Kauno technologijos universitetas

Informatikos fakultetas

Programų pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą

Baigiamasis magistro studijų projektas

Lukas Jokubauskas

Projekto autorius

Prof. Jevgenijus Toldinas

Vadovas

Kaunas, 2022

Kauno technologijos universitetas

Informatikos fakultetas

Programų pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą

Baigiamasis magistro studijų projektas

Informacijos ir informaciniu technologijų sauga (6211BX008)

Lukas Jokubauskas

Projekto autorius

Prof. Jevgenijus Toldinas

Vadovas

Doc. Nerijus Morkevičius

Recenzentas

Kaunas, 2022

Kauno technologijos universitetas

Informatikos fakultetas

Lukas Jokubauskas

Programų pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą

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 Jokubauskas

Patvirtinta elektroniniu būdu

Jokubauskas, Lukas. Programų pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą. Magistro baigiamasis projektas. Vadovas prof. Jevgenijus Toldinas; Kauno technologijos universitetas, Informatikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Informatikos inžinerija (Informatikos mokslai).

Reikšminiai žodžiai: pažeidžiamumų aptikimas, dinaminė analizė, RPC, gRPC, miglotoji logika.

Kaunas, 2022. 80 p.

Santrauka

Didėjantis programinės įrangos poreikis siekiant automatizuoti dalį žmogaus gyvenimo kasdieninių veiksmų daro programas vis labiau kompleksiškas. Programinė įranga dažnai susideda iš daugybės komponentų, pavyzdžiui, išorinės sąsajos (angl. *frontend*) ir vidinės sąsajos (angl. *backend*). Šie komponentai turi vykdyti duomenų mainus. Daugybė modernios asmeninių kompiuterių PĮ naudoja tokį patį saitynų architektūrinį sprendimą, kai veikia atskiras išorinės sąsajos ir vidinės sąsajos procesas. Komunikacijai tarp procesų yra naudojami operacinės sistemos suteikiami tarp-procesinės komunikacijos mechanizmai arba tam tikri karkasai. Netinkamai įgyvendinta tarp-procesinė komunikacija gali sukelti programinio kodo pažeidžiamumus, kuriuos galima išnaudoti piktavališkiems tikslams. Nepaisant esamų žinių apie programinės įrangos klaidų pavojus, pranešamų pažeidžiamumų skaičius vis auga. Tai paaiškina, kodėl programinės įrangos sauga tapo svarbia tyrimų dalimi ir kodėl reikia moksliškai tirti bei tobulinti metodus, skirtus programų pažeidžiamumų identifikavimui.

Šio darbo tikslas – pasiūlyti programų pažeidžiamumų aptikimo metodą naudojant nuotolinį procedūrų iškvietimą.

Darbo struktūra:

- Darbo teorinės analizės dalyje apžvelgiama programinės įrangos pažeidžiamumo sąvoka, pažeidžiamumų skirstymas ir pateikiami dažniausi pažeidžiamumai. Taip pat pateikiami egzistuojantys pažeidžiamumų aptikimo metodai, jų taikymas ir analizuojami egzistuojantys sprendimai. Galiausiai apžvelgiami esami tarp-procesinės komunikacijos karkasai. Skyriaus pabaigoje pateikiamos apibendrintos analizuotos teorinės medžiagos išvados;
- Antrojoje darbo dalyje aprašomas siūlomas pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą. Detalizuojami metode naudojami algoritmai ir pateikiamas veikimo principas;
- Trečiojoje darbo dalyje detalizuojama įgyvendinto sprendimo realizacija, naudotos technologijos;
- Ketvirtoje darbo dalyje pristatomi eksperimentinio tyrimo rezultatai. Aprašomi eksperimentinio tyrimo vertinimo kriterijai ir eiga. Pateikiama naudoto testavimo karkaso struktūra ir pažeidžiamumai. Siūlomas metodas aptiko 11 iš 12 pažeidžiamumų. Taip pat pristatomi lyginamų sprendimų rezultatai. Skyriaus pabaigoje pateikiami apibendrinti tyrimų rezultatai ir pagal juos suformuluotos išvados;
- Paskutinėje dalyje pateikiamos bendros magistrinio darbo išvados.

Jokubauskas, Lukas. Software Vulnerabilities Detection Method Using Remote Procedure Calls. Master's Final Degree Project. Supervisor prof. Jevgenijus Toldinas; Faculty of Informatics, Kaunas University of Technology.

Study field and area (study field group): Informatics Engineering (Computing).

Keywords: Vulnerability detection, Dynamic analysis, RPC, gRPC, Fuzzy logic.

Kaunas, 2022. 80 p.

Summary

The growing need for software to automate part of a person's daily activities makes applications increasingly complex. Computer software often comprises multiple components, such as a frontend application and a backend database, which obviously need to exchange information. Many modern desktop applications also follow the design of web software and have a separate frontend and backend processes. Inter-process communication mechanisms or third-party frameworks provided by the operating system are used for communication between processes. Improperly implemented inter-process communication can lead to code vulnerabilities that can be exploited for malicious purposes. Despite current knowledge about the dangers of software bugs, the number of reported vulnerabilities is growing. This explains why software security has become an important part of research and why methods to identify software vulnerabilities need to be researched and improved.

This work aims to propose a method for detecting vulnerabilities in applications using a remote invocation of procedures.

Work structure:

- The theoretical analysis section of the work reviews the concept of software vulnerabilities and the distribution of vulnerabilities and presents the most common vulnerabilities. Existing vulnerability detection methods, their application, and analysis of existing solutions are presented. Finally, the frameworks for inter-process communication are reviewed. At the end of the chapter, the conclusions of the analyzed theoretical material are summarized.
- The second part of the work describes the proposed vulnerability detection method using a remote procedure call. The algorithms used in the method are detailed and the principle of operation is presented.
- The third part of the work details the implementation of the implemented solution used technologies.
- The fourth part of the work presents the results of experimental research. The evaluation criteria and the course of the experimental study are described. The structure and vulnerabilities of the used test framework are presented. The proposed method detected 11 of the 12 vulnerabilities. The results of the compared solutions are also presented. At the end of the chapter, the results of the research are summarized, and the conclusions formulated according to them.
- The last part presents the general conclusions of the master's thesis.

Turinys

Lentelių sąrašas

Paveikslų sąrašas

Santrumpų ir terminų sąrašas

Santrumpos:

PĮ – programinė įranga;
IP – interneto protokolas;
UDP – greito duomenų perdavimo protokolas (angl. *User Datagram Protocol*);
RPC – nuotolinės procedūros kreipinys (angl. *Remote Procedure Call*);
WCF – Microsoft kompanijos sukurtas duomenų apsikeitimo karkasas (angl. *Windows Communication Foundation*);
gRPC – Google kompanijos sukurtas nuotolinių procedūrų kreipinių karkasas (angl. *Google Remote Procedure Call*);
HTTP – Hipertekstų persiuntimo protokolas saityno duomenims persiųsti. (angl. *HyperText Transfer Protocol*);
TCP – garantuoto duomenų pristatymo protokolas (angl. *Transmission Control Protocol*);
WSDL – saityno paslaugų aprašymo kalba (angl. *Web Services Description Language*);
MSMQ – Microsoft kompanijos sukurta pranešimų eilės realizacija (angl. *Microsoft Message Queuing*);
KCS – keitimų ciklų skaičius;
OS – operacinė sistema;
JSON – atviro standarto formatas, perduodantis duomenų objektus, sudarytus iš atributo ir reikšmės porų, lengvai skaitomame tekste (angl. *JavaScript Object Notation*);
SSD – puslaidininkinis diskas (angl. *Solid State Drive*);

Terminai:

Emuliavimas – duomenų apdorojimo sistemos naudojimas kitai duomenų apdorojimo sistemai imituoti tokiu būdu, kad būtų priimami tie patys duomenys;

Kompiliatorius – programa, verčianti skaitmeninės skaičiavimo mašinos ar kompiliavimo kompiuterinę programą iš programavimo kalbos į mašinos komandų sistemą ar kokią nors kitokią mašininio lygio kalbą;

Profiliavimas – dinaminės programų analizės forma, kuri matuoja programinės įrangos sudėtingumą, tam tikrų instrukcijų naudojimą arba dažnumą, metodų veikimo laiką;

Smėlio dėžė – saugumo mechanizmas, kuris atskiria veikiančią programinę įrangą vieną nuo kitos;

Stekas – duomenų struktūra veikianti LIFO (angl. *last in first out*) principu;

Heap atminties regionas – atminties dalis, kuri gali būti išskirta dinamiškai;

Įvadas

Pažeidžiamumai egzistuoja visur. Kiekvienas įrenginys, kurį naudojame savo kasdieniniame gyvenime gali turėti pažeidžiamumus. Didėjantis programinės įrangos poreikis siekiant automatizuoti dalį žmogaus gyvenimo kasdieninių veiksmų daro programas vis labiau kompleksiškas. Programinė įranga dažnai susideda iš daugybės komponentų, pavyzdžiui, išorinės sąsajos (angl. *frontend*) ir vidinės sąsajos (angl. *backend*). Šie komponentai turi vykdyti duomenų mainus. Daugybė modernios asmeninių kompiuterių PĮ naudoja tokį patį saitynų architektūrinį sprendimą, kai veikia atskiras išorinės sąsajos ir vidinės sąsajos procesas. Komunikacijai tarp procesų yra naudojami operacinės sistemos suteikiami tarp-procesinės komunikacijos mechanizmai arba tam tikri karkasai. Netinkamai įgyvendinta tarp-procesinė komunikacija gali sukelti programinio kodo pažeidžiamumus, kuriuos galima išnaudoti piktavališkiems tikslams. Pažeidžiamumas yra programos yda arba trūkumas, kuris gali būti išnaudojamas kenkėjiškų aktorių siekiant įgyti prieigą prie sistemų, duomenų arba sutrikdyti veiklą. Nepaisant esamų žinių apie programinės įrangos klaidų pavojus, pranešamų pažeidžiamumų skaičius vis auga. Tai paaiškina, kodėl programinės įrangos sauga tapo svarbia tyrimų dalimi ir kodėl reikia moksliškai tirti bei tobulinti metodus, skirtus programų pažeidžiamumų identifikavimui.

Darbo tikslas: pasiūlyti programų pažeidžiamumų aptikimo metodą naudojant nuotolinį procedūrų iškvietimą.

Darbo uždaviniai:

- Išanalizuoti programų pažeidžiamumo aptikimo metodus ir priemones;
- Pasiūlyti metodą, kuris aptiktų programų pažeidžiamumus naudojant nuotolinį procedūrų iškvietimą;
- Įvertinti pasiūlyto metodo veiksmingumą ir pateikti išvadas.

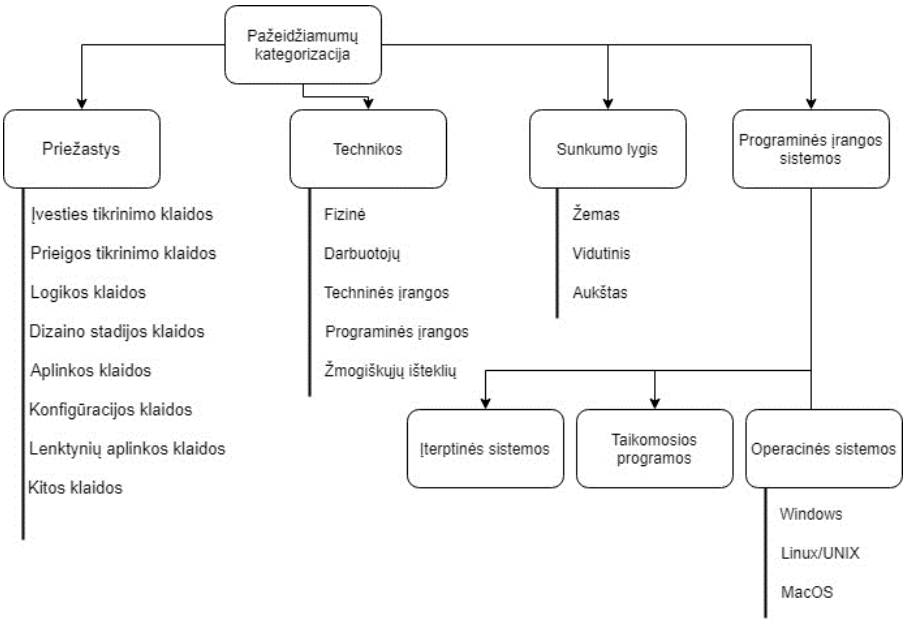
Dokumento struktūra. Darbą sudaro įvadas, su sprendžiamais uždaviniais susijusios informacijos analizė, metodinė dalis, realizacija ir eksperimentų rezultatai, išvados. Pirmajame darbo skyriuje apžvelgiama programinės įrangos pažeidžiamumo sąvoka, pažeidžiamumų skirstymas ir pateikiami dažniausi pažeidžiamumai. Taip pat pateikiami egzistuojantys pažeidžiamumų aptikimo metodai, jų taikymas ir analizuojami egzistuojantys sprendimai. Galiausiai apžvelgiami esami tarp-procesinės komunikacijos karkasai. Antrame darbo skyriuje aprašomas siūlomas pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą. Trečiajame skyriuje aprašoma siūlomo sprendimo realizacija. Ketvirtame skyriuje aprašomi eksperimentiniai tyrimai ir jų rezultatai. Kiekvieno skyriaus pabaigoje pateikiamos atitinkamos išvados, o apibendrintos magistro baigiamojo darbo išvados pateikiamos darbo pabaigoje.

1. Programų pažeidžiamumų aptikimo metodų analizė

Šiame skyriuje analizuojami kompiuterių programose dažniausiai aptinkami pažeidžiamumai, metodai, skirti programų pažeidžiamumų aptikimui, panašūs programų pažeidžiamumų aptikimo įrankiai, baltosios dėžės (angl. *white-box*), pilkosios dėžės (angl. *grey-box*) ir juodosios dėžės (angl. *black-box*) testavimo metodų pritaikymas programų pažeidžiamumų aptikime ir programų tarpprocesinės komunikacijos vykdymas naudojant nuotolinius procedūrų iškvietimo karkasus.

1.1. Programų pažeidžiamumai

Programinė įranga yra kompiuterio vykdomų instrukcijų seka, skirta tam tikriems veiksmams atlikti. Dažniausiai tokia įranga kuriama naudojant programavimo kalbas, vėliau kompiliuojant ar interpretuojant parašytą kodą. Taikomųjų programų kodo rašymo metu, neatsižvelgiant į naudojamą programavimo kalbą, yra įtraukiama klaidų, kurios gali sukelti pažeidžiamumus. Pažeidžiamumai taip pat yra įtraukiami su naujomis programinės įrangos funkcijomis [1]. Pati pažeidžiamumo sąvoka yra apibūdinama kaip trijų sudedamųjų dalių sąjunga [1]. Pirmoji dalis yra sistemos jautrumas išoriniams veiksniams. Antroji sudedamoji dalis yra kenkėjiško aktoriaus prieiga prie pažeidžiamumo. Trečioji dalis yra kenkėjiško aktoriaus galimybė išnaudoti esamą pažeidžiamumą. Kibernetiniai nusikaltimai yra įgyvendinami pasinaudojant programų pažeidžiamumais, todėl yra svarbu juos surasti, atpažinti ir ištaisyti. Pirmiausia reikia tinkamai suskirstyti programinės įrangos pažeidžiamumus tam, kad būtų galima rasti tikslią priežastį ir tinkamą sprendimo būdą. Tinkamas pažeidžiamumų skirstymas pagal kategorijas gali būti naudojamas programinės įrangos palaikymo laikotarpyje, kurio metu yra taisomos klaidos pagal pažeidžiamumo tipą ir įtaką. pav. 1 yra pateiktos aktualios pažeidžiamumų charakteristikos: priežastis, sunkumo lygis ir programinės įrangos sistemos tipas. Į technikos charakteristiką nėra atsižvelgiama, kadangi yra analizuojami tik programinės įrangos technikos pažeidžiamumai.



pav. 1 Pagrindinės techninės pažeidžiamumų skirstymo charakteristikos

Pasinaudojant šiomis charakteristikomis, pagal 2021 metais sudarytą dažniausių programinės įrangos pažeidžiamumų sąrašą [2], buvo išskirti šie dažniausi pažeidžiamumai: kreipimasis į nulinę rodyklę (angl. *null-pointer dereference)* ir talpyklos perpildymas (angl. *buffer overflow*).

Kreipimasis į nulinę rodyklę. Tai pažeidžiamumas, kuris apima visas programavimo sferas nuo asmeninių kompiuterių PĮ iki interneto paslaugų. Šis pažeidžiamumas įvyksta kai yra kreipiamasi į atminties rodyklę arba objektą, kuris yra neinicijuotas arba nustatytas kaip nulinė reikšmė. Priklausomai nuo programavimo kalbos, kreipimasis į nulinę rodyklę gali sukelti neapibrėžtą elgseną arba vykdymo klaidas. Šis pažeidžiamumas įgalina kenkėjiškas veiklas, pavyzdžiui, paslaugos trikdymo ataką [3]. Dažniausiai šis pažeidžiamumas yra randamas objektinėse ir nesaugios atminties (angl. *memory-unsafe*) programavimo kalbose, pavyzdžiui, C, C++ ir Java [4].

Talpyklos perpildymo pažeidžiamumas. Tai klaida, kuri įvyksta kai programa kopijuoja duomenis į talpyklą neįvykdžius duomenų dydžio patikrinimo [5]. Pažeidžiamumas gali būti išnaudojamas kai perkeliama atminties sritis yra didesnė nei galutinis atminties regionas. Jei galutinė talpykla yra heap atminties srityje, šis pažeidžiamumas vadinamas heap talpyklos perpildymo pažeidžiamumu. Jei galutinė talpykla yra steko atminties srityje, tai vadinama steko talpyklos perpildymo pažeidžiamumu. Pagrindinė pažeidžiamumo priežastis yra programavimo klaidos, t. y. netinkamas įvesties tikrinimas ir logikos klaidos [1]. Šis pažeidžiamumas aptinkamas tiek įterptinėse sistemose, tiek taikomosiose programose ir operacinėse sistemose [1]. Kenkėjiški aktoriai dažnai pasinaudoja šiuo pažeidžiamumu, kad įvykdytų kenkėjiško kodo vykdymą ar sutrikdytų paslaugos teikimą. Duomenų talpyklos perpildymui, kuriuo pasinaudojant negalima įvykdyti kenkėjiško kodo ar paslaugos trikdymo, yra priskiriamas vidutinis sunkumo lygis.

Apibendrinus kompiuterių programose dažniausiai aptinkamas klaidas buvo išskirti du pažeidžiamumai: kreipimasis į nulinę rodyklę (angl. *null-pointer dereference*) ir talpyklos perpildymas (angl. *buffer overflow*). Šie pažeidžiamumai yra glaudžiai susiję, kadangi pagrindinė pažeidžiamumų kilmės priežastis yra programos atminties vientisumo ir struktūros sutrikdymas.

1.2. Programų pažeidžiamumų aptikimo metodai

Šioje analizės dalyje bus apžvelgti taikomųjų programų pažeidžiamumo aptikimo metodai. Šios klaidos gali sukompromituoti programų veikimą arba operacinės sistemos saugumą. Yra išskiriamos dvi pagrindinės pažeidžiamumų aptikimo metodų kategorijos: statinis ir dinaminis pažeidžiamumų aptikimas [7]. Statinių metodų atveju nėra būtinybės vykdyti programinį kodą siekiant aptikti pažeidžiamumus. Dinaminių metodų atveju programinis kodas yra vykdomas kontroliuojamoje aplinkoje kur yra stebimas programos veikimas. Programinės įrangos pažeidžiamumai yra aptinkami atliekant programos veikimo ir elgsenos stebėsenos analizę.

1.2.1. Statiniai programų pažeidžiamumų aptikimo metodai

Statiniai programų pažeidžiamumų aptikimo metodai yra taikomi kodui, nevykdant pačios programos. Šios metodų kategorijos pagrindinis tikslas yra gauti specifinę informaciją iš programos kodo ar kitų artefaktų, kurie yra pateikiami kaip įvestis. Statiniai analizės metodai sugeba rasti klaidas programų kūrimo stadijoje, taip pat šie metodai gali būti efektyvūs ir pakankamai gerai automatizuoti [7]. Galiausiai, šie metodai gali padengti visas programinės įrangos vykdymo šakas [7]. Tačiau šie metodai susiduria su problemomis, kurios pastebimos tik programos vykdymo metu, t. y. nenumatytas veikimas, atminties rodyklių aritmetikos problemos. Egzistuoja keletas specifinių statinės pažeidžiamumų aptikimo kategorijos metodų, kurie bus aptarti šioje analizės dalyje.

Šablonais paremtas pažeidžiamumų aptikimas (angl. *pattern matching*). Naudojant šį metodą yra atliekama šabloninio teksto fragmento paieška programos kode ir pateikiamas atitikčių skaičius [6].

Kaip pavyzdį galima pateikti C++ kalbos kodą. Ieškomas šablonas gali būti bet koks kreipinys į pažeidžiamą funkciją, pavyzdžiui, *strcpy*. Šablonais paremtas aptikimas gali būti įgyvendintas naudojant paprastą teksto paiešką. Tačiau tai sukeltų daug netikrų pažeidžiamumų atvejų, kadangi surasti atitikmenys nėra toliau analizuojami. Rezultatai taip pat gali būti netikslūs dėl šablonų paieškos interpretacijų, kai rodomi tik tikslūs atitikmenys. Patobulinta šio metodo versija vadinama leksinė analize. Leksinėje analizėje atsiranda papildomas žingsnis, kuris yra atliekamas prieš šablonais paremtą paiešką. Programinis kodas yra paverčiamas į atskirų dalių eilę, kurios dalys vėliau yra tapatinamos su šablonais. Netikslių rezultatų skaičius išlieka aukštas, kadangi šis metodas, kaip ir prieš tai aptarta šablonais paremta statinė analizė, neįvertina kodo sintaksės ar gramatikos. Norint pagerinti leksinės analizės metodą buvo sukurtas nagrinėjimu paremtas pažeidžiamumų aptikimo metodas. Šiame metode atliekamas kodo dalių nagrinėjimas, kuriuo siekiama nustatyti tinkamą kodo dalių eiliškumą ir ar tas eiliškumas atitinką gramatiką. Visa tai yra sudedama į abstraktų sintaksės medį, kuris atitinka aukšto lygio programos struktūrą. Struktūra yra analizuojama norint rasti galimus pažeidžiamumus.

Duomenų srautu paremtas pažeidžiamumų aptikimas (angl. *data flow analysis*). Šio metodo tikslas yra nustatyti galimas kintamųjų ar išraiškų (angl. *expression*) reikšmes programos vykdymo metu [6]. Galimos kintamųjų ar išraiškų reikšmės gali būti apskaičiuotos tiek į ateitį, tiek į praeitį. Dažniausiai gauti rezultatai yra sudedami į rinkinį, pavyzdžiui, nustatant aritmetinių išraiškų rinkinio galimus rezultatus, kurie jau buvo anksčiau apskaičiuoti tam tikroje programos kodo vietoje, arba kintamųjų rinkinys, kuris bus reikalingas tolimesniam programos vykdymui, arba kintamųjų rinkinys, kuris turi pastovias reikšmes. Kadangi analizė tinkamai neįvertina galimų programos vykdymo šakų, rezultatai būna netikslūs. Informacija gauta panaudojant duomenų srauto analizę gali būti naudojama užtikrinant atitinkamas saugos savybes arba padėti kompiliatoriui generuoti efektyvų išeities kodą. Šis pažeidžiamumų aptikimo metodas yra specialiai skirtas talpyklos perpildymo pažeidžiamumų aptikimui [6]. Tai pat egzistuoja specializuotas programinio kodo pažeidimų analizės metodas (angl. *taint*) paremtas duomenų srauto analize. Šiuo metodu tikrinama kur programiniame kode gali atsirasti įvestys iš nepatikimų šaltinių, pavyzdžiui, vartotojo įvestis gali sukelti potencialią problemą, todėl tai pažymima kaip probleminė vieta. Šis duomenų srautas yra stebimas, kadangi jis gali pasiekti kritinius programos metodus.

Modelių tikrinimu paremtas pažeidžiamumų aptikimas (angl. *model checking*). Modelių tikrinimu paremtu pažeidžiamumų aptikimo metodu galima patikrinti ar programinės įrangos modelis atitinka numatytus kriterijus. Šis metodas apskaičiuoja galimas vykdymo meto programos būsenas nevykdant programos kodo. Surinktos programos būsenos yra patikrinamos siekiant nustatyti savybių tinkamumą. Tuo pasinaudojant galima aptikti pažeidžiamumus programiniame kode. Programa būna visiškai išanalizuota kai programos elgsena tenkina visas numatytas savybes. Jei egzistuoja bent viena programos vykdymo šaka, kuri netenkina numatytų savybių, programa yra netinkama. Modelių tikrinimas yra sudėtingas metodas, kadangi pats modelio detalizavimo procesas yra sudėtingas. Tačiau kai modelis yra parengtas programinės įrangos savybių testavimas tampa lengvesnis [6]. Derinant šį metodą su apribojimų analize galima atlikti talpyklos perpildymo pažeidžiamumų paiešką kode [6].

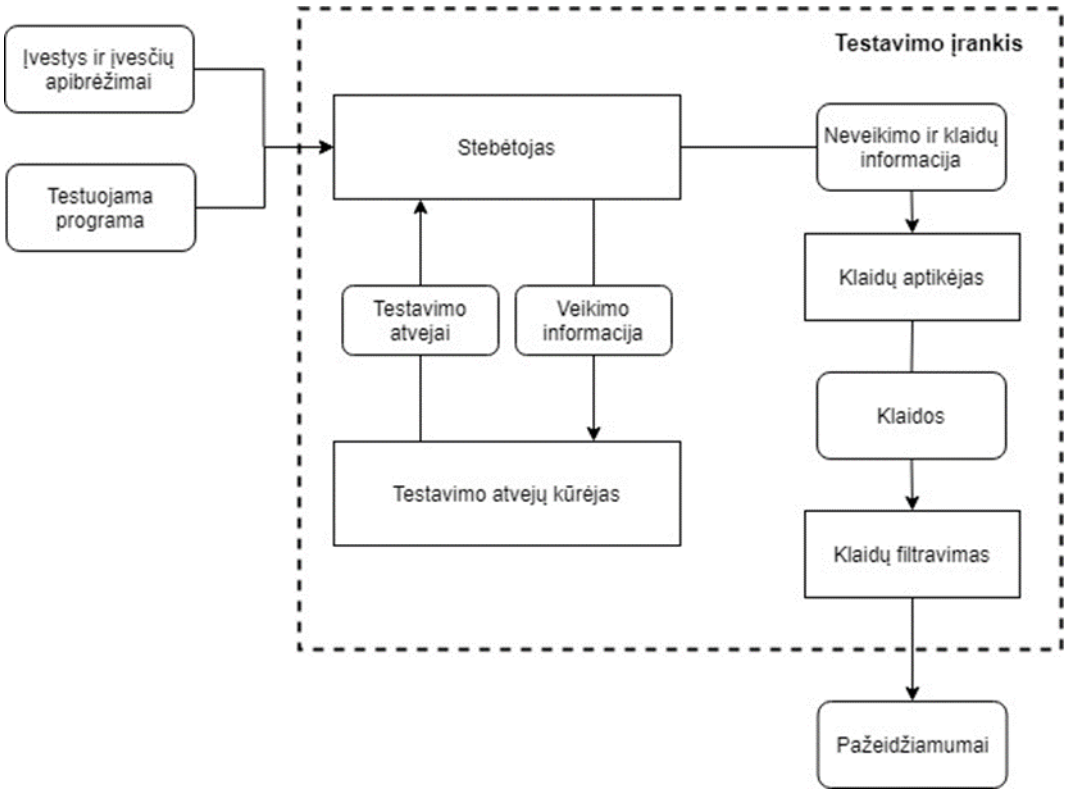
1.2.2. Dinaminiai programų pažeidžiamumų aptikimo metodai

Dinaminiai programų pažeidžiamumų aptikimo metodai nereikalauja programos kodo. Šie metodai randa galimus pažeidžiamumus panaudojant programos veikimo informaciją, kuri yra surenkama

vykdant testuojamą taikomąją programą. Lyginant su statinės analizės metodais, dinaminės analizės metodai gali būti naudojami plačiau, ypač kai neprieinamas testuojamos programos kodas. Kadangi šie metodai remiasi programos veikimo informacija, yra pasiekiamas didesnis tikslumas. Tačiau egzistuoja keletas trūkumų [7], pavyzdžiui, dinaminiai metodai yra ne tokie efektyvūs, nes reikalaujama vykdyti programą. Taip pat, prieš leidžiant programą yra reikalinga jos konfigūracija, todėl yra sunku automatizuoti pažeidžiamumų aptikimo procesą. Be to, dažniausiai yra naudojamos profiliavimo technikos, kurios padeda gauti detalesnę veikimo informaciją, tai prailgina testavimo proceso trukmę. Taip pat, šių metodų tikslumas priklauso nuo testavimo įvesčių kokybiškumo. Egzistuoja keletas specifinių dinaminės pažeidžiamumų aptikimo kategorijos metodų, kurie bus aptarti šioje analizės dalyje.

Dinaminių pažeidimų analize paremtas pažeidžiamumų aptikimas (angl. *dynamic taint analysis)*. Dinaminių pažeidimų analizė pastaraisiais metais yra plačiai naudojama daugelyje informacijos saugos sferų, pavyzdžiui, kenksmingos programinės įrangos analizavime, tinklo atakų aptikime ir apsaugoje, programų pažeidžiamumų aptikime, protokolų testavime. Dinaminė pažeidimų analizė nedaug skiriasi nuo statinės pažeidimų analizės. Veikimo principas išlieka nepakitęs – nustatomas pažeidimų šaltinis, pavyzdžiui, vartotojo įvestis, tada tikrinamas įvesties paplitimas programoje tikrinant ar įvestis yra naudojama pavojingose funkcijose [8]. Pasinaudojant šiuo metodu galima aptikti galimas įvesties tikrinimo problemas, kurios yra priskiriamos pažeidžiamumams. Šis metodas yra skirstomas į prijungtą ir neprijungtą analizę. Prijungta dinaminių pažeidimų analizė yra aktyvi vykdant programą ir baigiama nutraukus programos vykdymą. Neprijungta dinaminių pažeidimų analizė pirmiausia įrašo programos veikimo informacija į failus ir tik tada analizuoja taikomąją programą atkartojant jau įrašytą veikimo informaciją.

Miglotąja logika (angl. *fuzzing*) paremtas pažeidžiamumų aptikimas. Miglotąja logika paremtas pažeidžiamumų aptikimas naudoja netinkamą ar atsitiktinę įvestį ir pateikia ją programai siekiant išprovokuoti klaidingą programos elgseną ir pagal tai identifikuoti pažeidžiamumą. Metodo procesas yra pavaizduotas pav. 2. Pagrindinė šio metodo dedamoji yra duomenų arba testavimo atvejų kūrimas, kai intensyviu testavimu siekiama sutrikdyti programos veikimą. Taip pat yra svarbu pasirinkti tinkamus įrankius programos veikimo stebėjimui. Testavimui skirti duomenų rinkiniai gali būti kuriami dvejais būdais: atliekant mažus atsitiktinius keitimus duomenų rinkinyje arba kuriant duomenų rinkinius pagal tam tikrą duomenų formatą [9]. Mažais atsitiktiniais keitimais paremto pažeidžiamumų aptikimo metodo duomenų rinkiniai dažnai būna nepriimtini programoms, kadangi pakeisti duomenų rinkiniai per daug skiriasi nuo tikėtinos teisingos įvesties. Tačiau šis trūkumas yra įveikiamas naudojant duomenų rinkinius sukurtus pagal formatą.



pav. 2 Miglotosios logikos (angl. *fuzzing*) pažeidžiamumų aptikimo metodo veikimas

Šis metodas gali būti skirstomas į tris kategorijas pagal testuojamos programos supratimo lygį, tai yra juodosios, baltosios ir pilkosios dėžės kategorijos. Miglotąja logika paremtas juodosios dėžės pažeidžiamumų aptikimas remiasi atsitiktinių duomenų kūrimu. Šio tipo aptikimo metodai yra nebenaudojami dėl ilgos pažeidžiamumų atradimo laiko trukmės ir pačios programos vykdymo šakų padengimo problemos [8]. Baltosios dėžės kategorija, kitaip nei juodosios dėžės, gali naudoti programos kodą, dizaino savybes ir detalią programos vykdymo informaciją, kad pagerintų vykdymo šakų padengimą, tai pagreitina pažeidžiamumų aptikimo procesą [9]. Galiausiai lieka pilkosios dėžės kategorija, kurioje yra derinamos geriausios juodosios ir baltosios dėžių testavimo savybės. Pilkosios dėžės kategorija naudoja programos vykdymo šakų atpažinimą, kad sugebėtų atrasti naujas neištestuotas funkcijas programoje. Tačiau pilkosios dėžės kategorijos įrankiai nėra pritaikyti atlikti gilią programinės įrangos kodo analizę [8].

