KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
INFORMACIJOS IR INFORMACINIŲ TECHNOLOGIJŲ SAUGA

ANDRIUS ZONYS

STATINĖS KODO ANALIZĖS ĮRANKIŲ TYRIMAS IR TOBULINIMAS

Magistro baigiamasis darbas

Darbo vadovas
dr. J. Čeponis

KAUNAS, 2013

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
INFORMACIJOS IR INFORMACINIŲ TECHNOLOGIJŲ SAUGA

ANDRIUS ZONYS

STATINĖS KODO ANALIZĖS ĮRANKIŲ TYRIMAS IR TOBULINIMAS

Magistro baigiamasis darbas

Darbo vadovas
dr. J. Čeponis

Recenzentas
doc. dr. T. Blažauskas

KAUNAS, 2013

AUTORIŲ GARANTINIS RAŠTAS
DĖL PATEIKIAMO KŪRINIO

2013 - 05 - 22 d.

Kaunas

Autoriai, Andrius Zonys
(vardas, pavardė)

_____,

patvirtina, kad Kauno technologijos universitetui pateiktas baigiamasis bakalauro (magistro) darbas (toliau vadinama – Kūrinys) „Statinės kodo analizės įrankių tyrimas ir tobulinimas“
(kūrinio pavadinimas)

pagal Lietuvos Respublikos autorių ir gretutinių teisių įstatymą yra originalus ir užtikrina, kad

1) jį sukūrė ir parašė Kūrinyje įvardyti autoriai;
2) Kūrinys nėra ir nebus įteiktas kitoms institucijoms (universitetams) (tiek lietuvių, tiek užsienio kalba);
3) Kūrinyje nėra teiginių, neatitinkančių tikrovės, ar medžiagos, kuri galėtų pažeisti kito fizinio ar juridinio asmens intelektinės nuosavybės teises, leidėjų bei finansuotojų reikalavimus ir sąlygas;
4) visi Kūrinyje naudojami šaltiniai yra cituojami (su nuoroda į pirminį šaltinį ir autorių);
5) neprieštarauja dėl Kūrinio platinimo visomis oficialiomis sklaidos priemonėmis.
6) atlygins Kauno technologijos universitetui ir tretiesiems asmenims žalą ir nuostolius, atsiradusius dėl pažeidimų, susijusių su aukščiau išvardintų Autorių garantijų nesilaikymu;
7) Autoriai už šiame rašte pateiktos informacijos teisingumą atsako Lietuvos Respublikos įstatymų nustatyta tvarka.

Autoriai

Andrius Zonys	_____
(vardas, pavardė)	(parašas)
_____	_____
(vardas, pavardė)	(parašas)
_____	_____
(vardas, pavardė)	(parašas)
_____	_____
(vardas, pavardė)	(parašas)

SANTRAUKA

Šiame darbe aptariama statinė ir dinaminė kodo analizė, jų privalumai ir trūkumai, analizės įrankių tipai bei jų paskirtis. Analizuojami „Gendarme“, „Cppcheck“ ir „FindBugs“ statinės kodo analizės įrankiai. Plačiau analizuojamas laisvai platinamas statinės kodo analizės įrankis „Gendarme“. Pateikiamos spragos, rastos esamose „Gendarme“ taisyklėse bei siūlomi jų patobulinimai, kurie turėtų padėti aptikti daugiau klaidų ir padidinti programų spartą, stabilumą, saugumą ir kodo skaitomumą. Taip pat pateikiamos naujos sukurtos taisyklės, kurios turėtų padėti aptikti daugiau perteklinio, nelogiško kodo, kurio kompiliatorius neaptinka. Naujos taisyklės turėtų padėti supaprastinti programos kodą ir taip pagerinti jo skaitomumą bei aptikti daugiau spragų, kuriomis pasinaudojus galima įtakoti programų veikimą. Atliktas eksperimentinis tyrimas, kurio metu buvo išanalizuotos programos su patobulintu ir nepatobulintu „Gendarme“ statinės kodo analizės įrankiu. Pateikiami eksperimento rezultatai ir išvados.

SUMMARY

This paper discusses the static and dynamic code analysis, their advantages and disadvantages, types of analysis tools and their purpose. Analyzes the "Gendarme", "Cppcheck" and "FindBugs" static code analysis tools. Wider analyzes the freely distributed static source code analysis tool "Gendarme". Presented gaps which were found in existing "Gendarme" rules and the proposed modifications, which should help to detect more errors and improve program performance, stability, security and code readability. It also introduces new rules, which should help to detect more excessive, illogical code, which are not detected by the compiler. As well as simplify the code and thus to improve its readability and detect more vulnerabilities, which may be used to affect the functioning of applications. Experimental research was carried out in which some programs were analyzed with improved and not improved static code analysis tool "Gendarme". Experimental results and conclusions are presented.

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLŲ SĄRAŠAS

TERMINŲ IR SANTRUMPŲ ŽODYNAS

STL (Standard Template Library) – tai C++ biblioteka, kuri turi daug pagrindinių algoritmų ir duomenų struktūrų.

LINQ (Language-Integrated Query) – tai patobulinimų rinkinys skirtas Visual Studio programai. Jis papildo užklausų naudojimo galimybes C# programavimo kalboje.

CLR (Common Language Runtime) – tai .NET Framework virtualus mašinos komponentas atsakingas už .NET programų vykdymą.

DLL (Dynamic-link library) – tai Microsoft sukurta bendrų bibliotekų idėja skirta Microsoft Windows.

API (Application Programming Interface) – tai kodo specifikacija skirta programos komponentams kaip bendravimo sąsaja.

ECMA (European Computer Manufacturers Association) – tai tarptautinė, privati, ne pelno siekianti organizacija, kurios objektas yra informacijos ir komunikacijos sistemos.

CIL (Common Intermediate Language) – tai žemiausio lygmens žmogaus skaitomo programinio kodo kalba apibrėžta CLI reikalavimais.

CLI (Common Language Infrastructure) – tai atviri reikalavimai sukurti Microsoft, kurie atitinka ISO standartus. Jie aprašo vykdomo kodo ir aplinkos pagrindus, kuriuos naudoja .NET Framework ir Mono.

Cecil yra biblioteka skirta sugeneruoti ir kontroliuoti programas ir bibliotekas ECMA CIL formatu.

HTML (Hyper Text Markup Language) – tai kompiuterinė hiperteksto žymėjimo kalba, naudojama pateikti turinį internete.

XML (Extensible Markup Language) – tai bendros paskirties duomenų struktūrų bei jų turinio aprašomoji kalba.

SWF (System Windows Forms) – tai .NET Framework klasių rinkinys skirtas vartotojo sąsajai kurti.

XSS (Cross-site scripting) – tai IT sistemų pažeidžiamumas, dažniausiai aptinkamas tinklalapiuose, kuris leidžia įterpti papildomą programinį kodą į vartotojų peržiūrimą puslapį.

SQL (Structured Query Language) – populiariausia iš šiuo metu naudojamų kalbų, skirtų aprašyti duomenis ir manipuliuoti jais reliacinių duomenų bazių valdymo sistemose.

JRE (Java Runtime Environment) – tai Java darbinė terpė skirta vykdyti Java aplikacijas minimaliais reikalavimais.

JDK (Java Development Kit) – tai Java programavimo įrankių rinkinys.

ĮVADAS

Kompiuteris tampa neatskiriama mūsų gyvenimo dalimi. Jaunesniems žmonėms jau tampa sunku įsivaizduoti pasaulį be kompiuterių. Viena svarbiausių kompiuterio dalių yra programinė įranga. Dėl to programų skaičius sparčiai auga. Kuriama daug specializuotų ir nespecializuotų programų verslui ir žmonėms. Ir čia susiduriama su programų saugumo, programinio kodo optimizavimo ir programavimo klaidų problema, kurią šiame darbe bandoma spręsti. Darbas priklauso „Informacijos ir informacinių technologijų sauga“ studijų programai.

Darbo problematika ir aktualumas

Kai kurioms programoms saugumas yra nelabai svarbus, tačiau daugeliui, ypač verslui skirtoms programoms, jis yra labai svarbus. Informacijos paviešinimas, jos sugadinimas ar paslaugos nutraukimas gali privesti įmonę net iki bankroto. Atskleistomis verslo paslaptimis gali greit pasinaudoti konkurentai. Atskleidus vartotojų privačius duomenis ar esant blogai paslaugos kokybei, vartotojai praras pasitikėjimą įmone.

Saugumas bei kokybė yra labai brangus dalykas, ypač, jei programinis kodas yra ilgas [1]. Apie 50 % programų sukūrimo kainos sudaro testavimas ir derinimas [2], todėl ne visos, ypač mažosios įmonės, gali sau tai leisti. Jos taupo saugumo sąskaita ir tikisi, kad į jų sistemas nebus įsibrauta. Tačiau yra gana nebrangių ar net nemokamų programinių priemonių, padedančių užtikrinti geresnį kodo saugumą ir kokybę. Jos atlieką statinę ir dinaminę kodo analizę. Statinei ir dinaminiai kodo analizei yra sukurta nemažai įrankių, tačiau joks įrankis ar būdas negali užtikrinti visiško saugumo.

Darbo tikslas ir uždaviniai

Tyrimo tikslas – išanalizuoti esamus statinės kodo analizės įrankius ir pasiūlyti galimus patobulinimus. Šiam tikslui įgyvendinti iškelti uždaviniai:

1. Išanalizuoti esamus statinės kodo analizės įrankius.
2. Pasiūlyti pačių įrankių ar naudojamų algoritmų patobulinimus.
3. Praktiškai įgyvendinti šiuos pasiūlymus.
4. Ištirti ir įvertinti įgyvendintų patobulinimų pliusus ir minusus.

Tam, kad būtų galima įgyvendinti visus uždavinius, bus analizuojami tik atviro kodo statinės kodo analizės įrankiai.

Darbo struktūra

Šiame darbe išanalizuoti trys statinės kodo analizės įrankiai. Pasirinktas vienas įrankis ir jam siūlomi įvairūs patobulinimai, kad jis aptiktų daugiau kodo klaidų ir pasiūlytų kaip jas ištaisyti bei pagerinti kodo saugumą. Praktiškai įgyvendinti pateikti pasiūlymai ir ištirtas patobulintas įrankis. Nustatyta kuo nauji patobulinimai yra geresni ar blogesni.

1. STATINĖ KODO ANALIZĖ IR JOS ĮRANKIAI

Kodo analizės metodus galima suskirstyti į tris pagrindines kategorijas. Tai rankinis metodas, kuomet kodas peržiūrimas žmogaus. Statinis, kai atliekamas automatizuotas kodo tikrinimas jo nevykdant ir dinaminis, kuomet kodas tikrinamas programos veikimo metu. Plačiau apie statinę ir dinaminę kodo analizę aprašyta sekančiuose skyreliuose.

1.1. Statinė kodo analizė

Tai programinės įrangos kodo peržiūra jo nevykdant. Peržiūros metu nagrinėjamas programos kodas ir pagal žinomus klaidų šablonus bandoma aptikti klaidingas vietas.

Statinė kodo analizė gali būti atlikta gana anksti programavimo procese, kadangi ji atliekama nevykdant kodo. Programuotojai naudodami statinę kodo analizę iš karto po to kai parašomas kodas, gali aptikti ir išspręsti klaidas dar prieš modulių (angl. *unit*) ir integracijos testavimą. Kuo anksčiau aptinkama klaida, tuo pigiau kainuoja ją pataisyti. Tai pagrindinis statinės kodo analizės privalumas [3].

Statinės kodo analizės įrankiai turi aptikti klaidas programiniame kode ir pranešti apie jų vietas. Dauguma klaidų yra suskirstytos į grupes. Įrankiai gali ne tik pranešti apie klaidą bet ir pasiūlyti kaip ją ištaisyti. Kai kurie iš jų gali suformuoti išsamias ataskaitas apie aptiktas klaidas [4].

Įrankiai turi turėti būdą kaip ignoruoti klaidų pranešimus, nes jie gali pranešti apie klaidas, kurių iš tiesų nėra. Tokiu atveju reikia ignoruoti pranešimą parašant tam tikrą sąlygą prie tariamos klaidos, kurią supranta įrankis [4].

1.1.1. Statinės kodo analizės įrankiai

Statinės kodo analizės įrankiai būna skirtingų tipų. Pagal užduotis, kurias jie atlieka, juos galima suskirstyti į 3 kategorijas:

1. Aptinkantys klaidas programose. Šiai kategorijai priskiriami įrankiai aptinkantys tokias klaidas programose, kurios gali įtakoti programos veikimą arba kuriomis pasinaudojus yra galimybė jį pakeisti.
2. Aptinkantys netinkamą kodo formatavimą. Šio tipo įrankiai tikrina ar parašytas kodas atitinka įmonės kodo formatavimo standartus. Pavyzdžiui, tikrinama ar tinkamas eilučių atitraukimas, tikrinami tarpai tarp skyrybos, aritmetinių ženklų ir pan. Jie neaptinka realių klaidų, kurios įtakoja programos veikimą.
3. Skaičiuojantys programinio kodo metrikas. Šie įrankiai leidžia gauti tam tikrą programinės įrangos ar jos specifikacijų skaitmeninę vertę. Yra daug įvairių rodiklių, kurie gali būti apskaičiuoti šių įrankių pagalba.

Pagal analizuojamą kodą, statinės kodo analizės įrankius galima suskirstyti į dvi kategorijas. Vieni analizuoja tiesioginį kodą, kiti – sukompiliuotą baitų kodą. Ir vieni, ir kiti turi savo pliusų. Kuomet analizuojamas tiesioginis programuotojo parašytas kodas, kodas yra neiškraipomas prieš analizę. Tuo tarpu kompiliatoriaus sukompiliuotas ir optimizuotas dvejetainis kodas gali skirtis nuo originalaus. Tačiau dvejetainio kodo analizė yra kur kas spartesnė, o sparta labai svarbi analizuojant didelius kodo eilučių kiekius [5].

Nors įrankių tipai ir skiriasi, tačiau jie turi tuos pačius pagrindinius bruožus. Jie skaito kodą ir sukuria jo abstraktų modelį. Po to pagal klaidų sąrašus ir jų šablonus ieško atitinkančių vietų. Taip pat įrankiai atlieka ir duomenų srautų analizę. Bando numatyti reikšmes, kurias kintamasis gali įgyti tam tikrose kodo vietose.

Žmogaus klaidos yra kažkiek nuspėjamos, tačiau įrankiai negali aprašyti visų įmanomų klaidų. Dėl to dauguma įrankių leidžia patiems programuotojams apsirašyti taisykles skirtas klaidoms aptikti. Tokiu atveju, jei programuotojas yra linkęs daryti tam tikro tipo klaidas, jis gali susikurti taisykles aptinkančias tas klaidas.

1.1.2. Statinės kodo analizės stiprybės ir silpnybės

Kiekvienas įrankis turi sąrašą taisyklių, pagal kurias ieško kodo klaidų. Dauguma produktų turi galimybę pridėti papildomų taisyklių. Gali būti atlikta neišbaigto kodo analizė (neįgyvendinti visi metodai ar nesukurtos visos klasės). Kuo kodas labiau išbaigtas, tuo geriau įrankis gali aptikti klaidas [6].

Kartais testavimui reikalingi pradiniai duomenys. Testavimas taip pat gali reikalauti išorinių įrankių pagalbos. Dėl šių priežasčių kartais nepatogu naudoti statinės kodo analizės įrankius. Šie įrankiai neturi pakeisti testavimo, o tik jį papildyti.

Įrankiai gali aptikti retai atsitinkančias klaidas ar taip vadinamas atgalines duris (angl. *hidden back doors*). Jie analizuoja kodą jo nevykdant, dėl to gali išskaičiuoti kur kas daugiau ryšių tarp skirtingų modulių ir komponentų.

Vykdyti testus yra paprasta, tačiau parašyti testą, kuris pilnai ištestuotų kokį nors modulį nėra lengva. Kuomet aptinkamos naujos spragos, tuomet nauji testai turi būti parašyti joms aptikti. Jei aptinkamų klaidų sąrašas bus neatnaujinamas, tai įrankis neaptiks naujausių saugumo spragų.

1.2. Dinaminė kodo analizė

Priešingai nei statinė kodo analizė, dinaminė kodo analizė analizuoja kodą jo vykdymo metu. Nors ši technika nėra labai išsami, tačiau ji turi didelį privalumą, ji vienintelė analizuoja kodą jo veikimo metu. Tokiu būdu dinaminė kodo analizė padeda aptikti bandymus pakenkti programai ir aptikti besimodifikuojančias programas. Taip pat šio tipo įrankiai gali ištestuoti programą įvairiose aplinkose [7]. Tam, kad dinaminė kodo analizė būtų efektyvi, reikia ganėtinai didelio testų kiekio.

1.2.1. Dinaminės kodo analizės įrankių paskirtis ir tipai

Dinaminės kodo analizės įrankius galima suskirstyti į tris pagrindinius tipus, tai analizuojantys:

1. Atmintį. Atminties klaidas yra gana sunku aptikti. Šio tipo įrankius geriausia naudoti, kai problemą ne visada įmanoma atkartoti ar atminties naudojimas per daug, ar per greit išauga. Atminties spragos dažnai naudojamos programišių perimant programų kontrolę.
2. Kodo padengimą. Įrankiai apskaičiuoja kokia kodo dalis ar kokios funkcijos buvo ištestuotos. Naudojamas baltos dėžės testavimas. Naudojant kodo padengimą, lengviau aptikti neištestuotas kodo vietas ir užtikrinti geresnę testavimo kokybę.
3. Veikimo spartą. Šio tipo įrankiai matuoja kiek kartų buvo iškviesta viena ar kita funkcija bei kiek laiko funkcija buvo vykdoma. Naudojant šio tipo įrankius yra lengviau aptikti silpnąsias vietas (angl. *bottleneck*) bei pagerinti programos spartą.

Dinaminės kodo analizės įrankių yra daug ir įvairių. Yra mokamų, tokių kaip „JetBrains dotTrace“, „ANTS Memory Profiler“ ir laisvai platinamų, tokių kaip „EQATEC Profiler“, „SlimTune“. Tačiau rasti gerą ir laisvai platinamą dinaminės kodo analizės įrankį yra kur kas sunkiau nei gerą nemokamą statinės kodo analizės įrankį.

1.2.2. Dinaminės kodo analizės stiprybės ir silpnybės

Dinaminės kodo analizės privalumas yra tas, kad ji analizuoja kodą programos veikimo metu ir dėl to gali aptikti daugiau įvairių klaidų. Taip pat kai kurie dinaminės kodo analizės įrankiai leidžia pasirinkti ką analizuoti, o ką praleisti. Šio tipo analizę galima atlikti įvairioms programoms. Tačiau pagrindinė šios tipo analizės problema yra tai, kad sunku nusakyti tikslią pažeidžiamumo vietą kode. Dėl to tai reikalauja daugiau programuotojo pastangų bei užtrunkama daugiau laiko taisant rastą problemą.

1.3. Statinės kodo analizės įrankiai

Statinės kodo analizės atlikimui dažniausiai naudojami įvairūs įrankiai. Vieni įrankiai yra nemokami ir juos vartotojai gali modifikuoti ir pritaikyti savo reikmėms. Mokami įrankiai dažniausiai pasižymi geru techniniu palaikymu ir patogesne grafine vartotojo sąsaja. Statinė programinio kodo analizė gali būti realizuota kaip atskira programa arba integruojama į programavimo aplinką. Integruoti įrankiai leidžia atlikti kodo analizę tame pačiame lange, kuriame rašomas ir pats kodas. Tai padeda programuotojui paprasčiau ir anksčiau aptikti klaidas ir jas ištaisyti. Taip pat neleidžia išsisukti nuo tokių klaidų, nes kodo nepavyks sukompiliuoti, kol nebus ištaisytos visos klaidos.

Šiame skyrelyje analizuojami trys statinės kodo analizės įrankiai. Nagrinėjamos jų aptinkamos klaidos, rezultatų pateikimo būdai, integravimo ir naudojimo galimybės.

1.3.1. „Gendarme“ įrankis

„Gendarme“ įrankis yra atviro kodo, parašytas C# programavimo kalba. Todėl jis būtų tinkamas tolesniam tobulinimui. Turi aiškias taisykles, kurios svarbios testavimo metu. Nesuprasta taisyklė dažnai reiškia ignoruojamą klaidą arba dar blogiau, rasta klaida traktuojama kaip klaidingas pavojaus signalas [8].

„Gendarme“ [9] yra plečiamas, taisyklėmis paremtas įrankis skirtas aptikti klaidas .NET programose bei bibliotekose. „Gendarme“ analizuoja programas ir bibliotekas parašytas ECMA CIL [10] formatu ir ieško dažniausiai pasitaikančių programavimo klaidų, bei klaidų, kurių kompiliatorius įprastai neaptinka ar nebando aptikti. „Gendarme“ kodo analizei naudoja „Cecil“ biblioteką. Ši biblioteka, kaip ir pats įrankis, parašyti C# programavimo kalba.

