BERT ir jų variacijose. Transformatoriaus pagrindinė esmė yra gebėjimas naudoti savišifravimo (angl. *self-attention*) mechanizmus sudėtingų duomenų santykių ir priklausomybių aptikimui. Transformatoriaus architektūra pavaizduota 14 pav.



14 pav. Transformatoriaus architektūra [70]

Įvesties apdorojimas prasideda poziciniu užkodavimu (angl. *positional encoding*). Kadangi transformatorius neaptinka žodžių eilės savaime, užkoduotos sinusinėmis ir kosinusinėmis funkcijomis vertės pridedamos prie įvesties, siekiant išlaikyti vientisą seką.
Savišifravimo mechanizmas leidžia modeliui paskaičiuoti skirtingų žodžių svarbą sakinyje, nepaisant jų pozicijos sakinyje. Tą atlieka kelių dėmesingumo funkcijų (angl. *multi-head attention*) sluoksnis. Pagrindiniai skaičiavimai savišifravimui yra pateikiami 1 formulėje.

$$Attention(Q, K, V) = softmax\left(\frac{QK^t}{\sqrt{d_k}}\right)V \tag{1}$$

Čia Q, K, V yra užklausų, raktų ir verčių matricos atitinkamai, o d_k yra raktinių vektorių dimensionalumas, o *softmax* yra normalizuota eksponentinė funkcija (žr. 2 formulę).

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2}$$

Čia x – įvesties vektorius, susidedantis iš n elementų. x_i – i-tasis vektoriaus elementas.

2.4.2. LLaMA modelio architektūra

Visi šiame apraše aptariami *CodeLlama* šeimos modeliai yra paremti LLaMA (angl. *Large language model Meta AI*) [71] architektūra ir yra jos specializuotas variantas, pritaikytas kodui apdoroti. LLaMA architektūra remiasi aukščiau aprašyta transformatoriaus architektūra su šiais kūrėjų pagrindiniais įgyvendintais pakeitimais:

- Normalizuojamos kiekvieno transformatoriaus posluoksnio įvestys, o ne bendra išvestis;
- SwiGLU aktyvacijos funkcija vietoj ReLU;
- Vietoje absoliutaus pozicinio kodavimo, naudojamas rotorinis pozicinis kodavimas.

Visi šie pakeitimai padeda Llama šeimos modeliams pasiekti didesnį stabilumą ir rezultatyvumą.

2.4.3. Vektorinių kodo reprezentacijų išgavimas

Duomenims apdoroti ir toliau aptariamam eksperimentui buvo naudoti du *CodeLlama* variantai: 7B (7 milijardų parametrų) ir 13B (13 milijardų parametrų). Šie modeliai buvo atsisiųsti iš Meta AI puslapio ir išsaugoti lokaliai. Kadangi modelių tiesiogiai nebuvo galima naudoti vektorinių kodo reprezentacijų išgavimui, juos reikėjo pirma paversti į *GGML* suderinamą formatą. Tam pasitelktas programinis paketas *llama.cpp* [72]. Atlikti žingsniai eiliškumo tvarka:

- Klonuojama *llama.cpp* direktorija;
- Sukompiliuojamas paketas su komanda *make LLAMA_CUDA=1*. Čia *LLAMA_CUDA* dalis reiškia, kad bus naudojama *Nvidia CUDA* architektūra;
- Įkeliami norimi konvertuoti modeliai į */models* direktoriją;
- Kviečiama komanda *python convert.py {modelis}*, nurodant kelią iki modelio.

Modelio konversijai įvykus, jį galima dar kvantizuoti (t.y. sumažinti modelio svorių ir aktyvacijų tikslumą nuo didesnio bitų skaičiaus iki mažesnio bitų skaičiaus, taip sumažinant reikalavimus atminčiai), tačiau šiam sprendimui tai nebuvo daroma.

Vektorinėms reprezentacijoms gauti tada pasitelktas *LangChain* karkasas ir *llama-cpp-python* [73] biblioteka. Su šiomis bibliotekomis modeliai buvo užkrauti į atmintį ir pasinaudojant *embed_documents* funkcija, failų turinys verčiamas į vektorinę reprezentaciją. Šios vektorinės reprezentacijos vėliau gali būti naudojamos iškart spėjimui arba išsaugomos modelio treniravimui į *csv* failus.

2.5. Atsitiktiniai miškai

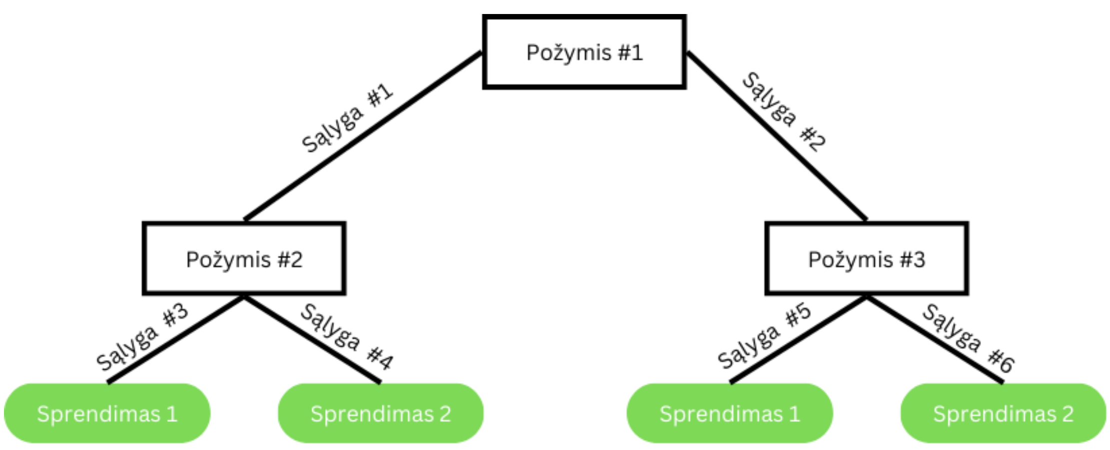
Siūlomam artefaktui realizuoti pasitelkiami atsitiktinių miškų (angl. *random forest*) modeliai. Atsitiktinių miškų modelis [74] yra ansamblinis mašininio mokymosi metodas, naudojamas klasifikacijos ir regresijos uždaviniams spresti. Atsitiktiniai miškai veikia sukonstruodami kelis sprendimo medžius ir suskaičiuodami arba visų sprendimų modą klasifikavimo uždaviniams arba sprendimų vidurkį regresijos uždaviniams.

2.5.1. Sprendimų medis

Pagrindinis atsitiktinio miško vienetas yra sprendimų medis (angl. *decision tree*). Sprendimų medis yra tarsi sprendimų grandinėlė kur:

1. Mazgai reprezentuoja duomenų rinkinio požymius;
2. Kraštinės tarp mazgų reprezentuoja sprendimo taisykles;
3. Terminalinės galūnės reprezentuoja spėjimus, kas gali būti arba klasė arba regresinė vertė.

15 pav. pateikiamas toks sprendimų medžio supaprastintas pavyzdys.



15 pav. Binarinės klasifikacijos sprendimo medžio pavyzdys

Šiuo sprendimų medžiu einant kiekviename žingsnyje prieinama prie galutinio sprendimo. Kiekviename mazge algoritmas įvertina visus galimus požymius ir nusprendžia, kuriuo požymiu remiantis, geriausia padalinti duomenis mazge. Šio žingsnio tikslas – surasti tokius požymius kiekvienam mazgui, kuris geriausiai dalina duomenis į atskiras klases (klasifikacijai) arba minimizuoja paklaidą (regresijai).

Šis dalijimas klasifikacijos uždaviniui spręsti gali būti vykdomas remiantis Gini-Simpsono indeksu arba Kullback-Leibler divergencija.

Gini-Simpsono indeksas yra matas, matuojantis dažnį, kuriuo bet kuris elementas bus suklasifikuotas neteisingai, atsitiktinai pasirinkus.

$$G = 1 - \sum_{i=1}^{C} p_i^2 \tag{3}$$

Čia p_i - tikimybė, kad duomenų vienetas bus klasifikuotas kaip i klasei priklausantis objektas iš C klasių.

Kullback-Leibler divergencija (kitaip vadinama informacijos padidėjimu (angl. *Information gain*)) parodo entropijos sumažėjimą, duomenų rinkinį padalinus, remiantis duomenų rinkinio požymiu. Kullback-Leibler divergencijos formulė pateikiama (4), o entropijos formulė pateikiama (5).

$$IG(D) = E(D) - \sum_{j=1}^{k} \frac{|D_j|}{|D|} E(D_j) \tag{4}$$

Čia D – duomenų rinkinys, D_j – padalintas duomenų rinkinys, remiantis j-tuoju atributu.

$$E = \sum_{i=1}^{C} p_i \log_2(p_i) \tag{5}$$

Čia p_i - elementų, esančių i klasėje proporcija rinkinyje.

2.5.2. Ansamblinis mokymasis

Ansamblinis mokymasis yra mašininio mokymosi paradigma, kuria naudojantis, keli mažesni modeliai sugrupuojami ir apmokomi spręsti tą pačią problemą. Pagrindinė mintis, kuria grindžiamas ansamblinis mokymasis yra tai, kad mažų modelių grupė yra stipresnė už vieną individualų modelį.

Egzistuoja keli modeliai, kuriais sukuriami modelių ansambliai. Specifiškai atsitiktiniuose miškuose yra naudojamas savirankos agregavimo (angl. *bootstrap aggregating*) metodas, apie kurį toliau ir bus rašoma.

Savirankos agregavimo procesas pradedamas sukuriant duomenų poaibius, kurie gali turėti pasikartojančius elementus (angl. *bootstrap sampling*) arba tarpusavyje nepasikartojančius elementus (angl. *pasting*). Kadangi siūlomame sprendime naudojamas pastarasis variantas, apie jį ir rašoma toliau:

1. Tegu egzistuoja pirminis duomenų rinkinys D, o n – išrinktų elementų kiekis.
2. Sukuriama B poaibių, kur $|B| = n$, atsitiktinai parenkant elementus be pakeitimo. Kiekvienas poaibis žymimas $D_b, b \in (1,2, \dots, B)$, kiekvienas jų yra atskirti tarpusavyje.
3. Skirtingiems medžiams apmokyti naudojami skirtingi poaibiai. Apmokomas modelis M_b poaibio D_b duomenimis.
4. Kiekviename sprendimo medžio mazge išrenkama m požymių iš p požymių ir naudojamas tik šis poaibis geriausio padalinimo nustatymui pagal aukščiau aprašytas formules.

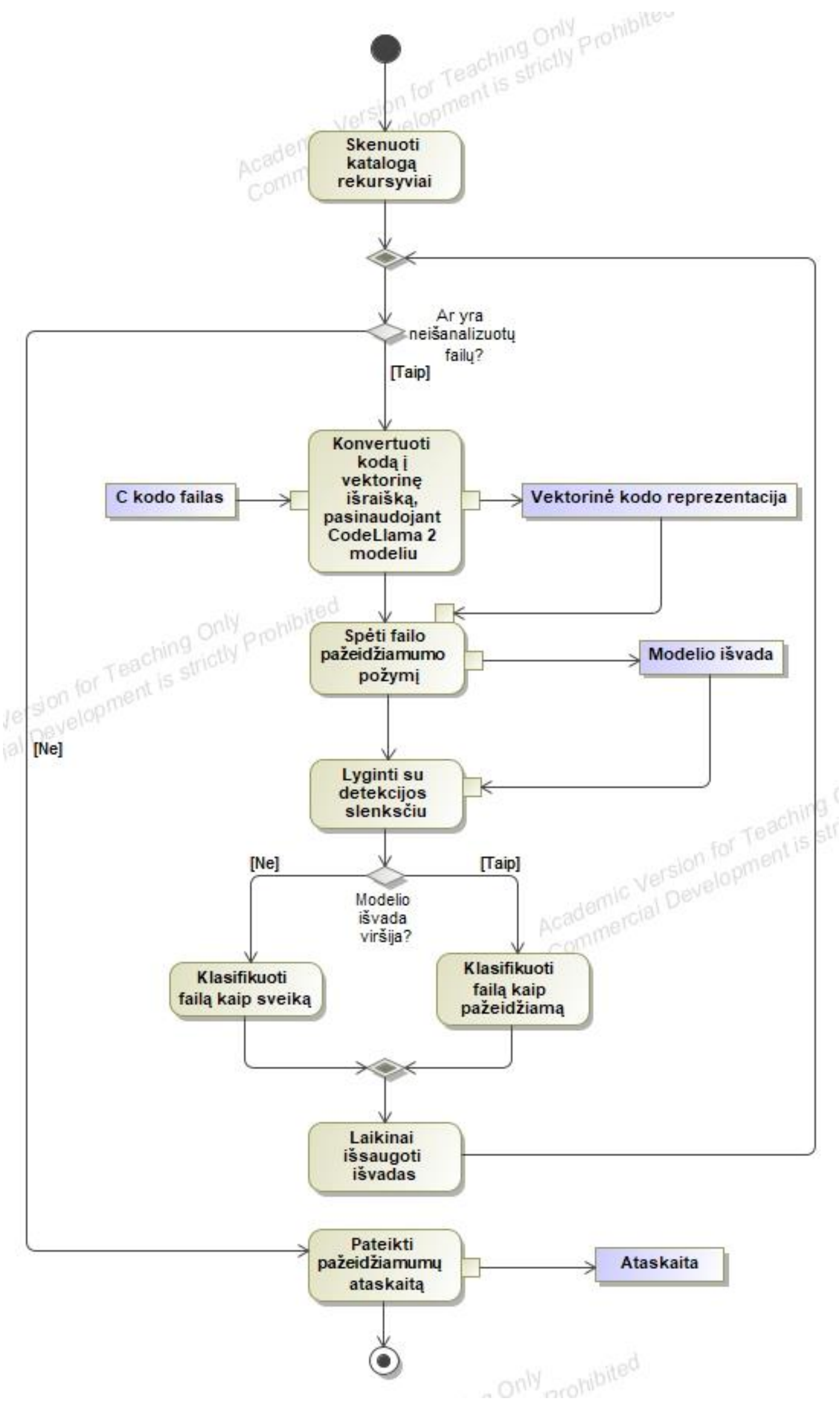
Kai modeliai apmokyti, jų sprendimai agreguojami. Klasifikacijos uždavinyje, modeliai „balsuoja“ už klasę ir klasė, kuri gauna daugumą balsų ir yra ansamblio spėjama klasė. Šis spėjimas aprašomas (6) formule.

$$\hat{y} = argmax(\{M_i(x)\}), i \in \{1,2, \dots, B\} \tag{6}$$

Čia M(x) – modelio spėjančioji funkcija.

2.6. Algoritmo naudojimo etapas

Siūlomas sprendimas yra pažeidžiamumų programiniame kode aptikimo algoritmas, kuris aprašomas UML veiklos diagrama (žr. 16 pav.).



Academic Version for Teaching Only
Commercial Development is strictly Prohibited

16 pav. Algoritmo vykdymo žingsniai su apmokytu modeliu

Šioje diagramoje aprašomi žingsniai:

1. Programos įvestis: aplanko reliatyvi direktorija nuo programos vykdymo;
2. Rekursyviai skenuojamas šakninis katalogas ieškant „*.c" failų kur * - failo pavadinimas;
3. Kiekvienam failui, rastam kataloguose, taikomi šie duomenų apdorojimo žingsniai:

   - Konvertuoti kodą į vektorinę išraišką pasinaudojant CodeLlama 2 modeliu. Įvestis: C failo turinys. Išvestis: skaitinių reikšmių vektorius, kurio dydis (4096, 1);
   - Spėti failo pažeidžiamumo požymį. Naudojamas apmokytas ankstesniame etape atsitiktinių miškų modelis. Įvestis: vektorius, kurio dydis yra (4096, 1). Išvestis: slankaus kablelio skaičius, nusakantis failo buvimo pažeidžiamu tikimybę;
   - Lyginti su detekcijos slenksčiu. Modelio išvada lyginama su detekcijos slenksčiu ir jei išvada viršija slenkstį, failas žymimas kaip pažeidžiamas, jei ne, tai kaip sveiką.

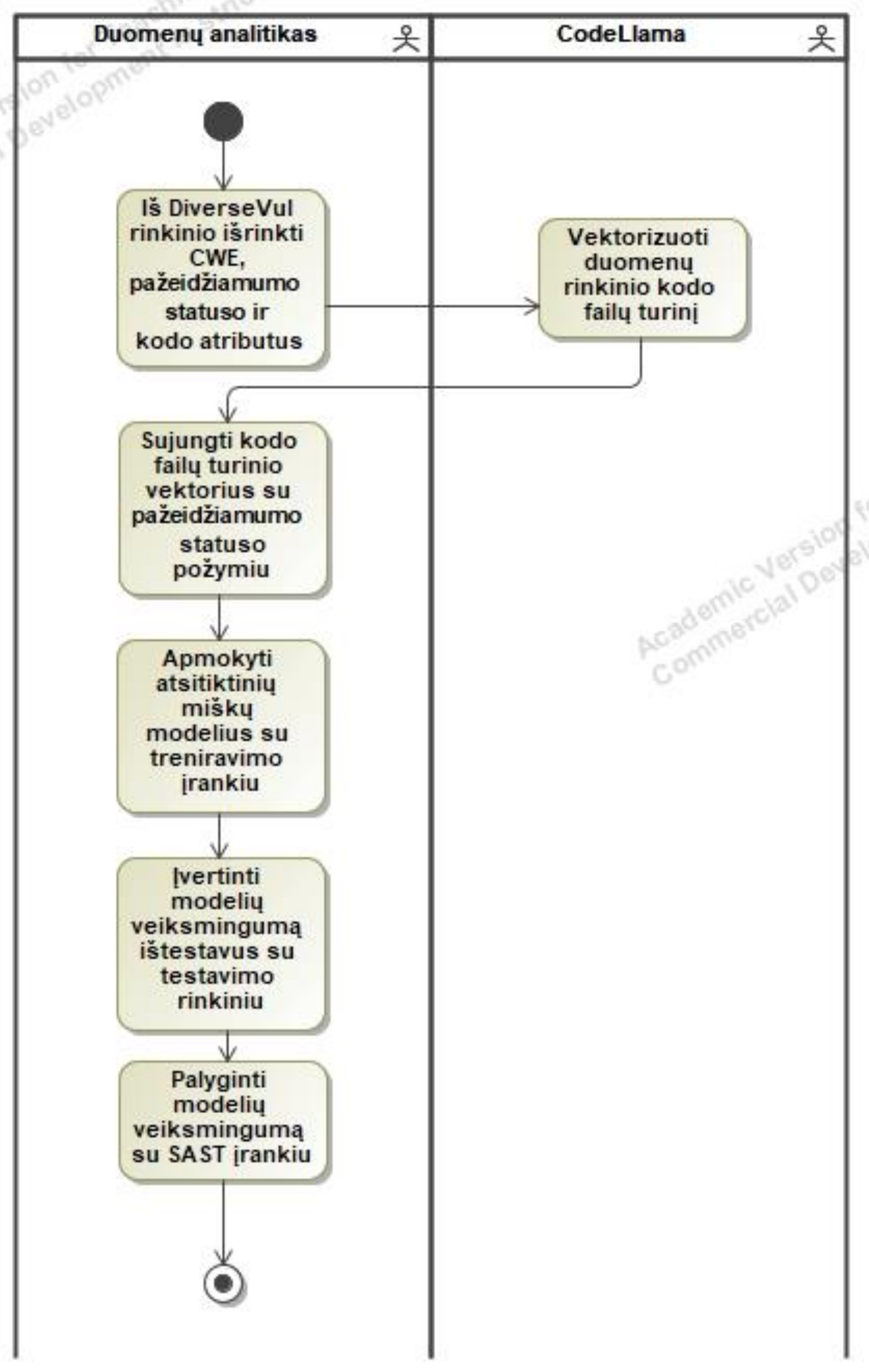
4. Pateikti pažeidžiamumų ataskaitą. Ataskaitoje pateikiama: failo pavadinimas, modelio išvada.

3. Eksperimentinis pažeidžiamumų programiniame kode aptikimo modelio tyrimas

Šiame skyriuje aprašomas vykdyto eksperimentinio tyrimo planas, jo rezultatai, pateikiamos taikymo rekomendacijos.

3.1. Eksperimento planas

Planuojamas eksperimentinis pažeidžiamumų aptikimo modelio tyrimas susideda iš kelių žingsnių, kurie detalizuojami UML 2.5 veiklos diagrama (žr. 17 pav.).