Simboliniu vykdymu (angl. *symbolic execution*) paremtas pažeidžiamumų aptikimas. Tai metodas tikrinantis ar tam tikros programinės įrangos savybės gali būti pažeistos panaudojant kitą programinę įrangą [8]. Pagrindinė idėja yra, kad vietoje bandymų pateikti nenumatytas programos įvestis yra bandoma įterpti programinius komponentus (angl. *program symbols*) siekiant kontroliuoti programos veikimą. Šie komponentai yra įterpiami siekiant atrasti naujus vykdymo kelius, kurie yra sunkiai surandami naudojant atsitiktinius įvesčių rinkinius. Šis metodas turi du režimus: prisijungęs ir neprisijungęs režimas. Neprisijungęs režimas suteikia galimybę sistemai tikrinti vieną galimą vykdymo šaką vienu metu, taip pat leidžia sistemai pasirinkti naujas vykdymo šakas iteraciniu būdu. Tačiau lyginant su prisijungusiu režimu, šis režimas yra neefektyvus, kadangi prisijungusiu režimu testavimo procesas yra atliekamas paraleliai kiekvienai vykdymo šakai. Ši technika užtikrina, kad kiekviena šaka bus įvykdoma tik kartą, tačiau, ši technika vykdymo metu naudoja daug operatyviosios atminties [8]. Didžiausia šio metodo problema yra vykdymo šakų sprogimas (angl. *path explosion*) [9]. Kai šis metodas yra naudojamas testuoti sudėtingas programas, kiekviena nauja

vykdymo šaka sukuria kelias naujas vykdymo šakas. Tai gali eksponentiškai padidinti vykdymo šakų skaičių, kurias reikia patikrinti.

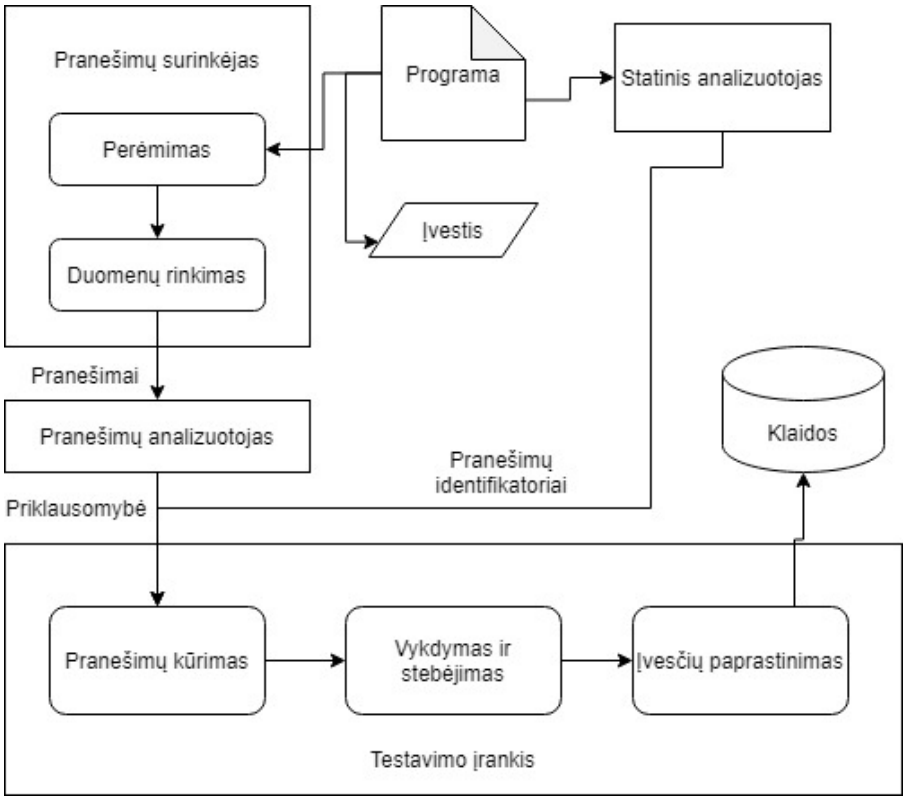
Apibendrinus statinius ir dinaminius programų pažeidžiamumų aptikimo metodus išsiaiškinta, kad tiek statiniai, tiek dinaminiai metodai turi savo privalumų ir trūkumų. Dinaminiai metodai yra priimtinesni dėl galimybės atrasti užslėptas klaidas ir pažeidžiamumus, kurie nerandami statinės kodo analizės metu. Taip pat dinaminis programų testavimas suteikia didesnį tikslumą klaidų radimo atveju. Kalbant apie konkretų dinaminį testavimo metodą, priimtiniausias yra miglotąja logika paremtas metodas, kadangi yra universaliausias, gali derinti tiek statinės, tiek dinaminės analizės privalumus.

1.3. Programų pažeidžiamumų aptikimo įrankiai

Šioje analizės dalyje bus apžvelgti panašūs taikomųjų programų pažeidžiamumų aptikimo įrankiai, kurie geba pasinaudoti programų tarp-procesinės komunikacijos kanalais.

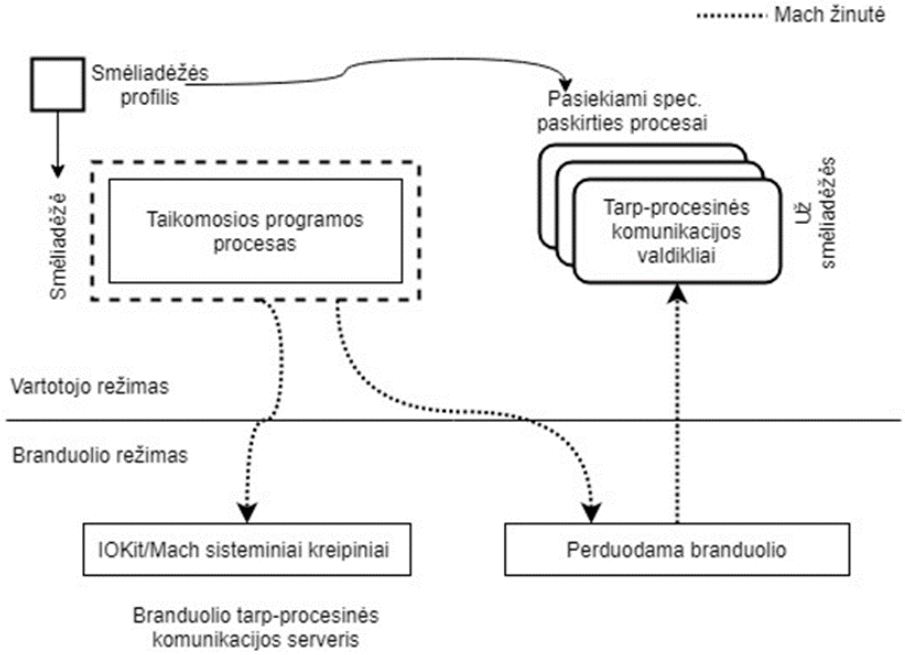
1.3.1. Programų pažeidžiamumų aptikimo įrankis *MachFuzzer*

Yang ir kt. [10] pasiūlė pažeidžiamumų aptikimo įrankį pasinaudojant miglotosios logikos (angl. *fuzzing*) pažeidžiamumų aptikimo metodu. Šis įrankis geba rasti tarp-procesinės komunikacijos klaidas programinėje įrangoje be programų kodo naudojant statinės ir dinaminės analizės derinį. Statinė analizė yra naudojama atpažinti tarp-procesinės komunikacijos žinučių formatus ir kurti naujas žinutes pasinaudojant atpažintais formatais. Dinaminė analizė yra naudojama numatyti galimus ryšius tarp tarp-procesinės komunikacijos žinučių ir suformuoti išsamią loginę schemą su tikimybių matrica. Detali *MachFuzzer* įrankio schema yra pateikta pav. 3.



pav. 3 *MachFuzzer* testavimo įrankio architektūrinė schema

Šis įrankio prototipas buvo specialiai sukurtas *MacOS* operacinės sistemos *WindowsServer* komponentui, kuris atsakingas už grafikos operacijas. Pagal įgyvendintą tyrimų studiją, šis įrankis sugebėjo atrasti 12 unikalių klaidų, susidedančių iš nulinės rodyklės panaudojimo (angl. *null pointer dereference*), panaudos po atlaisvinimo (angl. *use after free*), sveikojo skaičiaus perpildymo (angl. *integer overflow*), netinkamo atminties pasiekimo (angl. *bad memory access*) pažeidžiamumų. Trys iš šių pažeidžiamumų buvo išnaudoti pasiekiant privilegijų eskalavimą ir smėlio dėžės aplinkos apėjimą.



pav. 4 *MacOS* operacinės sistemos smėlio dėžės apėjimo mechanizmas

Dažniausiai smėlio dėžės (angl. *sandbox*) apėjimas *MacOS* operacinėje sistemoje įgyvendinamas pasinaudojant sukompromituotu procesu apsaugotu smėlio dėže, kuris komunikuoja su neapsaugotu išoriniu procesu. Šis procesas pavaizduotas pav. 4. Didžiausias šio įrankio trūkumas – nelankstumas operacinės sistemos ir taikomųjų programų atžvilgiu, kadangi šis prototipas buvo įgyvendintas konkrečiai *MacOS* operacinės sistemos daliai.

1.3.2. Programų pažeidžiamumų aptikimo įrankis *Peach Fuzzer*

Peach Fuzzer [11] yra gerai žinomas bendrinės paskirties miglotosios logikos (angl. *fuzzing*) pažeidžiamumų aptikimo metodo įrankis. Šio įrankio tiksliniai objektai yra įrenginių tvarkyklės, įvairių failų vartotojai, tinklo protokolai, tarp-procesinės komunikacijos protokolai, įterptiniai įrenginiai, operacinės sistemos ir kt. Šis pažeidžiamumų aptikimo įrankis susideda iš trijų dalių: įvesties rinkinių formatų apibrėžimai vadinami *Peach Pits*, kurie yra prieinami kaip individualūs objektai ar grupės susijusių formatų apibrėžimų, kurie vadinami *Pit Packs*. Kita dalis yra testų rinkiniai, kurie gali priversti duomenų rinkinių kūrimo esybę (angl. *mutator*) generuoti daugiau duomenų rinkinių ir atlikti testų. Trečioji dalis yra *minset* įrankis, kuris geba sumažinti vieno testavimo atvejo įvesčių failų skaičių. *Peach Fuzzer* įrankis išlaiko svarbią rolę pažeidžiamumų testavime, kadangi šis programinės įrangos rinkinys turi jau sukurtus įvesties rinkinių formatų apibrėžimus ir daugybę testavimo parinkčių. Didžiausias šio įrankio trūkumas – tai juodosios dėžės tipo įrankis, kuris nenaudoja programų veikimo informacijos siekiant atrasti neišanalizuotus veikimo kelius.

1.3.3. Programų pažeidžiamumų aptikimo įrankis *AFL*

AFL [13] yra gerai žinomas kodo padengimu paremtas miglotosios logikos (angl. *fuzzing*) pažeidžiamumų aptikimo metodo įrankis. Įrankis surenka informaciją apie vykdymo šakas remdamasis kodo profiliavimu. Atvirojo kodo programų profiliavimas yra sukuriamas kompiliavimo metu, o vykdomiesiems failams profiliavimas yra sukuriamas vykdymo metu per modifikuotą *QEMU* virtualią mašiną [14]. Testavimo atvejai, kurie gali aptikti naujas vykdymo šakas, turi didesnę tikimybę būti parinktiems kitoje įvesčių kūrimo iteracijoje. Rezultatai įrodė, kad *AFL* yra efektyvus įrankis randant pažeidžiamumus realioje programinėje įrangoje skirtoje suspausti, nuskaityti failus ir kt. Šis įrankis palaiko C, C++, Objective C programavimo kalbas ir veikia Linux paremtose operacinėse sistemose. Nors šis įrankis yra efektyvus, tačiau egzistuoja keletas trūkumų, pavyzdžiui, *AFL* pateikia ribotą kodo padengimo informaciją kai įvesties duomenys yra suspausti ar užšifruoti. Taip pat šis įrankis reikalauja daugiau laiko testuojant 64 bitų vykdomuosius failus ir tiesiogiai nepalaiko tinklo paslaugų testavimo. Šią problemą bandė išspęsti Chen ir kt. [12] savo darbe apie efektyvias miglotosios logikos pažeidžiamumų aptikimo strategijas skirtas komunikacijos protokolams. Šiame darbe buvo įgyvendintas karkasas skirtas būsenomis paremtam pažeidžiamumų aptikimui. Sistema susideda iš būsenomis paremto testavimo įrankio ir profiliuotos programos, kuri kartu su testavimo įrankiu atpažįsta, replikuoja ir persijungia tarp skirtingų protokolo būsenų išlaikant veikimo pastovumą testavimo metu. Visa tai padėjo pasiekti didesnį kodo padengimą ir rasti tris kartus daugiau veiklos sutrikimų *OpenSSL* bibliotekoje lyginant su įprastu *AFL* įrankiu. Nors protokolų būsenų problema buvo išspręsta, tačiau tai vis tiek neišsprendė *AFL* negebėjimo tiesiogiai testuoti tinklo paslaugų. Pham ir kt. [15] savo darbe pasiūlė *AFL* įrankio priedą, kuris nenaudodamas testavimo programų gali tiesiogiai prisijungti prie tinklo paslaugos, siųsti užklausas ir gauti atsakymus. Šis funkcionalumas buvo įgyvendintas panaudojant standartinę C kalbos tinklo prievadų (angl. *socket*) biblioteką. Šis *AFL* įrankio priedėlis taip pat naudoja būsenomis paremtą veikimą skirtą kontroliuoti testavimo procesą. Didžiausias trūkumas yra, kad šis įrankis gali testuoti protokolus, kurie remiasi atsakymo kodais, tačiau negali atlikti tinkamo testavimo su protokolais, kurie neturi atsakymo kodų.

1.3.4. Programų pažeidžiamumų aptikimo įrankis *Boofuzz*

Boofuzz [16] yra atvirojo kodo miglotosios logikos (angl. *fuzzing*) pažeidžiamumų aptikimo metodo įrankis skirtas testuoti tinklo protokolus. Šis įrankis naudoja blokais paremtą veikimo principą kurti individualias užklausas. Įrankis suteikia daug reikalingos informacijos kuriant naujus protokolų aprašymus. Prieš pradedant testavimą, vartotojas turi panaudoti šiuos duomenų formatus siekiant nurodyti, kurias protokolo vietas reikia keisti generuojant naujus testavimo atvejus. Šis įrankis geba klasifikuoti užfiksuotas klaidas, dirbti paraleliai ir atpažinti unikalias sekas, kurios sukelia klaidą. Šis įrankis buvo sukurtas panaudojant esamo *Sulley* [17] įrankio kodą, kuris jau yra nebepalaikomas. Lyginant su *Sulley*, *Boofuzz* įrankis ištaisė buvusias klaidas, įdiegė eterneto protokolo, IP sluoksnio, UDP protokolo transliavimo testavimo galimybę. Taip pat suteikė galimybę naudoti kitų protokolų tarpininkus. Didžiausias šio, kaip ir *Peach Fuzzer*, įrankio trūkumas – tai juodosios dėžės tipo įrankis, kuris nenaudoja programų veikimo informacijos siekiant atrasti neišanalizuotas veikimo šakas. Todėl šis įrankis gali neatrasti sudėtingesnių klaidų ir pažeidžiamumų.

Išanalizavus panašius esamus pažeidžiamumų aptikimo įrankius, kurie gali atlikti testavimą pasinaudojant programų tarp-procesinės komunikacijos kanalais, išsiaiškinta, kad tam buvo sukurtas tik vienas specializuotas prototipas (*MachFuzzer*), kuris buvo naudojamas *MacOS* operacinėje

sistemoje. Visi kiti aptarti įrankiai yra bendrosios paskirties tipo, kurie geba atlikti protokolais besinaudojančių programų testavimą. Atlikus tam tikrus derinimo procesus, šiuos įrankius galima pritaikyti naudoti tarp-procesinę komunikaciją, tačiau tai negarantuoja efektyvaus pažeidžiamumų aptikimo proceso.

1.4. Testavimo metodų taikymas programų pažeidžiamumų aptikimui

Dinaminis testavimas perteikia išorinę programinės įrangos perspektyvą, kuri parodo programavimo defektus. Skirtingai nuo statinio testavimo, kuriam yra reikalingas programinis kodas, dinaminis testavimas remiasi informacija gaunama iš vykdomos programos [14]. Pagal šią informaciją yra išskirtos trys kategorijos. Šioje analizės dalyje bus apžvelgtas baltosios dėžės (angl. *white-box*), pilkosios dėžės (angl. *grey-box*) ir juodosios dėžės (angl. *black-box*) testavimo metodų pritaikymas programų pažeidžiamumų aptikime.

1.4.1. Baltosios dėžės metodas

Baltosios dėžės metodu paremtas testavimas remiasi vidine programos struktūra, vykdymo ypatumais, kurie yra žinomi analizavimo įrankiui. Analizavimo įrankis parenka įvestis tam, kad sugebėtų atrasti programos kodo vykdymo šakas [18]. Kalbant apie baltosios dėžės testavimo metodo pritaikymą miglotosios logikos pažeidžiamumų aptikimo metodo įrankiuose, šio tipo įrankiai naudoja programų veikimo informaciją ir patį programos kodą [7]. Visą tai daroma siekiant padidinti programos kodo padengimą, kuris pagreitina testavimo procesą. Pradedant testavimo procesą su tam tikra įvestimi, įrankis pirmiausia surenka visų sąlyginių teiginių apribojimus (angl. *symbolic constrains*) ir vykdymo eigą su pateikta įvestimi. Po pirmojo vykdymo ciklo įrankis surenka visus apribojimus ir suformuoja naujos vykdymo šakos apribojimą. Tada iš vykdymo šakos vienas po kito yra pašalinami apribojimai taip sukuriant naujos vykdymo šakos apribojimą. Naujas testavimo atvejis leidžia programai eiti skirtinga vykdymo šaka. Teoriškai baltosios dėžės testavimo įrankiai gali sukurti testavimo atvejus, kurie padengs visas programos vykdymo šakas. Tačiau realybėje to nepavyksta pasiekti dėl daugybės problemų, pavyzdžiui, daugybės vykdymo šakų realioje programinėje įrangoje arba netikslumų sprendžiant naujus apribojimus. Todėl šio tipo įrankiai nepasiekia pilno testuojamos programos kodo padengimo [18]. Baltosios dėžės įrankiai randa labiau užslėptas klaidas skirtingai nuo juodosios dėžės metodika paremtų įrankių, tačiau baltosios dėžės įrankių kūrimas yra sudėtingesnis ir užima daugiau laiko [14]. Šio tipo testavimo įrankiai gali būti skirstomi į tris subkategorijas: padengimu paremti įrankiai (angl. *coverage guided fuzzers*), gramatika paremti įrankiai (angl. *grammar based fuzzers*) ir pažeidimais paremti įrankiai (angl. *taint based fuzzers*) [8]. Baltosios dėžės testavimo metodas gali būti naudojamas jei yra koncentruojamasi į rezultatų kokybę (įvairumą, pažeidžiamumus ir klaidų skaičių) ir norima pasiekti didesnį kodo padengimą, tačiau šis metodas yra nepraktiškas, kadangi sunaudoja daug piniginių ir laiko išteklių, taip pat susiduria su įvairiomis problemomis [14].

1.4.2. Juodosios dėžės metodas

Juodosios dėžės metodu paremtame testavime analizavimo įrankis neturi informacijos apie vidinę programos struktūrą ar jos veikimo principus [18]. Pagrindinis dalykas į kurį yra koncentruojamasi yra galimos įvestys ir tikėtinos kiekvienos įvesties išvestys. Kalbant apie juodosios dėžės testavimo pritaikymą miglotosios logikos pažeidžiamumų aptikimo metodo įrankiuose, šio tipo įrankiai nereikalauja jokios papildomos informacijos iš testuojamos programos. Vietoje to, įrankiai pagal taisykles atsitiktiniu būdu keičia teisingą įvestį tam, kad sukurtų klaidas sukeliančias įvestis.

Dažniausiai taip sukurtos įvestys būna atmetamos pirmomis programos vykdymo akimirkomis. Ši problema sumažina šansus rasti programų pažeidžiamumus. Norint surasti pažeidžiamumus reikia ištestuoti kuo daugiau programos funkcionalumo sukuriant tinkamą atsitiktinę įvestį. Šios problemos sprendimas buvo sukuriant techniką, kuri keičia mažą įvesties dalį. Keitimo būdai susideda iš bitų keitimo, baitų kopijavimo ar šalinimo ir kitų būdų [18]. Nauji testavimo įrankiai taip pat naudoja gramatiką ar įvestimi paremtą informaciją tam, kad sukurtų galimai teisingas įvestis [18]. Gramatika paremtas testavimas priima įvesties formato aprašymą, pagal kurį sukuriama daugybė skirtingų įvesčių. Juodosios dėžės įrankiai tipiškai randa labiau paviršutiniškas klaidas, skirtingai nuo baltosios dėžės metodika paremtų įrankių, tačiau šio tipo įrankiai yra nesudėtingi ir lengvai naudojami [14]. Šio tipo įrankiai daugiau nebėra kuriami dėl jų mažo kodo padengimo problemos ir testavimo procese sugaištamo laiko [8]. Juodosios dėžės testavimo metodas gali būti naudojamas jei yra koncentruojamasi į efektyvumą [14].

1.4.3. Pilkosios dėžės metodas

Pilkosios dėžės metodu paremtas testavimas yra juodosios ir baltosios dėžių metodų derinys, kuris leidžia efektyviai rasti programinės įrangos pažeidžiamumus su nepilnomis žiniomis apie programą. Pilkosios dėžės įrankiai negali būti tapatinami su juodosios dėžės įrankiais, kadangi yra naudojamas padengimu paremtas naujų vykdymo šakų nustatymas. Tuo pačiu metu pilkosios dėžės metodo įrankiai negali būti priskiriami baltosios dėžės metodo įrankiams, kadangi jie nėra sukurti naudoti detalią programų analizę ar spręsti vykdymo šakų apribojimus [8]. Dažniausiai naudojama technika pilkosios dėžės įrankiuose yra programos profiliavimas [18]. Tai reiškia, kad įrankis vykdymo metu gali gauti testuojamos programos kodo padengimą. Vėliau ši informacija yra naudojama pakeisti įvesčių kūrimo strategijas, kurios leistų padengti daugiau programos vykdymo šakų arba pagreitinti testavimo procesą. Tačiau tai negarantuoja, kad ši informacija sukurs geresnius testavimo atvejus skirtus naujoms vykdymo šakoms ar bus rastos specifinės klaidos. Jau anksčiau, 1.3.3 skyrelyje, aptartas pažeidžiamumų radimo įrankis *AFL* taip pat naudoja nesudėtingą programos profiliavimą, kuris leidžia gauti informaciją apie programos vykdymo šakų padengimą pagal testuojamą įvestį. Kita naudojama technika yra pažeidimų analizė, kuri praplečia testuojamos programos profiliavimą, sekant pažeidžiamų duomenų srautą [18]. Dėl to testavimo įrankis gali keisti specifinius įvesties laukus, kurie gali sukelti potencialius pažeidžiamumus. Pilkosios, kaip ir baltosios dėžės įrankiai randa užslėptas klaidas skirtingai nuo juodosios dėžės metodika paremtų įrankių, tačiau pilkosios dėžės įrankių kūrimas yra sudėtingesnis ir užima daugiau laiko [14]. Pilkosios dėžės testavimo metodas gali būti naudojamas jei yra koncentruojamasi į rezultatų kokybę (įvairumą, pažeidžiamumus ir klaidų skaičių) ir norima pasiekti didesnį kodo padengimą [14].

lentelė 1 Testavimo metodų palyginimas

	Baltosios dėžės metodas	Juodosios dėžės metodas	Pilkosios dėžės metodas
Kokybiški rezultatai (klaidų įvairovė, skaičius, pažeidžiamumas)	**X**		**X**

Greitas testavimas laiko atžvilgiu		**X**	
Nereikalauja programinio kodo		**X**	**X**
Didelė kodo padengimo procentinė dalis	**X**		**X**
Randamos paviršutiniškos klaidos	**X**	**X**	**X**
Randamos gilios klaidos	**X**		**X**

Apibendrinant galima teigti, kad pilkosios dėžės metodas yra tinkamiausias dinaminei programų pažeidžiamumų paieškai atlikti. Pagal kriterijus išvardintus lentelė 1 galima pamatyti, kad skirtingai nei baltosios dėžės metodas, šis nereikalauja papildomų duomenų (programinio kodo), tam, kad būtų galima atlikti testavimo procesą. Taip pat, lyginant su juodosios dėžės metodu, suteikia patikimesnius ir greitesnius rezultatus, kadangi atliekamas testuojamos programos profiliavimas. Tai leidžia pasiekti didesnį kodo padengimą nei su juodosios dėžės metodu.

1.5. Programų komunikacijos naudojant nuotolinių procedūrų iškvietimą

Vienas iš dažniausiai naudojamų programinės įrangos tarp-procesinės komunikacijos būdų yra nuotolinių procedūrų iškvietimas (angl. *Remote Procedure Call*). Nuotolinių procedūrų iškvietimas yra naudojamas iškviesti procedūrą, kuri egzistuoja kito proceso adresų erdvėje [19]. Tas procesas gali būti vykdomas tame pačiame ar kitame kompiuteryje tame pačiame tinkle. Nuotolinių procedūrų iškvietimo karkasai remiasi kliento-serverio modeliu ir dažniausiai veikia naudodami TCP/IP protokolą [20]. Egzistuoja daugybė nuotolinių procedūrų iškvietimo karkasų, todėl šioje analizės dalyje bus apžvelgti šie pagrindiniai karkasai: *gRPC*, *WCF* ir *Apache Thrift*.

gRPC [21] – vienas iš dažniausiai naudojamų nuotolinių procedūrų iškvietimo karkasų. Šis karkasas yra sukurtas *Google* kompanijos. Šis karkasas komunikacijai naudoja HTTP/2 protokolą, kuris veikia naudojant TCP protokolą. Duomenų objektų pavertimui į baitų srautą yra naudojamas *Protocol Buffers* (protobuf) formatas. *gRPC* karkasas naudoja *Protobuf* formatą kaip paslaugų ir pranešimų sąsajų aprašymo kalbą. Šis karkasas taip pat palaiko saugų duomenų perdavimą su autentifikacija ir šifravimu naudojant TLS protokolą. PĮ kūrėjai programų kūrimo sąsajas turi aprašyti atskiruose „.proto“ failuose naudojant *Protobuf* sąsajų aprašymo kalbą. Šie failai yra naudojami sukompiliuojant kodą skirtą serveriui ir klientui. *gRPC* programų kūrimo sąsają sudaro paslaugos ir pranešimai. Paslauga yra sudaryta iš nuotolinių procedūrų, kurios atlieka prieigos taškų funkciją. Pranešimas yra duomenų esybė, kuri sustruktūrizuota naudojant numeruojamus laukus. Laukas taip pat gali būti kitas pranešimas taip leidžiant naudoti pranešimus pranešimuose. Kai nuotolinių procedūrų prieigos taškas nuskaito pranešimą ir neatpažįsta tam tikro lauko, tas laukas yra ignoruojamas. Visi pranešimo laukai yra nebūtini, todėl jei laukas nėra pateikiamas duomenų objektų pavertimo į baitų srautą ar iš jo metu, to lauko reikšmė nustatoma į nulį ar kitą numatytą reikšmę. Šis karkasas gali būti naudojamas

skirtingose programavimo kalbose ir platformose, taip išlaikant nepriklausomumą tarp naudojamų platformų ir programavimo kalbų.

Windows Communication Foundation (**WCF**) karkasas yra orientuotas į paslaugas orientuotas programas [22]. Naudojant šį karkasą galima asinchroniškai siųsti pranešimus iš vienos programos į kitą. Klientai gali kviesti skirtingas paslaugas, o paslaugos gali būti naudojamos skirtingų klientų. Klientas prisijungia prie *WCF* paslaugos pasinaudojant prieigos tašku. Prieigos taškas turi savo adresą ir tam tikrus nustatymus, kuriais nurodoma kaip bus perduodami duomenys. Duomenys gali būti perduodami pasinaudojant skirtingomis technologijomis: HTTP, TCP protokolais, vardiniais tuneliais (angl. *named pipes*) ir *MSMQ* pranešimų eilėmis [22]. Paslaugos gali turėti viešai pasiekiamą WSDL sąsają, kuri yra sukuriama programos veikimo metu. Pasinaudojus šia sąsaja kiekvienas *WCF* klientas gali naudotis paslauga nepaisant platformų skirtumų. Šio karkaso veikimo principas yra visiškai skirtingas lyginant su kitais aprašomais karkasais, kurie naudoja žmonėms suprantamą atskirą sąsajų aprašymo kalbą. Šis karkasas gali būti naudojamas *Windows*, *Linux* ir *MacOS* operacinėse sistemose, tačiau jis yra skirtas tik *.NET* karkasu paremtomis programavimo kalboms, tokioms kaip C#, VB.NET ir kitos [23].

Apache Thrift, dar kitaip vadinamas *Thrift*, yra karkasas, kuris apjungia duomenų objektų pavertimą į baitų srautą ir nuotolinių procedūrų iškvietimo kodo kūrimą [24]. *Thrift* karkasas sukuria programų sąsajų kodą naudojant sąsajų aprašymo kalbą, kuria galima aprašyti paslaugas ir pranešimus tam skirtuose *.thrift* failuose. Karkase naudojamos sąsajų aprašymo kalbos sintaksė yra paremta C programavimo kalbos sintakse, todėl pranešimai ir kiti objektai yra aprašomi naudojant *struct* elementą [25]. Šis karkasas taip pat palaiko keletą komunikacijos protokolų. Karkasas gali naudoti skirtingus duomenų objektų pavertimo į baitų srautą metodus. Dvejetainis duomenų objektų pavertimas į baitų srautą naudojamas pasiekti kuo didesnį vykdymo greitį, o kompaktiškas pavertimas geba suspausti siunčiamą pranešimą. Šis karkasas yra sukurtas siekiant išlaikyti platformos ir programavimo kalbos nepriklausomumą, kadangi palaikoma keletas skirtingų programavimo kalbų.

Apibendrinant tarp-procesinės komunikacijos vykdymą naudojant nuotolinį procedūrų iškvietimą, kiekvienas programuotojas renkasi metodą ir karkasą, kuris labiausiai atitinka norimą tikslą ir gali prisitaikyti prie naudojamų technologijų. *gRPC* ir *Apache Thrift* karkasai gali būti naudojami nepriklausomai nuo programavimo kalbos pasirinkimo, lyginant su *WCF*, kuris gali būti naudojamas tik *.NET* karkasu paremtomis kalbomis (C#, VB.NET ir kt.). *WCF* karkasas gali nenaudoti atskiros sąsajų aprašymo kalbos, lyginant su *gRPC* ir *Apache Thrift*. Šie karkasai turi atskiras sąsajų aprašymo kalbas, kurios yra būtinos siekiant aprašyti pranešimus ir teikiamas paslaugas.

1.6. Analizės išvados

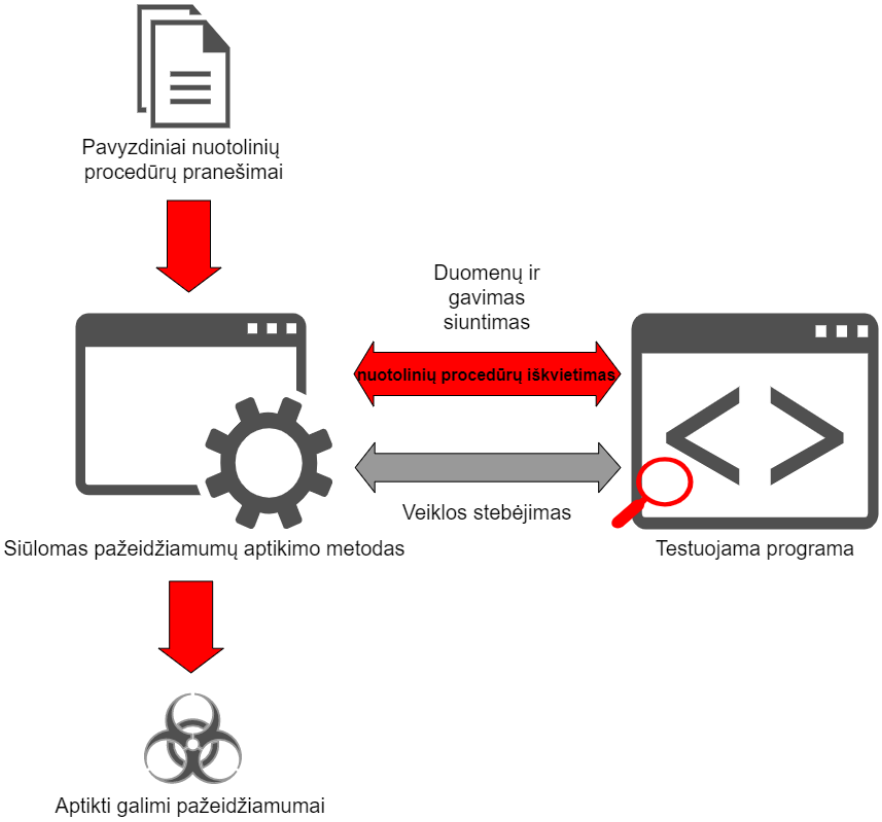
1. Buvo analizuojami trys pagrindiniai tarp-procesinėje komunikacijoje naudojami nuotolinių procedūrų iškvietimo karkasai: *gRPC, WCF* ir *Apache Thrift*. Siekiant universalumo buvo išskirti *gRPC* ir *Apache Thrift* karkasai, kurie gali būti lengvai įgyvendinami nepriklausomai nuo programavimo kalbos.
2. Išanalizavus juodosios, baltosios ir pilkosios dėžių testavimo metodų pritaikymą programų pažeidžiamumų aptikimui pastebėta, kad pilkosios dėžės metodas yra tinkamiausias atlikti dinaminę programų pažeidžiamumų paiešką. Skirtingai nei baltosios dėžės metodas, šis nereikalauja programinio kodo, tam, kad galėtų atlikti testavimo procesą. Lyginant su juodosios dėžės metodu, šis metodas suteikia patikimesnius ir greitesnius rezultatus.
3. Išanalizavus statinius ir dinaminius testavimo metodus nustatyta, kad priimtiniausias yra miglotąja logika paremtas metodas dėl savo universalumo ir gebėjimo derinti tiek statinės, tiek dinaminės analizės privalumus.
4. Remiantis analizės rezultatais nustatytas būtinumas sudaryti programų pažeidžiamumų aptikimo integruotą metodą naudojant nuotolinį procedūrų iškvietimą. Šis metodas galėtų aptikti kreipimosi į nulinę rodyklę ir talpyklos perpildymo pažeidžiamumus. Siūlant magistro darbo sprendimą tikslinga įgyvendinti dinaminį miglotosios logikos, pilkosios dėžės metodus ir panaudoti *gRPC* arba *Apache Thrift* karkasus komunikacijai su programine įranga.

