KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Eimantas Puškorius

PROGRAMINĖS ĮRANGOS KODO APSAUGA MASKAVIMO METODU

BAIGIAMASIS MAGISTRO DARBAS

Vadovas
Doc. dr. Jonas Čeponis

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

KOMPIUTERIŲ KATEDRA

PROGRAMINĖS ĮRANGOS KODO APSAUGA MASKAVIMO METODU

Baigiamasis magistro darbas

Informacijos ir informacinių technologijų sauga (kodas 621E10003)

Vadovas
(parašas) Doc. dr. Jonas Čeponis
(data)

Recenzentas
(parašas) Doc. dr. Dominykas Barisas
(data)

Projektą atliko
(parašas) Eimantas Puškorius
(data)

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS

(Fakultetas)

(Studento vardas, pavardė)

(Studijų programos pavadinimas, kodas)

„Baigiamojo projekto pavadinimas“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

20 ___ m. _______________ ____ d.

Kaunas

Patvirtinu, kad mano **Eimanto Puškoriaus** baigiamasis projektas tema „Programinės įrangos kodo apsauga maskavimo metodu“ yra parašytas visiškai savarankiškai, o visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

(vardą ir pavardę įrašyti ranka) *(parašas)*

Puškorius, E. „Programinės įrangos kodo apsauga maskavimo metodu“. Magistro baigiamasis projektas / vadovas doc. dr. Jonas Čeponis; Kauno technologijos universitetas, informatikos fakultetas, kompiuterių katedra.

Raktiniai žodžiai: išeities kodas, kodo maskavimas, programinės įrangos apsauga.

Kaunas, 2016. 56 p.

SANTRAUKA

Šiame darbe yra apžvelgiami pažeidimų tipai kurie dažnai palieka spragas įsilaužimams bei metodai, kuriais yra įsilaužiama į programinę įrangą ir kokia žala padaroma įsilaužus. Darbe taip pat apžvelgiami ir palyginami apsaugos metodai ir paaiškinama nuo kokių atakų apsisaugoti jie skirti bei kokius privalumus ir trūkumus jie turi.

Darbe giliau išanalizuotas buvo maskavimo metodas ir jam skirti įrankiai. Vienas iš atvirojo kodo įrankių patobulintas taip, kad turėtų kelių maskavimo įrankių funkcionalumą vienoje vietoje. Įrankis gali supainioti kodą taip, kad įsilaužėliui ir statinės analizės įrankiams jį analizuoti taptų sunku. Tokiu būdu sustiprinamas programinės įrangos saugumas. Programinė įranga naudoja šio darbo metu sukurtus algoritmus valdymo struktūrų painiojimui bei pradinius atvirojo kodo programinės įrangos metodus kurie pakeičia kintamųjų vardus, sukurtus įrankio kūrėjo.

Darbe atlikti testai parodo kompleksinio kodo greitaveikos skirtumus tarp originalaus ir sumaišyto kodų. Testų metu taip pat patikrintas ir kodo veikimas – ar kodas veikia taip pat kaip prieš jį sumaišant.

Puškorius, Eimantas. *Software Source Code Protection in Obfuscation Method* : *Master's* thesis in Informatics / supervisor assoc. prof. Jonas Čeponis. The Faculty of Informatics , Kaunas University of Technology.

Research area and field: information technology security.

Key words: source code, obfuscation, software security.

Kaunas, 2016. 56p.

SUMMARY

This paper discusses main types of breaches which create backdoors for hacking also, methods which is being used to hack software, and damage that can be done after software gets hacked. Paper also, discusses security methods and explains what kind of attacks they can help to prevent, and what pros and cons they have.

In this thesis obfuscation method and software for this method were analyzed more deeply. One of open source software was improved in such way that it is able to do few tools functionality in one tool. Software can obfuscate code in more advanced way, so code is much harder to analyze for a hacker and static analysis tools. This software uses flow structures obfuscation algorithms, that were created by doing this thesis and original algorithms that were created by Maurice Fonk – this tool creator.

Tests that were done for this tool compares original and obfuscated code throughput and functionality – does code do the same functionality without errors after obfuscation.

TURINYS

Lentelių sąrašas ... 7
Paveikslų sąrašas ... 8
Terminų ir santrumpų žodynas ... 9
Įvadas ... 10
1. Programinės įrangos pažeidžiamumai ir apsaugos ... 11
1.1. Pažeidimų priežastys ... 11
1.2. Dažniausi atakų tipai ... 12
1.3. Esami kodo apsaugos metodai ... 13
1.4. Kodo maskavimas ir įrankiai ... 18
1.5. Įsilaužimų ir apsaugų apibendrinimas ... 24
1.6. Statistika ... 25
1.7. Analizės išvados ... 27
2. Siūlomi „Naneu PHP obfuscator“ įrankio pakeitimai (projektavimas) ... 28
2.1. Valdymo struktūra ... 28
2.2. Valdymo struktūrų aptikimas bei pakeitimas ... 29
2.3. Kintamųjų ir funkcijų maskavimas ... 30
2.4. Valdymo struktūrų maskavimas ... 31
2.5. Projektavimo išvados ... 33
3. .Atlikti „Naneu PHP obfuscator“ įrankio pakeitimai ... 34
3.1. Valdymo struktūrų maskavimo algoritmai ... 34
3.2. Klasių diagrama ... 42
3.3. Realizacijos išvados ... 43
4. Greitaveikos ir kodo veikimo tyrimas ... 44
4.1. Kodo veikimo testavimas ... 44
4.2. Kodo greitaveikos tyrimas ... 49
4.3. Tyrimo išvados ... 50
5. Išvados ... 52
6. Literatūra ... 54
7. Priedai ... 56
7.1. Kodas su kuriuo atlikti tyrimai ... 56

LENTELIŲ SĄRAŠAS

1.1 lentelė. Maskavimo trūkumai ir sprendimai 20
1.2. lentelė. Įsilaužimai ir apsaugos 24
3.1. lentelė. „if“ ir „switch“ valdymo struktūrų palyginimas.. 34
3.2. lentelė. Originali „if“ valdymo struktūra 34
3.3. lentelė. Užmaskuota „if“ valdymo struktūra.. 35
3.2. lentelė. „if“ ir „elseif“ valdymo struktūros sąlygos 38
4.1. lentelė. Pirmas valdymo struktūrų maskavimo testavimas su asmeniniu projektu....................... 44
4.2. lentelė. Antras valdymo struktūrų maskavimo testavimas su asmeniniu projektu. 45
4.3. lentelė. Trečias valdymo struktūrų maskavimo testavimas su asmeniniu projektu. 46
4.4. lentelė. Kodo veikimo testas su failu. 47
4.5. lentelė. Kodo veikimo testas su maskavimo įrankiu.. 48
4.6. lentelė. Greitaveikos tyrimo rezultatai. 50

PAVEIKSLŲ SĄRAŠAS

1.1. pav. Derinimo aptikimas pertraukiant INT1 ir INT3 instrukcijas [7] 15
1.2. pav. Derinimo aptikimas laiko tikrinimo tarp instrukcijų metodu. [8] 16
1.3. pav. ProGuard maskavimo nustatymai 22
1.4. pav. Safenet maskavimo nustatymai 23
1.5. pav. Naneu PHP obfuscator komandinė eilutė 24
1.6. pav. Įsilaužėlių naudojamų kalbų statistika [9] 25
1.7. pav. Atakų paskirstymo statistika [10] 26
1.8. pav. Šalys kuriose daugiausiai naudojama nelegali PĮ [13] 26
2.1. pav. Kintamųjų maskavimo algoritmo veiklos diagrama 30
2.2. pav. Valdymo struktūrų maskavimo algoritmo veiklos diagrama 31
2.3. pav. Valdymo struktūrų ėjimas į gylį 33
3.1. pav. Bendra „if“ valdymo struktūros maskavimo veiklos diagrama 37
3.2. pav. Detali „if“ valdymo struktūros maskavimo algoritmo veiklos diagrama 38
3.3. pav. Detali „elseif“ valdymo struktūros maskavimo algoritmo veiklos diagrama 39
3.4. pav. Detali „else“ valdymo struktūros maskavimo algoritmo veiklos diagrama 40
3.5. pav. Detali „switch“ valdymo struktūros maskavimo algoritmo veiklos diagrama 41
3.6. pav. Valdymo struktūrų maskavimo įrankio klasių diagrama 42

TERMINŲ IR SANTRUMPŲ ŽODYNAS

PĮ – programinė įranga

http – užklausimo – atsakymo protokolas

Maskavimas, maišymas (ang. *Obfuscation*) – originalaus kodo slėpimas, supainiojimas.

v.i.labs – platforma kurios pagalba programinės įrangos kūrėjai gali sekti kaip naudojama jų programinė įranga, kada nelegaliai, kada legaliai ir t.t.

Kanarėlės reikšmė (angl. *Canary value*) – reikšmė kuri sunaikinama kai įvyksta buferio perpildymas ir tuomet nutraukiamas procesas.

Rummage – perkratymo algoritmas naudojamas maskavime kai klasių ir funkcijų vardai pakeičiami į logiškus, bet sunkiai suprantamus.

Ant task (liet. Skurzdėlės užduotis) – http serverio procesas kurio metu galima nustatyti vykdymui įvairias užduotis, nuo paprasto archyvavimo iki sudėtingų testavimo užduočių.

Vidinis serveris (angl. *backend server*) – serveris kuriame apdorojami serverio pusės veiksmai ir gražinami į išorinį serverį.

Išorinis serveris (angl. *Frontend server*) – serveris kuriame apdorojamos vartotojo užklausos ir siunčiami duomenys į vidinį serverį.

Užpakalinės durys (angl. *Backdoor*) – palikta spraga programinėje įrangoje.

Juodosios dėžės principas (angl. *Black box*) – veikimo principas kai klientas nieko nežino apie paslaugos teikėjo vykdomus algoritmus ir neturi galimybės prieiti prie kodo.

PHP, Java, C++, C# – serverio pusės programavimo kalbos.

JavaScript – kliento pusės kodavimo kalba

ĮVADAS

Atliekamas darbas priklauso informacijos ir informacinių technologijų saugos specializacijai.

Darbo problematika ir aktualumas

Šiuo laikotarpiu vyrauja dažnas PĮ piratavimas, dėl kurio kūrėjai praranda didelius pinigus, vartotojų kompiuteriai yra užkrečiami ir sužinomos PĮ kodo paslaptys. Nuo skirtingų įsilaužimo būdų naudojami skirtingi apsaugos būdai: kuo jie kompleksiškesni, tuo sunkiau prieinamas PĮ kodas.

Darbo tikslas ir uždaviniai

Dėl minėtos problemos yra didelis poreikis apsaugoti programinę įrangą ir jos kodą nuo įsilaužimų. Prieš naudojant/kuriant apsaugas reikia būti gerai išanalizavus grėsmes ir būtent kokiais metodais nuo jų saugotis. Darbo tikslas yra išanalizuoti grėsmes ir apsaugos metodus, bei praktiškai patobulinti vieną iš jų. Kadangi vienas iš plačiausiai naudojamų ir atviras interpretacijoms yra kodo maskavimo metodas, nuspręsta pasirinkti būtent jį.

Uždaviniai:

1. Susipažinti su pagrindiniais apsaugos nuo atvirkštinės inžinerijos metodais;
2. Remiantis analizės etapo rezultatais pasiūlyti apsaugos nuo atvirkštinės inžinerijos patobulinimus;
3. Numatyti priemones ir algoritmus, reikalingus geresniam apsaugos metodui;
4. Realizuoti pasiūlytus patobulinimus pasirinktoje aplinkoje;
5. Įvertinti gautus rezultatus.

Darbo rezultatai ir jų svarba

Darbo rezultatai bus išanalizuotos grėsmės bei apsaugos nuo jų ir praktiškai tobulinamas vienas iš maskavimo metodų. Jei algoritmo tobulinimas pavyks bus gautas unikalus maskavimo algoritmas. Unikalūs maskavimo algoritmai naudojami kur kas saugesniam maskavimui, nes paslėptą kodą įsilaužėliams su žinomais įrankiais nesunku sugrąžinti į originalų pavidalą ir perskaityti.

Darbo struktūra

Dokumentą sudaro įvadas, analizė, aprašyti įrankiui siūlomi pakeitimai, aprašyti įrankiui atlikti pakeitimai, tyrimas, išvados ir literatūra. Įvade supažindinama su problema, tikslais, uždaviniais bei būsimais rezultatais. Analizės skyriuje analizuojamos atakų prieš PĮ rūšys, galima žala bei apsaugos priemonės, maskavimo metodai. Analizės skyriuje taip pat pateikiami su šia problema susiję kitų autorių atliktų tyrimų duomenys. Sekančiuose 2 skyriuose pateikiami siūlomi ir atlikti įrankio pakeitimai. Dokumento gale pateikiamos viso darbo išvados ir literatūra.

1. PROGRAMINĖS ĮRANGOS PAŽEIDŽIAMUMAI IR APSAUGOS

Šios analizės tikslas yra:

- Apžvelgti atakų prieš PĮ tipus ir padaromą žalą;
- Išanalizuoti apsaugos metodus nuo atakų prieš PĮ;
- Pateikti kitų autorių turimus duomenis apie PĮ atakas ir apsaugos metodus bei statistikas.

1.1. Pažeidimų priežastys

Dažniausios priežastys, kodėl paliekamos spragos programinėje įrangoje ją kuriant, yra [1]:

Kenkėjiški kūrėjai ir testuotojai

Kūrimo arba testavimo metu gali būti specialiai paliekamos spragos produkte. Vėliau informacija apie spragas perduodama įsilaužėliams, kurie nesunkiai prieina prie programinės įrangos kodo arba išnaudoja spragas. Deja, šios priežasties išvengti sunku, programuotojai ir testuotojai turi būti labai kontroliuojami arba kitaip skatinami saugoti kuriamą produktą, bei neatskleisti jo silpnųjų vietų.

Programinės įrangos kūrėjų ar testuotojų atlaidumas arba nemokšiškumas

Jei kūrėjų ar testuotojų žinios neatitinka produktui kurti ar testuoti reikalingų žinių lygio, atsiranda didelė tikimybė, kad bus palikta spraga kurios jie nepastebės, tačiau ją pastebės įsilaužėliai. Kuriant sudėtingus produktus, palaipsniui turėtų būti keliama ir darbuotojų kvalifikacija ir investuojama į darbuotojus. Po truputį ši tendencija visur įsivyrauja ir didesnės įmonės „užsiaugina“ sau darbuotojus, kurie vėliau atitinka jų keliamus reikalavimus. Tai padeda šiek tiek išvengti ir kenkėjiškų kūrėjų bei testuotojų.

Per mažas dėmesys saugumui ir(arba) reikalavimų neapibrėžimas

Jei į saugumą atsižvelgiama tik atmestinai, tikimasi, kad į programinę įrangą nebus laužiamasi arba tiesiog net negalvojama apie saugumą, tokiu atveju, jei programinė įranga sudomins įsilaužėlius jie nesunkiai į ją įsilauš ir imtis saugumo priemonių po eksploatavimo bus kur kas brangiau arba nebeįmanoma jei produkto vardas visiškai sugadintas. Lygiai tokia pati grėsmė išlieka, jei neapibrėžiami tikslūs saugumo reikalavimai.

Netinkami įrankiai

Jei programinei įrangai kurti naudojami netinkami įrankiai, pavyzdžiui, ten, kur reikia aukšto saugumo, kodas rašomas jo nesilaikant arba naudojama netinkama programavimo kalba. Taip pat gali būti ir naudojami netinkami testavimo įrankiai, tokiu atveju bus paliekama spraga, kuria kas nors būtinai pasinaudos. Todėl visada būtina pasirinkti tinkamus įrankius, išmokti juos valdyti ir sekti jų atnaujinimus.

Architektūros ir projektavimo problemos

Spragos programinėje įrangoje gali atsirasti pasirinkus netinkamus sprendimus saugumui užtikrinti ar nesilaikant saugumo reikalavimų projektuojant. Yra tam tikri bendri saugumo reikalavimai kurių būtina visur laikytis, taip pat reikia atsižvelgti į projekto sudėtingumą ir jei reikalinga naudoti papildomus saugumo sprendimus. Dažna problema jog kuriant modulį be apsaugų, kuris planuojamas naudoti tik vidiniame tinkle, vėliau jis panaudojamas išorėje be apsaugų.

Išdėstymo problemos

Jei po testavimo yra nepašalinamos užpakalinės durys (angl. *Backdoor*), neapibrėžta teisingai naudojimo dokumentacija apie konfigūracijas ar duodami kodo pavyzdžiai kurie yra su saugumo spragomis, tada taip pat iškyla grėsmė saugumui. Taip pat svarbu paraginti vartotojus pereiti prie naujesnių versijų, jei tokios yra išleidžiamos nes saugumo sprendimai keičiasi gan greitai stengdamiesi pavyti įsilaužimo grėsmes. Vartotojui neatsinaujinus programinės įrangos, jo programa tampa lengvas taikinys įsilaužėliams.

Gyvavimo problemos

Neišleidžiami naujinimai radus spragas, neatliekamos saugumo analizės darant atnaujinimą bei neatliekamos esamo į rinką išleisto produkto saugumo analizės – visa tai gali būti spragos, atveriančios įsilaužėliams duris. Jei sistema ilgą laiką nebeatnaujinama yra beveik garantuota grėsmė jog į jį gali įsilaužti. Technologijos keičiasi greitai, spragos randamos dažnai ir jas reikia taisyti.

1.2. Dažniausi atakų tipai

Buferio perpildymas

Buferio perpildymas arba duomenų perpildymo klaida, kai buferis užpildytas duomenimis tiek, kad nebėra vietos į jį siunčiamiems naujiems duomenims įrašyti. Dažniausiai tai įvyksta, kai įtaisas nesuspėja apdoroti į jo buferį tiekiamų duomenų ir negali sustabdyti jų tiekimo. Buferio perpildymą galima išnaudoti norint paleisti kenkėjišką programą vartotojo kompiuteryje arba pakeisti programos, kurioje buferis perpildytas, veikimą. Buferio perpildymo pavyzdys Java kalboje:

```
int[]  buffer = new int[10];
for (int i =0; i < 15; i++)
   buffer[i] = 7;
```

Buferio dydis yra 10, bet bandoma įrašyti 15 elementų. Tokiu atveju buferis persipildys ir atsiras spraga įsilaužėliams.

Duomenų arba kodo išgavimas

Programinės įrangos kode gali būti jautrios informacijos kurios neturi pamatyti vartotojai, o juo labiau įsilaužėliai. Išgavus programinės įrangos išeities kodą ir gavus prieigą prie jautrios informacijos galima sužinoti įmonei jautrią informaciją arba tiesiog programos veikimo bei apsaugų principą. Pavyzdžiui, konfigūracinis failas su prisijungimo prie duomenų bazės duomenimis, kuriuos

sužinojus įsilaužėlis gali pasiekti jautrius duomenis jei nėra apribojimų pagal IP adresus kurie gali prisijungti prie duomenų bazės arba tiesiog sudėtingas unikalus algoritmas skirtas spręsti tam tikrai verslo problemai kurį įsilaužėlis nemokamai pasisavina.

Kodas išgavimas naudojant derintuves ir dekompiliatorius. Derintuvės padeda suprasti, kaip veikia PĮ apsaugos, o naudojant dekompiliatorių iš mašininio kodo gaunamas įsilaužėliui perskaitomas kodas.

Vientisumo pažeidimas ir sabotažas

Programinės įrangos natūralaus veikimo trikdymas ir keitimas padaro sistemą neprieinamą ar priverčia veikti ne taip, kaip turėtų. Pavyzdžiui, turime naujienų programą, kuri iš serverio kas 10 min paima didelį kiekį duomenų, patikrina su rodomais vartotojui programoje ir jei yra naujų naujienų jos pridedamos prie rodomų vartotojui. Atradus spragą programa verčiama į serverį kreiptis kas 0,1 s, taip sudarydama serveriui didžiulę apkrovą ir sutrikdydama jo, aptarnaujančio visas programas, veikimą.