Sebastien Pouliot sukūrė „Gendarme“ tam, kad palengvinti Mono[1] testavimą. Vėliau Aaron Tomb papildė „Gendarme“, kad būtų galima aptikti klaidas .NET programose. Nuo tada pagalbininkai ir taisyklės vis auga [9].

„Gendarme“ taisyklės, kurios skirtos kodo analizei yra naudojamos per paleidėjus (angl. *runners*). Tai programa atsakinga už taisyklių užkrovimą, analizę ir klaidų pateikimą. „Gendarme“ šiuo metu turi du skirtingus paleidėjus. Vienas paremtas komandinės eilutės sąsaja, kitas – SWF grafine sąsaja.

Komandinės eilutės paleidėjas gali išvesti rezultatus į komandinę eilutę, XML failus ar tinkamai suformuotus HTML failus.

SWF grindžiamas paleidėjas leidžia pažingsniui pasirinkti modulius (angl. *assembly*), taisykles, nustatymus ir atvaizduoti analizės rezultatus kaip ir komandinės eilutės paleidėjas. Tam, kad būtų paprastesnis, jis turi mažiau nustatymų, nei komandinės eilutės paleidėjas.

1.3.1.1. Įrankio taisyklės

„Gendarme“ statinės kodo analizės įrankis yra modulinis ir užkrauna taisykles iš tam tikrų sričių. Nėra jokių įkoduotų (angl. *hard-coded*), statinių taisyklių, kurios visuomet naudoja tas pačias reikšmes, nors jos gali kisti skirtingose programose.

Komandinės eilutės paleidėjas užkrauna visas taisykles, kurios yra nurodytos rules.xml konfigūravimo faile. Šis failas gali būti koreguojamas pagal kiekvieno vartotojo poreikius. Vediklinis (angl. *wizard*) paleidėjas parodys visas taisykles iš kiekvieno modulio esančio įdiegimo kataloge.

Šiuo metu taisyklės suskirstytos į tokias kategorijas:

- Gendarme.Rules.BadPractice
- Gendarme.Rules.Concurrency
- Gendarme.Rules.Correctness
- Gendarme.Rules.Design
- Gendarme.Rules.Design.Generic

[1] Mono – tai programinės įrangos platforma skirta palengvinti nuo platformos nepriklausomų programų kūrimą.

- Gendarme.Rules.Design.Linq
- Gendarme.Rules.Exceptions
- Gendarme.Rules.Interoperability
- Gendarme.Rules.Maintainability
- Gendarme.Rules.Naming
- Gendarme.Rules.Performance
- Gendarme.Rules.Portability
- Gendarme.Rules.Security
- Gendarme.Rules.Security.Cas
- Gendarme.Rules.Serialization
- Gendarme.Rules.Smells
- Gendarme.Rules.Ui

Šios kategorijos toliau nagrinėjamos 1.3.1.1.x skyreliuose.

1.3.1.1.1. Gendarme.Rules.BadPractice

Gendarme.Rules.BadPractice kategorijoje nagrinėjamos įvairaus tipo problemos, kurios nebūtinai sukels klaidų, tačiau yra geriau tą patį funkcionalumą įgyvendinti kitaip ar visiškai jo atsisakyti.

Kategorijos taisyklės:

AvoidAssemblyVersionMismatchRule. Ši taisyklė patikrina ar *[AssemblyVersion]* atributo reikšmė lygi *[AssemblyFileVersion]* atributo reikšmei kuomet abu yra aprašyti tame pačioje modulyje. Esant skirtingiems versijų numeriams galimi neaiškumai.

AvoidCallingProblematicMethodsRule. Ši taisyklė įspėja kuomet kreipiamasi į potencialiai pavojingą .NET Framework funkciją. Jei įmanoma, reikia pabandyti išvengti tokio API arba bent jau įsitikinti, kad kodas gali būti saugiai kviečiamas iš kitur.

AvoidVisibleConstantFieldRule. Ši taisyklė ieško pastovių kintamųjų, kurie yra matomi už modulio ribų. Jei tokie laukai naudojami už modulio ribų turės reikšmes nukopijuotas į kitus modulius, tuomet pakeitus kintamojo reikšmę, reiktų, kad visi moduliai, kurie naudoja šį kintamąjį, būtų perkompiliuoti. Jei kintamąjį pažymėsime kaip *static readonly***,** tai pakeitus jo reikšmę nereikės perkompiliuoti kitų modulių, kurie naudoja šį kintamąjį.

CheckNewExceptionWithoutThorwingRule. Ši taisyklė ieško išimtinių situacijų (angl. *exception*) objektų, kurie yra sukurti, bet neiškviečiami, negrąžinami ar neperduodami kitiems metodams kaip parametrai.

CheckNewThreadWithoutStartRule. Taisyklė ieško naujų gijų, kurios buvo sukurtos, bet nepaleistos, negražintos ar neperduotos kietiems metodams kaip parametras.

CloneMethodShouldNotReturnNullRule. Taisyklė tikrina ar visi Clone() metodai grąžina ne *null*.

ConstructorShouldNotCallVirtualMethodsRule. Ši taisyklė įspėja programuotoją, jei virtualus metodas yra kviečiamas iš paveldimos (angl. *non-sealed*) klasės konstruktoriaus. Problema yra tame, kad jei tėvo klasės metodas pakeičia (angl. *overrides*) vaiko klasės metodą, tai tuomet vaiko klasės metodas iškviečiamas dar prieš įvykdant tėvo klasės konstruktorių. Tai padaro kodą lengvai pažeidžiamą.

DisableDebuggingCodeRule. Ši taisyklė tikrina, kad programos, kurios nenaudoja komandinės eilutės, nekviestų *Console.WriteLine*. Tai yra dažnai naudojama derinimo (angl. *debug*) tikslais, bet toks kodas niekada neturėtu patekti į oficialią (angl. *release*) versiją. Jeigu nenorima ištrinti tokio kodo, tuomet galima tai pakeisti į *Debug.WriteLine* ar *Trace.WriteLine*. Tačiau reikia nepamiršti, kad TRACE dažnai būna įjungtas oficialiose versijose.

DoNotForgetNotImplementedMethodsRule. Ši taisyklė skirta tam, kad nepamiršti pabaigti metodus prieš išleidžiant oficialią programos versiją.

DoNotUseEnumIsAssignableFromRule. Ši taisyklė užtikrina, kad būtų kviečiama *type.IsEnum*, vietoje *typeof (Enum).IsAssignableFrom (type)*, nes taip paprasčiau ir suprantamiau.

DoNotUseGetInterfaceToCheckAssignabilityRule. Ši taisyklė tikrina *Type.GetInterface* kvietimą, kuris atrodo kaip užklausa tikrinanti ar šis tipas yra palaikomas. Problema tame, kad tik modulyje apibrėžti vardai unikaliai identifikuoja tipą. Taigi, naudojant sąsajos (angl. *interface*) vardą ar vardų sritį (angl. *namespace*), galima gauti netinkamą rezultatą.

EqualShouldHandleNullRule. Šis taisyklė užtikrina, kad *Equals(object)* metodas grąžintų *false*, kuomet *object* parametras yra *null*.

GetEntryAssemblyMayReturnNullRule. Ši taisyklė įspėja kuomet modulis be įvesties taško (pvz.: DLL ar biblioteka) kviečia *Assembly.GetEntryAssembly ()*. Tai yra problematiška, nes metodas visuomet gražina *null*, kai kviečiama ne iš pagrindinės programos, o iš išorės.

ObsoleteMessagesShouldNotBeEmptyRule. Ši taisyklė įspėja, jei bet koks tipas, kintamasis, įvykis, metodas ar konstruktorius yra deklaruotas su tuščiu *[Obsolete]* atributu. Tokioje situacijoje atributas yra labai svarbus, nes jis paaiškina, kodėl geriau to nenaudoti ir ką geriau naudoti.

OnlyUseDisposeForIDisposableTypesRule. Nekurti *Dispose* metodo klasėse, kurios nepaveldi *IDisposable* sąsajos, nes tai gali suklaidinti programuotojus.

PreferEmptyInstanceOverNullRule. Ši taisyklė tikrina ar visi metodai ir *property*, kurie gražina *string, array, collection* ar *enum* tipo reikšmes, negražina *null*. Dažniausiai geriau grąžinti tuščią reikšmę, nes tuomet kviečiančiajam nereikia papildomai tikrinti ar grąžinta reikšmė nelygi *null*.

PreferSafeHAndleRule. Su *native* kodu dažniausiai geriausia bendrauti naudojant *System.Runtime.InteropServices.SafeHandle* vietoj *System.IntPtr* ar *System.UIntPtr* nes:

- SafeHandles yra saugaus tipo.
- SafeHandles garantuoja, kad bus atlaisvinti (angl. *dispose*) resursai, kai įvyks išimtinės situacijos, tokios kaip gijos netikėtas nutraukimas ar dėklo perpildymas.
- SafeHandles yra atsparūs pakartotinio panaudojimo (angl. *recycle*) atakoms.
- Nereikia rašyti užbaigėjų (angl. *finalizer*), kuriuos kartais gali būti sudėtinga parašyti.

ReplaceIncompleteOddnessCheckRule. Ši taisyklė ieško problematiškų patikrinimų ar skaičius nelyginis. Dažnai tai yra realizuojama panaudojant modulį (x % 2 == 1). Tačiau tai neveiks su neigiamais skaičiais. Dar geresnis (greitesnis) būdas būtų tikrinti paskutinį skaičiaus bitą.

ToStringShouldNotReturnNullRule. Tikrinama ar pakeistas *ToString()* metodas negrąžina *null*. Geriau grąžinti *string.Empty* vietoje *null*.

1.3.1.1.2. Gendarme.Rules.Concurrecy

Gendarme.Rules.Concurrecy kategorijoje nagrinėjamos problemos susijusios su lygiagretumu.

Kategorijos taisyklės:

DoNotLockOnThisOrTypesRule. Ši taisyklė tikrina ar *lock* operatorius nėra naudojamas su *this* ar su *type*. Tai gali sukelti problemų, nes kiekvienas gali vykdyti *lock* operaciją su *this* ar su *type*. Ir jei kita gija vykdys *lock* operaciją su *this* ar su *type*, tuomet labai didelė aklavietės tikimybė. Tinkamas būdas būtų susikurti privatų objektą ir naudoti *lock* operaciją su juo.

DoNotLockOnWeakIdentityObjectsRule. Ši taisyklė užtikrina, kad *lock* operatorius nebūtų naudojamas su objektais su silpnu identiškumu. Tai tokie objektai, kurie gali būti naudojami įvairiose taikymo srityse. Šie tipai turi silpną identiškumą:

- System.MarshalByRefObject
- System.OutOfMemoryException
- System.Reflection.MemberInfo
- System.Reflection.ParameterInfo
- System.ExecutionEngineException
- System.StackOverflowException
- System.String
- System.Threading.Thread

DoNotUseLockedRegionOudsideMethodRule. Ši taisyklė tikrina ar metodas iškvietęs *System.Threading.Monitor.Enter* iškviečia ir System.*Threading.Monitor.Exit*. Iškviesti tik *Enter* viešuosiuose metoduose yra labai blogai, nes tuomet kviečiantysis turi pasirūpinti iškviesti *Exit*. O to daryti jis neprivalo. Tai padidina aklavietės susidarymo galimybę. Tai yra mažiau pavojinga privatiems metodams, nes *lock* operatorius yra valdomas toje pačioje klasėje. O analizuoti klasę yra lengviau. Lengviau aptikti ir ištaisyti su tuo susijusias klaidas.

DoNotUseMethodImplOptionsSynchronizedRule. Taisyklė skirta metodams, kurie yra apibrėžti kaip MethodImpl(MethodImplOptions.Synchronized)]. Veikimo metu šie metodai sinchronizuojami naudojant *lock(this)* paprastiems metodams ir *lock(typeof(X))* statiniams metodams. Tai gali sukelti problemų, nes kiekvienas gali vykdyti *lock* su *this* ar su *type*. Ir jei kita gyja vykdys *lock* su *this* ar su *type*, tuomet labai didelė aklavietės tikimybė. Tinkamas būdas būtų susikurti privatų objektą ir naudoti *lock* su juo.

DoNotUseThreadStaticWithInstanceFielsRule. Ši taisyklė skirta kintamiesiems, kurie apibrėžti *[ThreadStatic]* atributu, tačiau patys nėra statiniai. Atributas veiks tik su statiniais kintamaisiais, todėl nėra prasmės pridėti jį ne statiniams kintamiesiems.

DoubleCheckLockingRule. Taisyklė naudojama patikrinti *double-check* šabloną, kuris dažnai naudojamas *singleton* šablone. Ji įspėja apie galimą netinkamą naudojimą. Originalus CLR (1.x) negarantuoja, kad *double-check* teisingai veiks kelių gijų programose. Tačiau ši technika veikia tik x86 architektūroje. Tai yra dažniausiai naudojama architektūra, dėl to ši problema retai matoma. CLR 2 ir naujesnės turi stiprų atminties modelį, kuris užtikrina dvigubą *lock* patikrinimą (jei priskirtas *volatile* kintamajam).

NonConstantStaticFieldsShouldNotBeVisibleRule. Ši taisyklė įspėją, jei randa viešą statinį, neapibrėžtą kaip konstanta, kintamąjį. Kelių gijų programose prieiga prie tokių kintamųjų turi būti sinchronizuota.

ProtectCallToEventDelegatesRule. Taisyklė užtikrina, kad įvykio (angl. *event*) iškvietimas būtų saugiai atliktas. Visų pirma įvykis turi būti nukopijuojamas į lokalų kintamąjį, kad išvengti lenktyninių situacijų. Ir taip pat turi būti patikrintas ar nėra *null* prieš panaudojimą.

ReviewLockUsedOnlyForOperationsOnVariablesRule. Ši taisyklė patikrina ar *lock* operatorius yra naudojamas operacijoms atlikti tik su klasės ar lokaliais kintamaisiais. Jei vienintelis tikslas kritinėje dalyje yra užtikrinti, kad kintamasis bus pakeistas automatiškai, tuomet geresnis būdas yra naudoti *System.Threading.Interlocked*.

WriteStaticFieldFromInstanceMethodRule. Ši taisyklė naudojama metodams, kurie rašo bei priskiria reikšmes statiniams kintamiesiems patikrinti. Tai gali sukelti problemų kelių gijų programose.

1.3.1.1.3. Gendarme.Rules.Correctness

Gendarme.Rules.Correctness kategorijoje nagrinėjamos įvairaus tipo problemos susijusios su klaidingu tipo ar klasės naudojimu.

Kategorijos taisyklės:

AttributeStringLiteralsShouldParseCorrectlyRule. Kadangi atributai yra naudojami kompiliavimo metu, todėl tik konstantos gali būti perduotos konstruktoriams. Tai gali sukelti spartos klaidų, jei pavyzdžiui bus klaidingai suformuota URI eilutė.

AvoidConstructorsInStaticTypesRule. Ši taisyklė ieško klasių, kurios turi tik statinius narius, ir praneša klaidą, jei tokia klasė turi viešą konstruktorių. Tai buvo dažna klaida 1.x Framewok, nes C# pridėdavo standartinį, vieša konstruktorių, jei joks konstruktorius nebuvo nurodytas. 2.0 ir naujesnio Framework naudotojai, turi pakeisti klasės tipą į statinį, jei įmanoma.

AvoidFloatingPointEqualityRule. Dažniausiai slankiojo kablelio formos skaičiai negali būti teisingai palyginti naudojant lygybę ir nelygybę. Taip yra todėl, kad slankiojo kablelio formos skaičiai yra netikslūs ir daug operacijų su šiais skaičiais gali sukelti klaidų. Ši taisyklė neleidžia naudoti lygybės ir nelygybės su natūraliais ar racionaliais skaičių tipais. Tokios operacijos palyginime turėtų naudoti paklaidą.

BadRecursiveInvocationRule. Ši taisyklė aptinka kelis bendrus scenarijus, kur metodo kvietimas gali pakliūti į begalinę rekursiją.

CallingEqualsWithNullArgRule. Taisyklė patikrina ar metodas, kuris kviečia *Equals*, nepaduoda *null* parametro.

CheckParameterNullityInVisibleMethodRule. Taisyklė patikrina, kad parametrai, kurie gali įgyti *null* reikšmes visuose matomuose metoduose, turi būti patikrinti ar nėra *null* prieš juos naudojant.

DisposableFieldsShouldBeDisposedRule. Ši taisyklė užtikrina, kad visoms klasėms, turinčioms *IDisposible* sąsaja, būtų iškviestas *Dispose* metodas.

DoNotCompareWithNaNRule. IEEE 754 standarte parašyta, jog neįmanoma palyginti begalinių reikšmių. Toks palyginimas visada gražins *false*. Tam reiktų naudoti metodus *Single.IsNaN* ir *Duoble.IsNan*.

DoNotRecurseInEqualityRule. *operator==* ar *operator!=* gali kviesti save rekursiškai. Tai dažnai sukeliama pamirštant paversti (angl. *cast*) argumentą į *System.Object* prieš jį lyginant su *null*.

DoNotRoundIntegerRule. Ši taisyklė randa bandymus kviesti *Round*, *Ceiling*, *Floor*, *Truncate* sveikojo tipo kintamiesiems. Tai dažniausiai nutinka dėl rašybos klaidų kode ar nereikalingų operacijų naudojimo.

EnsureLocalDisposalRule. Ši taisyklė užtikrina, kad išvalomi (angl. *disposable*) lokalūs kintamieji būtų išvalyti prieš metodui baigiantis. Tam reikia pasitelkti į pagalbą *Using*, *try/finally*.

FinalizersShouldCallBaseClassFinalizerRule. Ši taisyklė įspėja, jei *finalizer* neiškviečia vaiko klasės *finalizer*. C# programose kompiliatorius tai atlieka pats, tačiau kai kurios kalbos (kaip IL) leidžia tokią elgseną.

MathodCanBeMadeStaticRule. Ši taisyklė užtikrina, kad metodai, kurie nenaudoja *this* būtų paversti į statinius. Tai truputį pagerina programos vykdymo spartą.

ProvideCorrectArgumentsToFormattingMethodsRule. Užtikrinama, kad argumentų būtų tiek, kiek nurodyta formatuojamoje eilutėje naudojant *String.Format*.

ProvideCorrectRegexPatternRule. Taisyklė užtikrina, kad teisinga reguliarioji išraiška yra naudojama kaip argumentas.

ProvideValidXmlStringRule. Taisyklė užtikrina, kad teisingos XML sąlygos butų perduodamos kaip argumentai.

ProvideValidPathExpressionRule. Užtikrina, kad teisingos XPath eilučių išraiškos būtų perduodamos kaip argumentai.

ReviewCastOnIntegerDivisionRule. Taisyklė aptinka sveikųjų skaičių dalybą, kurios rezultatas priskiriamas slankiojo kablelio tipo kintamiesiems. Tokiu atveju dalybos operacijos liekana prarandama.

ReviewCastOnIntegerMultiplicationRule. Taisyklė aptinka *integer* tipo skaičių daugybą, kurios rezultatas priskiriamas *long* tipo kintamajam be apibūdinimo (angl. *cast*). Dauginant maksimalų *integer* tipo skaičių iš didesnio nei vienas, gautume *long* tipo skaičių, tačiau neatliekant apibūdinimo, jis būtų paverstas į *integer* ir tik po to į *long*. Taip jis bus iškraipytas ir nebeteisingas.

ReviewDoubleAssigmentRule. Taisyklė tikrina ar kintamajam daug kartų nepriskiriama ta pati reikšmė. Dažniausiai tai aptinka rašymo klaidas, kurios paslepia tikras problemas.

ReviewSelfAssignmentRule. Suranda kintamuosius, kurie priskiriami patiems sau. Tai nepakeičia nieko, tačiau tai gali būti rašymo klaida, kuri paslepia tikrąją problemą.

ReviewUselessControlFlowRule. Aptinka blokus, kuriuose niekas neatliekama.

ReviewUseInt64BitsToDuobleRule. Tikrina vertimą iš sveikojo skaičiaus į realų naudojant dviprasmiškai pavadintą funkciją *BitConverter.Int64BitsToDuoble*. Ši funkcija verčia bitus, o ne reikšmę.

ReviewUseOfModuloOneIntegersRule. Tikrina ar neatliekama operacija skaičiaus moduliu 1 (x mod 1). Šios operacijos rezultatas visuomet bus lygus 1.

UseValueInPropertySetterRule. Taisyklė užtikrina, kad visi kintamojo „Setter“ metodai turėtų argumentą.

1.3.1.1.4. Gendarme.Rules.Design

Gendarme.Rules.Design kategoriją sudaro vienas didžiausių taisyklių rinkinių skirtas projektavimo klaidoms aptikti. Pavyzdžiui, visi *property* tipo kintamieji turi turėti *Get()* metodą. Abstrakčios klasės neturėtų turėti viešo konstruktoriaus. Sąsajos neturėtų būti tuščios ir pan. Dėl apimties, šios ir sekančių kategorijų taisyklės nenagrinėjamos detaliai.