2. Programų pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą

Šiame skyriuje aprašomas siūlomas programų pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą, modelis. Pateikiama architektūra, metodo sudedamųjų dalių detalūs aprašymai ir diagramos.

2.1. Programų pažeidžiamumų aptikimo metodo koncepcija ir modelis

Tarp-procesinė komunikacija sudėtingoje programinėje įrangoje užima vis didesnę taikomųjų programų kodo bazės dalį. Programinėje įrangoje naudojamasi nuotolinių procedūrų iškvietimo karkasais, kurie įgalina tarp-procesinius duomenų mainus. Netinkamai įgyvendintos nuotolinės procedūros gali sukelti programinio kodo pažeidžiamumus, kuriuos galima išnaudoti piktavališkiems tikslams. Dauguma programų pažeidžiamumų aptikimo metodų yra orientuoti į paviršutiniškus pažeidžiamumus esančius programiniame kode. Tačiau šie metodai susiduria su problemomis, kurios gali būti pastebimos tik programos vykdymo metu. Siekiant rasti pažeidžiamumus pasinaudojant nuotoliniu procedūrų iškvietimu, turėtų būti naudojamas tam skirtas modelis (žr. pav. 5).



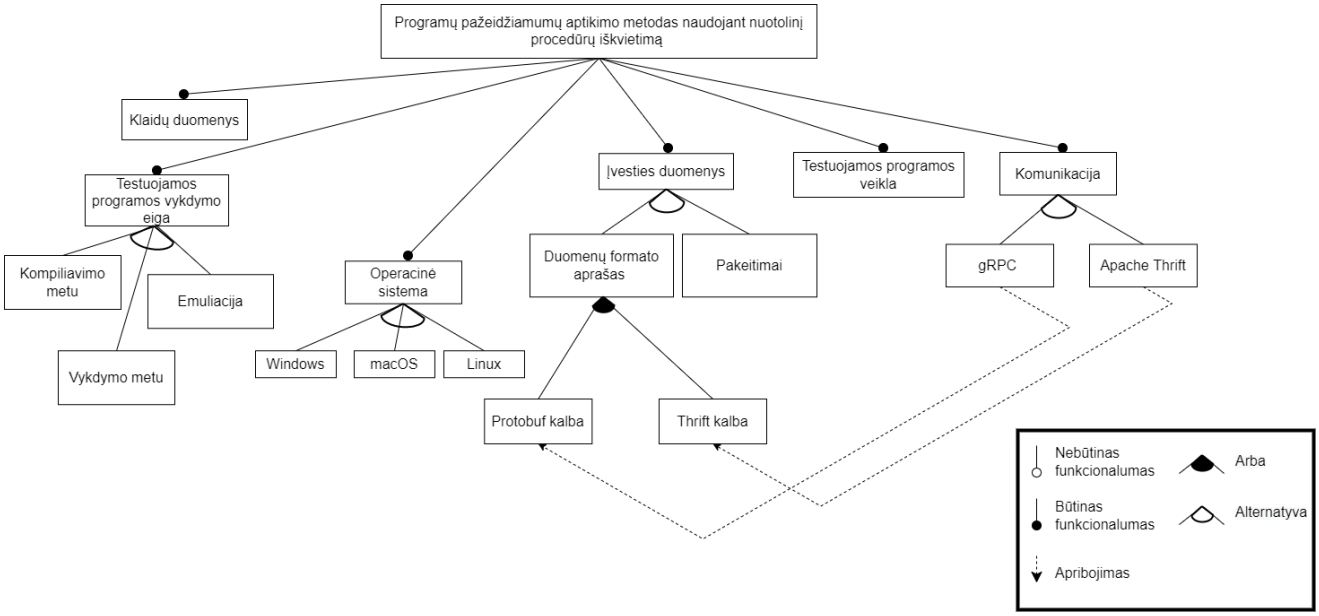
pav. 5 Programų pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą koncepcija

Kadangi metodas yra orientuotas į programų nuotolinių procedūrų iškvietimą, todėl pirmiausiai turi būti atlikta analizė siekiant išsiaiškinti kokie nuotolinių procedūrų pranešimai turi būti siunčiami testuojamai programai. Pranešimai turi būti kuriami pagal turimus pavyzdinius pranešimus. Sudarytas integruotas programų pažeidžiamumų aptikimo metodas naudojant nuotolinių procedūrų iškvietimą, turėtų atlikti šias funkcijas:

- Panaudoti programos vykdymo eigos stebėjimą siekiant padengti kuo daugiau testuojamos programos kodo;

- Kurti nuotolinių procedūrų pranešimus, kurie bus siunčiami panaudojant *gRPC* arba *Apache Thrift* nuotolinių procedūrų karkasą;
- Stebėti testuojamos programos veikimą ir registruoti veikimo sutrikimus;
- Analizuoti surinktus veikimo sutrikimus ir pranešti apie galimus pažeidžiamumus.

Dažniausiai dinaminiu miglotosios logikos metodu paremti pažeidžiamumo metodai (žr. pav. 2) susideda iš trijų pagrindinių dalių: testuojamos programos veiklos stebėjimo, testavimo įvesčių kūrimo ir komunikacijos su testuojama programa. Atsižvelgiant į tipinius pažeidžiamumų aptikimo metodus, nuspręsta papildyti metodą įgyvendinant programos vykdymo eigos stebėjimą siekiant padengti kuo daugiau testuojamos programos kodo. Taip pat praplėsti metodą duomenų siuntimo įgyvendinimu panaudojant *gRPC* arba *Apache Thrift* nuotolinių procedūrų karkasą. Pavyzdinio integruotojo testavimo metodo požymių diagrama pavaizduota pav. 6.



pav. 6 Programų pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą modelis

Metodas bus sudarytas iš šio funkcionalumo:

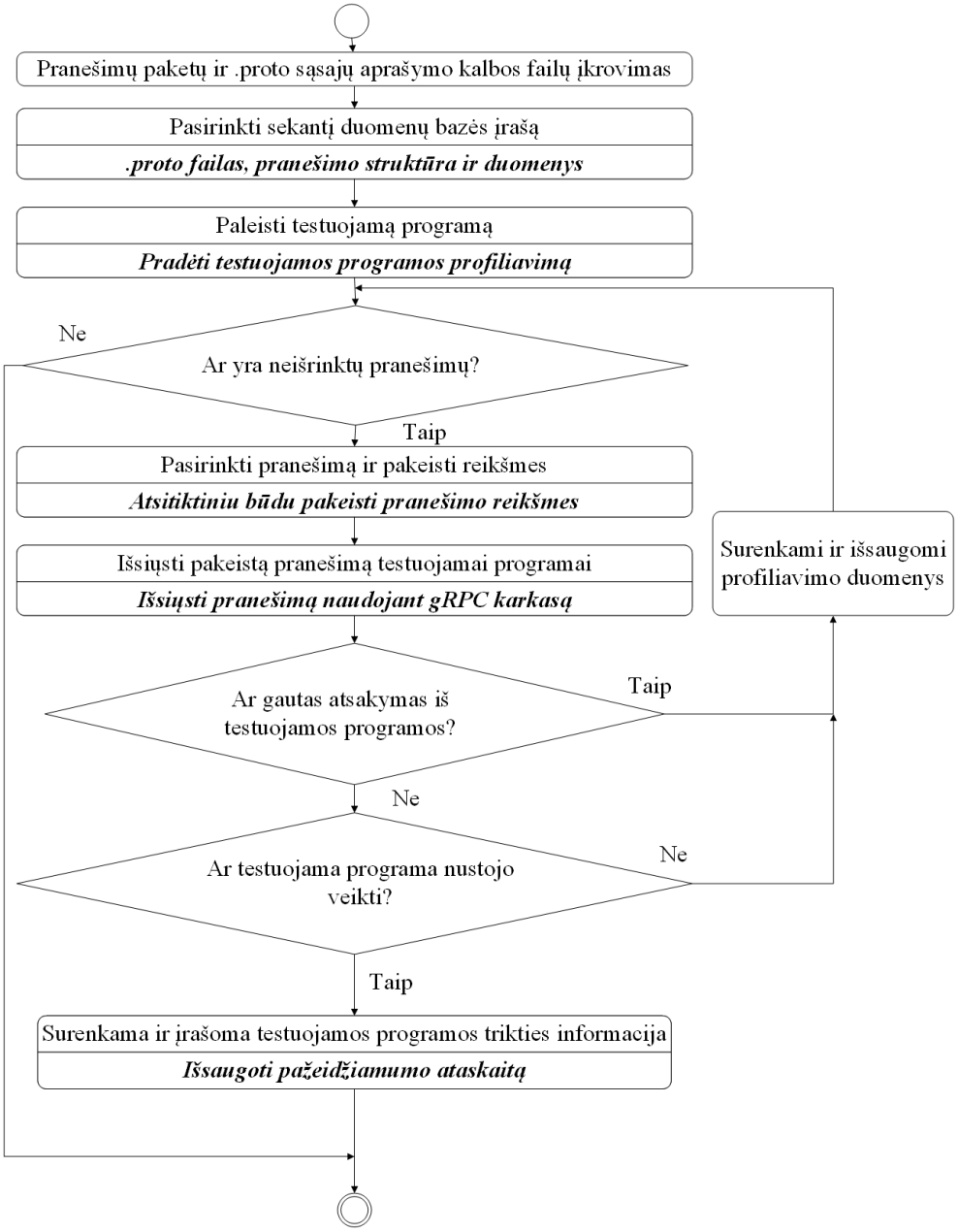
- Klaidų duomenų surinkimo – požymių diagramoje tai atitinka *Klaidų duomenys* požymį. Šiuo funkcionalumu bus siekiama atpažinti programos veikimo sutrikimus. Iš testuojamos programos klaidų išvesčių išgaunama informacija apie tikėtinas klaidas ir jų priežastis;
- Testuojamos programos vykdymo eigos informacijos gavimo – požymių diagramoje tai atitinka *Testuojamos programos vykdymo eiga* požymį. Šiuo funkcionalumu surenkama informacija apie testuojamos programos vykdomo kodo padengimą pagal duotus nuotolinių procedūrų pranešimus. Taip pat atrenkami ir prioretizuojami pranešimai pagal padengiamą kodo kiekį siekiant padidinti testavimo greitį ir efektyvumą. Siūlomas metodas naudos vykdymo eigos stebėjimą vykdymo metu;
- Nuotolinių procedūrų pranešimų kūrimo – požymių diagramoje tai atitinka *Įvesties duomenys* požymį. Šis funkcionalumas atsakingas už nuotolinių procedūrų pranešimų kūrimą pagal duotus pradinius pranešimus ir vykdymo eigos informaciją. Projektuojamo metodo nuotolinių procedūrų pranešimų kūrimas bus paremtas duomenų formatų aprašų technika, kuri naudos *Protobuf* kalbą;
- Programos veiklos stebėjimo – požymių diagramoje tai atitinka *Testuojamos programos veikla* požymį. Funkcionalumas yra skirtas programų veikimui stebėti ir išskirti įvykusius

veikimo sutrikimus (t. y. netikėtas programos veiklos nutraukimas, atsako į užklausas nebuvimas);
- Komunikacijos su testuojama programa – požymių diagramoje tai atitinka *Komunikacija* požymį. Funkcionalumas skirtas nuotolinių procedūrų pranešimų siuntimui ir atsakymų gavimui iš testuojamos programos proceso. Nuotolinių procedūrų iškvietimas bus vykdomas naudojant *gRPC* karkasą.

Metodo modelyje nėra apibrėžiama, kaip turėtų būti realizuotos specifinės dalys ir moduliai. Tai priklauso nuo funkcionalumo realizacijos. Pasirinkti realizacijos būdai privalo atitikti ir patenkinti darbe apibrėžiamų procesų veikimą. Šis metodas bus įgyvendinamas *Windows* operacinėje sistemoje.

Programų pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą, algoritmą, kuris yra pavaizduotas pav. 7, sudaro šie žingsniai:

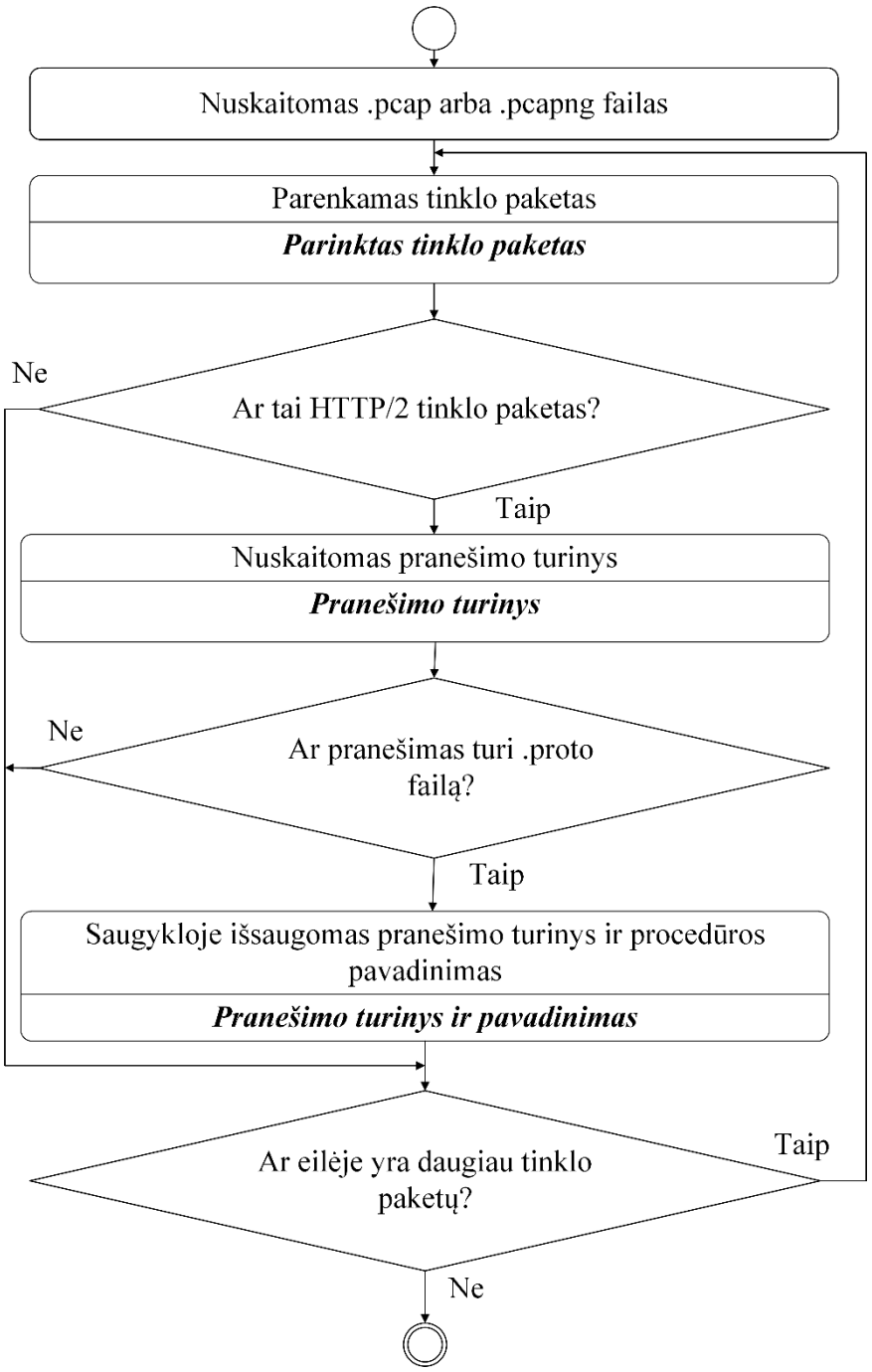
1. Įkeliami nuotolinių procedūrų pranešimų paketų ir *Protobuf* sąsajų aprašymo kalbos failai, kurie yra skirti užfiksuotiems pranešimams. Atliekamas duomenų apdorojimas.
2. Testuojama programa yra paleidžiama ir yra pradedamas vykdymo eigos stebėjimas (profiliavimas).
3. Pradedamas nuotolinių procedūrų kūrimo procesas, sukuriamas pakeistas pranešimas.
4. Naujai sukurtas nuotolinių procedūrų pranešimas yra siunčiamas testuojamai programai. Jei atsakas nėra gaunamas, patikrinama ar testuojama programa vis dar veikia.
5. Jei buvo gautas testuojamos programos atsakas, surenkama testuojamos programos vykdymo eigos informacija, kuri yra išsaugoma ir perduodama testavimo pranešimų kūrimo procesui. Pradedama nauja testavimo iteracija. Tai yra tęsiama kol yra neištestuotų pranešimų.
6. Jei testuojamos programos veiklos stebėjimo procesas užfiksuoja triktį susijusią su testuojama programa, klaidų atrinkimo procesas surenka ir apdoroja visą susijusią trikties informaciją. Algoritmas yra užbaigiamas.



pav. 7 Programų pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą algoritmas

2.2. Pranešimų paketų ir *Protobuf* sąsajų aprašymo kalbos failai

Programų pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą, pradedamas nuo pradinių nuotolinių procedūrų pranešimų nuskaitymo (pav. 8 pateikta algoritmo eiga). Metodas priims duomenis iš *pcap* ir *pcapng* tinklo paketų duomenų failų, kuriuos galima gauti panaudojant *tcpdump* ar *Wireshark* tinklo paketų analizatorius. Metodas taip pat priims *.proto Protobuf* sąsajų aprašymo kalbos failus, kurie bus naudojami filtruojant nereikalingus pranešimus ir siunčiant pranešimus testuojamai programai.



pav. 8 Pranešimų paketų ir *Protobuf* sąsajų aprašymo kalbos failų nuskaitymo algoritmas

Nuskaičius paketų ir sąsajų aprašymo kalbos failus yra pradedamas *Protobuf* pranešimų nuskaitymas. Šie pranešimai yra naudojami *gRPC* nuotolinių procedūrų iškvietimo karkase [21]. *gRPC* karkasas remiasi HTTP/2 [26] protokolu, todėl norint išrinkti pranešimus reikia išrinkti visas HTTP/2 protokolo užklausas. Nuskaičius HTTP/2 užklausos turinį yra atliekamas patikrinimas ar šis pranešimas yra skirtas bent vienai *proto* failuose aprašytai testuojamos programos paslaugai. Jei pranešimas turi paslaugos atitikmenį, duomenys yra išsaugomi. Jei atitikmuo nebuvo rastas, algoritmas tęsiamas su kita HTTP/2 užklausa. Iš šių užklausų yra išgaunami dvejetainio formato *Protobuf* pranešimai, kurie buvo sukurti pagal *Protocol Buffer* [27] sąsajų aprašymo kalbą. Šios kalbos pavyzdys pateiktas žemiau:

```
message testRequest {
    int32 a = 1;
}
message testResponse {
    int32 b = 1;
}
service testService {
    rpc Test(testRequest) returns (testResponse) {};
}
```

Aukščiau aprašytas pavyzdys, kuris turi *testRequest* užklausos ir *testResponse* atsakymo pranešimus. Pranešimai turi vieną skaitinį lauką. Skaičius šalia lauko yra unikalus identifikatorius, kuris bus naudojamas atpažinti lauką, pranešimą pavertus į *Wire* formatą. Pavyzdyje taip pat aprašoma *testService* paslauga, kuri turi *Test* metodą. Šis metodas naudoja abu aukščiau aprašytus pranešimus. Jei pagal šį aprašymą yra sukuriamas pranešimas su a reikšme, kuri lygi 150, šis pranešimas yra išvedamas šešioliktainiame formate. Pranešimo pavyzdys pateikiamas žemiau:

```
08 96 01
```

Žalia spalva pažymėtas skaičius pranešimo pradžioje nurodo užkuoduotos reikšmės tipą. Šiuo atveju tai yra *varint* tipo reikšmė. Raudona spalva pažymėta dalis nurodo užkuoduotą reikšmę. Tada šie du baitai yra paverčiami į dvejetainį formatą ir yra pašalinami reikšmingiausi bitai pažymėti raudona spalva.

```
96 01 = 1001 0110  0000 0001 = 001 0110  000 0001
```

Sudedame šiuos dvejetainius skaičius ir apsukame rezultatą, dvejetainį skaičių paverčiame į dešimtainį formatą.

```
001 0110 +
000 0001
__________
001 01110 = 10010110
128 + 16 + 4 + 2 = 150
```

Galiausiai, algoritmas išsaugo pranešimą šešioliktainiu formatu, taip pat yra išsaugomas paslaugos metodo adresas. Aukščiau aprašyto pavyzdžio atveju tai būtų *testService/Test*. Pranešimas užkoduotas šešioliktainiu formatu suteikia tik būtiną informaciją apie laukų skaičių ir jų reikšmes, tačiau nesuteikia informacijos apie tikslų laukų reikšmių tipą, laukų pavadinimus ir kitas pranešimo ypatybes. Tam, kad būtų galima gauti tikslius duomenų reikšmių tipus ir apribojimus reikia interpretuoti duomenis į realią pranešimo duomenų struktūrą. Tai yra atliekama panaudojant ankščiau algoritmo nuskaitytus *Protobuf* sąsajų aprašymo failus ir *Google* kompanijos sukurtas *Protobuf* programines bibliotekas. Apie tai nebus kalbama, kadangi *Google* kompanija suteikia visą reikiamą dokumentaciją ir įrankius.

Yra galimybė, kad *Protobuf* sąsajos aprašymo failai nebus pateikti. Tokiu atveju metodas neveiks, kadangi sąsajos aprašymo failai yra reikalingi interpretuojant duomenis į realias pranešimų duomenų struktūras, kurios yra siunčiamos testuojamai programai.

2.3. Programų pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą

Programų pažeidžiamumų aptikimo metodas naudojant nuotolinį procedūrų iškvietimą, tęsiamas startuojant testuojamą programą. Startavus testuojamai programai yra įgalinamas jos vykdymo eigos stebėjimas. Tada yra pradedamas nuotolinių procedūrų pranešimų kūrimo ir siuntimo ciklas, kuris yra tęsiamas kol yra likusių dar neišsiųstų pranešimų. Detalesnis kiekvieno etapo veikimas pristatomas kituose skyreliuose.

2.3.1. Programos vykdymo eigos stebėjimas

Siekiant padidinti testavimo metodo vykdymo greitį ir efektyvumą atrandant galimus pažeidžiamumus buvo įgyvendintas programų vykdymo eigos stebėjimas, kurio metu atliekami šie žingsniai:

1. Paleidžiama testuojama programa;
2. Pradedamas nurodytų funkcijų vykdymo eigos stebėjimas;
3. Testuojamai programai išsiunčiama testavimo įvestis;
4. Fiksuojami įvykdyti testuojamos programos instrukcijų blokai;
5. Baigiamas nurodytų funkcijų vykdymo eigos stebėjimas;
6. Gaunami įvykdyti testuojamos programos instrukcijų blokai.

Procesas prasideda nuo testuojamos programos paleidimo ir vykdymo eigos stebėjimo įgalinimo. Šis įgalinimas vykdomas panaudojant vieną iš žemiau aprašytų metodų. Įgalinimo metu pradedamas nurodytų testuojamos programos metodų vykdymo eigos stebėjimas. Metodai yra nurodomi tiesioginiu jų pavadinimu arba absoliučiu metodo adresu testuojamoje programoje. Programa, gavusi naują nuotolinių procedūrų pranešimą atlieka operacijas priklausomai nuo pranešimo tipo ir reikšmių. Šios operacijos yra interpretuojamos kaip instrukcijų blokai, kurie buvo įvykdyti metodo veikimo metu. Blokas sudarytas iš šios informacijos:

- Instrukcijų bloko pradžios adreso;
- Instrukcijų bloko pabaigos adreso;
- Modulio pavadinimo, kuriam priklauso šis instrukcijų blokas.

Baigus vykdyti programos metodą eigos stebėjimas yra sustabdomas ir visi vykdymo eigos duomenys yra perduodami tolimesniam įvesčių kūrimo ir siuntimo etapui.

Tam, kad procesas galėtų atlikti testuojamos programos vykdymo eigos stebėjimą, reikalingas papildomas testuojamos programos profiliavimas. Profiliavimą galima atlikti naudojant kelis skirtingus metodus: įkompiliuojant profiliavimo instrukcijas [28], užkraunant profiliavimo instrukcijas vykdymo pradžioje [28], ir atliekant visos programos emuliaciją virtualioje mašinoje [13]. Profiliavimo instrukcijų įkompiliavimas reikalauja turėti testuojamos programos kodą. Tokie testavimo įrankiai kaip *AFL* [13] suteikia savo nuosavą kompiliatorių, kuris prideda profiliavimo instrukcijas sukompiliuotoje programoje. Kitas būdas kaip atlikti programos profiliavimą yra užkraunant profiliavimo instrukcijas programos vykdymo pradžioje. Tai gali būti naudinga, kai neįmanoma gauti testuojamos programos kodo. Dinaminės analizės įrankiai tokie kaip *Valgrind* [29], *Frida* [30] ir *DynamoRIO* [31], suteikia galimybę užkrauti savo kodą testuojamos programos kontekste, taip įgalinant profiliavimą realiu laiku. Taip pat šie dinaminės analizės įrankiai gali būti naudojami nepriklausomai nuo operacinės sistemos ir platformos. Galiausiai, testuojamos programos

profiliavimą galima atlikti emuliuojant programos veikimą. Ši technika gali būti naudojama *AFL* įrankyje. Šios technikos veikimas remiasi *QEMU* [32] vartotojo emuliavimo režimu [13]. Šios technikos pagrindinis trūkumas yra sulėtėjusi testavimo greitaveika. Panaudojus vieną iš aukščiau paminėtų testuojamų programų profiliavimo būdų galima gauti informaciją apie kodo vykdymo eigą.

Rekomenduojama, kad prieš programos testavimo proceso pradžią būtų sudarytas testuojamos programos modulių ir funkcijų sąrašas, pagal kurį bus profiliuojama programa. Į profiliuojamų funkcijų sąrašą turi būti įtrauktos funkcijos, kurios atsakingos už *gRPC* pranešimų gavimą ir apdorojimą. Į profiliuojamų modulių sąrašą turi būti įtraukti visi moduliai, kuriuose yra profiliuojamos funkcijos (pagrindinis programos vykdomasis failas, bibliotekų failai), taip pat kiti moduliai, kurie turi funkcijas, kurios gali būti naudojamos vykdymo eigoje. Reikia vengti į profiliuojamų modulių sąrašą dėti bendrinamas standartines (angl. *shared*) bibliotekas, kadangi jos gali priversti procesą sukurti daug profiliavimo duomenų, kas sulėtintų ir pasunkintų testavimo procesą.

2.3.2. Įvesčių priklausomybių tikimybių radimas

Egzistuoja tikimybė, kad norint išnaudoti pažeidžiamumą reikia išsiųsti keletą susijusių nuotolinių procedūrų pranešimų, pavyzdžiui, pažeidžiamas kodas yra įvykdomas tuo metu kai pranešimai yra išsiunčiami šia eilės tvarka: $p1' \rightarrow p2' \rightarrow p3' \rightarrow p4'$. Modifikuojant pirmąjį pranešimą yra kuriamos šio pranešimo variacijos, kurios neturi įtakos pažeidžiamumo radimo progresui, nes visa eiga priklauso nuo kitų tolimesnių pranešimų. Yang ir kt. [10] šią problemą sprendė įgyvendinant tarp-procesinių pranešimų eiliškumo priklausomybių matricą ir slenkančio lango algoritmą. Ši matrica, pavaizduota pav. 9, leidžia nustatyti tikimybę, kad pranešimas p_i turi būti siunčiamas prieš p_j pranešimą.

$$\begin{bmatrix} P_{1,1} & \cdots & P_{1,N} \\ \vdots & \ddots & \vdots \\ P_{N,1} & \cdots & P_{N,N} \end{bmatrix}$$

pav. 9 Nuotolinių procedūrų pranešimų eiliškumo priklausomybių matrica

Tarkime, kad turime N skirtingų nuotolinių procedūrų pranešimų, todėl matrica turėtų būti $N \times N$ dydžio. Slenkančio lango algoritmas yra naudojamas surinktų nuotolinių procedūrų pranešimų eilėje. Lango dydis yra nustatomas pagal nuotolinių paslaugų būsenų perėjimo sudėtingumą. Kuo mažiau skirtingų nuotolinių procedūrų pranešimų yra išsiunčiama, tuo lango dydis yra mažesnis. Jei kiekviename lange pranešimas p_i yra prieš pranešimą p_j, tikimybė esanti matricos eilutėje i ir stulpelyje j, P_{ij} yra padidinama per vienetą. Tikimybė yra suskaičiuojama kiekvienai priklausomai pranešimų porai. Galiausiai, P_{ij} reikšmė turi būti normalizuota, tam, kad kiekvieno stulpelio ar eilutės suma lygi vienetui. Pagal sudarytą nuotolinių procedūrų pranešimų eiliškumo priklausomybių matricą galima nuspręsti ar pranešimai turi būti siunčiami eilės tvarka. Tai leidžia simuliuoti tikrą vykdymo eigą. Šis metodas bus pritaikytas šiame procese. Reikia pastebėti, kad šis priklausomybių radimo procesas bus atliekamas vieną kartą, testavimo proceso pradžioje.

Taip pat egzistuoja galimybė, kad gražinto atsakymo reikšmės gali būti kito nuotolinės procedūros pranešimo įvesties reikšmėmis. Tai yra vadinama reikšmių priklausomybe. Yang ir kt. [10] šią metodologiją apžvelgė ir panaudojo *MachFuzzer* įrankio prototipe. Panaudojant euristinį algoritmą galima patikrinti ar išsiųsto pranešimo rezultato skaitinės ar teksto reikšmės sutampa su sekančio pranešimo skaitinėmis ir tekstinėmis reikšmėmis. Jei sutapimas įvyks, bus išsaugoti sutampančių reikšmių laukų pavadinimai, taip sukuriant paieškos lentelę skirtą šių priklausomybių saugojimui. Lentelės eilutės atitiks priklausomų pranešimų porą. Šioje eilutėje taip pat bus saugoma potenciali reikšmių priklausomybė. Ši lentelė bus panaudota kuriant naujus nuotolinių procedūrų pranešimus su atitinkamomis reikšmėmis, kurios lemia pranešimų poros taisyklingumą. Šis metodas bus pritaikytas šiame procese.

