KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

KOMPIUTERIŲ KATEDRA

Nerijus Jusas

Programų apsaugos metodas, naudojant sistemos parašą

Magistro darbas

Darbo vadovas

doc. dr. A. Venčkauskas

Kaunas, 2012

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

KOMPIUTERIŲ KATEDRA

Nerijus Jusas

Programų apsaugos metodas, naudojant sistemos parašą

Magistro darbas

Recenzentas

doc. dr. E.Karčiauskas

2012-05-

Darbo vadovas

doc. dr. A. Venčkauskas

2012-05-

Darbą atliko:

Nerijus Jusas

2012-05-28

Kaunas, 2012

TURINYS

1. ĮVADAS

Taikomųjų programų kūrimui reikia investuoti daug laiko, pinigų ir pastangų. Nelegalus programinės įrangos naudojimas ir kopijavimas, žinomas kaip „programų piratavimas“, yra viena [35] didžiausių problemų su kuria susiduria taikomųjų programų kūrėjai. Business Software Alliance (*BSA*) tyrimai [1] parodė, kad pasauliniu mastu 43% programinės įrangos yra naudojama nelegaliai ir kasmet piratavimo mastai auga 2%. Artimiausiu metu šios tendencijos nesikeis, nes kompiuterinės sistemos sparčiai plečiasi tuose regionuose, kuriose piratavimo lygis labai aukštas.

Pagal BSA, Lietuvoje per 2006-2010 metų laikotarpį, nelegalios programinės įrangos vartojimas sumažėjo 3%. Tačiau ji vis dar sudarė 54% visos programinės įrangos. Dėl to programinės įrangos kūrėjai, 2010 metais, patyrė 38 milijonus nuostolių, skaičiuojant Jungtinių Amerikos Valstijų doleriais. Nors nelegalios programinės įrangos vartojimas Lietuvoje ir mažėja, tačiau kūrėjų nuostoliai didėja, nes programinės įrangos sudėtingumas auga ir jos gamybos kaštai didėja.

Piratavimo būdai nuo kurių labiausiai nukenčia programinės įrangos kūrėjai [2]:

1) Vartotojo piratavimas (*User piracy*). Vartotojo ar įmonės darbuotojo piratavimo veiksmai atsiranda, kai vartotojas ar darbuotojas padaro turimos legalios programinės įrangos kopijas asmeniniam naudojimui. Tai gali atsitikti, kai programa yra įdiegiama į kompiuterį, kuris yra nelicencijuotas. Tai nutinka kompanijose, kai yra įdiegiama daugiau programos kopijų nei turima licenzijų. Nors programinė įranga ir yra įsigyta, tačiau naudojama neteisėtai
2) Internetinis piratavimas (*Internet piracy*). Internetinis piratavimas atsiranda tada, kai kas nors nelegaliai parsisiunčia programinės įrangos kopiją iš interneto ir ją įdiegia į savo kompiuterį arba įrašo į kompaktinį diską. Tai yra populiariausias piratavimo būdas. Jis dažniausiai vyksta trimis būdais: nelegalios programinės įrangos parsisiuntimas iš talpinimo svetainės, pirkimas iš nelegalaus pardavėjo ir gaunant kopiją per dalinimosi svetaines (*Peer-to-Peer*).
3) Kietojo disko užpildymas (*Hard disc loading*). Toks atvejis atsiranda tada, kai kompiuterių perpardavėjas į kietąjį diską įrašo neteisėtai gautą programinę įrangą arba padaro daugiau nei leistina legalios programinės įrangos kopijų.

4) Programinės įrangos klastotės (*Software counterfeiting*). Tai nelegalus programinės įrangos kopijų darymas ir jų pardavinėjimas. Tokiu atveju kopijuojama visa programinės įrangos pakuotė įskaitant ir naudojimo instrukciją.
5) Serverio perkrovimas (*Server overloading*). Rečiausiai pasitaikantis programinės įrangos piratavimo būdas, kuris atsiranda darbo vietose, kai serveryje esančia programine įranga vienu metu bando naudotis per daug vartotojų. Taip nutinka, kai vieno vartojo licencijos programa yra įdiegiama į serverį, kur ja gali naudotis daugiau vartotojų nei numatyta licencijoje.