17 pav. Eksperimentinio tyrimo plano veiklos diagrama

Šio eksperimentinio tyrimo žingsnių paaiškinimas:

1. Iš DiverseVul rinkinio išrinkti CWE, pažeidžiamumo statuso ir kodo atributus. DiverseVul rinkinio struktūra susideda iš tokių atributų, kaip autorius, kodo įkėlimo į repozitoriją operacijos (angl. commit) ID. Šie atributai neturi reikšmingos vertės failo pažeidžiamumo statuso vertinimui. Dėl šios priežasties išrinkti atributai tik tie, kurie turi tiesioginę įtaką nusprendimui, ar failas pažeidžiamas, ar ne;
2. Vektorizuoti duomenų rinkinio failų turinį. Šiam žingsniui pasinaudota dvejais CodeLlama modelio variantais – su 7 milijardais ir su 13 milijardų parametrų. Skirtingais CodeLlama modeliais gauti vektorių rinkiniai naudojami skirtingų atsitiktinių miškų modelių apmokymui.
3. Sujungti kodo failų turinio vektorius su pažeidžiamumo statuso požymiu. Šiuo žingsniu gaunamas vientisas sužymėtas duomenų rinkinys, kurį galima naudoti modelių apmokymo reikmėms. Sujungimas vykdomas pagal failo ID;
4. Atlikti stratifikuotą pagal CWE duomenų rinkinio padalinimą į treniravimo ir testavimo rinkinius. Rinkinys dalijamas 80/20 proporcijos pagrindu, išsaugojant tas pačias CWE pažaidų proporcijas, siekiant, kad modelis būtų apmokytas su kiek įmanoma platesne duomenų aibe.
5. Apmokyti atsitiktinių miškų modelius su treniravimo rinkiniu. Naudojamų atsitiktinių miškų parametrai specifikuojami šiame skyrelyje žemiau;
6. Įvertinti modelių veiksmingumą ištestavus su testavimo rinkiniu. Modelio veiksmingumas vertinamas pagal žemiau šiame poskyryje specifikuotas metrikas;
7. Palyginti modelių veiksmingumą su SAST įrankiu. Modelių veiksmingumas lyginamas su Flawfinder įrankio gautais rezultatais. Iš abiejų testavimo rinkinių rezultatų gautos sumaišymų matricos naudojamos įvertinti modelio tikslumą, taiklumą, jautrumą ir specifiškumą.

Iš SAST įrankių palyginimui pasirinktas *Flawfinder*, nes šis įrankis suteikia pateikiamoje statinio kodo analizės ataskaitoje CWE identifikatorius – taip pat kaip ir duomenų rinkinyje.

Kaip minėta aukščiau, kodo vektorizavimui pasitelkti du *CodeLlama* 2 modelių variantai: turintis 7 milijardus hiperparametrų ir turintis 13 milijardų hiperparametrų. Šie kalbos modeliai ir jų naudojimas toliau atitinkamai žymimi 7B ir 13B. Verta pastebėti, kad šie modeliai grąžina skirtingo ilgio vektorius – 7B grąžina 4096 ilgio reikšmių vektorių, o 13B grąžina 5120 ilgio reikšmių vektorių.

Atsitiktinių miškų modeliai pasirinkti keturi. Jie skiriasi priimamu atributų skaičiumi ir perduodamų atributų medžiui skaičiumi. Perduodamų atributų medžiui skaičiaus skaičiavimo strategijos pasirinktos dvi: logaritmuojant 2 baze bendrą atributų skaičių ir traukiant kvadratinę šaknį iš atributų skaičiaus. Modelių specifikacijos pateiktos 13 lentelėje.

13 lentelė. Eksperimentiniame tyrime naudotų modelių parametrai

Modelio pavadinimas	**Medžių skaičius**	**Priimamų įvesties atributų skaičius**	**Atsitiktinės atributų saujos dydis**
7b_log2	256	4096	12
7b_sqrt	256	4096	64
13b_log2	256	5120	12
13b_sqrt	256	5120	72

Modelių testavimas ir lyginimas su kitais modeliais vykdomas atsižvelgiant į žemiau pateikiamas charakteristikas.

Modelio spėjimo tikslumas apibrėžiama kaip teisingo teigiamos klasės spėjimo (angl. *true positive, TP*) ir teisingo neigiamos klasės spėjimo kiekių sumos santykis su visu duomenų aibės kiekiu. Teisingo teigiamos, teisingo neigiamos, neteisingo neigiamos ir neteisingo teigiamos klasių spėjimų kiekiams suskaičiuoti pasitelkiami tokie įrankiai kaip sumaišymo matrica (angl. *confusion matrix*). Sumaišymų matricos pavyzdys pateikiamas 14 lentelėje.

14 lentelė. Sumaišymų matricos pavyzdys

Visa duomenų imtis (TP+TN+FP+FN)		**Klasifikatoriaus nustatyta reikšmė**	
		Neigiama (netaikinio) klasė	**Teigiama (taikinio) klasė**
Faktinė reikšmė	**Neigiama (netaikinio) klasė**	Teisingas neigiamos klasės spėjimas (TN)	Neteisingas teigiamos klasės spėjimas (FP)
	Teigiama (taikinio) klasė	Neteisingas neigiamos klasės spėjimas (FN)	Teisingas teigamos klasės spėjimas (TP)

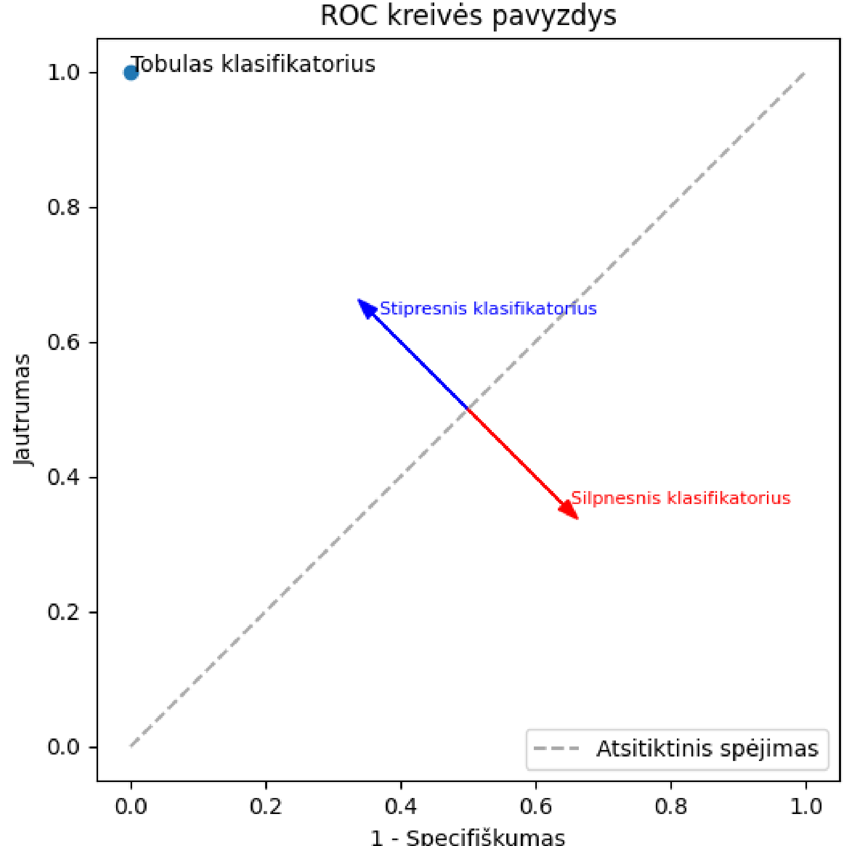
Lentelėje matoma, kad suprognozuotos klasifikatoriaus reikšmės gali būti skirstomos į keturias kategorijas. Teisingos teigiamos ir neigiamos klasių reikšmės reiškia, kad klasifikatoriaus nustatyta reikšmė ir faktinė reikšmė sutampa, o neteisingos teigiamos ir neigiamos klasių reikšmės reiškia, kad klasifikatoriaus nustatyta reikšmė yra priešinga faktinei reikšmei. Modelio tikslumas gali būti išreikštas (7) formule:

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN} \qquad (7)$$

Preciziškumas yra klasifikatoriaus matas, rodantis, kiek iš visų atpažįstamų kaip taikinio klasės atvejų iš tikro yra tokie. Šis matas yra svarbus, siekiant identifikuoti taikinio klasės klasifikavimo stiprumą. Dalykinės srities kontekste tai reikštų statistinį santykį, kiek iš failų identifikuotų kaip pažeidžiamų, iš tikro tokie yra. Šio rodiklio formulė pateikiama (8).

$$Precision = \frac{TP}{TP+FP} \qquad (8)$$

Plotas po sprendimus priimančiojo kreive (angl. *Area under receiver operating characteristic curve, AUC ROC*) yra matas, nusakantis modelio gebėjimą diskriminuoti tarp dviejų skirtingų klasių. ROC kreivės pavyzdys pateikiamas 18 pav.



18 pav. ROC kreivės pavyzdys

ROC kreivė nusako klasifikatoriaus santykį tarp modelio jautrio (angl. *sensitivity*) ir specifiškumo (angl. *specificity*). Jautrumas matuoja, kiek iš teigiamos klasės imties mėginių klasifikatorius teisingai suklasifikavo kaip tokius, o specifiškumas nusako, kiek iš neigiamos klasės imties mėginių klasifikatorius teisingai suklasifikavo kaip tokius. Jautrumas yra svarbus rodiklis, kai dalykinė modelio sritis reikalauja aukšto teigiamos klasės klasifikavimo stiprumo, o specifiškumas yra svarbu, siekiant nustatyti neigiamos klasės klasifikavimo stiprumą. Jautrumas ir specifiškumas šiame eksperimente yra svarbūs tiek kaip individualūs rodikliai, tiek kaip ROC kreivės komponentai, kadangi atliktos spragų analizės kokybė tiesiogiai priklauso nuo teisingai identifikuotų spragų ir sveikų failų. Šių dviejų rodiklių formulės pateikiamos (9, 10).

$$Sensitivity = \frac{TP}{TP+FN} \quad (9)$$

$$Specificity = \frac{TN}{TN+FP} \quad (10)$$

AUC ROC rodiklis yra vertinamas intervale nuo 0 iki 1, kur 0.5 įvertis indikuoja atsitiktinį spėjimą, o 1 įvertis rodo modelio tobulą gebėjimą atskirti klases. Kuo aukštesnis šis rodiklis, tuo geriau. Geri modelio rezultatai svyruoja nuo 0.8 iki 1.

Plotas po preciziškumo ir atkūrimo padidėjimo kreive (angl. *Area under precision recall gain curve, AUC PRG*). PRG kreivė yra skirta gauti informacijai apie modelio veiksmingumą apmokant jį su nesubalansuotais duomenimis. Esant nesubalansuotam duomenų rinkiniui, AUC ROC metrika gali pateikti pernelyg optimistiškus rezultatus dėl to, kad ši metrika suteikia vienodą svertą neteisingų

teigiamos klasės spėjimų metrikai, neatsižvelgdama į klasių disbalansą. Eksperimente naudojamų duomenų klasių santykis yra 94/6, o atsižvelgiant į dalykinę sritį, kurioje svarbu tinkamai identifikuoti teigiamus atvejus, pasirinkta naudoti šią metriką modelių įvertinimui. PRG kreivė modifikuoja tikslumo ir atsiminimo rodiklius atsižvelgdama į klasių disbalansą ir susideda iš dviejų komponenčių: tikslumo padidėjimo (angl. *Precision gain*) ir atkūrimo padidėjimo (angl. *Recall gain*). Tikslumo padidėjimas čia parodo, kiek modelis geriau spėja teigiamą klasę atlikus modifikacijas klasių disbalansui, lyginant su atsitiktiniu spėjimu, o atsiminimo padidėjimas atspindi pagerėjimą identifikuojant teisingus priėmimus lyginant su atsitiktiniu spėjimu. Aukštesnis AUC PRG rodiklis rodo geresnę modelio kokybę.

Cohen‘o Kappa įvertis yra statistinis rodiklis, naudojamas įvertinti sutarimą tarp dviejų klasifikatorių. Kappa įvertis yra naudingas, nes jis apima ir tuos sutarimo atvejus, kurie įvyksta atsitiktinai. Cohen’o Kappa apskaičiuojama pagal pateiktą 11 formulę.

$$\kappa = \frac{P_o - P_e}{1 - P_e} \quad (11)$$

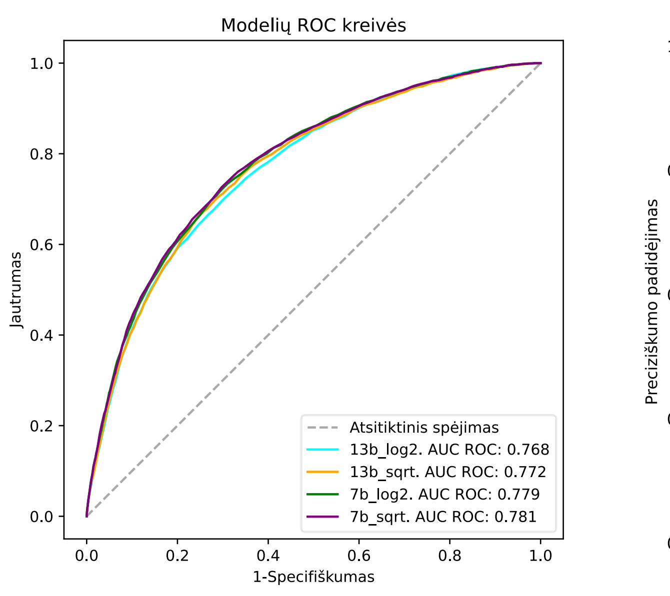
Čia P_o – stebimas sąlyginis sutarimas tarp dviejų klasifikatorių (mėginių, dėl kurių abu klasifikatoriai sutaria, aibės proporcija), P_e – hipotetinė atsitiktinio sutarimo tikimybė, kuri gali būti suskaičiuota naudojant kiekvienos kategorijos statistinius dažnius.

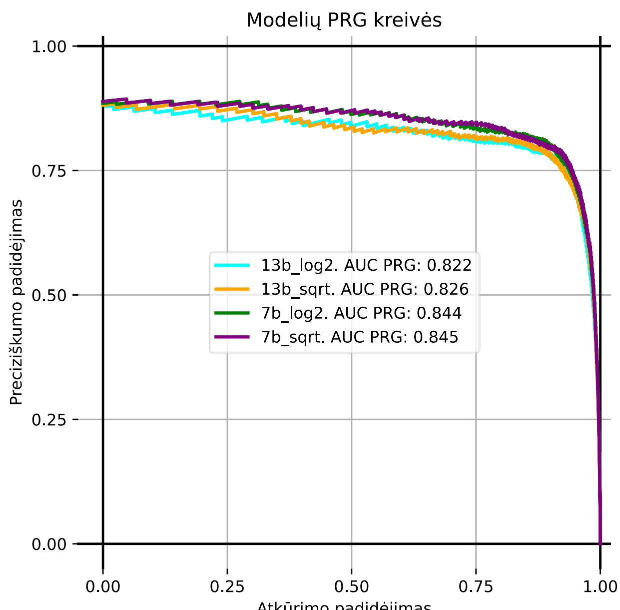
Kappa reikšmės svyruoja nuo -1 iki 1. 1 čia reiškia visšką sutarimą tarp klasifikatorių, 0 reiškia, kad nėra sutarimo tarp dviejų klasifikatorių, neskaitant atsitiktinio sutarimo. Mažesnės už 0 reikšmės reiškia sistematinį nesutarimą, prastesnis nei atsitiktinis sutarimas, o didesnės už nulį reiškia geresnį sutarimą nei atsitiktinis sutarimas.

Pirminė klasifikacijos tarp klasių riba parenkama pagal lygios paklaidų normos ribą (angl. *Equal error rate threshold, EER*). Tai yra klasifikavimo tikimybės taškas, kuriame neteisingų teigiamos ir neigiamos klasių spėjimo tikimybė sutampa. Ši metrika yra svarbi nesubalansuotuose rinkiniuose, kur yra svarbu valdyti neteisingų priėmimų ir neteisngų atmetimų kiekį. Eksperimento metu modelio tikslumas, sumaišymo matricos ir Cohen‘o Kappa įverčiai paskaičiuoti būtent šiame taške.

3.2. Eksperimento rezultatai

Atlikto eksperimento rezultatai pateikiami ir komentuojami šiame poskyryje. Treniruotų modelių ROC ir PRG kreivės pateikiamos žemiau (žr. 19 pav.).





19 pav. Detekcijos gerumo kreivės: ROC (kairėje) ir PRG (dešinėje)

Iš pateiktų ROC kreivių matoma, kad, nors ir modelių veiksmingumas paklaidų ribose sutampa, geriausias modelis yra *7b_log2*. Modelio AUC ROC (0.699) rodo sąlyginai vidutinį diferencijavimo tarp klasių gebėjimą, tačiau modelis klasifikuoja failus ženkliai geriau nei atsitiktinio spėjimo tikimybė. Iš PRG kreivių matoma, kad iš testuotų modelių, turint mintyje klasių disbalansą, geriausi rezultatai pasiekiami su *7b_log2* modeliu. Modelio 0.766 AUC PRG vertė reiškia modelio nuosekliai geresnį preciziškumo ir atsiminimo veiksmingumą lyginant su atsitiktiniu spėjimų naiviu klasifikatoriumi, kuris spėjimus parenka pagal klasių balansą.

Kitoms metrikoms suskaičiuoti parinktas klasifikavimo slenkstis kuris yra tolygus EER slenksčiui. Modelių EER slenksčiai matomi neteisingų priėmimų ir atmetimų santykio grafikuose (žr. 3 priedą). Šis taškas egzistuoja neteisingų priėmimų santykio ir neteisingų atmetimų santykio kreivių sankirtoje.

Nustačius modelių slenksčius, paskaičiuota sumaišymų matrica, Kappa rodiklis ir bendras modelio tikslumas. Apibendrinti modelių lyginamieji rezultatai pateikiami 15 lentelėje. Geriausio modelio rezultatai lentelėje paryškinti.

15 lentelė. Apibendrinti eksperimento metu tirtų modelių rezultatai, lyginant su *Flawfinder* statiniu analizatoriumi

Modelis	**AUC ROC**	**AUC PRG**	**Nustatytas klasifikavimo slenkstis**	**Tikslumas**	**Jautrumas (angl. *Sensitivity*)**	**Specifiškumas (angl. *Specificity*)**	**Taiklumas (angl. *Precision*)**	**Kappa įvertis**
13b_log2	0.768	0.822	0.488	69.38%	70.50%	69.30%	11.50%	0.116
13b_sqrt	0.772	0.826	0.492	71.10%	70.40%	71.10%	12.10%	0.127
7b_log2	0.779	0.844	**0.492**	71.39%	70.70%	71.40%	12.30%	0.130
7b_sqrt	**0.781**	**0.845**	0.492	71.37%	**71.20%**	**71.40%**	**12.30%**	**0.131**

Flawfinder statinis analizatorius	-	-	-	**85.582%**	22.81%	89.12%	10.57%	0.077

Lentelėje matomi rezultatai atskleidžia, kad geriausi rezultatai pasiekiami su 7b_sqrt. 0.131 Kappa įverčio vertė reiškia, kad tarp klasifikatoriaus ir duomenų yra minimalus sutarimas virš atsitiktinio spėjimo. Verta pastebėti, kad skirtumas tarp blogiausio ir geriausio rezultato yra minimalus. Rezultatai giliau išskaidyti pagal CWE identifikatorių pagal kurias buvo atliktas duomenų rinkinio stratifikuotas padalinimas. Šalia išfiltruotų reikšmių, pridėtos ir netaikinio klasės duomenys. Modelių rezultatai individualiais CWE identifikatoriais pažymėtiems failais pateikti 16 lentelė - 17 lentelė lentelėse. Geriausio modelio rezultatai lentelėse paryškinti.

16 lentelė. Modelių tikslumo rezultatai atskirais CWE identifikatoriais pažymėtiems failams

CWE identifikatorius	Modelis	Tikslumas	Kappa įvertis
CWE-119	13b_log2	69.37%	0.0144
	13b_sqrt	71.12%	0.0152
	7b_log2	**71.46%**	0.0156
	7b_sqrt	71.26%	**0.0158**
	Flawfinder statinis analizatorius	88.85%	0.0142
CWE-125	13b_log2	69.38%	0.0168
	13b_sqrt	71.20%	0.0185
	7b_log2	**71.48%**	**0.0187**
	7b_sqrt	71.43%	0.0186
	Flawfinder statinis analizatorius	88.88%	0.0085
CWE-20	13b_log2	69.34%	0.0110
	13b_sqrt	71.17%	**0.0124**
	7b_log2	**71.44%**	0.0116
	7b_sqrt	71.39%	0.0116
	Flawfinder statinis analizatorius	88.76%	0.0096
CWE-787	13b_log2	69.34%	0.0142
	13b_sqrt	71.20%	0.0153
	7b_log2	**71.44%**	**0.0164**
	7b_sqrt	71.39%	0.0159

	Flawfinder statinis analizatorius	88.79%	0.0125
CWE-Other	13b_log2	69.21%	0.0588
	13b_sqrt	70.10%	0.0657
	7b_log2	**71.28%**	0.0673
	7b_sqrt	71.26%	**0.0686**
	Flawfinder statinis analizatorius	88.79%	0.0125

Iš lentelės matyti, kad didžiojoje dalyje atvejų 7b_log2 yra geriausias modelis tikslumo atžvilgiu. Lyginant kiekvieno modelio failo pažeidžiamumo požymio atpažinimo galimybes, pastebėta, kad labai didelio skirtumo atpažįstant skirtingus tipus pažeidžiamumų turinčius failus nėra. Iš dalies manoma, kad tai yra dėl testinėje aibėje didelės netaikinio klasės failų didelio kiekio – apie 61 tūkst. failų prieš apie 3500 pažeidžiamų failų. Ta pati priežastis galioja ir dėl ko Flawfinder šiame teste pasirodė geriausiai – netaikinio aibės failų teisingai atpažinta 25% daugiau, tačiau to kaina – tris kartus mažiau teisingų teigiamos klasės atpažinimų. Iš rezultatų matyti reliatyviai reikšmingas Kappa įverčių skirtumas – CWE Other dauguma modelių pasiekė virš 50 procentų dydžio pagerėjimą lyginant su statiniu analizatoriumi. Tarp teisingų reikšmių ir spėjamų reikšmių visose kategorijose egzistuoja geresnis nei atsitiktinis sutapimas. Daugiau informacijos apie modelių veiksmingumą pateikia 17 lentelėje jautrumo ir specifiškumo rodmenys.