1.3.1.1.5. Gendarme.Rules.Design.Generic

Gendarme.Rules.Design.Generic kategorijos taisyklių rinkinys skirtas aptikti klaidas, kuomet naudojamasi .NET *System.Collection.Generics* klasėmis. Aptinkamos tokios klaidos kaip:

- ne visų *Generic* tipo parametrų naudojimas;
- *Generic* tipo naudojimas kitame *Generic* tipe;
- visada naudoti *Generic IEnumerable* sąsają;
- vietoje *ref*/*out* object naudoti *Generic* (*ref*/*out T*);
- jei įmanoma, naudoti *Generic* tipo *EventHandler* vietoje delegato.

1.3.1.1.6. Gendarme.Rules.Design.Linq

Gendarme.Rules.Design.Linq kategoriją sudaro taisyklių rinkinys skirtas aptikti klaidas susijusias su LINQ naudojimu. Kol kas čia yra tik viena taisyklė patarianti vengti papildančiuose (angl. *extension*) metoduose naudoti *System.Object* tipo parametrų. Patariama naudoti konkretaus tipo parametrus, nes kai kurios kalbos to nesupranta.

1.3.1.1.7. Gendarme.Rules.Exceptions

Gendarme.Rules.Exceptions kategorijos taisyklių rinkinyje visos taisyklės aptinka netinkamą išimčių (angl. *exceptions*) naudojimą. Aptinkamos klaidos susijusios su:

- išimčių naudojimu nenurodant jokios informacijos apie išimtį;
- ne konkrečių išimčių naudojimu;
- išimčių naudojimu netinkamuose vietose;
- išimčių perdavimu;
- ir kt.

1.3.1.1.8. Gendarme.Rules.Interoperability

Gendarme.Rules.Interoperability kategorijos taisyklių rinkinys skirtas aptikti sąveikos klaidas. Klaidas tarp *managed* ir *native* tipų naudojimo. Taip pat klaidas susijusias su *PInvoke* naudojimu.

1.3.1.1.9. Gendarme.Rules.Maintainability

Gendarme.Rules.Maintainability kategorijos taisyklės skirtos, kad programos netaptų labai sudėtingos, kurias sunku prižiūrėti. Taisyklės skaičiuoja metodų sudėtingumus ir rekomenduoja juos išskaidyti jei jie per sudėtingi. Patariama išvengti gilaus paveldėjimo. Patariama naudoti chronometrą vietoje laikrodžio, jei reikia apskaičiuoti laiko intervalą.

1.3.1.1.10. Gendarme.Rules.Naming

Gendarme.Rules.Naming kategorijos taisyklių rinkinys skirtas pavadinimų, rašybos klaidoms aptikti. Jis sako nenaudoti ne alfabetinių simbolių pavadinimuose. Nenaudoti klasės vardo metodų pavadinimuose. Nenaudoti „After“ ir „Before“ įvykių pavadinimuose. Ir daug kitų patarimų susijusių su pavadinimais, kurie palengvina kodo supratimą ir padaro jį aiškesnį.

1.3.1.1.11. Gendarme.Rules.Performance

Gendarme.Rules.Performance kategorijos taisyklių rinkinys skirtas aptikti klaidas, kurios žymiai mažina kodo našumą, spartą. Patariama nenaudoti daug privačių kintamųjų, didelių struktūrų. Nepalikti nenaudojamų privačiųjų kintamųjų, klasių. Neatlikinėti perteklinių veiksmų ir pan.

1.3.1.1.12. Gendarme.Rules.Portability

Gendarme.Rules.Portability kategorijoje esančios taisyklės aptinka kodo vietas, kurios gali neveikti naudojant kodą skirtingose aplinkose. Patariama nenaudoti įkoduotų kelių. Taip pat kelios taisyklės aptinka kodo vietas, kurios neveiks Unix aplinkoje.

1.3.1.1.13. Gendarme.Rules.Security

Gendarme.Rules.Security kategorijos taisyklės padeda išvengti kai kurių saugumo spragų. Pavyzdžiui, viešieji masyvo tipo kintamieji neturtėtų būti *readonly*, nes tai tik neleis pakeisti masyvo elementų, tačiau pats masyvas gali būti pakeistas. Taip pat, kai kurios taisyklės tikrina metodus, kurie perrašo standartinius „.NET Framework“ metodus tikrinančius sertifikatų teisingumą. Tikrinama ar tinkamai perrašyti metodai, ar nepalikta žinomų saugumo spragų.

1.3.1.1.14. Gendarme.Rules.Security.Cas

Gendarme.Rules.Security.Cas kategorijos taisyklės skirtos užtikrinti kodo prieigos saugumą (angl. *Code Access Security*). Patikrina ar priėjimą prie kodo ribojančios taisyklės naudojamos tinkamai.

1.3.1.1.15. Gendarme.Rules.Serialization

Gendarme.Rules.Serialization kategorijos taisyklės skirtos publikavimo, transliavimo (angl. *serialization*) klaidoms aptikti.

1.3.1.1.16. Gendarme.Rules.Smells

Gendarme.Rules.Smells kategorijos taisyklės aprašo kodo pertvarkymo taisykles. Patariama vengti vienodo kodo toje pačioje ar tos pačios giminystės klasėje. Patariama vengti ilgų klasių ir metodų, taip pat didelio parametrų skaičiaus.

1.3.1.1.17. Gendarme.Rules.UI

Gendarme.Rules.UI kategorijos taisyklės aprašo vartotojo sąsajos taisykles. Jos yra tik kelios. Jei kompiliuojama vykdomoji programa, kuri remiasi *gtk-sharp* ar *System.Windows.Forms* moduliu, tuomet patariama prie kompiliavimo argumentų nurodyti *–target:winexe*. Tokiu atveju nebus rodomas komandinės eilutės langas programos vykdymo metu.

1.3.2. „Cppcheck“ įrankis

„Cppcheck“ [11] yra statinės analizės įrankis skirtas C/C++ kalba rašytam kodui tikrinti. Skirtingai nei kompiliatoriai ar daugelis kitų statinės kodo analizės įrankių, šis įrankis neaptinka sintaksės klaidų. „Cppcheck“ dažniausiai aptinka tik klaidas, kurių įprastai neaptinka kompiliatorius. Šis įrankis rašytas C++ programavimo kalba.

Šis įrankis dar toli gražu nėra baigtas. Jis laikas nuo laiko patobulinamas. „Cppcheck“ retai klysta, rasti įspėjimai dažniausiai būna tikros klaidos. Tačiau šis įrankis neaptinka daugelio klaidų.

Aptinkamos klaidos:

- 64 bitų palaikymas. Tikrinama ar kodas veiks 32 ir 64 bitų architektūrose.
- Automatiniai kintamieji. Rodyklė į tokį kintamąjį yra teisinga tol, kol yra tame pačiame bloke. Todėl negalima gražinti rodyklių, nuorodų ar adresų į automatinius ar laikinus kintamuosius.

- Boost naudojimas.
- Ribų tikrinimas. Tikrinama ar neperžengiamos masyvo, sąrašo ar kito elemento ribos.
- Klasių tikrinimas. Tikrinama ar nepamiršti konstruktoriai, ar visi kintamieji ir funkcijos inicializuoti, ar nėra nenaudojamų privačių funkcijų ir kintamųjų.
- Išimčių naudojimas. Tikrinama ar išimtys naudojamos teisingai ir leistinose vietose.
- Tikrinamos palyginimų sąlygos. Tikrinama ar palyginimai nebus visuomet *true* arba *false*. Ar *if* ir *if else* sąlygos ne vienodos.
- Atminties klaidos. Išskirta atmintis turi būti naudojama. Jei konstruktoriuje išskirta atmintis, tai tuomet destruktoriuje ji turi būti atlaisvinta. Visuomet reikia nepamiršti atlaisvinti struktūrų narius. Funkcijoje, metode išskirta atmintis turi būti atlaisvinta prieš jam baigiantis.
- Išorinių funkcijų naudojimas. Įspėjama, jei naudojama funkcija yra išorinė.
- Tuščios rodyklės peradresavimas.
- Pasenusių funkcijų naudojimas. Įspėjama, jei naudojama kokia nors pasenusi funkcija, kuri jau turėtų būti nebenaudojama.
- STL naudojimas. Ieško netinkamo STL naudojimo. Ribų peržengimo, išvalyto iteratoriaus naudojimo, nereikalingų (perteklinių) sąlygų ir kt.
- Postfikso operatoriaus naudojimas. Patariama geriau naudoti prefikso operatorių.
- Kitos klaidos. Tikrinamos tokios dažniausiai pasitaikančios klaidos kaip dalyba iš nulio, blogas funkcijų *scanf*, *strtol*, *substr*, *sizeof*, *strcpy* naudojimas ir daug kitų.

„Cppcheck“ įrankis aptinka labai svarbias klaidas. Jis net nebando aptikti klaidų, kurias įprastai aptinka kompiliatorius. Be to jis retai klysta. Tačiau šis įrankis dar neaptinka daug reikšmingų C++ programavimo kalbai būdingų klaidų.

1.3.3. „FindBugs“ įrankis

„FindBugs“ [12] yra atviro kodo statinės kodo analizės įrankis skirtas Java programinio kodo analizei. Jam reikalingas JRE (ar JDK) 1.5.0 ar naujesnis. Tačiau jis gali analizuoti programas sukompiliuotas su bet kuria Java versija nuo 1.0 iki 1.8. Šiuo metu naujausia versija yra 2.0.2 išleista 2012 m. Gruodžio 10 d. Ieškant žinomų netinkamo kodo šablonų (angl. *pattern*) įrankis iš esmės remiasi paprasta vidinių procedūrų analize. Šis įrankis nebando surasti visų klaidų esančių programiniame kode. Priešingai, jis nufiltruoja įspėjimus, kurie gali būti netikri, ir žemo poveikio klaidas. Įspėjimai suskirstyti į daugiau nei 380 šablonų, o šie sugrupuoti į kategorijas tokias kaip teisingumas, bloga praktika ar saugumas [13].

2.0.2 versijoje kiekvienas įspėjimas turi reitingą ir patikimumo laipsnį. Įspėjimams skiriamas reitingas nuo 1 iki 20 ir jie sugrupuojami į tokias kategorijas: baisiausi (reitingas nuo 1 iki 4), baisūs (nuo 5 iki 9), keliantys nerimą (nuo 10 iki 14) ir keliantys susirūpinimą (nuo 15 iki 20). Reitingo parinkimas įspėjimui yra subjektyvus, grindžiamas programuotojų praktika, problemos rimtumu ir galimu poveikiu. Be to, atsižvelgiama ir į tikimybę kaip greitai klaida gali būti surasta vykdant kodą. Pavyzdžiui, yra manoma, kad aptikti amžiną ciklą vykdant programą yra nesunku, nes gedimo atveju programa dažniausiai pateikia prasmingos informacijos apie klaidą ir jos vietą. Dėl to šiam įspėjimui skiriamas mažesnis reitingas. Tuo tarpu patikimumo laipsnis nusako įspėjimo patikimumą, tikrumą. Jei neabejojama, kad problema tikrai egzistuoja, tuomet patikimumas yra aukštas. Jei abejojama dėl įspėjimo patikimumo tuomet vidutinis. Ir žemas, kuomet labai abejojama dėl patikimumo. Patikimumas leidžia palyginti įspėjimus, kurie priklauso tam pačiam problemos šablonui. Įspėjimų iš skirtingų šablonų lyginimui, naudojamas reitingas.

„FindBugs“ gali būti vykdomas daugeliu būdų: kaip komandinės eilutės sąsaja, kaip programa su grafine vartotojo sąsaja (kuri taip pat gali būti naudojama per internetą, naudojant Java Web Start), kaip įskiepis (angl. *plugin*) keletui populiariausių programavimo aplinkų (Eclipse, Maven, Netbeans, Hudson, IntelliJ) ar programos kompiliavimo metu. Programa paremta grafine vartotojo sąsaja turi galimybę klasifikuoti ir komentuoti apie kiekvieną problemą. Taip pat ši programa leidžia rašyti aptiktas įrankio klaidas į centrinę klaidų duomenų bazę.

2.0.2 programos versija aptinka 402 klaidas, kurios suskirstytos į tokias 9 kategorijas:

1. Bad practice. Kaip ir „Gendarme“ statinės kodo analizės įrankis, „FindBugs“ įrankis šioje kategorijoje nagrinėja įvairaus tipo problemas, kurios nebūtinai sukels klaidų, tačiau yra geriau tą patį funkcionalumą įgyvendinti kitaip ar visiškai jo atsisakyti. Pvz.: geriau yra naudoti *clear* metodą vietoj *removeAll* norint išvalyti rinkinius, nes taip yra aiškiau ir efektyviau, be to kai kuriems rinkiniams *removeAll* gali išmesti *ConcurrentModificationException* išimtį. Šiai kategorijai priskirtos 85 taisyklės.
2. Correctness. Tai daugiausiai taisyklių turinti kategorija. Jai priskiriamos 143 taisyklės. Čia nagrinėjamos įvairaus tipo problemos susijusios su klaidingu tipo, klasės ar klasės metodo naudojimu, jo įgyvendinimu.
3. Experimental. Šiai kategorijai priskiriamos taisyklės, kurių įgyvendinimas ar aptinkamos klaidos yra svarstytinos. Laukiama vartotojų atsiliepimų apie šioje kategorijoje esančias taisykles. Šiuo metų kategorijai priskirtos 3 taisyklės. Dvi iš jų nėra naujos, tačiau aptinka tas pačias klaidas kitokiu būdu. Tikimasi, kad šis būdas bus geresnis.
4. Internationalization. Šioje kategorijoje esančios taisyklės aptinka klaidas galinčias kilti dėl skirtingų kalbų naudojimo, skirtingų vietos ar kalbos nustatymų. Tai mažiausia kategorija turinti 2 taisykles.
5. Malicious code vulnerability. Šios kategorijos taisyklės aptinka pažeidžiamumus, kuriais gali pasinaudoti kenkėjiškos programos. Pvz.: aptinkami *public* metodai ar kintamieji, kurie turėtų būtų *protected* ar *final*.
6. Multithreaded correctness. Šioje kategorijoje nagrinėjamos problemos susijusios su lygiagretumu. Pvz.: tikrinama ar teisingai naudojamas *Lock* objektas, ar visuomet jis atrakinamas. Kategorijai priklauso 45 taisyklės.
7. Performance. Šios kategorijos taisyklių rinkinys, kurį sudaro 27 taisyklės, skirtas aptikti klaidas, kurios mažina kodo našumą, spartą. Patariama nepalikti nenaudojamų privačiųjų kintamųjų ir klasių. Neatlikinėti perteklinių veiksmų. Siūloma, naudoti kuriuos alternatyvius metodus norint pagerinti spartą ir pan.
8. Security. Šios kategorijos taisyklės bando aptikti tokias saugumo spragas, kaip SQL injekcijos ar XSS atakos. Nuo *Malicious code vulnerability* aptinkamų pažeidžiamumų šios kategorijos taisyklės skiriasi tuo, kad aptinka pažeidžiamumus, kuriais galima pasinaudoti tiesiogiai per vartotojo sąsaja (be papildo kodo rašymo).
9. Dodgy code. Šią kategoriją sudaro taisyklės aptinkančios dviprasmes ar nelogiškas kodo vietas. Pvz.: savęs priskyrimas (x = x), dvigubas priskyrimas (x = x = 17), tos pačios sąlygos naudojimas *switch* komandos atveju ir pan.

„FindBugs“ įrankis aptinka labai daug įvairių klaidų. Jis ne tik gali būti įdiegtas į pagrindines Java programavimo aplinka ar naudojamas per komandinės eilutės sąsaja, bet gali būti naudojamas ir kaip atskira programa. Tačiau šio įrankio trūkumai yra ne itin patogus konfigūravimas ir nelabai aiškus, patogus rezultatų pateikimas.

1.3.4. Įrankių analizės apibendrinimas

1.1 lentelėje pateikiamas išanalizuotų statinės kodo analizės įrankių palyginimas.

1.1 lentelė. Statinės kodo analizės įrankių palyginimas.

Įrankis	Analizuojama programavimo kalba	Taisylių kiekis	Analizuojamo kodo tipas	Naudojimo būdai	Rezultatų pateikimas
Gendarme	C#	254	Sukompiliuotas	Progama, komandinės eilutės sąsaja	XML, HTML
Cppcheck	C++	~125	Sukompiliuotas	Komandinės eilutės sąsaja, įskiepis	XML, HTML
FindBugs	Java	402	Sukompiliuotas	Progama, komandinės eilutės sąsaja, įskiepis	HTML

„Gendarme“ ir „FindBugs“ įrankiai yra labiau ištobulinti ir aptinka daugiau įvairių klaidų nei „Cppcheck“. Tačiau kiekvienas programuotojas gali daryti tik jam būdingų klaidų, dėl to suskaičiuoti ir aptikti visus įmanomus pažeidžiamumus yra neįmanoma. Taip pat pastarieji įrankiai turi ir daugiausiai panašumų, nes jų analizuojamos programavimo kalbos (C# ir Java) yra panašiausios. Visi įrankiai analizuoja sukompiliuotą baitų kodą, nes tokia analizė daug spartesnė.

Tolesniam tobulinimui pasirinktas „Gendarme“ statinės kodo analizės įrankis, nes C# ir .NET populiarumas vis auga [14]. „Gendarme“ kodas aiškesnis, taisyklės ir aptinkamos klaidos aiškiai atskirtos. Patogiai konfigūruojamas ir rezultatų pateikimas aiškesnis.

2. SIŪLOMI „GENDARME“ ĮRANKIO PATOBULINIMAI

Siekdami patobulinti statinę kodo analizę, pasinaudojome „Gendarme“ įrankiu. Išanalizavus jau esančias įrankio taisykles, buvo pastebėta, kad keletą jų galima patobulinti siekiant pagerinti saugumą, greitaveiką stabilumą ir kodo skaitomumą. Keletas galimų ir reikalingų analizės patobulinimų nebuvo galima realizuoti esamose taisyklėse, todėl buvo pasiūlytos naujos taisyklės, kurios aprašytos 2.2 skyrelyje.

2.1. Siūlomi esamų taisyklių patobulinimai

Šioje dalyje aprašomi siūlomi esamų „Gendarme“ įrankio taisyklių patobulinimai. Šie patobulinimai turėtų padėti aptikti daugiau klaidų ir pagerinti programų spartą, stabilumą, saugumą ir pagerinti kodo skaitomumą. Patobulinimai atrasti bandant įvairius tam tikros klaidos variantus ir tikrinant ar tai klaidai skirta taisyklė ją aptinka. Taip buvo atrasta keletas taisyklių patobulinimų, kurie plačiau aprašomi skyreliuose 2.1.x.

2.1.1. ReviewLockUsedOnlyForOperationsOnVariablesRule taisyklės patobulinimas

ReviewLockUsedOnlyForOperationsOnVariablesRule taisyklė patikrina ar *lock* operatorius yra naudojamas operacijoms atlikti tik su klasės ar lokaliais kintamaisiais. Jei vienintelis tikslas kritinėje dalyje yra užtikrinti, kad kintamasis bus pakeistas automatiškai, tuomet geresnis būdas yra naudoti *System.Threading.Interlocked*.

Netinkamo programinio kodo pavyzdys:

```
lock (m_lockObject)
{
    count++;
    m_someSharedObject = anotherObject;
}
```

Geras pavyzdys:

```
Interlocked.Increment (count);
Interlocked.Exchange (m_someSharedObject, anotherObject);
```

Pavyzdys, kurio programa nesiūlo patobulinti:

```
lock (m_lockObject)
{
    Interlocked.Increment (count);
    m_someSharedObject = anotherObject;
}
```

lock operatorius naudojamas apsaugoti kritinius programos resursus. *lock* viduje esančios operacijos atliekamos tik vienos gijos vienu metu. Kelios gijos negali patekti į *lock* vidų, jei operatorius užrakinimui naudoja tą patį objektą. Pavyzdžiui, jei operacijas su banko sąskaita bus leidžiama atlikti kelioms gijoms vienu metu, tuomet galima tokia situacija:

1. Gija A nuskaito sąskaitos likutį, kuris lygus 100lt.
2. Gija B taip pat nuskaito sąskaitos likutį.
3. Abi gijos patikrina ar pakanka pinigų operacijai atlikti.
4. A atlieka pervedimo operaciją ir pervedusi 60lt įrašo sąskaitos likutį 40lt.
5. B atlieka operaciją pagal debeto sutartį ir pervedusi 30lt įrašo sąskaitos likutį 70lt.

Galutinis rezultatas sąskaitoje po dviejų operacijų bus 70lt. Kad šito išvengti naudojamas *lock* operatorius. Jei viena gija patenka į *lock* vidų, tuomet kitos, norinčios atlikti operacijas su tais pačiais resursais laukia, kol jie atsilaisvins.

System.Threading.Interlocked klasės metodai įvairias operacijas paverčia į nedalomas operacijas. Pavyzdžiui, sumos operacija atliekama per 3 žemesnio lygio operacijas (reikšmės

nuskaitymas, padidinimas, įrašymas). Čia taip pat galimos įvairios klaidos, kai kelios gijos atlieka operacijas su tuo pačiu kintamuoju. Pvz.: *Interlocked.Add* metodas atlieka sumos operaciją kaip nedaloma ir leidžia išvengti įvairių su gijomis susijusių klaidų.