Virusų platinimas

Įsilaužėliams išgavus mokamos programos išeities kodą jie į jį įterpia virusą ir paleidžia programą į internetą kaip „nulaužtą“, nemokamą. Vartotojai, pasitikintys gamintojo vardu, parsisiunčia programą ir taip užkrečia savo kompiuterį. Žinomą tam tikros antivirusinės programos aptinka žalingą kodą, tačiau jei virusas naujas, jos bejėgės, nes dauguma jų dirba tikrindamos ar nėra žalingo kodo, kuris anksčiau buvo užfiksuotas ir įrašytas į antivirusinių programų kūrėjų duomenų bazės.

Įsilaužimo tipų yra ir daugiau, jie priklauso nuo įsilaužėlio tikslų, kurie dažniausiai yra:

- gauti informaciją;
- kontroliuoti programinę įranga ar sistemą;
- užkrėsti programinę įrangą ar sistemą;
- sutrikdyti programinės įrangos ar sistemos veiklą.

Už visų šių tikslų dažniausiai yra finansinė nauda. Gavus tam tikrą informaciją – ją galima parduoti. Kontroliuojant programinę įrangą ar sistemą – galima pridaryti žalos konkurentui. Užkrėtus PĮ arba sistemą – galima gauti prieigą prie vartotojų kompiuterių iš kurių vėliau rengiamos atakos, surengus ataką ir sutrikdžius PĮ arba sistemos veiklą – galima padaryti. žalos konkurentams arba uždirbti pinigų iš tų kurie jos nori padaryti.

1.3. Esami kodo apsaugos metodai

Kodo maskavimas (angl. *Code obfuscation*)

Kodo maskavimas yra platus terminas, jis apima įvairius maskavimo metodus ir lygmenis. Šio proceso metu gali būti maskuojamos duomenų struktūros ir valdymo struktūros.

Gali būti maskuojami tik kintamieji (duomenų struktūros maskavimas) ir automatiškai nuimami komentarai (dauguma maskavimo programų turi šią funkciją). Jei norima neapkrauti programos ir didelis saugumas nėra prioritetas šis metodas puikiai tiks. Pavyzdys - „Java“ kalba:

Originalus kodas	Užmaskuotas kodas
`String first,second;` `double choice;`	`String l10,O11;` `double l100;`

Galima maskuoti ir visa kodą keičiant valdymo struktūras. Pavyzdys:

<table>
<tr><td>Originalus kodas</td><td>Užmaskuotas kodas</td></tr>
<tr><td><pre>
function original()
{
   if ($a == 5) {
      echo “veikia”;
   }
}
</pre></td><td><pre>
function obfuscated()
{
   switch($a) {
      case 1:
         echo “ne”;
         break;
       case 3:
         break;
       case 5:
         echo “veikia”;
       case (‘atsitiktinis‘):
         echo “atsitiktinis”;
         break;
    }
}
</pre></td></tr>
</table>

Pavyzdyje buvo pakeista valdymo struktūra. Kodas šiuo atveju dar yra suprantamas, bet tai tik labai elementarus pavyzdys, praktikoje naudojami kur kas sudėtingesni struktūrų pakeitimai. Kodas taip pat atrodo daug sudėtingiau kai jo originali struktūra yra didelė ir sudėtinga.

Maskuoti galima net ir klasių bei failų pavadinimus, kad būtų sunkiau atsekti kas su kuo susiję. Maskuojant klasių ar funkcijų pavadinimus reikia įdėti į programą papildomus parametrus konfigūruojant maskavimą. Parametruose turi būti pasirinkimas, kokių klasių ar funkcijų nemaskuoti nes pavyzdžiui užmaskavus konstruktorių programa nebeveiks.

Nepatartina naudoti visiems žinomų maskavimo įrankių. Tokių įrankių algoritmai daugiau ar mažiau žinomi ir įsilaužėliams, todėl unikalūs maskavimo algoritmai visada duoda daugiausiai naudos.

Maskuojant kai kurių klasių ar metodų pervadinti yra neįmanoma, tad geriausia naudoti maskavimo algoritmą kuris pavadinimus pakeičia į realius (*Rummage* algoritmas), bet sunkiai suprantamus negu, kad į visiškai nelogiškus (standartinis algoritmas), pavyzdys [6]:

Prieš maskavimą	Po standartinio algoritmo	Po “Rummage” algoritmo

class Program
 	
class Settings
 	
class Reflected
    field Count
    method Update
 	
class LicenseChecker
    field LicenseData
    method get_UserName
    method IsLicenseValid
    method ReadLicense
	class a
	
	class b
	
	class Reflected
	    field Count
	    method Update
	
	class c
	    field a
	    method b
	    method c
	    method d
		Class BrowseEntry
		
		Class Fixed
		
		Class Reflected
		  field Count
		  method Update
		
		Class InfoState
		  field Binder
		  method SetChanged
		  method Remove
		  method Service

Kaip matome po standartinio algoritmo yra aišku, kurios klasės nepavyko pervadinti, o po „Rummage“ ne. "Rummage“ algoritmas pakeičia klasių pavadinimus į atsitiktinius tačiau logiškus ir suprantamus žodžius, kurie susilieja su neužmaskuotais pavadinimais.

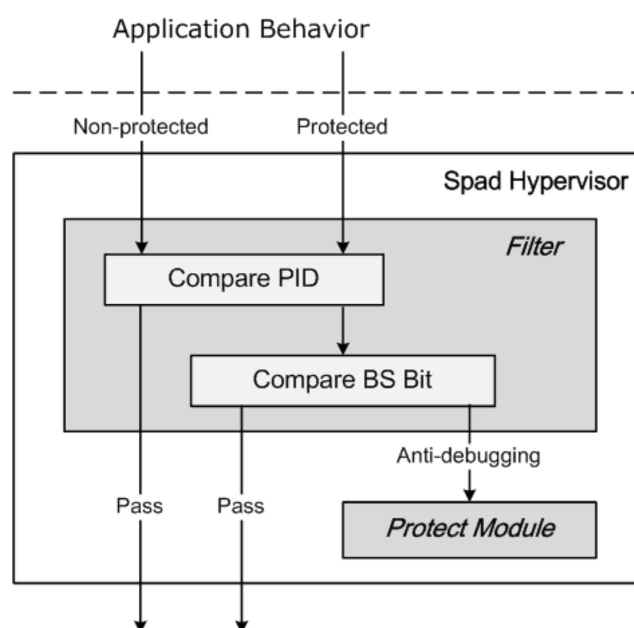
Stabdos taškai (angl. *Breakpoints*)

Derintuvės gali pertraukti instrukcijas INT1 ir INT3 x86 sistemose [5].

INT1 – tai vieno žingsnio pertraukimas kai programa derinama po vieną instrukciją vienu metu.

INT3 – tai pertraukimai, kurie gali būti sudėlioti bet kur programoje ir procesoriui juos apdorojant, kontrolė perduodama derintuvei.

Norint patikrinti, ar derintuvė nebuvo naudojama, reikia patikrinti, ar vykdant INT1 ir INT3 instrukcijas jos nebuvo pertrauktos. Pavyzdys pateiktas 1.1. paveiksle.



1.1. pav. Derinimo aptikimas pertraukiant INT1 ir INT3 instrukcijas [7]

„*Spad*“ prižiūryklė (angl. *Hypervizor*) patikrina, kada DR6 registro BS bitas yra pertrauktas. Jeigu BS bitas išjungtas, tada programa pereina į apsaugos režimą.

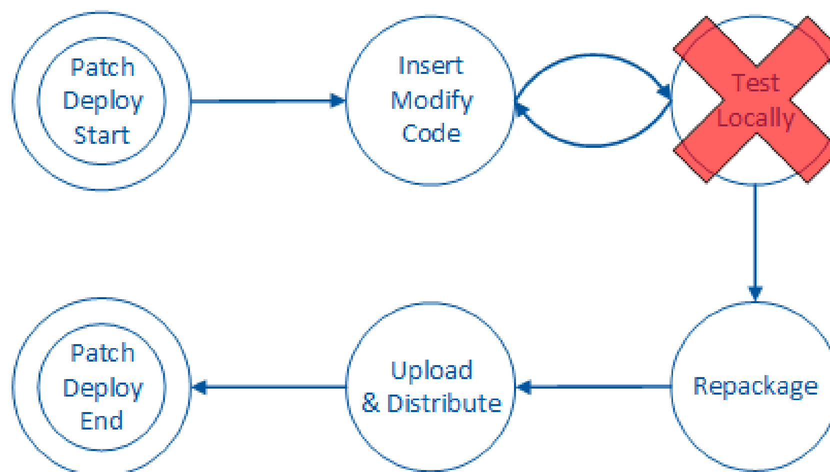
Apsauga nuo derintuvių (angl. *Anti-debugging*)

Vienas iš paprasčiausių būdų apsisaugoti nuo derintuvių yra įdiegti į programinę įrangą žinomų derintuvių aptikimą. Jei įsilaužėlis yra įjungęs derinimo procesą, jis yra aptinkamas ir programinė įranga nepasileidžia.

Skirtingose programavimo kalbose yra specialios funkcijos bei bibliotekos, skirtos aptikti, jei įjungta programa yra derinama. Pavyzdys C++ kalboje [4]:

```
BOOL WINAPI CheckRemoteDebuggerPresent(
   _In_      HANDLE hProcess,
   _Inout_   PBOOL pbDebuggerPresent
);
```

Kitas apsaugos metodas nuo derinimo yra laiko tarp instrukcijų vykdymo tikrinimas (angl. *Runtime verification*). Derinant programinę įrangą vientisumo kontrolė patikrina jai žinomus laikus tarp instrukcijų vykdymo ir jei jie skiriasi, tokiu atveju programinės įrangos kūrėjas turi nuspręsti, ką daryti, suprogramuodamas išimtis tokiems variantams [3]. Derinimo aptikimas turi būti įvykdytas iki testavimo fazės kaip pavaizduota 1.2 paveiksle ir tuomet testavimas nutraukiamas, kad įsilaužėlis negalėtų perprasti PĮ veikimo.



1.2. pav. Derinimo aptikimas laiko tikrinimo tarp instrukcijų metodu. [8]

Idealios derintuvės, kurių aptikti būtų beveik neįmanoma - „tyliosios derintuvės“ (angl. *Stealth debuggers*), šiuo metu yra kuriamos, deja, nesėkmingai. Programinės įrangos kūrėjų laimei jos nėra pilnai išvystytos ir jas vis dar galima aptikti, tačiau daug sunkiau.

Programinės įrangos ženklinimas ir parašai

Kaip papildomos apsaugos priemonės yra naudojami programinės įrangos ženklinimai ir (arba) parašai. Šis apsaugos metodas dažniausiai naudojamas programinėje įrangoje, kuri naudojama vartotojų kompiuteriuose ir gali veikti be interneto.

Vandens ženklai nepadeda nuo įsilaužimų į programinę įrangą, tačiau jie padeda aptikti tikrąjį programinės įrangos savininką [11]. Jie paslepiami programinės įrangos kode, dažniausiai keliose vietose. Jei įsilaužėlis panaudos savo programinėje įrangoje kodą, kurį jis gavo neteisėtai, vėliau vandens ženklas gali būti "ištraukiamas" iš programinės įrangos kodo specialiais algoritmais, kurie parodys, kad kodas ar jo dalys įsilaužėliui nepriklauso ir tuomet bus nesunku įrodyti, kad kodas buvo pavogtas ir panaudotas be leidimo. Norint, kad vandens ženklą būtų sunku aptikti, jį reikia įterpti asemblerio lygmenyje pavyzdžiui:

```
movl      $44, %eax
```

Vandens ženklo reikšmė yra 4436.

Be abejo, vandens ženklą galima panaudoti ir aukštesniame lygmenyje: Java, C++ ar PHP programavimo kalbų kode, tačiau jį bus lengviau aptikti ir pašalinti.

Parašai yra skirti aptikti, ar nupirkta programinė įranga naudojama viename kompiuteryje, ar buvo paplatinta ir naudojama ne pagal licenciją. Paleidus programinę įrangą patikrinama, ar jos procesas veikia tame pačiame kompiuteryje, kuriame ji buvo pirmą kartą įdiegta. Pirmą kartą diegiant programinę įrangą, analizuojamas kompiuteris, jo parametrai ir įvairūs unikalūs numeriai. Kiekvieną kartą paleidus programinę įrangą, turimi duomenys patikrinami. Dėl šios priežasties, įrašant "nulaužtą" programinę įrangą, vartotojo giduose prašoma išjungti internetą ir nedaryti jokių atnaujinimų, nes diegimo metu gauti parametrai saugomi ne tik lokaliai, bet ir gamintojo duomenų bazėse.

Kodo išskaidymas

Kodo išskaidymas skirstomas į linijinį nuskaitymą (angl. *Liner-Scaning*) ir rekursinį žingsniavimą (angl. *Recursion-marching*) [12].

Linijinio nuskaitymo metu algoritmai tiesiog pereina per mašininį kodą ir visą jį išverčia į asemblerio instrukcijas. Pagrindinis jo trūkumas - nemokėjimas atskirti duomenų nuo kodo.

Rekursino žingsniavimo algoritmai skenuoja visus įmanomus vykdymo kelius ir išardo vykdymo instrukcijas kiekvienoje atšakoje, kai aptinka perdavimo instrukciją. Šiuo metodu galima prieiti prie statinių duomenų kode, tačiau negalima prieiti prie dinaminės informacijos.

- Linijinio nuskaitymo algoritmams apsauga yra niekinis kodas (angl. *Junk code*).

  Niekinis kodas yra skaidomas į duomenis ir perdėtas niekines instrukcijas. Jis ne tik, kad pakenks išrinkėjui (angl. *Disassembler*), bet ir padarys kodą sunkiai skaitomu. Tačiau ši apsauga yra niekinė rekursinio žingsniavimo algoritmo metu.

- Rekursinio žingsniavimo algoritmui apsauga yra kodo šifravimas (angl. *Code encryption*).

  Pagrindinis metodas yra besikeičiantis kodas (angl. *Self modifying code*). Jis keičia niekines instrukcijas originaliu kodu, kurį norime apsaugoti, o originalų kodą niekinėmis instrukcijomis ir tai daro kas tam tikrą laiko intervalą. Kadangi kodas gaunasi dinaminis, rekursinio žingsniavimo statiniams algoritmams jis tampa neįveikiamas.

  Dažniausiai šis metodas naudojamas žemo lygmens programavimo kalbose, tokiose kaip asemblerio kalba, tačiau yra pavyzdžių, kur naudojamas ir aukšto lygmens kodavimo kalbose, tokiose kaip JavaScript:

```
var a = function (x) {return x + 1};
// Priskiriama nauja a reikšmė
a = new Function('x', 'return x + 2');
```

Esamas kodas šiuo atveju nekeičiamas, tačiau keitimo iliuzija gali būti sukuriama keičiant funkcijos rodykles. Tokioje kalboje kaip JavaScript kodo išskaidymas nėra labai naudingas, nes įvairių įrankių pagalba galima nesunkiai išgauti vykdomą kodą, o ne išskaidytą.

Fizinės įrangos apsauga

Programinę įrangą galima apsaugoti ir naudojant fizinę įrangą. Tam naudojami įvairūs saugūs procesoriai, lustinės kortelės bei raktai. Dažniausiai užšifruojant įrenginį yra apsisaugoma nuo įsilaužėlių, kurie neturi tinkamos įrangos įsilaužimui, tačiau prieš pasiruošusius įsilaužėlius to ne visada pakanka - reikalinga ir programinė apsauga. Fizinės įrangos apsaugos metodai plačiau šiame darbe neanalizuojami, nes šiame darbe telkiamasi į apsaugos metodus programiniame lygmenyje.

1.4. Kodo maskavimas ir įrankiai

Kodo maskavimas yra efektyviausias, kai maskuojamas kodas yra parašytas su žemo lygmens programavimo kalbomis [20]. Žemo lygmens programavimo kalbose galima keisti kodą vykdymo metu, kodas ir duomenys gali būti maišomi, bei valdymo struktūros gali būti nesunkiai keičiamos bet kokia tvarka. Dėl šių savybių kodo maskavimas tampa daug paprastesnis maskavimo metu ir daug sudėtingesnis bandant išgauti originalų kodą iš užmaskuoto.

Maskavimo rūšys

Maskavimo metodų yra sukurta labai daug, vieni jų efektyvūs prieš vieno tipo atakas, kiti prieš kito tipo. Pagrindiniai maskavimo metodai yra:

- klasių, metodų ir kintamųjų maskavimas;
- masyvų maskavimas;
- valdymo struktūrų maskavimas;
- kriptografinis maskavimas;
- iniciatyvus (angl. *Proactive*) maskavimas.

Galima rasti ir kitokių maskavimo metodų, kurie nėra tokie populiarūs. Maskavimo apsaugos būdas yra labai platus ir atviras galimybėms.

Maskuojant klasių, metodų ir kintamųjų pavadinimus, siekiama apsunkinti įsilaužimą programuotojui, kuris pats, be įrankių analizuoja kodą. Šis metodas efektyvus, kai įsilaužėliui reikalingas PĮ kodas, siekiant jį panaudoti kur nors kitur. Tai yra bazinė apsauga nuo kodo vagystės.

Masyvų maskavimas dažniausiai naudojamas su statiniais masyvais, kur duomenys nesikeičia [17]. Naudojant šį metodą su dinaminiais masyvais, galima pakenkti greitaveikai, nes nežinant masyvo struktūros, elementų kiekio ar gylio, nėra galimybės tinkamai optimizuoti maskuojamą kodą. Maskuojant masyvus galima keisti jų rodykles arba struktūrą. Maskavimas keičiant masyvo struktūrą yra gan paprastas: vienos dimensijos masyvą [m x n] galima pakeisti į kelių dimensijų masyvą [m, n]. Masyvai taip pat maskuojami juos išskaidant. Vieną masyvą galima išskaidyti į kelis vienodo dydžio masyvus, tačiau tam reikia sugeneruoti funkciją, kuri vėliau apdoros duomenų paėmimą iš atskirų masyvų.

Prieš maskuojant masyvą, labai svarbu patikrinti ar jis yra tinkamas maskavimui. Jei algoritmas parašytas maskavimui vienos dimensijos masyvui, o bus paduodamas dviejų ar daugiau dimensijų masyvui, tuomet masyvas gali būti netinkamai užmaskuotas ir dėl to maskuojamos programos logika veiks ne taip, kaip turėtų.

Valdymo struktūrų maskavimas daromas keičiant valdymo struktūrų tipus, gylius bei pozicijas. Plačiau apie valdymo struktūrų maskavimą aprašyta 2.4 skyriuje.

Kriptografinis maskavimas yra viena iš dar tiriamų sričių. Iki dabar nėra nei vieno komercinio maskavimo įrankio naudojančio šią technologiją [18]. Pilnai ištobulinus šį maskavimo metodą, teoriškai programa taptų neįveikiama. Tikslus veikimo principas šiam metodui nėra dar surastas. Šis maskavimas paremtas juodosios dėžės (angl. *Black box*) principu, kai iš užmaskuotos vartotojo programos gaunama įvestis, ji užšifruojama, ir serveryje esanti juodosios dėžės programa sugeneruoja išvestį nieko nežinodama apie duomenis, kuriuos ji gavo.

Iniciatyvus maskavimas yra dar ganėtinai naujas metodas. Jis buvo sugalvotas 2010 metais [19]. Šio maskavimo rezultatas yra kodas, esantis serverio pusėje, kuris kiekvieno vykdymo metu yra per naują užmaskuojamas. Serveris yra „perkraunamas“ kiekvieno vykdymo metu, užmaskuojant kodą kitu algoritmu, naudojant slaptąjį raktą, kuris yra serveryje. Šis metodas gali naudoti visus kitus maskavimo metodus kartu su kriptografiniu sprendimu. Šio maskavimo principas naudoja jau ankščiau sugalvotą principą, kuris teigia: geriau nutraukti programos vykdymą nei atiduoti kontrolę įsilaužėliui. Įsilaužėliui pradėjus dekompiliuoti programą, ji tiesiog nutraukiama su klaida. Tad norint naudoti šį metodą, reikalingas derintuvių ir dekompiliatorių aptikimas bei kriptografinis

sprendimas sujungtas su įvairiais maskavimo metodais. Nors metodas ir sugalvotas prieš keletą metų, tačiau įrankių sukurta yra labai mažai, galbūt dėl sprendimo sudėtingumo.