2.3.3. Nuotolinių procedūrų pranešimų sukūrimas ir siuntimas

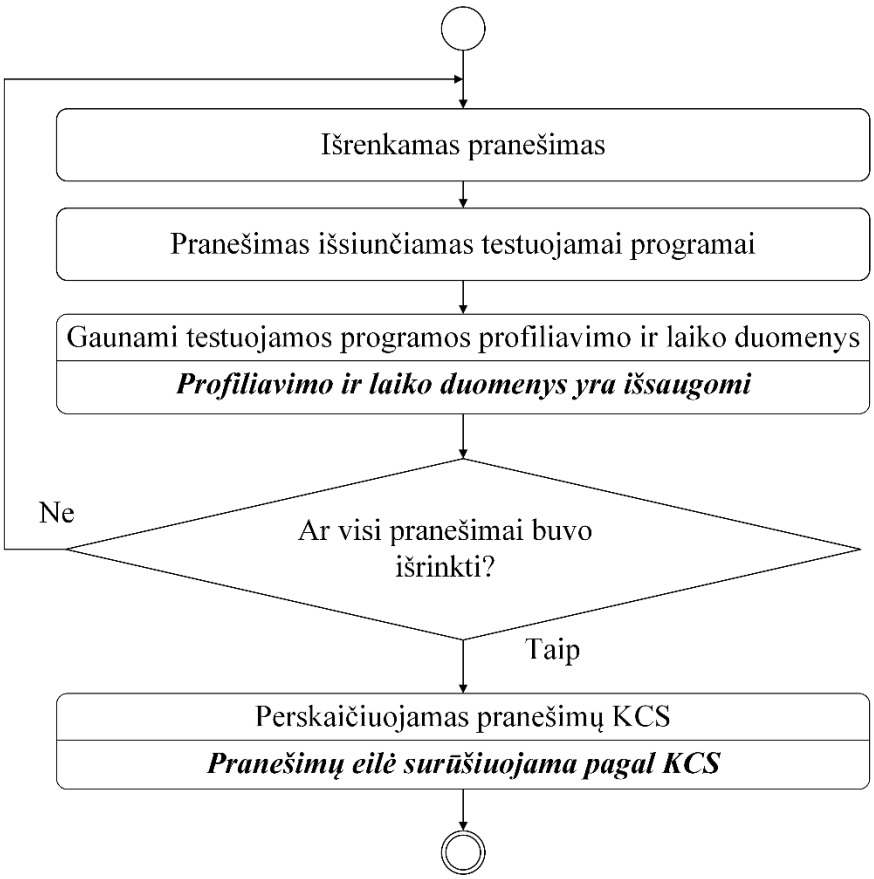
Startavus testuojamai programai ir įgalinus vykdymo eigos stebėjimą yra pradedamas nuotolinių procedūrų pranešimų kūrimas ir siuntimas. Pats pranešimų kūrimas ir siuntimas bus vykdomas panaudojant dvi strategijas:

- Neatsižvelgiant į pranešimų priklausomybes – tai reiškia, kad kiekvienas pranešimas bus traktuojamas kaip visiškai atskiras. Pranešimai bus siunčiami vienas po kito, nepriklausomai nuo pirmumo būtinybės;
- Atsižvelgiant į pranešimų priklausomybes – kiekvienam pranešimui bus ieškoma susijusių pranešimų, kurie turi būti išsiųsti prieš sukurtą pranešimą. Suradus susijusius pranešimus, testavimo metu jie bus išsiunčiami prieš sukurtą pranešimą.

Numatomas nuotolinių procedūrų pranešimų keitimų ciklų skaičius. Abiejų strategijų atveju, prieš testavimo įvesčių siuntimą, reikia apskaičiuoti numatomą testavimo įvesčių keitimų ciklų skaičių. Diagramose bus naudojama sutrumpinta termino forma KCS. Panaši metrika, vadinama energija, yra naudojama *AFL* [13] miglotosios logikos metodo testavimo įrankyje. Taip yra apibūdinama kiek ilgai turėtų būti keičiama viena testavimo įvestis ir iš jos kuriamos naujos įvestys, kol yra pereinama prie kitos įvesties. *AFL* įrankis apskaičiuoja testavimo įvesties energiją pagal šiuos kriterijus [33]:

- Testuojamos programos kodo padengimą – prioretizuojamos įvestys, kurios padengia daugiau programos kodo;
- Vykdymo laiką – prioretizuojamos įvestys, kurios yra įvykdomos greičiau;
- Įvesčių radimo laiką – prioretizuojamos įvestys, kurios buvo rastos vėliausiai.

Energijos principas bus naudojamas šiame metode. Be aukščiau aprašytų kriterijų, bus įtrauktas papildomas kriterijus – nuotolinės procedūros pranešimo sudėtingumas. Tai reiškia, kad pranešimas su daugiau reikšmių laukų turės didesnį numatomą testavimo įvesčių keitimų ciklų skaičių nei įvestis su mažesniu laukų skaičiumi. Supaprastintas numatomo testavimo įvesčių keitimų ciklų skaičiaus apskaičiavimo algoritmas pateiktas pav. 10.



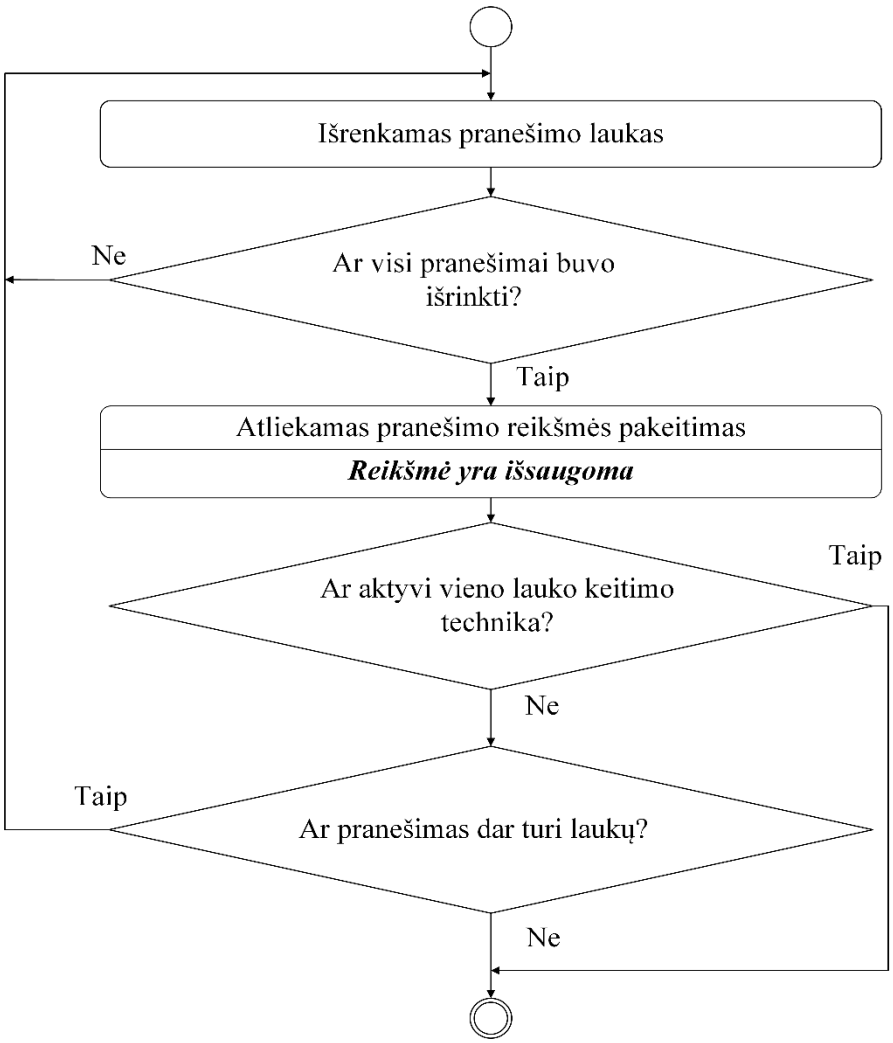
pav. 10 Supaprastintas numatomo testavimo įvesčių keitimų ciklų skaičiaus apskaičiavimo algoritmas

Kaip galima matyti iš algoritmo diagramos, kiekvienas originalus pranešimas yra išsiunčiamas testuojamai programai. Pasinaudojus vykdymo eigos stebėjimu, kuris buvo aprašytas ankstesniame skyrelyje, yra gaunamas kiekvieno pranešimo kodo padengimas (instrukcijų blokų pavidalu) ir laikas, per kurį programa apdorojo šią testavimo įvestį. Tada šie duomenys yra pridedami prie testavimo įvesties. Kai sąraše nebelieka pranešimų, kuriuos reikėtų išsiųsti, yra pradedamas keitimų ciklų skaičiaus perskaičiavimas atsižvelgiant į aukščiau ankščiau aprašytus kriterijus. Galiausiai pranešimai, išrikiuoti pagal apskaičiuotą vertę mažėjimo tvarka yra sudedami į eilę, kurią naudos testavimo ciklas.

Nuotolinių procedūrų pranešimų sukūrimas. *Protobuf* sąsajų aprašymo kalbos techninis aprašas [27] nurodo, kad pranešimai gali turėti 15 skaliarinių reikšmių tipų. Testavimo įvesčių keitimo metu originalaus *Protobuf* sąsajų aprašymo kalbos pranešimo reikšmės yra pakeičiamos pagal techniką nurodytą lentelė 2. Pats naujų testavimo įvesčių kūrimas gali būti atliekamas dvejais būdais:

- Vienu metu keičiant tik vieno lauko reikšmę;
- Vienu metu keičiant visas laukų reikšmes.

Yra galimybė, kad specifiniai pranešimo laukai negali būti keičiami, nes laukai yra atsakingi už pranešimų sąryšį. Naujų įvesčių kūrimo algoritmas leidžia nurodyti laukus, kurie nebus keičiami. Supaprastinta algoritmo diagrama pavaizduota pav. 11.



pav. 11 Supaprastintas naujų testavimo įvesčių kūrimo algoritmas

Algoritmas veikia keisdamas kiekvieną pranešimo lauką. Pirmiausia yra patikrinama ar galimi šio lauko reikšmės pakeitimai. Jei pakeitimai leidžiami, tai atliekama pasinaudojus technikomis, kurios aprašytos žemiau esančioje lentelė 2. Jei lauko pakeitimai nėra leidžiami, yra išrenkamas kitas pranešimo laukas. Jei taikomas vieno lauko keitimo metodas, algoritmas užbaigiamas. Jei keičiami visi pranešimo laukai, tikrinama ar dar yra nekeistų laukų ir jei tai tiesa, ciklas yra tęsiamas.

lentelė 2 *Protobuf* sąsajų aprašymo kalbos skaliarinių reikšmių tipai ir keitimo technikos

Protobuf sąsajų aprašymo kalbos pranešimo reikšmės tipas	Taikoma pakeitimo technika
string	Teksto eilutės ilginimas iki 2^{32} simbolių. Teksto eilutės turinio panaikinimas.
bool	Reikšmės keitimas į true arba false.
bytes	Baitų pridėjimas iki 2^{32} baitų. Visiškas baitų pašalinimas.

sfixed64, sfixed32, fixed64, fixed32, sint32, sint64, uint32, unit64, int32, int64	Pagal lauko skaitinės reikmės tipą yra keičiama skaitinė reikšmė sukeliant skaitinės reikšmės perpildymą (angl. *overflow*) arba atvirkštinį perpildymą (angl. *underflow*).
float, double	Keičiamos reikšmės, kurios gali išprovokuoti nenumatytas klaidas, pvz. neskaičius (*NaN*), maksimali teigiama, minimali teigiama, maksimali neigiama, minimali neigiama reikšmės, teigiama ir neigiama *epsilon* reikšmė, teigiamos ir neigiamos begalybės reikšmė.

Protobuf sąsajų aprašymo kalbos techninis aprašas taip pat nurodo specialius laukų tipus, kurie gali būti naudojami pranešimuose:

- Išvardijimo laukas (angl. *enumeration*) – laukas, kurio reikšmė yra iš įvardytų pasikartojančių reikšmių sąrašo. Lauko reikšmės atsitiktiniu būdu yra keičiamos į vieną iš reikšmių iš nurodyto sąrašo;
- Pasikartojantis laukas – specialus laukas, kuris nurodo, kad reikšmė gali būti pakartota neribotą kartų skaičių. Tai leidžia įgyvendinti sąrašų funkcionalumą. Jei testuojama programa netikrina sąrašo reikšmių kiekio ar susiduria su problemomis apdorojant didelį duomenų kiekį, tai gali sukelti tam tikrus pažeidžiamumus. Reikšmės iš sąrašo gali būti pašalinamos arba pridedamos.

Atlikus panešimo keitimus pagal aukščiau išvardintas taisykles, pranešimas yra išsiunčiamas testuojamai programai. Pranešimo siuntimo procesas aprašytas kitame skyrelyje.

Nuotolinių procedūrų pranešimų siuntimas. Nuotolinės procedūros pranešimas yra siunčiamas kai tik yra pakeičiamas ir sukuriamas naujas pranešimas. Kadangi metodas yra skirtas *gRPC* nuotolinių procedūrų iškvietimo karkasui, įvestys yra siunčiamos programai panaudojant HTTP/2 protokolą į testavimo proceso pradžioje nustatytą adresą ir prievadą, kuris yra naudojamas testuojamos programos nuotolinių procedūrų pranešimų siuntimui ir gavimui. Tam, kad būtų galima išsiųsti pranešimą, pirmiausia, reikia suformuoti HTTP/2 užklausą. Pavyzdinė užklausa pateikta žemiau:

```
:method = POST
:scheme = http
:path = /google.pubsub.v2.PublisherService/CreateTopic
:authority = pubsub.googleapis.com
grpc-timeout = 1S
content-type = application/grpc
grpc-encoding = gzip

<Pranešimas su ilgio reikšme>
```

Tipinė HTTP/2 užklausa yra sudaryta iš šių komponentų:

- *:method* antraštė – HTTP metodo antraštės reikšmė yra pastovi, t. y. POST;
- *:scheme* antraštė – tai gali būti *http* arba *https* reikšmė priklausomai nuo to ar komunikacija yra vykdoma saugiu kanalu naudojant TLS;
- *:path* antraštė – nurodomas specifinės paslaugos ir procedūros pavadinimas į kurį bus kreipiamasi;

- *:authority* antraštė – nurodomas pranešimo gavėjo adresas;
- *grpc-timeout* antraštė – maksimalus atsakymo laukimo laikas, kurį lauks klientas;
- *content-type* antraštė – užklausoje perduodamų duomenų tipas, kuris visada turėtų prasidėti *application/grpc* tekstu;
- *grpc-encoding* antraštė – papildomas turinio kodavimas, gali būti viena iš reikšmių: *gzip*, *deflate*, *identity* arba *snappy*;
- Pranešimo turinio dalis – ši dalis yra sudaryta iš trijų komponentų:
    - Turinio suspaudimo vėliavėlės – reikšmė gali būti 1 arba 0 priklausomai ar duomenys yra suspausti;
    - Turinio dydis – atvaizduojamas kaip keturių baitų beženklis (angl. *unsigned*) skaičius.
    - Pranešimas – dvejetainio formato duomenys.

Suformuotą užklausą su sukurtu nuotolinės procedūros pranešimu galima siųsti testuojamai programai. Išsiuntus užklausą yra gaunamas atsakymas iš testuojamos programos. Pavyzdinis HTTP/2 atsakymas pateiktas žemiau:

```
:status = 200
grpc-encoding = gzip
content-type = application/grpc

<Pranešimas su ilgio reikšme>

grpc-status = 0
```

Tipinis HTTP/2 atsakymas yra sudarytas iš šių komponentų:

- *:status* antraštė – HTTP statuso kodas, kuris yra 200 jei užklausa buvo priimta. Jei užklausa buvo netinkama, gražinamas atitinkamas HTTP klaidos kodas;
- *grpc-encoding* antraštė – papildomas turinio kodavimas, gali būti viena iš reikšmių: *gzip*, *deflate*, *identity* arba *snappy*;
- *content-type* antraštė – užklausoje perduodamų duomenų tipas, kuris visada turėtų prasidėti *application/grpc* tekstu;
- *grpc-status* antraštė – gražinamas skaitinis klaidos kodas jei pranešimo apdorojimo eigoje įvyko klaida. Jei pranešimas buvo apdorotas sėkmingai, klaidos kodas yra lygus 0;
- Pranešimo turinio dalis – ši dalis yra sudaryta iš trijų komponentų, kurie buvo aprašyti ankščiau.

Priklausomai nuo to ar buvo gautas atsakymas ir kokios yra atsakymo antraščių reikšmės, procesas nustato ar testavimo įvestis buvo priimta ar ne. Sėkmingas duomenų išsiuntimas ir priėmimas nustatymas pagal šias sąlygas:

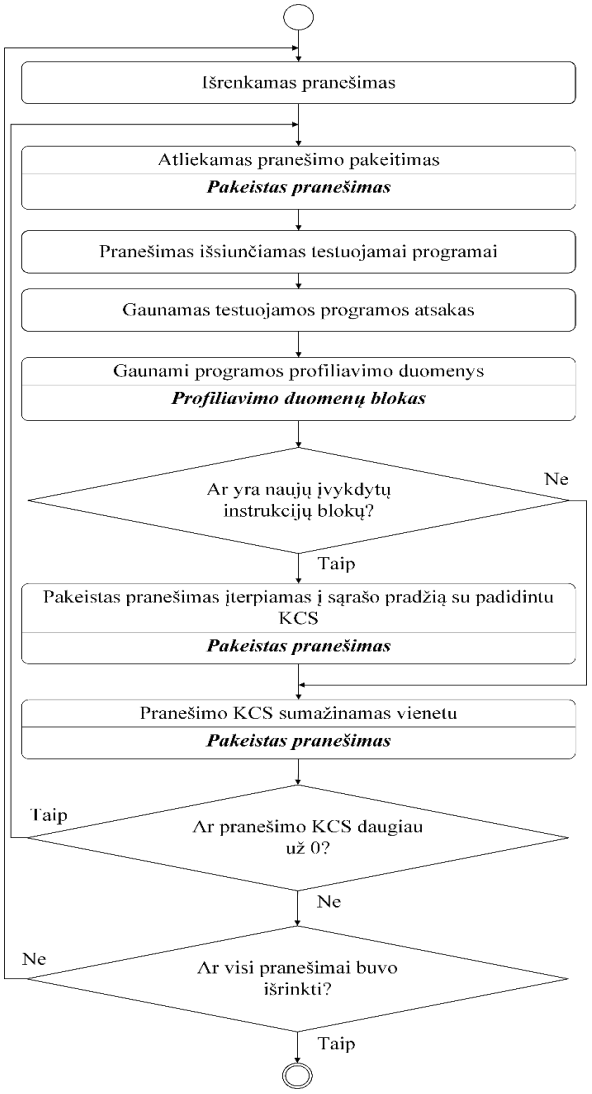
- Siuntimo metu nebuvo gauta jokia operacinės sistemos lygio klaida susijusi su naudojamu komunikacijos prievadu;
- Buvo gautas atsakymas į išsiųstą užklausą su HTTP/2 200 statuso kodu bei nuline *grpc-status* antraštės reikšme.

Atitikus aukščiau paminėtas sąlygas yra laikoma, kad testuojama programa priėmė nuotolinės procedūros pranešimą. Nesėkmingas duomenų išsiuntimas nustatomas esant bent vienai iš šių sąlygų:

- Siuntimo metu buvo gauta operacinės sistemos lygio klaida susijusi su naudojamu komunikacijos prievadu;
- Nebuvo atsakyta į išsiųstą užklausą;
- Gautas atsakymas su nesėkmę indikuojančiu HTTP/2 statuso kodu;
- Gautas atsakymas su HTTP/2 200 statuso kodu, tačiau *grpc-status* antraštės reikšmė nelygi nuliui.

Nepavykus išsiųsti duomenų procesas sukuria įvykį apie nuotolinės procedūros pranešimo siuntimo sutrikimą, kuris yra apdorojamas 2.3.4 skyrelyje aprašytame programos veiklos stebėjimo procese. Nuotolinės procedūros pranešimo siuntimo procesas yra užbaigiamas.

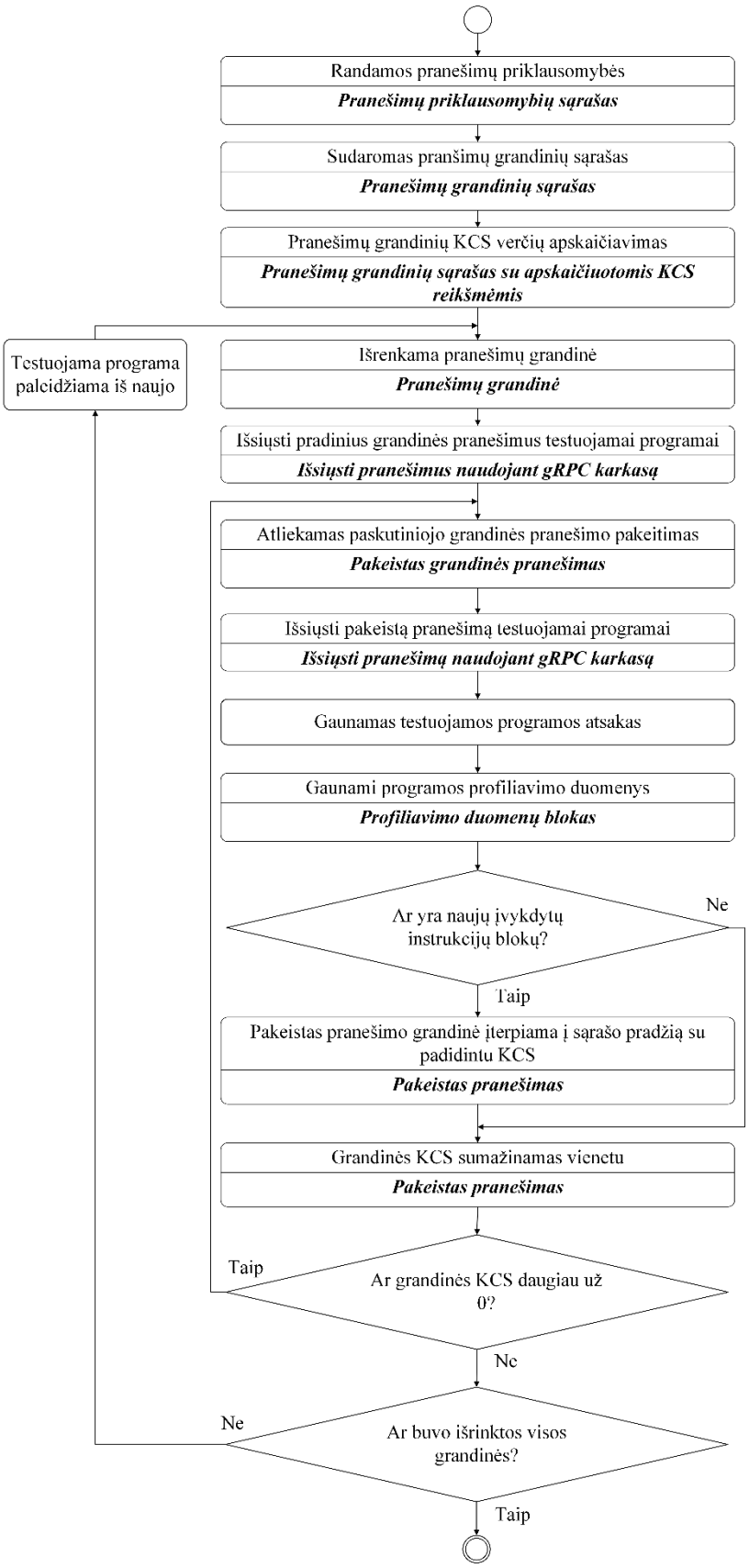
Nuotolinių procedūrų pranešimų sukūrimo ir siuntimo strategija. Atlikus testavimo įvesčių keitimų ciklų reikšmių skaičiavimus galima pradėti pranešimų kūrimo ir siuntimo ciklą. Kaip buvo minėta ankščiau, nuotolinių procedūrų pranešimų kūrimas ir siuntimas gali būti atliekamas panaudojant dvi strategijas. Pirmoji strategija neatsižvelgia į pranešimų priklausomybes, todėl kiekvienas pranešimas siunčiamas atskirai. Šios strategijos supaprastintas algoritmas pavaizduotas pav. 12.



pav. 12 Supaprastintas įvesčių kūrimo ir siuntimo ciklo, kuris neatsižvelgia į pranešimų priklausomybes, algoritmas

Nuotolinių procedūrų pranešimų kūrimo ir siuntimo ciklas, kuris neatsižvelgia į pranešimų priklausomybes prasideda nuo pranešimo išrinkimo iš sąrašo. Tada yra atliekamas pranešimo reikšmių pakeitimas ir siuntimas testuojamai programai. Gavus atsaką iš programos yra gaunami ir tikrinami programos vykdymo eigos duomenys. Jei išsiųstas pranešimas sugebėjo priversti testuojamą programą vykdyti ankščiau nefiksuotas instrukcijas, pranešimas yra įtraukiamas į pranešimų sąrašo pradžią su padidintu numatomu testavimo įvesčių keitimų ciklų skaičiumi. Jei po pranešimo išsiuntimo nebuvo įvykdytos naujos instrukcijos, pranešimo KCS yra sumažinamas vienetu. Jei pranešimo KCS yra lygus nuliui, ciklas parenka kitą pranešimą sąraše. Tai vyksta kol sąraše yra pranešimų su KCS didesniu už nulį.

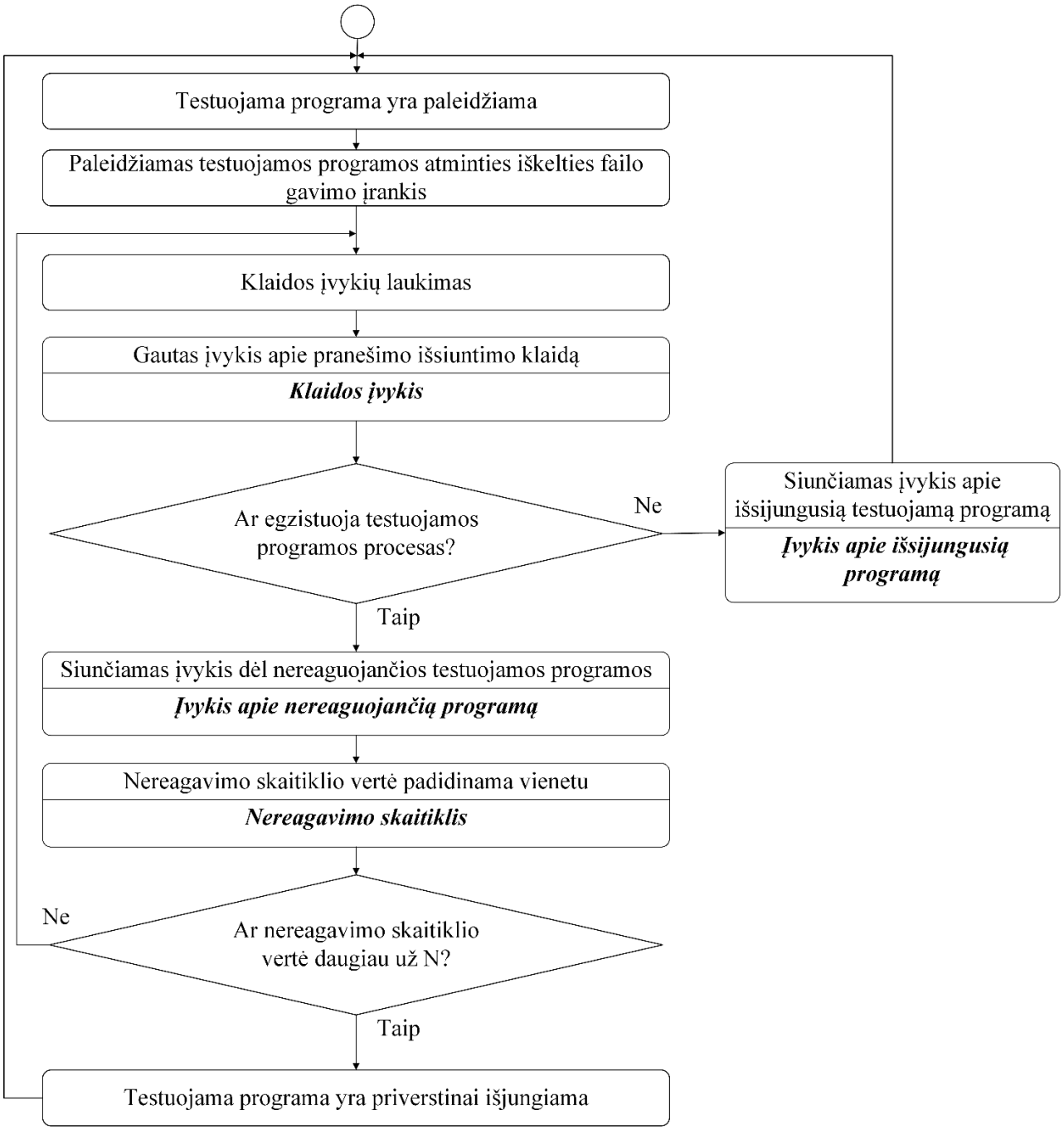
Nuotolinių procedūrų pranešimų kūrimo ir siuntimo strategija, kuri atsižvelgia į pranešimų priklausomybes. Antroji nuotolinių procedūrų pranešimų kūrimo ir siuntimo strategija kiekvienam pranešimui ieško susijusių pranešimų, kurie turi būti išsiųsti prieš sukurtą pranešimą. Šios strategijos supaprastintas algoritmas pavaizduotas pav. 13. Algoritmas prasideda nuo pranešimų priklausomybių ryšių radimo. Pagal šiuos ryšius yra sudaromos pranešimų grandinės. Pranešimų grandinę galima pavaizduoti kaip pranešimų sąrašą $\{p_1', \dots, p_n'\}$, kuriame pranešimai turi būti išsiųsti vienas po kito. Visos šios galimos grandinės yra sudedamos į sąrašą. Toliau yra apskaičiuojamos numatomos pranešimų grandinių testavimo įvesčių keitimų ciklų skaičių vertės. Skaičiavimo principas išlieka toks pat, kaip ir pirmosios įvesčių kūrimo ir siuntimo strategijoje, tačiau šiuo atveju, pranešimų grandinės KCS yra lygus paskutinio grandinės pranešimo KCS. Cikle keičiamas ir siunčiamas tik paskutinis grandinės pranešimas, o pirmieji grandinės pranešimai siunčiami tik vieną kartą, prieš ciklo pradžią. Apskaičiavus grandinių KCS vertes yra išrenkama pirmoji grandinė ir išsiunčiami pirmieji jos pranešimai. Kiekvieno pranešimo atsakymo reikšmės yra panaudojamos kito pranešimo įvestims. Išsiuntus pradinius grandinės pranešimus ir išsaugojus priešpaskutinio pranešimo atsakymo reikšmes, yra pradedamas paskutinio grandinės pranešimo keitimo ir siuntimo ciklas. Toliau atliekamas grandinės paskutinio pranešimo reikšmių pakeitimas ir siuntimas testuojamai programai. Gavus atsaką iš programos yra gaunami ir tikrinami programos vykdymo eigos duomenys. Jei išsiųstas pranešimas testuojamą programą privertė vykdyti ankščiau nefiksuotas instrukcijas, grandinė su šiuo pakeistu pranešimu yra įtraukiama į pranešimų grandinių sąrašo pradžią. Šios grandinės KCS padidinamas vienetu. Jei po pranešimo išsiuntimo nebuvo įvykdytos naujos instrukcijos, grandinės KCS yra sumažinamas vienetu ir pradedama kita keitimo ir siuntimo iteracija. Jei KCS yra lygus nuliui, ciklas parenka kitą grandinę sąraše. Tai vyksta kol sąraše yra grandinių su KCS didesniu nei nulis.



pav. 13 Supaprastintas įvesčių kūrimo ir siuntimo ciklo, kuris atsižvelgia į pranešimų priklausomybes, algoritmas

2.3.4. Programų veiklos stebėjimas

Testuojamos programos veiklos stebėjimas yra pradedamas metodo pradžioje ir yra vykdomas visą testavimo laikotarpį. Šis procesas yra atsakingas už testuojamos programos veikimo stebėjimą, paleidimą ir išjungimą, taip pat užtikrina, kad panaudojant išorinius įrankius bus sukurtas testuojamos programos atminties atvaizdo failas. Supaprastintas testuojamų programų veiklos stebėjimo algoritmas pavaizduotas pav. 14.



pav. 14 Supaprastintas programų veiklos stebėjimo algoritmas

Testuojamos programos veiklos stebėjimas prasideda nuo klaidos įvykio, kuris yra sukuriamas kai nepavyksta išsiųsti pranešimo testuojamai programai. Komunikacija gali sutrikti dėl kelių priežasčių:

- Testuojama programa nutraukė savo veikimą;

- Testuojama programa nenutraukė veikimo, tačiau neatsako į siunčiamus pranešimus.

Testuojamos programos veiklos nutraukimas yra aptinkamas panaudojant operacinės sistemos programų kūrimo sąsajas. *Windows* operacinėje sistemoje yra naudojama *GetExitCodeProcess* funkcija, kuri gražina programos išėjimo kodą. Galimos dvi išeitys:

- Jei kodas atitinka *STILL_ACTIVE* reikšmę, testuojama programa vis dar veikia;
- Jei kodas atitinka bet kokią kitą reikšmę, tai reiškia, kad programa nutraukė veikimą.

Jei testuojama programa nutraukė veikimą, sukuriamas įvykis apie testuojamos programos veiklos nutraukimą. Tada metodas surenka visą susijusią apie veiklos nutraukimą, galiausiai testuojama programa yra paleidžiama iš naujo. Programos paleidimas yra atliekamas pasinaudojant operacinės sistemos programų kūrimo sąsaja. *Windows* OS galima naudoti *CreateProcess* funkciją, kuri sukuria naują procesą. Paleidus testuojamą programą kartu yra paleidžiamas išorinis įrankis, kuris programos klaidos atveju sukuria proceso atminties atvaizdo failą. *Windows* OS atveju yra naudojamas *Microsoft ProcDump* [34] įrankis.

Tuo tarpu testuojamos programos nereagavimas į užklausas yra nustatomas pagal šias sąlygas:

- Programos procesas egzistuoja;
- Gauta klaida apie testuojamos programos duomenų siuntimo/gavimo problemą.