Trūkumas yra tas, kad „Gendarme“ įrankis neatpažįsta *System.Threading.Interlocked* klasės metodų kaip automatinių metodų su kintamaisiais. Ir apskritai reiktų pasvarstyti ar verta naudoti *Interlocked* klasės metodus *lock* operatoriaus viduje. Jo naudojimas nieko blogo nedaro, jei *lock* bloke yra ne tik automatinės operacijos su kintamaisiais. Tačiau, jei bloke yra tik automatinės operacijos ir *Interlocked* metodai, tuomet įrankis nebesiūlo atsisakyti *lock* operatoriaus ir vietoj jo naudoti *Iterlocked* metodus. Siūlau patobulinti įrankį, kad *Iterlocked* klasės metodai būtų traktuojami kaip automatinės operacijos su kintamaisiais.

2.1.2. CheckParametersNullityInVisibleMethodsRule taisyklės patobulinimas

CheckParametersNullityInVisibleMethodsRule taisyklė sako, jog matomų metodų parametrai turėtų būti patikrinti ar jų reikšmė ne *null* prieš juos naudojant. Siūlau patobulinti taisyklę, kad ji aptiktų klaidą, jei prieš parametro naudojimą buvo iškviesti tik *public* metodai. Tarkime, kad iškviestieji *public* metodai laikosi šios taisyklės ir patikrinę perduoto parametro reikšmę nutraukia savo veiklą nepakeitę parametro. Tokiu atveju parametras išlieka *null* ir jo naudojimas gali iššaukti programos „nulūžimą“. Bet ir kitu atveju, kai iškviečiamas *public* metodas, yra visai logiška patikrinti ar parametras ne *null*, nes mes negalime garantuoti, jog *public* metodai visada grąžins ne *null* reikšmę.

Netinkamo programinio kodo pavyzdys:

```
public void CheckParametersNullityInVisibleMethodsRule (string name)
{
    if (name.Length > 10)
        Console.WriteLine ("Name: " + name);
}
```

Geras pavyzdys:

```
public void CheckParametersNullityInVisibleMethodsRule (string name)
{
    if (name != null && name.Length > 10)
        Console.WriteLine ("Name: " + name);
}
```

Tačiau ši taisyklė nebeaptiks klaidos, jei prieš tai mes iškviesime bet kokį metodą ir jam kaip parametrą perduosime šio metodo parametrą.

Pavyzdys, kuriame programa neaptinka klaidos:

```
public void CheckParametersNullityInVisibleMethodsRule (string name)
{
    string nameCopy = string.Copy (name);
    if (name.Length > 10)
        Console.WriteLine ("Name: " + nameCopy);
}
```

Tai turėtų padėti sumažinti *NullReferenceException* išimties tikimybę. O tai savo ruožtu padėtų išvengti programos „nulūžimo“ ar klaidų pranešimų galutiniams vartotojams.

2.1.3. ConsiderAddingInterfaceRule taisyklės patobulinimas

ConsiderAddingInterfaceRule taisyklė praneša apie klaidą, jei klasė implementuoja visus interfeiso narius, tačiau neimplementuoja paties interfeiso. Ši taisyklė atsižvelgia į didžiąsias ir mažąsias raides. Siūlyčiau, neatsižvelgti į tai, nes galimi atvejai, kad skirtingi programuotojai tą patį trumpinį rašo skirtingų dydžių raidėmis. Be to nerekomenduojamas atvejis, kad du metodai skirtųsi tik didžiųjų ir mažųjų raidžių rašymu. Šiuo atveju, tai klasės ir interfeiso metodai.

Netinkamo programinio kodo pavyzdys, kai programa nesiūlo paveldėti sąsajos:

```
public interface ITest
{
    void CreateDGNFile ();
    bool IsECInstance ();
}

public class Testas
{
    void CreateDgnFile () {}
    bool IsEcInstance () { return true; }
}
```

Sąsajos paveldėjimas dažniausiai suteikia klasei daugiau lankstumo. Taip pat susieja klasės ir sąsajos pavadinimus ne tik semantiškai. Tai daro kodą aiškesnį ir lankstesnį.

2.1.4. AvoidUnusedVariablesRule taisyklės patobulinimas

AvoidUnusedVariablesRule taisyklė galėtų papildyti ir pakeisti esančią AvoidUnusedPrivateFieldsRule taisyklę. Esama taisyklė aptinka klaidą tik tada, jei privatus klasės kintamasis yra sukurtas ir vėliau jokia reikšmė jam nėra priskiriama. Taisyklę būtų galima patobulinti, kad ji praneštų apie klaidą ir tais atvejais, kai kintamajam yra tik priskiriama reikšmė, o jis pats niekur nenaudojamas. Taip pat, tai galėtų galioti ir privatiems, ir metodų kintamiesiems.

Netinkamo programinio kodo pavyzdys:

```
public class Testas
{
    private string m_unused;

    public void AvoidUnusedVariablesRule (string anyString)
    {
        string localUnusedVariable = anyString;
        m_unused = anyString;

        Console.WriteLine ("AvoidUnusedVariablesRule called.");
    }
}
```

Šiuo atveju „Gendarme“ taisyklės neaptinka jokios klaidos, nors akivaizdžiai matosi, jog klasės ir metodo kintamieji yra pertekliniai.

Geras pavyzdys:

```
public class Testas
{
    public void AvoidUnusedVariablesRule ()
    {
        Console.WriteLine ("AvoidUnusedVariablesRule called.");
    }
}
```

Šie patobulinimai turėtų padėti sumažinti klases ir taip pagerinti programų greitaveiką. Taip pat sumažinti naudojamos atminties kiekį.

2.1.5. ArrayFieldsShouldNotBeReadOnlyRule taisyklės patobulinimas

ArrayFieldsShouldNotBeReadOnlyRule taisyklė sako, kad masyvai neturėtų būti deklaruoti kaip *public readonly*, nes tokiu atveju negalima pakeisti pačio masyvo, bet galima pakeisti masyvo

reikšmes. Tačiau ši taisykle neaptinka klaidos, kai taip deklaruojame sąrašą ar rinkinį (angl. *collection*).

Pavyzdys:

```
public class Testas
{
     // klaidą aptinka ArrayFieldsShouldNotBeReadOnlyRule taisyklė.
     public readonly int[] readOnlyArray;

     // klaidą aptinka DoNotExposeGenericListRule taisyklė.
     public readonly System.Collections.Generic.List<int> readOnlyList =
new List<int> ();

     // jokia klaida neaptinkama
     public readonly System.Collections.ObjectModel.Collection<int>
readOnlyList = new Collection<int> ();
     public readonly System.Collections.Generic.ICollection<int>
readOnlyCollection = new List<int> ();
}
```

Tiesa, jei deklaruojame viešą sąrašą, kita taisyklė (DoNotExposeGenericListRule) sako, kad to daryti nereikėtų. Vietoj to reiktų sukurti viešą *property* ir per jį grąžinti sąrašo kopiją. Siūlau papildyti šią taisyklę, kad ji neleistų deklaruoti ir *public readonly* sąrašų bei rinkinių, nes šiuo atveju duomenys sąrašuose ir rinkiniuose taip pat gali būti pakeisti iš išorės. Šis patobulinimas turėtų padidinti programų saugumą.

2.2. Siūlomos naujos taisyklės

Šioje dalyje aprašomos naujos siūlomos taisyklės, kurių „Gendarme v.2.10.8“ įrankis neturi. Idėjos šioms taisyklėms kilo programuojant (rašant įvairias funkcijas programoms) ir analizuojant esamas taisykles. Šios naujos taisyklės turėtų padėti aptikti daugiau perteklinio, nelogiško kodo, kurio kompiliatorius neaptinka. Taip pat supaprastinti programos kodą ir taip pagerinti jo skaitomumą bei aptikti daugiau spragų, kuriomis pasinaudojus galima „nulaužti“ programas.

2.2.1. AvoidCaseSensitiveNamesRule taisyklės pasiūlymas

AvoidCaseSensitiveNamesRule siūloma taisyklė turės aptikti metodų, kintamųjų pavadinimus, kurie skiriasi tik didžiosiomis ar mažosiomis raidėmis ir parodyti įspėjimą, kad geriau vengti tokių pavadinimų. Tai labai apsunkina kodo skaitymą ir galima nesunkiai įvelti klaidų.

Netinkamo programinio kodo pavyzdys, kurio reikia vengti:

```
public void AvoidCaseSensitiveNamesRule () { }
public void AvoidCaseSensitiveNamesRULE () { }
```

Ši nauja taisyklė tūrėtų padėti rašyti tvarkingesnį ir aiškesnį programinį kodą. Tuo pačiu turėtų padėti išvengti kvailų klaidų, kurios kyla dėl netvarkingo kodo.

2.2.2. AvoidReturnSameValueRule taisyklės pasiūlymas

AvoidReturnSameValueRule nauja taisyklė patikrins ar metodas gali gražinti kelias reikšmes. Jei jis visuomet gražina vieną ir tą pačią reikšmę, tuomet kokia prasmė ją gražinti? Jei vis dėlto reikšmę reikia gražinti, tuomet galima sukurti *public readonly const* pastovų kintamąjį arba grąžinti jo reikšmę kaip funkcijos reikšmę.

Netinkamo programinio kodo pavyzdys:

```
public bool DoSomething (int index)
{
    if (index < 0)
        return false;
```

```
    DoSomethingMore (index);
    return false;
}
```

Geras pavyzdys:

```
public bool DoSomething (int index)
{
    if (index < 0)
        return false;

    DoSomethingMore (index);
    return true;
}
```

Tai turėtų palengvinti programos skaitomumą ir padėti išvengti išsiblaškymo klaidų. Taip pat priminti, kad metodas dar nėra pilnai įgyvendintas, nes visuomet grąžinama ta pati reikšmė.

2.2.3. AvoidUsingNullAfterNullityCheckRule taisyklės pasiūlymas

Prieš naudojant *public* metodo kintamąjį mes dažnai tikriname ar jo reikšmė nelygi *null*. Tačiau esant sudėtingoms *if* sąlygoms galima situacija, kuomet kaip tik bus naudojami *null* kintamieji. Pavyzdyje pateikta gana paprasta *if* sąlyga. Programuotojas rašydamas tokias sąlygas retai padaro klaidų, tačiau dažnai tokios klaidos atsiranda taisant ar perrašant sudėtingą kodą. Siūlau sukurti taisyklę, kuri tikrins ar aptiktas *null* kintamasis nebus vėliau naudojamas.

Netinkamo programinio kodo pavyzdys:

```
public void AvoidUsingNullVariables (string param)
{
    if (param != null || param.Length > 3)
        Console.WriteLine ("Parameter is acceptable.");

    Console.WriteLine ("Parameter is NOT acceptable.");
}
```

Geras pavyzdys:

```
public void AvoidUsingNullVariables (string param)
{
    if (param != null && param.Length > 3)
        Console.WriteLine ("Parameter is acceptable.");

    Console.WriteLine ("Parameter is NOT acceptable.");
}
```

Kaip ir egzistuojanti CheckParametersNullityInVisibleMethodsRule taisyklė, ši taisyklė turėtų padėti sumažinti *NullReferenceException* išimties tikimybę. O tai savo ruožtu padėtų išvengti programos „nulūžimo“ ar klaidų pranešimų galutiniams vartotojams.

2.2.4. AvoidInfiniteLoopRule taisyklės pasiūlymas

Begalinis ciklas – tai programavimo klaida, kuri gali sukelti programos pakibimą.

Netinkamo programinio kodo pavyzdžiai:

```
public void AvoidInfiniteLoop ()
{
    int a = 0;
    while (true)
        DoSomething (a++);
```

```
}

public void AvoidInfiniteLoop (bool visible)
{
    while (visible)
        DoSomething ();
}
```

„Gendarme“ įrankis neaptinka klaidų šiuose pavyzdžiuose, nors čia aiškiai matosi amžinas ciklas. Tiesa, amžinas ciklas neretai gali būti naudojamas sąmoningai, dėl to *while* (*true*) nelaikysime klaida. Klaida laikysime šio ciklo nenutraukimą. Kaip matome abiejuose pavyzdžiuose to nėra. Todėl siūlau sukurti taisyklę, kuri patikrins ar amžini ciklai gali būti nutraukiami. T.y. ar amžinuose cikluose prie kokių nors sąlygų gali būti iškviečiamos *break* ar *return* komandos.

Geras pavyzdys:

```
public void AvoidInfiniteLoop ()
{
    int a = 0;
    while (true)
    {
        if (a > 1000 || !DoSomething (a++))
            break;
    }
}

public void AvoidInfiniteLoop (bool visible)
{
    while (visible)
    {
        if (!DoSomething ())
            break;
    }
}
```

Ši taisyklė turėtų padėti aptikti amžinus ciklus ir taip išvengti programų pakibimo. Amžinas ciklas kokioje nors programos dalyje reiškia ne tik tai, kad programa užstrigs ir teks ją perkrauti, bet ir tai, kad tam tikru programos funkcionalumu tiesiog nebus įmanoma naudotis.

2.2.5. AvoidUnusedConstantsRule taisyklės pasiūlymas

Analizuojamas įrankis neaptinka nenaudojamos privačios konstantos. Nenaudojamą privačią konstantą galima prilyginti nenaudojamam kintamajam, nes jų trūkumai yra tie patys. Ši siūloma nauja taisyklės bandys aptikti visas nenaudojamas privačias konstantas.

Netinkamo programinio kodo pavyzdys:

```
public class Test
{
    private const string c_unusedConst = "Unused";
}
```

Ši nauja taisyklė taip pat turėtų padėti sumažinti klases ir taip pagerinti programų greitaveiką bei sumažinti naudojamos atminties kiekį.

2.2.6. AvoidRedundantIfConditionsRule taisyklės pasiūlymas

„Visual Studio 2010“ kompiliatorius aptinka klaidas kai lyginamas kintamasis pats su savimi (pvz.: $a == a$, $a < a$ ir pan.). Jis taip pat aptinka nepasiekiamą kodą, kai *if* sąlyga visuomet

false (pvz.: *if (5 < 5) DoSomething ()*). Tačiau nei kompiliatorius, nei „Gendarme“ įrankis neranda klaidų, kai sąlyga visuomet *true* ir nėra nepasiekiamo kodo.

Taigi siūlau sukurti taisyklę, kuri tikrins ar nėra perteklinių *if* sąlygų. Jei bus aptikta sąlyga ar jos dalis, kuri visuomet bus tenkinama, tai bus siūloma tokią sąlygą išmesti. Į šią taisyklę taip pat įtrauksime ir perteklinį *switch* operatorių. *switch* operatorius yra perteklinis, kai visuomet pereinama per visas operatoriaus sąlygas iki paskutinės.

Netinkamo programinio kodo pavyzdys:

```
public void AvoidRedundantConditions ()
{
    if (5 > 3 || 5 == 5 || 5 < 3)
         DoSomething ();

    switch (index)
    {
        case 1:
        case 2:
        case 3:
        default:
            DoSomething (5);
            break;
    }
}
```

Geras pavyzdys:

```
public void AvoidRedundantConditions ()
{
    DoSomething ();
    DoSomething (5);
}
```

Vėl gi, ši nauja taisyklė turėtų padėti supaprastini kodą. O tai turėtų padėti išvengti kvailų klaidų, kurios kyla dėl netvarkingo kodo.

2.2.7. AvoidVisibleEventHandlersRule taisyklės pasiūlymas

Tarkime turime viešą įvykį apdorojantį metodą. O tas metodas apdoroja telefono sąskaitos papildymo įvykį. Tokiu atveju galime suvaidinti sąskaitos papildymo įvykį ir iškviesti sąskaitos papildymą apdorojantį metodą.

Netinkamo programinio kodo pavyzdys:

```
public void ButtonKeyDown (object sender, KeyEventArgs e)
{
    DoSomething ();
}
```

Geras pavyzdys:

```
private void ButtonKeyDown (object sender, KeyEventArgs e)
{
    DoSomething ();
}
```

Norint to išvengti, įvykius apdorojantys metodai turėtų būti ne vieši. Taigi sukursime taisyklę, kuri ieškos viešų įvykius apdorojančių metodų ir siūlys juos pakeisti į ne viešus. Tai turėtų padėti pagerinti programų saugumą.

2.2.8. AvoidSQLInjectionRule taisyklės pasiūlymas

SQL injekcijos yra vienas iš tinklapių nulaužimo būdų, ir pats populiariausias pažeidžiamumas 2008-2009 metais [15]. Tai, galimybė puslapyje, naudojančiam SQL duomenų bazę, įvykdyti norimą užklausą, kurios pagalba galima išgauti informaciją esančią duomenų bazėje. SQL injekcija galima tuomet, kai puslapis netikrina vartotojo įvestų duomenų ir juos naudoja SQL užklausos formavimui. Šiuo metu „Gendarme“ įrankis neturi jokių taisyklių padedančiu aptikti SQL injekciją. Siūlau sukurti taisyklę, kuri bandys aptikti galimą SQL injekciją. Ji ieškos kodo vietų, kur sudarinėjant SQL komandos tekstą, bus naudojami kintamieji, o ne SQL komandos parametrai.

Netinkamo programinio kodo pavyzdys:

```
public class SqlQueries
{
    public static object AvoidSQLInjection
    (
    string connection,
    string name,
    string password
    )
    {
        SqlConnection someConnection = new SqlConnection (connection);
        object accountNumber = null;
        using (SqlCommand someCommand = new SqlCommand ())
        {
            someCommand.Connection = someConnection;
            someCommand.CommandText = "SELECT AccountNumber " +
               "FROM Users " +
               "WHERE Username='" + name +
               "' AND Password='" + password + "'";
            someConnection.Open ();
            accountNumber = someCommand.ExecuteScalar ();
            someConnection.Close ();
        }
        return accountNumber;
    }
}
```

Šiuo atveju, jei į vartotojo vardo laukelį įrašysime „' OR 1=1 --“, rezultate gausime tokią užklausą „SELECT AccountNumber FROM Users WHERE Username=” OR 1=1“. Akivaizdu, jog WHERE sąlyga bus visuomet tenkinama.

Geras pavyzdys:

```
public class SqlQueries
{
    public static object AvoidSQLInjection
    (
    string connection,
    string name,
    string password
    )
    {
        SqlConnection someConnection = new SqlConnection (connection);
        object accountNumber = null;
        using (SqlCommand someCommand = new SqlCommand ())
        {
            someCommand.Connection = someConnection;
            someCommand.Parameters.Add(
                "@username", SqlDbType.NChar).Value = name;
            someCommand.Parameters.Add(
```

```
                "@password", SqlDbType.NChar).Value = password;
            someCommand.CommandText = "SELECT AccountNumber " +
                "FROM Users " +
                "WHERE Username=@username AND Password=@password";
            someConnection.Open ();
            accountNumber = someCommand.ExecuteScalar ();
            someConnection.Close ();
        }
        return accountNumber;
    }
}
```

Šiuo atveju į varotojo vardo laukelį įrašę tą pačią reikšmę („’ OR 1=1 --“) gausime jau kitokią (SELECT AccountNumber FROM Users WHERE Username=”” OR 1=1 --’ AND Password=’anything’) užklausą. Ši užklausa nesugadinta, ji ieškos vartotojo kurio vardas „’ OR 1=1 --“.

2.3. Siūlomų patobulinimų ir naujų taisyklių nauda

Siūlomi patobulinimai bei naujos taisyklės visų pirma turėtų pagerinti programų saugumą. Be to patobulinimai turėtų padėti sumažinti klases ir taip pagerinti programų greitaveiką. Taisyklių rinkinio išplėtimas padės sumažinti naudojamos atminties kiekį, padės sumažinti *NullReferenceException* išimties tikimybę. O tai savo ruožtu padėtų išvengti programos „nulūžimo“ ar klaidų pranešimų galutiniams vartotojams. Taip pat turėtų suteikti programiniam kodui daugiau lankstumo, paprastumo ir aiškumo. Paprastas ir aiškus kodas palengvina programų skaitomumą ir padeda išvengti kvailų klaidų, kurios kyla dėl netvarkingo kodo.

3. „GENDARME“ ĮRANKIO PATOBULINIMAI

Siekiant patobulinti statinės kodo analizės įrankį „Gendarme“ buvo pasiūlyti esamų taisyklių patobulinimai (aprašyti 2.1 skyrelyje) ir naujų taisyklių sukūrimas (2.2 skyrelis). Patobulinimų įgyvendinimui buvo naudojama „Visual Studio 2010“ programavimo aplinka ir C# programavimo kalba. Naujų ir patobulintų taisyklių veiksmingumo tyrimas aprašytas 4 skyriuje.

3.1. Esamų taisyklių patobulinimai

Įgyvendinti visi siūlyti taisyklių patobulinimai. Be to buvo patobulinta RemoveUnusedLocalVariablesRule taisyklė, nes tam tikrus siūlytus AvoidUnusedPrivateFieldsRule taisyklės patobulinimus buvo kur kas teisingiau ir patogiau įgyvendinti šioje taisyklėje.