Be šių penkių piratavimo būdų, programinės įrangos kūrėjai kenčia dėl intelektinės nuosavybės vagysčių, dažniausia tai yra programose naudojamų algoritmų. Algoritmų vogimas – procesas, kai panaudojus automatines ar rankines atvirkštinės inžinerijos (*Reverse engineering*) priemonėmis iš programos yra išgaunamas vertingas algoritmas ir įdedamas į kitą kuriamą programą, kuri tampa pradinės programos konkurente. Apsisaugojimui nuo piratavimo ar intelektinės nuosavybės vagysčių yra naudojami programiniai ir aparatūriniai apsaugos būdai. Aparatūriniai apsaugos metodai priklauso nuo išorinės aparatinės įrangos ir apsaugai yra naudojami rečiau.

Programiniai apsaugos metodai dažniausiai yra naudojami apsaugoti programas nuo atvirkštinės inžinerijos. Taikant programinius apsaugos metodus yra siekiama kaip įmanoma labiau apsunkinti programos „nulaužėjo“ galimybės panaudoti atvirkštinės inžinerijos automatines ir rankines priemones (aprašytas 2.1 skyriuje). Kuriomis pasinaudojus galima apeiti programos apsaugos metodus ar išgauti norimą algoritmą. Norint apsunkinti tokių priemonių naudojimą taikomųjų programų apsaugai naudojamos įvairios transformacijas, programinio kodo ir/ar programos valdymo srauto supainiojimui.

Be transformacijų prie programinių apsaugos metodų yra priskiriama ir programinio kodo ar taikomosios programos atskirų dalių glaudinimo ir/ar šifravimo metodai. Pastarieji metodai patikimai apsaugo nuo statinio programų nagrinėjo (programa nėra vykdoma), nes atakuotojas pirmiausia turi išskleisti ar iššifruoti apsaugotą programos dalį. Be to, šią apsaugos technologiją galima lengvai panaudoti kartu su kitomis programinėmis apsaugos priemonėmis. Todėl panaudojus įvairias jų kombinacijas galima pasiekti gerų rezultatų kovoje prieš atvirkštinę inžineriją. Tačiau toks apsaugos metodas turi ir problemų – kaip generuoti ir kur saugoti reikalingus raktus.

Šio magistrinio darbo tikslas – atlikti taikomųjų programų programinių apsaugos metodų tyrimą ir pasiūlyti programinį apsaugos metodą, paremtą atskirų programos modulių šifravimu. Kur šifravimo/iššifravimo raktų generavimui nebūtų naudojamos papildomos aparatūrinės įrangos ir infrastruktūros raktų saugojimui. Programos kodas saugomas šifruotoje formoje, šifravimo raktai generuojami realiame laike, pagal poreikį, prieš vykdant šifruotą programos modulį.

2. TAIKOMŲJŲ PROGRAMŲ APSAUGOS METODŲ ANALIZĖ

Didėjant programinės įrangos naudojimui, sparčiai didėja ir programinių atvirkštinės inžinerijos priemonių skaičius, kurios yra lengvai prieinamos visiems kas nori. Esant tokiai situacijai buvo sugalvota nemažai būdų ir priemonių kaip būtų galima apsisaugoti nuo atvirkštinės inžinerijos. Vieni iš pasiūlytų apsaugos metodų yra teoriniai kiti praktiniai, tačiau visus juos galima suskirstyti į dvi grupes: programinius ir aparatūrinius. Programiniai apsaugos mechanizmai yra įdiegiami į taikomąsias programas, kur yra apsaugoma arba visa programa arba tik tam tikros jos dalys, pvz. algoritmai. Aparatūriniai apsaugos metodai priklauso nuo papildomos išorinės aparatūros. Prieš atliekant taikomųjų programų apsaugos metodų analizę buvo apžvelgtos priemonės naudojamos programų „nulaužimui“.