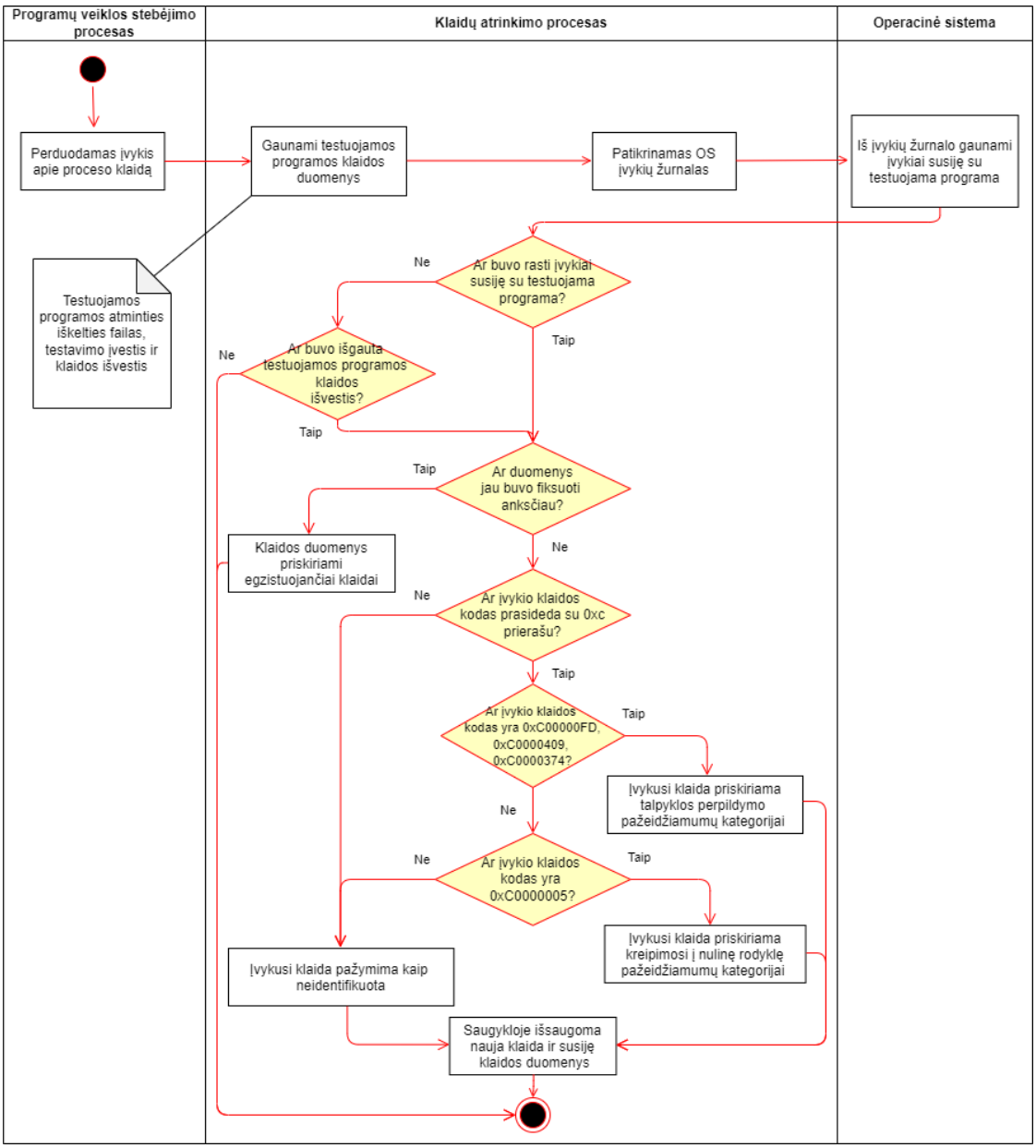
Testuojamos programos veikimas gali būti patikrintas naudojant aukščiau aprašytą OS programų kūrimo sąsają. Jei testuojamos programos procesas egzistuoja, bet neatsakoma į išsiųstas užklausas, yra sukuriamas įvykis apie ne atsakymą į pranešimus ir vienetu padidinama ne atsakymo skaitiklio vertė. Jei kitą kartą ši skaitiklio vertė neviršys N reikšmės, testavimo procesas bus tęsiamas. Jei skaitiklio vertė viršys N vertę, testuojama programa bus išjungiama ir įjungiama iš naujo.

Proceso veikla gali būti nutraukiama pasinaudojant operacinės sistemos programų kūrimo sąsają. *Windows* OS galima naudoti *TerminateProcess* funkciją, kuri nutraukia programos proceso veikimą. Tai įvykdžius testuojama programa yra paleidžiama iš naujo.

2.3.5. Testuojamos programos klaidų atrinkimas

Siekiant padėti vartotojams rasti potencialias klaidas testuojamoje programoje, siūlomas metodas suteikia klaidų atrinkimo funkcionalumą. Detali veikimo schema pateikta pav. 15. Kai iš 2.3.4 skyrelyje aprašyto programų veiklos stebėjimo algoritmo yra gaunamas įvykis apie testuojamos programos veiklos sutrikimą, klaidų atrinkimo procesas perima testuojamos programos klaidos duomenis iš programų veiklos stebėjimo proceso:

- Klaidos išvestį (angl. *stack trace*);
- Atminties atvaizdo failą;
- Testavimo įvestį.



pav. 15 Klaidų atrinkimo proceso veiklos diagrama

Tada klaidų atrinkimo procesas papildomai kreipiasi į operacinės sistemos įvykių žurnalą siekiant gauti bet kokią informaciją apie testuojamos programos sutrikimą, kurią užfiksavo OS. Priklausomai nuo operacinės sistemos ir programavimo kalbos, kuria buvo sukurta testuojama programinė įranga, veikimo sutrikimai yra fiksuojami skirtingai. Windows operacinėje sistemoje netikėti veiklos sutrikimai yra fiksuojami *Event Log* įvykių žurnale. Pavyzdinis veiklos sutrikimo įvykis atrodo taip:

```
Faulting application name: program.exe, version: 0.0.0.0, time stamp: 0x60a20ae4
Faulting module name: ucrtbased.dll, version: 10.0.18362.1, time stamp: 0x31dcf470
Exception code: 0xc0000005
Fault offset: 0x000e48b7
Faulting process id: 0x1834
Faulting application start time: 0x01d74ae50d0fe301
Faulting application path: C:\program.exe
Faulting module path: C:\Windows\SYSTEM32\ucrtbased.dll
Report Id: 4ff0f674-d1bb-4c86-b2d5-c6116250c428
Faulting package full name:
```

Įvykyje yra nurodomas programos vykdomojo failo pavadinimas, modulis, kuris sukėlė veiklos sutrikimą ir kiti duomenys. Svarbiausi parametrai yra klaidos kodas (angl. *exception code*), klaidą sukėlusio modulio pavadinimas (angl. *faulting module name*) ir klaidos vietos adreso postūmis (angl. *fault offset*). Tai yra minimali informaciją, kurią galima gauti *Windows* OS. Pagal aukščiau pateiktą pavyzdį galima išgauti informaciją, kad *program.exe* nutraukė veikimą dėl atminties prieigos pažeidimo, kurį atitinka klaidos kodas 0xc0000005. Šią klaidą sukėlė funkcija esanti *ucrtbased.dll* bibliotekoje, pasiekiama reliatyviuoju adresu 0x000e48b7.

Klaidų atrinkimo procesas pagal pateiktą klaidos išvestį ir įvykių žurnalo įrašus gali atpažinti klaidos tipą ir apytikslę priežastį. Yra išskiriamos trys sąlygos:

- Jei Windows OS klaidos kodas prasideda su 0xc prierašu, tai reiškia, kad įvyko atminties prieigos pažeidimas. Pagal tai aptinkami galimi kreipimosi į nulinę rodyklę ir talpyklos perpildymo pažeidžiamumai;
    - Jei Windows OS klaidos kodas yra 0xC00000FD, 0xC0000409, 0xC0000374, tai reiškia galimą talpyklos perpildymo pažeidžiamumą;
    - Jei Windows OS klaidos kodas yra 0xC0000005, tai reiškia galimą kreipimosi į nulinę rodyklę pažeidžiamumą. Kadangi tokiam atvejui *Windows* OS neturi specifinio klaidos kodo, prieiga prie netinkamo atminties adreso yra traktuojama kaip galimas šios rūšies pažeidžiamumas.
- Jei programa nutraukė veikimą, tačiau klaidos kodas neindikuoja atminties prieigos pažeidimo, tai yra pažymima kaip neidentifikuotas programos veiklos sutrikdymas;
- Jei testuojama programa nutraukė veikimą ir nebuvo rastas susijęs klaidos įvykis, tai yra klasifikuojama kaip savaiminis sutrikimas.

Įvykus veiklos sutrikimui, kuris atitinka vieną iš aukščiau nurodytų sąlygų, klaidų atrinkimo procesas kreipiasi į duomenų saugyklą siekiant patikrinti ar egzistuoja jau užfiksuota klaida. Panašumo tikrinimas yra vykdomas pagal šias reikšmes:

- Klaidos kodą;
- Klaidą sukėlusio modulio pavadinimą;
- Klaidą sukėlusios funkcijos reliatyvų adresą.

Jei buvo identifikuota, kad esama klaida buvo užfiksuota, esami tarp-procesinės mainų srities pranešimai yra pridedami prie užfiksuotos klaidos duomenų saugykloje ir procesas yra užbaigiamas. Jei testavimo metu šis veiklos sutrikimas nebuvo užfiksuotas, klaidų atrinkimo procesas į duomenų saugyklą įrašo informaciją apie įvykusią klaidą ir testavimo įvestis, kurios galimai išprovokavo šią klaidą. Įrašomą informaciją sudaro:

- Klaidos kodas;
- Nustatyta klaidos priežastis;
- Klaidą sukėlusio modulio pavadinimas ir vieta failų sistemoje;
- Klaidą sukėlusios funkcijos reliatyvus adresas;
- Testuojamos programos proceso atminties atvaizdo failo vieta failų sistemoje;
- Nuotolinių procedūrų pranešimas, kuris išprovokavo šią klaidą;
- Klaidos išvestis (jei egzistuoja);
- Susiję OS įvykių žurnalo įrašai (jei egzistuoja).

Klaidų atrinkimo procesas yra užbaigiamas įrašius aukščiau paminėtą informaciją į duomenų saugyklą.

2.4. Išvados

1. Pasiūlytas programų pažeidžiamumų aptikimo metodas naudojant *gRPC* nuotolinį procedūrų iškvietimo karkasą;
2. Pasiūlytas metodas leidžia stebėti testuojamos programinės įrangos vykdymą ir veikimą, atlikti nuotolinių procedūrų pranešimų keitimą pagal testuojamos programos esamą būseną ir išsiųsti naujai sukurtus pranešimus;
3. Metodo pagrindinis išskirtinumas – esamos testuojamos programinės įrangos nuotolinių procedūrų pranešimų išnaudojimas siekiant sumažinti testavimo metu siunčiamų atmestų pranešimų procentinę dalį ir nuotolinių procedūrų pranešimų manipuliavimas realiu laiku, priklausomai nuo testuojamos programos įvykdytų instrukcijų;
4. Remiantis pasiūlytu programų pažeidžiamumų aptikimo metodu naudojant *gRPC* nuotolinį procedūrų iškvietimo karkasą, bus realizuotas prototipas skirtas eksperimentiniam tyrimui.

3. Programų pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą prototipas

Šiame skyriuje aprašomas programų pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą, prototipas. Pateikiamas naudojamų technologijų ir įrankių aprašas, prototipo paketų, duomenų srautų ir diegimo diagramos.

3.1. Eksperimentinio tyrimo įrankiai ir technologijos

Siekiant automatizuoti metodo algoritmą buvo sukurtas prototipas pasinaudojant *Go* ir *JavaScript* programavimo kalbomis. *Go* kalba pasirinkta dėl galimybės ateityje prototipą naudoti ir kitose platformose. *JavaScript* kalba naudojama tik scenarijaus kūrimui, kuris yra įkeliamas į testuojamą programinę įrangą. Prototipas bus kviečiamas iš terminalo sąsajos, todėl neturi įprastos grafinės sąsajos. Programavimui buvo naudota *Visual Studio Code* programinė įranga su įdiegtu *Go* 1.16.9 programų kūrimo rinkiniu.

3.2. Prototipo sandara

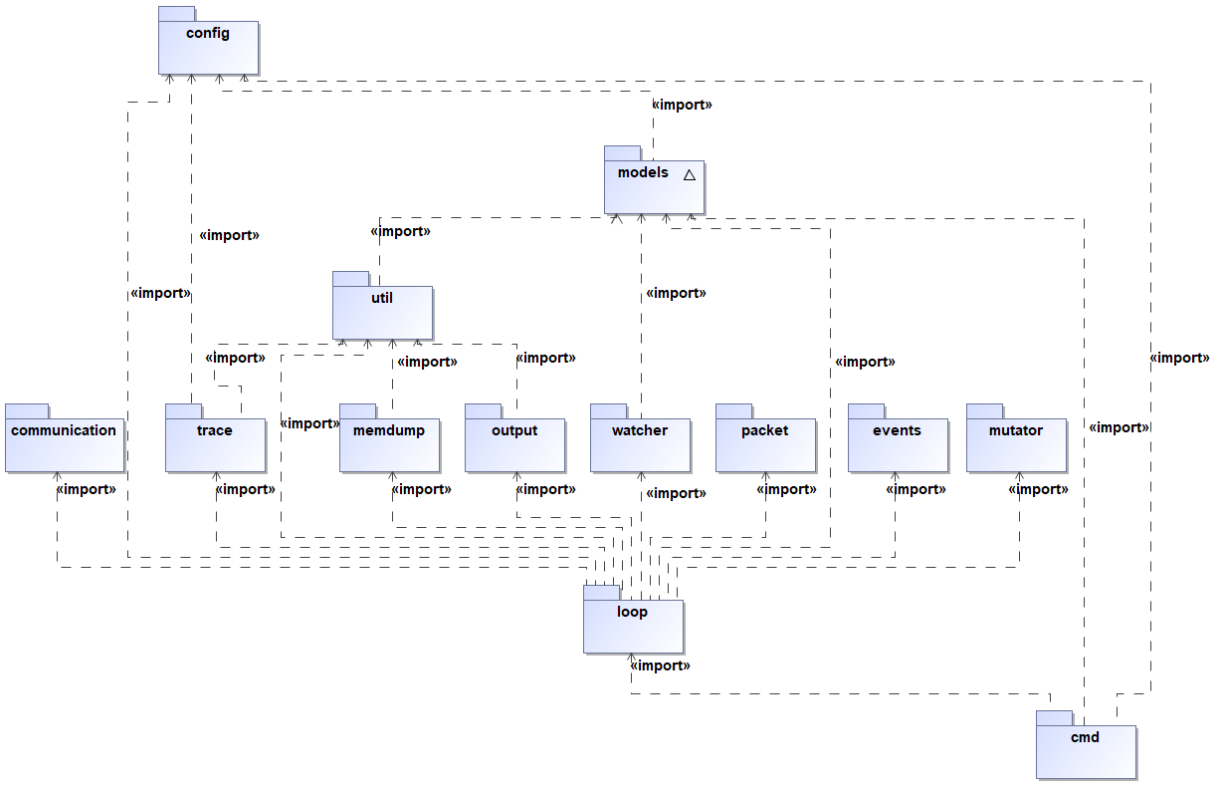
Prototipas yra sudarytas iš trijų dalių:

- Pagrindinio prototipo kodo – kodas atsakingas už *Protobuf* failų nuskaitymą, interpretavimą, *gRPC* nuotolinių procedūrų pranešimų kūrimą, siuntimą ir sąsajų nustatymą, testuojamos programos veiklos stebėjimą ir vykdymo duomenų apdorojimą, klaidų atrinkimą;
- *Frida* dinaminės analizės įrankio bibliotekos ir programavimo sąsajos – pasinaudojus šia biblioteka yra atliekamas testuojamos programos vykdymo eigos stebėjimas. Visi duomenys yra naudojami prototipe įgyvendintiems algoritmams;
- Pagalbiniai įrankiai – *SysInternals ProcDump* įrankis skirtas gauti testuojamos programos proceso atminties atvaizdo failą.

Pagrindinį prototipą sudaro 14 paketų, kurie atsakingi už tam tikrą prototipo funkcionalumą. Paketų diagrama yra pavaizduota pav. 16. Toliau aprašomi nurodytų paketų funkcionalumai:

- *cmd* paketas – prototipo pradinis veikimo taškas, kuriame nuskaitomas nustatymų failas ir paleidžiamas pagrindinis algoritmas;
- *loop* paketas – vieta, kur įgyvendintas pagrindinis prototipo algoritmas ir iškviečiamos reikalingos algoritmo pagalbinės funkcijos;
- *mutator* paketas – naudojamas pakeistų nuotolinių procedūrų pranešimų kūrimui;
- *watcher* paketas – skirtas paleisti testuojamą programą ir stebėti programos veikimo būseną, veiklos sutrikimo atveju išgauti klaidos tekstą (angl. *stack trace*);
- *events* paketas – naudojamas išgauti įvykius apie testuojamos programos triktį iš OS įvykių žurnalo;
- *communication* paketas – skirtas siųsti ir gauti *gRPC* nuotolines procedūras;
- *models* paketas – naudojamas prototipe naudojamų duomenų struktūrų laikymui;
- *util* paketas – skirtas laikyti dažniausiai naudojamas pagalbines funkcijas;
- *memdump* paketas – naudojamas iškviesti išorinį įrankį, kuris išgauna testuojamos programos proceso atminties atvaizdo failą. Prototipo atveju, yra naudojamas *SysInternals ProcDump* įrankis;

- *output* paketas – atsakingas už funkcijas, kurios atlieka testuojamos programos veikimo sutrikimo duomenų įrašymą į failą. Duomenys faile yra pateikiami JSON formatu;
- *packet* paketas – skirtas funkcijoms, kurios nuskaito ir apdoroja *pcapng* ir *pcap* failus, nuskaito *gRPC* nuotolinių procedūrų paketus ir jų duomenis. Taip pat šiame pakete yra laikomos funkcijos naudojamos *Protobuf* sąsajų aprašymo kalbos failų nuskaitymui ir apdorojimui. Galiausiai čia yra įgyvendintas funkcionalumas atsakingas už nuotolinių procedūrų pranešimų ir pranešimų laukų sąsajų atpažinimą;
- *trace* paketas – naudojamas įgyvendinti testuojamos programos vykdymo eigos duomenų surinkimą. Prototipe yra naudojama *Frida* dinaminės analizės biblioteka ir jos programų kūrimo sąsaja;
- *frida-go* paketas – skirtas pasinaudoti *Frida* bibliotekos suteiktomis funkcijomis;
- *config* paketas – atsakingas už nustatymų failo nuskaitymą ir nustatymų apdorojimą.



pav. 16 Prototipo paketų diagrama

3.3. Pradinių duomenų įvestis ir apdorojimas

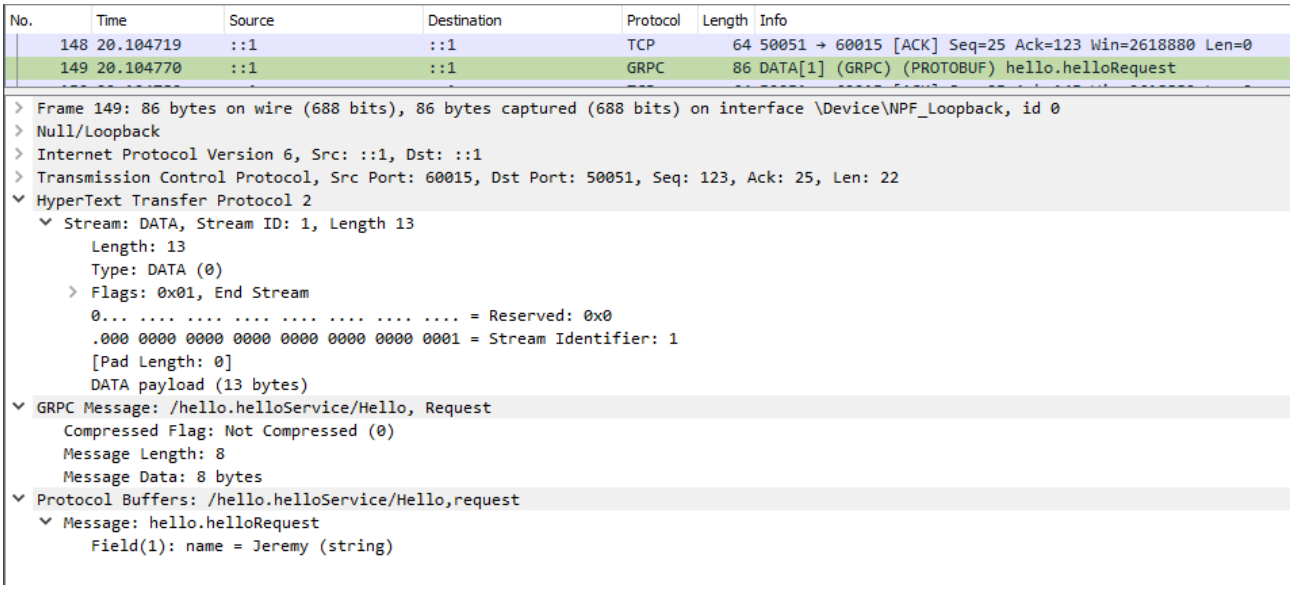
Prototipas, prieš pradėdamas vykdyti pagrindinį algoritmą, turi gauti reikalingus nustatymus. Programa nuskaito nustatymus iš JSON formato failo. Nustatymai detaliau aprašyti lentelė 3.

lentelė 3 Prototipe naudojami nustatymai ir jų paskirtis

Nustatymo pavadinimas	Paskirtis
pathToExecutable	Nurodomas kelias iki testuojamos programos vykdymo failo failų sistemoje.
executableArgs	Masyvas, kuriame nurodomi parametrai su kuriais turi būti paleidžiama testuojama programa.

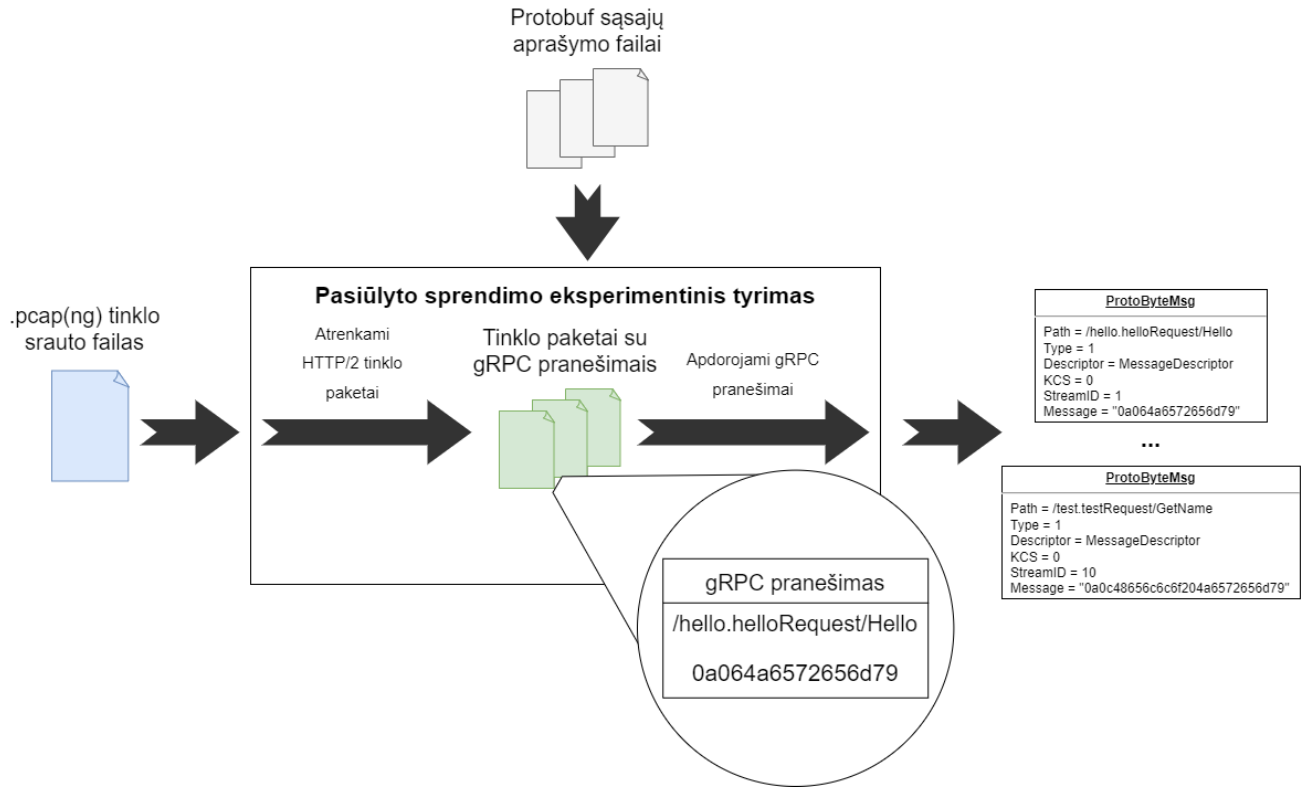
outputPath	Nurodo išvesties failų įrašymo vietą failų sistemoje.
dumpExecutablePath	Nurodomas kelias iki proceso atminties iškelties failo kūrimo programos vietos failų sistemoje.
performMemoryDump	Nusako ar reikia atlikti testuojamos programos atminties atvaizdo kūrimą.
handlers	Masyvas, kuriame yra aprašomos testuojamos programos funkcijos, kurioms bus atliekamas vykdymo eigos stebėjimas. Kiekvienas masyvo elementas turi tris laukus: *method*, *module* ir *handler*. *Method* nurodo *gRPC* nuotolinės paslaugos funkcijos adresą. *Module* nurodo testuojamos programos modulio pavadinimą, kuriame bus atliekamas vykdymo eigos stebėjimas. *Handler* nurodo funkcijos, kurios stebėjimas bus atliekamas, pavadinimą arba jos reliatyvų adresą vykdomajame faile.
host	Nurodomas testuojamos programos *gRPC* paslaugos serverio adreso pavadinimas.
port	Nurodomas testuojamos programos *gRPC* paslaugos prievadas.
ssl	Nurodo ar *gRPC* nuotolinių procedūrų siuntimui ir gavimui bus naudojamas SSL protokolas.
performDryRun	Nusako ar prieš testavimą reikia atlikti bandomąjį *gRPC* nuotolinės procedūros siuntimą. Taip patikrinama ar testuojama programa veikia tinkamai.
singleFieldMutation	Nusako ar bus taikoma vieno lauko nuotolinių procedūrų pranešimų kūrimo strategija.
dependencyUnawareSending	Nusako ar bus taikoma pranešimų grandinių siuntimo strategija.
protoFilesPath	Nurodomas aplankalas failų sistemoje kur yra laikomi testuojamos programos *Protobuf* sąsajų aprašymo kalbos failai.
protoFilesIncludePath	Masyvas, kuriame nurodomi aplankalai failų sistemoje, kuriuose yra papildomi testuojamos programos *Protobuf* sąsajų aprašymo kalbos failai.
pcapFilePath	Nurodoma užfiksuotų testuojamos programos komunikacijos tinklo paketų įrašo failo vieta.
maxMsgSize	Nurodomas maksimalus *gRPC* pranešimo dydis, kurį gali priimti testuojamas procesas.
useInstrumentation	Nurodoma ar testavimo metu bus atliekamas programos vykdymo eigos stebėjimas ir susijusių duomenų surinkimas.

Nuskaičius ir apdorojus nustatymų failą, yra atliekamas užfiksuotų testuojamos programos komunikacijos tinklo paketų nuskaitymas ir apdorojimas. Pavyzdinis tinklo paketas ir *gRPC* nuotolinės procedūros duomenys pavaizduoti pav. 17.



pav. 17 Pavyzdinis tinklo paketas ir *gRPC* nuotolinės procedūros duomenys *Wireshark* tinklo paketų analizavimo įrankyje

Kaip galima matyti iš aukščiau pateikto *Wireshark* įrankio paveikslo, iš HTTP/2 paketo yra išrenkamas */hello.helloService/Hello gRPC* nuotolinė procedūra. Iš šios procedūros yra gaunami duomenys, šio pavyzdžio atveju procedūroje yra vienintelis laukas name ir jo reikšmė yra *Jeremy*. Prototipe šis funkcionalumas yra įgyvendintas panaudojant *gopacket*, *http2* ir *protoreflect* bibliotekas. Funkcionalumo proceso diagrama pateikta pav. 18. *Gopacket* biblioteka atlieka paketų nuskaitymą iš pateiktų *pcap* ir *pacapng* failų. *Http2* biblioteka reikalinga HTTP/2 paketų apdorojimui ir interpretavimui. Galiausiai *protoreflect* biblioteka atlieka *gRPC* nuotolinėse procedūrose esančių duomenų interpretavimą. Apdorotų pranešimų duomenys yra išsaugomi *ProtoByteMsg* vidinėje duomenų struktūroje. Ši struktūra turi adreso lauką, požymį ar tai užklausa ar rezultatas, nuorodą į *Protobuf* sąsajų aprašymo failą, procedūros transliacijos (angl. *stream*) identifikatorių ir patį procedūros turinį, kuris yra laikomas šešioliktainiu *Wire* formatu. KCS laukas yra užpildomas vėliau.



pav. 18 Tinklo srautų paketų apdorojimo ir interpretavimo procesas

3.4. Sąsajų tarp pranešimų ir pranešimų laukų nustatymas

Atlikus nuotolinių procedūrų pranešimų nuskaitymą ir apdorojimą yra įgyvendinamas pranešimų ir pranešimų laukų sąsajų nustatymas. Šie veiksmai yra atliekami pagal 2.3.2 skyrelyje aprašytą algoritmą. Pranešimų priklausomybės nustatymo rezultatas yra tikimybių matrica, kurios dydis yra N stulpelių ir N eilučių (N yra apdorotų pranešimų skaičius). Pavyzdinis rezultatas yra pateiktas žemiau esančioje matricoje.

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0.66 & 0.33 \\ 0 & 0 & 0 \end{bmatrix} \quad (1)$$

Šioje matricoje yra vaizduojamos trijų pranešimų priklausomybių tikimybės. Matricos reikšmės yra normalizuotos, vienetas reiškia, kad pranešimas p_i turi būti siunčiamas prieš p_j pranešimą. Taip pat yra atliekamas pranešimų reikšmių priklausomybių radimas.