3.1.1. ReviewLockUsedOnlyForOperationsOnVariablesRule taisyklės patobulinimo įgyvendinimas

ReviewLockUsedOnlyForOperationsOnVariablesRule taisyklė priklauso „Gendarme.Rules.Concurrency“ kategorijai ir tikrina ar *lock* operatorius yra naudojamas operacijoms atlikti tik su klasės ar lokaliais kintamaisiais. Jei vienintelis tikslas kritinėje dalyje yra užtikrinti, kad kintamasis bus pakeistas automatiškai, tuomet geresnis būdas yra naudoti *System.Threading.Interlocked*.

Blogo programinio kodo pavyzdys, kuriame programa neaptikdavo klaidos:

```
lock (m_lockObject)
{
    count++;
    m_someSharedObject = anotherObject;
    Interlocked.Add (pages, 5);
}
```

Gero programinio kodo pavyzdys:

```
Interlocked.Increment (count);
Interlocked.Exchange (m_someSharedObject, anotherObject);
Interlocked.Add (pages, 5);
```

Taisyklė tikrina visas *lock* viduje esančias operacijas kol aptinka tokią, kuri nėra automatinė operacija su kintamuoju. Jei tokia operacija neaptinkama visoje zonoje, tuomet registruojama klaida. Patobulinta taisyklė dabar atpažįsta visus *System.Threading.Interlocked* klasės metodus kaip automatinius metodus su kintamaisiais. Taigi algoritmas nebenutraukiamas aptikus *Iterlocked* klasės metodą.

3.1.2. CheckParametersNullityInVisibleMethodsRule taisyklės patobulinimo įgyvendinimas

CheckParametersNullityInVisibleMethodsRule taisyklė priklauso „Gendarme.Rules.Correctness“ kategorijai. Ji sako, jog matomų metodų parametrai turėtų būti patikrinti ar ne *null* prieš juos naudojant.

Netinkamo programinio kodo pavyzdys, kuriame programa neaptikdavo klaidos:

```
public void CheckParametersNullityInVisibleMethodsRule (string name)
{
    string nameCopy = string.Copy (name);
    if (name.Length > 10)
        Console.WriteLine ("Name: " + name);
}
```

Gero programinio kodo pavyzdys:

```
public void CheckParametersNullityInVisibleMethodsRule (string name)
```

```
{
    string nameCopy = string.Copy (name);
    if (name != null && name.Length > 10)
        Console.WriteLine ("Name: " + name);
}
```

Tačiau ši taisyklė nebeaptikdavo klaidos, jei prieš tai būdavo iškviečiamas bet koks metodas ir jam kaip parametras perduodamas šio metodo parametras. Patobulinta taisyklė šio veiksmo nebetraktuoja kaip *null* reikšmės patikrinimo. Tačiau išlieka dar viena problema, algoritmas tik aptinka ar yra patikrinimas *param != null* ar *param == null* tačiau netikrina kas vyksta toliau. O po to gali sekti toks veiksmas: *if (param == null) param.Update ()*, kurį vykdant programa savo darbą baigs nenumatytu būdu. Tačiau šią spragą turėtų aptikti nauja AvoidUsingNullAfterNullityCheckRule taisyklė.

3.1.3. ConsiderAddingInterfaceRule taisyklės patobulinimo įgyvendinimas

ConsiderAddingInterfaceRule taisyklė priklauso „Gendarme.Rules.Design“ kategorijai. Ji praneša apie klaidą, jei klasė įgyvendina visus sąsajos narius, bet nepaveldi pačios sąsajos. Tačiau ši taisyklė nebesiūlydavo paveldėti sąsajos, jei klasės ir sąsajos nariai skirdavosi didžiosiomis ir mažosiomis raidėmis.

Netinkamo programinio kodo pavyzdys, kai programa nesiūlydavo paveldėti sąsajos:

```
public interface ITest
{
    void CreateDGNFile ();
    bool IsECInstance ();
}

public class Testas
{
    static void CreateDgnFile () {}
    bool IsEcInstance () { return true; }
}
```

Patobulinta taisyklė nebeatsižvelgia į didžiąsias ir mažąsias raides. Ieškant metodo su atitinkamu pavadinimu dabar naudojamas *StringComparison.Ordinal* palyginimo tipas. Taip pat patobulinta taisyklė palyginimuose dabar naudoja ir statinius metodus.

3.1.4. AvoidUnusedPrivateFieldsRule taisyklės patobulinimo įgyvendinimas

AvoidUnusedPrivateFieldsRule taisyklė priklauso „Gendarme.Rules.Performance“ kategorijai. Taisyklė skirta nenaudojamiems privatiesiems klasės kintamiesiems aptikti. Patobulinta taisyklė neaptinka klaidos tik tuomet, kai privataus kintamojo reikšmė yra nuskaitoma kurioje nors klasės operacijoje. Taip pat patobulinta taisyklė aptinka ir nenaudojamas privačias konstantas.

Netinkamo programinio kodo pavyzdys, kuriame programa neaptikdavo klaidos:

```
public class Testas
{
    private string m_unused;
    private const string m_unusedConst;

    public void AvoidUnusedPrivateFieldsRule ()
    {
        m_unused = "Unused private variable.";
        m_unusedConst = "Unused private const.";
        DoSomething ();
    }
}
```

Gero programinio kodo pavyzdys:

```
public class Testas
{
    public void AvoidUnusedPrivateFieldsRule ()
    {
        DoSomething ();
    }
}
```

Patobulinta taisyklė tikrina visas klasės operacijas tol, kol aptinka tokią, kurioje yra nuskaitoma kintamojo reikšmė. Anksčiau buvo ieškoma operacijos kur naudojamas kintamasis, neįvertinant ar jam reikšmė priskiriama ar nuskaitoma.

3.1.5. RemoveUnusedLocalVariablesRule taisyklės patobulinimo įgyvendinimas

RemoveUnusedLocalVariablesRule taisyklė priklauso „Gendarme.Rules.Performance“ kategorijai. Taisyklė skirta nenaudojamiems metodo kintamiesiems aptikti. Patobulinta taisyklė neaptinka klaidos tik tuomet, kai metodo kintamojo reikšmė yra nuskaitoma kurioje nors metodo operacijoje.

Netinkamo programinio kodo pavyzdys, kuriame programa neaptikdavo klaidos:

```
public void AvoidUnusedLocalVariablesRule ()
{
    String unused = "Unused local variable.";
    DoSomething ();
}
```

Gero programinio kodo pavyzdys:

```
public void AvoidUnusedLocalVariablesRule ()
{
    DoSomething ();
}
```

Taisyklė tikrina visas klasės operacijas tol, kol aptinka tokią, kurioje yra nuskaitoma kintamojo reikšmė. Anksčiau buvo ieškoma operacijos kur naudojamas kintamasis, neįvertinant ar jam reikšmė priskiriama ar nuskaitoma.

3.1.6. ArrayFieldsShouldNotBeReadOnlyRule taisyklės patobulinimo įgyvendinimas

ArrayFieldsShouldNotBeReadOnlyRule taisyklė priklauso „Gendarme.Rules.Security“ kategorijai. Ši taisyklė aptinka masyvus, kurie yra deklaruoti kaip *public readonly* ir praneša apie klaidą, nes tokiu atveju negalima pakeisti pačio masyvo, bet galima pakeisti masyvo reikšmes. Patobulinta taisyklė aptinka ir taip pat deklaruotus sąrašus ir rinkiniu (angl. *collection*), nes jiems galioja ta pati saugumo spraga.

Pavyzdys, kuriame programa neaptikdavo klaidos:

```
public class Testas
{
     public readonly System.Collections.ObjectModel.Collection<int>
readOnlyCollection = new Collection<int> ();
     public readonly System.Collections.Generic.ICollection<int>
readOnlyICollection = new List<int> ();
}
```

Gero programinio kodo pavyzdys:

```
public class Testas
{
     private readonly System.Collections.ObjectModel.Collection<int>
```

```
readOnlyCollection = new Collection<int> ();
     private readonly System.Collections.Generic.ICollection<int>
readOnlyICollection = new List<int> ();
    public int[] GetCollectionAsArray ()
    {
    int[] array = new int[readOnlyCollection.Count];
    readOnlyCollection.CopyTo (array, 0);
    return array;
    }

    public int[] GetICollectionAsArray ()
    {
    int[] array = new int[readOnlyICollection.Count];
    readOnlyICollection.CopyTo (array, 0);
    return array;
    }
}
```

Taisyklė tikrina visų klasių kintamuosius. Pirmiausiai patikrinama ar kintamąjį galima tik skaityti ir ar jis yra viešas. Po to tikrinamas tipas ir, jei jis yra masyvas, sąrašas ar rinkinys, spausdinamas klaidos pranešimas.

3.2. Naujos taisyklės

Įgyvendinta dauguma pasiūlytų taisyklių. Neįgyvendinta tik AvoidRedundantIfConditionsRule taisyklė, nes perteklinis kodas buvo optimizuotas kompiliavimo metu. Taip pat vietoj naujos AvoidUnusedConstantsRule taisyklės buvo patobulinta AvoidUnusedPrivateFieldsRule taisyklė.

Naujų taisyklių pridėjimas ganėtinai paprastas. Nauja taisyklė turi paveldėti abstrakčią *Rule* klasę, kuri savo ruožtu paveldi ir implementuoja *IRule* sąsają. To pakanka, kad taisyklė būtų aptikta ir sukurtas jos objektas. Kad taisyklė kažką atliktų, reikia paveldėti ir implementuoti *ITypeRule* ar *IMethodRule*, ar abi sąsajas. *ITypeRule* turi *CheckType* metodą, kuris iškviečiamas kiekvienai klasei, sąsajai, delegatui ir pan. *IMethodRule* turi CheckMethod metodą, kuris iškviečiamas kiekvienam klasės metodui. Klasės ar metodo analizės metu aptikus klaidą galima iškarto ją užregistruoti iškvietus *Runner*.*Report* metodą. Šis metodas naudoja taisyklės *Problem* ir *Solution* atributus, kurie nurodo kokia problema ir kaip ją reikėtų pataisyti.

3.2.1. AvoidCaseSensitiveNamesRule taisyklės įgyvendinimas

Nauja AvoidCaseSensitiveNamesRule taisyklė priskirta „Gendarme.Rules.Naming“ kategorijai. Ji aptinka metodų bei klasės ir metodų kintamųjų pavadinimus, kurie skiriasi tik didžiosiomis ar mažosiomis raidėmis ir praneša apie klaidas, jei tokių pavadinimų randama. Klasės ir metodų kintamųjų pavadinimai lyginami atskirai.

Netinkamo programinio kodo pavyzdys, kurio reikia vengti:

```
private bool   checkBox;
private string checkbox;
public void AvoidCaseSensitiveNamesRule (int count)
{
    int    value = count;
    string Value = count.ToString ();
    DoSomething (value, Value);
}

public void AvoidCaseSensitiveNamesRULE ()
{
    DoSomething ();
```

```
}
```

Geras pavyzdys:

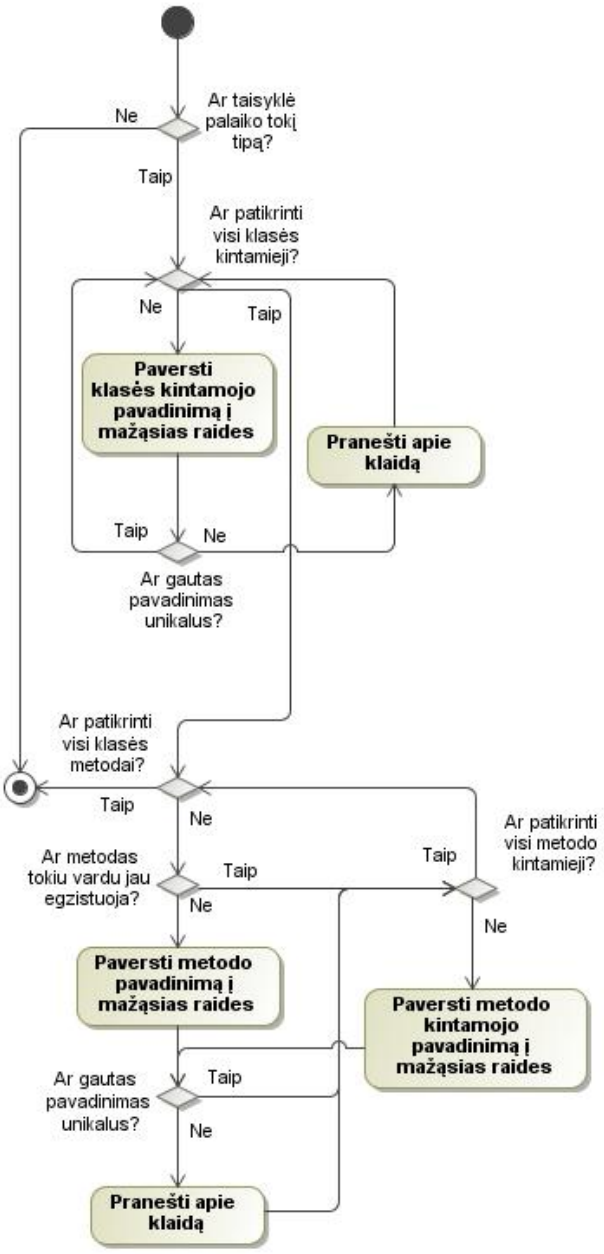
```
private bool   checkBoxState;
private string checkBoxName;
public void AvoidCaseSensitiveNamesRule (int count)
{
    int    intValue    = count;
    string stringValue = count.ToString ();
    DoSomething (intValue, stringValue);
}

public void AvoidCaseSensitiveNamesRule ()
{
    DoSomething ();
}
```

```
1 [Problem("This type contains fields, methods or method variables which are different
only in case.")]
2 [Solution("Rename the names which are different only in case, because it is confusing
and hard to maintain.")]
3 public class AvoidCaseSensitiveNamesRule : Rule, ITypeRule
4 {
5
6     public RuleResult CheckType (TypeDefinition type)
7     {
8         // rule does not apply to enums and delegates
9         // nor does it apply to generated code
10        if (type.IsEnum || type.IsDelegate() || type.IsGeneratedCode())
11            return RuleResult.DoesNotApply;
12
13        ICollection<String> namesInLowerCase = new List<String>();
14        foreach (FieldDefinition field in type.Fields)
15        {
16            if (!IsNameUnique (field.Name, namesInLowerCase))
17                Runner.Report (field, Severity.Medium, Confidence.Normal);
18        }
19
20        // scan all methods
21        namesInLowerCase.Clear();
22        ICollection<String> normalNames = new List<String>();
23        foreach (MethodDefinition method in type.Methods)
24        {
25            // methods with the same names can exist.
26            if (!normalNames.Contains(method.Name))
27            {
28                if (IsNameUnique (method.Name, namesInLowerCase))
29                    normalNames.Add(method.Name);
30                else
31                    Runner.Report(method, Severity.Medium, Confidence.Normal);
32            }
33
34            if (!method.HasBody)
35                continue;
36
37            ICollection<String> methodVariableNames = new List<String>();
38            foreach (VariableDefinition variable in method.Body.Variables)
39            {
40                if (!IsNameUnique (variable.Name, methodVariableNames))
41                    Runner.Report(variable, Severity.Medium, Confidence.Normal);
42            }
43        }
44
45        return Runner.CurrentRuleResult;
46    }
47
48    private bool IsNameUnique (String name, ICollection<String> namesInLowerCase)
49    {
```

```
50          String nameInLowerCase = name.ToLowerInvariant ();
51          if (namesInLowerCase.Contains (nameInLowerCase))
52              return false;
53
54          namesInLowerCase.Add (nameInLowerCase);
55          return true;
56      }
57}
```

Aukščiau pateiktas naujos taisyklės kodas bei 3.1 pav. veiklos diagrama. AvoidCaseSensitiveNamesRule klasė privalomai paveldi abstrakčią *Rule* klasę ir implementuoja *ITypeRule* sąsają. *CheckType* metodas yra iškviečiamas kiekvienai klasei. Pirmiausiai patikrinami visų klasės kintamųjų vardai (14-18 eil.). Kintamojo pavadinimas verčiamas į mažąsias raides (50 eil.) ir jei kintamojo su tokiu pavadinimu klasės kintamųjų sąraše dar nėra (51 eil.), tuomet jis įtraukiamas į šį sąrašą (54 eil.). Tačiau, jei tokiu pavadinimu jau yra, tuomet registruojama klaida (17 eil.). Registruojant klaidą reikia nurodyti klaidos sunkumą ir jos tikrumą. Taip pat registruojant klaidą, jos problematika ir galimas sprendimas paimamas iš klasės *Problem* ir *Solution* atributų.



3.1 pav. AvoidCaseSensitiveNamesRule taisyklės veiklos diagrama

Vėliau analizuojami metodų pavadinimai. Čia jau nepakanka vieno sąrašo su metodų pavadinimais mažosiomis raidėmis, nes keli metodai tokiu pat pavadinimu tik su skirtingais parametrais gali egzistuoti. Čia prisideda patikrinimas ar originalių metodų pavadinimų sąraše toks metodo pavadinimas jau egzistuoja (26 eil.). Jei ne, tuomet vykdoma ta pati logika kaip ir su klasės kintamųjų vardais (28-31 eil.). Jei taip, tuomet šis pavadinimas yra tinkamas ir toliau tikrinami metodo kintamųjų vardai (37-41 eil.). Tačiau prieš tai dar patikrinama ar tas metodas turi realizaciją (34 eil.), jei jis jos neturi, tuomet neturės ir metodo kintamųjų.

Metodo kintamųjų pavadinimai tikrinami pagal tą patį principą, kaip ir klasės kintamųjų pavadinimai.

3.2.2. AvoidReturnSameValueRule taisyklės įgyvendinimas

Nauja AvoidReturnSameValueRule taisyklė priskirta „Gendarme.Rules.Maintainability“ kategorijai. Ji tikrina ar metodas gali grąžinti kelias reikšmes. Jei jis visuomet grąžina vieną ir tą pačią reikšmę, tuomet registruojama klaida. Tai turėtų palengvinti programos skaitomumą ir padėti išvengti išsiblaškymo klaidų.

Netinkamo programinio kodo pavyzdys:

```
public bool DoSomething (int index)
{
    if (index < 0)
        return false;

    DoSomethingMore (index);
    return false;
}
```

Geras pavyzdys:

```
public bool DoSomething (int index)
{
    if (index < 0)
        return false;

    DoSomethingMore (index);
    return true;
}
```

Algoritmas lygina visas grąžinamas reikšmes su pirmąja tol, kol jos skiriasi. Jei visos gražinamos reikšmės sutampa su pirmąja, tai reiškia, kad jos sutampa ir tarpusavyje. Tokiu atveju registruojama klaida. Jei algoritmas aptinka, kad grąžinama reikšmė yra kintamasis, parametras, kito metodo grąžinama reikšmė (pvz.: *x.Count()*), išraiška ar *bool* tipo metoduose palyginimas (pvz.: *return a > b;*), tuomet traktuojama, kad grąžinamos skirtingos reikšmės.

3.2.3. AvoidUsingNullAfterNullityCheckRule taisyklės įgyvendinimas

Nauja AvoidUsingNullAfterNullityCheckRule taisyklė priskirta „Gendarme.Rules.Design“ kategorijai. Ši taisyklė turėtų padėti sumažinti *NullReferenceException* išimties tikimybę. O tai savo ruožtu padėtų išvengti programos „nulūžimo“ ar klaidų pranešimų galutiniams vartotojams.

Netinkamo programinio kodo pavyzdys:

```
public void AvoidUsingNullVariables (string param)
{
    if (param != null || param.Length > 3)
        Console.WriteLine ("Parameter is acceptable.");

    Console.WriteLine ("Parameter is NOT acceptable.");
}
```

Geras pavyzdys:

```
public void AvoidUsingNullVariables (string param)
{
    if (param != null && param.Length > 3)
        Console.WriteLine ("Parameter is acceptable.");

    Console.WriteLine ("Parameter is NOT acceptable.");
}
```

Šios taisyklės algoritmas ieško sąlygų, kuriose tikrinama ar kintamasis, parametras lygus *null*. Jei taip, tuomet pasirenkama ta šaka, kai kintamasis lygus *null* ir tikrinami sekantys veiksmai iki metodo galo arba iki kol kintamajam priskiriama nauja reikšmė. Tikrinama ar nenaudojamas (nebandoma kviesti jo metodų) šis kintamasis, kurio reikšmė lygi *null*.

3.2.4. AvoidInfiniteLoopRule taisyklės įgyvendinimas

Nauja AvoidInfiniteLoopRule taisyklė priskirta „Gendarme.Rules.Design“ kategorijai. Ji aptiks amžinus ciklus tuomet, kai ciklo viduje neaptiks *break* ar *return* komandos arba kai prieš juos esančiose sąlygose kintamieji negalės įgyti naujų reikšmių.