17 lentelė. Modelių ir įrankio jautrumo ir specifiškumo rezultatai atskirais CWE identifikatoriais pažymėtiems failams

CWE identifikatorius	Modelis	Jautrumas	Specifiškumas	Taiklumas
CWE-119	13b_log2	**81.00%**	69.30%	1.20%
	13b_sqrt	78.90%	71.10%	1.20%
	7b_log2	79.60%	71.40%	1.20%
	7b_sqrt	80.30%	**71.40%**	**1.20%**
	Flawfinder statinis analizatorius	28.90%	89.12%	1.18%
CWE-125	13b_log2	80.70%	69.30%	1.40%
	13b_sqrt	80.70%	71.10%	1.50%
	7b_log2	**80.70%**	**71.40%**	**1.50%**
	7b_sqrt	80.40%	71.40%	1.50%
	Flawfinder statinis analizatorius	20.10%	89.12%	0.90%
CWE-20	13b_log2	76.40%	69.30%	0.90%

	13b_sqrt	77.60%	71.10%	1.00%
	7b_log2	73.80%	71.40%	1.00%
	7b_sqrt	73.80%	71.40%	1.00%
	Flawfinder statinis analizatorius	25.32%	89.12%	0.96%
CWE-787	13b_log2	73.60%	69.30%	1.20%
	13b_sqrt	72.40%	71.10%	1.30%
	7b_log2	74.50%	71.40%	1.30%
	7b_sqrt	73.30%	71.40%	1.30%
	Flawfinder statinis analizatorius	24.55%	89.12%	1.24%
CWE-Other	13b_log2	65.50%	69.30%	6.00%
	13b_sqrt	65.80%	71.10%	6.00%
	7b_log2	66.10%	71.40%	6.00%
	7b_sqrt	**67.00%**	**71.40%**	**6.00%**
	Flawfinder statinis analizatorius	24.55%	89.12%	1.17%

Lentelėje pateikti duomenys atskleidžia, kad lyginant su *Flawfinder* statiniu analizatoriumi, visų modelių teisingų teigiamos klasės (t.y. teisingai identifikuotų pažeidžiamų failų) spėjimų proporcija tris kartus nuosekliai didesnė. Nors *Flawfinder* statinis analizatorius apskritai patiria 21 proc. mažiau klaidų identifikuojant sveikus failus, dalykinėje srityje, kurioje svarbiau identifikuoti pažeidžiamus failus, tris kartus didesnis veiksmingumas identifikuojant pažeidžiamus failus pateisina penktadaliu mažesnį tikslumą tinkamai, identifikuojant sveikus failus. Taip pat pastebėta, kad dirbtinio intelekto modeliai pasirodo geriau ir lyginant iš tikro teisingų taikinio spėjimų ir visų taikinio spėjimų proporciją – pasiektas vidutiniškai nuo 0.2 procento iki 2 procentų geresnis spėjimo veiksmingumas lyginant su *Flawfinder* įrankiu. Tai reiškia, kad dirbtinio intelekto modeliai ne tik aptinka daugiau pažeidžiamų failų, bet ir iš šių aptiktų failų padaro mažiau klaidų nei statinis analizatorius. Taip pat, pastebėta, kad modeliai išmoksta skirtingomis *CWE* pažaidomis pažymėtus failus diferencijuoti nuo sveikų skirtingu gebėjimu. Pavyzdžiui, iš C*WE*-125 pažaidomis pažymėtų failų, modeliai sugebėjo atpažinti virš 80%, kai tuo tarpu *Flawfinder* tą atliko keturiskart prasčiau. O prasčiausiai identifikuoti modeliams sekėsi *CWE-Other* grupe pažymėtus failus. Tikėtina, kad dėl skirtingų *CWE* pažaidų tipų kiekiškumo ir jų skirtumų turinyje, modeliams apibendrinti mokymąsi pavyko sunkiau.

Eksperimentiškai ištyrus sukurtų modelių kiekybinius rodiklius, nustatyta, kad geriausiai pasirodžiusi konfigūracija yra *CodeLlama* 7 milijardų hiperparametrų didysis kalbos modelis ir atsitiktinių miškų modelis, kurio saujos dydis medžiui parenkamas taikant kvadratinės šaknies iš paduodamų požymių kiekio taisyklę. Ši modelio konfigūracija pasirodė geriau už statinį analizatorių *Flawfinder*, aptikdama teisingai daugiau nei tris kartus daugiau pažeidžiamų failų. Tiesa, statinis analizatorius korektiškai aptinka 21% daugiau sveikų failų nei minėtas modelis. Tačiau turint mintyje tai, kad

klaidingai identifikuoti kaip nepažeidžiami failai gali būti daug žalingesni, šį skirtumą galima pateisinti.

Papildomai eksperimento metu parinktas atsitiktinių miškų modelių detekcijos slenkstis toks, kuriuo naudojantis teisingų taikinio klasės tikslumas (jautrumas) sutaptų su įrankio *Flawfinder* minėtu jautrumu (žr. 12 formulę).

$$Sensitivity(modelio) = Sensitivity(Flawfinder) \quad (12)$$

Toks santykis nustatytas esant detekcijos slenksčiui 0.7. Rezultatai pateikiami žemiau (žr. 18 lentelė, sumaišymo matricos - 19 ir 20 lentelėse).

18 lentelė. Atsitiktinių miškų modelių palyginimai esant lygioms teigiamų rezultatų atpažinimo galimybėms

Įrankis / modelis	**AUC ROC**	**AUC PRG**	**Tikslumas**	**Jautrumas**	**Specifiškumas**	**Taiklumas**	**Kappa įvertis**
7b_log2 modelis (nekoreguotas slenkstis)	0.779	0.844	69.383%	70.70%	71.40%	12.30%	0.130
7b_log2 modelis (detekcijos slenkstis – 0.7)			**92.47%**	19.86%	**96.56%**	24.59%	0.181
7b_sqrt modelis (nekoreguotas slenkstis)	**0.781**	**0.845**	71.104%	**71.20%**	71.40%	12.30%	0.131
7b_sqrt modelis (detekcijos slenkstis – 0.7)			92.206%	22.64%	96.13%	**24.80%**	**0.196**
Flawfinder statinis kodo analizatorius	Netaikoma	Netaikoma	85.582%	22.81%	89.12%	10.57%	0.077

19 lentelė. Įrankio *Flawfinder* eksperimento metu gauta sumaišymo matrica

Modelis / įrankis: Flawfinder		**Klasifikatoriaus nustatyta reikšmė**	
		Neigiama (netaikinio) klasė	Teigiama (taikinio) klasė
Faktinė reikšmė	Neigiama (netaikinio) klasė	56259	6866
	Teigiama (taikinio) klasė	2748	812

20 lentelė. 7b_sqrt modelio eksperimento metu gauta sumaišymo matrica

Modelis / įrankis: 7b_sqrt (koreguotas slenkstis)		**Klasifikatoriaus nustatyta reikšmė**	
		Neigiama (netaikinio) klasė	Teigiama (taikinio) klasė
Faktinė reikšmė	Neigiama (netaikinio) klasė	60682	2443
	Teigiama (taikinio) klasė	2754	806

Iš lentelėje matomų rezultatų akivaizdu, jog esant apylygėms teisingų taikinio klasės spėjimų atpažinimo sąlygoms (žr. 19 lentelė - 20 lentelė), pasiekiamas pastebimas padidėjimas modelio tikslumo, specifiškumo, taiklumo ir Kappa įverčių atžvilgiu. Ko gero, stipriausias rezultatų pokytis yra apie 64% sumažėjęs neteisingai identifikuotų taikinio klasės spėjimų skaičius (7b_sqrt modelis su 0.7 klasifikavimo slenksčiu). Tai rodo dirbtinio intelekto metodo pranašumą prieš statinius analizatorius – binarinio klasifikavimo uždavinyje suvienodinus teisingus taikinio spėjimus tarp dviejų įrankių, su sukurtu modeliu pasiekiamas reikšmingas neteisingai identifikuotų teigiamų reikšmių kiekio sumažėjimas.

3.3. Sprendimo taikymo rekomendacijos

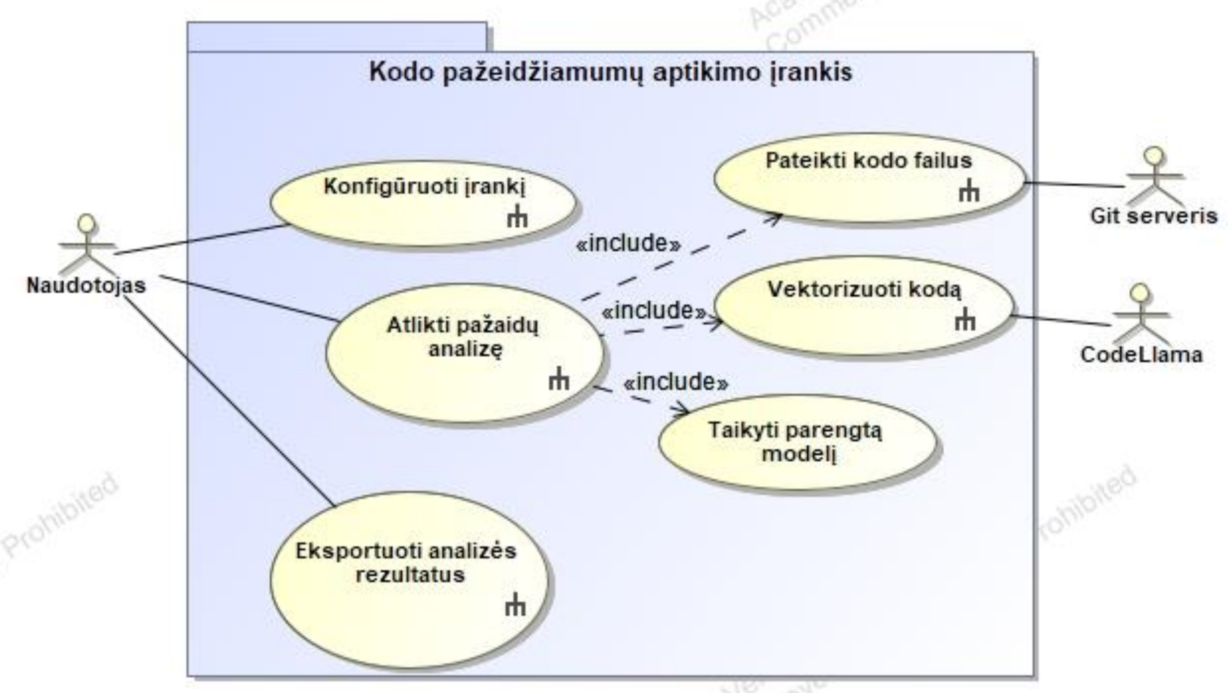
Realizuotą atsitiktinių miškų įrankį tinka naudoti, jei siekiama antrinės nuomonės ar failas yra pažeidžiamas arba siekiant efektyvaus pirminio rezultatų filtravimo. Modelį tinka naudoti kuriant tiek paprastas tiek sudėtingas C programas ir siekiant išanalizuoti egzistuojančių programų C failus. Dėl CodeLlama aukštų reikalavimų atminčiai ir atliekamų operacijų pobūdžio, sprendimas netinka naudoti, kai analizės laikas yra svarbus faktorius, kadangi naudojant modelių kombinaciją, vieno failo analizė naudojant centrinį procesorių trunka apie 30-40s. Todėl norint naudoti modelį primygtinai rekomenduojama vektorizuoti failą pasinaudojant grafinį procesorių arba tenzorinį procesorių dėl jų efektyvumo dirbant su matricomis. Taip pat, rekomenduojama turėti mintyje ir atminties sunaudojimą: 7 milijardų CodeLlama modelis reikalauja 13 GB operatyviosios atminties. Norint naudoti sisteminiu lygmeniu šį sprendimą, siūloma laikyti CodeLlama modelį laikyti užkrautą į operatyvią atmintį, taip sumažinant laiko tarpą tarp failo vektorizavimui perdavimo ir gauto pažeidžiamumo požymio spėjimo. Taip pat sprendimas netinka būti naudojamas su kitomis kalbomis nei C.

4. Pažeidžiamumų programiniame kode aptikimo įrankio eksperimentinės realizacijos projektas

Šiame skyriuje pateikiamas įrankio eksperimentinės realizacijos panaudojimo atvejų modelis ir įrankio loginė architektūra.

4.1. Įrankio eksperimentinės realizacijos panaudojimo atvejų modelis

Eksperimentinės realizacijos funkciniams reikalavimams specifikuoti pasinaudota *UML* panaudojimo atvejų diagrama.

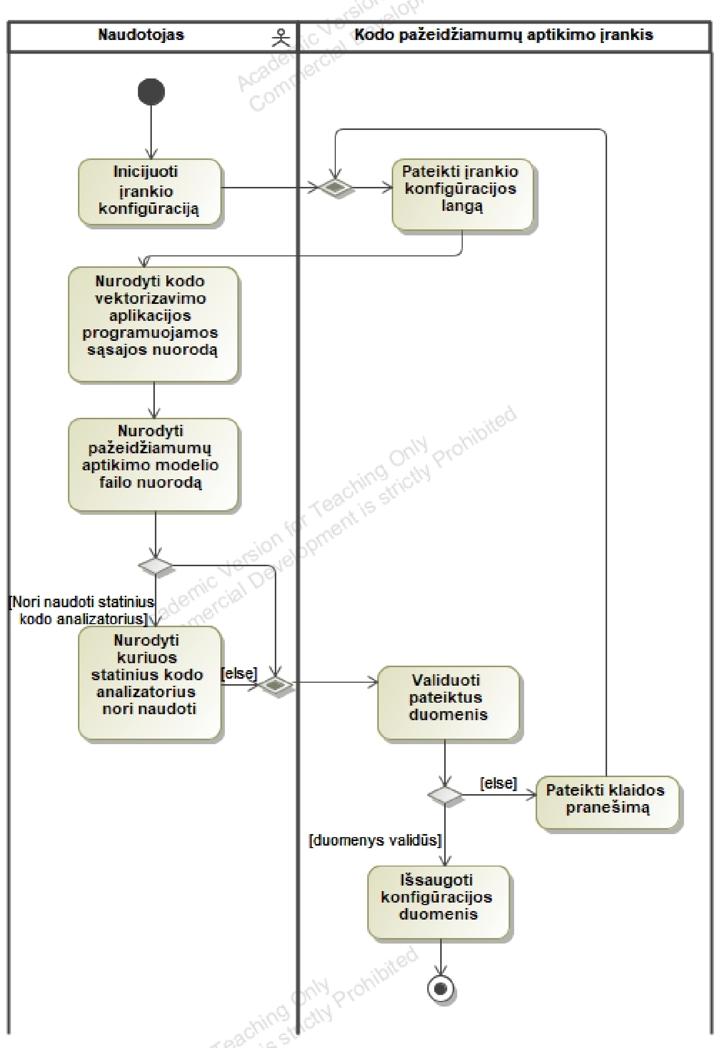


20 pav. Programinio kodo pažeidžiamumų aptikimo įrankio panaudojimo atvejų diagrama

Sistemos veikime dalyvauja šie išoriniai aktoriai:

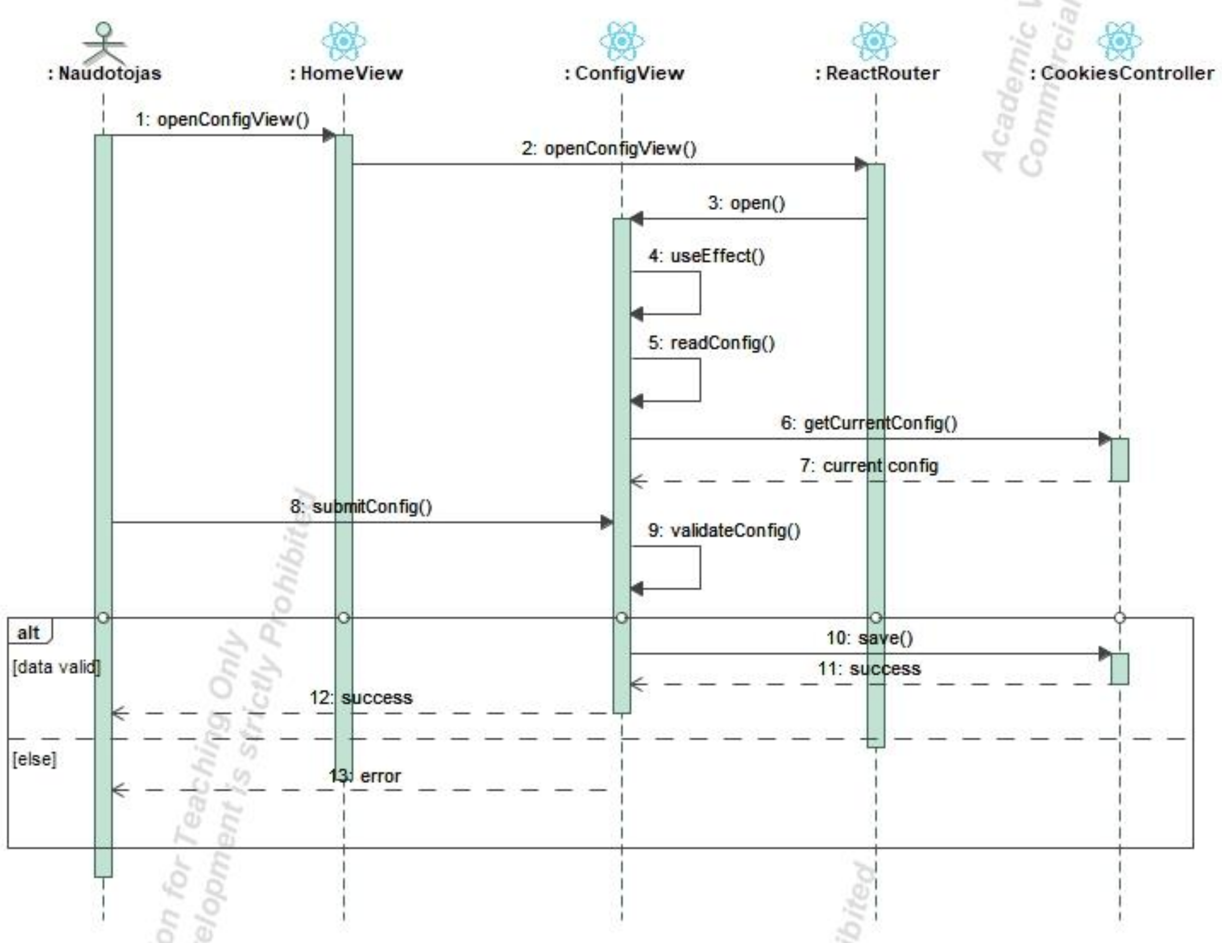
- Naudotojas – sistemos naudotojas, pateikiantis sistemai failų katalogą analizei;
- Git serveris – naudotojas gali pateikti viešai prieinamą git repozitoriją ir sistema kreipiasi į nurodytą git serverį. Git serveris sistemai grąžina failų katalogą, kurie bus išanalizuoti;
- CodeLlama – didysis kalbos modelis. CodeLlama naudojamas Langchain didžiųjų kalbos modelių karkaso apvalkale. Atsakingas už kodo vektorinių reprezentacijų grąžinimą sistemai.

Panaudojimo atvejai detalizuojami UML veiklos ir sekų diagramomis (žr. 21 pav. - 31 pav.). Panaudojimo atvejams specifikuoti pasitelktos panaudojimo atvejų lentelės (žr. 21 - 26 lentelėse).



21 pav. Panaudojimo atvejo „Konfigūruoti įrankį“ veiklos diagrama

Kadangi įrankio veikimui reikalinga nurodyti kodo vektorizavimo aplikacijos programuojamosios sąsajos nuorodą ir failų klasifikavimo nuorodą, reikia sukonfigūruoti įrankį. Naudotojas turėtų nurodyti privalomai šias dvi nuorodas bei pasirinktinai nurodyti, ar naudojami statiniai kodo analizatoriai. Ši informacija validuojama taisyklėmis (t.y. ar korektiškai nurodytas adresas). Jeigu validacija sėkminga, informacija išsaugojama naudotojo naršyklės sausainėliuose (angl. *cookies*).