2.1. Apsaugos metodų „nulaužimui“ naudojamos priemonės

Nesvarbu nuo kokių grėsmių programinė įranga yra apsaugota ar nuo kopijavimo, ar nuo algoritmų vagysčių, atakuotojai, norėdami „nulaužti“ apsaugas naudoja atvirkštinę inžineriją ir dviejų tipų priemones – derintuvus (*Debugger*) arba ardymo (*Disassemblers*) [3]:

1) **Atvirkštinė inžinerija** – nagrinėja dvejetainį failą, paversdama mašininį kodą į asemblerio kalbos kodą, kad būtų lengviau nustatyti, kokia programos struktūra ir kaip ji veikia [36].
2) **Derintuvai** – priemonės leidžiančios atakuotojui atsekti programos veikimą instrukcija po instrukcijos ir sustabdyti jos veikimą bet kurioje jos vietoje ir sekti kaip ji veikia. Tiesa, kad programų, parašytų aukštesnio lygio programavimo kalba (kaip C++, Delphi ar Visual Basic) veikimas gali būti atsekamas tik naudojant asemblerius, tačiau atakuotojai ir be jų dažnai gerai supranta kaip veikia programa.
3) **Ardymo** – priemonės kurios gali programos kodą išskaidyti į asemblerio kalbą. Ardymo programos turi vieną privalumą prieš derintuvus, tai kad jos visada verčia į asemblerio kalbą, todėl atakuotojui reikia žinoti tik vieną programavimo kalbą. Išverstos asemblerio kalbos kokybė priklauso nuo naudojamos ardymo programos. Geriausios programos komentuoja išverstą kodą, taip padarydamos jį suprantamesnį atakuotojui.

Programos veikimo, struktūros ar „nulaužimo“ procesas, naudojant atvirkštinę inžineriją ir derintuvus bei ardymo programos, gali būti atliekamas dviem būdais: dinamiškai ir statiškai [25]. Dinaminio analizavimo metu programa yra vykdoma, vykdymo metu yra sekamos vykdymo

instrukcijos. Šio metodo silpnoji vieta yra ta, kad instrukciją galima sekti tik tam tikru keliu ir net atlikus kelis programos paleidimus nėra garantijos, kad visi galimi keliai įvykdyti. Darant statinę programos analizę pačios programos vykdyti nereikia. Yra imamas dvejetainis failas ir jis analizuojamas.

Yra daugybė programinių paketų, kurie naudojami atvirkštinės analizės atlikimui. Vieni nuo kitų skiria tik funkcionalumu, be to dauguma šios programinės įrangos buvo sukurta kaip pagalbinės priemonės programų kūrėjams. Šie paketai naudojami tiek programų testavimui, tiek „nulaužimams“, tiek pasiūlytų apsaugos metodų patikimumui tirti. Literatūroje dažniausiai minimi tokie programiniai paketai:

OllyDbg

OllyDbg yra derintuvas skirtas x86 platformai ir skirtas dvejetainio kodo analizei, kuri yra naudinga, kai nėra programinio kodo. Programos pagalba galima sekti registrus, atpažinti procedūras, programinės sąsajos iškvietimus, sukeitimus, lenteles, konstantas taip pat galima surasti objekto failus bei naudojamas bibliotekas. Kadangi ši programinė įranga yra nemokama, tai ji plačiai paplitusi tarp programų „nulaužėjų“.

SoftIce

Derintuvas skirtas Microsoft Windows aplinkai. Svarbiausia, jis yra sukurtas veikti taip, kad operacinė sistema nežinotų apie jo buvimą. Jeigu reikia, SoftIce gali sustabdyti visas Windows vykdomas operacijas. Tai yra labai svarbu norint derinti tvarkykles. Dėl to jog ji veikia žemame operacijų lygyje, SoftIce yra populiarus tarp programų nulaužėjų.

IDA Pro

Interaktyvi programinio kodo ardymo priemonė, skirta atvirkštiniai inžinerijai. Palaiko daug įvairiausių vykdomųjų formatų skirtų skirtingiems procesoriams ir operacinėms sistemoms. IDA vykdo daug automatinės programinio kodo analizės, naudojant kryžmines nuorodas (*Cross-references*) tarp kodo dalių, žinias apie programinės sąsajos kviečiamus kintamuosius ir kitą informaciją. IDA taip pat turi interaktyviąsias funkcijas, kurios padera išardyti programos kodą ir po to dalis po dalį sudaryti pradinę programą.