Maskavimo trūkumai ir sprendimai

Yra sukurta labai daug įrankių skirtų maskavimui, ir visi jie turi savų privalumų bei trūkumų. Pavieniai maskavimo įrankiai kuriami su tam tikru specifiniu tikslu, o vėliau išplitę internete su visais esamais trūkumais, kuriais pasinaudoja įsilaužėliai, naudojami apsaugoti PĮ išeities kodui [2]. Dėl šios priežasties yra sunku surasti tinkamą maskavimo įrankį kuris PĮ kodą pilnai apsaugotų ir nepadarytų žalos PĮ veikimui ar greitaveikai. Dėl to yra kuriami didelių kompanijų komerciniai įrankiai, kurie kainuoja didelius pinigus, tačiau ir jie yra ne visoms situacijoms tinkami.

Tokios kompanijos kaip „Google“, turi savo parašytus maskavimo įrankius, kurie maskuoja kodą pagal jiems tinkamus parametrus ir taisykles. Tai daro ir kitos didžiosios įmonės kurios gali skirti resursų tokio įrankio kūrimui.

Žemiau pateiktoje lentelėje išvardinta keletas dažnai pasitaikančių maskavimo įrankių trūkumų ir sprendimai trūkumams išspręsti.

1.1 lentelė. Maskavimo trūkumai ir sprendimai

Trūkumai	Sprendimai	Įrankis turintis sprendimą
Dėklo pėdsako (ang. *Stack trace*) žemėlapis laikomas lokaliai ir prieinamas įsilaužėliams, dėl to išgaunamas originalus, neužmaskuotas kodas.	Dėklo pėdsako (ang. *Stack trace*) žemėlapį laikyti nutolusiame serveryje, prie kurio priėjimą turi tik PĮ kūrėjas ar atsakingi asmenys.	ProGuard
Užmaskuotas kodas vis dar perskaitomas tiek žmogaus, tiek automatinių įrankių.	Maskuoti kodą keliais sluoksniais (ang. *Multi-layer*) ir skirtingais algoritmais.	Safenet
Dažniausiai PHP programavimo kalbos maskavimo įrankiai tiesiog užkoduoja kodą su „base64_encode()“ ar kita panašia funkcija, kurį vėliau galima vykdyti su „eval()“ funkcija, tačiau lygiai taip pat	PHP programavimo kalbos kodą keisti su nestandartiniais algoritmais, kurie apsunkintų kodo skaitymą (PHP kodą paslėpti labai sunku, galima tiesiog apsunkinti įsilaužėliui kodo skaitymą).	Naneu PHP obfuscator

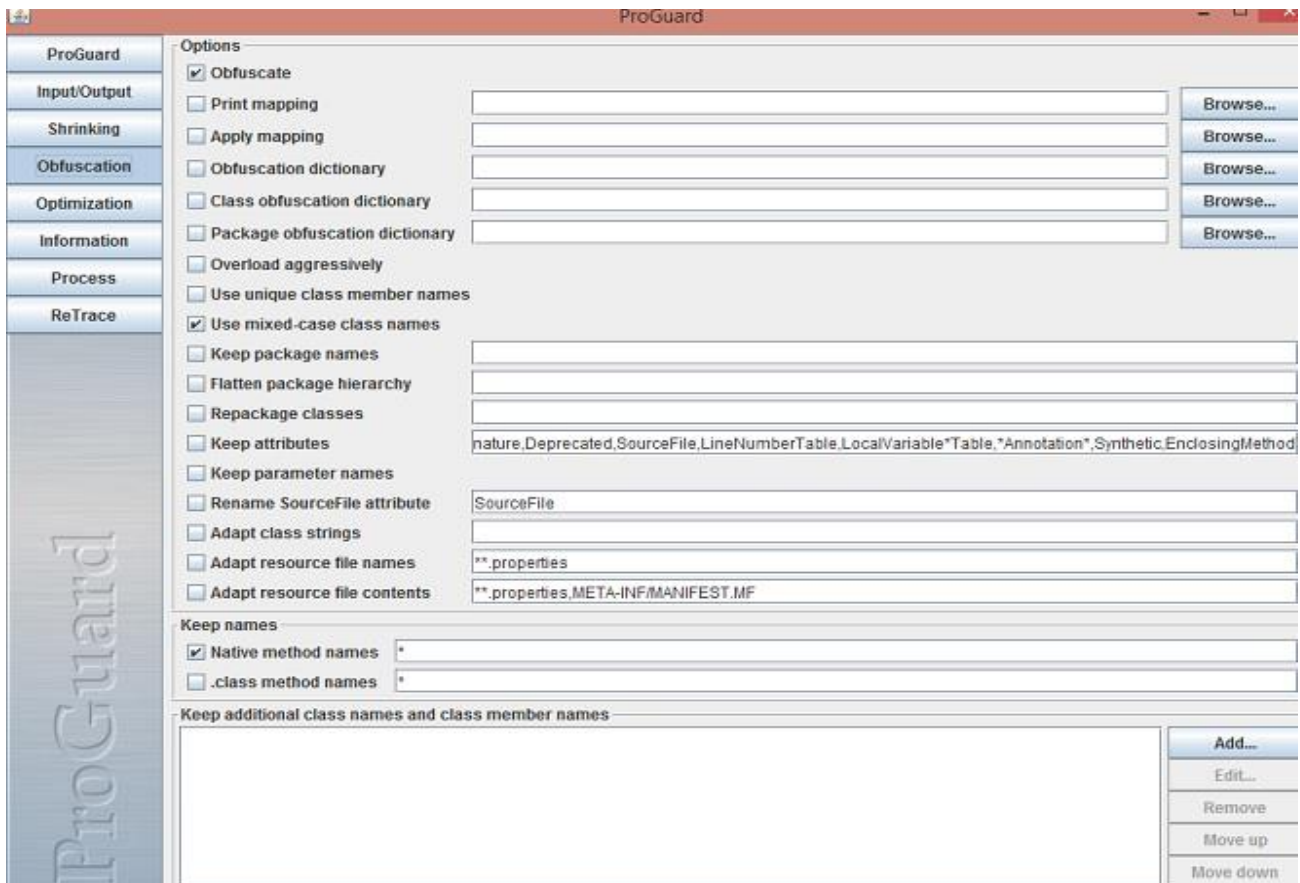
lengvai galima jį išversti į pradinį kodą.		
Maskuojant kodą, keičiant valdymo struktūras, jis stipriai padidėja, dėl to nukenčia greitaveika.	Įterpti kodo optimizavimo, sutraukimo (ang. *Shrinking*) funkcijas į maskavimo įrankį, kurių pagalba ten kur įmanoma, maskuojant kodas būna sutraukiamas. Taip pat reikia laiku nutraukti perteklinių valdymo struktūrų veikimą, kad jos nebūtų vykdomos tada, kai nereikia.	ProGuard, yGuard, Safenet
Naudojant atspindžio programavimą (ang. *Reflection programming*), klasių bei naudojamų funkcijų pavadinimai negali būti maskuojami, nes kodas keičiamas programos vykdymo metu (ang. *Runtime*).	Įterpti į įrankį papildomą nustatymą, kuris leistų detaliai pasirinkti kurių kodo vietų nemaskuoti.	ProGuard, Safenet, yGuard, Naneu PHP obfuscator

Maskavimo įrankių apžvalga

ProGuard:

Tai vienas iš pažangiausių atvirojo kodo maskavimo įrankių skirtų Java programavimo kalbai. Pats įrankis taip pat suprogramuotas Java programavimo kalba. Šiame įrankyje gausu nustatymų, kurie leidžia ne tik įvairiais metodais maskuoti kodą, bet ir jį optimizuoti. Maskuojant optimizavimas yra ypač svarbus, nes dažniausiai greitaveika nukenčia maskavimo metu. Šį įrankį galima rasti adresu: http://proguard.sourceforge.net/

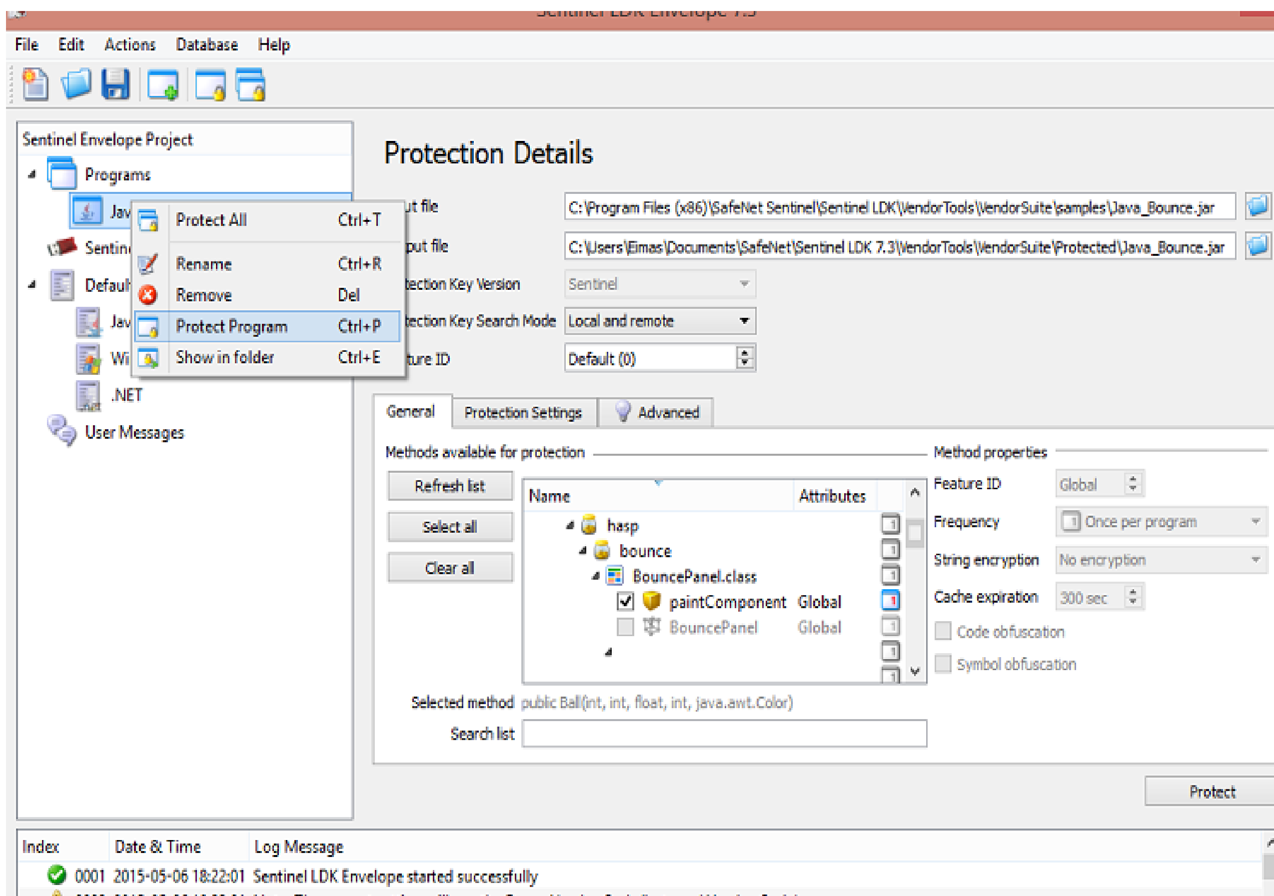
Žemiau pateiktame paveiksle pavaizduotas „ProGuard“ maskavimo įrankio grafinis vaizdas, kuriame matoma daugybė nustatymų. Šie nustatymai išsprendžia kai kurias problemas minėtas 1.1 lentelėje.



1.3. pav. ProGuard maskavimo nustatymai

Safenet:

Įrankis yra mokamas ir skirtas Java ir C# programavimo kalbų maskavimui. Šis įrankis yra plačiai naudojamas įvairių garsių kompanijų, tokių kaip „Siemens“, „Cannon“, „Philips“ ir kt. Algoritmas suteikia dviejų sluoksnių maskavimo galimybę, kuri ženkliai apsunkina darbą įsilaužėliams, norintiems išgauti iš PĮ originalų kodą. Šiame įrankyje taip pat yra reikiami nustatymai, tokie kaip klasių ar funkcijų pasirinkimas, kurių maskuoti nenorime. Kodo sutraukimas (ang. *Shrinking*) yra vykdomas automatiškai, ir nustatymo jį išjungti nėra. Grafinė įrankio sąsają pavaizduota 1.4 paveiksle. Oficialus tinklalapis: http://www2.safenet-inc.com



1.4. pav. Safenet maskavimo nustatymai

yGuard:

Šis įrankis yra skirtas su Java programavimo kalba kurtai PĮ, grafinės sąsajos jis neturi. Pagrindinės įrankio užduotys yra kodo maskavimas ir sutraukimas [15]. Įrankis išarchyvuojamas ir įkeliamas prie kitų kompiliuojamų failų su konfigūracijomis „XML“ tipo faile. Tuomet „build.xml“ faile įrašomos papildomos eilutės:

```
<target name="yguard">
      <taskdef name="yguard"
      classname="com.yworks.yguard.YGuardTask"
      classpath="yguard.jar"/>
      <yguard>
          <!-- insert your yguard elements here -->
      </yguard>
</target>
```

Kompiliuojant paleidžiamas automatinių užduočių servisas „Ant task“ [14] su maskavimo užduotimi. Šis įrankis taip pat turi visas reikiamas konfigūracijas.

Produkto oficialus tinklalapis yra: https://www.yworks.com/products/yguard

Naneu PHP Obfuscator:

Tai vienas iš geriausių PHP atvirojo kodo maskavimo įrankių. Šis įrankis naudoja unikalius algoritmus maskavimui, priešingai nei dauguma PHP maskavimo įrankių, kurie naudoja „encode“ funkcijas ir vykdomas kodas tuomet su „eval()“ funkcija, kas nėra saugu ir netenka maskavimo prasmės. Šis įrankis atitinka dauguma keliamų reikalavimų geriems maskavimo įrankiams. Jis turi:

- unikalų maskavimo algoritmą;
- konfigūracijas, kurių pagalba išskiriamos klasės ar kintamieji, kuriuos reikia ignoruoti (to reikia naudojant atspindžio programavimą);
- kodo optimizavimo funkcijas.

Jo trūkumas yra toks, kad nėra funkcijų, kurios maskuotų kodą keičiant valdymo struktūras, jame pakeičiami tik klasių, funkcijų ir kintamųjų pavadinimai, tačiau struktūra išlieka, nors ir viskas sutraukiama, bet skyrus šiek tiek daugiau laiko, net ir sudėtingą, maskuotą sistemą su šiuo įrankiu įsilaužėlis gali perprasti ir po truputį išgauti originalų kodą.

Šis įrankis veikia komandinės eilutės pagalba, žemiau pateiktame paveiksle galima matyti pavyzdį:

```
            php-obfuscator-master]# ./bin/obfuscate obfuscate /var/www/html /va
r/www/test
Copying input directory /var/www/html to /var/www/test
Obfuscating /html/a.php
[root@312386 php-obfuscator-master]#
```

1.5. pav. Naneu PHP obfuscator komandinė eilutė

Kaip matome pavyzdyje struktūra lieka nepakitusi, dėl to kodą galime lengvai suprasti.

Originalus kodas:	Užmaskuotas kodas:

for ($i=0; $i<10; $i++){
echo $i."
";
}
	for ($spda6bba = 0; $spda6bba < 10; $spda6bba++) { echo $spda6bba . '
	'; }

1.5. Įsilaužimų ir apsaugų apibendrinimas

Žemiau pateikiama lentelė su pagrindiniais įsilaužimo būdais ir metodais skirtais apsisaugoti nuo jų.

1.2. lentelė. Įsilaužimai ir apsaugos

Įsilaužimas/Apsauga	Kodo maskavimas	Stabdos taškai	Apsauga nuo derintuvių	Programinės įrangos ženklinimas ir parašai	Kodo išskaidymas	Kanarėlės reikšmė skirta aptikti buferio perpildymui
Buferio perpildymas						+
Duomenų(kodo) išgavimas	+	+	+		+	

Vientisumo pažeidimas ir sabotažas		+	+		+	
Virusų platinimas	+	+	+	+	`+	

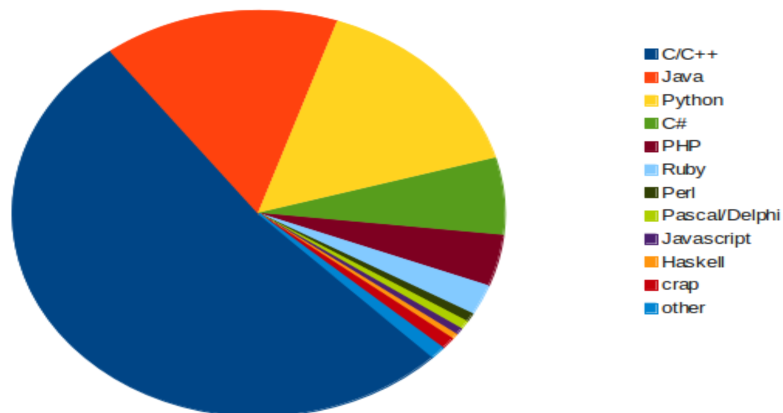
Norint išvengti buferio perpildymo, reikėtų nedaryti klaidų. Tačiau klaidos yra daromos, tad yra sugalvota kanarėlės reikšmė, kuri yra sunaikinama, jei buferis perpildomas ir tuomet nutraukiamas procesas.

Saugantis nuo duomenų (kodo) išgavimo, reikia slėpti kodą bei saugotis nuo derintuvių ir dekompiliatorių, tam naudojamos kompleksinės apsaugos. Tokia pati situacija ir su virusų platinimu, norint įsilaužėliui įterpti žalingą kodą į PĮ, pirma jam reikia išgauti originalų kodą.

Norint apsisaugoti nuo vientisumo pažeidimo ir sabotažo, reikia įsilaužėliui neleisti sužinoti apie PĮ apsaugas, todėl reikia saugotis derintuvių. Įsilaužėliui sužinojus apie apsaugas ar spragas, jis jomis pasinaudos, įvykdys sabotažą ar vientisumo pažeidimą ir sutrikdys sistemą.

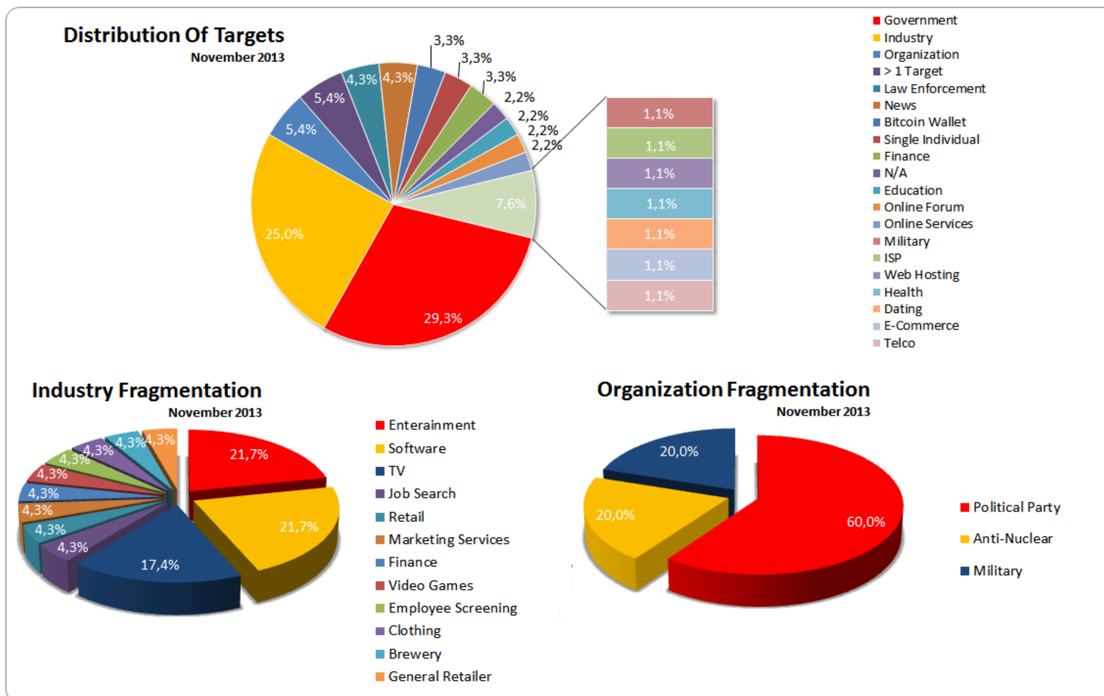
1.6. Statistika

Socialinio tinklo „Facebook“ surengto įsilaužėlių konkurso naudojamų kalbų statistika (išviso dalyvavo 20348 įsilaužėliai).