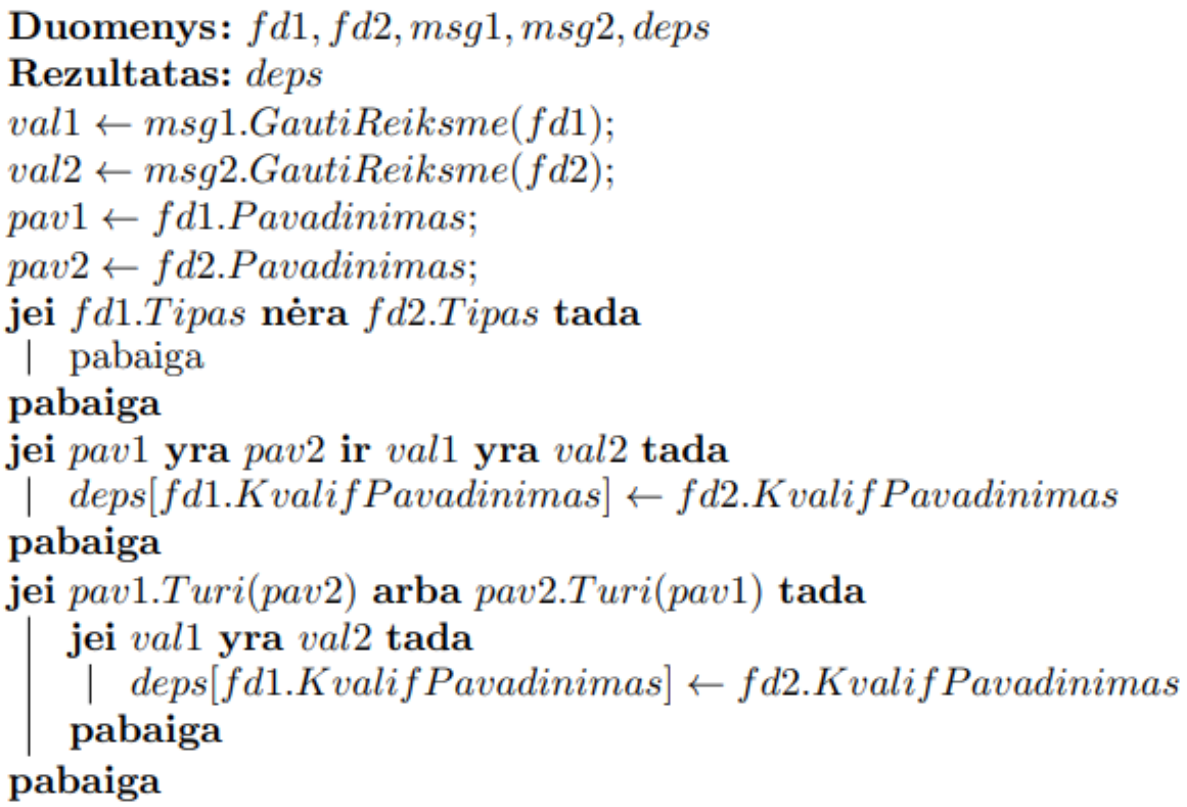
Duomenys: $msg1, msg2, pbMsg1, pbMsg2, mFd1, mFd2$
Rezultatas: $deps$
$deps \leftarrow [][]$;
kiekvienam $i = 0$ **iki** $len(mFd1)$ **atlikti**
  kiekvienam $j = 0$ **iki** $len(mFd2)$ **atlikti**
    jei $mFd1[i]$ **yra** *sarašas* **ir** $mFd2[j]$ **nėra** *sarašas* **tada**
      $deps \leftarrow$ $getRelationshipsFromRepMessages(fd1, fd2, pbMsg1, pbMsg2)$
    pabaiga
    jei $mFd2[j]$ **yra** *sarašas* **ir** $mFd1[i]$ **nėra** *sarašas* **tada**
      $deps \leftarrow$ $getRelationshipsFromRepMessages(fd2, fd1, pbMsg2, pbMsg1)$
    pabaiga
    jei $mFd1[i]$ **yra** $PRANEŠIMAS$ **ir** $mFd1[i]$ **nėra** *sarašas* **tada**
      $deps \leftarrow$ $getRelationshipsFromMessages(fd1, fd2, pbMsg1, pbMsg2)$
    pabaiga
    jei $mFd2[j]$ **yra** $PRANEŠIMAS$ **ir** $mFd2[j]$ **nėra** *sarašas* **tada**
      $deps \leftarrow$ $getRelationshipsFromMessages(fd2, fd1, pbMsg2, pbMsg1)$
    pabaiga
    $deps \leftarrow$ $getRelationshipsFromFields(fd1, fd2, pbMsg1, pbMsg2)$
  pabaiga
pabaiga

pav. 19 Pagrindinis pranešimų reikšmių priklausomybių nustatymo algoritmas

pav. 19 galima matyti reikšmių priklausomybių nustatymo pagrindinį algoritmą. Į algoritmą yra įkeliami dviejų tikrinamų pranešimų duomenų objektai (*msg1, msg2, pbMsg1, pbMsg2*), pranešimų laukų masyvai (*mFd1* ir *mFd2*). Gaunamas rezultatas yra asociatyvus masyvas, kuriame yra nurodoma, kuris pirmo pranešimo laukas yra priklausomas nuo antrojo pranešimo lauko. Algoritme yra tikrinami visi pirmojo ir antrojo pranešimo laukai. Kiekviena laukų pora yra tikrinama pagal tam tikras sąlygas. Jas tenkinant yra iškviečiamas atitinkamas priklausomybių nustatymo metodas:

- Jeigu vienas iš pranešimų poros laukas yra sąrašo tipo (*Protobuf* sąsajų aprašymo kalbos faile laukas yra žymimas su prierašu *repeated*), o kitas poros pranešimo laukas nėra sąrašo tipo – kviečiamas *getRelationshipsFromRepMessages()* metodas;
- Jeigu vienas iš pranešimų poros laukas yra įterptinis pranešimas (*Protobuf* sąsajų aprašymo kalbos faile laukas yra žymimas su pranešimo pavadinimo prierašu), o kitas poros pranešimo laukas nėra sąrašo tipo – kviečiamas *getRelationshipsFromMessages()* metodas;
- Jei nėra tenkinama nei viena iš aukščiau išvardintų sąlygų yra manoma, kad tai paprastų laukų pora – kviečiamas *getRelationshipsFromFields()* metodas.

Kiekvienas priklausomybių nustatymo metodas veikia pagal algoritmą, kuris yra pavaizduotas pav. 20.



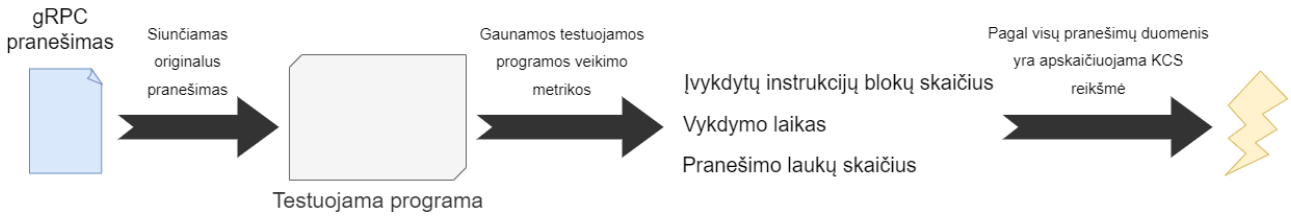
pav. 20 Pranešimų reikšmių priklausomybių nustatymo algoritmas

Aukščiau pateiktame paveiksle galima matyti reikšmių priklausomybių nustatymo pagrindinį algoritmą. Šis algoritmas, iš esmės, yra pritaikomas visuose priklausomybių nustatymo metoduose. Į algoritmą yra įkeliami dviejų tikrinamų pranešimų duomenų objektai (*msg1*, *msg2*) ir pranešimų laukų aprašų objektai (*fd1* ir *fd2*). Gaunamas asociatyvus masyvas, kuriame yra nurodoma, ar pirmo pranešimo laukas yra priklausomas nuo antrojo pranešimo lauko. Pirmiausia yra patikrinami laukų tipai. Jei tipai nesutampa, algoritmas yra nutraukiamas. Jei pranešimų laukų pavadinimai ir reikšmės sutampa, yra įrašoma nauja sąsaja tarp pranešimų laukų pavadinimų. Į pranešimų laukų pavadinimus įeina lauko pranešimo pavadinimas, pvz.: *pranešimas.Laukas*. Galiausiai, jei viename iš lyginamų pranešimų laukų pavadinimų yra dalis kito pranešimo pavadinimo ir lyginamų laukų reikšmės sutampa, yra įrašoma nauja pranešimų laukų pavadinimų sąsaja.

3.5. Numatomo nuotolinių procedūrų pranešimų testavimo įvesčių keitimų ciklų skaičiavimas

Atlikus pranešimų laukų priklausomybių nustatymą yra įvykdomas nuotolinių procedūrų pranešimų numatomo keitimų ciklų skaičiaus nustatymas. Šis skaičius rodo kiek bus atliekama vieno pranešimo keitinių. Šiam skaičiui pasiekus nulį, bus parenkamas kitas eilėje esantis pranešimas. Keitimų ciklų skaičius yra naudojamas pranešimų keitimo ir siuntimo procese. Supaprastintas skaičiavimo procesas pavaizduotas pav. 21. Proceso metu yra siunčiami originalūs nuotolinių procedūrų pranešimai. Gavus atsakymą iš testuojamos programos yra surenkama testuojamos programos vykdymo metrika. Metriką sudaro:

- Įvykdytų instrukcijų blokų skaičius;
- Vykdymo laikas (milisekundėmis);
- Pranešimo laukų skaičius.

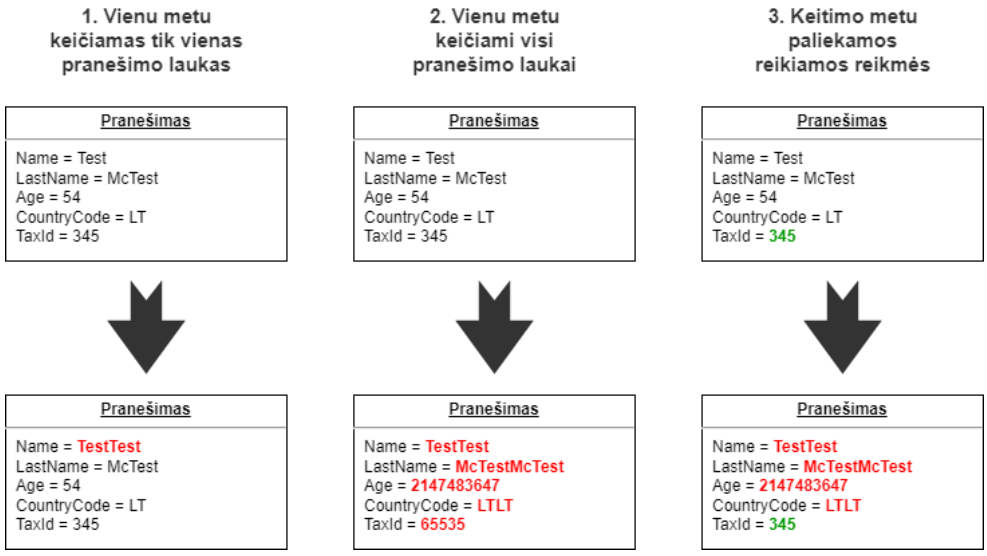


pav. 21 Supaprastintas nuotolinių procedūrų pranešimų pradinės energijos skaičiavimo procesas

Šios veikimo metrikos yra išsaugomos atskiruose masyvuose. Tada šis veikimo metrikų rinkimo žingsnis yra kartojamas su visais kitais nuotolinių procedūrų pranešimais. Gavus visų nuotolinių procedūrų pranešimų metrikas yra atliekamas duomenų normalizavimas. Įvykdytų instrukcijų blokų ir procedūrų pranešimų metrikos yra normalizuojamos į intervalą nuo 1 iki 10. Vykdymo laikai yra normalizuojami tame pačiame intervale, tik atvirkščiai, kadangi norima suteikti pirmumo teisę pranešimams, kurie trunka mažiausiai laiko. Normalizavimo reikšmių intervalas pasirinktas dėl bendro keitimų ciklo skaičiaus dydžio, kadangi didesni rėžiai reikštų didesnius ciklų skaičius ir prailgintų testavimo trukmę. Kiekvieno pranešimo normalizuotos metrikos yra susumuojamos ir yra gaunamas pranešimo keitimų ciklo skaičius. Galiausiai nuotolinių pranešimų eilė yra perrūšiuojama pagal pranešimo keitimų ciklo skaičių mažėjimo tvarka.

3.6. Nuotolinių procedūrų pranešimų kūrimas atliekant keitinius

Atlikus nuotolinių procedūrų pranešimų numatomo keitimų ciklų skaičiaus nustatymą, pradedamas naujų pranešimų kūrimas atliekant keitinius esamame pranešime. Kaip galima matyti pav. 22, egzistuoja dvi pagrindinės pranešimų kūrimo strategijos: vienoje iteracijoje yra atliekamas vieno lauko arba visų laukų reikšmių pakeitimas. Taip pat yra išskiriamas ir kitas atvejis, kai tam tikra pranešimo lauko reikšmė yra konkreti ir negali būti keičiama. Tuo atveju gali būti taikomos abi ankstesnės strategijos tik reikiamuose laukuose yra paliekamos konkrečios reikšmės. Pavyzdys yra pateiktas žemiau esančiame paveiksle. Norima pranešimų kūrimo strategija yra nurodoma nustatymuose. Pats pakeitimų procesas detaliau yra aprašytas 2.3.3 skyrelyje.



pav. 22 Nuotolinių procedūrų pranešimų kūrimo strategijos

Kalbant apie reikšmių pakeitimus, abiejų keitimo strategijų principas išlieka nepakitęs. Pranešimų laukų pakeitimai yra atliekami atsitiktinai pakeičiant reikšmes į tas, kurios įrašytos metodo prototipe. Šios reikšmės pateikiamos lentelė 4.

lentelė 4 Skaitinių laukų pakaitalai pagal reikšmių tipus

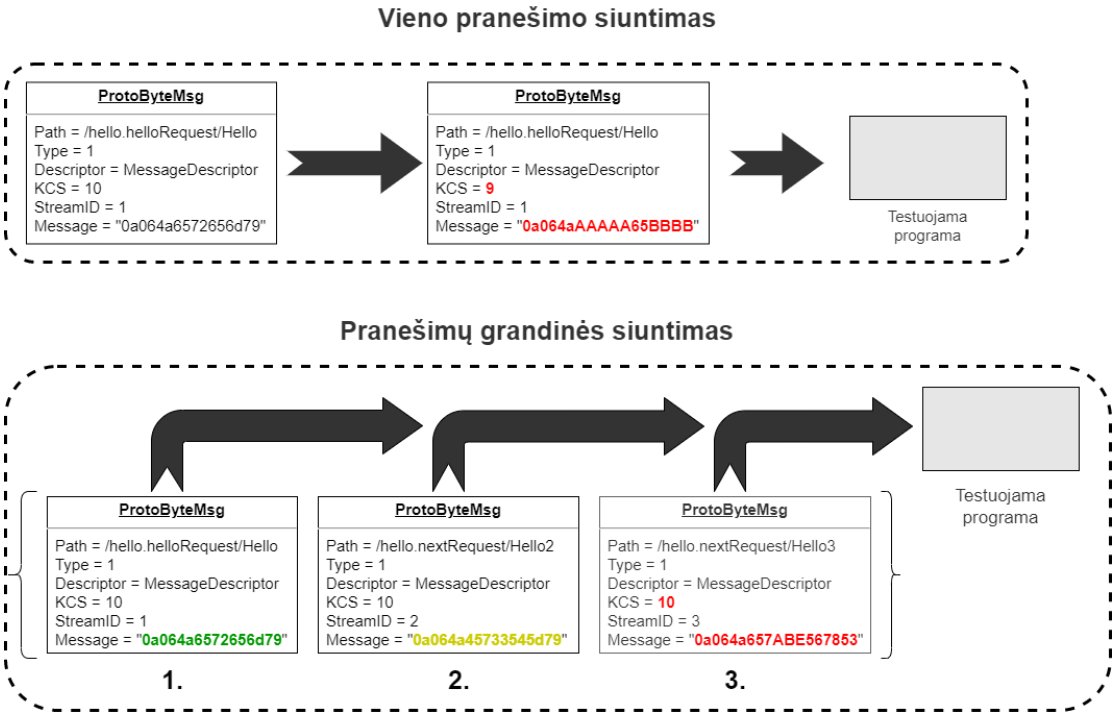
Protobuf sąsajų aprašymo kalbos pranešimo reikšmės tipas	Keičiamų reikšmių intervalas	
	Mažiausia reikšmė	Didžiausia reikšmė
string	Tuščia teksto eilutė	2^{32} ilgio teksto eilutė
bool	0	1
bytes	Tuščias baitų masyvas	2^{32} ilgio baitų masyvas
int32, sint32, sfixed32	-2147483648	2147483647
uint32, fixed32	0	4294967295
int64, sint64, sfixed64	-9223372036854775808	9223372036854775807
unit64, fixed64	0	18446744073709551615
float	1.175494351 E - 38	3.402823466 E + 38
double	2.2250738585072014 E - 308	1.7976931348623158 E + 308

Šios reikšmės yra įvairūs ekstremumai, kurios gali sukelti pažeidžiamumus testuojamose programose jei nėra tinkamai atliekamas įvesties patikrinimas. Šios reikšmės yra paimtos iš *Go* programavimo kalbos *math* bibliotekos. Teksto eilučių ir baitų tipo laukų pakeitimai atliekami tuo pačiu principu, didinant reikšmės ilgį iki nustatyto limito. Jei yra pasiekiamas limitas, lauko reikšmė yra išvaloma. Algoritmo pagrindinį principą galima pamatyti pav. 23.

Duomenys: fd, msg
Rezultatas: msg
$lim \leftarrow$ Atsitiktinis skaičius iki 10;
$curVal \leftarrow msg.GautiReiksme(fd)$;
$newVal \leftarrow pakartoti(curVal, lim)$;
jei $newVal.Ilgis > 2^{32}$ **tada**
| išvalyti keičiamo lauko reikšmę
pabaiga
pakeisti esamo lauko reikšmę į $newVal$

pav. 23 Bendrinis teksto eilučių ir baitų laukų keitimo algoritmas

Pakeisti pranešimai yra išsiunčiami testuojamai programai. Siuntimas gali būti atliekamas panaudojant dvi strategijas, kurios yra pavaizduotos pav. 24. Nuotolinių procedūrų pranešimai gali būti traktuojami kaip neturintys priklausomybių, todėl kiekvienas pranešimas būtų keičiamas ir siunčiamas atskirai, kol pranešimo KCS yra daugiau už nulį. Supaprastintas šios strategijos procesas pavaizduotas žemiau esančio paveikslo viršutinėje dalyje. Egzistuoja ir kita siuntimo strategija, kuri bus naudojama susijusiuose nuotolinių procedūrų pranešimuose. Jei nustatoma, kad pranešimas yra priklausomas nuo kitų pranešimų, yra sudaromos susijusių pranešimų grandinės. Tada yra išsiunčiami nepakeisti pradiniai grandinės pranešimai. Galiausiai yra išsiunčiamas pakeistas paskutinis grandinės pranešimas ir sumažinama jo KCS reikšmė. Šis pranešimas yra keičiamas kol KCS yra daugiau už nulį. Tada yra pereinama prie kitos pranešimų grandinės. Antroji pranešimų siuntimo strategija pavaizduota žemiau esančio paveikslo apatinėje dalyje. Detalesnis strategijų veikimas aprašytas 2.3.3 skyrelyje. Verta paminėti, kad norima pranešimų siuntimo strategija yra nurodoma prototipo nustatymų faile.

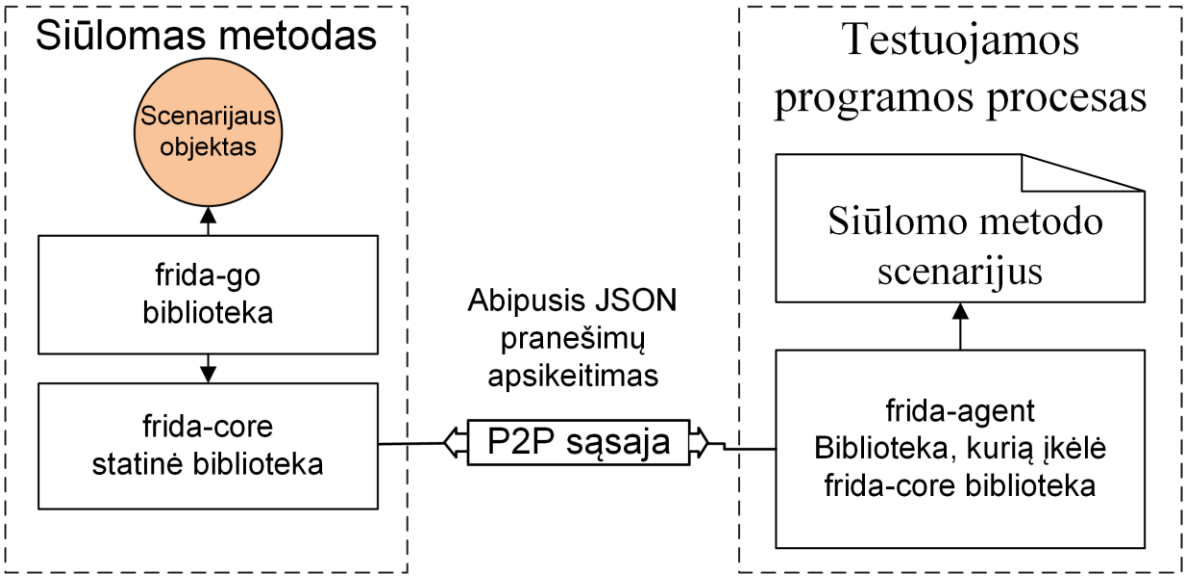


pav. 24 Nuotolinių procedūrų pranešimų siuntimo strategijos

3.7. Testuojamos programos vykdymo eigos stebėjimas

Testuojamos programos vykdymo eigos stebėjimui yra naudojama *Frida* dinaminės analizės biblioteka. Tam, kad būtų galima naudoti bibliotekos programų kūrimo sąsają *Go* programavimo kalboje, yra panaudota *frida-go* biblioteka, kuri leidžia iškviesti reikalingas *Frida* bibliotekos funkcijas. Įgalinus testuojamos programos vykdymo eigos stebėjimą, vykdymo metu *Frida* biblioteka įterpia papildomą kodą į testuojamą programą, kas leidžia vykdyti *JavaScript* kalbos scenarijus testuojamos programos kontekste. Šie scenarijai turi pilną prieigą prie testuojamos programos atminties taip pat geba pakeisti funkcijų vykdymo eigą. Prototipo atveju, kai testuojamoje programoje yra iškviečiama nuotolinė procedūra, scenarijus pradeda fiksuoti vykdomos procedūros instrukcijų blokus. Tai įvykdyta pasinaudojus *Frida* bibliotekos *Interceptor* ir *Stalker* programų kūrimo sąsajomis. Pabaigus testuojamos programos metodo vykdymą įvykdytų instrukcijų blokų duomenys yra persiunčiami prototipui. Be šių duomenų taip pat yra fiksuojama testuojamos programos

nuotolinės procedūros vykdymo trukmė. Duomenys iš testuojamos programos yra siunčiami *Frida* bibliotekos sukurtu komunikacijos kanalu, kuris leidžia apsikeisti duomenimis tarp prototipo ir testuojamoje programoje vykdomo scenarijaus kodo. Šis kanalas taip pat naudojamas iškviesti *JavaScript* scenarijaus metodus. Įgyvendinimo schema pavaizduota pav. 25.



pav. 25 Programos vykdymo eigos stebėjimo įgyvendinimas panaudojant *Frida* biblioteką

Kaip buvo minėta ankščiau, scenarijus yra įgyvendintas su *JavaScript* programavimo kalba. Scenarijuje yra šeši metodai, kuriuos kviečia prototipas. Metodai ir jų paskirtis pateikti lentelė 5.

lentelė 5 *JavaScript* kalbos scenarijuje įgyvendinti metodai

Metodas	Paskirtis
startCoverageFeed	Įgalina pasirinktos testuojamos programos nuotolinės procedūros vykdymo stebėjimą ir užfiksuoja iškvietimo laiką. Įvykdžius testuojamos programos nuotolinę procedūrą yra išsaugomi įvykdytų instrukcijų blokai ir apskaičiuojamas procedūros vykdymo laikas.
setTarget	Nustatoma testuojamos programos nuotolinė procedūra, kuriai bus atliekamas vykdymo eigos stebėjimas. Galima pateikti nuotolinės procedūros metodo pavadinimą arba jo reliatyvų adresą vykdomajame faile.
stopCoverageFeed	Baigiamas pasirinktos testuojamos programos nuotolinės procedūros vykdymo eigos stebėjimas.
getCoverage	Gaunami įvykdytų instrukcijų blokų duomenys.
getExecTime	Gaunamas nuotolinės procedūros vykdymo laikas (milisekundėmis).
clearCoverage	Ištrinami užfiksuoti įvykdytų instrukcijų blokai.

3.8. Programų pažeidžiamumų aptikimo metodo prototipo vartotojo sąsaja

Sukonfigūravus ir paleidus programą vartotojas gali matyti terminalo sąsają, kurioje yra atvaizduojami trys pagrindiniai blokai – testavimo proceso laiko ir trukmės informacija (*Timing*), bendrieji testavimo proceso rezultatai (*Overall results*) ir esamas testavimo proceso progresas (*Progress*). Detalus sąsajos vaizdas yra pavaizduotas pav. 26.

pav. 26 Pagrindinės informacijos apie testavimo eigą atvaizdavimas

Timing srityje pateikiama ši informacija:

- Testavimo proceso veikimo laikas (*Run time*);
- Laikas, kada paskutinį kartą testuojama programa įvykdė naujus instrukcijų blokus (*Last new path*);
- Laikas, kada paskutinį kartą testuojama programa nustojo veikti (*Last unique crash*);
- Laikas, kada paskutinį kartą testuojama programa neatsakė į užklausas (*Last unique hang*).

Overall results srityje pateikiama ši informacija:

- Dabartinės testavimo ciklo iteracijos numeris (*Current cycle*);
- Eilėje esančių nuotolinių procedūrų pranešimų skaičius (*Messages in queue*);
- Naujų instrukcijų blokų skaičius (*Total paths*);
- Unikalių testuojamos programos veikimo sustojimo atvejų skaičius (*Unique crashes*);
- Unikalių testuojamos programos ne atsakymo į užklausas atvejų skaičius (*Unique hangs*).

Progress srityje pateikiama ši informacija:

- Kiek iš viso buvo išsiųsta ir sėkmingai įvykdyta nuotolinių procedūrų pranešimų (*Total executions*);
- Testavimo proceso greitaveika matuojama išsiųstais pranešimais per sekundę (*Execution speed*);
- Pranešimas, kuris šiuo metu yra keičiamas ir siunčiamas testuojamai programai (*Current message*);
- Esamo pranešimo testavimo proceso vykdymo eiga (*Message progress*).

3.9. Programų pažeidžiamumų aptikimo metodo rezultatai

Testavimo proceso metu programai nutraukus veikimą siūlomas sprendimas užfiksuoja svarbiausią informaciją ir ją įrašo į failų sistemą, JSON formatu. Duomenys, kurie yra išsaugomi faile yra nurodyti 2.3.5 skyrelyje. Detalus duomenų aprašymas yra pateikiamas lentelė 6.

lentelė 6 Veikimo nutraukimo metu įrašomi duomenys

Duomenų lauko pavadinimas	Reikšmės aprašymas
errorCode	Klaidos kodas, kuris yra gaunamas iš *executableOutput* arba *executableEvents* laukų, pvz. 0xc0000005.
errorCause	Nustatyta klaidos priežastis. Priežastis yra nustatoma pagal sąlygas aprašytas 2.3.5 skyrelyje. Jei klaida yra susijusi su talpyklos perpildymu, lauko reikšmėje bus nurodoma *Buffer overflow*. Jei klaida bus susijusi su kreipimusi į nulinę rodyklę, lauko reikšmėje bus nurodoma *Null-pointer dereference*. Jei negalima nustatyti priežasties, lauko reikšmėje bus nurodoma *Unknown*.
moduleName	Testuojamos programos komponentas, kuriame įvyko klaida, pvz. *server.exe*.
faultFunction	Testuojamos programos funkcija, kurioje testavimo metu įvyko klaida. Tai gali būti konkretus funkcijos pavadinimas arba funkcijos adreso postūmis (angl. *offset*) nuo programos failo pradžios, pvz. 0x33D0 arba *MethodThree*.
methodPath	Pilnai kvalifikuotas *gRPC* nuotolinių procedūrų prieigos taško adresas, į kurį siunčiant pranešimą įvyko testuojamos programos veiklos sutrikimas, pvz. *test.testService/MethodThree*.
executableOutput	Testuojamos programos klaidos išvestis.
executableEvents	OS įvykių žurnalo įvykiai susiję su testuojamos programos veiklos sutrikimu.
memoryDumpPath	Testuojamos programos atminties iškelties įrašo vieta failų sistemoje.
crashMessage	*gRPC* nuotolinių procedūrų pranešimas, kuris sukėlė testuojamos programos veiklos sutrikimą. Pranešimas pateikiamas šešioliktainiame *Wire* formate.

3.10. Išvados

1. Skyriuje pasiūlytas pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą, prototipas. Naudojant *Go* programavimo kalbą buvo sukurta terminalo sąsajos programa, kuri automatizuoja pažeidžiamumų aptikimo procesą.
2. Prototipas priima JSON nustatymų, *pcap(ng)* tinklo srauto ir *Protobuf* sąsajų aprašymo failus. Apdorojus duomenis bus išskiriamos *gRPC* nuotolinių procedūrų pranešimų ir pranešimų reikšmių priklausomybės. Priklausomai nuo testavimo tipo bus siunčiami pavieniai pranešimai arba jų serijos susietos rastomis priklausomybėmis. Po pranešimų išsiuntimo bus stebima ar testuojama programa nenutraukė savo veiklos.
3. Programos vykdymo eigos stebėjimas yra atliekamas panaudojant *Frida* dinaminės analizės biblioteką ir jos suteikiamu *Interceptor* ir *Stalker* funkcionalumu. Į testuojamos programos procesą bus įterpiamas *JavaScript* scenarijus, kuris atliks vykdymo eigos stebėjimo funkciją.
4. Pasinaudojant sukurtu prototipu bus atliktas eksperimentinis tyrimas.

4. Programų pažeidžiamumų aptikimo metodo naudojant nuotolinį procedūrų iškvietimą eksperimentinis tyrimas

Remiantis trečiame skyriuje įgyvendinta metodo realizacija buvo suformuotas ir atliktas tyrimas leidžiantis pamatyti metodo veiksmingumą realiomis sąlygomis. Šiame skyriuje bus aprašytas eksperimentinis tyrimas ir pateikiami jo rezultatai.

4.1. Eksperimentinio tyrimo įvertinimo kriterijai ir tyrimo aplinka

Siekiant įvertinti pasiūlyto metodo veiksmingumą ir efektyvumą bus įgyvendintas šis testavimo scenarijus: bus sukurta testavimo platforma su 15 testavimo procedūrų, kurios yra iškviečiamos panaudojant *gRPC* nuotolinių procedūrų karkasą. Dvylika iš penkiolikos procedūrų turi tam tikrus talpyklos perpildymo ir kreipimosi į nulinę rodyklę pažeidžiamumus, kurie pasireiškia testavimo platformos veiklos sutrikdymu.

Dviejose testavimo procedūrose (*HtmlDocOne* ir *HtmlDocTwo*) buvo panaudotas 1.9.11 versijos *htmldoc* [35] programos kodas, kuris turi du talpyklos perpildymo pažeidžiamumus: CVE-2021-23206 [37] ir CVE-2021-26259 [36].

Eksperimentinis tyrimas bus vertinamas pagal šiuos kriterijus:

- Kiek iš viso buvo rasta testuojamų procedūrų pažeidžiamumų;
- Laiko trukmę, sekundėmis, kuri nurodo per kiek laiko buvo rastas pažeidžiamumas pažeidžiamoje nuotolinėje procedūroje;
- Naujai sukurtų ir išsiųstų nuotolinių procedūrų skaičius iki pažeidžiamumo radimo momento.

Eksperimentinio tyrimo metu pasiūlytas metodas bus lyginamas su panašiais egzistuojančiais įrankiais: *proto-fuzzer* [38] ir *WinAFL* [39] su *libprotobuf-mutator* [40]. Pastarieji įrankiai bus naudojami rasti pažeidžiamumus anksčiau paminėtoje testavimo platformoje. Gauti rezultatai bus lyginami su pasiūlyto metodo rezultatais.

Metodo eksperimentinis tyrimas yra atliekamas viename asmeniniame kompiuteryje, kuriame yra įdiegta ši techninė ir programinė įranga:

- Windows 10 Pro 19043.1165 64 bitų operacinė sistema;
- AMD Ryzen 5 2600 šešių fizinių ir dvylikos loginių branduolių procesorius @ 3.40GHz;
- 16 GB darbinės atminties;
- 250 GB SSD diskas;
- *proto-fuzzer* 0.0.3 versija
- *WinAFL* 1.16b su *libprotobuf-mutator* biblioteka.

Eksperimentiniam tyrimui buvo sukurta testavimo platforma – C++ kalbos pagrindu įgyvendinta programa, kuri geba vykdyti *gRPC* nuotolines procedūras. Visos procedūros yra suskirstytos į keturias kategorijas:

- Nesudėtingos procedūros – procedūros be sudėtingų alternatyvių vykdymo kelių;
- Procedūros su alternatyviais vykdymo keliais – procedūros, kurios priklausomai nuo įvesties duomenų gali vykdyti skirtingas operacijas;

- Susijusios procedūros – procedūros, kurios turi nustatytą vykdymo tvarką, t. y. procedūra Nr. 1 turi būti įvykdyta prieš procedūrą Nr. 2;
- Procedūros su realios programinės įrangos pažeidžiamumais – procedūrose iškviečiamas kodas iš bibliotekos ar programinės įrangos, kurioje buvo rasti jau žinomi pažeidžiamumai.

Penkių iš penkiolikos procedūrų pavadinimai baigiasi žodžiais *PartOne*, *PartTwo,* ir *PartThree*. Tai reiškia, kad procedūros turi būti kviečiamos eilės tvarka, pvz. *MethodNinePartOne* procedūra turi būti siunčiama prieš *MethodNinePartTwoBad* procedūrą.

Kompiliuojant testavimo platformą buvo keičiami specifiniai kompiliatoriaus nustatymai siekiant įgalinti kuo daugiau suteikiamų atminties ir programos veikimo saugos priemonių. Buvo pakeisti šie nustatymai:

- *Optimization* - /Od;
- *Inline Function Expansion* - /Ob0;
- *Basic Runtime Checks* - /RTCsu;
- *Control Flow Guard* - /guard:cf.
- *Preprocessor Definitions* - _CRT_SECURE_NO_WARNINGS

Pati platforma buvo kompiliuojama su *Release* tipo nustatymais, 64 bitų architektūros procesoriams.

4.2. Eksperimentinis tyrimas

Siekiant įvertinti metodo efektyvumą randant talpyklos perpildymo ir kreipimosi į nulinę rodyklę pažeidžiamumus, buvo panaudotas pasiūlytas metodas. Pasiūlyto metodo rezultatai bus lyginami su kitais dviem sprendimais. Kiekvieno sprendimo testavimas buvo atliekamas iki kol bus rastas pažeidžiamumas testuojamoje procedūroje, testavimo metodas užbaigs darbą pats arba pasibaigs vienos valandos laiko limitas.

Siekiant eksperimentinio tyrimo metu panaudoti *proto-fuzzer* įrankį buvo sukurtas paprastas JavaScript kalbos scenarijus, kuris naudoja *proto-fuzzer* įrankio kodą. Tai buvo padaryta, nes pats įrankis suteikia tik nuotolinių procedūrų kūrimo ir siuntimo funkcionalumą, tačiau visas kodas turi būti individualiai pritaikomas kiekvienam pranešimui. Prieš pradedant testavimą reikėjo atlikti įrankio kodo pakeitimus, kadangi esamas kodas neveikė su naujausiomis bibliotekomis. Pagrindinė įrankio logika nebuvo keičiama.