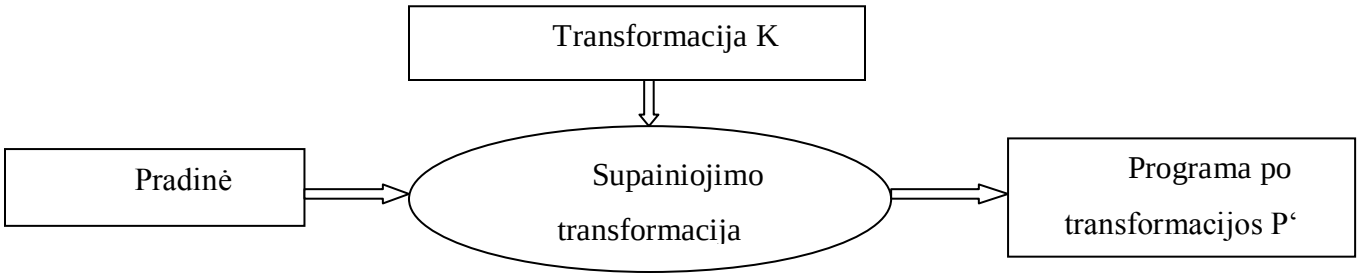
2.2. Programiniai apsaugos metodai

Taikomųjų programų ar jose naudojamų algoritmų apsaugai dažniausiai yra taikomi tokie programiniai apsaugos būdai: programinio kodo supainiojimas, derintuvų aptikimas, apsauga nuo išskaidymo į asemblerio programavimo kalbą, programinio kodo glaudinimas ar šifravimas ir

vandens ženklinimas. Programinio kodo supainiojimas yra geriausia priemonė apsaugai nuo atvirkštinės inžinerijos, kuri kelia didžiausią grėsmę programinės įrangos kūrėjams interpretacinėmis kalbomis. Derintuvų aptikimo metodai leidžia taikomajai programai nustatyti ar yra naudojamas derintuvas ir imtis atitinkamų priemonių. Taikant apsaugą nuo išskaidymo – išskaidytą programą bandoma padaryti kaip įmanoma sunkiau suprantamą. Vandens ženklinimas į taikomąją programą įterpia informaciją apie autorines teises.

2.2.1. Programinio kodo supainiojimas

Taikomųjų programų programinio kodo supainiojimo technika skirta padaryti programos valdymo srautą ar duomenų struktūras, kuriose yra saugoma svarbi informacija, kaip įmanoma sunkiau suprantamas atakuotojams, naudojantiems atvirkštinės inžinerijos priemones. Collberg [26] supainiojimo transformaciją K apibrėžia kaip programos P transformavimą į programą P‘ taip, kad matomas programų P ir P‘ veikimas nesiskirtų. Schematinis proceso veikimas yra parodyta 1 pav. Originalios programos P ir supainiotos programos P‘ funkcionalumas vartotojui turi nesiskirti, kas gali nutikti atliekant transformaciją. Kodo supainiojimo technologijos dažniausiai yra naudojamos programoms parašytoms .NET, Java ir kitomis interpretacinėmis programavimo kalbomis.



1 pav. Programinio kodo supainiojimo veikimo principas [4].

Kaip matyti iš 1 paveikslo, kodo supainiojimo metodo teikiamas saugumas didele dalimi priklauso nuo pasirinktos transformacijos ***K***. Kurios vienas iš pagrindinių atributų, naudojamas matuoti supainiojimo patikimumą yra vadinama stiprumu (*Potency*) [5]. Stiprumas nurodo kaip sunkiai suprantamas ir analizuojamas tampa supainiotas kodas jį palyginus su pradiniu. Jeigu ***K*** turės nepakankamą supainiojimo stiprumą, tai panaudojus atvirkštinę inžineriją ir atitinkamas programines priemones, atakuotojai galės greitai perprasti supainiotą programinį kodą ir įveikti apsaugą bei atkurti pradinę programą ar apsaugotą algoritmą.