Netinkamo programinio kodo pavyzdys:

```
public void AvoidInfiniteLoop ()
{
    int a = 0;
    while (true)
        DoSomething (a++);

    while (a > 0)
        DoSomething ();
}
```

Geras pavyzdys:

```
public void AvoidInfiniteLoop ()
{
    int a = 0;
    while (true)
    {
        if (DoSomething (a++))
            break;
    }

    while (a > 0)
    {
        DoSomething ();
        a--;
    }
}
```

Šios taisyklės algoritmas pirmiausiai ieško ciklo, o jį aptikti yra gana nesunku. Jei aptinkama, kad x instrukcijoje yra šuolis žemyn į x+y instrukciją, o po to iš x+y+z instrukcijos atitikus tam tikroms sąlygoms vėl šokama aukštyn į x+1 instrukciją, tai reiškia aptikome ciklą. Visos instrukcijos esančios tarp x+1 ir x+y-1 yra veiksmai cikle. O ciklo sąlygos veiksmai aprašyti instrukcijose tarp x+y ir x+y+z.

Aptikus ciklą susirenkami visi sąlygoje naudojami kintamieji. Po to tikrinamos visos ciklo vidaus instrukcijos ir bandoma aptikti ar sąlygoje naudojami kintamieji gali įgyti naujas reikšmes. Taip pat ar egzistuoja *break*, ar *return* instrukcijos. Jei tokios sąlygos neaptinkamos, tuomet pranešama apie klaidą.

3.2.5. EventHandlersShouldNotBeVisibleRule taisyklės įgyvendinimas

Nauja EventHandlersShouldNotBeVisibleRule taisyklė priskirta „Gendarme.Rules.Security“ kategorijai. Taisyklė tikrina ar įvykius apdorojantys metodai yra vieši. Jei taip, tuomet registruojama klaida, nes šis įvykis gali būti suvaidinamas.

Netinkamo programinio kodo pavyzdys:

```
public void ButtonKeyDown (object sender, KeyEventArgs e)
{
```

```
    DoSomething ();
}
```

Geras pavyzdys:

```
private void ButtonKeyDown (object sender, KeyEventArgs e)
{
    DoSomething ();
}
```

Taisyklėje įgyvendintas algoritmas ieško vietų, kur yra sukuriama nauja klasė ir jai kaip parametras perduodamas metodas. Jei perduodamas metodas yra viešas, tuomet registruojama klaida.

3.2.6. AvoidSQLInjectionRule taisyklės įgyvendinimas

Nauja AvoidSQLInjectionRule taisyklė skirta aptikti galimas SQL injekcijas. Ji priskirta „Gendarme.Rules.Security“ kategorijai. Ši taisyklė turėtų padėti pagerinti programų saugumą.

Netinkamo programinio kodo pavyzdys:

```
public class SqlQueries
{
    public static object AvoidSQLInjection
    (
    string connection,
    string name,
    string password
    )
    {
        SqlConnection someConnection = new SqlConnection (connection);
        object accountNumber = null;
        using (SqlCommand someCommand = new SqlCommand ())
        {
            someCommand.Connection = someConnection;

            someCommand.CommandText = "SELECT AccountNumber " +
               "FROM Users " +
               "WHERE Username='" + name +
               "' AND Password='" + password + "'";
            someConnection.Open ();
            accountNumber = someCommand.ExecuteScalar ();
            someConnection.Close ();
        }
        return accountNumber;
    }
}
```

Taisyklė ieško vietų kur yra priskiriamas *SqlCommand.CommandText* kintamasis. Jei priskyrimo metu naudojami kintamieji, o ne komandos parametrai, tuomet registruojama klaida.

3.3. Įgyvendintų patobulinimų ir naujų taisyklių apibendrinimas

Įgyvendinti visi siūlyti taisyklių patobulinimai. Taip pat buvo patobulinta RemoveUnusedLocalVariablesRule taisyklė, nes tam tikrus siūlytus AvoidUnusedPrivateFieldsRule taisyklės patobulinimus buvo kur kas teisingiau ir patogiau įgyvendinti šioje taisyklėje.

Įgyvendintos ir dauguma pasiūlytų naujų taisyklių. Neįgyvendinta tik AvoidRedundantIfConditionsRule taisyklė, nes perteklinis kodas buvo optimizuotas kompiliavimo metu. Taip pat vietoj naujos AvoidUnusedConstantsRule taisyklės buvo patobulinta AvoidUnusedPrivateFieldsRule taisyklė, nes taip buvo kur kas patogiau.

4. ATVIRO KODO PROGRAMŲ TESTAVIMAS

Eksperimentinio tyrimo metu buvo išanalizuotas 5 atviro kodo C# programų kodas. Žemiau pateikiamos aptiktos klaidos programose. Programinio kodo eilučių kiekis apskaičiuotas pasinaudojant „Cloc v1.58“ programa.

Programos pasirinktos iš sourceforge.net puslapio. Buvo pasirinktas filtras ieškoti tik angliškų, parašytų C# programavimo kalba, skirtų „Windows“ operacinėms sistemoms (įrankis sukurtas tokiai OS) ir prieinamų vartotojams, stabilių. Surikiavus rezultatus pagal populiarumą buvo pasirinktos programos, kurias sudaro nuo 15 iki 50 tūkstančių programinio kodo eilučių. Kur kodo eilučių mažiau, ten aptinkamos tik kelios naujos klaidos ir dažniausiai tos pačios taisyklės jas aptinka. Kur kodo eilučių daugiau, ten aptinkama dešimtys tūkstančių klaidų, o norint patikrinti naujai aptiktas klaidas reikėtų daug laiko.

Taip pat buvo stengtasi pasirinkti programas, kuriose saugumas yra svarbus arba kurios skirtos saugumui užtikrinti. Pvz. žaidimuose ar turinio peržiūros programose jis nėra labai svarbus.

4.1. „ZedGraph“ programos kodo analizė

„ZedGraph v515“ yra klasių biblioteka skirta įvairių 2D grafikų, stulpelinių ir skritulinių diagramų braižymui. Biblioteką sudaro 21520 C# programinio kodo eilučių (be komentarų ir tuščių eilučių). Patobulintų taisyklių rezultatai analizuojant šią programą pateikti 4.1 lentelėje.

4.1 lentelė. Patobulintų taisyklių resultatai „ZedGraph“ programoje

Nr.	Taisyklė	Aptiktos klaidos		Skirtumas
		Prieš	Po	
1.	ReviewLockUsedOnlyForOperationsOnVariablesRule	2	2	0
2.	CheckParametersNullityInVisibleMethodsRule	239	366	127
3.	ConsiderAddingInterfaceRule	0	0	0
4.	AvoidUnusedPrivateFieldsRule	0	0	0
5.	RemoveUnusedLocalVariablesRule	0	83	83
6.	ArrayFieldsShouldNotBeReadOnlyRule	2	2	0
	Viso:	**243**	**453**	**210**

Iš rezultatų pateiktų lentelėje matyti, kad 2 patobulintos taisyklės aptiko kur kas daugiau klaidų nei kitos. CheckParametersNullityInVisibleMethodsRule taisyklė aptiko 366 klaidas, t.y. 127 klaidomis daugiau nei prieš patobulinimus. RemoveUnusedLocalVariablesRule taisyklė aptiko 83 klaidas. Kitos dvi taisyklės aptiko po dvi klaidas, tačiau neaptiko naujų klaidų. Viso aptikta 210 klaidų daugiau nei prieš patobulinimus.

Naujos taisyklės šioje programoje neaptiko jokių klaidų. Iš viso prieš patobulinimus buvo aptiktos 2874 klaidos, o po jų 3084.

4.2. „PDFsharp“ programos kodo analizė

„PDFsharp v1.32“ programa skirta PDF failų kūrimui ir redagavimui. Ją sudaro 52600 C# programinio kodo eilučių. Šios programos analizės rezultatai pateikiami 4.2 ir 4.3 lentelėse.

4.2 lentelė. Patobulintų taisyklių rezultatai „PDFsharp“ programoje

Nr.	Taisyklė	Aptiktos klaidos		Skirtumas
		Prieš	Po	
1.	ReviewLockUsedOnlyForOperationsOnVariablesRule	0	0	0
2.	CheckParametersNullityInVisibleMethodsRule	68	79	11
3.	ConsiderAddingInterfaceRule	0	0	0
4.	AvoidUnusedPrivateFieldsRule	0	13	13
5.	RemoveUnusedLocalVariablesRule	0	23	23
6.	ArrayFieldsShouldNotBeReadOnlyRule	0	0	0
	Viso:	**68**	**115**	**47**

Analizuojant šios programos kodą 3 patobulintos taisyklės aptiko daugiau klaidų nei prieš patobulinimus. Dvi tos pačios kaip ir „ZedGraph“ programos atveju - CheckParametersNullityInVisibleMethodsRule ir RemoveUnusedLocalVariablesRule. Jos aptiko 34 klaidomis daugiau. Be to 13 klaidų aptiko AvoidUnusedPrivateFieldsRule taisyklė. Viso patobulintos taisyklės aptiko 47 klaidomis daugiau.

4.3 lentelė. Naujų taisyklių rezultatai „PDFsharp“ programoje

Nr.	Taisyklė	Aptiktos klaidos
1.	AvoidCaseSensitiveNamesRule	185
2.	AvoidReturnSameValueRule	7
3.	AvoidUsingNullAfterNullityCheckRule	0
4.	AvoidInfiniteLoopRule	0
5.	EventHandlersShouldNotBeVisibleRule	0
6.	AvoidSQLInjectionRule	0
	Viso:	**192**

Analizuojant šią programą klaidų aptiko ir dvi naujos taisyklės. AvoidCaseSensitiveNamesRule taisyklė aptiko 185 klaidas. AvoidReturnSameValueRule taisyklė aptiko 7 klaidas. O iš viso prieš patobulinimus buvo aptiktos 5249 klaidos. Po jų – 5488 klaidos.

4.3. „NetworkMiner“ programos kodo analizė

„NetworkMiner v1.4.1“ programa skirta tinklo srautams stebėti ir analizuoti. Jos paskirtis – padėti pagerinti tinklo saugumą. Ją sudaro 18065 C# programinio kodo eilutės. Šios programos analizės rezultatai pateikiami 4.4 ir 4.5 lentelėse.

4.4 lentelė. Patobulintų taisyklių rezultatai „NetworkMiner“ programoje

Nr.	Taisyklė	Aptiktos klaidos		Skirtumas
		Prieš	Po	
1.	ReviewLockUsedOnlyForOperationsOnVariablesRule	0	0	0
2.	CheckParametersNullityInVisibleMethodsRule	65	69	4
3.	ConsiderAddingInterfaceRule	30	30	0
4.	AvoidUnusedPrivateFieldsRule	8	75 (63)	67 (55)
5.	RemoveUnusedLocalVariablesRule	0	29	29
6.	ArrayFieldsShouldNotBeReadOnlyRule	0	0	0
	Viso:	**103**	**203**	**100**

Tos pačios 3 taisyklės kaip anksčiau aptiko daugiau klaidų. Bendrai jos aptiko 100 klaidų daugiau nei prie patobulinimus. Tiesa AvoidUnusedPrivateFieldsRule taisyklė aptiko 67 klaidomis daugiau, tačiau 12 iš jų nėra teisingos (skliausteliuose nurodytas tikrų klaidų kiekis). Ši taisyklė blogai aptinka klaidas kai metoduose naudojamas *yield* raktažodis (angl. *keyword*). Taip buvo ir prieš patobulinimus, tačiau tikimybė aptikti neteisingą klaidą buvo kur kas mažesnė, nes pakakdavo aptikti, kad kintamajam priskiriama ar nuskaitoma reikšmė. Dabar reikia aptikti, kad reikšmė yra nuskaitoma. Ir jei tai yra atliekama metode su *yield* raktažodžiu, tuomet kintamojo nuskaitymas nėra aptinkamas.

4.5 lentelė. Naujų taisyklių rezultatai „NetworkMiner“ programoje

Nr.	Taisyklė	Aptiktos klaidos
1.	AvoidCaseSensitiveNamesRule	0
2.	AvoidReturnSameValueRule	1
3.	AvoidUsingNullAfterNullityCheckRule	3
4.	AvoidInfiniteLoopRule	2
5.	EventHandlersShouldNotBeVisibleRule	3
6.	AvoidSQLInjectionRule	0
	Viso:	**9**

Net 4 naujos taisyklės šioje programoje aptiko klaidų. Viso jos aptiko 9 naujas klaidas. Tiesa AvoidInfiniteLoopRule taisyklės aptiktos klaidos tikriausiai būtų ignoruojamos, nes amžinas ciklas ten naudojamas specialiai. Amžini ciklai ten vykdomi skirtingose gijose. Jie apdoroja įvairius ateinančius pranešimus. Prieš patobulinimus įrankis aptiko 3216 klaidas, o po jų 3325 klaidas.

4.4. „WatiN“ programos kodo analizė

„WatiN v2.1“ programą sudaro 28265 C# programinio kodo eilutės. Ši programa skirta palengvinti Web programų testavimo automatizavimą. Tyrimo rezultatai pateikiami žemiau.

4.6 lentelė. Patobulintų taisyklių rezultatai „WatiN“ programoje

Nr.	Taisyklė	Aptiktos klaidos		Skirtumas
		Prieš	Po	
1.	ReviewLockUsedOnlyForOperationsOnVariablesRule	3	3	0
2.	CheckParametersNullityInVisibleMethodsRule	80	97 (87)	17 (7)
3.	ConsiderAddingInterfaceRule	3	3	0
4.	AvoidUnusedPrivateFieldsRule	0	4 (0)	4 (0)
5.	RemoveUnusedLocalVariablesRule	0	2	2
6.	ArrayFieldsShouldNotBeReadOnlyRule	1	4	3
	Viso:	**87**	**113**	**26**

Visos tobulintos taisyklės šioje programoje aptiko klaidų. 4 iš jų aptiko daugiau klaidų nei prieš patobulinimus. Tačiau vėl AvoidUnusedPrivateFieldsRule taisyklė aptiko netikrų klaidų kur yra naudojamas *yield* raktažodis. CheckParametersNullityInVisibleMethodsRule taisyklės aptiktos klaidos teisingos, tačiau vienoje vietoje parametras yra naudojamas tarp *try-catch* raktažodžių, o gaudomos visos išimtys. Tokiu atveju, net ir esant *null* parametro reikšmei, programa nebaigs darbo nenumatytu būdu. Taip pat ši programa turi metodą (*CanHandleWindow*) skirtą patikrinti ar *Window* tipo parametras gali būti naudojamas. Ši taisyklė tokios funkcijos kvietimo nesupranta kaip *nullity* patikrinimo ir dėl to užfiksavo 10 netikrų klaidų.

4.7 lentelė. Naujų taisyklių rezultatai „WatiN“ programoje

Nr.	Taisyklė	Aptiktos klaidos
1.	AvoidCaseSensitiveNamesRule	1
2.	AvoidReturnSameValueRule	4
3.	AvoidUsingNullAfterNullityCheckRule	4
4.	AvoidInfiniteLoopRule	0
5.	EventHandlersShouldNotBeVisibleRule	0
6.	AvoidSQLInjectionRule	0
	Viso:	**9**

Naujos taisyklės šioje programoje aptiko 9 klaidas. Po 4 klaidas aptiko AvoidReturnSameValueRule ir AvoidUsingNullAfterNullityCheckRule taisyklės. Vieną klaidą aptiko AvoidCaseSensitiveNamesRule taisyklė. Be to analizuojant šią programą EventHandlersShouldNotBeVisibleRule taisyklė aptiko 70 viešų įvykius apdorojančių metodų. Tačiau paaiškėjo, kad visi šie metodai tai liamda išraiškos (angl. *lambda expressions*) sudarytos panaudojant => operatorių. Įdomu tai, kad šios išraiškos sudaromos metodo viduje, tačiau yra viešos. Ši taisyklė buvo patobulinta, kad neaptiktų liamdą išraiškų kaip įvykius apdorojančių metodų. Po patobulinimų aptiktos 1848 klaidos, prieš juos buvo aptinkama 1813 klaidų.

4.5. „ASProxy“ programos kodo anlizė

„ASProxy v5.5“ programą sudaro 14450 C# programinio kodo eilučių. Ši programa paslepia duomenis apie vartotoją ir leidžia jam naršyti internete anonimiškai. Ji skirta vartotojo saugumui padidinti. Šios programos analizės rezultatai pateikiami 4.8 ir 4.9 lentelėse.

4.8 lentelė. Patobulintų taisyklių rezultatai „ASProxy“ programoje

Nr.	Taisyklė	Aptiktos klaidos		Skirtumas
		Prieš	Po	
1.	ReviewLockUsedOnlyForOperationsOnVariablesRule	0	0	0
2.	CheckParametersNullityInVisibleMethodsRule	200	238	38
3.	ConsiderAddingInterfaceRule	1	1	0
4.	AvoidUnusedPrivateFieldsRule	0	2	2
5.	RemoveUnusedLocalVariablesRule	0	0	0
6.	ArrayFieldsShouldNotBeReadOnlyRule	4	4	0
	Viso:	**205**	**245**	**40**

Vėlgi daugiausiai klaidų aptiko ReviewLockUsedOnlyForOperationsOnVariablesRule taisyklė. Taip pat dvi naujas klaidas aptiko AvoidUnusedPrivateFieldsRule taisyklė. Viso aptikta 40 naujų klaidų.

4.9 lentelė. Naujų taisyklių rezultatai „ASProxy“ programoje

Nr.	Taisyklė	Aptiktos klaidos
1.	AvoidCaseSensitiveNamesRule	4
2.	AvoidReturnSameValueRule	1
3.	AvoidUsingNullAfterNullityCheckRule	3
4.	AvoidInfiniteLoopRule	0
5.	EventHandlersShouldNotBeVisibleRule	1
6.	AvoidSQLInjectionRule	0
	Viso:	**9**

Naujos taisyklės aptiko 9 klaidas. Iš viso prieš patobulinimus aptikta 3230 klaidų, o po jų 3279 klaidos.

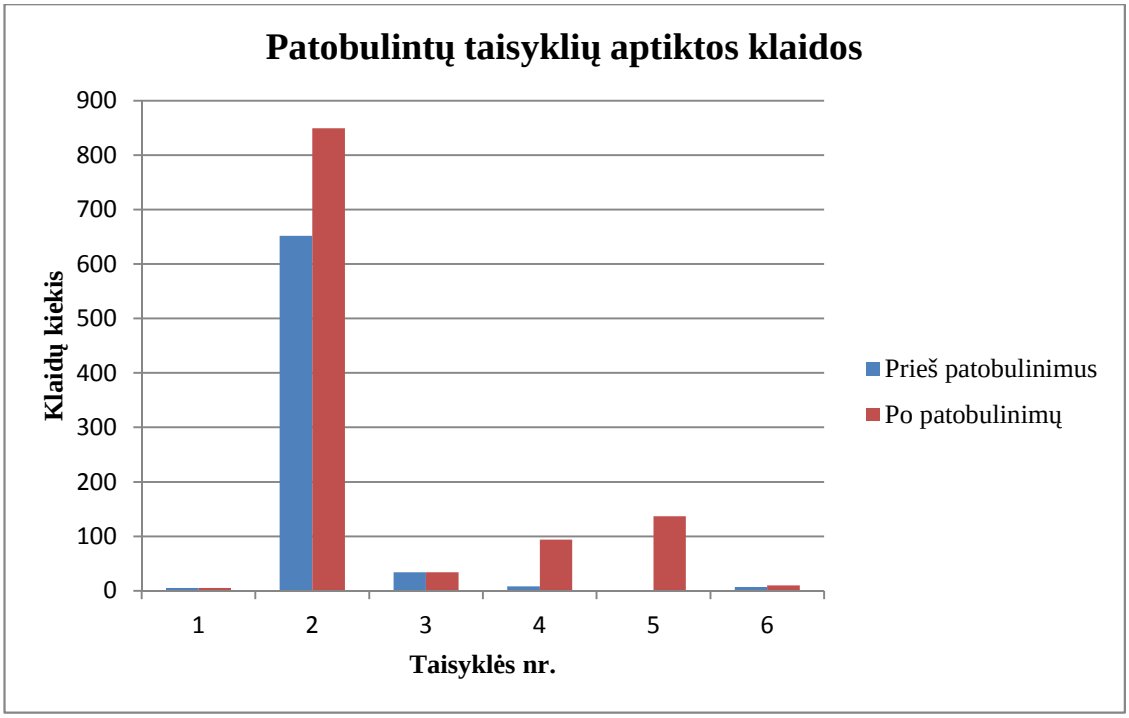
4.6. Patobulintų taisyklių aptiktos klaidos

Atvirojo kodo programų tikrinimas su patobulintomis taisyklėmis leido aptikti nemažai likusių klaidų programiniame kode. Bendri patobulintų taisyklių testavimo rezultatai, patikrinus 134900 kodo eilučių, pateikti 4.10 lentelėje.