Eksperimentinio tyrimo metu buvo panaudotas *WinAFL* įrankis su *libprotobuf-mutator Protobuf* pranešimų keitimo biblioteka. Pirmiausia buvo sukompiliuotas *WinAFL* įrankis. Šis įrankis palaiko tris galimus programų profiliavimo būdus: naudojant *DynamoRIO* dinaminės analizės programinę įrangą, *Intel PT* centrinio procesoriaus profiliavimo funkcionalumą ir *Syzygy* profiliavimo karkasą. Eksperimentiniam tyrimui buvo pasirinktas *DynamoRIO* programinė įranga, kadangi tyrimas yra vykdomas kompiuteryje, kuriame veikia *AMD* centrinis procesorius, o *Intel PT* profiliavimas yra pasiekiamas tik *Intel* centriniuose procesoriuose. *Syzygy* profiliavimo karkasą galima naudoti tik su 32 bitų architektūros programomis.

Norint panaudoti *libprotobuf-mutator* pranešimų keitimo biblioteką, ją reikėjo sukompiliuoti į atskirą bibliotekos failą, kuris buvo naudojamas *WinAFL* įrankyje. Veikimo metu, šis įrankis įkrauna šią

biblioteką ir perduoda įvesties duomenis bibliotekoje įgyvendintiems nuotolinių pranešimų kūrimo metodams. Naujai sukurti pranešimai yra siunčiami testuojamai programai. Iš pradžių, testavimą buvo bandoma atlikti su nepakeista testuojama programa, tačiau testavimo proceso pradžioje, programa dažnai užbaigdavo savo veikimą dėl *WinAFL* įrankyje naudojamo *DynamoRIO* profiliavimo PĮ. Buvo nuspręsta sukurti atskirą testojamos programos versiją, kuri duomenų apsikeitimui nenaudoja *gRPC* karkaso. Visos procedūros buvo nuskaitomos iš failo, o tada iškviečiamas atsakingas metodas, kuris apdoroja nuskaitytą nuotolinės procedūros pranešimą. Visos įgyvendintos procedūros buvo perkeltos į šią pakeistą programą, kuri bus testuojama *WinAFL* įrankio. Testuojama programa buvo kompiliuojama su tais pačiais nustatymais, kaip ir originali testavimo platforma. Kiekviena procedūra buvo testuojama vieną valandą. Per tą laikotarpį buvo stebima ar buvo rastas pažeidžiamumas, jei taip, buvo pasižymimas laiko tarpas per kiek laiko buvo rastas pažeidžiamumas ir kiek pranešimų buvo sukurta ir perduota testuojamai programai.

Eksperimentinio tyrimo metu naudojant pasiūlytą metodą pirmiausia buvo sukurti pavyzdiniai nuotolinių procedūrų pranešimų srauto failai. Šie failai buvo gauti naudojant *Wireshark* tinklo srauto stebėjimo įrankį. Šis įrankis fiksavo tinklo srautą, kol buvo kviečiamos pažeidžiamos nuotolinės procedūros su pažeidžiamumų neišprovokuojančiomis įvestimis. Kiekvienai pažeidžiamai procedūrai buvo sukurtas vienas tinklo srauto (.pcapng) failas. Kitas žingsnis buvo gauti testuojamos programos metodų adresų postūmius (angl. *offset*), kurie bus naudojami programos profiliavimo procese. Eksperimentinio tyrimo metu, pirmai dešimčiai metodų buvo naudojami nustatymai, kurie yra pateikti lentelė 7.

lentelė 7 Eksperimentinio tyrimo metu naudoti pasiūlyto metodo nustatymai

Nustatymo pavadinimas	Reikšmė
pathToExecutable	C:\\Source\\Testing\\Server\\server.exe
executableArgs	[]
outputPath	C:\\Source\\Output"
dumpExecutablePath	C:\\Source\\gIPCFuzz\\procdump64.exe
performMemoryDump	true
handlers	[{"method": "test.testService/MethodOne", "module": "server.exe", "handler": "0x7A40"}, {"method": "test.testService/MethodTwo", "module": "server.exe", "handler": "0xB490"}, {"method": "test.testService/MethodThree", "module": "server.exe", "handler": "0xAD00"}, {"method": "test.testService/MethodFour", "module": "server.exe", "handler": "0x6900"}, {"method": "test.testService/MethodFive", "module": "server.exe", "handler": "0x6610"}, {"method": "test.testService/MethodSix", "module": "server.exe", "handler": "0x8770"}, {"method": "test.testService/MethodSeven", "module": "server.exe", "handler": "0x8100"},

	{"method": "test.testService/MethodEight", "module": "server.exe", "handler": "0x60F0"}, {"method": "test.testService/HtmlDoc", "module": "server.exe", "handler": "0x356D0"}, {"method": "test.testService/HtmlDoc", "module": "server.exe", "handler": "0x357C0"} {"method": "test.testService/MethodNinePartOne", "module": "server.exe", "handler": "0x6D10"}, {"method": "test.testService/MethodNinePartTwo", "module": "server.exe", "handler": "0x6F80"}, {"method": "test.testService/MethodTenPartOne", "module": "server.exe", "handler": "0x9700"}, {"method": "test.testService/MethodTenPartTwo", "module": "server.exe", "handler": "0xA7C0"}, {"method": "test.testService/MethodTenPartThree", "module": "server.exe", "handler": "0x9970"}]
host	localhost
port	50051
ssl	false
performDryRun	false
singleFieldMutation	false
dependencyUnawareSending	true
protoFilesPath	C:\\Source\\Testing\\Protos
protoFilesIncludePath	["C:\\Source\\Testing\\Protos"]
pcapFilePath	C:\\Source\\Testing*.pcapng
maxMsgSize	4194304
useInstrumentation	true

Procedūros *MethodNinePartTwo* ir *MethodTenPartThree* buvo testuojamos su *dependencyUnawareSending* nustatymu, kurio reikšmė yra *true*. Tai pakeičia metodo vykdymo režimą, kas leidžia ieškoti susijusių nuotolinių procedūrų ir atitinkamai siųsti pranešimų grandines.

lentelė 8 Eksperimentinio tyrimo rezultatai

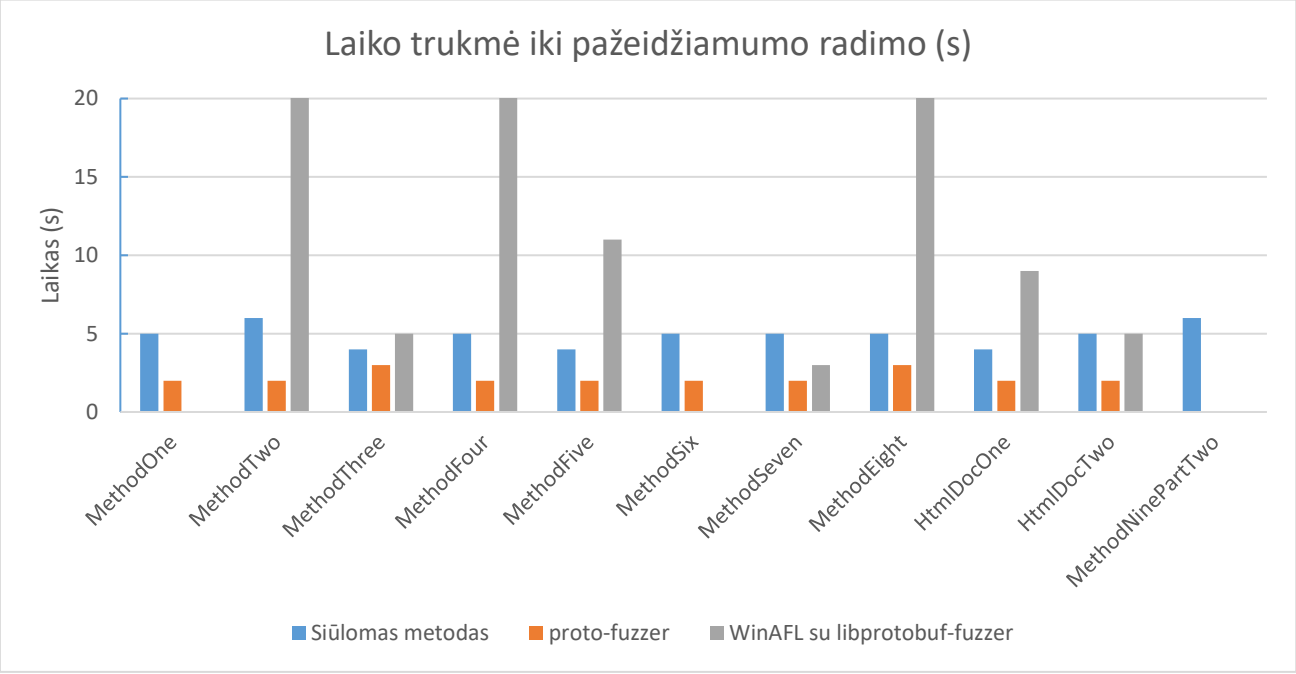
Procedūros pavadinimas	Pažeidžiamumo aprašymas	Siūlomas metodas		WinAFL su libprotobuf-mutator		proto-fuzzer	
		Išsiųstų pranešimų sk./praėjęs laikas iki pažeidžiamumo radimo (s)	Aptiktas pažeidžiamumas	Išsiųstų pranešimų sk./praėjęs laikas iki pažeidžiamumo radimo (s)	Aptiktas pažeidžiamumas	Išsiųstų pranešimų sk./praėjęs laikas iki pažeidžiamumo radimo (s)	Aptiktas pažeidžiamumas
MethodOne	Steko talpyklos perpildymas kai įvestis yra ilgesnė nei 240 simbolių	5/5 sek.	Taip	665000/-	Ne	10/2 sek.	Taip
MethodTwo	Heap talpyklos perpildymas kai įvestyje yra unikodo simbolių	4/6 sek.	Taip	3737/28 sek.	Taip	3/2 sek.	Taip
MethodThree	Steko talpyklos perpildymas kai įvestis yra ilgesnė nei 256 simboliai ir atskirai pateiktas įvesties ilgis yra didesnis nei 256	2/4 sek.	Taip	1357/5 sek.	Taip	7/3 sek.	Taip
MethodFour	Kreipimasis į nulinę rodyklę,	3/5 sek.	Taip	5043/78 sek.	Taip	8/2 sek.	Taip

	kai išskiriama atmintis su dydžiu, kuris perpildo sveikojo skaičiaus tipą						
MethodFive	Kreipimasis į nulinę rodyklę, kai išskiriama atmintis su dydžiu kuris gaunamas keičiant netinkamą įvesties slankiojo kablelio skaičių į sveikąjį skaičių	5/4 sek.	Taip	4983/11 sek.	Taip	4/2 sek.	Taip
MethodSix	Steko talpyklos perpildymas kai įvesties teksto eilutė yra ilgesnė nei 240 simbolių ir išvardijimo reikšmė lygi 2	6/5 sek.	Taip	649000/-	Ne	10/2 sek.	Taip
MethodSeven	Heap talpyklos perpildymas, kai išskiriama atmintis su	7/5 sek.	Taip	962/3 sek.	Taip	2/2 sek.	Taip

	mažesniu dydžiu, o įrašymui naudojama didesnė įvesties reikšmė						
MethodEight	Kreipimasis į nulinę rodyklę, kai išskiriama atmintis su dydžiu, kuris mažesnis už 0	7/5 sek.	Taip	10900/67 sek.	Taip	34/3 sek.	Taip
MethodNinePartTwo	Heap talpyklos perpildymas kai įvestis yra didesnė nei 500 simbolių	16/6 sek.	Taip	-	Ne*	-	Ne*
MethodTenPartThree	Heap talpyklos perpildymas kai įvestis yra didesnė nei 500 simbolių	77 pr.	Ne	-	Ne*	-	Ne*
HtmlDocOne	Heap talpyklos perpildymas kai įvedamas netinkamas skaičius (CVE-2021-26259 [36])	3/4 sek.	Taip	2139/9 sek.	Taip	8/2 sek.	Taip

HtmlDocTwo	Steko talpyklos perpildymas kai įvedamas netinkamas skaičius (CVE-2021-23206 [37])	3/5 sek.	Taip	1901/5 sek.	Taip	6/2 sek.	Taip

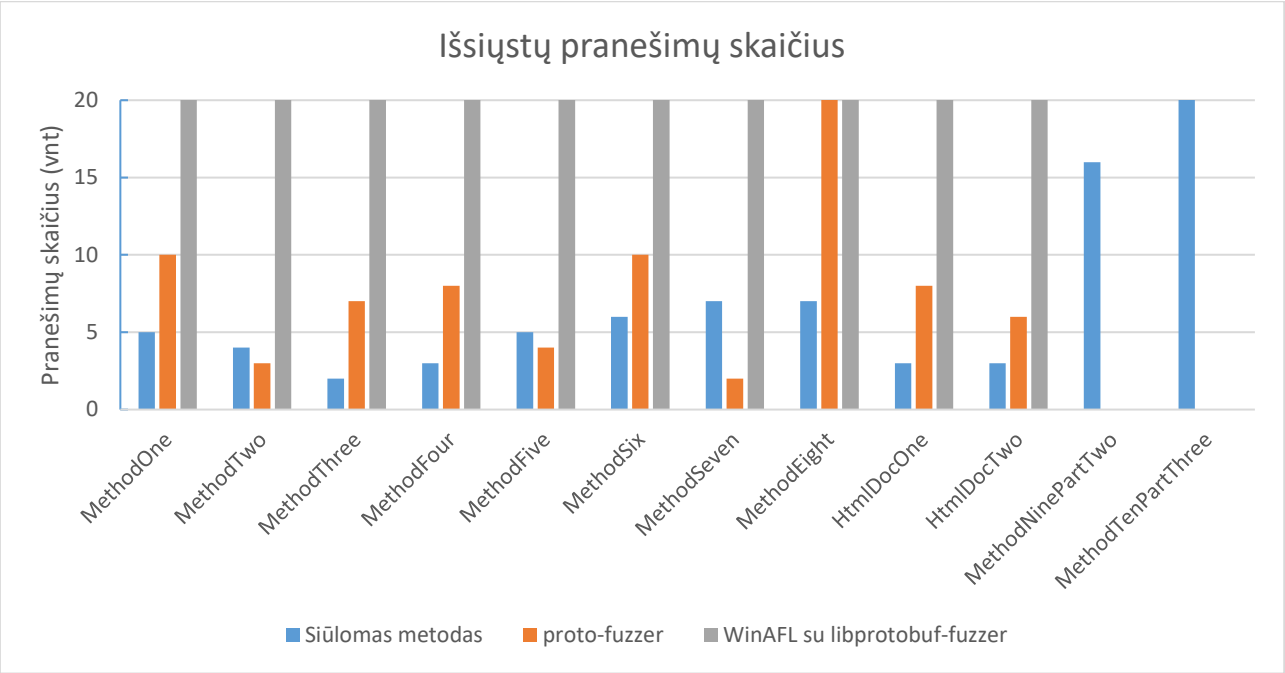
Eksperimentinio tyrimo rezultatai yra pateikti lentelė 8. Abu kitų autorių sprendimai negali rasti pažeidžiamumų, kai procedūros yra susijusios, todėl rezultatų lentelėje yra žymima, kad šie metodai nesugebėjo rasti pateiktų pažeidžiamumų. Šie rezultatai pažymėti žvaigždute. Kaip galima matyti iš aukščiau pateiktos lentelės, siūlomas metodas sugebėjo rasti 11 iš 12 egzistuojančių pažeidžiamumų. Paskutinis pažeidžiamumas nebuvo rastas, nes siūlomas metodas yra įgyvendintas taip, kad klaidingi duomenys turi būti vedami paskutiniame nuotolinių procedūrų grandinės pranešime. Šiuo atveju, klaidingi duomenys buvo vedami antrame grandinės pranešime, kai paskutinis pranešimas tik iškviečia procedūrą su jau anksčiau įvestais duomenimis. Kalbant apie pasiūlyto metodo greitaveiką matuojant laiką nuo sprendimo paleidimo iki pažeidžiamumo radimo momento, siūlomas metodas nusileido *proto-fuzzer* sprendimui, tačiau buvo greitesnis už *WinAFL* su *libprotobuf-mutator* sprendimą 6 iš 8 testavimo atvejų, kai buvo rasti pažeidžiamumai. Greitaveikos tyrimo rezultatai yra pateikti pav. 27.



pav. 27 Pasiūlyto metodo greitaveikos randant pažeidžiamumus palyginimas

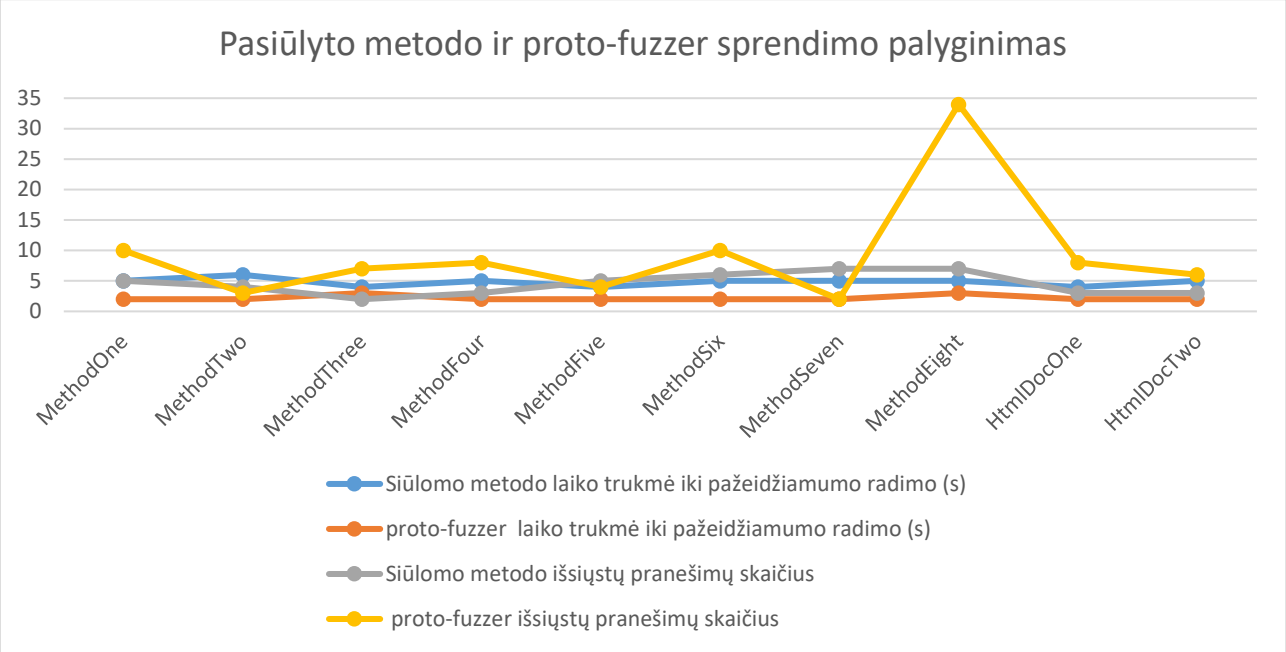
Lyginant sprendimų efektyvumą pagal išsiųstų pranešimų skaičių, siūlomas metodas 7 atvejuose iš 10 išsiuntė mažiausią pranešimų skaičių, kurie sukėlė atitinkamus pažeidžiamumus. Šiame tyrime verta išskirti *WinAFL* su *libprotobuf-mutator* sprendimą, kuris pareikalavo labai didelio pranešimų

skaičiaus, tam, kad sugebėtų atrasti pažeidžiamumus. Tai lėmė faktas, kad *libprotobuf-mutator* bibliotekos pranešimų kūrimo algoritmai nėra ištobulinti, todėl norint pasiekti geresnių rezultatų reikia papildomos pranešimų kūrimo logikos. Išsiųstų pranešimų tyrimo rezultatai pateikti pav. 28.



pav. 28 Pasiūlyto metodo išsiųstų pranešimų skaičiaus randant pažeidžiamumus palyginimas

Lyginant siūlomą metodą ir geriausius rezultatus, iš abiejų lyginamų sprendimų, parodžiusį *proto-fuzzer* sprendimą, galima pastebėti, kad 9 iš 10 atvejų, siūlomas metodas yra iki trijų kartų lėtesnis. Lėtumą siūlomame metode lemia pradinis numatomo keitimų ciklų skaičiaus nustatymas ir atliekamas kodo profiliavimas. Kalbant apie sprendimų efektyvumą, pagal surinktus duomenis, siūlomas metodas išsiunčia iki penkių kartų mažiau pranešimų lyginant su kitu sprendimu. Šį skirtumą lemia įgyvendintų pranešimų kūrimo algoritmų sudėtingumas ir duomenų rinkinių kiekis. Detalesni duomenys yra pavaizduoti pav. 29.



pav. 29 Pasiūlyto metodo ir *proto-fuzzer* sprendimo palyginimas

Siūlomas metodas taip pat buvo ištirtas panaudojant dvi skirtingas pranešimų kūrimo strategijas: vienos ir visų reikšmių keitimas vienu metu. Kaip galima matyti lentelė 9 ir pav. 30, akivaizdūs skirtumai tarp strategijų pastebimi nuo *MethodSix* procedūros. Nuo šios procedūros, visi pranešimai iki *HtmlDocOne* ir *HtmlDocTwo* procedūrų turi po kelis laukus. Galime pastebėti, kad yra kuriama daugiau pranešimų per tą patį veikimo laiką, lyginant su pranešimais kurtais panaudojant visų reikšmių keitimo strategiją.

lentelė 9 Pasiūlyto metodo eksperimentinio tyrimo rezultatai taikant skirtingas reikšmių pakeitimo strategijas

Procedūra	Vienos reikšmės pakeitimo strategija		Visų reikšmių pakeitimo strategija	
	Išsiųstų pranešimų skaičius	Praėjęs laikas iki pažeidžiamumo radimo (s)	Išsiųstų pranešimų skaičius	Praėjęs laikas iki pažeidžiamumo radimo (s)
MethodOne	5	3	2	5
MethodTwo	5	5	4	6
MethodThree	2	3	2	4
MethodFour	2	5	3	5
MethodFive	2	5	5	4
MethodSix	11	5	6	5
MethodSeven	13	5	7	5
MethodEight	11	4	7	5
HtmlDocOne	3	4	3	5
HtmlDocTwo	3	5	4	4
MethodNinePartTwo	16	6	16	6
MethodTenPartThree	256	-	77	-

pav. 30 Pasiūlyto metodo išsiųstų pranešimų skaičius naudojant skirtingas pranešimo keitimo strategijas

4.3. Eksperimentinio tyrimo rezultatų apibendrinimas

1. Buvo atliktas eksperimentinis tyrimas: siūlomas metodas ir kiti du lyginami sprendimai buvo panaudoti siekiant rasti 12 pažeidžiamumų sukurtoje testavimo platformoje. Tyrimo metu surinkti duomenys apie kiekvieno sprendimo pažeidžiamumų radimo laikus ir išsiųstų pranešimų skaičius.
2. Siūlomas metodas aptiko 11 iš 12 galimų pažeidžiamumų. Pastebėta, kad:
    a. Siūlomas metodas greitaveika nusileidžia *proto-fuzzer* sprendimui, tačiau per visą veikimo laiką išsiunčia mažiau pranešimų nei analogas;
    b. Siūlomo metodo lėtumą sukelia pradžioje atliekamas numatomo keitimų ciklų skaičiaus nustatymas ir kodo profiliavimas. Tačiau keitimų ciklų skaičiavimas leidžia prioretizuoti sudėtingas procedūras, kas leistų testuojamoje programoje pažeidžiamumus rasti greičiau, kai vienu siūlomo sprendimo paleidimu yra testuojamos visos norimos procedūros;
    c. Pasiūlyto metodo ir *proto-fuzzer* sprendimo išsiųstų pranešimų skaičiaus skirtumas parodo, kad šis metodas sugeba greičiau suformuoti pažeidžiamumus sukeliančius pranešimus;
    d. Pasiūlyto metodo visų reikšmių keitimo strategija per panašų laiką leidžia sumažinti išsiunčiamų pranešimų skaičių, kurie sukelia pažeidžiamumus;
    e. Pažeidžiamumas paskutinėje procedūroje nebuvo aptiktas, nes pranešimų grandinių logika buvo įgyvendinta su idėja, kad visų pažeidžiamų procedūrų duomenys bus įvedami grandinės pabaigoje, o ne ankstesniuose pranešimuose.
3. Nustatyta, kad didelę įtaką pažeidžiamumų radimui turi pranešimų laukų reikšmių keitimo logika, t. y. kokios reikšmės bus naudojamos, kokia yra keitimo tvarka ir reikšmių teisingumas.

Išvados

1. Išanalizuoti trys pagrindiniai tarp-procesinėje komunikacijoje naudojami nuotolinių procedūrų iškvietimo karkasai: *gRPC, WCF* ir *Apache Thrift*. Išanalizuoti karkasai lengvai įgyvendinami nepriklausomai nuo programavimo kalbos.
2. Išanalizavus juodosios, baltosios ir pilkosios dėžių testavimo metodų panaudojimą programų pažeidžiamumų aptikimui pastebėta, kad pilkosios dėžės metodas yra tinkamiausias atlikti dinaminę programų pažeidžiamumų paiešką. Skirtingai nei baltosios dėžės metodas, pilkosios dėžės metodas nereikalauja programinio kodo testavimo procesui atlikti. Lyginant su juodosios dėžės metodu, pilkosios dėžės metodas suteikia patikimesnius ir greitesnius rezultatus.
3. Išanalizavus statinius ir dinaminius testavimo metodus nustatyta, kad priimtiniausias yra miglotąja logika paremtas metodas dėl savo universalumo ir gebėjimo derinti tiek statinės, tiek dinaminės analizės privalumus.
4. Remiantis analizės rezultatais nustatytas būtinumas sudaryti programų pažeidžiamumų aptikimo integruotą metodą naudojant nuotolinį procedūrų iškvietimą. Šis metodas galėtų aptikti kreipimosi į nulinę rodyklę ir talpyklos perpildymo pažeidžiamumus.
5. Siūlant magistro darbo sprendimą tikslinga įgyvendinti dinaminį miglotosios logikos, pilkosios dėžės metodus ir panaudoti *gRPC* arba *Apache Thrift* karkasus komunikacijai su programine įranga.
6. Eksperimentiškai ištyrus programų pažeidžiamumų aptikimo metodą, naudojant nuotolinį procedūrų iškvietimą, nustatyta, kad metodas aptiko 11 iš 12 galimų pažeidžiamumų. Siūlomas metodas, iki trijų kartų, greitaveika nusileidžia vienam iš lyginamų sprendimų. Nustatyta, kad lėtumą sukelia pradžioje atliekamas numatomo keitimų ciklų skaičiaus nustatymas ir kodo profiliavimas. Tačiau keitimų ciklų skaičiavimas leidžia prioretizuoti sudėtingas procedūras, kas leistų testuojamoje programoje pažeidžiamumus rasti greičiau, kai vienu siūlomo sprendimo paleidimu yra testuojamos visos norimos procedūros.
7. Lyginant su *proto-fuzzer* sprendimu, šis metodas išsiunčia iki penkių kartų mažiau nuotolinių procedūrų pranešimų ir sugeba efektyviau suformuoti pažeidžiamumus iššaukiančius pranešimus.
8. Pasiūlyto metodo visų reikšmių keitimo strategija per panašų laiką leidžia dar labiau sumažinti išsiunčiamų pranešimų skaičių, kurie iššaukia pažeidžiamumus. Tai lemia skirtinga pranešimų laukų reikšmių keitimo logika, t. y. kokios reikšmės bus naudojamos, kokia yra keitimo tvarka ir reikšmių teisingumas.
9. Parašytas straipsnis ir perskaitytas pranešimas tarptautinėje konferencijoje IVUS 2022 (the 27th International Conference on Information Technology „IVUS 2022“ May 12, 2022) [41].

Literatūros sąrašas

1. GARG, S. - SINGH, R.K. - MOHAPATRA, A.K. Analysis of software vulnerability classification based on different technical parameters [interaktyvus]. Iš: *Information Security Journal: A Global Perspective*. 2019. Vol. 28, no. 1–2, p. 1–19 [žiūrėta 2020-12-27]. ISSN 1939-3555. Prieiga per internetą: https://www.tandfonline.com/doi/full/10.1080/19393555.2019.1628325;
2. CWE - 2021 CWE Top 25 Most Dangerous Software Weaknesses [interaktyvus]. 2021 [žiūrėta 2022-04-16]. Prieiga per internetą: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html;
3. NVD - CVE-2017-3730 [interaktyvus]. 2017 [žiūrėta 2022-04-16]. Prieiga per internetą: https://nvd.nist.gov/vuln/detail/CVE-2017-3730;
4. TOMASSI, D.A. - RUBIO-GONZÁLEZ, C. On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions [interaktyvus]. Iš: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021. p. 292–303 [žiūrėta 2022-04-16]. ISSN 2643-1572. Prieiga per internetą: https://ieeexplore.ieee.org/document/9678535?arnumber=9678535;
5. MOUZARANI, M. - SADEGHIYAN, B. - ZOLFAGHARI, M. A smart fuzzing method for detecting heap-based vulnerabilities in executable codes [interaktyvus]. Iš: *Security and Communication Networks*. 2016. Vol. 9, no. 18, p. 5098–5115 [žiūrėta 2020-10-18]. ISSN 1939-0122. Prieiga per internetą: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1681;
6. JIMENEZ, W. - MAMMAR, A. - CAVALLI, A. Software Vulnerabilities, Prevention and Detection Methods: A Review 1 [interaktyvus]. 2010. p. 12 [žiūrėta 2020-12-01]. Prieiga per internetą: https://www.researchgate.net/publication/253704494_Software_Vulnerabilities_Prevention_and_Detection_Methods_A_Review_1;
7. CHEN, D. - ZHANG, Y. - WEI, W. - WANG, S. - HUANG, R. - LI, X. - QU, B. - JIANG, S. Efficient vulnerability detection based on an optimized rule-checking static analysis technique [interaktyvus]. Iš: *Frontiers of Information Technology & Electronic Engineering*. 2017. Vol. 18, no. 3, p. 332–345 [žiūrėta 2020-12-27]. ISSN: 2095-9230. Prieiga per internetą: https://doi.org/10.1631/FITEE.1500379;
8. ZAAZAA, O. - BAKKALI, H.E. Dynamic vulnerability detection approaches and tools: State of the Art [interaktyvus]. Iš: *2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS)*. 2020. p. 1–6 [žiūrėta 2020-12-27]. ISBN: 978-1-7281-8085-4. Prieigą per internetą: https://doi.org/10.1109/ICDS50568.2020.9268686;
9. CHEN, C. - CUI, B. - MA, J. - WU, R. - GUO, J. - LIU, W. A systematic review of fuzzing techniques [interaktyvus]. Iš: *Computers & Security*. 2018. Vol. 75, p. 118–137 [žiūrėta 2020-12-28]. ISSN: 0167-4048. Prieiga per internetą: http://www.sciencedirect.com/science/article/pii/S0167404818300658;
10. YANG, K. - ZHAO, H. - ZHANG, C. - ZHUGE, J. - DUAN, H. Fuzzing IPC with Knowledge Inference [interaktyvus]. Iš: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. 2019. p. 11–1109 [žiūrėta 2020-12-27]. ISBN: 978-1-72814-222-7. Prieiga per internetą: https://ieeexplore.ieee.org/document/9049571/;
11. Peach Fuzzer: Discover Unknown Vulnerabilities [interaktyvus]. 2020 [žiūrėta 2021-01-02]. Prieiga per internetą: https://www.peach.tech/products/peach-fuzzer/;