Kodo painiojimas yra plačia tyrinėjama, programų apsaugos sritis, todėl yra įvairiausių pasiūlymų kaip būtų galima atlikti transformaciją. [22] straipsnyje siūloma daugumą valdymo perdavimų pakeisti signalais ir įterpti netikrus valdymo perdavimus bei niekam tinkamą kodą po kiekvieno signalo.

Supainiojimo transformacijos pagal skirtingus supainiojimo principus ir objektus, galima būtų suskirstytos į tokias grupes [5]: schematinis supainiojimas, duomenų supainiojimas, kontrolinio srauto supainiojimas.

2.2.1.1. Schematinis supainiojimas (*Layuot obfuscation*)

Schematinis supainiojimo metodas daugiausia naudoja ištrynimo ir pervadinimo transformacijas. Naudojant šias apsaugos priemenes iš programų yra pašalinama nereikalinga informacija: derinimo (*Debugging*), komentarai, metodai, kurie nenaudojami. Tokio supainiojimo pavyzdys būtų: klasių supainiojimas, kai iš 2 klasių padaroma viena. Kaip atliekamas klasių supainiojimo apsauga[6] yra pateikta 2 paveiksle.

```
public class C1
{
    private int number;
   protected Object o;
    public Ct(){
      number=10;
      o=new Object();
}
   public boolean m(){
      return number <0;
   }
}
public class C2
{
     protected float number;
     public C2(){
       number=5.2;
  }
    public C2(float c){
        number=c;
}
    public void m(){
      number=1.0;
   }
}
```

```
public class Ct
{
   private int number;
   protected Object o;
   protected float number1;
   public Ct(){
     number=10;
     o=new Object();
}
public Ct(int c){
     number1=5.2;
}
public Ct(float c){
      number1=c;
}
public boolean m(){
      return number<0;
}
public void m1(){
   number1=1.0;
  }
}
```

2 pav. Klasių supainiojimo metodas.

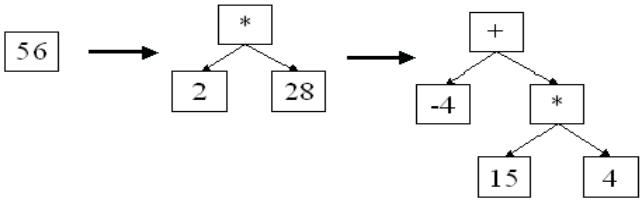
Kaip matome iš 2 paveikslo originalią programą sudaro klasės *C1* ir *C2*, o *Ct* – gaunama atlikus transformaciją. Supainiotoji klasė *Ct* turi tris kintamuosius: *private number, protected Object o* ir *protected float number1*. Kadangi *C1* ir *C2* turėjo kintamuosius tuo pačiu vardu *number*, atliekant transformaciją klasėje *C2* jis buvo pakeistas į *number1*. Toks pats vardų sukeitimas yra atliekamas ir tada, kai klasėse yra metodai su vienodais pavadinimais. Visi klasės metodai yra pervadinami į *m1*, taip pat yra įdedamas papildomas klaidinamas kintamasis *c*.

Atlikus tokią transformaciją programos veikimo algoritmas tampa sunkiau suprantama atakuotojui.

2.2.1.2. Duomenų supainiojimas

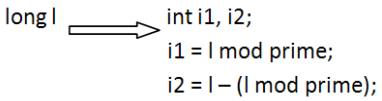
Duomenų supainiojimo metodas pakeičia programų duomenis ir duomenų struktūras [37]. Tokie veiksmai yra atliekami panaudojus: masyvų pertvarkymus, paveldėjimo ryšių pakeitimus, statinių kintamųjų keitimas į procedūrinius, koduojant eilutes, kintamųjų suskaldymą ir t.t. Visus taikomus suskaldymo metodus galima suskirstyti į [6]:

1) Išskaidymo medis. Statiniai kintamieji yra pakeičiami dinaminiais, tam panaudojant išskaidymo medį.



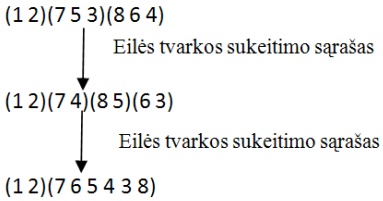
3 pav. Išskaidymų medis.