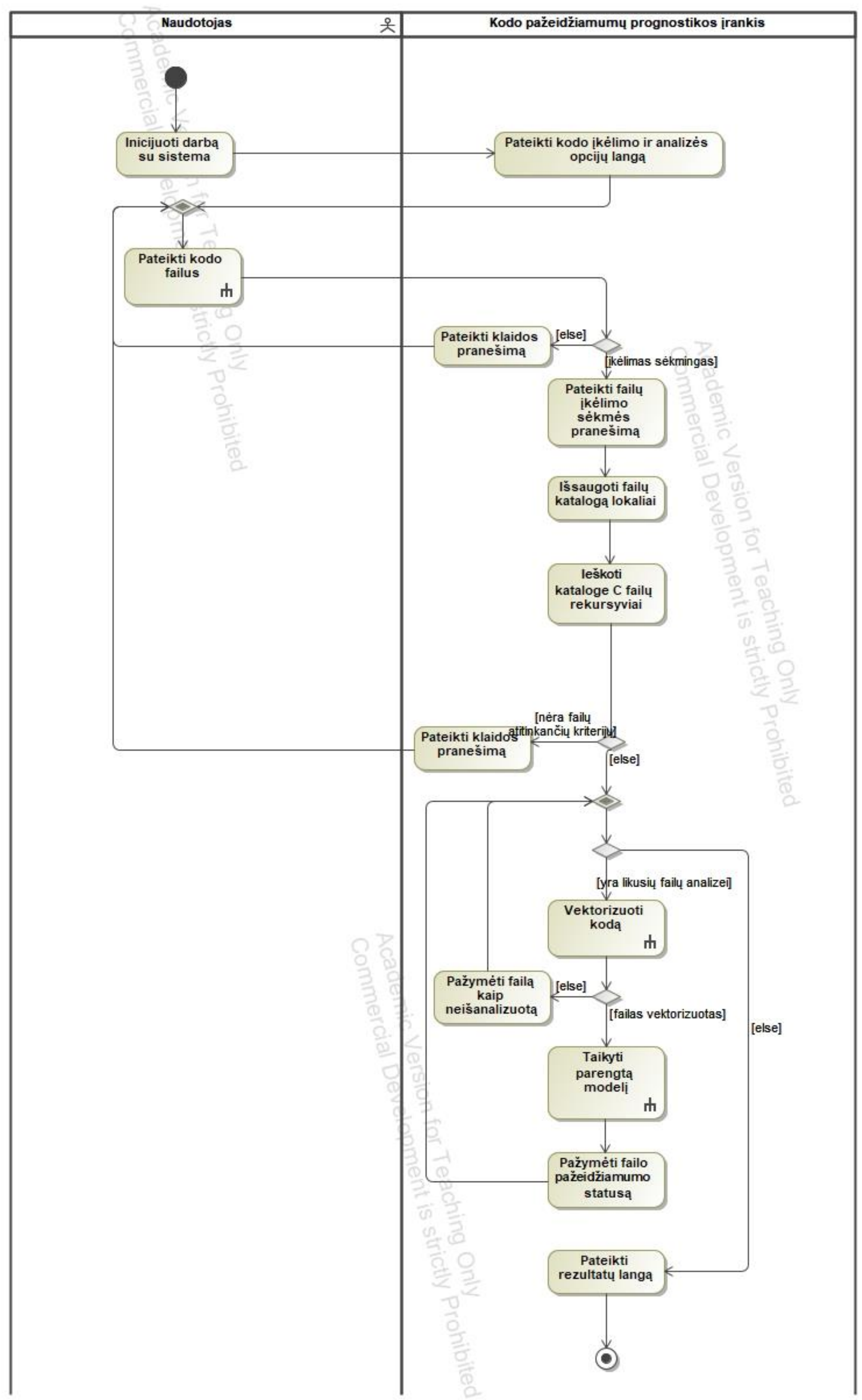


22 pav. Panaudojimo atvejo „Konfigūruoti įrankį“ sekų diagrama

Sekų diagramoje matoma naudotojo sąveika su sistema. Konfigūracijos lango komponentui užkrovus langą, sistema bando iš sausainėlių paimti dabartinę konfigūraciją. Jei ji randama, konfigūracijoje rastos vertės užpildomos atitinkamuose laukeliuose. Naudotojas šių laukelių vertes gali keisti 8 žingsnyje. Šių verčių validavimas vykdomas kliento aplikacijos dalyje. Konfigūracijos panaudojimo atvejo specifikacija aprašyta 21 lentelėje.

21 lentelė. Panaudojimo atvejo „Konfigūruoti įrankį“ specifikacijos lentelė

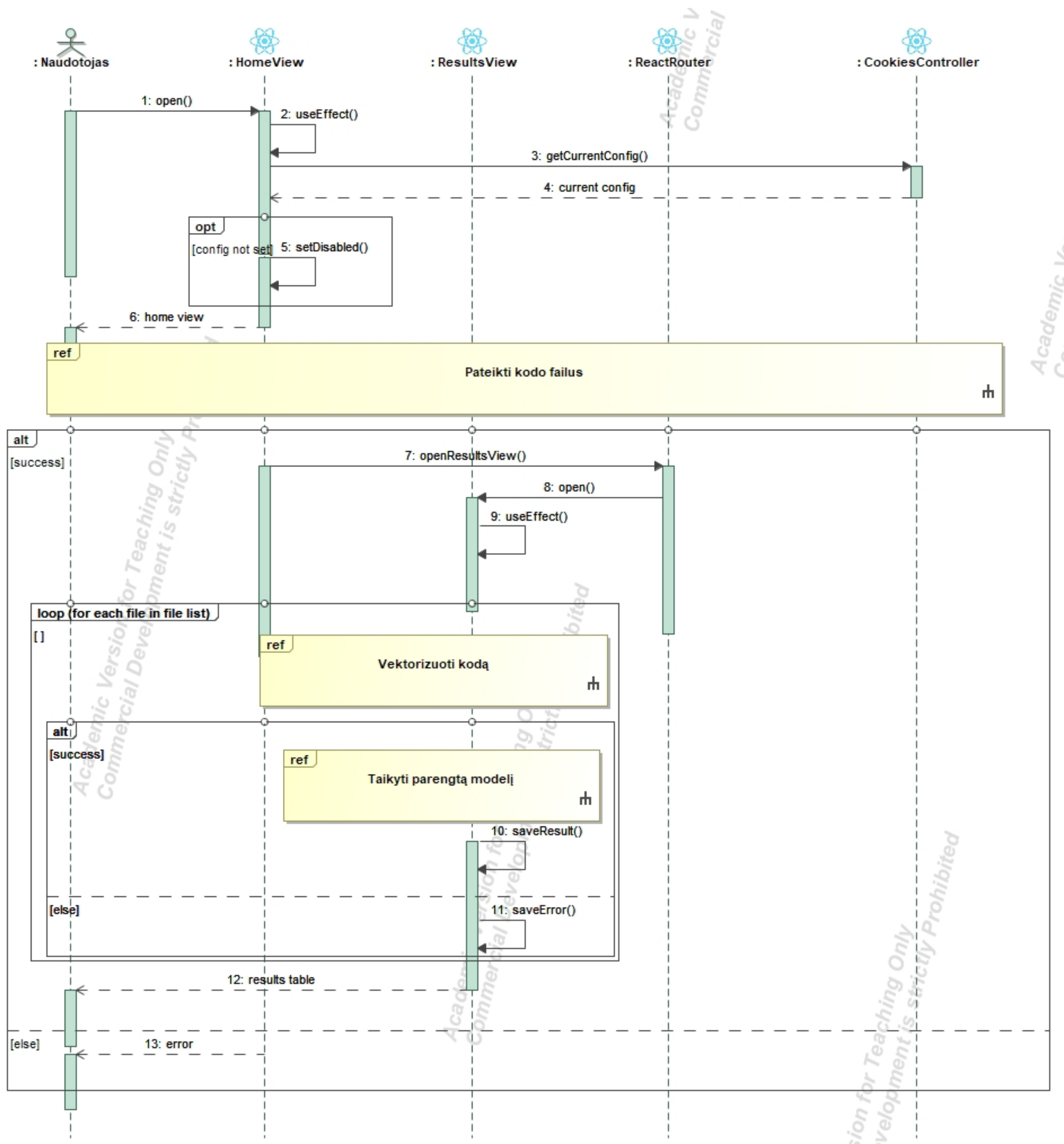
<table>
<tr><td colspan="3">PA 1: Konfigūruoti įrankį</td></tr>
<tr><td colspan="3">Tikslas. Suteikti naudotojui galimybę konfigūruoti pažeidžiamumų aptikimo įrankio nustatymus.</td></tr>
<tr><td colspan="3">Aprašymas. Naudotojas gali pateikti failų vektorizavimo aplikacijos programuojamosios sąsajos nuorodą, pasirinkti aptikimo modelio nuorodą ir nurodyti, kokius kodo analizatorius norima naudoti.</td></tr>
<tr><td colspan="2">Prieš sąlyga</td><td>-</td></tr>
<tr><td colspan="2">Aktorius</td><td>Naudotojas</td></tr>
<tr><td rowspan="3">Susiję panaudojimo atvejai</td><td>Išplečiantys PA</td><td>-</td></tr>
<tr><td>Apimami PA</td><td>-</td></tr>
<tr><td>Specializuojami PA</td><td>-</td></tr>
<tr><td colspan="2">Po sąlyga</td><td>Konfigūracija išsaugota sėkmingai.</td></tr>
</table>



23 pav. Panaudojimo atvejo „Atlikti pažaidų analizę“ veiklos diagrama

Panaudojimo atvejo veiklos diagramoje detalizuojamas 2.6 poskyryje numatyto algoritmo realizacijos veikimo principas. Naudotojui pradėjus darbą su sistema, jam pateikiamas kodo įkėlimo ir analizės opcijų langas. Šiame lange galima pasirinkti ar įkelti katalogą *zip* formatu, ar nurodyti viešą Git repozitoriją. Taip pat, jeigu įrankio konfigūracijos lange yra nurodytas statinio kodo

analizatoriaus poreikis, failai papildomai išanalizuojami pasinaudojant *Flawfinder* statinės kodo analizės įrankiu. Bendri rezultatai pateikiami analizės rezultatų lange. Panaudojimo atvejo realizacijos lygio sekų diagrama pateikiama 24 pav.

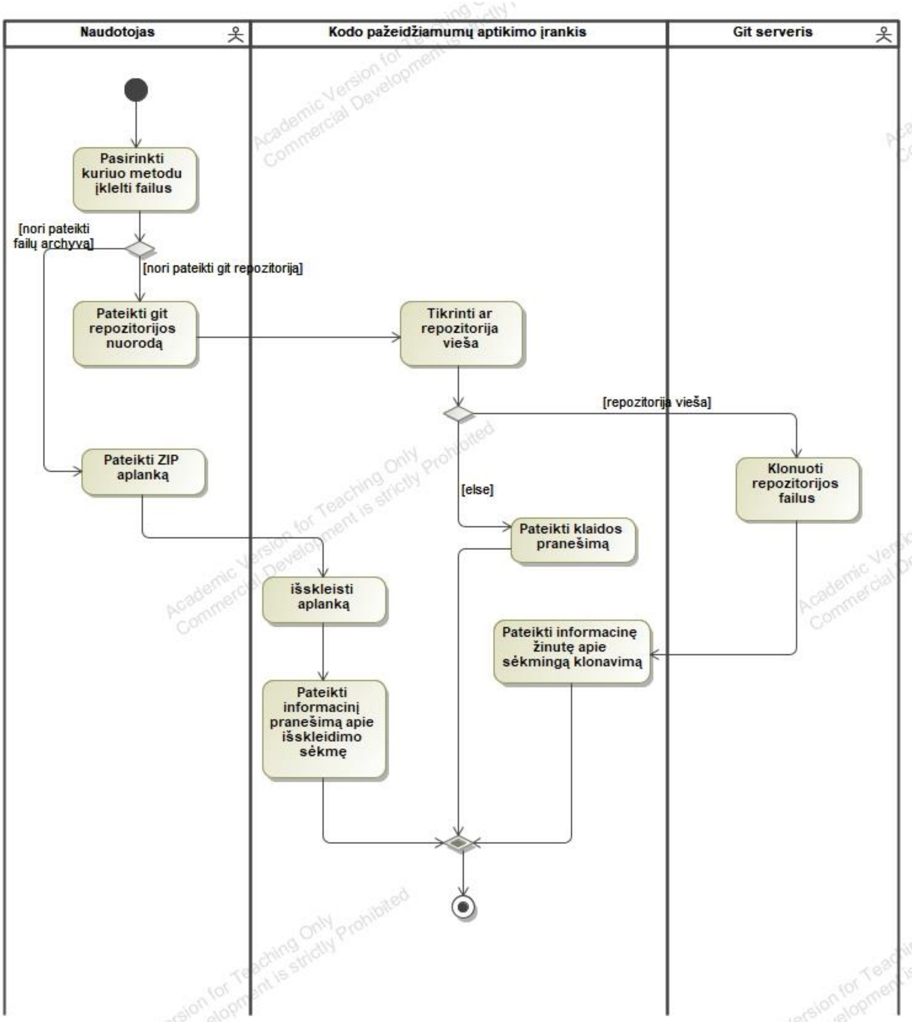


24 pav. Panaudojimo atvejo „Atlikti pažaidų analizę“ sekų diagrama

Sekų diagramoje matoma naudotojo sąveika su sistema atliekant pažaidų analizę. Verta atkreipti dėmesį, kad „Pateikti kodo failus“ panaudojimo atvejis grąžina analizuotinų failų sąrašą, kuris vėliau naudojamas aprašytame cikle. Kodas vektorizuojamas po vieną failą, o vektorizavimo sėkmės atveju, gautas vektorius naudojamas taikant parengtą modelį. Panaudojimo atvejo specifikacija pateikiama 22 lentelėje.

22 lentelė. Panaudojimo atvejo „Atlikti pažaidų analizę“ specifikacijos lentelė

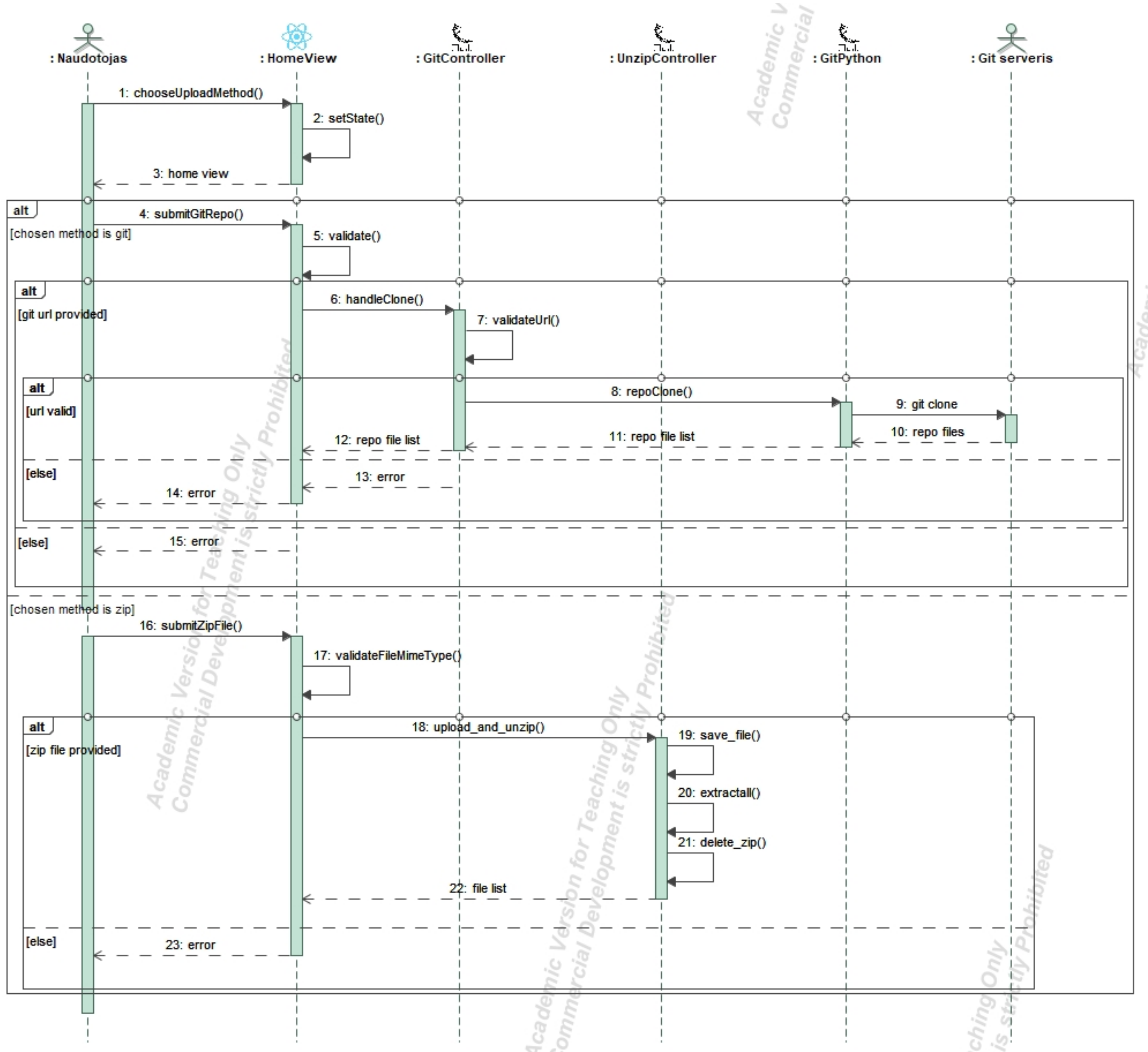
PA 2: Atlikti pažaidų analizę		
Tikslas. Suteikti naudotojui galimybę išanalizuoti pateiktus failus SVP ir SAST metodais.		
Aprašymas. Naudotojas inicijuoja analizę. Sistemai pateikus failų įkėlimo dialogą, jis įkelia savo C failus, kuriuos nori išanalizuoti. Failai vektorizuojami ir tada analizuojami pasinaudojant dirbtinio intelekto modeliu.		
Prieš sąlyga		Įrankis yra sukonfigūruotas.
Aktorius		Naudotojas
Susiję panaudojimo atvejai	Išplečiantys PA	-
	Apimami PA	PA3, PA4, PA5
	Specializuojami PA	-
Po sąlyga		Pažeidžiamumų analizė atlikta ir rezultatai pateikti naudotojui



25 pav. Panaudojimo atvejo „Pateikti kodo failus“ veiklos diagrama

Norint atlikti kodo analizę, reikia pateikti kodo failus sistemai. Naudotojas gali pasirinkti, ar failus pateikti *zip* archyvo pavidalu ar nurodant *Git* repozitoriją. Jeigu pasirenkamas pirmasis variantas, failai yra išskleidžiami lokalioje serverio repozitorijoje ir parengiami naudojimui. Jeigu yra

pasirenkamas antrasis variantas, sistema papildomai tikrina, ar nurodyta repozitorija yra klonuotina (t.y. vieša). Jei ne, pateikiamas klaidos pranešimas. Jei repozitorija yra vieša, ji yra klonuojama į lokalią aplinką. Sėkmingo failų pateikimo atveju, pateikiamas sėkmės pranešimas. Panaudojimo atvejo realizacijos lygio sekų diagrama pateikiama 26 pav.



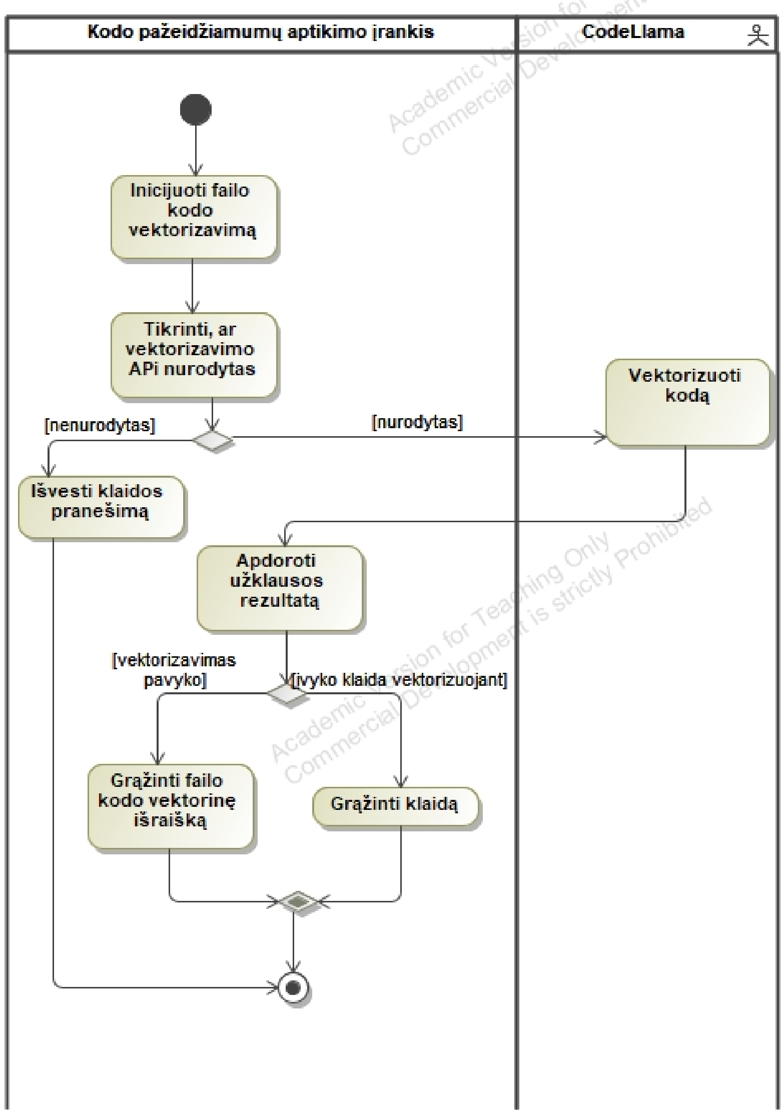
26 pav. Panaudojimo atvejo „Pateikti kodo failus“ sekų diagrama

Sekų diagramoje numatyti du failų įkėlimo atvejai, per *Git* serverį arba pasinaudojant tiesiogine failų įkėlimo sąsaja. Verta pastebėti, kad failų sąrašas šiame etape negrąžinamas naudotojui (12 ir 22 žingsniai). Taip yra todėl, nes klientinė aplikacijos dalis (*HomeView*) šią informaciją naudoja tolimesnei failų analizei (žr. 24 pav.). Panaudojimo atvejo specifikacija pateikiama 23 lentelėje.