4.10 lentelė. Patobulintų taisyklių aptiktos klaidos

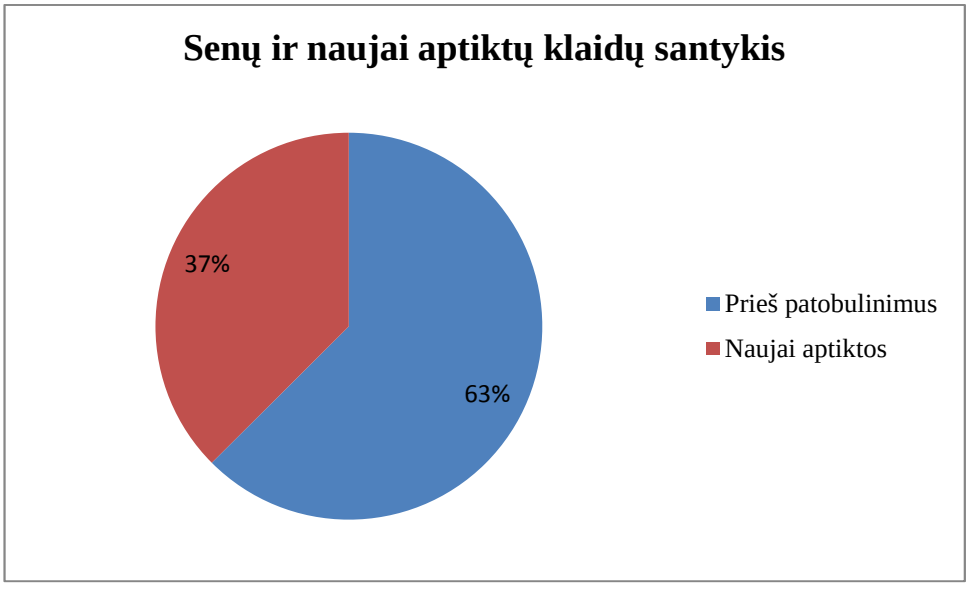
Nr.	Taisyklė	Aptiktos klaidos		Skirtumas
		Prieš	Po	
1.	ReviewLockUsedOnlyForOperationsOnVariablesRule	5	5	0
2.	CheckParametersNullityInVisibleMethodsRule	625	849(839)	197 (187)
3.	ConsiderAddingInterfaceRule	34	34	0
4.	AvoidUnusedPrivateFieldsRule	8	94 (78)	86 (70)
5.	RemoveUnusedLocalVariablesRule	0	137	137
6.	ArrayFieldsShouldNotBeReadOnlyRule	7	10	3
	Viso:	**706**	**1129**	**423 (397)**

4 iš 6 patobulintų taisyklių eksperimento metu aptiko daugiau klaidų nei prieš patobulinimus. Daugiausiai, 197 naujas klaidas, aptiko CheckParametersNullityInVisibleMethodsRule taisyklė. Ji ir prieš patobulinimus aptiko daugiausiai klaidų. Tačiau 10 naujai aptiktų klaidų tektų ignoruoti, nes viena programa turi atskirą metodą patikrinti ar tam tikro tipo kintamasis yra validus. RemoveUnusedLocalVariablesRule taisyklė prieš patobulinimus neaptikusi nei vienos klaidos, po jų aptiko 137. AvoidUnusedPrivateFieldsRule taisyklė prieš patobulinimus aptikusi 8 klaidas, po jų aptinka 94. Tačiau 16 iš jų nėra klaidos. Šią taisyklę dar reiktų tobulinti, kad ji teisingai aptiktų klasės kintamuosius metoduose, kurie naudoja *yield* raktažodį. ArrayFieldsShouldNotBeReadOnlyRule taisyklė aptiko 3 naujas klaidas.



4.1 pav. Patobulintų taisyklių aptiktos klaidos

4.1 pav. pateikiami 4.10 lentelės rezultatai grafiškai. Čia labiausiai išsiskiria CheckParametersNullityInVisibleMethodsRule taisyklė. Prieš patobulinimus ir po jų ji aptinka kur kas daugiau klaidų nei kitos tobulintos taisyklės. Galbūt taip nutinka tada, kai programuotojai kuria programą, o ne panaudojamą komponentą. Kurdami panaudojamus komponentus jie galvoja, kad vieši metodai bus naudojami kitų programuotojų ir tokių klaidų nepadaro, tačiau kurdami galutinį produktą, programuotojai dažnai net nesusimasto, kad jų sukurtus viešus metodus be jų pačių gali dar kažkas naudoti.



4.2 pav. Senų ir naujai aptiktų klaidų santykis

Patobulintos taisyklės aptiko 423 klaidomis daugiau, o tai yra 60 % daugiau nei prieš patobulinimus. Iš viso naujai aptiktos klaidos sudaro 37 % visų tobulintų taisyklių aptiktų klaidų (žr. 4.2 pav.).

Deja ne visos naujai aptiktos klaidos yra tikros klaidos. 4.10 lentelėje skliausteliuose nurodytos tikros klaidos. Patobulintos taisyklės aptiko 26 netikras klaidas. Taigi, 6 % naujai aptiktų

klaidų yra neteisingos klaidos, tačiau tai yra priimtinas rezultatas, nes teisingai aptiktos klaidos gali padėti pagerinti kodo kokybę.

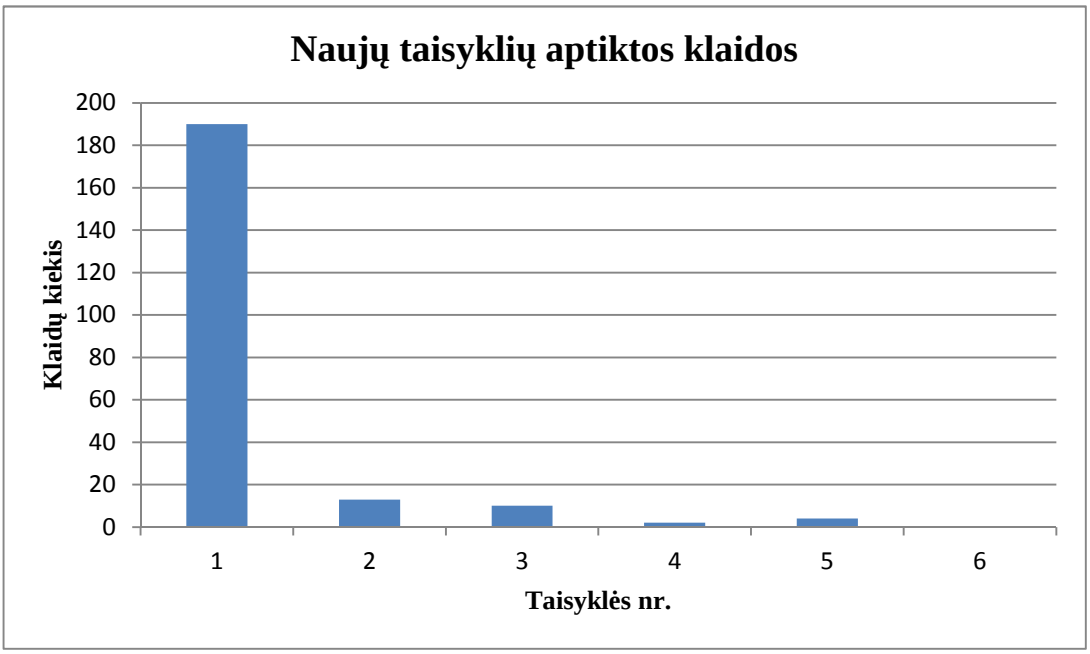
4.7. Naujų taisyklių aptiktos klaidos

„Gendarme“ įrankio taisyklių rinkinio praplėtimas padėjo aptikti naujas programinio kodo klaidas. Ištestavus 5 atviro kodo programas, buvo aptiktas nemažas kiekis likusių klaidų, kurias ištaisius pagerėtų programų veikimas ir jas pačias būtų lengviau tobulinti. Bendri naujų taisyklių panaudojimo rezultatai pateikti 4.11 lentelėje.

4.11 lentelė. Naujų taisyklių aptiktos klaidos

Nr.	Taisyklė	Aptiktos klaidos
1.	AvoidCaseSensitiveNamesRule	190
2.	AvoidReturnSameValueRule	13
3.	AvoidUsingNullAfterNullityCheckRule	10
4.	AvoidInfiniteLoopRule	2
5.	EventHandlersShouldNotBeVisibleRule	4
6.	AvoidSQLInjectionRule	0
	Viso:	**219**

5 iš 6 naujų taisyklių aptiko klaidų. Daugiausiai jų, net 190, aptiko AvoidCaseSensitiveNamesRule taisyklė. Kitos taisyklės aptiko mažiau, nuo 2 iki 13 klaidų. Vienintelė nauja AvoidSQLInjectionRule taisyklė neaptiko klaidų. Galbūt analizuotose programose SQL nebuvo naudojamas. Iš viso naujos taisyklės aptiko 219 naujų klaidų. Visos aptiktos klaidos teisingos, tačiau tektų ignoruoti AvoidInfiniteLoopRule taisyklės aptiktas klaidas, nes ten amžinas ciklas yra naudojamas specialiai.

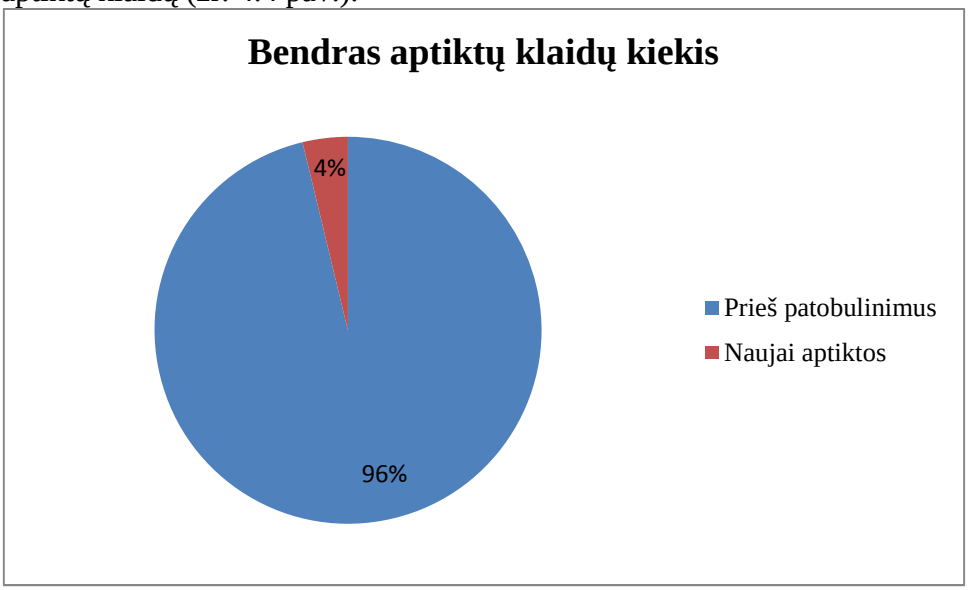


4.3 pav. Naujų taisyklių aptiktos klaidos

4.3 pav. pateikiami rezultatai grafiškai. Kaip ir minėta anksčiau, labiausiai išsiskiria AvoidCaseSensitiveNamesRule taisyklė aptikusi daug daugiau klaidų nei kitos naujos taisyklės.

4.8. Bendri rezultatai

Iš viso prieš patobulinimus buvo aptiktos 16382 klaidos, o po jų 17024. Taigi aptiktos 642 naujos klaidos, t.y. 4 % daugiau nei prieš patobulinimus. Be to, naujai aptiktos klaidos taip pat sudaro 4 % visų aptiktų klaidų (žr. 4.4 pav.).



4.4 pav. Bendras aptiktų klaidų kiekis.

4.9. Testavimo rezultatų apibendrinimas

Išanalizavus 5 programas, kurių bendras programinio kodo ilgis yra 134900 eilutės, su nepatobulintu „Gendarme“ įrankiu buvo aptiktos 16382 klaidos. Po patobulinimų įrankis aptiko 17024 klaidas. Taigi aptiktos 642 naujos klaidos, o tai sudaro 4 % visų aptiktų klaidų.

4 iš 6 patobulintų taisyklių eksperimento metu aptiko daugiau klaidų nei prieš patobulinimus. Bendrai šios taisyklės aptiko 423 naujas klaidas, o tai sudaro 37 % visų šių taisyklių aptiktų klaidų ir yra 60 % daugiau nei prieš patobulinimus. Prieš patobulinimus ir po jų daugiausiai klaidų aptiko CheckParametersNullityInVisibleMethodsRule taisyklė. Deja, ne visos patobulintų taisyklių aptiktos klaidos yra teisingos. 26 iš naujai aptiktų klaidų yra neteisingos, tačiau tai priimtinas rezultatas, nes tai sudaro tik 6 % naujai aptiktų klaidų.

Naujos taisyklės aptiko 219 klaidų. Tik viena AvoidSQLInjectionRule taisyklė eksperimento metu neaptiko nei vienos klaidos. Daugiausiai, net 190 klaidų aptiko AvoidCaseSensitiveNamesRule taisyklė. Visos naujų taisyklių aptiktos klaidos yra tikros.

5. IŠVADOS

1. Kodo analizės įrankių naudojimas - vienas iš pigiausių ir anksčiausiai saugumo spragas padedančių aptikti būdų.
2. „Gendarme“ ir „FindBugs“ įrankiai labiau ištobulinti ir aptinka daugiau įvairių klaidų nei „Cppcheck“. Tarp „Gendarme“ aptinkamų klaidų nemažai tokių, kurias aptinka ir kompiliatorius. Tuo tarpu „Cppcheck“ nebando aptikti tokių klaidų.
3. Tolesniam tobulinimui pasirinktas „Gendarme“ statinės kodo analizės įrankis, kurio kodas aiškus, taisyklės ir aptinkamos klaidos atskirtos. Esamos „Gendarme“ taisyklės yra gana išbaigtos, spragų esamose taisyklėse nebuvo rasta daug.
4. „Gendarme“ įrankiui buvo pasiūlyta keletas esamų taisyklių patobulinimų, kurie turėtų padėti aptikti daugiau klaidų ir pagerinti programų spartą, stabilumą, saugumą ir pagerinti kodo skaitomumą.
5. Pasiūlytos naujos taisyklės, kurios turėtų padėti aptikti daugiau perteklinio kodo, kurio neaptinka kompiliatorius, supaprastinti programos kodą ir pagerinti jo skaitomumą, aptikti daugiau kritinių programų klaidų.
6. Patobulintos 6 esamos taisyklės bei sukurta tiek pat naujų. Nepavyko įgyvendinti tik vienos naujos taisyklės dėl to, kad analizuojamas dvejetainis kodas buvo optimizuotas kompiliavimo metu.
7. Eksperimento metu išanalizuotos 5 atviro kodo programos, kurių bendras programinio kodo ilgis yra 134900 eilutės.
8. 4 iš 6 patobulintų taisyklių eksperimento metu aptiko daugiau klaidų nei prieš patobulinimus.
9. Patobulintos taisyklės aptiko 423 naujas klaidas, o tai sudaro 37 % visų šių taisyklių aptiktų klaidų ir yra 60 % daugiau nei prieš patobulinimus. 6 % naujų klaidų, aptiktų patobulintų taisyklių, yra klaidingai aptiktos.
10. 5 iš 6 naujų taisyklių eksperimento metų aptiko 219 klaidų. Visos klaidos aptiktos naujų taisyklių yra tikros klaidos.
11. Eksperimento metu patvirtinta, kad patobulinimai ir naujos taisyklės tikrai padėtų aptikti daugiau klaidų programose ir pagerinti jų saugumą. Tiesiogiai su saugumu susijusios taisyklės (CheckParametersNullityInVisibleMethodsRule, ArrayFieldsShouldNotBeReadOnlyRule, AvoidUsingNullAfterNullityCheckRule ir EventHandlersShouldNotBeVisibleRule) aptiko 214 naujų klaidų.
12. Darbo rezultatai pristatyti konferencijoje „Informacinė visuomenė ir universitetinės studijos“ (IVUS 2013) (žr. 7.1 priedą).

6. LITERATŪRA

[1] D. Baca, K. Petersen, B. Carlsson, L. Lundberg, „Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?“, 2009 International Conference on Availability, Reliability and Security, March 2009, pp. 804-810.

[2] M. Bradley, F. Cassez, A. Fehnker, T. Given-Wilson, and R. Huuck, „High Performance Static Analysis for Industry“, Tools for Automatic Program Analysis, vol. 289, December 2012, pp. 3-14.

[3] Dr. Paul E. Black, „Static Analyzers in Software Engineering“, CROSSTALK The Journal of Defence Software Engineering. March/April 2009.

[4] P. Anderson, „Measuring the Value of Static-Analysis Tool Deployments“, IEEE Security & Privacy, vol. 10, iss. 3, May/June 2012, pp. 40-47.

[5] A. G. Bardas, „Static Code Analysis“, Journal of Information Systems & Operation Management, vol. 4, no. 2, December 2010.

[6] B. Chess, G. McGraw, „Static Analysis for Security“, Security & Privacy, vol. 2, iss. 6, November/December 2004.

[7] U. Bayer, A. Moser, C. Kruegel, E. Kirda, „Dynamic analysis of malicious code“, Journal in Computer Virology, vol. 2, no. 1, August 2006.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B.Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler. „A Few Billion Lines of Code Later: Using Static Code Analysis to Find Bugs in the Real World“, Communications of the ACM, vol. 53, iss. 2, February 2010, pp. 66-75.

[9] „Gendarme“. Mono [interaktyvus]. [Žiūrėta 2012-01-27]. Prieiga per internetą: http://www.mono-project.com/Gendarme.

[10] „Standard ECMA-335, Common Language Infrastructure (CLI)“. Ecma International. 6th Edition, June 2012.

[11] Geeknet. „Cppcheck - A tool for static C/C++ code analysis“. Sourceforge [interaktyvus]. [Žiūrėta 2012-01-27]. Prieiga per internetą: http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page.

[12] „FindBugs“. Sourceforge [interaktyvus]. [Žiūrėta 2013-04-10]. Prieiga per internetą: http://findbugs.sourceforge.net/index.html.

[13] N. Ayewah, W. Pugh, „The Google FindBugs fixit“, ISSTA ’10, July 2010, pp. 241-252.

[14] „Microsoft .Net Framework Support“. StatOwl [interaktyvus]. [Žiūrėta 2013-02-23]. Prieiga per internetą: http://www.statowl.com/microsoft_dotnet.php.

[15] National Vulnerability Database [interaktyvus]. [Žiūrėta 2013-02-21]. Prieiga per internetą: http://web.nvd.nist.gov/view/vuln/statistics.

7. PRIEDAI

7.1. priedas. Straipsnis publikuotas „Informacinė visuomenė ir universitetinės studijos“ (IVUS 2013) konferencijoje.

Statinės kodo analizės įrankio „Gendarme“ tobulinimas

Andrius Zonys
Kompiuterių katedra
Kauno technologijos universitetas
Kaunas, Lietuva
a.zonys@gmail.com

Jonas Čeponis
Kompiuterių katedra
Kauno technologijos universitetas
Kaunas, Lietuva
jonas.ceponis@ktu.lt

Santrauka. **Šiame darbe aptariama statinė ir dinaminė kodo analizė, analizės įrankių tipai bei jų paskirtis. Nagrinėjami įrankiai, skirti .NET programų analizei. Plačiau analizuojamas laisvai platinamas statinės kodo analizės įrankis „Gendarme“. Pateikiamos spragos, rastos esamose „Gendarme“ taisyklėse bei siūlomi jų patobulinimai, kurie turėtų padėti aptikti daugiau klaidų ir padidinti programų spartą, stabilumą, saugumą ir kodo skaitomumą.**

Reikšminiai žodžiai: statinė analizė, kodo tikrinimas, kodo tikrinimo taisyklės.

I. ĮVADAS

Kompiuteris tampa neatskiriama mūsų gyvenimo dalimi, o viena svarbiausių kompiuterio dalių yra programinė įranga. Programų skaičius sparčiai auga, nuolat kuriama daug specializuotų ir nespecializuotų programų verslui ir žmonėms. Kai kurioms programoms saugumas yra nelabai svarbus, tačiau daugeliui, ypač verslui skirtoms programoms, jis yra labai svarbus. Saugumas bei kokybė yra labai brangus dalykas, ypač, jei programinis kodas yra ilgas [1]. Apie 50 % programų kūrimo kainos sudaro testavimas ir derinimas [2], todėl ne visos, ypač mažosios įmonės, gali sau tai leisti. Jos taupo saugumo sąskaita ir tikisi, jog į jų sistemas nebus įsibrauta. Tačiau yra gana nebrangių ar net nemokamų programinių priemonių, padedančių užtikrinti geresnį kodo saugumą ir kokybę. Jos atlieka statinę ir dinaminę kodo analizę. Šiai analizei yra sukurta nemažai įrankių, tačiau joks įrankis ar būdas negali užtikrinti visiško saugumo.

Šiame darbe analizuojami nemokami statinės kodo analizės įrankiai .NET programoms tikrinti, ir siūlomi pasirinkto įrankio patobulinimai, kurie padės aptikti daugiau kodo klaidų ir pagerinti kodo saugumą.