2) Dalybos su liekana naudojimas. Sudėtinius skaičius, pasinaudojus dalyba su liekana, galima suskaidyti į mažesnius. Kaip tai daroma yra pavaizduota 4 pav., kur kintamasis *prime* yra pirminis skaičius.



4 pav. Dalybos su liekana naudojimas.

3) Eilės tvarkos sukeitimo sąrašai. Sveikųjų skaičių supainiojimui naudojama eilės tvarkos sukeitimo sąrašai.

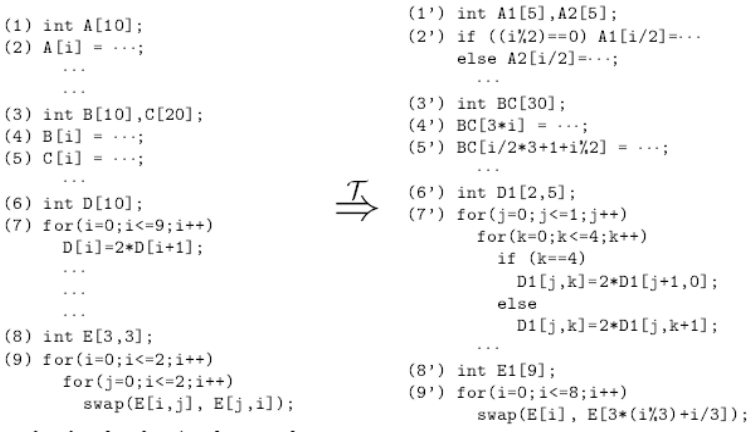


5 pav. Eilės tvarkos sukeitimo sąrašų sudarymas.

4) Loginių operacijų naudojimas. Kintamųjų išskaidymui galima naudoti logines operacijas, tokias, kaip NOT, OR, AND, XOR ir t.t. pvz:

11110000 =11110000 AND 11110000

5) Masyvų pertvarkymas. Masyvą galima išskaidyti į kelis mažesnius masyvus, sujungti du ir daugiau masyvų į vieną, suklastoti masyvus (padidinant arba sumažinant tam tikrus dydžius).



6 pav. Masyvų pertvarkymo metodai.

2.2.1.3. Valdymo srauto supainiojimas

Naudojant valdymo srauto supainiojimo metodą į programas yra įterpiamas srauto supainiojimo šakos, taip pat naudojama valdymo srauto pertvarkymas [25]. Tokios apsaugos pagrindinis tikslas, padaryti programos valdymo srautą kaip įmanoma sunkiau suprantamą, taip apsunkinant programos analizavimą. Tam tikslui gali būti naudojami miglotieji (*Fuzzy*) spėjimai. Valdymo srauto supainiojimo metodai skirstomi į tris kategorijas [7]:

1) Valdymo srauto sukaupimas (*Control aggregation obfuscations*) – pakeičia programos instrukcijų grupavimo tvarką. Įterpimas ir matmenų keitimas yra vieni iš efektyviausių būdų, kuriais painiojami metodai ir metodų kvietimo tvarka.
2) Valdymo srauto tvarkos painiojimas (*Control ordering obfuscations*) – pakeičia instrukcijų kvietimo tvarką. Pavyzdžiui, ciklai gali būti sudaryti taip, kad skaičiavimus atlikinėtų atvirkštine tvarka.
3) Valdymo srauto skaičiavimų painiojimas (*Control computation obfuscations*) – programoje paslepia tikrąjį kontrolės srautą, pvz. į programą yra įterpiamos instrukcijos, kurios nieko nedaro.

2.2.2. Apsauga nuo programų derinimo (*Antidebugging*)

Programų kūrėjai naudoja dinaminio derinimo (*Dynamic debugging*) technologijas klaidų taisymui ir pažeidimų aptikimui programose. Šią technologiją naudoja ir programų „nulaužėjai“, kurie sekdami programos veikimą gali ją „nulaužti“ arba apeiti naudojamus apsaugos metodus.