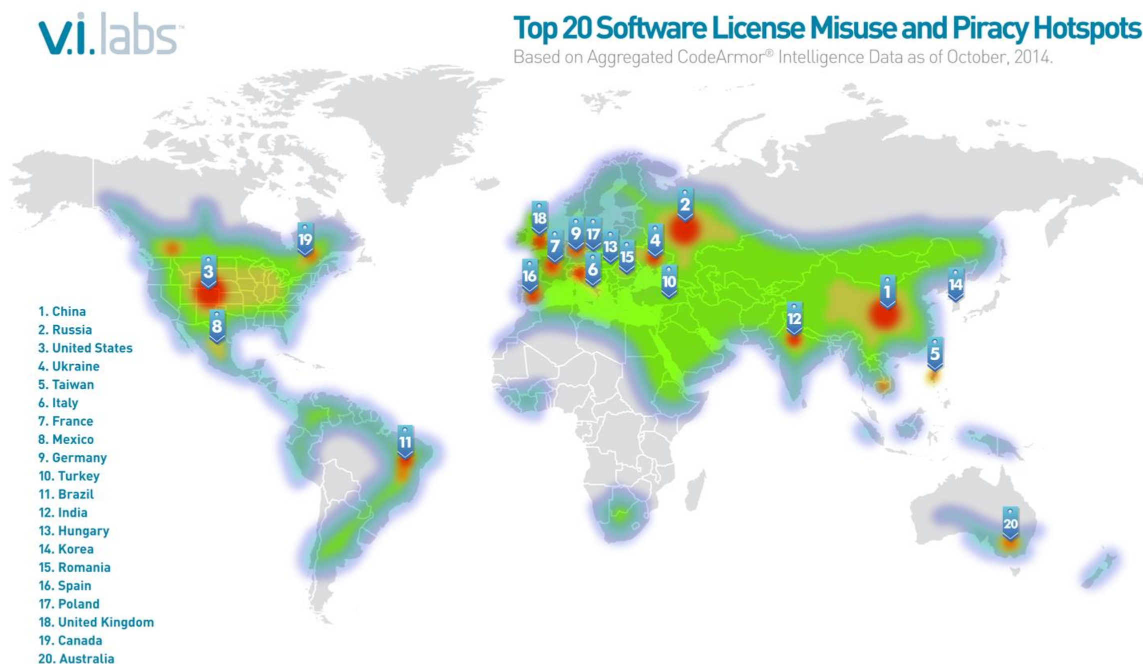
1.6. pav. Įsilaužėlių naudojamų kalbų statistika [9]

Dažniausiai naudojamos žemesnio lygmens C ir C++ kalbos, su kuriomis galima valdyti procesus bei atmintį. Be abejo, tai nėra visiškai realaus pasaulio statistika, nes aukšto lygio įsilaužėliai naudoja įvairias kalbas rašyti savo įrankiams, kurias jie renkasi pagal situaciją.



1.7. pav. Atakų paskirstymo statistika [10]

Dažniausiai atakuojami sektoriai yra industrijos ir valdžios institucijų. Taip pat 21.7% atakų sudaro programinės įrangos įsilaužimai – tai yra dideli skaičiai.



1.8. pav. Šalys kuriose daugiausiai naudojama nelegali PĮ [13]

Kaip matome iš pateiktų „v.i.labs“ surinktų duomenų, labiausiai paplitusi nelegali PĮ yra Kinijoje, Rusijoje ir JAV. Tai natūralu, nes tai yra didelės valstybės, plačiai vystančios kompiuterijos technologijas. Todėl įsilaužėlių jose gyvena taip pat nemažai.

1.7. Analizės išvados

Įsilaužimo į PĮ būdų yra daug, taip pat kaip ir apsaugos metodų, todėl norint efektyviai apsaugoti PĮ, reikia naudoti kompleksinius bei nestandartinius algoritmus (nesinaudoti gerai žinomais maskavimo įrankiais, generatoriais). Naudojant pavienius apsaugos metodus, galima apsisaugoti nuo tam tikrų įsilaužimo metodų, tačiau universalios apsaugos, kuri apsaugotų nuo visų įsilaužimo būdų, nėra, todėl reikia analizuoti galimas grėsmes ir taikyti kompleksinius apsaugų metodus.

Plačiausiai naudojamos įsilaužėlių programavimo kalbos yra žemo lygmens, su kuriomis galima valdyti procesus ir atmintį. Dažniausi įsilaužimai vyksta prieš valdžios įstaigas ir verslą. Pagrindinės priežastys būna - dėl pramogos arba siekiant įsilaužti į konkrečią programinę įrangą.

2. SIŪLOMI „NANEU PHP OBFUSCATOR“ ĮRANKIO PAKEITIMAI (PROJEKTAVIMAS)

Ištyrus keletą įrankių buvo atrastas „Naneu PHP obfuscator“ įrankis, kuris yra vis dar tobulinamas ir turi unikalų kodo maskavimo algoritmą bei atitinka pagrindinius maskavimo įrankių reikalavimus. Vienas iš jo trūkumų yra tai, kad įrankis negali maskuoti kodo keičiant valdymo struktūras, ir užmaskuotas kodas tampa pažeidžiamas dėl statinės analizės įrankių. Kodo maskavimas keičiant valdymo struktūras ir bus kuriamas šiam atvirojo kodo įrankiui.

2.1. Valdymo struktūra

Maskuojant „if“ valdymo struktūrą, ji bus keičiama į „switch“ valdymo struktūrą. Galimas maskuotos „if“ valdymo struktūros pavyzdys prieš ir po pakeistų valdymo struktūrų ir kintamųjų pavadinimų:

<table>
<tr><th>Originalus kodas</th><th>Užmaskuotas kodas</th></tr>
<tr><td>
<pre>
if ($a == $c) {
   return true;
}
</pre>
</td><td>
<pre>
$cbstgdzs = $a;
switch($a){
   case 1:
      return false;
      break;
   case “works“:
      return false;
      break;
   case $$cbstgdzs:
      return false;
      break;
   case $cbstgdzs:
      return true;
      break;
   default:
      return false;
}
</pre>
</td></tr>
</table>

Tai tik paprasta „if“ valdymo struktūra. Jei sujungtume keletą algoritmų valdymo struktūroms keisti, sąlyga taptų dar sunkiau skaitoma. Šis įrankis taip pat sutraukia kodą, dėl to įsilaužėliui tenka sugaišti dar daugiau laiko, bandant suprasti kodą. Valdymo struktūrų maskavimas šiame darbe ir bus daromas. Norint pasiekti tikslą, reikia sukurti papildomas funkcijas, kurios modifikuos išgautą originalų kodą. Kodo išgavimo išskaidymo funkcijos jau yra įgyvendintos šiame įrankyje, tad šias funkcijas reikės panaudoti, šiek tiek pakeitus jų funkcionalumą ir parašyti algoritmus valdymo struktūrų maskavimui.

Aukščiau esame pavyzdyje kodas vis dar pažeidžiamas ir gali būti išanalizuotas su statinės analizės įrankiais. Valdymo struktūros yra pakeistos tačiau jos išlieka tame pačiame lygmenyje.

Norint apsisaugoti nuo statinės analizės įrankių reikėtų pakeisti valdymo struktūrų lygmenį [16], pavyzdžiui:

```
$cbstgdzs = $a;
while ($cbstgdzs == $a) {
   switch($a){
      case 1:
         return false;
         break;
      case "works":
         return false;
         break;
      case $$cbstgdzs:
         return false;
         break;
      case $cbstgdzs:
         return true;
         break;
      default:
         if (1 == 1) {
            switch ($cbstgdzs) {
               case $cbstgdzs:
                  return true;
                  break;
               default:
                  break;
            }
            return false;
            break;
   }
}
```

Šiame pavyzdyje tikroji valdymo struktūra, kuri grąžina „true“ yra perkelta į kitą lygmenį ir randasi viduje perteklinių valdymo struktūrų. Statinės analizės įrankis grąžintų kitokią kodo struktūrą negu ta kuri yra iš tikrųjų. Vietoje vienos valdymo struktūros su 1 sąlyga, užmaskuotame kode yra 8 sąlygos su 2 papildomomis valdymo struktūromis.

2.2. Valdymo struktūrų aptikimas bei pakeitimas

Viena iš užduočių yra aptikti valdymo struktūras. Jos gali būti aprašytos skirtingai pagal programuotojo stilių, kuris programuoja, o pakeitimus reikia padaryti vienodai nepriklausomai nuo stiliaus ar modeliavimo metodų. Šis maskavimo įrankis yra ganėtinai didelis, turi įvairių funkcijų, kai kurios neveikiančios (įrankis atvirojo kodo ir yra vis tobulinamas), todėl, prieš kuriant kodo maskavimą keičiant valdymo struktūras, teks kruopščiai išanalizuoti visą jo kodą ir nuspręsti kurias iš jau esamų sukurtų funkcijų galima panaudoti.

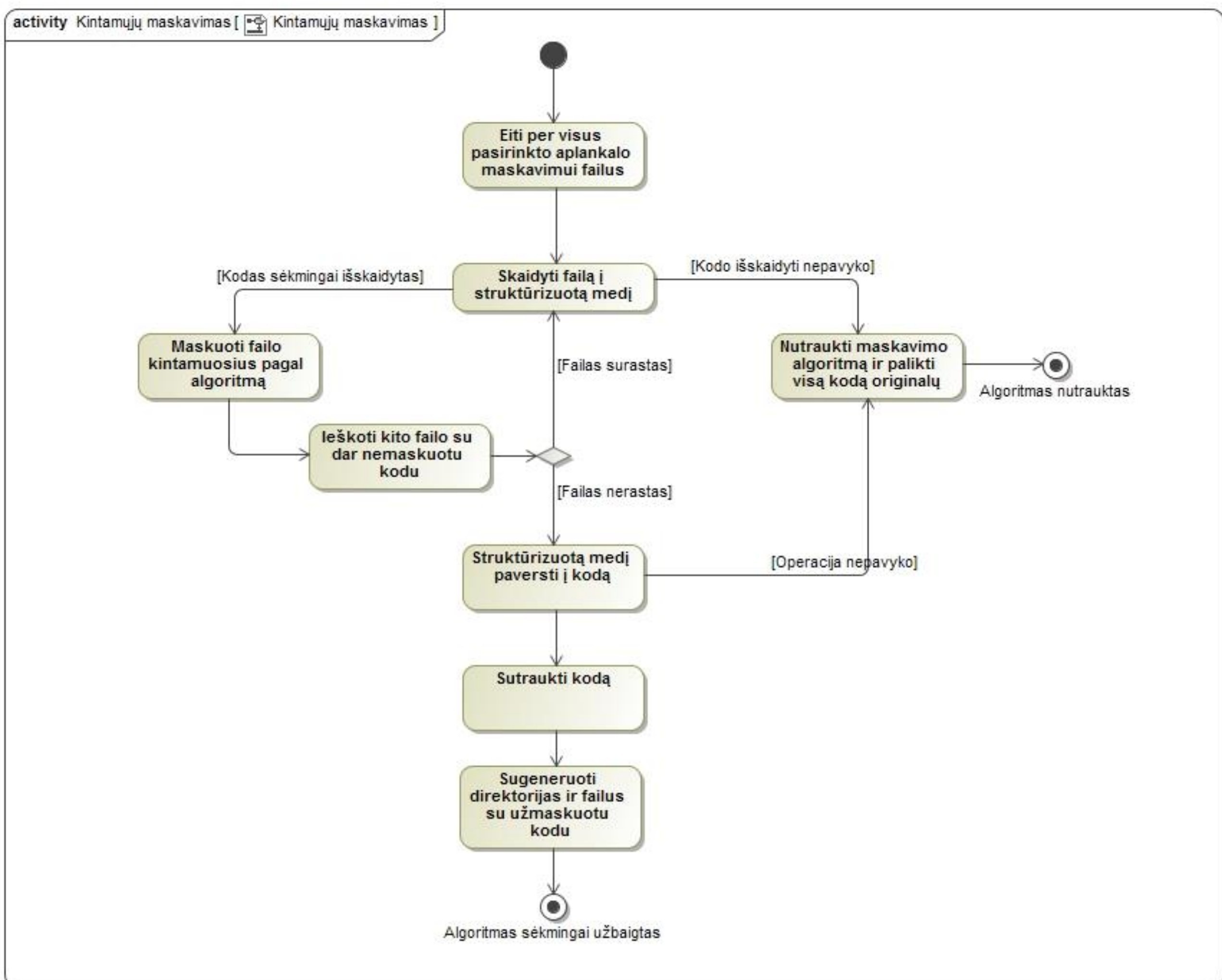
Absoliučiai visoms valdymo struktūroms ir jų atvejams sukurti maskavimo neįmanoma, vien „if“ valdymo struktūroje gali būti naudojama labai daug skirtingų funkcijų bei sąlygų. Tai viena

iš priežasčių, dėl ko geri maskavimo įrankiai kuriami pagal maskuojamą kodą, o ne naudojami universalūs, kurie maskuoja viską ir prieinami visiems.

2.3. Kintamųjų ir funkcijų maskavimas

Kintamųjų maskavimo algoritmas jau yra parašytas „Naneu PHP obfuscator“ įrankyje. Šio algoritmo funkcionalumas, kuriame kodas yra išskaidomas į struktūrizuotą medį ir paverčiamas iš struktūrizuoto medžio į kodą, kuris taip pat sutraukiamas, bus panaudotas valdymo struktūrų maskavimo algoritme.

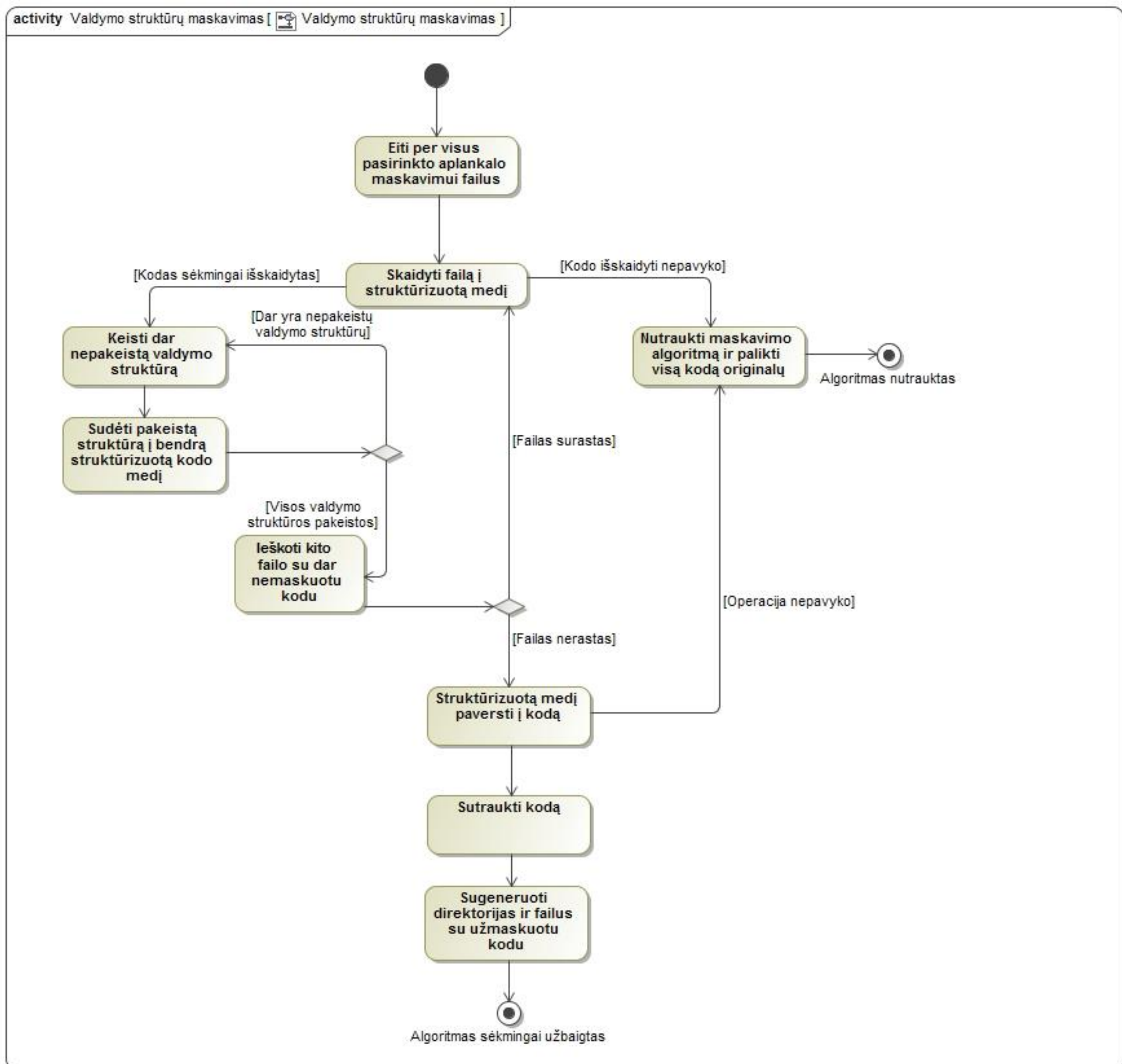
Įrankyje taip pat yra klaidų kurios maskuojant sugadina kodą. Tai dar viena iš priežasčių pilnai nepasitikėti jau sukurtais įrankiais, o kurti savo pagal poreikius. Jei įrankis atvirojo kodo, kaip šiuo atveju tokį funkcionalumą galima pakoreguoti, tačiau jei įrankis komercinis, apie tokias klaidas reikėtų pranešti kūrėjams, laukti pataisymo ir tikėtis, jog daugiau tokių atvejų nepasitaikys. Kaip šiame įrankyje kintamųjų maskavimas vyksta pavaizduota žemiau esančioje veiklos diagramoje.



2.1. pav. Kintamųjų maskavimo algoritmo veiklos diagrama

2.4. Valdymo struktūrų maskavimas

Valdymo struktūrų maskavimui, kaip ir kintamųjų maskavimui, pirmiausia išskaidomas kodas į struktūrizuotą medį. Su struktūrizuotu medžiu iš eilės atliekamas skirtingų valdymo struktūrų maskavimas. Pakeitus valdymo struktūrą, visi pakeitimai sudedami į struktūrizuotą medį ir atliekamas kitos valdymo struktūros maskavimas. Šis ciklas vykdomas tol, kol pakeičiamos visos valdymo struktūros, kurioms yra parašyti maskavimo algoritmai. Žemiau pateiktoje veiklos diagramoje pavaizduotas bendras valdymo struktūrų maskavimo algoritmas.



2.2. pav. Valdymo struktūrų maskavimo algoritmo veiklos diagrama

Algoritmas

Bendras maskavimas vyksta taip:

- einama per visus aplankalus ir failus;
- kodas esantis failuose išskaidomas į struktūrizuotą medį;

- einama per išskaidytą kodo struktūrizuotą medį ir tikrinama, ar yra valdymo struktūra;
- atliekami vidiniai maskavimo algoritmai atitinkamai pagal valdymo struktūros tipą ir einama į gylį, kol užmaskuojamos visos valdymo struktūros faile;
- struktūrizuotas medis paverčiamas atgal į kodą;
- kodas sutraukiamas;
- sugeneruojami falai ir direktorijos iš esamų struktūrizuotų medžių.