II. PROGRAMINIO KODO ANALIZĖ

Kodo analizės metodus galima suskirstyti į tris pagrindines kategorijas: tai rankinis metodas, kai kodas peržiūrimas žmogaus, statinis – kai kodas tikrinamas automatizuotai jo nevykdant, ir dinaminis – kai kodas tikrinamas veikiant programai.

A. Statinė kodo analizė

Tai – programinės įrangos kodo peržiūra jo nevykdant. Peržiūros metu nagrinėjamas programos kodas, ir pagal žinomus klaidų šablonus bandoma aptikti klaidingas jo vietas.

Statinės kodo analizės įrankiai turi aptikti programinio kodo klaidas ir pranešti apie jų vietas arba turėti priemonių, leidžiančių ignoruoti aptiktas klaidas. Dauguma klaidų yra suskirstytos į grupes. Įrankiai gali ne tik pranešti apie klaidą, bet ir pasiūlyti, kaip ją ištaisyti. Kai kurie iš jų gali pateikti išsamias ataskaitas apie aptiktas klaidas [3].

Statinė kodo analizė gali būti atlikta gana anksti programavimo procese, kadangi ji atliekama nevykdant kodo. Programuotojai, naudodami statinę kodo analizę, iš karto po to, kai parašomas kodas, gali aptikti ir ištaisyti klaidas dar prieš modulių (angl. *unit*) ir integracijos testavimą. Kuo anksčiau aptinkama klaida, tuo pigiau kainuoja ją pataisyti. Tai – pagrindinis statinės kodo analizės privalumas [4].

Statinės kodo analizės įrankiai būna skirtingų tipų. Vieni analizuoja tiesioginį kodą, kiti – sukompiliuotą baitų kodą. Ir vieni, ir kiti turi savų pliusų. Kai analizuojamas tiesioginis programuotojo parašytas kodas, jis prieš analizę neiškraipomas. O – kompiliatoriaus sukompiliuotas ir optimizuotas dvejetainis kodas gali skirtis nuo originalaus. Tačiau dvejetainio kodo analizė yra kur kas spartesnė, o sparta labai svarbi tiriant daug kodo eilučių [5].

Nors įrankių tipai ir skiriasi, jie turi tuos pačius pagrindinius bruožus: skaito kodą ir sukuria jo abstraktų modelį, pagal klaidų sąrašus ir jų šablonus ieško atitinkančių vietų. Taip pat įrankiai analizuoja duomenų srautus, bando numatyti reikšmes, kurias kintamasis gali įgyti tam tikrose kodo vietose.

Dauguma produktų gali atlikti neišbaigto kodo analizę (nepritaikyti visi metodai ar nesukurtos visos klasės). Kuo kodas išbaigtesnis, tuo geriau įrankis gali aptikti klaidas [6].

Įrankiai gali aptikti retai pasitaikančias klaidas ar slaptas duris (angl. *hidden back doors*). Jie analizuoja kodą jo nevykdant, dėl to gali išskaičiuoti kur kas daugiau ryšių tarp skirtingų modulių ir komponentų.

Testuoti paprasta, tačiau parašyti testą, kuris iki galo patikrintų kokį nors modulį, nėra lengva. Kai aptinkama naujų spragų, turi būti parašyti nauji testai joms aptikti. Jei aptinkamų klaidų sąrašas bus neatnaujinamas, įrankis neaptiks naujausių saugumo spragų.

B. Dinaminė kodo analizė

Priešingai nei statinė kodo analizė, dinaminės kodo analizės metu tiriamas kodas jį vykdant. Nors ši technika nepateikia išsamių duomenų, tačiau – turi didelį privalumą – vienintelė analizuoja kodą jį vykdant. Tokiu būdu dinaminė kodo analizė padeda aptikti bandymus pakenkti programai ir aptikti besimodifikuojančias programas. Taip pat šio tipo įrankiai gali išnagrinėti programą įvairiose aplinkose [7]. Kad dinaminė kodo analizė būtų efektyvi, reikia ganėtinai daug testų.

Dinaminės kodo analizės įrankius galima suskirstyti į tris pagrindinius tipus: tai analizuojantys atmintį, kodo padengimą ar veikimo spartą. Atminties klaidas aptikti gana sunku. Šio tipo įrankius geriausia naudoti, kai problemą ne visada įmanoma atkartoti ar atminties naudojimas per daug ar per greit išauga. Atminties spragas dažnai naudoja programišiai perimdami programų kontrolę. Įrankiai apskaičiuoja, kokia kodo dalis ar kokios funkcijos buvo patikrintos. Pasitelkiamas baltos dėžės testavimas. Naudojant kodo padengimą, lengviau aptikti nepatikrintas kodo vietas ir užtikrinti geresnę testavimo kokybę. Šio tipo įrankiai skaičiuoja, kiek kartų buvo iškviesta viena ar kita funkcija bei kiek laiko funkcija buvo vykdoma. Naudojant šio tipo įrankius lengviau aptikti silpnąsias vietas ir pagerinti programos spartą.

Dinaminės kodo analizės privalumas yra tas, kad kodas analizuojamas programai veikiant ir dėl to galima aptikti daugiau įvairių klaidų. Taip pat kai kurie dinaminės kodo analizės įrankiai leidžia pasirinkti ką analizuoti, o ką praleisti. Šio tipo analizę galima atlikti įvairioms programoms. Tačiau pagrindinė šio tipo analizės problema yra ta, kad sunku nusakyti tikslią pažeidžiamą kodo vietą. Dėl to tai reikalauja daugiau programuotojo pastangų ir ilgiau trunka ištaisyti rastą klaidą.

III. STATINĖS KODO ANALIZĖS ĮRANKIAI

Statinei .NET programų kodo analizei yra sukurta įvairių įrankių – tiek mokamų, tiek nemokamų. Šiame darbe nagrinėjame nemokamus statinės kodo analizės įrankius „FxCop“, „Smokey“ ir „Gandarme“.

„FxCop“ [8] – tai programa, kuri analizuoja kodą, parašytą .NET CLR formatu, ir ataskaitose pateikia informaciją apie aptiktas projektavimo, lokalizacijos, našumo ir saugumo problemas. Dauguma problemų susijusios su programavimo ir projektavimo taisyklių pažeidimais, išdėstytais „Microsoft“ projektavimo rekomendacijose. Jos skirtos padėti rašyti patikimą ir lengvai palaikomą kodą naudojant „.NET Framework“.

„FxCop“ yra suprojektuota visiškam integravimui į programinės įrangos kūrimo ciklą. Ji platinama ir kaip programa, kuri turi grafinę vartotojo sąsają interaktyviam darbui, ir kaip komandinės eilutės sąsaja grindžiamas įrankis, kuris gali būti integruotas į kompiliavimo procesą ar integruotas į „Microsoft Visual Studio“ programavimo aplinką.

„Smokey“ [9] yra atvirojo kodo komandinės eilutės sąsaja grindžiamas įrankis, skirtas .NET ir „Mono“ programų analizei. Tai panašus įrankis į „FxCop“ ir „Gendarme“. „Smokey v1.4“ versija palaiko .NET 3.5 ir „Mono 2.0“ versijas ir turi 233 taisykles. Jis gali sugeneruoti aptiktų klaidų ataskaitas HTML, XML ar teksto formatais.

„Gendarme“ [10] yra plečiamas, taisyklėmis grindžiamas įrankis, skirtas aptikti klaidoms .NET programose bei bibliotekose. Įrankis analizuoja programas ir bibliotekas, parašytas ECMA CIL [11] formatu, ir ieško dažniausiai pasitaikančių programavimo klaidų bei tokių, kurių kompiliatorius įprastai neaptinka ar nebando aptikti. „Gendarme“ kodui analizuoti naudoja biblioteką „Cecil“. Ši biblioteka, kaip ir pats įrankis, parašyta C# programavimo kalba.

„Gendarme“ taisyklės, kurios skirtos kodo analizei, yra naudojamos per paleidėjus (angl. *runners*). Tai – programos dalis atsakinga už taisyklių paleidimą, analizę ir klaidų pateikimą. „Gendarme“ šiuo metu turi du skirtingus paleidėjus. Vienas grindžiamas komandinės eilutės sąsaja, kitas – SWF grafine sąsaja. Komandinės eilutės paleidėjas gali išvesti rezultatus į komandinę eilutę, XML failus ar tinkamai suformuotus HTML failus. SWF grindžiamas paleidėjas leidžia pažingsniui pasirinkti modulius (angl. *assembly*), taisykles, nustatymus ir pavaizduoti analizės rezultatus kaip ir komandinės eilutės paleidėjas. Kad būtų paprastesnis, jis turi mažiau nustatymų, nei komandinės eilutės paleidėjas.

Toliau tobulinti buvo pasirinkta įrankį „Gendarme“ įrankis, kadangi jis yra atvirojo kodo, parašytas C# programavimo kalba, turi aiškias taisykles, kurios svarbios testavimo metu. Nesuprasta taisyklė dažnai reiškia ignoruojamą klaidą arba dar blogiau – rasta klaida traktuojama kaip netikro pavojaus signalas [12]. „FxCop“ nėra atvirojo kodo, o tai sumažina tobulinimo galimybės. „Smokey“ yra atvirojo kodo, tačiau turi tik komandinės eilutės sąsają ir paskutinį kartą buvo atnaujinta tik 2008 m., dėl to nepalaiko naujausių .NET ir „Mono“ versijų.

IV. ĮRANKIO „GENDARME“ TAISYKLIŲ PATOBULINIMAS

Šioje dalyje aprašomi siūlomi esamų įrankio „Gendarme“ taisyklių patobulinimai. Šie patobulinimai turėtų padėti aptikti daugiau klaidų ir padidinti programų spartą, stabilumą, saugumą ir kodo skaitomumą. Prie kiekvienos taisyklės pateikiami netinkamo ir gero programinio kodo pavyzdžiai. Netinkamas kodas – tai kodas, kuris turi saugumo spragą ar tiesiog nėra optimalus variantas ir kurį reikia taisyti. Gero programinio kodo pavyzdys parodo, kaip vieną ar kitą problemą siūloma išspręsti.

A. ReviewLockUsedOnlyForOperationsOnVariablesRule

Ši taisyklė priklauso „Gendarme.Rules.Concurrency“ kategorijai ir tikrina, ar `lock` operatorius yra naudojamas tik operacijoms su klasės ar lokaliais kintamaisiais atlikti. Jei vienintelis tikslas kritinėje dalyje yra užtikrinti, kad kintamasis bus pakeistas automatiškai, tuomet geriau naudoti `System.Threading.Interlocked`.

Netinkamo programinio kodo pavyzdys, kuriame programa neaptikdavo klaidos:

```
lock (m_lockObject)
{
    count++;
    m_someSharedObject = anotherObject;
```

```
    Interlocked.Add (pages, 5);
}
```

Gero programinio kodo pavyzdys:

```
Interlocked.Increment (count);
Interlocked.Exchange (m_someSharedObject,
anotherObject);
Interlocked.Add (pages, 5);
```

Taisyklė tikrina visas `lock` viduje esančias operacijas, kol aptinka tokią, kuri nėra automatinė operacija su kintamuoju. Jei tokia operacija neaptinkama visoje zonoje, tuomet registruojama klaida. Patobulinta taisyklė dabar atpažįsta visus `System.Threading.Interlocked` klasės metodus kaip automatinius metodus su kintamaisiais. Taigi algoritmas nebenutraukiamas aptikus `Iterlocked` klasės metodą.

B. CheckParametersNullityInVisibleMethodsRule

Taisyklė priklauso „Gendarme.Rules.Correctness“ kategorijai. Ji sako, jog matomų metodų parametrai turėtų būti patikrinti, ar ne `null` prieš juos naudojant.

Netinkamo programinio kodo pavyzdys, kai programa neaptikdavo, jog nėra `null` patikrinimo:

```
public void CheckParametersNullity (string
name)
{
    string nameCopy = string.Copy (name);
    if (name.Length > 10)
        Console.WriteLine ("Name: " + name);
}
```

Gero programinio kodo pavyzdys:

```
public void CheckParametersNullity (string
name)
{
    string nameCopy = string.Copy (name);
    if (name != null && name.Length > 10)
        Console.WriteLine ("Name: " + name);
}
```

Tačiau ši taisyklė nebeaptikdavo klaidos, jei prieš tai būdavo taikomas bet koks metodas ir jam kaip parametras buvo perduodamas šio metodo parametras. Patobulinta taisyklė šio veiksmo nebetraktuoja kaip `null` reikšmės patikrinimo. Tačiau išlieka dar viena problema: algoritmas tik aptinka, ar yra patikrinimas `param != null` ar `param == null`, tačiau netikrina, kas vyksta toliau. O po to gali eiti toks veiksmas: `if (param == null) param.Update ()`, kurį vykdydama programa savo darbą baigs nenumatytu būdu.

C. ConsiderAddingInterfaceRule

Taisyklė priklauso „Gendarme.Rules.Design“ kategorijai. Ji praneša apie klaidą, jei klasė įgyvendina visus sąsajos narius, tačiau nepaveldi pačios sąsajos. Patobulinta taisyklė nebeatsižvelgia į didžiąsias ir mažąsias raides. Ieškant metodo su atitinkamu pavadinimu dabar naudojamas `StringComparison.Ordinal` palyginimo tipas. Taip pat patobulinta taisyklė palyginimuose dabar taiko ir statinius metodus.

Netinkamo programinio kodo pavyzdys, kai programa nesiūlydavo paveldėti sąsajos:

```
public interface ITest
{
    void CreateDGNFile ();
    bool IsECInstance ();
}
public class Test
{
    static void CreateDgnFile () {}
    bool IsEcInstance () { return true; }
}
```

D. AvoidUnusedPrivateFieldsRule

Priklauso „Gendarme.Rules.Performance“ kategorijai. Taisyklė skirta nenaudojamiems privatiesiems klasės kintamiesiems aptikti. Patobulinta taisyklė neaptinka klaidos tik tuomet, kai privataus kintamojo reikšmė yra nuskaitoma kurioje nors klasės operacijoje. Taip pat patobulinta taisyklė aptinka ir nenaudojamas privačias konstantas.

Netinkamo programinio kodo pavyzdys, kai programa neaptikdavo klaidos:

```
public class Test
{
    private string m_unused;
    private const string m_unusedConst;
    public void AvoidUnusedPrivateFieldsRule
()
    {
        m_unused = "Unused private variable";
        m_unusedConst = "Unused private
const";
        DoSomething ();
    }
}
```

Gero programinio kodo pavyzdys:

```
public class Test
{
    public void AvoidUnusedPrivateFieldsRule
()
    {
        DoSomething ();
    }
}
```

Patobulinta taisyklė tikrina visas klasės operacijas tol, kol aptinka tokią, kurioje yra nuskaitoma kintamojo reikšmė. Anksčiau buvo ieškoma operacijos, kur naudojamas kintamasis, neįvertinant, ar jam reikšmė priskiriama, ar nuskaitoma.

E. RemoveUnusedLocalVariablesRule

Taisyklė priklauso „Gendarme.Rules.Performance“ kategorijai. Taisyklė skirta nenaudojamiems metodo kintamiesiems aptikti. Patobulinta taisyklė neaptinka klaidos tik tuomet, kai metodo kintamojo reikšmė yra nuskaitoma kurioje nors metodo operacijoje.

Netinkamo programinio kodo pavyzdys, kuriame programa neaptikdavo klaidos:

```
public void AvoidUnusedLocalVariablesRule ()
{
```

```
    String unused = "Unused local variable.";
    DoSomething ();
}
```

Gero programinio kodo pavyzdys:

```
public void AvoidUnusedLocalVariablesRule ()
{
    DoSomething ();
}
```

Taisyklė tikrina visas klasės operacijas tol, kol aptinka tokią, kurioje yra nuskaitoma kintamojo reikšmė. Anksčiau buvo ieškoma operacijos, kur naudojamas kintamasis, neįvertinant, ar jam reikšmė priskiriama, ar nuskaitoma.

*F. ArrayFieldsShouldNotBeReadOnly*Rule

Taisyklė priklauso „Gendarme.Rules.Security“ kategorijai. Ši taisyklė aptinka masyvus, kurie yra deklaruoti kaip `public readonly` ir praneša apie klaidą, nes tokiu atveju negalima pakeisti paties masyvo, bet galima pakeisti jo reikšmes. Patobulinta taisyklė aptinka ir deklaruotus sąrašus bei rinkinius (angl. *collection*), nes jiems galioja ta pati saugumo spraga.

Pavyzdys, kuriame programa neaptikdavo klaidos:

```
public class Test
{
     public readonly
System.Collections.ObjectModel.Collection<int>
readOnlyCollection = new Collection<int> ();
     public readonly
System.Collections.Generic.ICollection<int>
readOnlyICollection = new List<int> ();
 }
```

Gero programinio kodo pavyzdys:

```
public class Test
{
     private readonly
System.Collections.ObjectModel.Collection<int>
ROCollection = new Collection<int> ();
     private readonly
System.Collections.Generic.ICollection<int>
ROICollection = new List<int> ();
    public int[] GetCollectionAsArray ()    {
    int[] array = new int[ROCollection.Count];
    ROCollection.CopyTo (array, 0);
    return array;
    }
    public int[] GetICollectionAsArray ()    {
    int[] array =new int[ROICollection.Count];
    ROICollection.CopyTo (array, 0);
    return array;
    }
}
```

Taisyklė tikrina visų klasių kintamuosius. Pirmiausiai patikrinama, ar kintamąjį galima tik skaityti ir ar jis yra viešas. Po to tikrinamas tipas ir, jei jis yra masyvas, sąrašas ar rinkinys, spausdinamas klaidos pranešimas.

V. IŠVADOS

Nemokamų statinės kodo analizės įrankių naudojimas – vienas pigiausių ir anksčiausiai saugumo spragas padedančių aptikti būdų. Kodo analizės įrankis „Gendarme“ skirtas aptikti klaidoms .NET programose. .NET platformos naudojimas vis labiau auga [13], taigi įrankis „Gendarme“ tampa vis aktualesnis. Programoje „Gendarme“ taisyklės ir aptinkamos klaidos aiškiai atskirtos, įrankis patogiai konfigūruojamas, o analizės rezultatai pateikiami aiškiai.

Esamos įrankio „Gendarme“ taisyklės yra gana išbaigtos, tačiau vis tiek buvo rasta likusių spragų, kurias verta ištaisyti. Įrankiui buvo patobulintos 6 esamos taisyklės. Patobulinimai turėtų padėti aptikti daugiau klaidų ir padidinti programų spartą, stabilumą, saugumą ir kodo skaitomumą.

Tolimesniuose darbuose numatoma pasiūlyti naujų taisyklių. Šios naujos taisyklės turėtų padėti aptikti daugiau perteklinio kodo, kurio neaptinka kompiliatorius, supaprastinti programos kodą ir pagerinti jo skaitomumą, aptikti daugiau kritinių programų klaidų.

LITERATŪRA

[1] D. Baca, K. Petersen, B. Carlsson, L. Lundberg, „Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?“, 2009 International Conference on Availability, Reliability and Security, March 2009, pp. 804-810.

[2] M. Bradley, F. Cassez, A. Fehnker, T. Given-Wilson, and R. Huuck, „High Performance Static Analysis for Industry“, Tools for Automatic Program Analysis, vol. 289, December 2012, pp. 3-14.

[3] Dr. Paul E. Black, „Static Analyzers in Software Engineering“, CROSSTALK The Journal of Defence Software Engineering. March/April 2009.

[4] P. Anderson, „Measuring the Value of Static-Analysis Tool Deployments“, IEEE Security & Privacy, vol. 10, iss. 3, May/June 2012, pp. 40-47.

[5] A. G. Bardas, „Static Code Analysis“, Journal of Information Systems & Operation Management, vol. 4, no. 2, December 2010.

[6] B. Chess, G. McGraw, „Static Analysis for Security“, Security & Privacy, vol. 2, iss. 6, November/December 2004.

[7] U. Bayer, A. Moser, C. Kruegel, E. Kirda, „Dynamic analysis of malicious code“, Journal in Computer Virology, vol. 2, no. 1, August 2006.

[8] „FxCop“. Microsoft [interaktyvus]. [Žiūrėta 2013-04-06]. Prieiga per internetą: http://msdn.microsoft.com/en-us/library/bb429476%28v=vs.80%29.aspx.

[9] „Smokey“. Google Project Hosting [interaktyvus]. [Žiūrėta 2013-04-06]. Prieiga per internetą: http://code.google.com/p/smokey/.

[10] „Gendarme“. Mono [interaktyvus]. [Žiūrėta 2012-01-27]. Prieiga per internetą: http://www.mono-project.com/Gendarme.

[11] „Standard ECMA-335, Common Language Infrastructure (CLI)“. Ecma International. 6th Edition, June 2012.

[12] Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler. „A Few Billion Lines of Code Later: Using Static Code Analysis to Find Bugs in the Real World“, Communications of the ACM, vol. 53, iss. 2, February 2010, pp. 66-75.

[13] „Microsoft .Net Framework Support“. StatOwl [interaktyvus]. [Žiūrėta 2013-02-23]. Prieiga per internetą: http://www.statowl.com/microsoft_dotn