12. CHEN, Y. - LAN, T. - VENKATARAMANI, G. Exploring Effective Fuzzing Strategies to Analyze Communication Protocols [interaktyvus]. Iš: *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation - FEAST'19*. 2019. p. 17–23 [žiūrėta 2020-12-27]. ISBN: 978-1-4503-6834-6. Prieiga per internetą: https://doi.org/10.1145/3338502.3359762;
13. American Fuzzy Lop [interaktyvus]. 2021 [žiūrėta 2021-01-02]. Prieiga per internetą: https://github.com/google/AFL;
14. LIANG, H. - PEI, X. - JIA, X. - SHEN, W. - ZHANG, J. Fuzzing: State of the Art [interaktyvus]. Iš: *IEEE Transactions on Reliability*. 2018. Vol. 67, no. 3, p. 1199–1218 [žiūrėta 2020-12-27]. ISSN: 1558-1721. Prieiga per internetą: https://doi.org/10.1109/TR.2018.2834476;
15. PHAM, V.-T. - BOHME, M. - ROYCHOUDHURY, A. AFLNET: A Greybox Fuzzer for Network Protocols [interaktyvus]. Iš: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020. p. 460–465 [žiūrėta 2021-01-02]. ISBN: 978-1-7281-5779-5. Prieiga per internetą: https://doi.org/10.1109/ICST46399.2020.00062;
16. Boofuzz, Network protocol fuzzing for humans [interaktyvus]. 2021 [žiūrėta 2021-01-02]. Prieiga per internetą: https://github.com/jtpereyda/boofuzz;
17. OpenRCE/sulley [interaktyvus]. 2021 [žiūrėta 2021-01-02]. Prieiga per internetą: https://github.com/OpenRCE/sulley;
18. SHEAKH, Dr.T. A Comparative Study of Software Testing Techniques Viz. White Box Testing Black Box Testing and Grey Box Testing [interaktyvus]. Iš: *International Journal of Allied Practice, Research and Review Website: www.ijaprr.com*. 2015. p. 1-8 [žiūrėta 2020-12-27]. ISSN 2350-1294. Prieiga per internetą: https://www.researchgate.net/publication/276028491_A_Comparative_Study_of_Software_Testing_Techniques_Viz_White_Box_Testing_Black_Box_Testing_and_Grey_Box_Testing;
19. BAGCI, H. - KARA, A. A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication [interaktyvus]. Iš: *Proceedings of the 11th International Joint Conference on Software Technologies*. 2016. p. 117–124 [žiūrėta 2021-12-28]. ISBN: 978-989-758-194-6. Prieiga per internetą: https://doi.org/10.5220/0005931201170124;
20. SOUMAGNE, J. - KIMPE, D. - ZOUNMEVO, J. - CHAARAWI, M. - KOZIOL, Q. - AFSAHI, A. - ROSS, R. Mercury: Enabling remote procedure call for high-performance computing [interaktyvus]. Iš: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 2013. p. 1–8 [žiūrėta 2020-12-27]. ISBN: 978-1-4799-0898-1. Prieiga per internetą: https://doi.org/10.1109/CLUSTER.2013.6702617;
21. gRPC Authors. gRPC Documentation. [interaktyvus]. 2021 [žiūrėta 2021-05-08]. Prieiga per internetą: https://grpc.io/docs/;
22. Windows Communication Foundation, Microsoft. [interaktyvus]. 2021 [žiūrėta 2021-08-23]. Prieiga per internetą: https://docs.microsoft.com/en-us/dotnet/framework/wcf/;
23. FLORIN ILCA, L. - BALAN, T. Windows Communication Foundation Penetration Testing Methodology [interaktyvus]. Iš: *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*. 2021. p. 1–4 [žiūrėta 2021-08-23]. ISBN: 978-1-6654-4995-3. Prieiga per internetą: https://doi.org/10.1109/EMES52337.2021.9484145;

24. Apache Software Foundation. Apache Thrift. [interaktyvus]. 2021 [žiūrėta 2021-05-08]. Prieiga per internetą: https://thrift.apache.org/;
25. SLEE, M. - AGARWAL, A. - KWIATKOWSKI, M. Thrift: Scalable Cross-Language Services Implementation [interaktyvus]. 2007. p. 8 [žiūrėta 2021-05-08]. Prieiga per internetą: https://thrift.apache.org/static/files/thrift-20070401.pdf;
26. Internet Engineering Task Force, Hypertext Transfer Protocol Version 2 (HTTP/2) [interaktyvus]. 2015 [žiūrėta 2021-05-20]. Prieiga per internetą: https://datatracker.ietf.org/doc/html/rfc7540;
27. Google. Protocol Buffers Language Guide [interaktyvus]. 2021 [žiūrėta 2021-08-20]. Prieiga per internetą: https://developers.google.com/protocol-buffers/docs/overview;
28. FELL, J. A Review of Fuzzing Tools and Methods [interaktyvus]. Iš: *PenTest Magazine*. 2017. p. 8 [žiūrėta 2021-05-08]. Prieiga per internetą: https://www.semanticscholar.org/paper/A-Review-of-Fuzzing-Tools-and-Methods-Fell/67798c16cfbf2582852f62efb1a70b131c971da1;
29. Valgrind Developers. Valgrind [interaktyvus]. 2021 [žiūrėta 2021-05-08]. Prieiga per internetą: https://valgrind.org/;
30. Frida [interaktyvus]. 2021 [žiūrėta 2021-05-08]. Prieiga per internetą: https://frida.re/;
31. DynamoRIO [interaktyvus]. 2021 [žiūrėta 2021-05-08]. Prieiga per internetą: https://dynamorio.org/;
32. QEMU [interaktyvus]. 2021 [žiūrėta 2021-05-08]. Prieiga per internetą: https://www.qemu.org/;
33. KARAMCHETI, S. - MANN, G. - ROSENBERG, D. Adaptive Grey-Box Fuzz-Testing with Thompson Sampling [interaktyvus]. Iš: *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*. 2018. p. 37–47. ISBN: 9781450360043. Prieiga per internetą: https://doi.org/10.1145/3270101.3270108;
34. M. Russinovich, A. Richards. ProcDump v10.0 [interaktyvus]. 2021 [žiūrėta 2021-05-08]. Prieiga per internetą: https://docs.microsoft.com/en-us/sysinternals/downloads/procdump;
35. HTMLDOC [interaktyvus]. 2022 [žiūrėta 2022-04-05]. Prieiga per internetą: https://www.msweet.org/htmldoc/;
36. CVE-2021-26259, National Vulnerability Database. [interaktyvus]. Prieiga per internetą: https://nvd.nist.gov/vuln/detail/CVE-2021-26259. [žiūrėta 2022-04-11]
37. CVE-2021-23206, National Vulnerability Database. [interaktyvus]. Prieiga per internetą: https://nvd.nist.gov/vuln/detail/CVE-2021-23206. [žiūrėta 2022-04-11]
38. proto-fuzzer [interaktyvus]. 2022 [žiūrėta 2022-03-26]. Prieiga per internetą: https://github.com/ChaosCabbage/proto-fuzzer;
39. WinAFL [interaktyvus]. 2022 [žiūrėta 2022-03-26]. Prieiga per internetą: https://github.com/googleprojectzero/winafl;
40. libprotobuf-murator [interaktyvus]. 2022 [žiūrėta 2022-03-26]. Prieiga per internetą: https://github.com/google/libprotobuf-mutator;
41. IVUS 2022 konferencijos programa [interaktyvus]. 2022 [žiūrėta 2022-05-12]. Prieiga per internetą: https://ivus.vdu.lt/wp-content/uploads/2022/05/IVUS_programme_2022.pdf;

Priedai

1 priedas. Straipsnis

Detecting Applications Vulnerabilities Using Remote Procedure Calls

Lukas Jokubauskas
Department of Computer Sciences
Kaunas University of Technology
Kaunas, Lithuania
l.jokubauskas@ktu.edu

Jevgenijus Toldinas
Department of Computer Sciences
Kaunas University of Technology
Kaunas, Lithuania
eugenijus.toldinas@ktu.lt

Borisas Lozinskis
Department of Computer Sciences
Kaunas University of Technology
Kaunas, Lithuania
borisas.lozinskis@ktu.lt

***Abstract*— because the quantity of software systems and applications is growing all the time, so is the number of vulnerabilities. The techniques for finding applications vulnerabilities are classified into two main categories: static analysis and dynamic analysis. In this paper, we present a novel method for detecting applications vulnerabilities using the remote procedure call approach, namely Detecting Applications Vulnerabilities using Google Remote Procedure Call (DAVuGRPC) that aims to utilize statically created taint and its dynamic fuzzification during the execution of the application.**

Keywords—Vulnerability detection, Dynamic analysis, Taint dataset, RPC, gRPC

I. Introduction

A software vulnerability can be defined as a defect, weakness, or simply an error in an application that can be exploited by an attacker to change the system's regular behavior [1]. Because the quantity of software systems and applications is growing all the time, so is the number of vulnerabilities. There are various types of application vulnerabilities: injection, cross-site scripting, broken authentication and session management, format string, insecure direct object reference, and many others [2]. In the software industry, vulnerability identification and remediation have been a core and vital operation. Hackers can take advantage of undetected flaws and wreak significant damage to people [3]. While program analysis tools do exist, they often only discover a small subset of probable errors based on predefined rules. With the recent widespread availability of open-source repositories, data-driven methodologies for discovering vulnerability trends have become possible [4].

The techniques for finding applications vulnerabilities are classified into two main categories: static analysis and dynamic analysis [5]. Static application analysis entails methods for inspecting source code or compiled binary without running it. Dynamic analysis is studying application while it is running, with the use of a debugger or other techniques, such as [1]:

- Fault injection is a testing approach that introduces problems to an application to test its behavior. To generate the possible faults, some knowledge of the application is required.
- Fuzzing testing involves feeding the application with random data to see if it can handle it correctly.
- Dynamic taint during the execution of the application, the tainted data is monitored to determine its appropriate validation before accessing sensitive functions.
- Sanitization is a method of avoiding vulnerabilities caused by the use of user-supplied data by implementing newly included functions or custom routines whose main objective is to evaluate or sanitize any input from users before using it inside an application.

Most of the time, cyber security specialists do not have access to the source code of the applications they are testing. As a result, most cyber security specialists aim to automate some tasks using dynamic methodologies. The power of these strategies resides in the fact that the number of false positives is extremely low, and the precision is extremely high [6].

The methods offered by operating systems that allow processes to handle shared data or interact are referred to as interprocess communication (IPC) [7]. IPC is a set of methods for communicating with two processes that may or may not be on the same machine. Direct and indirect communication, synchronous and asynchronous communication, and explicit buffering are all covered.

Remote procedure call (RPC) methods are widely used in the creation of distributed systems because they lower the system's complexity and development costs. The primary purpose of an RPC is to make remote procedure calls transparent to users, allowing them to make remote procedure calls in the same way that they would make local procedure calls [8] and in general means of calling a procedure from a server to a client and receiving the result as a message.

In this paper, we present a novel method for detecting applications vulnerabilities using the remote procedure call approach, namely **D**etecting **A**pplications **V**ulnerabilities **u**sing **G**oogle **R**emote **P**rocedure **C**all (DAVuGRPC) that aims to utilize statically created taint and its dynamic use during the execution of the application. For that purpose, we employ the fuzzification technique for the taint dataset.

The rest of the paper is organized as follows. The second section discusses the related works. The third section overviews application programming interfaces. The fourth section describes the gRPC payload. The fifth section presents the proposed application's vulnerabilities detection method using gRPC. The evaluation framework and experimental setup are presented in section six. The seventh section presents experimental results. The last section concludes the paper with a discussion of future work.

II. Related Work

In [6] an overview of software testing is offered, with a focus on its function in software reliability and comparative effort. Thus, authors [6] present a comparative study of software testing techniques: white box testing, black-box testing, and grey-box testing. In Black Box Testing, the

following techniques are employed: equivalence partitioning, boundary value analysis, fuzzing, cause-effect graph, orthogonal array testing, and all pair testing. White Box Testing employs the following techniques: desk checking, code walkthrough, formal inspections, control flow testing, basis path testing, data flow testing, and loop testing. Grey Box Testing is a product testing method that combines the strategies of Black Box Testing and White Box Testing. The inner structure of the thing being tested is unknown to the analyst in Black Box Testing, whereas the inner structure is known in White Box Testing. The internal structure is partly known in Grey Box Testing. This entails having access to core data structures and algorithms in order to construct test cases, yet testing at the user, or black box, level. Grey Box Testing employs the following techniques: orthogonal array testing, matrix testing, regression testing, and pattern testing. When comparing the three testing approaches of White Box, Grey Box, and Black Box, authors [9] discovered that White Box testing produces better outcomes in terms of software reliability.

Fuzzing is a popular and successful method for detecting security flaws in the software when a system is tested by processing test cases generated by another program in a continuous loop. Simultaneously, the system is being watched for any flaws that may have been disclosed as a result of processing this data. Fuzzing strategies are classified into three groups based on the role they play: sample generation techniques, dynamic analysis approaches, and static analysis techniques [10]. Random mutation, grammatical representation, and scheduling algorithms are three types of sample generation approaches that are used to choose and mutate seeds as well as restrict and generate new samples. To assist in the generation of the new sample, dynamic analysis techniques are employed to acquire dynamic information on the running application. Symbolic expressions, the executed path, taint information on the sample, and codes are all included in this data. Control flow analysis and data flow slices are examples of static analysis. Although static analysis frequently yields false-positive results, it can be used in conjunction with other methods to get extremely useful pretreatment data.

In [11] proposed a system that combines machine learning and bandit-based optimization with state-of-the-art grey-box fuzzing approaches. Authors show significant improvements over numerous state-of-the-art grey-box fuzzers, such as AFL, FidgetyAFL, and the recently released FairFuzz, by using Thompson Sampling to adaptively learn distributions over mutation operators. The first concolic execution-based smart fuzzing method for detecting heap-based buffer overflow in executables was provided in [13]. The suggested fuzzer runs the binary program and determines the path and vulnerability restrictions for the executed path symbolically. It combines the constraints to generate test data that traverses the execution path and detects any flaws. The fuzzer removes each path constraint one at a time and solves the resulting constraints to generate test data that follows novel execution paths. The tainted data is propagated by the proposed algorithm through direct assignment and arithmetic operations.

Inter-Process Communication (IPC) refers to a variety of approaches for one-way or two-way data transmission between threads in one or more processes that can run on a single computer or multiple computers connected by a network [13], [14]. Message passing, synchronization, shared memory, and remote procedure calls (RPC) are some of the IPC approaches that can be divided into groups based on how they communicate shared memory and message passing [15]. The authors in [16] introduced direct IPC (dIPC) to marry the isolation of processes with the performance of synchronous function calls because IPC imposes overheads on a variety of different environments. Threads in one process can call a function on another process, offering the same performance as if the two processes were a single composite application, but without jeopardizing their isolation.

There has been an increase in serialization-based vulnerabilities in online applications in recent years, which has resulted in serious instances exposing the private data of millions of people. An expandable tool for detecting deserialization and object injection vulnerabilities in Java and PHP-based online applications was proposed in [17]. In addition, presented the first deserialization test environment, which may be used to evaluate deserialization vulnerability detection tools as well as for teaching reasons.

III. Application Programming Interfaces

Application Programming Interfaces (APIs) are software intermediaries that define certain rules and determinations for applications to interact and communicate with one another. An API is in charge of delivering a user's response to a system, which is then returned to the user by the system. Representational State Transfer (REST), RPC, and query language for APIs (GraphQL) are the three basic models for creating APIs [18]. The response from the back-end data is delivered to the clients (or users) through the JSON or XML communications format when using REST APIs. The HTTP protocol is commonly used in this architectural style.

The acronym gRPC [19] stands for Google Remote Procedure Call, and it is an RPC-based variation. This technology is based on an HTTP 2.0 RPC API implementation, but HTTP is not presented to the API developer or the server. As a result, there's no need to worry about how RPC principles are mapped to HTTP, which simplifies things. The goal of gRPC is to speed up data transmission between microservices. It is based on the concept of selecting a service, then establishing methods and parameters to allow for remote calling and return types. It also describes the RPC API paradigm in an interface description language (IDL), which makes determining remote operations easier. Protocol Buffers (Protobuf) are used by default in the IDL to describe the service interface as well as the structure of payload messages. gRPC can handle four types of interactions:

- **Unary** – when the client makes a single request and gets a single answer.
- **Server streaming** – in response to a client's request, the server sends a stream of messages. When all of the data has been transmitted, the server sends a status message to conclude the operation.
- **Client streaming** – the client delivers a stream of messages to the server, which responds with a single message.
- **Bidirectional streaming** – the client and server streams are autonomous, which means they can send messages in any sequence. Bidirectional streaming is started and stopped by the client.

gRPC is a great choice for multi-language systems, real-time streaming, and IoT systems that require light-weight message transfer, such as serialized Protobuf messages. Furthermore, gRPC should be considered for mobile apps because it does not require the use of a browser and can profit from fewer messages, preserving the speed of mobile processors [18].

IV. gRPC PAYLOAD DATA STRUCTURE

By default, gRPC serializes payload data using Protobuf. Protocol buffers are a language-independent, platform-independent, and flexible framework for serializing structured data in a forward and backward compatible manner. It's similar to JSON but smaller and faster, plus it creates native language bindings. Protocol buffers are made up of the definition language (in .proto files), the code generated by the proto compiler to interact with data, language-specific runtime libraries, and the serialization format for data written to a file (or sent across a network connection) [20]. Protocol buffers workflow is shown in Fig. 1.

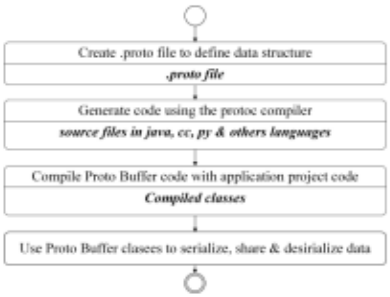


Fig. 1. Protocol buffers workflow

Utility methods to obtain data from files and streams, extract specific values from the data, check if data exists, serialize data back to a file or stream, and other helpful functions are included in the code created by protocol buffers.

Protocol buffer messages and services are described by engineer-authored .proto files (see Fig. 2).

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

Fig. 2. An example of .proto definition [20]

You can define whether a field is optional, repeated (proto2 and proto3), or single when defining .proto files (proto3). Setting a field to required is not an option in proto3, and it is strongly discouraged in proto2 [21].

V. DETECTING APPLICATIONS VULNERABILITIES METHOD USING GRPC

The stages of processing and interpreting network traffic packets are depicted in Fig. 3. A general framework for detecting application vulnerabilities using gRPC is shown in Fig. 4. There are two basic messaging strategies: changing the values of one field or all fields in one loop. There's also the situation where a message field's value is fixed and cannot be modified.

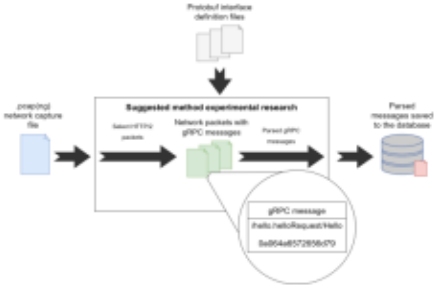


Fig. 3. The stages of processing and interpreting network traffic packets

Both of the preceding solutions can be used in this scenario, but only if the required fields are left intact (see Fig. 4.). In the settings, you can define the messaging technique you want to employ.

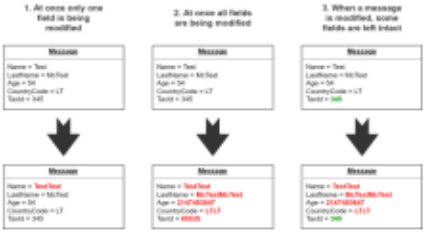


Fig. 4. A general framework for detecting application vulnerabilities using gRPC

The premise remains the same for both change techniques when it comes to fields modifications. The numeric message fields are modified by altering the values in the message using fuzzy logic. The range of substituted values for numeric fields is divided into value types and ranges (see TABLE I.)

TABLE I. THE RANGE OF SUBSTITUTED VALUES FOR NUMERIC FIELDS

Value type	Value range	
	The smallest possible value	*The largest possible value*
bool	0	1
string	min length = null	max length = 2^{32}
int32, sint32 sfixed32	-2147483648	2147483647
uint32, fixed32	0	4294967295
int64, sint64 sfixed64	-9223372036854775808	9223372036854775807
uint64, fixed64	0	18446744073709551615
float	1.175494351 E - 38	3.402823466 E + 38
double	2.2250738585072014 E - 308	1.7976931348623158 E + 308

The method for detecting vulnerabilities in applications using gRPC starts with scanning the initial remote procedure messages (see Fig. 5). The method will accept data that can be retrieved using the Tcpdump or Wireshark network packet analyzer from .pcap or .pcapng files. The method will also accept Protobuf files .proto, which are used to filter out unnecessary messages and send messages to the application under test. Because protobuf messages are utilized in the gRPC remote procedure call framework, which is based on the HTTP/2 protocol [22], protobuf messages must be requested in all HTTP/2 protocol requests. After reviewing the contents of the HTTP/2 request, it is determined whether this message is intended for at least one of the services described in the .proto files of the tested software. The data is saved if the message has a service match. If no match is detected, the algorithm repeats the process with a new HTTP/2 request. Protobuf messages in binary format are extracted from these queries, which were constructed using the protocol buffer's interface description language [21].

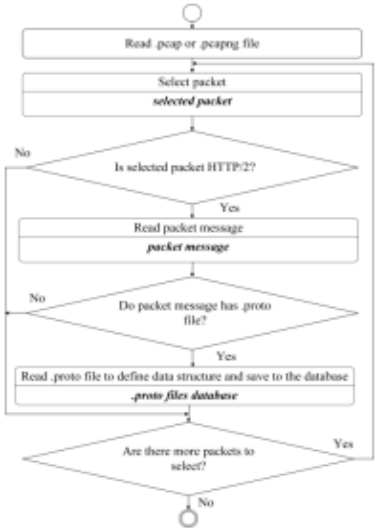


Fig. 5. Packages scanning process for extracting remote procedure messages

After all remote procedure calls, messages structures, and data types are saved to the database, the process of detecting applications vulnerabilities using gRPC starts. The process of the proposed method is depicted in Fig. 6. Starting vulnerability detection, .proto file, messages structure, and data types uploaded from the database. The execution monitoring procedure and the application under test are both started. The gRPC message creator using fuzzy logic changes values of the message data accordingly to the types and possible values given in TABLE I. . The message with the highest expected number of message change cycles is chosen in the first iteration, and changed values of the message data is constructed based on it. The messages are created in subsequent cycles depending on the execution progress and the tested application replies to the gRPC sent messages. The received response is sent on to the process for creating the test report. The application is being tested if it is still running if no reply is received. Verification of the tested application progress is sent to the report generating procedure. A new test iteration is started after the gRPC message generating process receives the execution status and response data from the application under test.

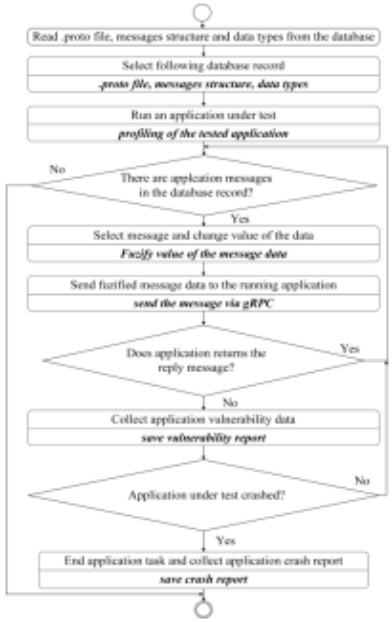


Fig. 6. The process of proposed method for detecting applications vulnerabilities using gRPC

Application activity monitoring process detects the tested application fault (no response) the crash report process collects all relevant fault data and saves the application crush report.

VI. Evaluation Framework and Experimental Setup

A general framework for evaluation of the proposed method for detecting applications vulnerabilities using gRPC depicted in Fig. 7.

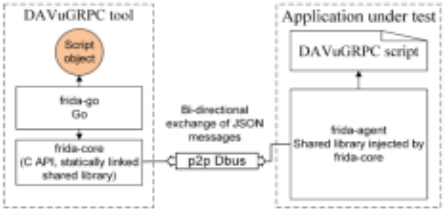


Fig. 7. A general framework for evaluation of the proposed method

The Frida dynamic analysis library is used to track the application under test execution. In order to use the library programming interface in the Go programming language, we use the frida-go library, which allows us to use the Frida library's needed functions. The Frida library inserts additional

code during execution that permits JavaScript to be performed after enabling the application under test execution. These scripts have full access to the application under test memory and can also change how functions are executed.

When a method in the application under test is called in the DAVuGRPC tool, the script begins to capture blocks of executed method instructions. The Frida Library's Interceptor and Stalker development API were used to do this. The completed instruction blocks are transmitted to the DAVuGRPC tool at the end of the application under test method. In addition to this information, the application the under test method's execution time is recorded. Data from the application under the test is sent using the Frida library's p2p Dbus communication channel, which allows data to be exchanged between the DAVuGRPC tool and the application under test script code. This p2p Dbus channel is also used when JavaScript scripting methods are invoked.

Basic information regarding the testing process is represented in Fig. 8.

Fig. 8. DAVuGRPC tool basic information regarding the testing process

The user can see the terminal interface after configuring and running the DAVuGRPC tool, which displays three main blocks: information on the time and duration of the test process (Timing), the overall results of the test process (Overall results), and the current progress of the test process (Progress).

VII. Experimental Results

Our experiments are performed using AMD Ryzen 5 2600 processor with six physical and twelve logical cores @ 3.40GHz; 16 GB RAM; Windows 10 Pro 64 bits OS; proto-fuzzer v. 0.0.3.

For the experimental investigation, a testing platform was created with applications written in the C++ programming language that uses gRPC. There have been twenty-three remote procedures implemented: ten procedures has various types of vulnerabilities and thirteen without any vulnerability. All procedures are categorized into three groups:

- Procedures without complex alternative execution path;
- Procedures with alternative execution paths that can perform different operations depending on the input data;
- Related procedures that have established enforcement procedures, i.e. procedure no. 1 must be completed before procedure no. 2.

In order to use the proto-fuzzer tool in the experimental study, a simple JavaScript script was used that uses the code of the proto-fuzzer tool. This was done because the tool itself only provides the functionality to create and send remote procedures, but the entire code must be applied individually to each message. The generated script with the maximum limit of 1000 times created new remote procedure messages and sent them to the application under test.

The proposed method was used to construct sample message flow files for remote processes in the experimental study. Wireshark network traffic monitor was used to collect these files. While executing susceptible remote procedures with non-vulnerable inputs, this program recorded network traffic. For each susceptible procedure, a network traffic (.pcapng) file was created. The next stage was to get the addresses (offsets) of the methods in the application under test, which would be utilized in the DAVuGRPC tool.

The results of the experiments are summarized in Table II. Based on these results we can evaluate that the proposed method detects stack-based, heap-based, and null pointer vulnerabilities in the short time sending a small number of gRPC messages. proto-fuzzer did not manage to find any vulnerabilities. It was not included in the results.

TABLE II. Evaluation of the DAVuGRPC tool

Procedures names	**Results of the proposed method**	
	Vulnerability definition	***No. of messages / Detection time***
MethodOneBad	Stack-based buffer overflow when input is longer than 240 characters	5 / 5 sec.
MethodTwoBad	Heap-based buffer overflow when input contains Unicode characters string	4 / 6 sec
MethodThreeBad	Stack-based buffer overflow when input is longer than 256 characters and input length is greater than 256 characters	2 / 4 sec.
MethodFourBad	Null pointer reference when malloc is called with input value that can overflow memory allocation	3 / 5 sec.
MethodFiveBad	Null pointer reference when malloc is called with invalid double input value which is casted to the integer	5 / 4 sec.
MethodSixBad	Stack-based buffer overflow when input is longer than 240 symbols and input enumeration value is 2	6 / 5 sec.
MethodSevenBad	Heap-based buffer overflow when smaller memory is allocated and than higher input value is used for writing to the buffer	7 / 5 sec.
MethodEightBad	Null pointer reference when malloc is called with negative input value	7 / 5 sec.
MethodNinePartTwoBad	A heap-based buffer overflow when input is longer than 500 symbols	16 / 6 sec.
MethodTenPartThreeBad	A heap-based buffer overflow when input is longer than 500 symbols	77 / Vulnerability not detected

VIII. Conclusion

The goal of gRPC is to speed up data transmission between microservices. It also describes the RPC API paradigm in an interface description language (IDL), which makes determining remote operations easier. The main results of this paper are as follows:

- gRPC could be successfully used applications vulnerabilities detection;
- To track running application under test and prepare vulnerability report the Frida dynamic analysis library will be used.
- Proposed applications vulnerabilities method using remote procedure calls and realized DAVuGRPC tool shows acceptable results for stack-based, heap-based, and null pointer vulnerabilities with the short time whereas the small number of gRPC messages has been sent.

Future work will be as follows:

- Add nested messages value fuzzing.
- Implement complex fuzzification logic with recognition dependencies between the same values in the messages.
- Add additional dynamic instrumentation framework support since the current Frida implementation is unstable.
- Add compressed gRPC messages support.

References

[1] W. Jimenez , A. Mammar and A. Cavalli, "Software Vulnerabilities, Prevention and Detection Methods: A Review", July 2010. PenTest Magazine [Online]. Available: http://www-lor.int-evry.fr/~anna/files/sec-mda09.pdf [Accessed February 16, 2022]

[2] S. Garg, R.K. Singh and A.K. Mohapatra "Analysis of software vulnerability classification based on different technical parameters" 2019. Information Security Journal: A Global Perspective, 28:1-2, pp. 1-19, doi: https://doi.org/10.1080/19393555.2019.1628325

[3] J. Fan, Yi. Li, S. Wang, and T. N. Nguyen. "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries" 2020. Proceedings of the 17th International Conference on Mining Software Repositories. Association for Computing Machinery, New York, NY, USA, pp. 508–512. Doi: https://doi.org/10.1145/3379597.3387501

[4] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood and M. W. McConley. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning." 2018. 17th IEEE International Conference on Machine Learning and Applications (ICMLA'2018) pp. 757-762. Doi: https://doi.org/10.1109/ICMLA.2018.00120

[5] J. Fell, "A Review of Fuzzing Tools and Methods", March 10, 2017. PenTest Magazine [Online]. Available: https://wcventure.github.io/FuzzingPaper/Paper/2017_review.pdf [Accessed February 16, 2022]

[6] O. Zaazaa and H. El Bakkali, "Dynamic vulnerability detection approaches and tools: State of the Art," 2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS), 2020, pp. 1-6, doi: https://doi.org/10.1109/ICDS50568.2020.9268686

[7] K. Yang, H. Zhao, C. Zhang, J. Zhuge and H. Duan, "Fuzzing IPC with Knowledge Inference," 2019 38th Symposium on Reliable Distributed Systems (SRDS), 2019, pp. 11-1109, doi: https://doi.org/10.1109/SRDS47363.2019.00012

[8] H. Bagci, and A. Kara, "A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication", 2016. In Proceedings of the 11th International Joint Conference on Software Technologies - ICSOFT-EA, (ICSOFT 2016) ISBN 978-989-758-194-6, pages 117-124. doi: https://doi.org/10.5220/0005931201170124

[9] T. Hussain, S. Satyaveer, and M. Seth, "A Comparative Study of Software Testing Techniques Viz. White Box Testing Black Box Testing and Grey Box Testing." IJAPRR International Peer Reviewed Refereed Journal, Vol. II, Issue V, 2015.

[10] C. Chen, C. Baojiang, M. Jinxin, W. Runpu, G. Jianchao and L. Wenqian, "A systematic review of fuzzing techniques" 2018 Computers & Security, Volume 75, pp. 118-137, ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2018.02.002

[11] S. Karamcheti, G. Mann and D. Rosenberg, "Adaptive Grey-Box Fuzz-Testing with Thompson Sampling" 2018 In Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security (AISec '18). Association for Computing Machinery, New York, NY, USA, pp. 37–47. doi: https://doi.org/10.1145/3270101.3270108

[12] M. Mouzarani, B. Sadeghiyan and M. Zolfaghari, "A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes," 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC), 2015, pp. 42-49, doi: https://doi.org/10.1109/PRDC.2015.10

[13] Z. Spasov, D. Bogdanova, and M. Skopje, "Inter-Process Communication, Analysis, Guidelines And Its Impact On Computer Security" 2010 The 7th International Conference for Informatics and Information Technology (CIIT 2010). Institute of Informatics. Available: http://ciit.finki.ukim.mk/data/papers/7CiiT/7CiiT-11.pdf [Accessed March 5, 2022]

[14] N. C. Will, T. Heinrich, A. B. Viescinski and C. A. Maziero, "Trusted Inter-Process Communication Using Hardware Enclaves," 2021 IEEE International Systems Conference (SysCon), 2021, pp. 1-7, doi: https://doi.org/10.1109/SysCon48628.2021.9447066

[15] D. Hamed, "Inter-Process Communication (IPC) in Distributed Environments: An Investigation and Performance Analysis of Some Middleware Technologies" 2020. International Journal of Modern Education & Computer Science. Vol. 12 Issue 2, pp. 36-52. doi: https://doi.org/10.5815/ijmecs.2020.02.05

[16] L.Vilanova, M. Jordà, N. Navarro, Y. Etsion, and M. Valero. "Direct Inter-Process Communication (dIPC): Repurposing the CODOMs Architecture to Accelerate IPC" 2017. In Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17). Association for Computing Machinery, New York, NY, USA, pp. 16–31. doi: https://doi.org/10.1145/3064176.3064197

[17] N. Koutroumpouchos, G.Lavdanis, E. Veroni, C. Ntantogian, and C. Xenakis. "ObjectMap: detecting insecure object deserialization" 2019. In Proceedings of the 23rd Pan-Hellenic Conference on Informatics (PCI '19). Association for Computing Machinery, New York, NY, USA, pp. 67–72. doi: https://doi.org/10.1145/3368640.3368680

[18] M. Berga, A. Santos, "gRPC vs REST: comparing APIs architectural styles" June 03, 2021. Imaginary Cloud [Online]. Available: https://www.imaginarycloud.com/blog/grpc-vs-rest/ [Accessed March 12, 2022]

[19] gRPC a high performance, open source universal RPC framework. [Online]. Available: https://grpc.io/ [Accessed March 12, 2022]

[20] Protocol Buffers Overview. [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview [Accessed March 12, 2022]

[21] Protocol Buffers Language Guide. [Online]. Available: https://developers.google.com/protocol-buffers/docs/proto#specifying-rules [Accessed March 12, 2022]

[22] Internet Engineering Task Force, Hypertext Transfer Protocol Version2 (HTTP/2). [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7540 [Accessed March 12, 2022]