23 lentelė. Panaudojimo atvejo „Pateikti kodo failus“ specifikacijos lentelė

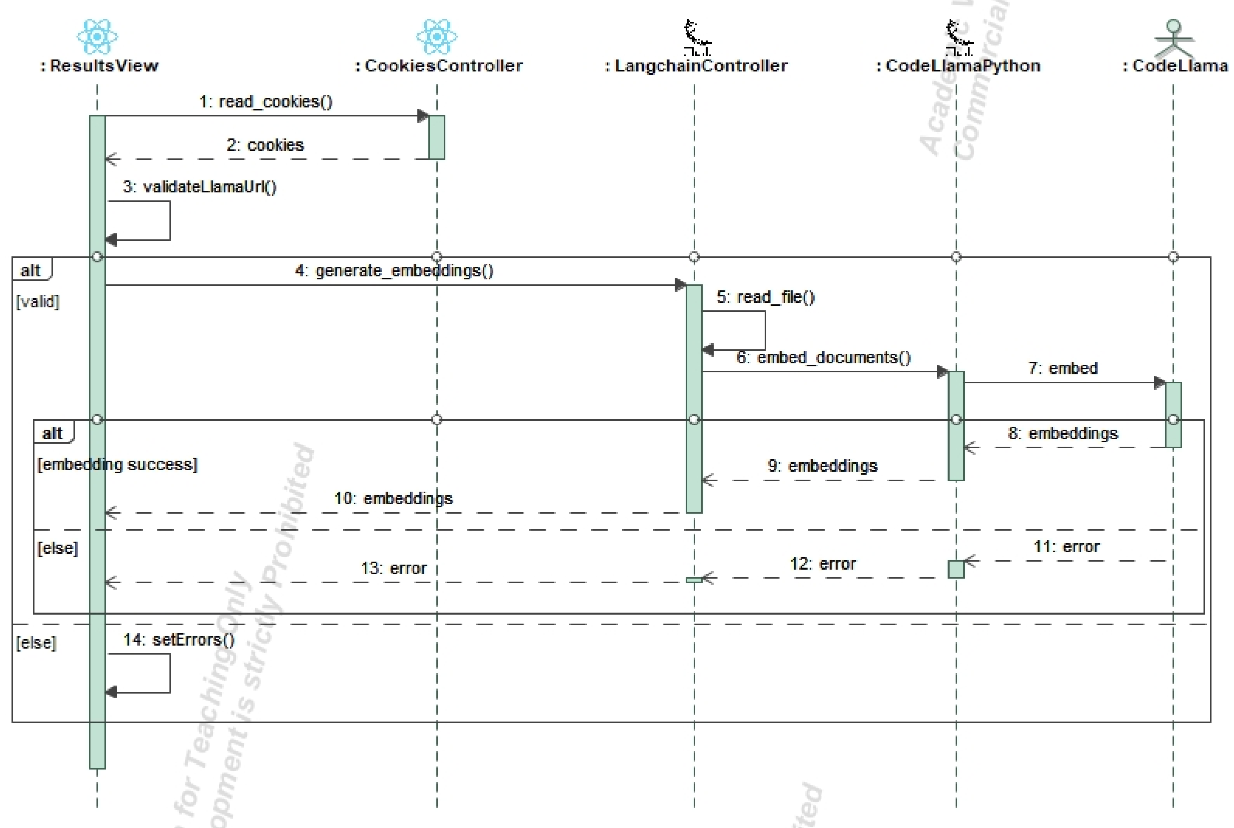
PA 3: Pateikti kodo failus
Tikslas. Suteikti naudotojui galimybę įkelti failus analizei
Aprašymas. Naudotojas gali pateikti failus dviem būdais: nurodant Git repozitoriją arba įkeliant zip failą, kuris yra išskleidžiamas.

Prieš sąlyga		-
Aktorius		Naudotojai
Susiję panaudojimo atvejai	Išplečiantys PA	-
	Apimami PA	-
	Specializuojami PA	-
Po sąlyga		Failai yra įkelti.



27 pav. Panaudojimo atvejo „Vektorizuoti kodą“ veiklos diagrama

Sistema vektorizuoja kodo failus, pasinaudodama *CodeLlama* didžiuoju kalbos modeliu. Tam būtina nurodyti kodo failų vektorizavimo aplikacijos programuojamosios sąsajos nuorodą. Jeigu tai nėra padaroma, kodo failų vektorizavimas negali būti vykdomas ir pateikiamas klaidos pranešimas. Sėkmės atveju, *CodeLlama* pateikia sistemai kodo failo vektorinę išraišką, kuri gali būti naudojama kaip įvestis dirbtinio intelekto modeliui. Panaudojimo atvejo realizacijos lygio sekų diagrama pateikiama 28 pav.

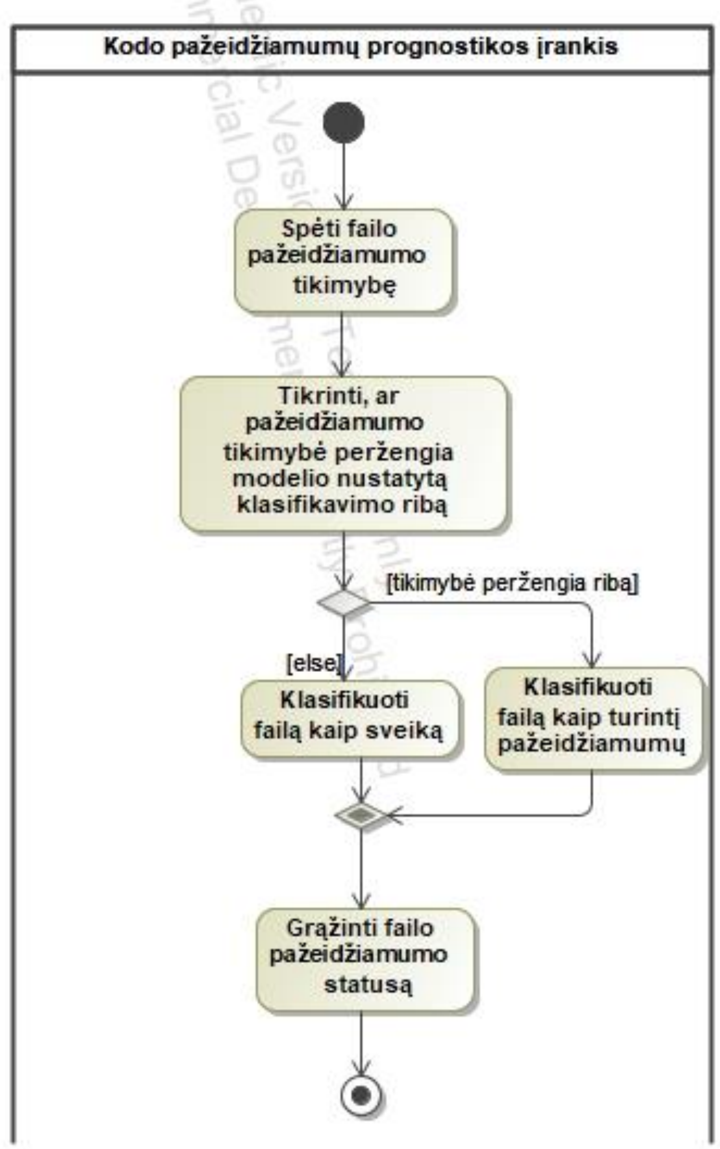


28 pav. Panaudojimo atvejo „Vektorizuoti kodą“ sekų diagrama

Sekų diagramoje matomas sistemos veikimo principas siekiant failų turinį paversti skaičių vektoriumi. CodeLlama modelio integravimui panaudotos dvi bibliotekos, į kurias kreipiamasi – *Langchain* ir *CodeLlamaPython*. Šios bibliotekos naudoja nurodytą lokaliose direktorijose esantį *CodeLlama* modelį ir grąžina vektorizuotą failo turinį. Panaudojimo atvejo specifikacija pateikiama 24 lentelėje.

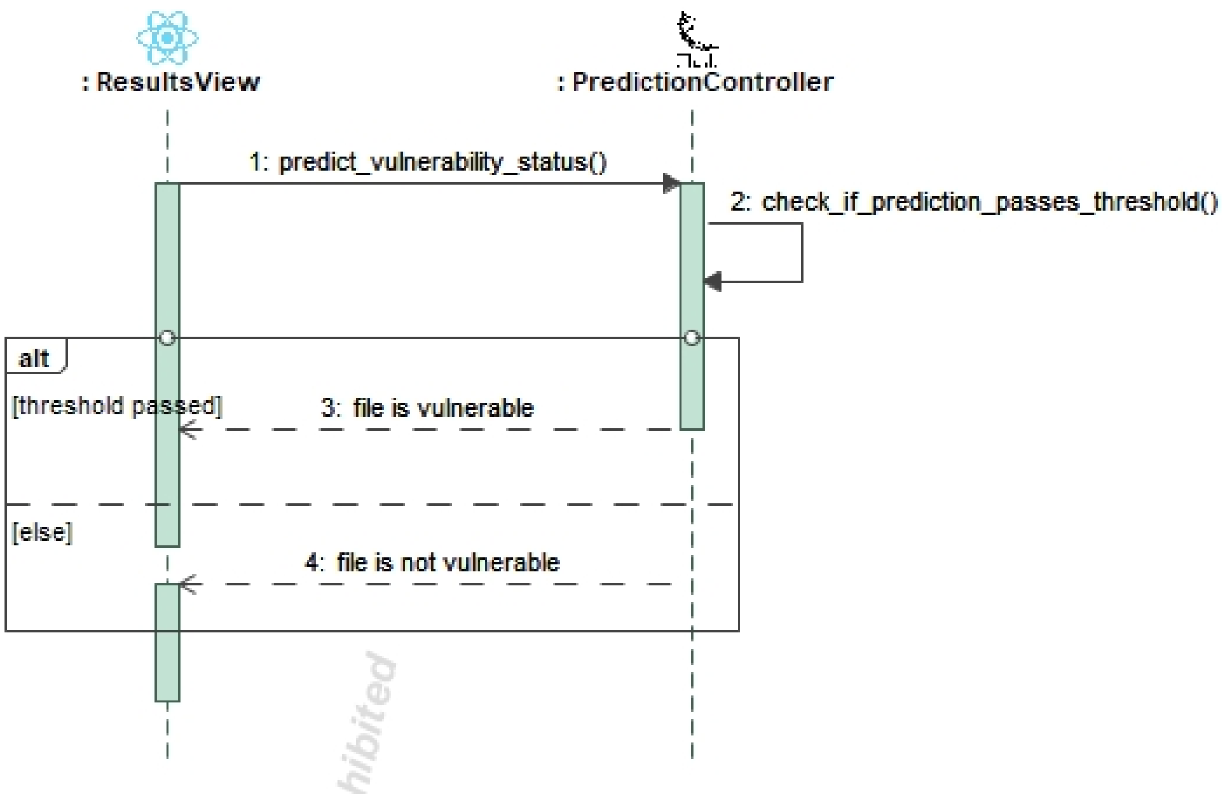
24 lentelė. Panaudojimo atvejo „Vektorizuoti kodą“ specifikacijos lentelė

<table>
<tr><td colspan="3">PA 4: Vektorizuoti kodą</td></tr>
<tr><td colspan="3">Tikslas. Suteikti sistemai prieigą vektorizuoti kodą pasinaudojant CodeLlama didžiuoju kalbos modeliu.</td></tr>
<tr><td colspan="3">Aprašymas. Sistema nusiunčia failus apdoroti CodeLlama vektorizavimo funkcijai. Grąžinama arba failo kodo vektorinė išraiška arba klaida.</td></tr>
<tr><td colspan="2">Prieš sąlyga</td><td>Įrankis sukonfigūruotas tinkamai</td></tr>
<tr><td colspan="2">Aktorius</td><td>CodeLlama</td></tr>
<tr><td rowspan="3">Susiję panaudojimo atvejai</td><td>Išplečiantys PA</td><td>-</td></tr>
<tr><td>Apimami PA</td><td>-</td></tr>
<tr><td>Specializuojami PA</td><td>-</td></tr>
<tr><td colspan="2">Po sąlyga</td><td>Pateikiama failo kodo vektorinė išraiška.</td></tr>
</table>



29 pav. Panaudojimo atvejo „Taikyti parengtą modelį“ veiklos diagrama

Parengto modelio taikymas yra pagrindinis šios sistemos darbo rezultatas. Dirbtinio intelekto modelis priima vektorinę failo kodo išraišką ir pagal jos požymius spėja failo pažeidžiamumo tikimybę. Ši tikimybė tada yra lyginama su eksperimentinio tyrimo metu (žr. 3.2 poskyrį) nustatyta klasifikavimo riba. Jeigu failo pažeidžiamumo tikimybė viršija šią ribą, failas klasifikuojamas kaip turintis pažeidžiamumų. Panaudojimo atvejo realizacijos lygio sekų diagrama pateikiama 30 pav.

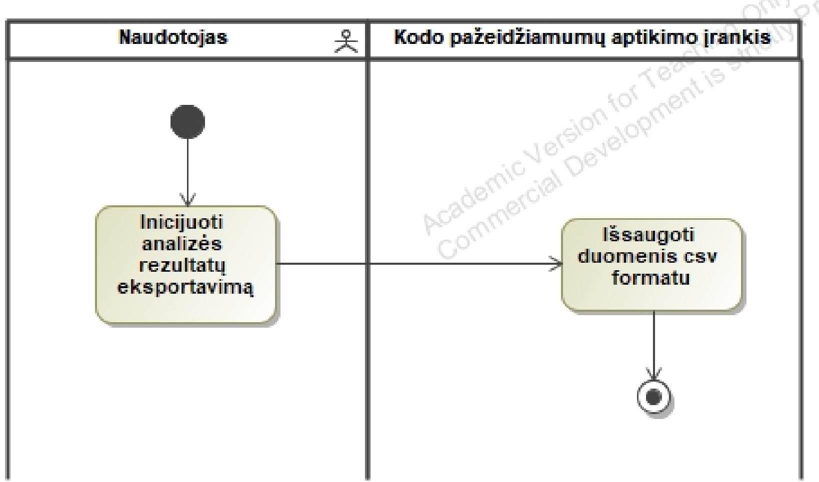


30 pav. Panaudojimo atvejo „Taikyti parengtą modelį“ sekų diagrama

Sekų diagramoje vaizduojamas pažeidžiamumo spėjimas individualiam failui. Spėjamas pažeidžiamumo požymis grąžinamas kaip tikimybė, kuris lyginamas su klasifikacijos riba ir ši tikimybė lyginimo metu paverčiama Būlio verte, kuri ir grąžinama sistemai. Šio panaudojimo atvejo specifikacija yra aprašyta 25 lentelėje.

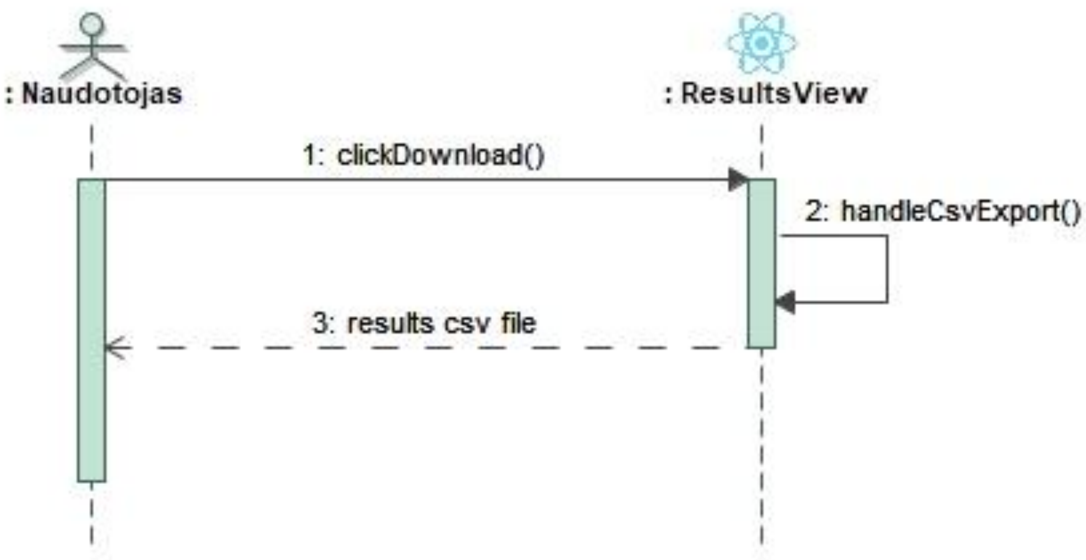
25 lentelė. Panaudojimo atvejo „Taikyti parengtą modelį“ specifikacijos lentelė

PA 5: Taikyti parengtą modelį		
Tikslas. Suteikti sistemai pažeidžiamumo spėjimo funkciją		
Aprašymas. Sistema naudoja dirbtinio intelekto modelį, kad spėtų failo pažeidžiamumo požymį ir lygina su eksperimentiniu būdu nustatyta klasifikavimo riba.		
Prieš sąlyga		-
Aktorius		-
Susiję panaudojimo atvejai	Išplečiantys PA	-
	Apimami PA	-
	Specializuojami PA	-
Po sąlyga		Grąžintas failo pažeidžiamumo statusas.



31 pav. Panaudojimo atvejo „Eksportuoti analizės rezultatus“ veiklos diagrama

Tolimesniam rezultatų naudojimui, naudotojas gali eksportuoti rezultatus kaip *CSV* (angl. trump. *Comma separated values,* liet. kableliais atskirtos vertės). Šis veiksmas yra atliekamas analizės rezultatų lange. Panaudojimo atvejo realizacijos lygio sekų diagrama pateikiama 32 pav.



32 pav. Panaudojimo atvejo „Eksportuoti analizės rezultatus“ sekų diagrama

Sekų diagramoje matomas naudotojo sąveikos su sistema principas eksportuojant analizės rezultatus. *CSV* failo generavimas ir vėliau jo eksportavimas atliekamas klientinėje sistemos dalyje. Panaudojimo atvejo specifikacija pateikiama 26 lentelėje.

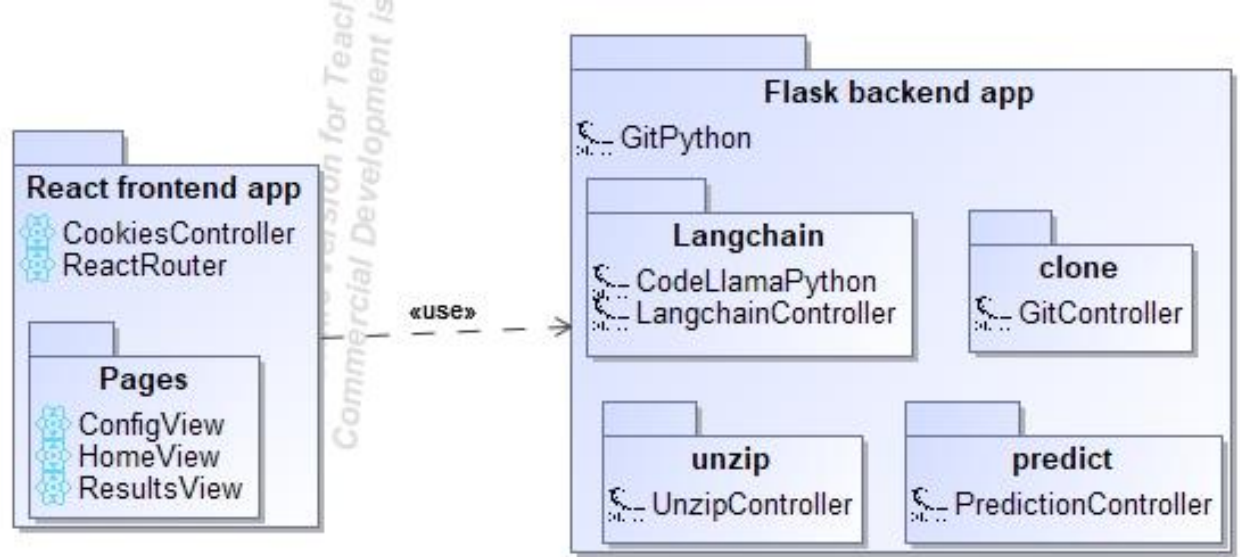
26 lentelė. Panaudojimo atvejo „Eksportuoti analizės rezultatus“ specifikacijos lentelė

PA 6: Eksportuoti analizės rezultatus
Tikslas. Suteikti naudotojui galimybę atsisiųsti analizės rezultatus CSV formatu
Aprašymas. Naudotojas paspaudžia rezultatų eksportavimo mygtuką, sistema išsaugo rezultatus į csv failą ir išsiunčia naudotojui.

Prieš sąlyga		-
Aktorius		Naudotojas
Susiję panaudojimo atvejai	Išplečiantys PA	-
	Apimami PA	-
	Specializuojami PA	-
Po sąlyga		CSV failas išsiųstas naudotojui

4.2. Kodo spragų aptikimo eksperimentinės realizacijos loginė architektūra

Sekančiose diagramose specifikuojamas sistemos statinis vaizdas ir loginė architektūra. UML paketų diagramoje (žr. 33 pav.) aprašyta sistemos artefaktų pasiskirstymas aplinkose. Verta pastebėti, kad virtualios *Python* aplinkos suteikiami moduliai nevaizduojami. Klientinė dalis susideda iš kelių *React* komponentų, iš kurių trys yra puslapių komponentai. *Flask* serverio dalis susideda iš keturių aplankų, kuriuose sudėti pagrindinę sistemos logiką atliekantys *Flask* failai.