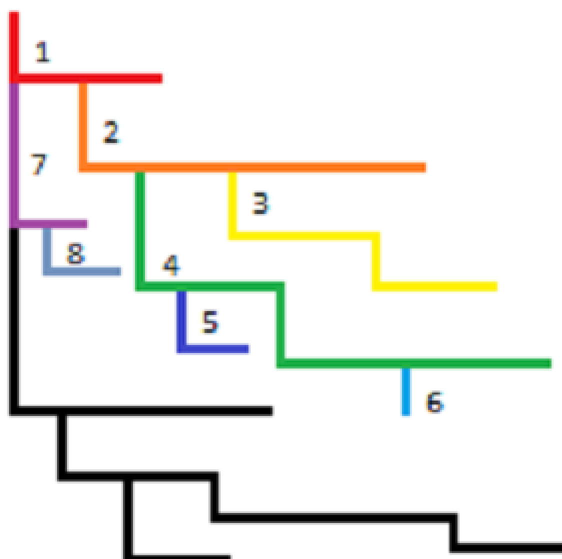
Gilesni sluoksniai

Valdymo struktūrose esantis kodas gali turėti ir daugiau viduje esančių valdymo struktūrų, tad reikia vykdyti maskavimą tol, kol yra valdymo struktūrų į gylį. Dėl to maskavimas turi vykti dinamiškai nepriklausomai nuo kodo gylio.

Tinkamiausias algoritmas tokiam kodui yra rekursinis:

- Surasti pirmojo sluoksnio visas valdymo struktūras;
- Eiti per pirmąją valdymo struktūrą ir joje surasti visas valdymo struktūras;
- Vėl eiti per pirmąją valdymo struktūrą ir taip kol pasiekiama giliausia struktūrizuoto kodo medžio dalis;
- Eiti per vieną žingsnį atgal ir sekančioje valdymo struktūroje eiti vėl gilyn, kai pasiekiama giliausia vieta, algoritmą vėl kartoti.

Einant per medžio elementus tuo pačiu metu reikia atlikti ir maskavimo veiksmus, bei iškart įterpti maskuotą kodo struktūrą atgal į kodo struktūros medį. Vizualiai žingsnių eiliškumas atrodo taip (kiekviena linija yra valdymo struktūra):



2.3. pav. Valdymo struktūrų ėjimas į gylį

2.5. Projektavimo išvados

Projektuojant svarbu pasirinkti tinkamą algoritmą valdymo struktūrų paieškai. Visas algoritmas turi veikti nuosekliai, ir negali būti praleista nei viena valdymo struktūra.

Maskavimo algoritmo sudėtingumas projektuojant nėra prioritetas. Sukūrus įrankį maskavimui su tvarkinga struktūra, nesunkiai galima keisti maskavimo algoritmus atsižvelgiant į kodą kurį reikia užmaskuoti.

Projektuojant nuspėti visų galimų valdymo struktūrų atvejų beveik neįmanoma, todėl reikėtų parašyti algoritmus tik pagrindiniams atvejams ir vėliau testavimo metu aptikus nestandartinius valdymo struktūrų panaudojimo atvejus, maskavimo algoritmus papildyti išimtimis. Kuriant tokį įrankį svarbu, kad vėliau nebūtų sunku pridėti papildomas funkcijas ar išimtis.

3. .ATLIKTI „NANEU PHP OBFUSCATOR“ ĮRANKIO PAKEITIMAI

Šis atvirojo kodo maskavimo įrankis buvo patobulintas taip, kad gebėtų maskuoti dalį valdymo struktūrų. Buvo panaudoti esami algoritmai kodo išskaidymui į struktūrizuotą medį ir pavertimui atgal į kodą. Patobulintas įrankis geba maskuoti „if“ valdymo struktūras. Beabejo „if“ valdymo struktūroje galima naudoti įvairiausias funkcijas, ir visos jos tikrai nėra maskuojamos. Įrankis buvo patobulintas taip, kad radus algoritmuose neaprašytą atvejį, jo tiesiog nemaskuotų, taip apsisaugant nuo klaidų maskavimo metu.

3.1. Valdymo struktūrų maskavimo algoritmai

„if“ valdymo struktūra yra keičiama į „switch“ valdymo struktūrą. Kodo pavyzdys PHP programavimo kalboje:

3.1. lentelė. „if“ ir „switch“ valdymo struktūrų palyginimas

<table>
<tr><th>„if“ valdymo struktūra</th><th>„switch“ valdymo struktūra</th></tr>
<tr><td><pre>if ($var == 5) {
   $var2 = 6;
}</pre></td><td><pre>switch ($var) {
   $var2 = 6;
   break;
}</pre></td></tr>
</table>

Norint užmaskuoti kodą keičiant valdymo struktūras, vien pakeisti valdymo struktūrą neužtenka, nes kodas įsilaužėliui bei statinės analizės įrankiams lieka gerai suprantamas. Siekiant užmaskuoti kodą keičiant valdymo struktūras, turi būti pridėta perteklinių algoritmų bei duomenų. Kuo daugiau perteklinių algoritmų bei duomenų bus pridedama, tuo sudėtingesnis bus kodo skaitymas. Tačiau nereikia pamiršti, kad neteisingai valdant perteklines valdymo struktūras, nukentės greitaveika. Norint apsisaugoti nuo statinės analizės įrankių, vienas iš svarbiausių kriterijų yra keisti originalaus kodo valdymo struktūrų lygmenis. Originalų kodą su elementariomis valdymo struktūromis galime matyti 2.2. lentelėje.

3.2. lentelė. Originali „if“ valdymo struktūra

„if“ valdymo struktūra (originalus kodas)

```
$var = "test";
$var2 = "cba";
if ($var === "test") {
   if ($var2 == "abc") {
      echo "abc";
      if (1 == 1) {
        echo "useless statement";
      }
   } else {
       echo "not abc";
   }
} elseif ($var === "live") {
   echo "live";
} else { die(); }
```

3.3. lentelė. Užmaskuota „if“ valdymo struktūra

„if“ valdymo struktūra (užmaskuotas kodas)

```
$var = 'test';
$var2 = 'cba';
if (1) {
    $nm5729ba1910ada = 'df5729ba1910ad4';
    $nm5729ba1910ae1 = 'rn5729ba1910ad1';
    while ($nm5729ba1910ae1 != '5729ba1910ace') {
        switch ($var) {
            case 'test':
                if (1) {
                    $nm5729ba191166f = 'df5729ba1911669';
                    $nm5729ba1911676 = 'rn5729ba1911667';
                    while ($nm5729ba1911676 != '5729ba1911663') {
                        switch ($var2) {
                            default:
                                $nm5729ba1911676 = '5729ba1911663';
                                break;
                            case 'abc':
                                echo 'abc';
                                if (1 == 1) {
                                    echo 'useless statement';
                                }
                                $nm5729ba191166f = 'as5729ba191166c';
                                break 2;
                        }
                    }
                    while ($nm5729ba191166f != 'as5729ba191166c') {
                        echo 'not abc';
                        $nm5729ba191166f = 'as5729ba191166c';
                    }
                }
                $nm5729ba1910ada = 'as5729ba1910ad7';
                break 2;
            default:
                $nm5729ba1910ae1 = '5729ba1910ace';
                break;
        }
        $nm5729ba1910ffa = 'rn5729ba1910ad1';
        while ($nm5729ba1910ffa != '5729ba1910ace') {
            switch ($var) {
                case 'live':
                    echo 'live';
                    $nm5729ba1910ffa = '5729ba1910ace';
                    $nm5729ba1910ae1 = '5729ba1910ace';
                    $nm5729ba1910ada = 'as5729ba1910ad7';
                    break 3;
                default:
                    $nm5729ba1910ffa = '5729ba1910ace';
                    $nm5729ba1910ae1 = '5729ba1910ace';
                    break;
            }
        }
    }
    while ($nm5729ba1910ada != 'as5729ba1910ad7') {
        die;
        $nm5729ba1910ada = 'as5729ba1910ad7';
    }}
```

Užmaskuotas kodas matomas 2.3. lentelėje. Užmaskuotame kode yra pakeisti valdymo struktūrų lygmenys. „elseif“ valdymo struktūros yra perkeltos giliau. Tokį kodą išanalizavus su statinės analizės įrankiu, jis neparodys tikrosios kodo struktūros. Tikros kodo struktūros neišgavus su įrankiais, tenka kodą analizuoti pačiam įsilaužėliui. Jei kodas yra pakankamai didelis ir sudarytas iš daug failų, kuriuose yra daug valdymo struktūrų, užmaskuotą kodą pilnai suprasti įsilaužėliui gali užtrukti per ilgai. Kodo išgavimo kaštai galimai taps didesni nei nauda išgavus jį. Štai taip atrodo pilnai užmaskuotas kodas naudojant ir valdymo struktūrų maskavimą, ir kintamųjų, ir sutraukimo algoritmus:

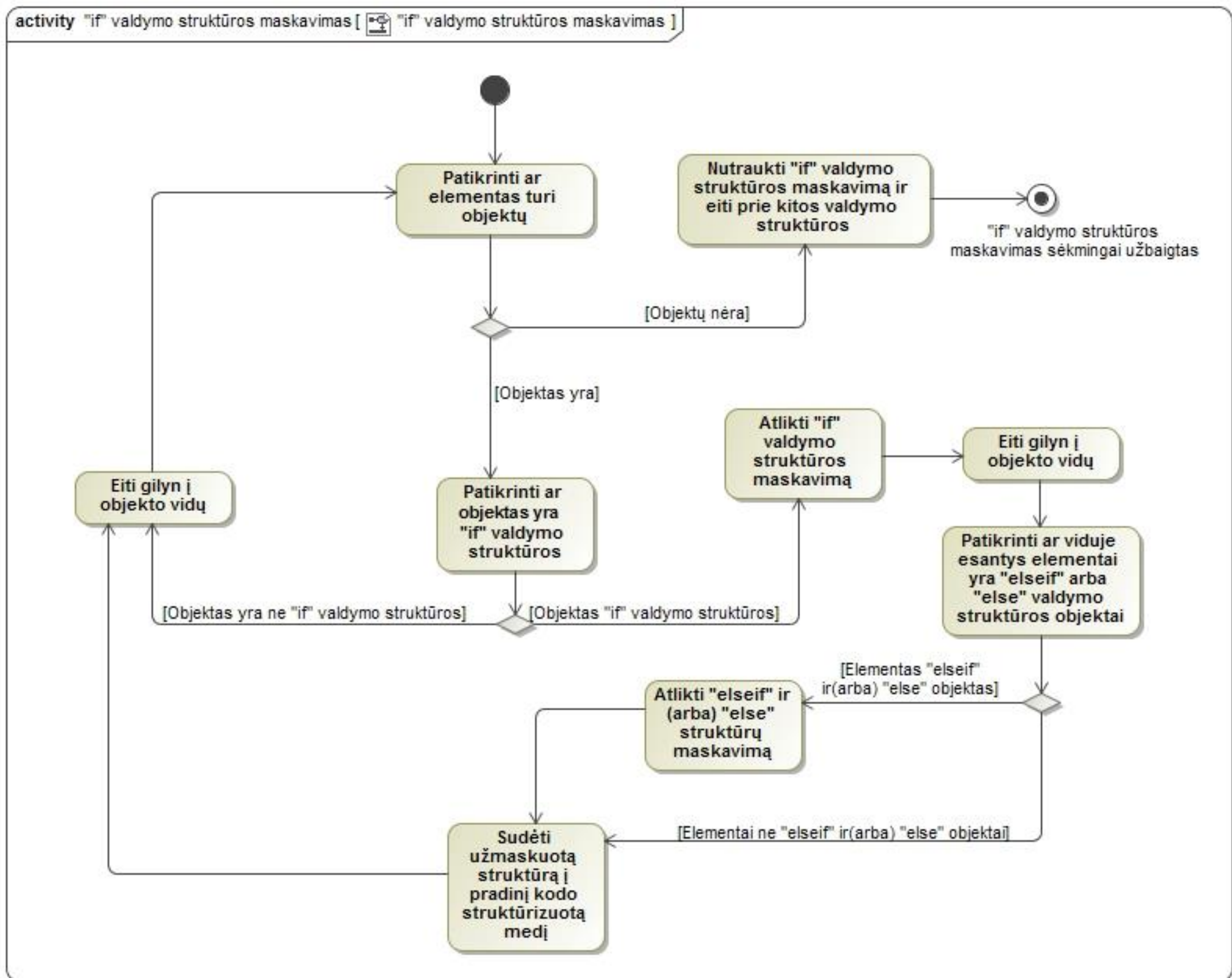
```
      $spa5991e = 'test'; $spaffe7c = 'cba'; if (1) { $spf9d3dc =
'df5729ba4722dac'; $sp76d6a8 = 'rn5729ba4722da8'; while ($sp76d6a8 !=
'5729ba4722da5') { switch ($spa5991e) { default: $sp76d6a8 =
'5729ba4722da5'; break; case 'test': if (1) { $spf8edf8 =
'df5729ba4723bfb'; $spe2ae4f = 'rn5729ba4723bf9'; while ($spe2ae4f !=
'5729ba4723bf5') { switch ($spaffe7c) { case 'abc': echo 'abc'; if (1 ==
1) { echo 'useless statement'; } $spf8edf8 = 'as5729ba4723bff'; break 2;
default: $spe2ae4f = '5729ba4723bf5'; break; } } while ($spf8edf8 !=
'as5729ba4723bff') { echo 'not abc'; $spf8edf8 = 'as5729ba4723bff'; } }
$spf9d3dc = 'as5729ba4722dae'; break 2; } $sp2da063 = 'rn5729ba4722da8';
while ($sp2da063 != '5729ba4722da5') { switch ($spa5991e) { case 'live':
echo 'live'; $sp2da063 = '5729ba4722da5'; $sp76d6a8 = '5729ba4722da5';
$spf9d3dc = 'as5729ba4722dae'; break 3; default: $sp2da063 =
'5729ba4722da5'; $sp76d6a8 = '5729ba4722da5'; break; } } } while
($spf9d3dc != 'as5729ba4722dae') { die; $spf9d3dc = 'as5729ba4722dae'; }}
```

Šis kodas atliks tokias pat funkcijas kaip ir originalus kodas, pateiktas 2.2 lentelėje, tačiau skaityti jį daug sudėtingiau. Greitaveika šiuo atveju šiek tiek nukentės, jei valdymo struktūrų kiekiai bus dideli. Tačiau maskuojant „if“ valdymo struktūras nėra ciklų, kurie ilgai truktų, tiesiog pridėtiniai „while“ ciklai, kurie vykdomi tik kartą, ir keletas papildomų „if“ sąlygų, kurių vykdymas užtruks šiek tiek ilgiau.

Maskuojant „if“ valdymo struktūrą buvo susikurtos taisyklės, kurios atspindi pagrindinę valdymo struktūrų maskavimo taisyklę – valdymo struktūros turi būti išskaidytos į skirtingus sluoksnius. Sukurtos taisyklės yra:

- “if” valdymo struktūros maskavimas turi vykti,, esant įvairaus tipo parametrams ir turi būti apsuptos perteklinėmis “if” bei “while” valdymo struktūromis;
- “elseif” valdymo struktūros maskavimas turi vykti, esant įvairiaus tipo parametrams ir turi būti gilesniame sluoksnyje nei “if” valdymo struktūra bei turi būti apsuptos perteklinėmis “while” valdymo struktūromis. “elseif” valdymo struktūrą keičiant į “case” sąlygą,, jai turi būti sukuriama atskira “switch” valdymo struktūra;
- “else” valdymo struktūros maskavimas turi būti išoriniame sluoksnyje nuo “if” valdymo struktūros, apsuptas “while” pertekline valdymo struktūra.

Laikantis šių taisyklių buvo sukurtas algoritmas “if” valdymo struktūros maskavimui. Žemiau pateikiama bendra “if”, “elseif” ir “else” valdymo struktūrų maskavimo veiklos diagrama.

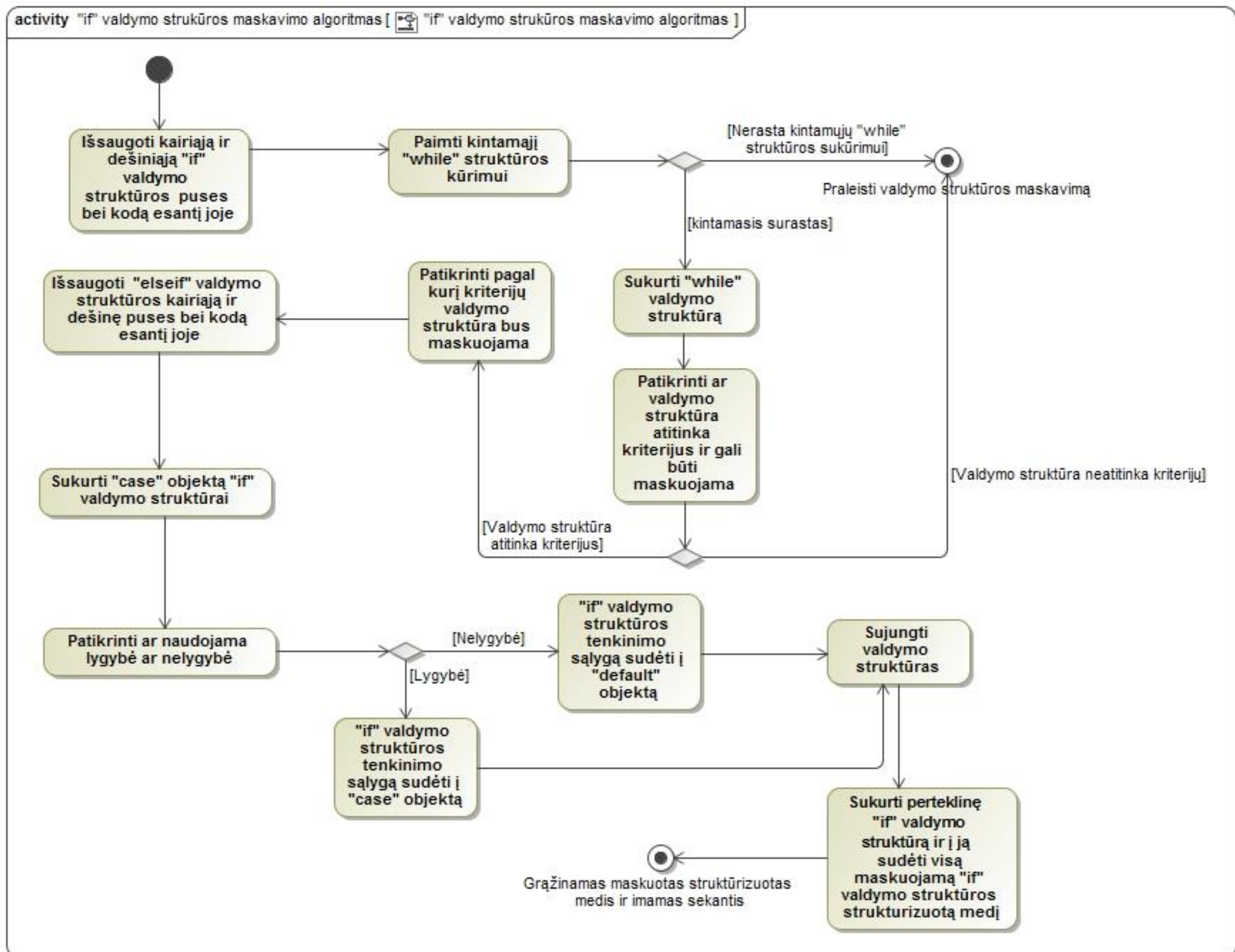


3.1. pav. Bendra „if“ valdymo struktūros maskavimo veiklos diagrama

„if“ valdymo struktūros maskavimo algoritmas susideda iš 5 pagrindinių žingsnių:

1. Išsaugomos kairioji ir dešinioji „if“ ir „elseif“ valdymo struktūrų sąlygos dalys bei kodas esantis „if“, „elseif“, „else“ valdymo struktūrose;
2. Sukuriami „case“ objektai iš išsaugotų reikšmių. Kuriant „case“ objektus kartais naudojama ir „default“ sąlyga „if“ valdymo struktūrai, taip yra didesnė tikimybė jog bus apsisaugojama nuo automatinių įrankių skirtų atpainioti kodą, kurie tikrina sąlygos reikšmes. „default“ sąlyga taip pat reikalinga, kai yra nelygybės sąlyga.
3. Sukuriamas „switch“ valdymo struktūros objektas ir į jį sudedami „case“ objektai
4. Pridedamos „if“ ir „while“ valdymo struktūros siekiant pakeisti maskuojamų valdymo strūktūrų lygmenis ir išdėstymą.
5. Viskas sudedama atgal į struktūrizuotą medį.