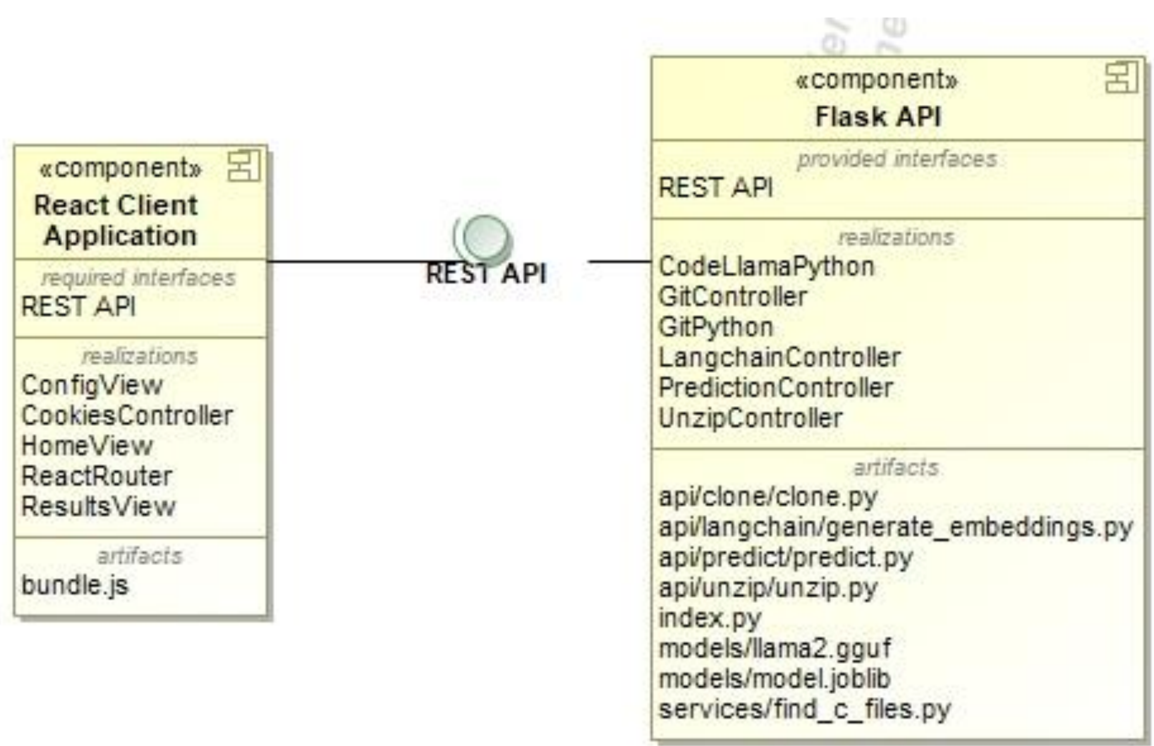


33 pav. Realizuoto programinės įrangos pažeidžiamumų prognostikos įrankio paketų diagrama

Kodo spragų aptikimo įrankio realizacija susideda iš dviejų komponentinių dalių, kurios komunikuoja *REST API* principu:

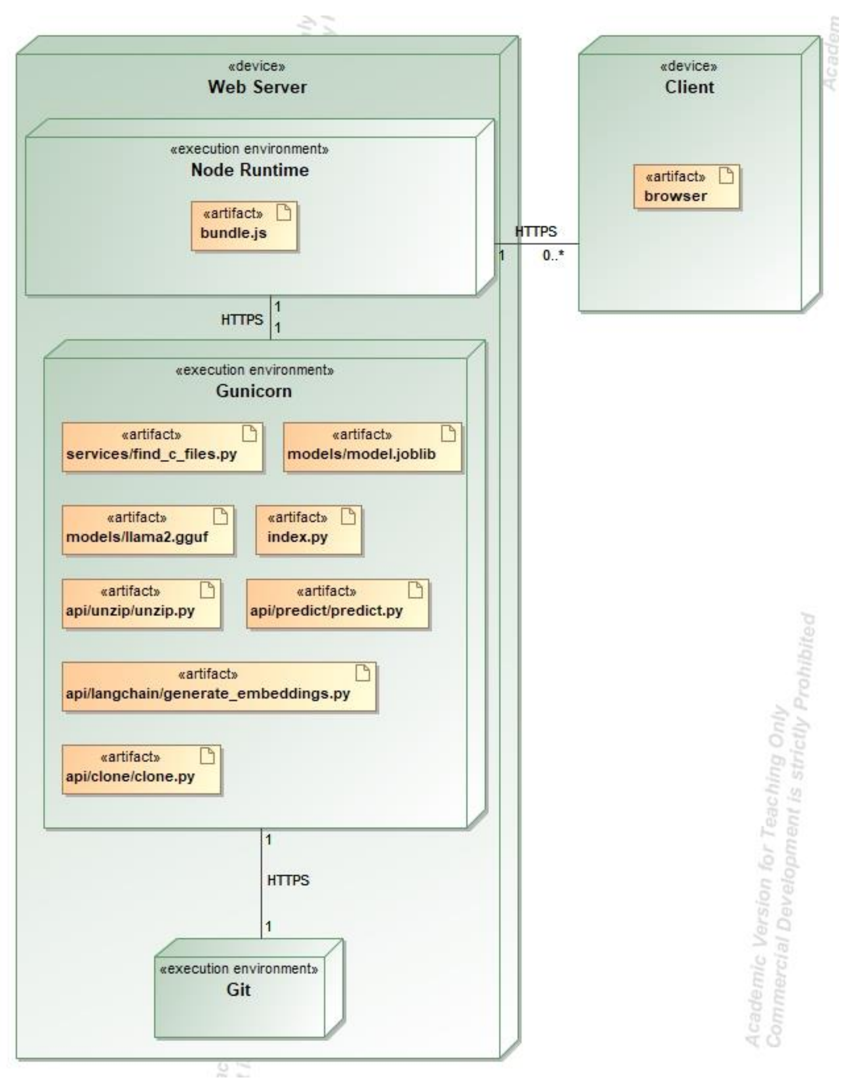
- Flask serverinės dalies komponentas atsakingas už kodo vektorizavimą ir atsitiktinių miškų modelio spėjimų atlikimą;
- Klientinės dalies komponentas atsakingas už grafinės sąsajos pateikimą naudotojui ir galimybę suteikti naudotojui patogiai perduoti failus per grafinę sąsają.

Sistemos komponentai ir komponentus realizuojantys artefaktai atvaizduojami 34 pav.



34 pav. Kodo spragų aptikimo įrankio sistemos komponentų diagrama

Sistemos diegimo diagrama pateikiama 35 pav. Tai yra pavyzdinis ir siūlomas sistemos diegimo atvejis, kadangi nors ir sistema yra moduliari ir ją galima išskirstyti per kelis serverius, siūloma ją talpinti viename serveryje dėl komunikacijos greitaveikos tarp klientinės dalies ir serverinės dalies. Šiuo atveju, *Node Runtime* paleidimo aplinkoje talpinami artefaktai, susiję su *React* klientine dalimi, o *Flask* daliai pasirinkta *Gunicorn* paleidimo aplinka dėl lygiagrečių užklausų palaikymo. *Git* paleidimo aplinka pavaizduota, nes *Flask* kreipiasi į ją serverio viduje esant naudotojo pasirinkimui įkelti failus per *Git* paslaugą.



35 pav. Programinio kodo spragų aptikimo įrankio diegimo diagrama

5. Programinio kodo spragų aptikimo įrankio realizacija ir testavimas

Šiame skyriuje aprašoma programinio kodo spragų aptikimo įrankio realizacija. Apžvelgiamas testavimas.

5.1. Sprendimo realizacijos ir veikimo aprašas

Programinio kodo spragų įrankio sprendimas realizuotas pasitelkiant *Javascript* biblioteka *React* klientinei daliai, o serverio dalis realizuota pasinaudojant *Python* biblioteka *Flask*.

Programinio kodo spragų įrankio sistemoje naudojamos bibliotekos ir technologijos:

- Grafinei naudotojo sąsajai atvaizduoti naudota elementų biblioteka Material UI;
- Serverio dalies ir modelio integracijai naudotos Langchain ir ir LlamaCpp Python bibliotekos.

Realizuotas sistemos funkcionalumas apima failų pateikimą apdorojimui, patį apdorojimą statiniais analizatoriais ir sukurtu SVP modeliu, rezultatų ataskaitos pateikimą ir jų eksportavimą. Pats sistemos naudotojo gidas pateikiamas 4 priede.

5.2. Testavimo modelis, duomenys, rezultatai

Testavimo tikslas – užtikrinti, kad sukurtas įrankis dirba be trikdžių. Testuojama rankiniu būdu. Testavimo lentelės aprašomos 27 lentelėje ir 28 lentelėje.

27 lentelė. Sistemos konfigūracijos testavimo atvejis

Pradinės sąlygos		
Neišsaugotos pradinės programinės įrangos parinktys (nesukonfigūruota programinė įranga)		
Veiksmas	**Tikėtinas rezultatas**	**Gautas tikėtinas rezultatas**
Paspausti mygtuką „Configure it here“	Atvaizduojamas konfigūracijos langas	+
Įvesti nekorektišką konfigūraciją ir paspausti išsaugojimo mygtuką	Atvaizduojamas klaidos pranešimas	+
Įvesti korektišką konfigūraciją ir paspausti išsaugojimo mygtuką	Atvaizduojamas sėkmės pranešimas	+

28 lentelė. Sistemos naudojimo testavimo atvejis

Pradinės sąlygos		
Programinė įranga sukonfigūruota		
Veiksmas	**Tikėtinas rezultatas**	**Gautas tikėtinas rezultatas**
Įvesti klaidingus duomenis ir paspausti apdorojimo mygtuką	Atvaizduojamas klaidos pranešimas	+
Įvesti korektiškus duomenis ir paspausti apdorojimo mygtuką	Atvaizduojama pažaidų analizės ataskaita	+

Išvados

1. Statinės kodo analizės metodų analizės metu išsiaiškinta, kad mokslininkai bando mažinti spragų identifikavimo klaidas įvairiais būdais, vienas iš jų yra analizuoti kodą ir ataskaitas mašininio mokymosi metodais. Atlikta programų sistemų pažaidų prognostikos analizė atskleidė, kad bandyta atlikti analizę pasitelkiant duomenis iš įvairių pažaidų duomenų bazių, tačiau dažniausiai naudota yra NVD. Pastebėta tendencija imti subrinkinį iš NVD kaip duomenų rinkinį. Tačiau tai, kad pasirinkti duomenų rinkiniai neatspindi realių scenarijų, sukelia įvairių problemų, kaip pvz., ryškiai neatitinkantis tikslumas. Atlikta statinės kodo analizės ir programų sistemų pažaidų prognostikos susietumo galimybių analizė atskleidė, kad statinės programų sistemų analizės įrankių ir mašininio mokymosi metodų susietumo sprendimų paieška yra aktuali, o iš rezultatų galima spręsti, kad yra vietos pateikti geresnį sprendimą. Kai mažinamas neteisingai identifikuotų spragų skaičius, atsiranda rizika, kad išmesime tam tikrą iš tikro naudingą ir vertingą teisingą spragos atvejį. Pastebėta, kad rinkoje nėra labai daug įrankių kurie darytų spragų identifikavimą dirbtinio intelekto metodais.
2. Darbo modelio apmokymui pasirinktas duomenų rinkinys DiverseVul dėl savo realistiškumo, naujumo, standartizuoto pažaidų tipų žymėjimo CWE notacijoje ir funkcijų įvairovės. Dėl sąlyginai žemo klaidingų taikinio klasės atvejų kiekio ir naudojamo nekompiliuojamo statinės analizės metodo lyginamąjai eksperimento analizei pasirinktas *Flawfinder* statinis analizatorius.
3. Darbe pasiūlytas spragų programiniame kode aptikimo sprendimas, paremtas mašininiu mokymusi. Sprendimas susideda iš failo vektorizavimo dalies ir pažeidžiamumo požymio spėjimo. Failo vektorizavimui pasitelktas CodeLlama 2 7b kodo didysis kalbos modelis. Pažeidžiamumo požymio spėjimui pasitelkta atsitiktinių miškų architektūra dėl savo atsparumo didelio dimensionalumo duomenims ir atsparumo persimokymui.
4. Eksperimentiškai ištyrus sukurtų modelių kiekybinius rodiklius, nustatyta, kad geriausiai pasirodžiusi konfigūracija yra CodeLlama 7 milijardų hiperparametrų didysis kalbos modelis ir atsitiktinių miškų modelis, kurio saujos dydis medžiui parenkamas taikant kvadratinės šaknies iš požymių kiekio taisyklę. Ši modelio konfigūracija pasirodė geriau už statinį analizatorių *Flawfinder*, aptikdama teisingai daugiau nei tris kartus daugiau pažeidžiamų failų. Tiesa, statinis analizatorius korektiškai aptinka 21% daugiau sveikų failų nei minėtas modelis. Tačiau turint mintyje tai, kad klaidingai identifikuoti kaip nepažeidžiami failai gali būti daug žalingesni, šį skirtumą galima pateisinti. Taip pat, pastebėta, kad modeliai išmoksta skirtingomis *CWE* pažaidomis pažymėtus failus diferencijuoti nuo sveikų skirtingu gebėjimu. Pavyzdžiui, iš C*WE*-125 pažaidomis pažymėtų failų, modeliai sugebėjo atpažinti virš 80%, kai tuo tarpu *Flawfinder* tą atliko keturiskart prasčiau. O prasčiausiai identifikuoti modeliams sekėsi *CWE-Other* grupe pažymėtus failus. Tikėtina, kad dėl skirtingų *CWE* pažaidų tipų kiekiškumo ir jų skirtumų turinyje, modeliams apibendrinti mokymąsi pavyko sunkiau. Identifikuota, kad parinkus detekcijos slenkstį tokį, kad teisingų taikinio klasės jautrumas sutaptų su įrankio *Flawfinder* jautrumu, pastebėtas 64% klaidingų taikinio klasės atvejų sumažėjimas, lyginant su minimu statinės analizės įrankiu.
5. Atsižvelgiant į atliktą empirinį tyrimą, suprojektuotas, realizuotas demonstracinis kodo spragų aptikimo įrankis. Įrankis ištestuotas rankiniu testavimu pagal scenarijus. Nustatyta, kad įrankis veikia taip, kaip numatyta.
6. Darbo metu sukurtą modelį rekomenduojama naudoti su CodeLlama 2 modelio 7b variantu. Norint naudoti modelį primygtinai rekomenduojama vektorizuoti failą pasinaudojant grafinį procesorių arba tenzorinį procesorių dėl jų efektyvumo dirbant su matricomis. Rekomenduojama

parinkti detekcijos slenkstį 0.48, jei siekiama filtravimo arba 0.7, jei siekiama kruopštesnio teigiamų atvejų identifikavimo funkcionalumo.

Literatūros sąrašas

1. TELANG, R. and WATTAL, S. An Empirical Analysis of the Impact of Software Vulnerability Announcements on Firm Stock Price. *IEEE Transactions on Software* Engineering. 2007, 33(8), 544-557.
2. MAITRI, A. A Survey of Financial Losses Due to Malware. In *ICTCS '16: Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies.* Udaipur, 2016.
3. SAHA, R. K., et. al. Understanding the triaging and fixing processes of long lived bugs. *Information and Software Technology,* 65, 2015. 114-128.
4. ZHONG, H. and SU, Z. An Empirical Study on Real Bug Fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* Florence, 2015.
5. KAUR, A. and NAYYAR, R. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science.* 2020 171, 2023-2029.
6. AT&T. *Introduction to SAST* [Interaktyvus]. [Žiūrėta 2020-02-23]. Prieiga per: https://cybersecurity.att.com/blogs/security-essentials/introduction-to-sast
7. ALORAINI, B., et. al. An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software.* 2019, 158.
8. CROFT, R., et. al. An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing. In *ESEM '21: Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* New York, 2021.
9. NADEEM, M., et. al. High false positive detection of security vulnerabilities: a case study. In *ACM-SE '12: Proceedings of the 50th Annual Southeast Regional Conference.* New York, 2012.
10. DAS PURBA M., et al, Software Vulnerability Detection using Large Language Models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW).* Florence, 2023.
11. NGUYEN-DUC, A., et. al. On the adoption of static analysis for software security assessment–A case study of an open-source e-government project. *Computers & Security*. 2021, 111.
12. NIST. *Software Vulnerability - NIST CSRC* [Interaktyvus]. [Žiūrėta 2022-11-22]. Prieiga per: https://csrc.nist.gov/glossary/term/software_vulnerability
13. GARTNER. *Definition of SAST* [Interaktyvus]. [Žiūrėta 2022-11-28]. Prieiga per: https://www.gartner.com/en/information-technology/glossary/static-application-security-testing-sast
14. OWASP. *Static Code Analysis* [Interaktyvus]. [Žiūrėta 2022-11-28]. Prieiga per: https://owasp.org/www-community/controls/Static_Code_Analysis
15. VIEGA, J., et al. Token-based scanning of source code for security problems. *ACM Transactions on Information and System Security*. 2022, 5(3), 238-261.

16. CSRC. *False positive definition* [Interaktyvus]. [Žiūrėta 2022-11-28]. Prieiga per: https://csrc.nist.gov/glossary/term/false_positive

17. AKREMI, A. Software Security Static Analysis False Alerts Handling Approaches. *International Journal of Advanced Computer Science and Applications(IJACSA)*. 2021, 12(11).

18. LIPP, S., et al. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *ISSTA 2022: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* New York, 2022.

19. BROWNLEE, J. *Train-Test Split for Evaluating Machine Learning Algorithms* [Interaktyvus]. [Žiūrėta 2022-11-30]. Prieiga per: https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/

20. ALQAHTANI S. S. A study on the use of vulnerabilities databases in software engineering domain. *Computers & Security*. 2022, 116.

21. NVD. *NVD Vulnerability Database* [Interaktyvus]. [Žiūrėta 2022-11-30]. Prieiga per: https://nvd.nist.gov/vuln/search

22. OWASP. *OWASP Vulnerabilities* [Interaktyvus]. [Žiūrėta 2022-11-30]. Prieiga per: https://owasp.org/www-community/vulnerabilities/

23. CVE. *CVE details* [Interaktyvus]. [Žiūrėta 2022-11-30]. Prieiga per: https://www.cvedetails.com/

24. CWE. *Common Weakness Enumeration* [Interaktyvus]. [Žiūrėta 2022-11-30]. Prieiga per: https://cwe.mitre.org/

25. ZHANG, S., et al. An Empirical Study on Using the National Vulnerability Database to Predict Software Vulnerabilities. In *International Conference on Database and Expert Systems Applications.* Toulouse, 2011.

26. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Official Common Platform Enumeration (CPE) Dictionary* [Interaktyvus]. [Žiūrėta 2022-11-28]. Prieiga per: https://nvd.nist.gov/products/cpe

27. NMAP. *Common Platform Enumeration (CPE)* [Interaktyvus]. [Žiūrėta 2022-11-28]. Prieiga per: https://nmap.org/book/output-formats-cpe.html

28. NIST. *NVD - Vulnerability Metrics* [Interaktyvus]. [Žiūrėta 2022-11-28]. Prieiga per: https://nvd.nist.gov/vuln-metrics/cvss

29. FORAIN, I., et al. Towards System Security: What a Comparison of National Vulnerability Databases Reveals. In *2022 17th Iberian Conference on Information Systems and Technologies (CISTI).* Madrid, 2022.

30. MORIUCHI, P. and LADD, B. China's Ministry of State Security Likely Influences National Network Vulnerability Publications [Interaktyvus]. 2017. [Žiūrėta 2022-11-28]. Prieiga per: https://www.recordedfuture.com/chinese-mss-vulnerability-influence

31. YOSIFOVA, V., et al. Predicting Vulnerability Type in Common Vulnerabilities and Exposures (CVE) Database with Machine Learning Classifiers. In *2021 12th National Conference with International Participation (ELECTRONICA).* Sofia, 2021.

32. HARER. A. Harer and et. al. Automated software vulnerability detection with machine learning. *arXiv.* 2018.

33. KIM, Y. Convolutional Neural Networks for Sentence Classification. *ArXiv.* 2014.

34. DEEPAI, *Natural Language Processing definition* [Interaktyvus]. [Žiūrėta 2022-11-28]. Prieiga per: https://deepai.org/machine-learning-glossary-and-terms/natural-language-processing

35. ZIEMS, N. and WU, S. Security Vulnerability Detection Using Deep Learning Natural Language Processing. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS).* Vancouver, 2021.

36. HOCHREITER, S. and SCHMIDHUBER J. Long Short-Term Memory. *Neural Computation.* 1997, 9(8), 1735-1780.

37. JABEEN, G., et al. Machine learning techniques for software vulnerability prediction: a comparative study. *Applied Intelligence*. 2022, 52, 17614–17635.

38. KHARE, A., et al. *ArXiV: Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities* [Interaktyvus]. 2023. [Žiūrėta 2022-11-28]. Prieiga per: https://arxiv.org/abs/2311.16169

39. NOEVER D. A. *Can Large Language Models Find And Fix Vulnerable Software?* [interaktyvus]. ArXiv, 2023. [Žiūrėta 2022-11-28]. Prieiga per: https://doi.org/10.48550/arXiv.2308.10345

40. THAPA, C., et al. *Transformer-Based Language Models for Software Vulnerability Detection* [Interaktyvus]. ArXiv, 2022. [Žiūrėta 2022-11-28]. Prieiga per: https://doi.org/10.48550/arXiv.2204.03214

41. LIN, G., et al. Distilled and Contextualized Neural Models Benchmarked for Vulnerable Function Detection. *Mathematics.* 2022, 10 (23).

42. FU, M. and TANTITHAMTHAVORN C. LineVul: a transformer-based line-level vulnerability prediction. In *MSR '22: Proceedings of the 19th International Conference on Mining Software Repositories.* New York, 2022.

43. CHAKRABORTY, S., et al. Deep Learning Based Vulnerability Detection: Are We There Yet?. *IEEE Transactions on Software Engineering*. 2022, 48(9), 3280-3296.