Nors ir visos 3 („if“, „elseif“ ir „else“) valdymo struktūros yra susiję su viena – „if“ valdymo struktūra, maskavimas joms rašomas atskirai. Sekančiame paveiksle pateikiama detali „if“ valdymo struktūros veiklos diagrama.



3.2. pav. Detali „if“ valdymo struktūros maskavimo algoritmo veiklos diagrama

Maskuojant „if“ valdymo struktūrą algoritmas yra izoliuotas nuo „elseif“ valdymo struktūros maskavimo algoritmo. Užmaskuoti kodo struktūrizuoto medžio elementai sujungiami tik pačiame gale prieš kuriant perteklinę „if“ valdymo struktūrą, kuri viską apgaubia. Maskavimo algoritmas tiesiogiai priklauso nuo to, ar yra maskuojamoje valdymo struktūroje „elseif“ valdymo struktūra, ar nėra, ar yra lygybė, ar nelygybė. Algoritmas taip pat priklauso nuo duomenų bei kintamųjų tipų esančių „if“ valdymo struktūroje.

Pavyzdinės galimos skirtingos „if“ valdymo struktūros sąlygos:

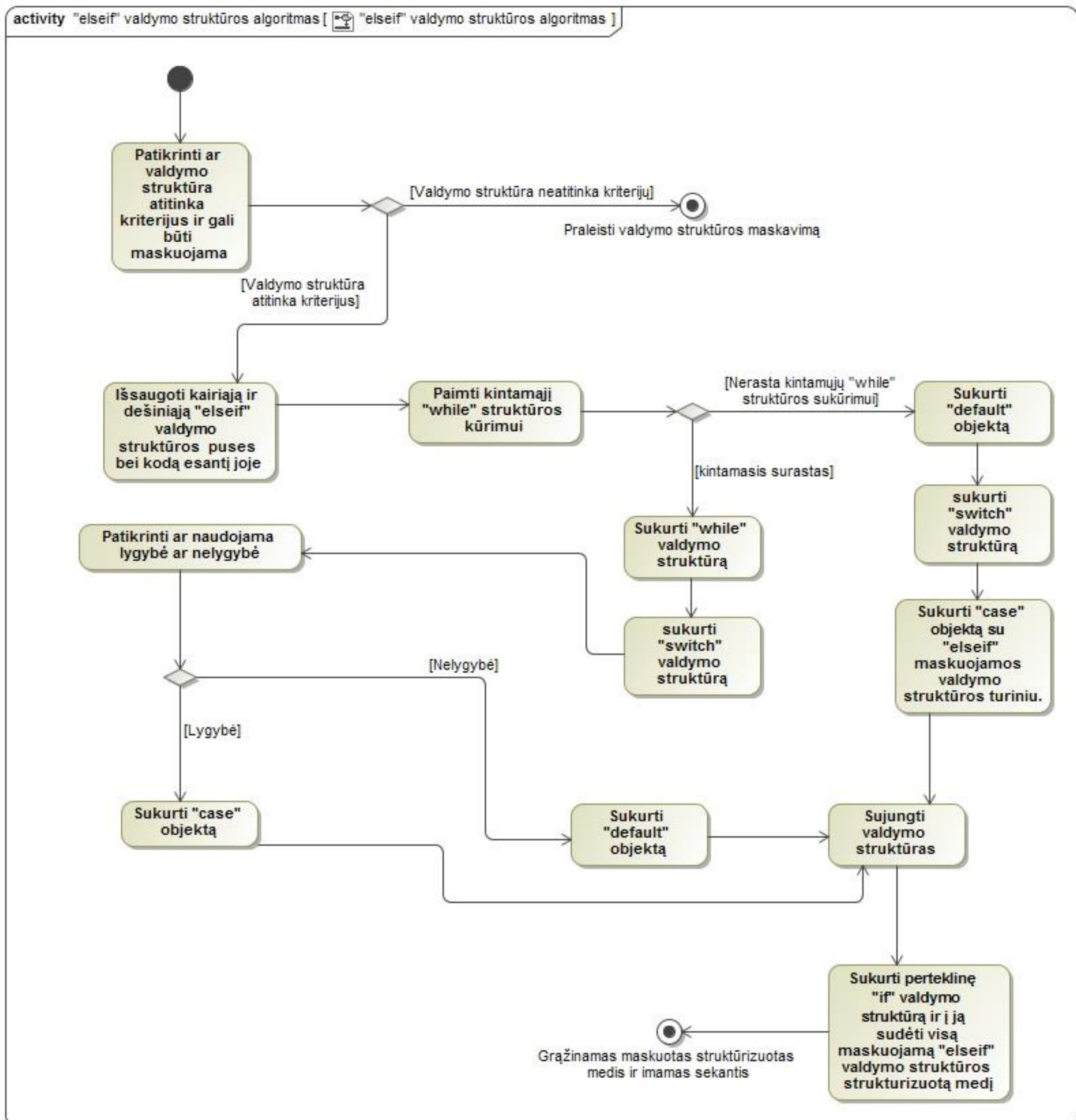
3.2. lentelė. „if“ ir „elseif“ valdymo struktūros sąlygos

`if ($var == $var2)`
`if ($var != $var2)`
`if ($var == 0)`
`if ($var == 'test')`
`if ($var == false)`
`if ($var != false)`

if (false)
if (5 == 5)

Visais išvardintais atvejais programa maskuodama kodą elgsis skirtingai. Bus parenkamas tinkamas maskavimo algoritmas, kad užmaskuotas kodas būtų veikiantis ir veiktų taip pat kaip originalus. Visi kiti atvejai, kuriuose sąlygose yra funkcijos, yra praleidžiami ir nemaskuojami.

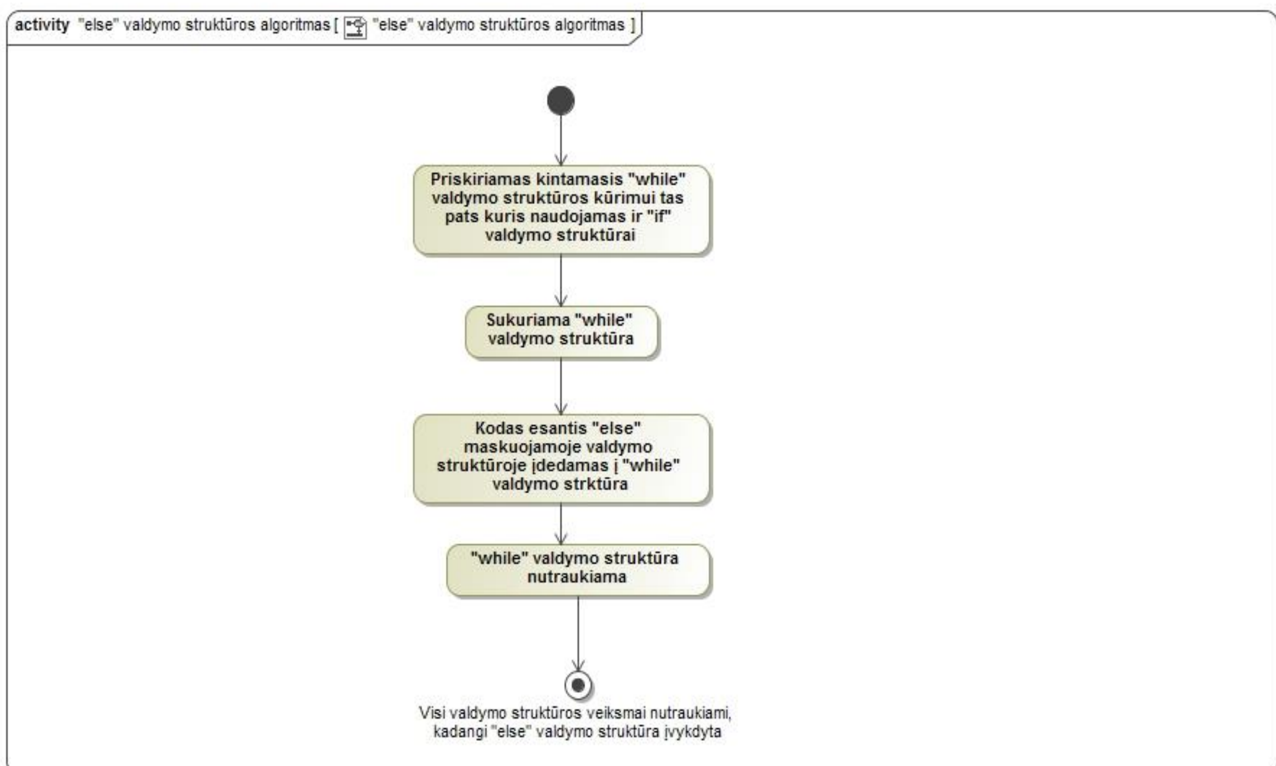
Maskuojant „elseif" valdymo struktūrą, algoritmo pradžia ir pabaiga išlieka panaši, kaip ir maskuojant „if" valdymo struktūra. Išsaugomos yra kairė ir dešinė sąlygos pusės bei gale visos užmaskuotos ir sukurtos perteklinės valdymo struktūros sujungiamos. Detali „elseif" valdymo struktūros schema pateikiama žemiau esančiame paveiksle.



3.3. pav. Detali „elseif" valdymo struktūros maskavimo algoritmo veiklos diagrama

Maskuojant „elseif“ valdymo struktūrą, sąlygos yra tokios pat kaip ir „if“ valdymo struktūrai. „elseif“ sąlygos pavaizduotos 3.2 lentelėje. Skirtumas nuo „if“ valdymo struktūros maskavimo toks, kad, jei nerandamas kintamasis „while“ valdymo struktūros sukūrimui, elementas vis tiek maskuojamas be perteklines „while“ valdymo struktūros.

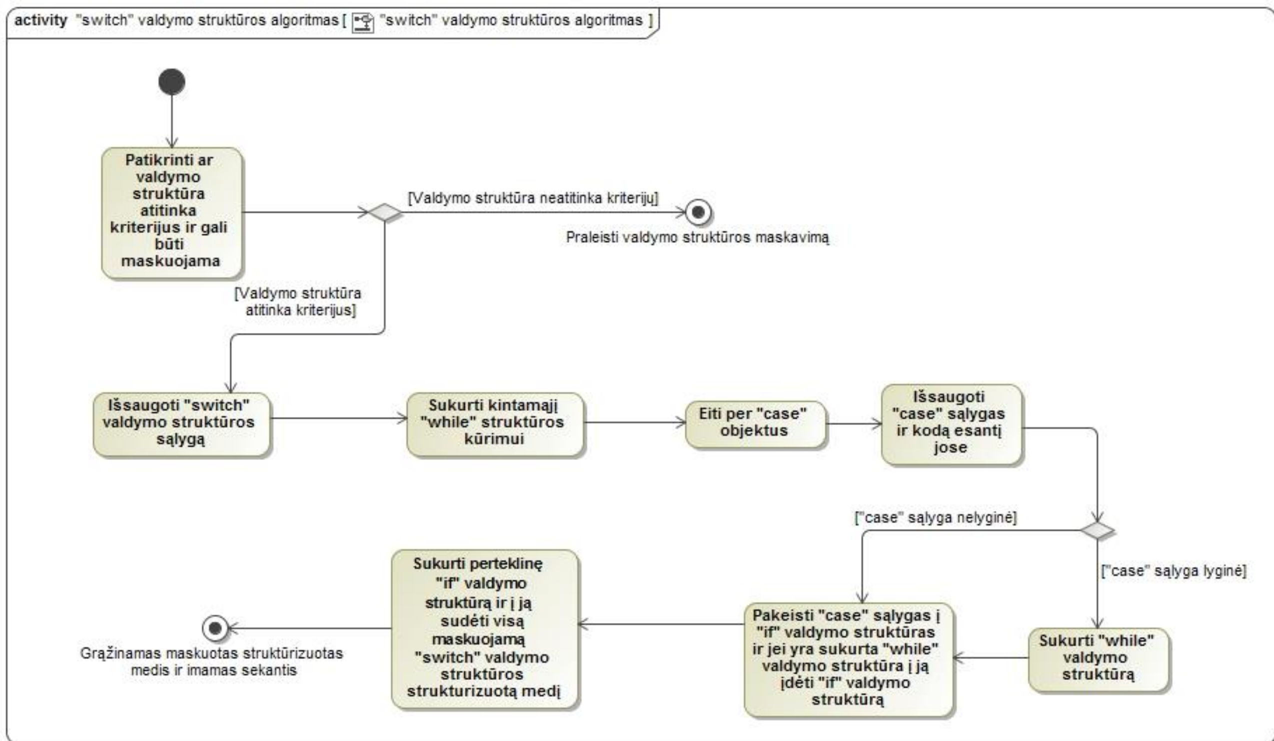
Valdymo struktūros „else“ maskavimui vykdomi kur kas paprastesni veiksmai. Šiai valdymo struktūrai sukuriama „while“ valdymo struktūra, į kurią ir sudedamas visas kodas buvęs „else“ valdymo struktūroje. Tuomet „while“ valdymo struktūra pridedama prie pagrindinės, perteklinės „while“ valdymo struktūros. Veiksmų seka pavaizduota 3.4 paveiksle.



3.4. pav. Detali „else“ valdymo struktūros maskavimo algoritmo veiklos diagrama

Valdymo struktūros „switch“ maskavimas šiek tiek panašus į „elseif“. Kaip ir „elseif“ valdymo struktūros, taip ir „switch“ valdymo struktūros „case“ sąlygos yra išmėtomos po skirtingus lygmenis. Skirtumas tik tas, kad „elseif“ valdymo struktūros maskuojamos po vieną, o „case“ yra sąlygos kurios priklauso „switch“ valdymo struktūrai, todėl yra užmaskuojamos vienos iteracijos metu.

„switch“ valdymo struktūras maskuoti yra lengviau nei „if“ ar „elseif“, nes jose rečiau naudojamos įvairios funkcijos. Dažniausiai šio tipo valdymo struktūrose naudojami tiesiog kintamieji, o jų sąlygose „case“ - reikšmės. „switch“ valdymo struktūros veiklos diagrama pavaizduota 3.5 paveiksle.



3.5. pav. Detali „switch“ valdymo struktūros maskavimo algoritmo veiklos diagrama

Išskaidyto kodo struktūrinis medis

Žemiau pateiktame pavyzdyje matoma išskaidytos valdymo struktūros medžio dalis:

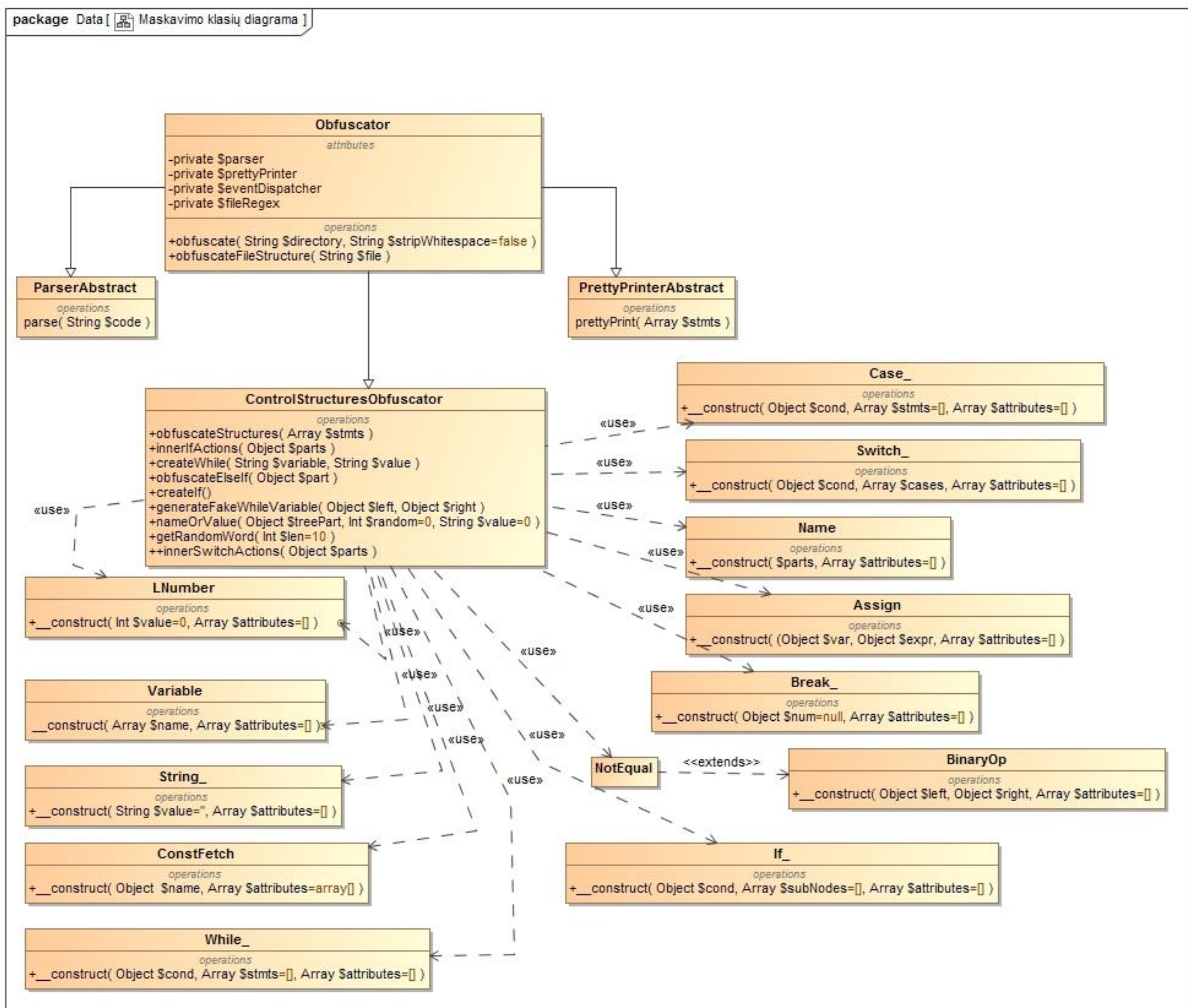
```
object(PhpParser\Node\Stmt\If_)[44]
  public 'cond' =>
    object(PhpParser\Node\Expr\BinaryOp\Identical)[15]
      public 'left' =>
        object(PhpParser\Node\Expr\Variable)[13]
          public 'name' => string 'a' (length=1)
          protected 'attributes' =>
            array (size=2)
              ...
      public 'right' =>
        object(PhpParser\Node\Scalar\String_)[14]
          public 'value' => string 'test' (length=4)
          protected 'attributes' =>
            array (size=2)
              ...
      protected 'attributes' =>
        array (size=2)
          'startLine' => int 3
          'endLine' => int 3
  public 'stmts' =>
              ...
  public 'elseifs' =>
    array (size=2)
              ...
  public 'else' =>
            ...
```

Šiame medyje yra išskaidytos "if" valdymo struktūros kodas. `PhpParser\Node\Stmt\If_`" yra "if" valdymo struktūros objekto klasė, viduje yra "`PhpParser\Node\Expr\BinaryOp\Identical`" objektas, kuris reiškia palyginimo su tipu sąlygą (===), "`left`" yra kairioji sąlygos dalis, o "`right`" dešinioji. Elementas "`stmts`" yra kodas, kuris vykdomas įvykdžius sąlygą.

Tokiu principu yra išskaidomas absoliučiai visas kodas. Maskuojant elementai bei objektai keičiami vienas su kitu, pridedami papildomi pertekliniai objektai bei duomenys, ir keičiama jų tvarka. Galiausiai visas pakeistas struktūrizuotas medis "PrettyPrinter" bibliotekos pagalba paverčiamas atgal į veikiantį kodą.

3.2. Klasių diagrama

Klasių diagramoje pavaizduotos tik klasės, kurios buvo panaudotos valdymo struktūrų maskavimui. Pats įrankis yra labai didelis, todėl visos klasės nesusiję su kurtu maskavimo algoritmu, schemoje neatvaizduotos.



3.6. pav. Valdymo struktūrų maskavimo įrankio klasių diagrama

Per komandinę eilutę iškviečiamas „Obfuscator“ objektas. Kodas iš failų į struktūrizuotą medį paverčiamas „ParseAbstract“ objekto pagalba, o po visų maskavimo veiksmų iš medžio į kodą paverčiamas „PrettyPrinterAbstract“ klasėje. Pagrindiniai maskavimo veiksmai atliekami “ControlStructuresObfuscation” klasėje, kuri naudoja pagalbinius objektus kodo keitimui bei kūrimui. Visi atvaizduoti objektai turintys tik konstruktorių naudojami sukurti kodui struktūrizuoto medžio pavidale.

3.3. Realizacijos išvados

Kuriant šį maskavimo įrankį dažniausiai buvo susiduriama su kodo veikimo iškraipymu, kai užmaskuotas kodas veikia kitaip nei originalus. Galiausiai pavyko sukurti sistemą, kuri maskuoja „if“, „elseif“ ir „else“ valdymo struktūras nepakeičiant kodo veikimo. Sistema maskuodama kodą pakeičia valdymo struktūrų lygmenis ir prideda papildomas, perteklines valdymo struktūras.

Sistema sukurta taip, kad pridėti maskavimą papildomoms valdymo struktūroms yra labai nesunku. Norint pridėti papildomus maskavimo algoritmus skirtingoms valdymo struktūroms, reikia parašyti algoritmą ir pridėti metodą, kuris maskuoja į „*obfuscateStructures()*“ metodą. Kodo išskaidymas į struktūrizuotą medį, ėjimas į gylį per visas valdymo struktūras, struktūrizuoto medžio pavertimas į kodą, bei failų generavimas veiks iš karto su naujai parašytais algoritmais.

Sudėtingiausia dalis, kuriant algoritmus valdymo struktūrų maskavimui, yra užtikrinimas, kad užmaskavus kodą, jis veiks taip, kaip ir originalus. Tam reikia atlikti daug testų su įvairiais skirtingais valdymo struktūrų išdėstymo būdais. Vien pasikeitus palyginimo operatoriui ar kintamojo tipui ,reikia rašyti atskirą maskavimo algoritmą.

4. GREITAVEIKOS IR KODO VEIKIMO TYRIMAS

Panaudojus maskavimo įrankį, labai svarbu ištestuoti užmaskuotą kodą, ar jis veikia taip, kaip turėtų veikti. Mažas maskavimo įrankių universalumas yra viena iš priežasčių kurti savo maskavimo įrankį, kuris maskuos pagal specialiai sukurtus algoritmus. Maskavimo algoritmai turi tinkamai veikti su maskuojamu kodu. Patikrinus kodo veikimą, buvo surasta atvejų, kai užmaskuotas kodas veikia kitaip nei turėtų arba išvis neveikia.

Maskuojant valdymo struktūras, gali stipriai nukentėti greitaveika. Maskuojant ciklinę valdymo struktūrą padidinant ciklų arba duomenų skaičių, programa gali sulėtėti tiek, kad nebeatitiks reikalavimų. Tai dar viena iš priežasčių kurti individualius maskavimo algoritmus, su kuriais užmaskavus programinę įrangą, ji vis dar tenkins reikalavimus.

4.1. Kodo veikimo testavimas

Valdymo struktūrų maskavimas šio darbo metu sukurtais algoritmais buvo ištestuotas su:

- Projektu, kurį sudaro 91 failas su kodu ir konfigūracijomis. Užmaskuoti failai vidiniame serveryje (angl. *backend server*);
- Failu kuriame daug valdymo struktūrų;
- Visa maskavimo programa kuri buvo tobulinta šio darbo metu ir jos bibliotekomis.

4.1. lentelė. Pirmas valdymo struktūrų maskavimo testavimas su asmeniniu projektu.

Testuojama funkcija	Testavimo atvejis	Laukiami rezultatai	Gauti rezultatai
Prisijungimas prie sistemos.	Prisijungiama suvedus teisingus prisijungimo duomenis, kurie siunčiami į vidinį serverį ir gražinamas atsakymas į išorinį serverį.	Sėkmingas prisijungimas, rodomi vartotojo duomenys.	Prisijungta sėkmingai
Paieška pagal parametrus.	Nueinama į paieškos langą, kuriame pasirenkami paieškos parametrai.	Paieškos parametrai parsiunčiami iš vidinio serverio į išorinį ir atvaizduojami vartotojui.	Paieškos parametrų parsiųsti nepavyksta, gaunama klaida.

Gavus klaidą parsiunčiant paieškos parametrus iš vidinio serverio, buvo rasta klaida valdymo struktūroje, kurioje tikrinama ar kintamasis nelygus nuliui: `if ($var != 0)`. Tokiu atveju kodas buvo maskuojamas į:

```
switch ($var) {
    case $var !== 0:
        //success
        break;
    …
}
```

Tokiu atveju gaunama sąlyga:

```
if ($var == ($var !== 0))
```

Gavus tokią sąlygą, kodas suveikia nekorektiškai. Algoritmai buvo pakeisti, kad tikrinant sąlygą su 0, užmaskuotas kodas būtų pakeičiamas į:

```
switch ($var) {
   default:
      //success
      break;
   case '0':
      break;
   …
}
```

4.2. lentelė. Antras valdymo struktūrų maskavimo testavimas su asmeniniu projektu.

Testuojama funkcija	Testavimo atvejis	Laukiami rezultatai	Gauti rezultatai
Prisijungimas prie sistemos.	Prisijungiama suvedus teisingus prisijungimo duomenis, kurie siunčiami į vidinį serverį ir gražinamas atsakymas į išorinį serverį.	Sėkmingas prisijungimas, rodomi vartotojo duomenys.	Prisijungta sėkmingai.
Paieška pagal parametrus.	Nueinama į paieškos langą, kuriame pasirenkami paieškos parametrai.	Paieškos parametrai parsiunčiami iš vidinio serverio į išorinį ir atvaizduojami vartotojui.	Paieškos parametrai parsiunčiami sėkmingai.
Skelbimų paieška.	Pasirinkus paieškos parametrus, juos išsaugome duomenų bazėje, paspausdami	Paieška sukurta, vartotojas nukeltas į paieškų langą, vidinis serveris grąžina	Paieška nesukuriama, gaunama vidinio serverio klaida.

	sukurti paiešką.	sukurtos paieškos duomenis ir atsakymą jog viskas gerai.	

Gavus klaidą iš serverio be jokio sutartinio atsakymo, buvo surasta jog maskuojant valdymo struktūrą kurioje yra sąlyga:

```
if ($var === false) {
        …
```

Gaunamas neteisingai užmaskuotas kodas:

```
switch ($var === $false) {
        …
```

Taip nutiko dėl to nes lyginant „boolean“ tipo reikšmę, išskaidžius kodą į medį nepriskiriamas joks tipas, ir reikšmė tolesniuose algoritmuose naudojama kaip kintamasis (tipai išskaidytame medyje gali būti: kintamasis, reikšmė, skaičius...). Radus „false“ arba „true“ buvo panaikintas tipo tikrinimas ir į medį įterpiama tik reikšmė.

4.3. lentelė. Trečias valdymo struktūrų maskavimo testavimas su asmeniniu projektu.

Testuojama funkcija	Testavimo atvejis	Laukiami rezultatai	Gauti rezultatai
Prisijungimas prie sistemos.	Prisijungiama suvedus teisingus prisijungimo duomenis, kurie siunčiami į vidinį serverį ir gražinamas atsakymas į išorinį serverį.	Sėkmingas prisijungimas, rodomi vartotojo duomenys.	Prisijungta sėkmingai.
Paieška pagal parametrus.	Nueinama į paieškos langą, kuriame pasirenkami paieškos parametrai.	Paieškos parametrai parsiunčiami iš vidinio serverio į išorinį ir atvaizduojami vartotojui.	Paieškos parametrai parsiunčiami sėkmingai.
Skelbimų paieškos sukūrimas.	Pasirinkus paieškos parametrus, juos išsaugome duomenų bazėje, paspausdami sukurti paiešką.	Paieška sukurta, vartotojas nukeltas į paieškų langą, vidinis serveris grąžina sukurtos paieškos duomenis ir atsakymą jog viskas gerai.	Grąžinti sukurtos paieškos duomenys, vidinis serveris grąžino atsakymą jog viskas gerai, vartotojas nukeltas į paieškų langą.

Skelbimų paieškos algoritmai vidiniame serveryje.	Sukūrus paiešką tam tikri duomenys apie skelbimus turi atsirasti duomenų bazėje.	Duomenys atsiranda periodiškai duomenų bazėje.	Duomenys atsiranda periodiškai duomenų bazėje.

Kadangi pagrindinis, sudėtingiausias programos algoritmas, kuris paima skelbimų duomenis, juos sudeda, apdoroja ir grąžina vartotojui - veikia, galima daryti prielaidą jog visos smulkesnės programos funkcijos veikia tai pat. Šiuose algoritmuose yra daugiausia valdymo struktūrų iš visos programos ir jos visos veikia teisingai su užmaskuotu kodu.

Testas su failu, kuriame daug valdymo struktūrų

Testas atliekamas su failu kuriame yra:

- Didžiausias valdymo struktūrų gylis - 10;
- Kode naudojamos "if", "elseif", "else" ir "while" valdymo struktūros;
- Skirtingi duomenų tipai valdymo struktūrose;
- 10 skirtingų vietų kur išvedamas tekstas į ekraną.

4.4. lentelė. Kodo veikimo testas su failu.

Testuojamas funkcionalumas	Nemaskuoto kodo vykdymo rezultatai (išvedimas į ekraną)	Maskuoto kodo vykdymo rezultatai
Maskavimas į gylį.	Pasiekta giliausia valdymo struktūrų vieta ir į ekraną išvesta: • „Giliausia vieta“.	Pasiekta giliausia valdymo struktūrų vieta ir į ekraną išvesta: • „Giliausia vieta“.
Valdymo struktūrų maskavimas su kintamaisiais	Į ekraną išvesta: • “elseif“; • „if“; • „else“.	Į ekraną išvesta: • “elseif“; • „if“; • „else“.
Valdymo struktūrų maskavimas be kintamųjų	Į ekraną išvesta: • “if palyginimas be kintamųjų“; • „elseif palyginimas be kintamųjų“; • „bool palyginimas be kintamųjų“.	Į ekraną išvesta: • “if palyginimas be kintamųjų“; • elseif palyginimas be kintamųjų“; • „bool palyginimas be kintamųjų“.
Valdymo struktūrų maskavimas su skirtingais	Į ekraną išvesta:	Į ekraną išvesta:

duomenų tipais ir operatoriais	• “nelygybė“; • „palyginimas su skaičiumi“; • „bool tipo palyginimas“; • „nelygybė“.	• “nelygybė“; • „palyginimas su skaičiumi“; • „bool tipo palyginimas“; • „nelygybė“.

Ištestuoti buvo visi galimi variantai kurie išvardinti 3.2 lentelėje. Kodas buvo išvestas visose vietose kur buvo testuojama, tad visos valdymo struktūros suveikė taip pat kaip ir originaliame, nemaskuotame kode.

Testas su maskavimo įrankiu ir visomis jo bibliotekomis

Šio testo metu buvo užmaskuotas viso įrankio ir jo bibliotekų kodas keičiant valdymo struktūras. Tuomet buvo bandoma naudoti užmaskuotą įrankį kito kodo maskavimui.

4.5. lentelė. Kodo veikimo testas su maskavimo įrankiu.

Testuojama funkcija	Testavimo atvejis	Laukiami rezultatai	Gauti rezultatai
Maskavimo paleidimas per komandinę eilutę	Per komandinę eilutę paleidžiamas maskavimo algoritmas iš užmaskuoto kodo	Komandinėje eilutėje išvedami failų pavadinimai, kurie buvo sėkmingai užmaskuoti	Komandinėje eilutėje išvesti failų pavadinimai, kurie buvo sėkmingai užmaskuoti
Užmaskuoto kodo veikimas	Paleidžiamas vykdymui kodas, kuris buvo užmaskuotas su užmaskuotu įrankiu	Į ekraną išvedama „Veikia“	Į ekraną išvesta „Veikia“

Užmaskuotame įrankyje yra 1086 failai. Be abejo, visi failai nebuvo maskuojami, nes yra konfigūracinių failų bei failų, kuriuose nėra “if”, “elseif”, “else” valdymo struktūrų su 3.2 lentelėje išvardintomis sąlygomis. Tačiau užmaskuotas įrankis sėkmingai veikia. Kodo pavyzdys iš šio darbo metu rašyto algoritmo:

Originalus kodas:

```
if ($key === 'elseifs') {
   $this->obfuscateElseIf($part);
   //Write else turned into case into array
} elseif ($key === 'else') {
   //Get needed values for further actions (else)
   $this->variableWhileElement = $this->firstFakeVariableElement;
   $elseWhile = $this->createWhile(
                       $this->forElseVariable,
                       $this->forElseValue
                    );
   $breakIndex = end($part->stmts);
```

```
    $part->stmts[$breakIndex + 1] = new Node\Expr\Assign(
                                        $this->forElseVariable,
                                        $this->forElseValue
                                    );
    $elseWhile->stmts = $part->stmts;
    $else = $elseWhile;
}
```

Užmaskuotas kodas:

```
if (1) {
    $nm5729d8a2176c3 = 'df5729d8a2176bc';
    $nm5729d8a2176ca = 'rn5729d8a2176b8';
    while ($nm5729d8a2176ca != '5729d8a2176b5') {
        switch ($key) {
            case 'elseifs':
                $this->obfuscateElseIf($part);
                $nm5729d8a2176c3 = 'as5729d8a2176bf';
                break 2;
            default:
                $nm5729d8a2176ca = '5729d8a2176b5';
                 break;
        }
        $nm5729d8a217707 = 'rn5729d8a2176b8';
        while ($nm5729d8a217707 != '5729d8a2176b5') {
            switch ($key) {
                case 'else':
                    $this->variableWhileElement =
                        $this->firstFakeVariableElement;
                    $elseWhile = $this->createWhile(
                                    $this->forElseVariable, $this->forElseValue
                                 );
                     $nm5729d8a217707 = '5729d8a2176b5';
                     $nm5729d8a2176ca = '5729d8a2176b5';
                     $nm5729d8a2176c3 = 'as5729d8a2176bf';
                     break 3;
                default:
                    $nm5729d8a217707 = '5729d8a2176b5';
                    $nm5729d8a2176ca = '5729d8a2176b5';
                    break;
            }
        }
    }
}
```

4.2. Kodo greitaveikos tyrimas

Greitaveikos tyrimas visais atvejais buvo atliekamas po 10 kartų ir išvedamas rezultatų vidurkis. Tiriant greitaveiką buvo matuojama:

- užmaskuoto failo su daug valdymo struktūrų įvykdymo laikas;
- valdymo struktūrų veiksmai su daugiau nei 50000 įrašų iš duomenų bazės;
- maskavimo įrankio algoritmų vykdymo laikas.

Tiriant greitaveiką buvo naudojamos PHP programavimo kalbos funkcijos:

```
$time_start = microtime(true);
//Kodas
$time_end = microtime(true);
$execution_time = ($time_end - $time_start);
echo 'Vykdymo laikas: '.$execution_time.' s';
```

Žemiau pateikiama lentelė su greitaveikos testų rezultatais.

4.6. lentelė. Greitaveikos tyrimo rezultatai.

Testavimo objektas	Originalaus kodo vidutinis vykdymo laikas	Užmaskuoto kodo vidutinis vykdymo laikas
Nedidelis failas su daug valdymo struktūrų	3.1s	4.2s
Veiksmai su duomenimis iš duomenų bazės (virš 50000 įrašų)	0.3s	0.34s
Maskavimo įrankio algoritmų vykdymas	27s	27.6s

Labiausiai skyrėsi pirmojo tyrimo laikai. Taip yra todėl, nes tiriamame faile yra daug valdymo struktūrų, kurias užmaskavus, buvo sukurta dar daugiau. Didelis skirtumas matomas ir dėl to, kad failas yra nedidelis. Jei tai būtų didelė sistema, skirtumas procentais taptų kur kas mažesnis.

Veiksmų su duomenimis laikas beveik nesiskyrė, nes šiame maskavimo algoritme nėra pridedama papildomų ciklinių valdymo struktūrų, kurios suktų ciklą daugiau nei 1 kartą. Skirtumas tarp laikų yra tik dėl papildomai pridėtų perteklinių valdymo struktūrų, skirtų pakeisti valdymo struktūrų lygmenis ir supainioti statinės analizės įrankius, kaip ir pirmojo tyrimo atveju.

Trečio tyrimo metu kodas buvo maskuojamas su užmaskuotu įrankiu. Nors ir užmaskuotų valdymo struktūrų kiekis šiame įrankyje yra didžiulis, tačiau jų gylis yra nedidelis, todėl laikas mažai skiriasi. Taip pat yra paklaida, kuri atsiranda nuskaitant direktorijas ir įrašant užmaskuotus failus.

4.3. Tyrimo išvados

Kodo veikimo tyrimas yra vienas iš geriausių metodų maskavimo įrankio tobulinimui. Maskavimo algoritmai yra pakankamai painūs ir tik maskuojant įvairų kodą randamos klaidos ir daromi pataisymai, visko suplanuoti praktiškai neįmanoma tokioms sistemoms.

Ištyrus užmaskuoto kodo veikimą su patobulintu įrankiu, buvo surasta keletas situacijų, kai maskavimo algoritmas veikė netinkamai. Loginės klaidos buvo pašalintos ir tolesnių testų metų, visi rezultatai buvo sėkmingi. Tiek failas su daug valdymo struktūrų, tiek sistema sudaryta iš 91 failo, tiek pats maskavimo įrankis veikia be klaidų su užmaskuotu kodu.

Teisingai atliktas valdymo srautų maskavimas greitaveikai didelės įtakos neturi. Kaip testo rezultatai parodė, skirtumas pasirodo tik tokiu atveju, jei yra labai daug valdymo struktūrų, tai yra normalu, nes sukuriamos perteklinės valdymo struktūros kiekvienai maskuojamai, tad valdymo struktūrų vykdymas užtrunka ilgiau. Nors ir sukuriama dvigubai daugiau valdymo struktūrų, vykdymo laikas skiriasi tik apie 25%. Tokie rezultatai gauti, nes perteklinės valdymo struktūros nutraukiamos laiku bei tinkamu laiku pereinama į aukštesnius lygmenis. Tai dar vienas svarbus aspektas maskuojant valdymo struktūras – protingas lygmenų keitimas, kad perteklinių valdymo struktūrų vykdymas neužtruktų per ilgai.

Užmaskavus kodo valdymo struktūras, kodo apimtis gali padidėti dvigubai ar net trigubai. Tyrimo metu naudotas failas su daug valdymo struktūrų yra 80 eilučių. Užmaskuotas šis failas yra 174 eilučių. Todėl valdymo struktūrų maskavimas nėra geras sprendimas technologijose, kur kodas siunčiamas vartotojui interneto ryšiu. Pavyzdžiui, jei padidintume interaktyvaus tinklalapio JavaScript failus dvigubai ar trigubai, tinklalapio greitaveika taptų labai lėta.

Testavimo metu buvo išbandytos ir originalios maskavimo PĮ funkcijos, kurios pakeičia klasių, funkcijų bei kintamųjų vardus, taip pat sutraukia kodą. Užmaskavus patį maskavimo įrankį su šiais parametrais, įrankis neveikė. Viena iš priežasčių - maskuojamos yra visos funkcijos, net ir tokios kaip konstruktorius ar griovėjas (angl. *Destructor).* Pakeitus konstruktoriaus ar griovėjo pavadinimą kodas tampa neveikiantis. Priežasčių galėjo būti ir daugiau, kodėl kodas neveikė. Tai puikus pavyzdys, kodėl nereikia visada pasitikėti kitų programuotojų parašytais maskavimo įrankiais. Jie gali sugadinti PĮ ir klaidas rasti gali būti labai sunku, jei nėra įsigilinta į maskavimo įrankio veikimą ir kodą.

Norint parašyti universalų maskavimo įrankį visoms valdymo struktūroms ir jų sąlygoms, reikėtų nedidelio dirbtinio intelekto. Programavimo kalbose atsiranda vis naujų funkcijų, programavimo šablonų ir norint turėti pilnai atuomatizuotą maskavimo įrankį valdymo struktūroms, reikėtų jog jis mokytųsi rasdamas naujus dalykus. Kitu atveju tokiam įrankiui reikia pastovios priežiūros ir palaikymo. Tai dar viena iš priežasčių, kodėl universalus maskavimo įrankis visiems atvejams negali egzistuoti arba yra per brangus palaikymui atsižvelgiant į jo naudą.