44. CROFT, R., et al. Data Preparation for Software Vulnerability Prediction: A Systematic Literature Review. *EEE Transactions on Software Engineering*. 2023, 49(3), 1044 - 1063.

45. NIST. *Static Analysis Tool Exposition (SATE) IV* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://www.nist.gov/itl/ssd/software-quality-group/static-analysis-tool-exposition-sate-iv

46. RUSSEL, R., et al. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA).* Orlando, FL, USA, 2018.

47. RUSSEL R. and KIM, L. *Draper VDISC Dataset - Vulnerability Detection in Source Code* [Interaktyvus]. 2021. [Žiūrėta 2023-06-20]. Prieiga per: https://osf.io/d45bw/

48. GRAHN, D. and ZHANG J. An Analysis of C/C++ Datasets for Machine Learning-Assisted Software Vulnerability Detection. In *Proceedings of the Conference on Applied Machine Learning for Information Security.* 2021.

49. CHAKRABORTY, S. *ReVeal (Chromium and Debian vulnerability data)* [Interaktyvus]. 2020. [Žiūrėta 2023-06-20]. Prieiga per: https://drive.google.com/drive/folders/1KuIYgFcvWUXheDhT--cBALsfy1I4utOy.

50. CHEN, Y., et al. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *RAID '23: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses.* Hong Kong, 2023.

51. KOC, U., et al. Learning a classifier for false positive error reports emitted by static code analysis tools. In *MAPL 2017: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages.* New York, 2017.

52. YOON, J., et al. Reducing False Alarms from an Industrial-Strength Static Analyzer by SVM. In *2014 21st Asia-Pacific Software Engineering Conference.* Jeju, 2014.

53. RIBEIRO, A., et al. Ranking warnings from multiple source code static analyzers via ensemble learning. In *OpenSym '19: Proceedings of the 15th International Symposium on Open Collaboration.* New York, 2019.

54. LEE, S., et al. Classifying False Positive Static Checker Alarms in Continuous Integration Using Convolutional Neural Networks. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST).* XI'an, 2019.

55. CHECKMARX. *Checkmarx SAST* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://checkmarx.com/cxsast-source-code-scanning/

56. CHECKMARX. *SAST User guide*. [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://checkmarx.com/resource/documents/en/34965-46398-sast-user-guide.html

57. SONARSOURCE. *SonarQube* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://www.sonarsource.com/products/sonarqube

58. SONARSOURCE. *SonarQube Documentation* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://docs.sonarqube.org/latest/

59. SNYK. *Snyk Coda* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://snyk.io/product/snyk-code/

60. SNYK. *Snyk User Documentation* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://docs.snyk.io/

61. VERACODE. *Veracode static analysis* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://www.veracode.com/products/binary-static-analysis-sast

62. VERACODE. *Getting started with Veracode* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://docs.veracode.com/r/c_getting_started

63. DEEPSOURCE. *DeepSource SAST* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://deepsource.io/platform/sast/

64. DEEPSOURCE. *DeepSource documentation* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://deepsource.io/docs/.

65. CHECKMARX. *Checkmarx Knowledge Center - Improving Security Analysis* [interaktyvus]. 2021. [Žiūrėta 2023-06-20]. Prieiga per: https://checkmarx.atlassian.net/wiki/spaces/KC/pages/5406753/Improving+Security+Analysis

66. WAGNER, D. *DiverseVul* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://drive.google.com/file/d/12IWKhmLhq7qn5B_iXgn5YerOQtkH-6RG/view.

67. XIAN, Z., et al. TransformCode: A Contrastive Learning Framework for Code Embedding via Subtree Transformation. 2023.

68. KARAMPATSIS R. M. and SUTTON, C. SCELMo: Source Code Embeddings from Language Models. 2020.

69. RADFORD, A., et al. Improving language understanding by generative pre-training. 2018.

70. VASWANI, A., et al. Attention is all you need. In *31st Conference on Neural Information Processing Systems (NIPS 2017).* Long Beach, 2017.

71. TOUVRON, H., et al. LLaMA: Open and Efficient Foundation Language Models. 2023.

72. GERGANOV, G. *Llama.cpp gthub* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://github.com/ggerganov/llama.cpp

73. LANGCHAIN. *Llama.cpp* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://python.langchain.com/v0.1/docs/integrations/llms/llamacpp

74. BREIMAN, L. Random Forests. *Machine Learning.* 2001, 45, 5-32.

75. FIRST. *Common Vulnerability Scoring System version 3.1: Specification Document* [Interaktyvus]. [Žiūrėta 2023-06-20]. Prieiga per: https://www.first.org/cvss/specification-document

76. WILLIAMS, J. *OWASP Risk Rating Methodology* [Interaktyvus]. OWASP. [Žiūrėta 2023-06-20]. Prieiga per: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology.

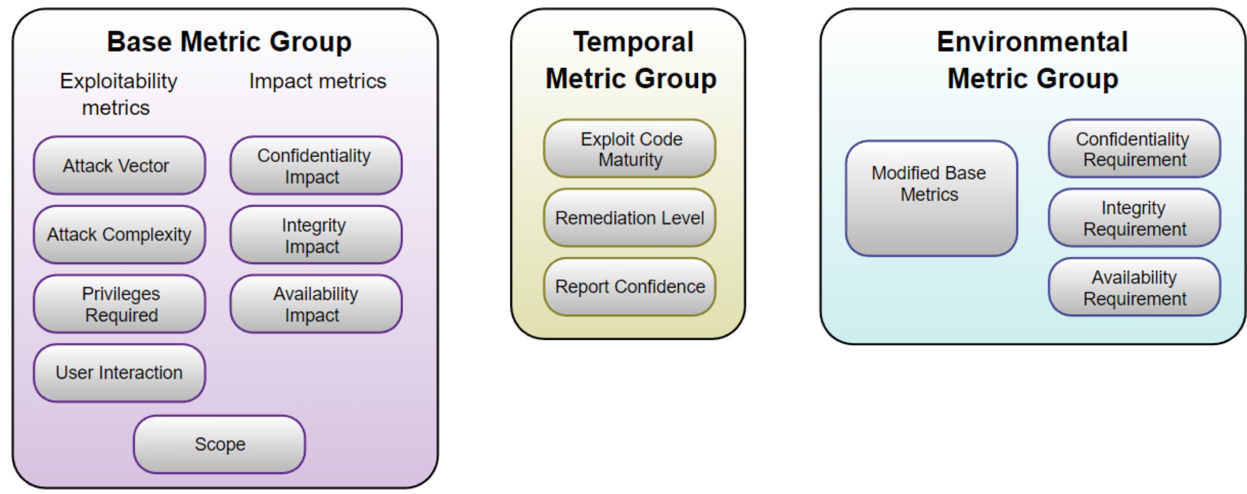
77. WALDEN, J., et al. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *2014 IEEE 25th International Symposium on Software Reliability Engineering.* Naples, Italy, 2014.

78. WALDEN, J. *PHP Security vulnerability dataset* [Interaktyvus]. [Žiūrėta 2023-06-20] Prieiga per: https://seam.cs.umd.edu/webvuldata/

Priedai

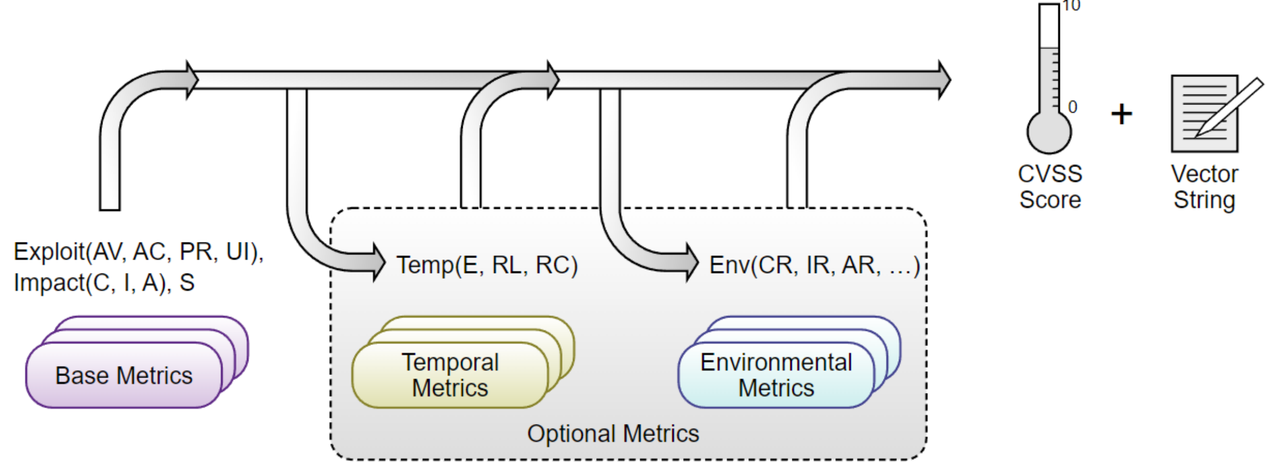
1 Priedas. Common Vulnerability Scoring System (CVSS) analizė

CVSS – bendroji pažaidų vertinimo sistema (angl. *Common Vulnerability Scoring System*) yra metodas skirtas kokybiniam pažaidos sunkumo įvertinimui. CVSS susideda iš trijų metrikų grupių – bazinės, laikinės ir aplinkos. CVSS įvertis reprezentuojamas kaip vektorių eilutė, t.y. suspausta tekstinė verčių, naudotų išvesti įvertį, reprezentacija. Pagal CVSS 3.1 specifikacijos versiją, numatytos trys aukščiau išvardytos pažaidų metrikų grupės, kurios iliustruojamos 36 pav.:



36 pav. CVSS specifikacijos nurodomos pažaidų metrikų grupės [75]

Pagal pateiktą iliustraciją, bazinė metrikų grupė skirstoma į išnaudojamumo (angl. *exploitability*) ir poveikio (angl. *impact*) metrikas. Išnaudojamumo metrikos atspindi pažeidžiamo sistemos komponento charakteristikas. Poveikio metrikos apibūdina pasėkmes paveiktam sistemos komponentui sėkmingo pažeidžiamo komponento išnaudojimo atveju. Nepaisant to, kad pažeidžiamas komponentas yra paprastai programinė, aparatinė ar tinklo įranga, poveikis gali būti juntamas už pažeidžiamos sistemos ribų. Šį poveikį už sistemos ribų apibūdina apimties (angl. *scope*) metrika. Šios metrikos vertinamos pagal specifinę metodiką. Metrikų vertinimo metodika pateikiama 37 pav.:



37 pav. CVSS specifikacijos nurodomas metrikų vertinimo algoritmas [75]

Pagal pateiktą iliustraciją, bazinių metrikų formulė susideda iš dviejų mažesnių formulių ir apimties metrikos vertinimo. CVSS specifikacija pateikia patogias vertinimo lenteles iš kurių susintezuota viena (pateikiama 29 lentelėje).

29 lentelė. CVSS specifikacijos metrikų vertės ir jų aprašai [75]

Metrika	**Metrikos vertinimas**	**Metrikos vertinimo aprašas**
Pažeidžiamo komponento išnaudojamumas		
Atakos vektorius (AV)	Network (N)	Pažeidžiamas komponentas pasiekiamas tinklu ir gali būti atakuojamas iš bet kurio interneto taško. Tam, kad komponentas būtų vertinamas kaip „išnaudojamas per nuotolį“, ataka turi galėti būti įvykdyta iš interneto prieigos punkto, nutolusio nuo atakuojamojo taško per vieną tinklo mazgą (pvz., maršrutizatorių). Atakų pavyzdžiai: *DoS* (angl. *Denial of Service*).
	Adjacent (A)	Pažeidžiamas komponentas pasiekiamas tinklu, bet atakos pobūdis apriboja ataką protokolo lygmenyje arba įrenginio topologiniame lygmenyje tinkle. Ataka privalo būti vykdoma iš to paties fizinio (pvz. *Bluetooth*), loginio (vietinis IP potinklis) arba iš saugaus ar kitaip apriboto administracinio domeno (MPLS arba saugus VPN į administracinę tinklo zoną). Šalutinių atakos vektorių atakų pavyzdžiai yra ARP protokolo klastotė arba kaimyno atpažinties protokolo (angl. *neighbor discovery protocol, NDP*) atakos.
	Local (L)	Pažeidžiamas komponentas tinklu nepasiekiamas ir atakos vektorius yra per skaitymo/rašymo/vykdymo galimybes. Galimi du variantai: ataka vykdoma pasiekus sistemą lokaliai (arba per SSH protokolą), arba naudojamasi kito žmogaus prieiga prie sistemos atakos įgyvendinimui (pvz., socialinės inžinerijos atakos).
	Physical (P)	Ataka reikalauja fizinio sąlyčio ar manipuliacijos su pažeidžiamu komponentu. Atakos pavyzdžiai gali būti piktosios tarnaitės ataka, periferinių įrenginių atakos (USB tiesioginė atminties prieiga).

Atakos sudėtingumas (AC)	Low (L)	Ypatingos prieigos arba lengvinančios aplinkybės neegzistuoja. Ataka gali būti atliekama pakartotinai sėkmingai tam pačiam pažeidžiamam komponentui.
	High (H)	Sėkminga ataka priklauso nuo faktorių už atakuojančios pusės kontrolės. Atakuojanti pusė privalo gan stipriai pasirengti prieš vykdant ataką.
Reikalaujamos privilegijos (PR)	None (N)	Atakuojančiai pusei nereikia prieigos prie pažeidžiamos sistemos nustatymų ar failų sėkmingai atakai įvykdyti.
	Low (L)	Užpuolikui užtenka bazinių privilegijų, kurios suteikia prieigą prie nejautrų resursų arba failų kuriuos turi naudotojas.
	High (H)	Užpuolikui reikia administracinės prieigos prie pažeidžiamo komponento ir prieigos prie visų komponento failų, nustatymų.
Naudotojo įsikišimas (UI)	None (N)	Pažeidžiama sistema gali būti išnaudojama be naudotojo įsikišimo
	Required (R)	Sistemos pažaidos išnaudojimas reikalauja naudotojo įsikišimo.
Pokyčio apimtis		
Apimtis (S)	Unchanged (U)	Išnaudota pažaida gali paveikti tik tuos resursus, kurie yra toje pačioje saugumo sistemoje.
	Changed (C)	Išnaudota pažaida gali paveikti resursus už pažeistos sistemos ribų.
Poveikis paveiktam komponentui		
Konfidencialumas (C)	High (H)	Visiškas sistemoje esančių duomenų konfidencialumo praradimas sėkmingos atakos atveju arba ataka pasiekia ribotą duomenų kiekį, bet šie sukelia rimtų rizikų.
	Low (L)	Dalinis konfidencialumo praradimas. Dalis informacijos pasiekiama, bet užpuolikas neturi kontrolės, kokia informacija pasiekiama. Pasiekta informacija nesukelia tiesioginio ir/ar rimto poveikio pažeistam komponentui.
	None (N)	Konfidencialumo neprarandama.
Sistemos vientisumas (I)	High (H)	Visiškas vientisumo praradimas arba visiškas apsaugos praradimas. Užpuolikas gali modifikuoti norimus failus, saugojamus paveikto komponento arba gali tik dalį, tačiau modifikuojami failai sukelia rimtų pasekmių.
	Low (L)	Duomenų modifikacijos įmanomos, bet užpuolikas neturi pasekmės kontrolės arba modifikacija ribota.
	None (N)	Neprarandama vientisumo.
Pasiekiamumas (A)	High (H)	Visiškas pasiekiamumo praradimas. Užpuolikas gali uždrausti pilnai prieigą prie paveiktų komponentų, arba dalinė prieiga prarasta, tačiau tai kelia didelių pasekmių.
	Low (L)	Pajėgumai sumažinti, resurso prieigos trūkiai. Esant pakartotinei pažaidos išnaudojimo galimybei, prieiga nelieka visiškai sutrikdyta.
	None (N)	Poveikio pasiekiamumui nėra.

Šios metrikos, kurios išvardintos lentelėje vertinamos pagal iš anksto numatytas vertes ir formules. Bazinis įvertis priklauso nuo trijų komponenčių: tarpinio poveikio įverčio (angl. *Impact Sub-Score, ISS*), poveikio (*Impact*) ir išnaudojamumo (*Exploitability*). Jų formulės pateikiamos 13-16 lygtyse:

$$ISS = 1 - \big((1 - C) \times (1 - I) \times (1 - A)\big)$$

13 lygtis. Tarpinio poveikio įverčio formulė [75]

$$Impact = \begin{cases} 6.42 \times ISS, \quad S = "U" \\ 7.52 \times (ISS - 0.029) - 3.25 \times (ISS - 0.02)^{15}, \quad S = "C" \end{cases}$$

14 lygtis. Poveikio pažeidžiamam komponentui skaičiavimo formulė [75]

$$Exploitability = 8.22 \times AV \times AC \times PR \times UI$$

15 lygtis. Pažeidžiamo komponento išnaudojamumo skaičiavimo formulė [75]

$$BaseScore = \begin{cases} 0, \quad Impact \leq 0 \\ \lceil \min(Impact + Exploitability, 10) \rceil, \quad S = "U" \\ \lceil \min(1.08 \times (Impact + Exploitability), 10) \rceil, \quad S = "U" \end{cases}$$

16 lygtis. Bazinių metrikų įverčio skaičiavimo formulė [75]

Šiose formulėse:

- C – poveikio komponento konfidencialumui vertinimas;
- I – poveikio komponento vientisumui (angl. *integrity*) vertinimas;
- A – poveikio komponento pasiekiamumui (angl. *availability*) vertinimas;
- AV – atakos vektoriaus vertinimas;
- AC – atakos sudėtingumo vertinimas;
- PR – reikalaujamų privilegijų vertinimas;
- UI – naudotojo įsikišimo vertinimas;
- S – poveikio apimties vertinimas.

Šie vertinimai turi konstantines vertes, kurios aprašytos specifikacijoje (30 lentelė):

30 lentelė. CVSS bazinių metrikų įverčių skaitinės reprezentacijos [75]

Metrika	Metrikos vertinimas	Metrikos vertinimo skaitinė reprezentacija
Pažeidžiamo komponento išnaudojamumas		

Atakos vektorius (AV)	Network (N)	0.85
	Adjacent (A)	0.62
	Local (L)	0.55
	Physical (P)	0.2
Atakos sudėtingumas (AC)	Low (L)	0.77
	High (H)	0.44
Reikalaujamos privilegijos (PR)	None (N)	0.85
	Low (L)	0.62 (jeigu poveikio apimtis vertinama kaip pakitusi – 0.68)
	High (H)	0.27 (jeigu poveikio apimtis vertinama kaip pakitusi – 0.5)
Naudotojo įsikišimas (UI)	None (N)	0.85
	Required (R)	0.62
Poveikis paveiktam komponentui		
Konfidencialumas (C)	High (H)	0.56
	Low (L)	0.22
	None (N)	0
Sistemos vientisumas (I)	High (H)	0.56
	Low (L)	0.22
	None (N)	0
Pasiekiamumas (A)	High (H)	0.56
	Low (L)	0.22
	None (N)	0

2 Priedas. OWASP rizikos vertinimo metodikos analizė

OWASP organizacija siūlo savo būdą vertinti programų sistemų pažaidų rizikas – OWASP rizikos vertinimo metodiką (angl. *OWASP Risk Rating Methodology*) [76]. OWASP metodikos įvertis skaičiuojamas pagal formulę:

$$R = LI$$

17 lygtis. OWASP rizikos skaičiavimo formulė [76]

Čia:

- R – rizikos įvertis;
- L – rizikos pasiteisinimo (angl. *likelihood*) įvertis;
- I – rizikos poveikio (angl. *Impact*) įvertis.

Rizikos pasiteisinimas vertinamas pagal kelis faktorius.

Pirmas faktorius yra pavojaus agentas (angl. *threat agent*). Pavojaus agentu vadinamas užpuolikas arba užpuolikų grupė. Pavojaus agentas vertinamas pagal šiuos aspektus (kiekvienam aspektui šiame ir sekančiuose faktoriuose turėtų būti pasirenkama pati blogiausia opcija):

- Sugebėjimai. Pavojaus agento sugebėjimai vertinami pagal šią skalę:
  - 1 – neturintis techninių žinių;
  - 3 – turintis kažkiek techninių žinių;
  - 5 – pažengęs kompiuterio naudotojas;
  - 6 – kompiuterių tinklų ir programavimo žinių turintis naudotojas;
  - 9 – laužimosi į sistemas žinios.
- Motyvas. Kiek motyuotas pavojaus agentas yra surasti ir išnaudoti pažaidą?
  - 1 – naudos iš to jokios arba mažai;
  - 4 – galima nauda;
  - 9 – didelė nauda.
- Galimybė. Kokių resursų reikia pavojaus agentui sėkmingam spragos išnaudojimui?
  - 0 – pilna prieiga;
  - 4 – ypatinga prieiga;
  - 7 – nežymi prieiga;
  - 9 – prieigos nereikia.
- Dydis. Koks pavojaus agento grupės dydis?
  - 2 – sistemų kūrėjai, administratoriai;
  - 4 – intraneto naudotojai;
  - 5 – verslo partneriai;
  - 6 – autentifikuoti naudotojai;
  - 9 – anoniminiai interneto naudotojai.