5. IŠVADOS

1. Vieno apsaugos metodo nepakanka apsisaugoti nuo visų galimų įsilaužimo būdų. Norint turėti saugią programinę įrangą, reikia sujungti apsaugos metodus pagal poreikį.
2. Tobulinami yra ne tik apsaugos metodai, bet ir įsilaužimo į programinę įrangą metodai ir įrankiai, pavyzdžiui: tyliosios derintuvės, kurių aptikti beveik neįmanoma.
3. Svarbu apsaugoti ne tik programinės įrangos kodą, bet ir įrankių kodą, kuriais programinė įranga apsaugoma. Žinant apsaugos įrankių veikimo modelį, galima sukurti labai tikslius įrankius apsaugoto kodo analizei.
4. Kodo maskavimas padeda apsaugoti kodą nuo kodo analizės, kurią daro programuotojas skaitydamas kodą bei statinės analizės įrankių, kurie analizuoja kodo vykdymą. Norint apsisaugoti nuo programuotojo analizės, reikia maskuoti klases, kintamuosius, sutraukti kodą bei pridėti perteklinius duomenis tam, kad kodas taptų kuo sunkiau skaitomas. Norint apsisaugoti nuo statinės analizės įrankių, reikia maskuoti valdymo struktūras.

   Valdymo struktūros turi būti maskuojamos taip, kad būtų pakeisti valdymo struktūrų lygmenys ir jos taptų viena nuo kitos nepriklausomos. Nepakeitus lygmens, statinės analizės įrankiai galės išanalizuoti programuotojui sunkiai skaitomą kodą. Maskuojant valdymo struktūras svarbu pridėti perteklines valdymo struktūras, tačiau jos turi būti protingai valdomos, kad greitaveika nesumažėtų tiek, kad programinė įranga nebeatitiks reikalavimų.
5. Maskavimas efektyviausiai apsaugo programinę įranga, kai yra naudojamas žemo lygio programavimo kalbomis parašytam kodui apsaugoti. Žemesnio lygmens programavimo kalbos turi tokias savybes, kaip: kodo keitimas vykdymo metu, duomenų ir kodo apjungimas bei struktūrų keitimas, todėl maskuoti žemesnio lygio programavimo kalbomis parašytą kodą efektyvu, nes galima pritaikyti įvairesnius ir sudėtingesnius algoritmus.
6. Jei iš maskavimo tikimasi stiprios kodo apsaugos, maskavimo įrankis turi būti kuriamas specialiai maskuojamai sistemai. Kai kurias funkcijas galima panaudoti jau sukurtų įrankių, tačiau pilnai pasitikėti jais nereikėtų. Kiekviena sistema sukurta naudojant skirtingus kodo modeliavimo metodus, todėl maskavimo algoritmas turi būti pritaikytas būtent maskuojamam kodui.
7. Kuriant maskavimo įrankį nereikėtų naudotis jau sukurtais algoritmais, kurie visur išplatinti. Reikia laikytis tik pagrindinių principų, tokių kaip: keičiant valdymo struktūras, jas reikia keisti lygmenimis. Kuo unikalesnis maskavimo algoritmas, tuo mažesnė tikimybė, kad jam bus sukurtas ar pritaikytas statinės analizės įrankis.
8. Sukūrus teisingą kodo maskavimo algoritmą, nepriklausomai nuo to, ar maskuojamos valdymo struktūros, ar klasės, ar kintamieji, ar pridedamas perteklinis kodas, greitaveika turi

skirtis labai nežymiai. Dėl greitaveikos tinkamai užmaskuoti ciklines valdymo struktūras labai sudėtinga. Jei bus pridedama per daug perteklinių valdymo struktūrų arba jos bus netinkamai valdomos, greitaveika gali išaugti dešimtis ar net šimtus kartų.

9. Testuojant užmaskuotą kodą labai svarbu patikrinti visą jo funkcionalumą. Šiam tikslui puikiai tinka elementų testai (angl. *Unit tests*). Tam maskuojamas kodas turi būti parašytas taip, kad tokie testai būtų palaikomi. Jei galutinis rezultatas įvykdžius kodą yra toks pat nemaskuoto kodo ir užmaskuoto kodo, tuomet didelė tikimybė, kad viskas sėkmingai užmaskuota. Išskirtiniais atvejais gali teki atlikti funkcinius testus arba testavimą „rankomis".

10. Ilgą laiką kodo maskavimas buvo nusistovėjusi sritis ir dauguma metodų buvo seni, o tobulinami tik maskavimo algoritmai. Atsiradus vis dar tiriamam kriptografiniam maskavimui, ši sritis tapo ir vėl daug žadanti programinės įrangos apsaugos sferoje. Atsirado galimybė sukurti tokį maskavimo tipą, kuris teoriškai yra neįveikiamas.

6. LITERATŪRA

[1] Vacius Jusas, Tomas Blažauskas, Šarūnas Packevičius, „Programų sistemų apsaugos inžinerija“, [Tinkle]. Available: https://www.ebooks.ktu.lt/eb/240/programu-sistemu-apsaugos-inzinerija/ [Kreiptasi 15 11 2014]

[2] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, Wenke Lee, „Impeding Malware Analysis Using Conditional Code Obfuscation“, [Tinkle]. Available: http://cyber4.us/sites/default/files/Impeding%20Malware%20Analysis%20Using%20Conditional%20Code%20Obfuscation-NDSS2008.pdf [Kreiptasi 15 11 2014]

[3] Michael Lesk, Martin R. Stytz, Roland R. Trope, „Software Protection through Anti-Debugging“, [Tinkle]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4218560 [Kreiptasi 20 11 2014]

[4] Microsoft corporation,“ CheckRemoteDebuggerPresent function“, [Tinkle]. Available: http://msdn.microsoft.com/en-us/library/ms679280%28VS.85%29.aspx [Kreiptasi 20 11 2014]

[5] YASH, „Debugger and Breakpoints“, [Tinkle]. Available: http://www.ksyash.com/2011/01/210/ [Kreiptasi 20 11 2014]

[6] „Rename classes, methods, etc.“, [Tinkle]. Available: http://www.aldaray.com/Rummage/Doc/Rename-classes-methods [Kreiptasi 20 11 2014]

[7] „Debugging Model“, [Tinkle]. Available: http://tcloud.sjtu.edu.cn/wiki/index.php/VM:EuroSys10_paper#2._Debugging_Model [Kreiptasi 15 11 2014]

[8] „Architectural Principles That Prevent Code Modification or Reverse Engineering“, [Tinkle]. Available: https://www.owasp.org/index.php/Architectural_Principles_That_Prevent_Code_Modification_or_Reverse_Engineering [Kreiptasi 15 11 2014]

[9] „Data Analysis: Facebook Hacker Cup 2013“, [Tinkle]. Available: http://mishadoff.com/blog/data-analysis-facebook-hacker-cup/ [Kreiptasi 15 11 2014]

[10] „September 2013 Cyber Attacks Statistics“, [Tinkle]. Available: http://hackmageddon.com/2013/10/20/september-2013-cyber-attacks-statistics [Kreiptasi 15 11 2014]

[11] Dr James Hamilton, „What is software watermarking?“, [Tinkle]. Available: https://jameshamilton.eu/research/what-software-watermarking [Kreiptasi 08 12 2014]

[12] Azizah Abd Manaf, Shamsul Sahibuddin, Rabiah Ahmad, Salwani Mohad Daud, Eyas El-Qawashmeh, „Informatics engineering and information science“, [Tinkle]. Available: http://goo.gl/WfceVN [Kreiptasi 08 12 2014]

[13] Michael Goff, „Top 20 Countries for Software Piracy and License Misuse“, [Tinkle]. Available: http://www.vilabs.com/news-section/code-confidential/top-20-countries-software-piracy-2014 [Kreiptasi 08 12 2014]

[14] „Tutorial: Writing Tasks“, [Tinkle]. Available: https://ant.apache.org/manual/tutorial-writing-tasks. [Kreiptasi 01 05 2015]

[15] „yGard ant task documentation“, [Tinkle]. Available: https://www.yworks.com/products/yguard/yguard_ant_howto.html#installation [Kreiptasi 04 05 2015]

[16] T´ımea L´aszl´o and Akos Kiss, „Obfuscating C++ Programs via Control Flow Flattening“, [Tinkle]. Available: http://www.inf.u-szeged.hu/~akiss/pub/pdf/laszlo_obfuscating.pdf [Kreiptasi 15 02 2016]

[17] Stephen Drape, „Obfuscation of Abstract Data-Types“, [Tinkle]. Available: http://www.cs.ox.ac.uk/stephen.drape/papers/thesis.pdf [Kreiptasi 25 04 2016]

[18] Matthew Green, „Cryptographic obfuscation and 'unhackable' software“, [Tinkle]. Available: http://blog.cryptographyengineering.com/2014/02/cryptographic-obfuscation-and.html [Kreiptasi 24 04 2016]

[19] Tom Roeder, Fred B. Schneider, „Proactive Obfuscation“, [Tinkle]. Available: https://www.cs.cornell.edu/fbs/publications/ProactiveObfuscTOCS.pdf [Kreiptasi 28 04 2016]

[20] Jean Yves Marion, Daniel Reynaud, „Obfuscation by Interpretation“, [Tinkle]. Available: https://indefinitestudies.files.wordpress.com/2008/08/obfuscation.pdf [Kreiptasi 04 05 2016]

7. PRIEDAI

7.1. Kodas su kuriuo atlikti tyrimai

Originalus kodas failo su daug valdymo struktūrų:

```
//Tekstai kurie išvedami į ekraną
$output1 = "elseif";
$output2 = "if";
$output3 = "nelygybė";
$output4 = "palyginimas su skaičiumi";
$output5 = "bool tipo palyginimas";
$output6 = "Giliausia vieta";
$output7 = "if palyginimas be kintamųjų";
$output8 = "elseif palyginimas be kintamųjų";
$output9 = "bool palyginimas be kintamųjų";
$output10 = "else";

//Kintamieji sąlygoms tikrinti
$var1 = "testas";
$var2 = 5;
$var3 = 0;
$var4 = true;
$var5 = false;
$var6 = [0 => 1];

//Testuojamas kodas
if ($var6[0] == 5) {

} elseif ($var2 == 5) {
    //elseif testas
    echo $output1."<br>";
    if ($var1 == false) {

    } else {
        if ($var1 == "testas") {
            //if testas
            echo $output2."<br>";
            if ($var2 != "test3") {
                //nelygybės testas
                echo $output3."<br>";
                if ($var2 == 5) {
                    //Palyginimo su skaičium testas
                    echo $output4."<br>";
                    if ($var1 != "testas"){

                    } elseif ($var5 == false) {
                        //bool tipo palyginimas
                        echo $output5."<br>";
                        if (5 != 5) {

                        } else {
                            //else testas
                            if ($var1 == "testas") {
                                if ($var5 != true) {
                                    echo $output6."<br>";
                                } else {
```

```
                                "neveikia";
                            }
                        }
                        if (5 == 5) {
                            //if palyginimo be kintamųjų testas
                            echo $output7."<br>";
                        }
                        if (false) {

                        } elseif (5 == 5) {
                            //elseif palyginimo be kintamųjų testas
                            echo $output8."<br>";
                        }
                        if (true) {
                            //bool tipo palyginmas be kintamųjų
                            echo $output9."<br>";
                        }
                    }
                }
            }
        }
    }
}
if($var1 == 2) {

} else {
    echo $output10."<br>";
}
```

Užmaskuotas kodas su daug valdymo struktūrų:

```
$output1 = 'elseif';
$output2 = 'if';
$output3 = 'nelygybė';
$output4 = 'palyginimas su skaičiumi';
$output5 = 'bool tipo palyginimas';
$output6 = 'Giliausia vieta';
$output7 = 'if palyginimas be kintamųjų';
$output8 = 'elseif palyginimas be kintamųjų';
$output9 = 'bool palyginimas be kintamųjų';
$output10 = 'else';
$var1 = 'testas';
$var2 = 5;
$var3 = 0;
$var4 = true;
$var5 = false;
$var6 = array(0 => 1);
if ($var6[0] == 5) {
} elseif ($var2 == 5) {
    echo $output1 . '<br>';
    if (1) {
        $nm572a2c94eaeb1 = 'df572a2c94eaea5';
        $nm572a2c94eaebe = 'rn572a2c94eae9f';
        while ($nm572a2c94eaebe != '572a2c94eae98') {
            switch ($var1) {
                case false:
                    $nm572a2c94eb40a = '572a2c94eae98';
                    $nm572a2c94eaeb1 = 'as572a2c94eaeab';
```

```
                default:
                    $nm572a2c94eaebe = '572a2c94eae98';
                    break;
            }
        }
        while ($nm572a2c94eaeb1 != 'as572a2c94eaeab') {
            if (1) {
                $nm572a2c94ee8e8 = 'df572a2c94ee8e2';
                $nm572a2c94ee8ee = 'rn572a2c94ee8e0';
                while ($nm572a2c94ee8ee != '572a2c94ee8dd') {
                    switch ($var1) {
                        case 'testas':
                            echo $output2 . '<br>';
                            if (1) {
                                $nm572a2c94eeb82 = 'df572a2c94eeb7c';
                                $nm572a2c94eeb88 = 'rn572a2c94eeb79';
                                while         ($nm572a2c94eeb88         !=
'572a2c94eeb77') {
                                    switch ($var2) {
                                        default:
                                            echo $output3 . '<br>';
                                            if (1) {
                                                $nm572a2c94eee08        =
'df572a2c94eee02';
                                                $nm572a2c94eee0d        =
'rn572a2c94eedff';
                                                while    ($nm572a2c94eee0d
!= '572a2c94eedfc') {
                                                    switch ($var2) {
                                                        default:

$nm572a2c94eee0d = '572a2c94eedfc';

                                                            break;
                                                        case '5':
                                                            echo  $output4
. '<br>';
                                                            if (1) {

$nm572a2c94ef106 = 'df572a2c94ef100';

$nm572a2c94ef10b = 'rn572a2c94ef0fe';
                                                                while
($nm572a2c94ef10b != '572a2c94ef0fa') {

switch ($var1) {

default:

$nm572a2c94ef166 = '572a2c94ef0fa';

$nm572a2c94ef106 = 'as572a2c94ef103';

case 'testas':

$nm572a2c94ef10b = '572a2c94ef0fa';

break ;
                                                                }
```

```
$nm572a2c94ef12e = 'rn572a2c94ef0fe';
                                                                          while
($nm572a2c94ef12e != '572a2c94ef0fa') {
switch ($var5) {
case false:
echo $output5 . '<br>';
if (5 != 5) {
} else {
if (1) {
$nm572a2c94ef6c5 = 'df572a2c94ef6bf';
$nm572a2c94ef6ca = 'rn572a2c94ef6bc';
while ($nm572a2c94ef6ca != '572a2c94ef6b9') {
switch ($var1) {
default:
$nm572a2c94ef6ca = '572a2c94ef6b9';
break;
case 'testas':
if (1) {
$nm572a2c94ef9a3 = 'df572a2c94ef99d';
$nm572a2c94ef9a9 = 'rn572a2c94ef991';
while ($nm572a2c94ef9a9 != '572a2c94ef98e') {
switch ($var5) {
case true:
$nm572a2c94ef9a9 = '572a2c94ef98e';
break ;
default:
echo $output6 . '<br>';
$nm572a2c94ef9a3 = 'as572a2c94ef9a0';
break 2;
}
```

```
}
while ($nm572a2c94ef9a3 != 'as572a2c94ef9a0') {
'neveikia';
$nm572a2c94ef9a3 = 'as572a2c94ef9a0';
}
}
$nm572a2c94ef6c5 = 'as572a2c94ef6c2';
break 2;
}
}
}
if (5 == 5) {
echo $output7 . '<br>';
}
if (false) {
} elseif (5 == 5) {
echo $output8 . '<br>';
}
if (true) {
echo $output9 . '<br>';
}
}
$nm572a2c94ef12e = '572a2c94ef0fa';
$nm572a2c94ef10b = '572a2c94ef0fa';
$nm572a2c94ef106 = 'as572a2c94ef103';
break 3;
default:
$nm572a2c94ef12e = '572a2c94ef0fa';
$nm572a2c94ef10b = '572a2c94ef0fa';
```

```
break;
                                                                        }
                                                                    }
                                                                }
                                                            }

$nm572a2c94eee08 = 'as572a2c94eee05';
                                                            break 2;
                                                    }
                                                }
                                            }
                                            $nm572a2c94eeb82              =
'as572a2c94eeb7f';
                                            break 2;
                                        case 'test3':
                                            $nm572a2c94eeb88              =
'572a2c94eeb77';
                                            break ;
                                    }
                                }
                            }
                            $nm572a2c94ee8e8 = 'as572a2c94ee8e5';
                            break 2;
                        default:
                            $nm572a2c94ee8ee = '572a2c94ee8dd';
                            break;
                    }
                }
            }
            $nm572a2c94eaeb1 = 'as572a2c94eaeab';
        }
    }
}
if (1) {
    $nm572a2c94f06eb = 'df572a2c94f06e5';
    $nm572a2c94f06f1 = 'rn572a2c94f06e3';
    while ($nm572a2c94f06f1 != '572a2c94f06e0') {
        switch ($var1) {
            default:
                $nm572a2c94f06f1 = '572a2c94f06e0';
                break;
            case '2':
                $nm572a2c94f0743 = '572a2c94f06e0';
                $nm572a2c94f06eb = 'as572a2c94f06e8';
        }
    }
    while ($nm572a2c94f06eb != 'as572a2c94f06e8') {
        echo $output10 . '<br>';
        $nm572a2c94f06eb = 'as572a2c94f06e8';
    }
}
```

Originalus kodas su duomenimis iš duomenų bazės:

```
$dbobj = $db->query('SELECT * FROM `test`');
$result = $dbobj->fetchAll();

if ($result[0]['name'] == 'testas') {
```

```
    if (true) {
        foreach ($result as $resultRow) {
            if ($resultRow['name'] != 'testas'){

            } elseif ($resultRow['id'] == 51200) {
                if ($result[5]['surname'] == 'testauskas') {
                    if (false) {

                    } else {
                            echo "<pre>";
                            print_r($resultRow);
                            echo "</pre>";
                    }
                }
            }
        }
    }
}
```

Užmaskuotas kodas su duomenimis iš duomenų bazės:

```
$db = new PDO('mysql:host=localhost;dbname=test', 'root');
$dbobj = $db->query('SELECT * FROM `test`');
$result = $dbobj->fetchAll();
$test = $result[0]['name'];
$test5 = $result[5]['name'];
if (1) {
    $nm572a0ddaaa14e = 'df572a0ddaaa148';
    $nm572a0ddaaa155 = 'rn572a0ddaaa145';
    while ($nm572a0ddaaa155 != '572a0ddaaa142') {
        switch ($test) {
            default:
                $nm572a0ddaaa155 = '572a0ddaaa142';
                break;
            case 'testas':
                if (true) {
                    foreach ($result as $resultRow) {
                        $rowId = $resultRow['id'];
                        $rowName = $resultRow['name'];
                        if ($resultRow['name'] != 'testas') {
                        } elseif ($rowId == 51200) {
                            if (1) {
                                $nm572a0ddaaea3c = 'df572a0ddaaea36';
                                $nm572a0ddaaea42 = 'rn572a0ddaaea33';
                                while         ($nm572a0ddaaea42         !=
'572a0ddaaea30') {
                                    switch ($test5) {
                                        case 'testauskas':
                                            if (false) {
                                            } else {
                                                echo '<pre>';
                                                print_r($resultRow);
                                                echo '</pre>';
                                            }
                                            $nm572a0ddaaea3c                =
'as572a0ddaaea39';

                                            break 2;
                                        default:
                                            $nm572a0ddaaea42                =
```

```
'572a0ddaaea30';
                                                  break;
                                         }
                                    }
                               }
                          }
                     }
                }
                $nm572a0ddaaa14e = 'as572a0ddaaa14b';
                break 2;
        }
    }
}
```