Kitas faktorius kurį reikėtų įvertinti yra pačios pažaidos faktorius. Šis faktorius susideda iš šių aspektų:

- Atradimo lengvumas.
  - 1 – beveik neįmanoma;
  - 3 – sunku;
  - 7 – lengva;
  - 9 – atrandama automatiniais įrankiais.
- Išnaudojamumo lengvumas.
  - 1 – teorinis;
  - 3 – sunkus;
  - 5 – lengvas;

    - 9 – išnaudojama automatiniais įrankiais.
- Žinomumas pavojaus agentui.
    - 1 – nežinoma;
    - 4 – paslėpta;
    - 6 – akivaizdi;
    - 9 – viešai žinoma informacija.
- Atpažinimas įsilaužimo atveju.
    - 1 – aktyvi detekcija;
    - 3 – užfiksuojama žurnale ir pranešama naudotojui;
    - 8 – užfiksuojama, bet nepranešama;
    - 9 – nefiksuojama niekur.

Vertinant poveikį, vertinami du faktoriai – techninis ir verslo. Techninio poveikio faktorius vertinamas pagal šiuos aspektus:

- Konfidencialumo praradimas. Kiek duomenų atskleista ir kokie?
    - 2 – minimalus nekritinės informacijos kiekis;
    - 6 – minimalus kritinės informacijos kiekis arba didelis nekritinės informacijos kiekis;
    - 7 – didelis kritinės informacijos kiekis;
    - 9 – visa informacija.
- Vientisumo praradimas.
    - 1 – minimalus truputį sugadintų duomenų kiekis;
    - 3 – minimalus rimtai sugadintų duomenų kiekis;
    - 5 – daug truputį sugadintų duomenų;
    - 7 – daug rimtai sugadintų duomenų;
    - 9 – visi duomenys visiškai sugadinti;
- Prieigos praradimas
    - 1 – minimalūs antriniai servisai pertraukti;
    - 5 – pirminiai servisai minimaliai pertraukti arba rimtai pertraukti antriniai servisai;
    - 7 – pirminiai servisai rimtai pertraukti;
    - 9 – visi servisai visiškai prarasti;
- Atsekamumo praradimas. Kiek pavyksta atsekti pavojaus agento veiksmus iki individo?
    - 1 – pilnai atsekama;
    - 7 – galimai atsekama;
    - 9 – visiškai neatsekama (anonimiška).

Verslo faktoriai reikalauja gilaus supratimo apie įmonę. Verslo rizika dažniausiai vertinama individualiai, tačiau OWASP pateikia bendrinius aspektus, kurie svarbūs didžiajai daugumai verslo organizacijų:

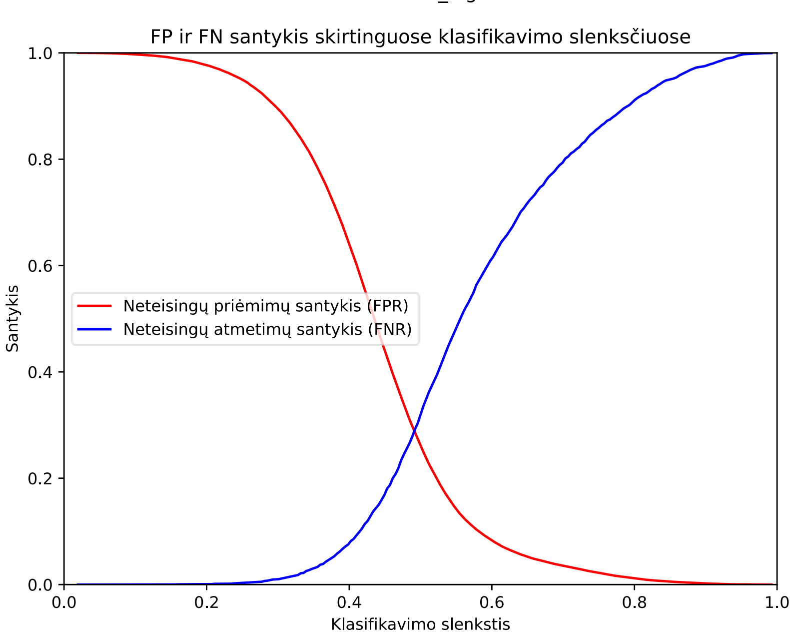
- Finansinė žala
  - 1 – mažiau nei kainuoja sutaisyti saugumo spragą;
  - 3 – minimalus smūgis metiniam pelnui;
  - 7 – pastebimas smūgis metiniam pelnui;
  - 9 - bankrotas
- Reputacinė žala
  - 1 – minimali žala
  - 4 – didžiųjų klientų praradimas
  - 5 – prestižo praradimas
  - 9 – ryški žala prekiniam ženklui
- Verslo pažeidimai sukelti saugumo standartų neatitikties
  - 2 – maži pažeidimai
  - 5 – aiškūs pažeidimai
  - 7 – aukšto profilio pažeidimai
- Privatumo pažeidimas
  - 3 – vienas individas
  - 5 – šimtai individų
  - 7 – tūkstančiai individų
  - 9 – milijonai individų.

Suvertinus šiuos faktorius, išvedamas vidurkis ir paskaičiuojama pagal formulę. Rezultatas pateikiamas matricoje:

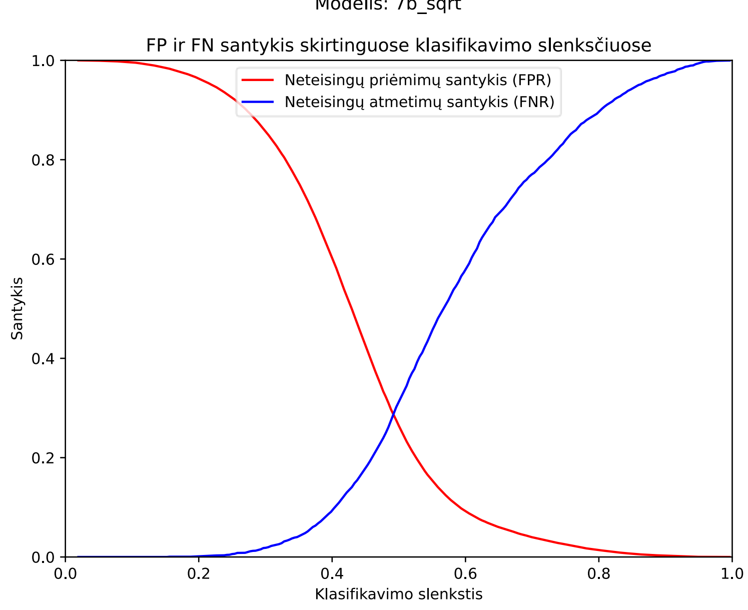
31 lentelė. OWASP metodologijos rizikos vertinimo matrica [76]

	Bendra rizika			
Poveikis	**Aukštas**	Vidutinis	Aukštas	Kritinis
	Vidutinis	Žemas	Vidutinis	Aukštas
	Žemas	Nulinis	Žemas	Vidutinis
		Žemas	**Vidutinis**	**Aukštas**
	Tikėtinumas			

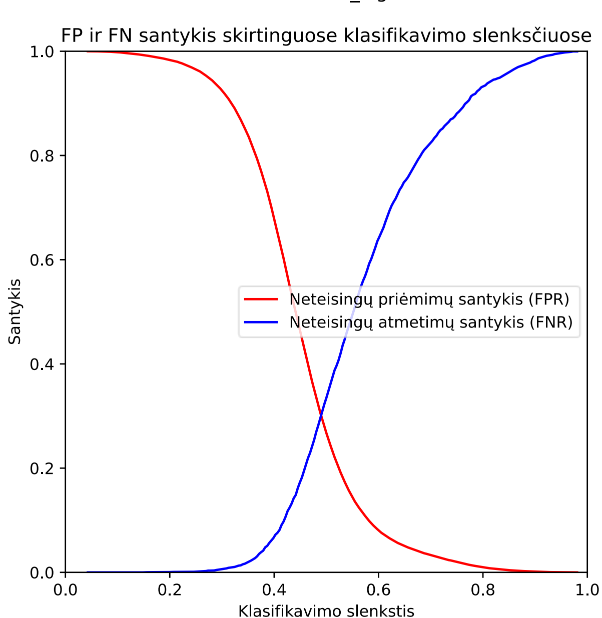
3 Priedas. EER slenksčio radimo grafikai modelių klasifikavimo slenksčio nustatymui



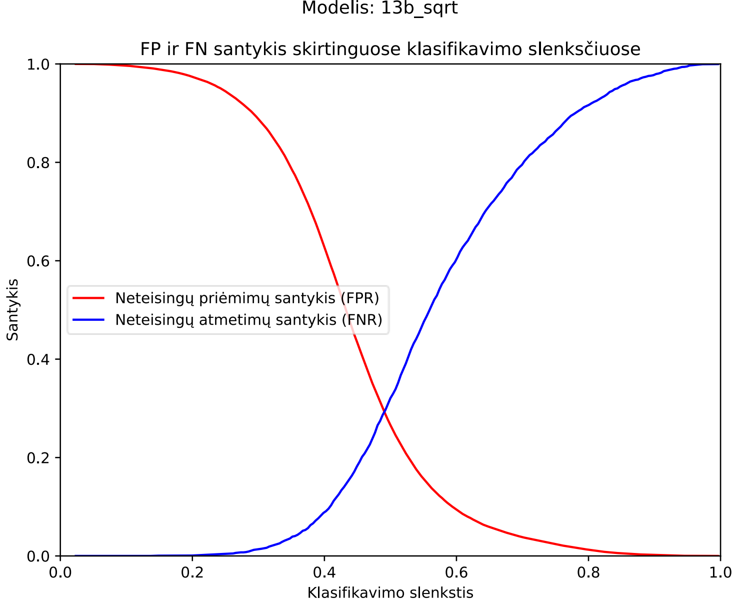
38 pav. 7b_log2 modelio neteisingų priėmimų ir atmetimų santykio kreivės



39 pav. 7b_sqrt modelio neteisingų priėmimų ir atmetimų santykio kreivės



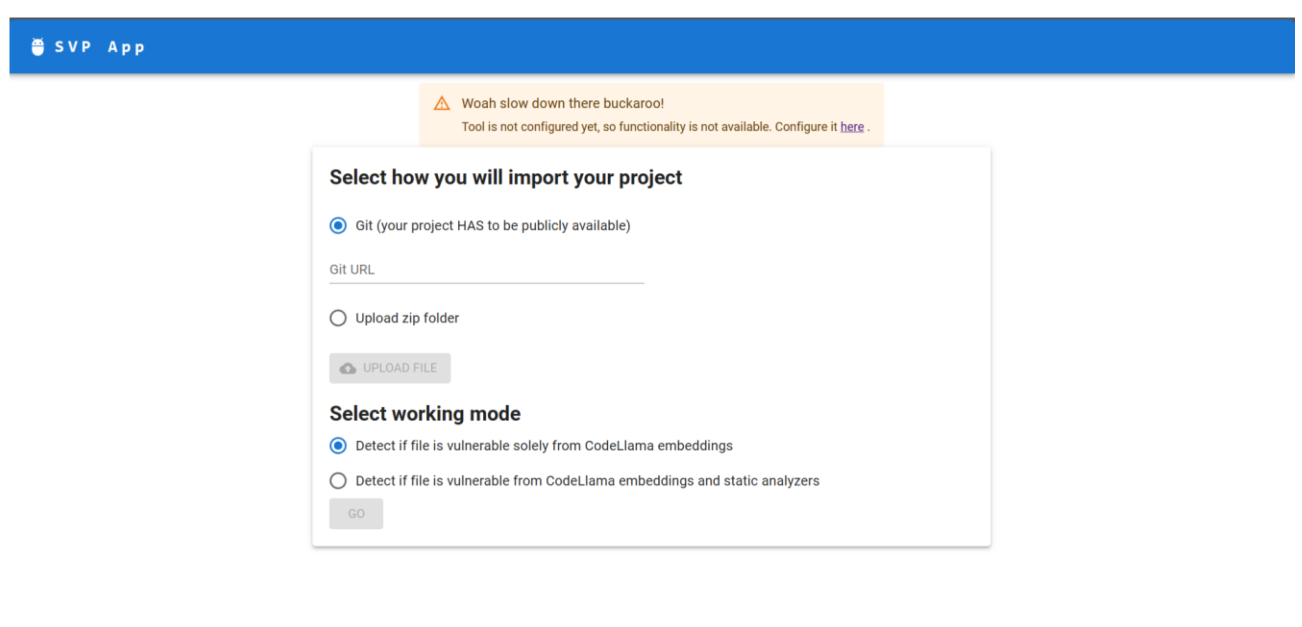
40 pav. 13b_log2 modelio neteisingų priėmimų ir atmetimų santykio kreivės



41 pav. 13b_log2 modelio neteisingų priėmimų ir atmetimų santykio kreivės

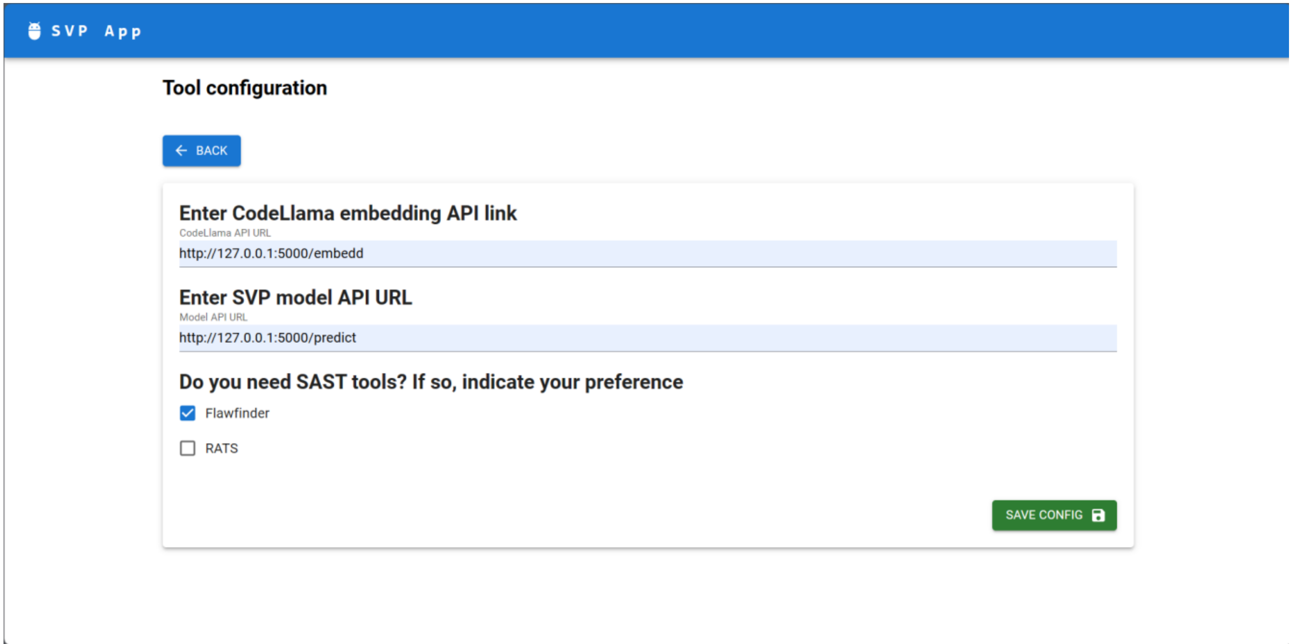
4 Priedas. Naudotojo vadovas

Norint naudotis sistema, reikia nueiti į jos namų puslapį. Jei sistema naudojamasi pirmą kartą, sistema paprašys ją sukonfigūruoti, kitu atveju jokių veiksmų atlikti nebus galima (žr. 42 pav.)



42 pav. Nesukonfigūruotos programinio kodo spragų aptikimo įrankio namų puslapis

Paspaudus nuorodą, atveriamas įrankio konfigūracijos langas. Pavyzdinė konfigūracija pateikiama 43 pav., bet esant poreikiui, ją galima keisti.

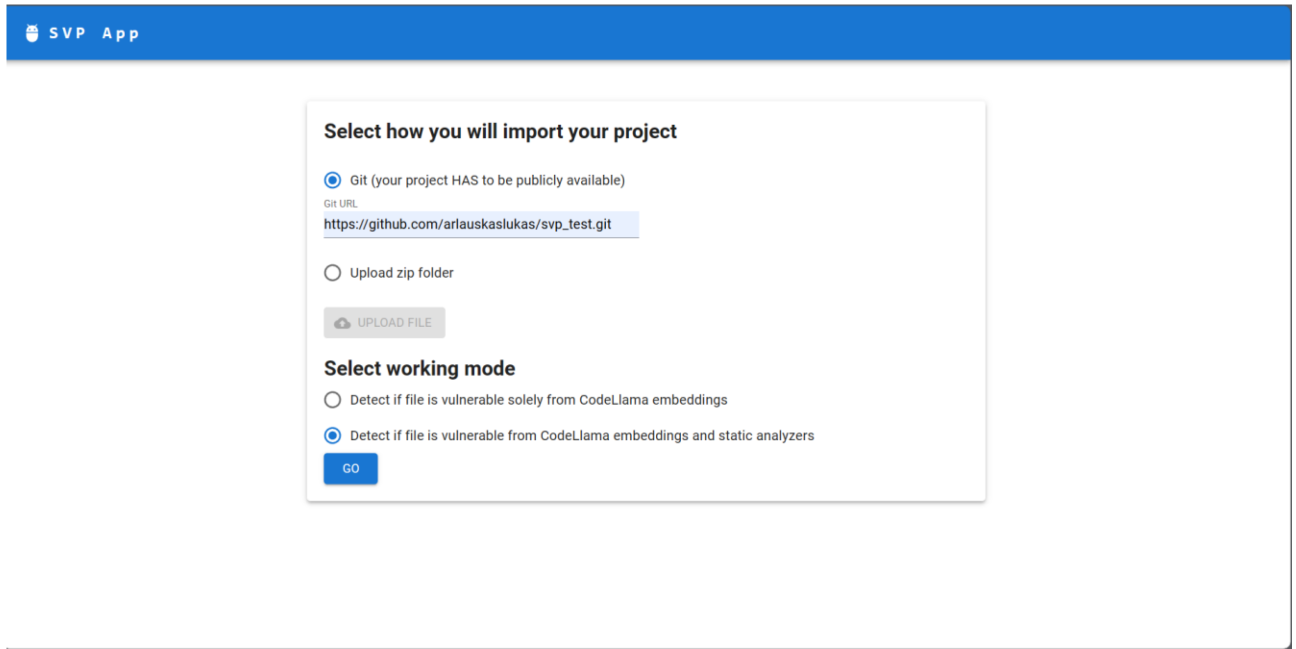


43 pav. Įrankio konfigūracijos langas

Sėkmingai išsaugojus rezultatus, sistema atvaizduoja sėkmės pranešimą (žr. 44 pav.)

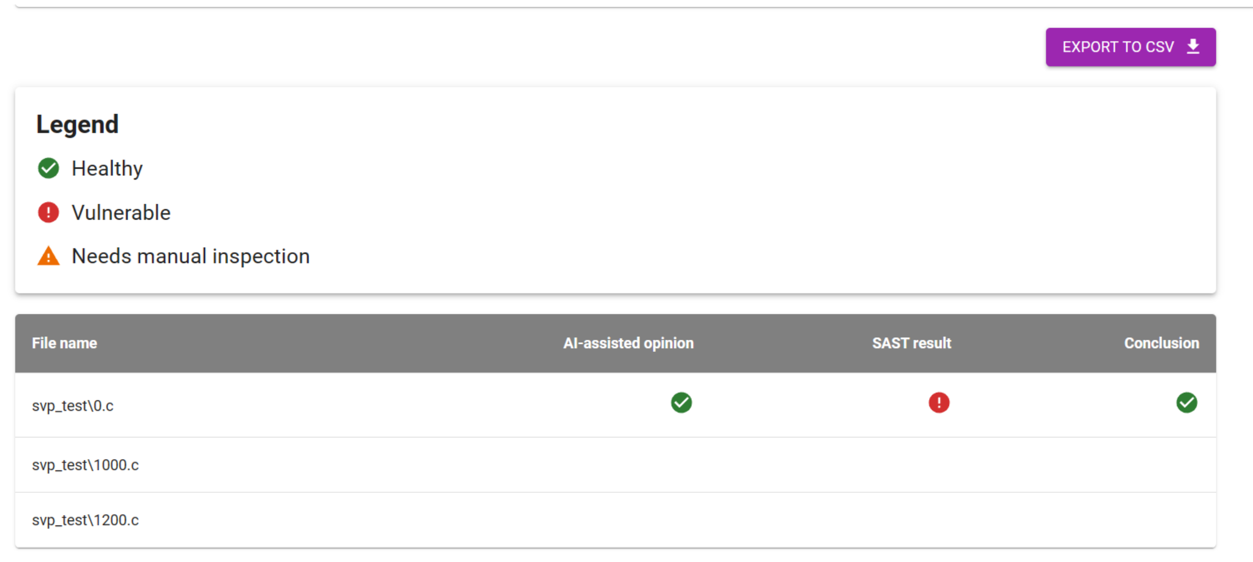
44 pav. Įrankio konfigūracijos sėkmės pranešimas

Tada, sugrįžus į sukonfigūruotos sistemos namų puslapį, galima atlikti veiksmus (žr. 45 pav.)



45 pav. Sukonfigūruotos sistemos namų puslapis

Paspaudus mygtuką GO, užkraunamas statinės analizės rezultatų puslapis (žr. 46 pav.). Failai kurie identifikuoti kaip sveiki, žymimi žalia ikonėle, tie, kurie ne – raudona. Failai esantys netoli klasifikavimo slenksčio žymimi papildomai geltona ikonėle. Esant poreikiui, šiuos failus galima eksportuoti CSV formatu.



46 pav. Statinės analizės rezultatų puslapis