Apsauga nuo programų derinimo apima metodus ir gudrybes, kaip apsaugoti programinę įrangą, kai yra naudojami derintuvai (*Debugger*) ir apsunkinti atvirkštinę inžineriją [8]. Naudojant šią technologiją bandoma uždrausti naudotis derintuvais. Programos gali būti suprogramuotos taip kad aptiktų techninio, programinio ir vartotojo lygio derintuvus. Kadangi derintuvai yra programos veikiančios darbinėje atmintyje ir paliekančios tam tikrus pėdsakus, pakeičia atitinkamas vėliavėles, jei programa yra derinama. Taikomoji programa apsaugota nuo derinimo, aptikusi derintuvą gali imtis įvairių veiksmų, tokių kaip derintuvo išjungimas. Derintuvų aptikimui naudojami sekantys metodai:

1) Laiko stebėjimas – kiekvieną kartą paleidžiant programą ji palygina du laiko pavyzdžius tarp kelių skirtingų instrukcijų vykdymo. Jei jie labai skiriasi, apsaugota programa nusprendžia, kad yra derinama. Derinant procesą, keletas centrinio procesoriaus ciklų yra skirta atlikti derinimo veiksmus, ėjimui per instrukcijas todėl instrukcijų vykdymo laikai išauga.
2) Išimtinės situacijos [9,21] – yra stebimi visi operacinės sistemos procesai. Jeigu procesas yra derinamas, derintuvas prilyginamas išimtiniai situacijai. Kai tokia situacija atsiranda programa imasi atitinkamų veiksmų.
3) Žinomų derinimo priemonių, programinių langų pavadinimų aptikimas (OLLYDBG skirtas aptikti OllyDbg ar WinDbgFrameClass – WinDbg). Taip pat yra stebimi visi sistemos procesai ir tikrinama ar yra paleista derinimo priemonė (pvz. OLLYDBG.EXE, windbg.EXE).
4) Dauguma derintuvų aptikimo technikų tikrina *BeingDebugged* vėliavėlę, procesų aplinkos bloke (*Process environment block*). Apsaugota programa tikrina vėliavėlės reikšmę taip nustatydama ar procesas yra derinamas vartotojo lygio derintuvo. Be *BeingDebugged* gali būti tikrinama ir *NtGlobalFlag* vėliavėlės reikšmė.
5) Atskaitos taškų (*Breakpoints*) aptikimas. Yra tikrinama ar instrukcijos *INT3* ir *INT1* nebuvo pertrauktos, jei buvo tai reiškia, kad procesas yra derinamas.

2.2.3. Apsauga nuo programinio kodo išskaidymo (*Antidisassembly*)

Programų programinio kodo skaidymas į dalis tai programinio kodo objektų pavertimas į asemblerio kalbą [10]. Toks pavertimas dažniausiai yra atliekamas naudojant linijinio nuskaitymo (*Liner-scanning*) ir rekursijos žingsnių (*Recursion-marching*) technologijas. Naudojant linijinį nuskaitymą, visas programos kodas yra išskaidomas į mašininį kodą, kuris yra paverčiamas į asemblerio instrukcijas. Šio metodo silpnoji vieta yra ta kad sunku atskirti programinį kodą nuo duomenų. Rekursijos žingsnių metodas programą skaido paeiliui ir seka kiekvieną instrukciją kol randa pervertimo instrukcijas (*Transfer instruction*). Šis metodas gali atskirti duomenis nuo kodo. Dažniausiai naudojami apsaugos būdai nuo programinio kodo išskaidymo į dalis yra pateikti žemiau.

2.2. 3.1. Bevertis kodas (*JunkCode*)

Bevertis kodas yra skirstomas į duomenis ir bevertes instrukcijas. Tokios instrukcijos skirtos asemblerio programoms suklaidinti ir turi būti įterpiamos taip, kad būtų išsaugotas programos vientisumas ir jos atrodytų kaip vykdomų instrukcijų dalis ir nepasiekiamos vykdymo metu [11]. Norint tai padaryti pirmiausia reikia išsirikti programos kodo blokus, kuriuose galima įterpti tokias instrukcijas (7 pav.). Pasirinkti blokai turi užtikrinti, kad įterptos instrukcijos būtų nepasiekiamos programos vykdymo metu. Tai padarius reikia nuspręsti kokią bevertę informaciją reikia įterpti, kad kaip įmanoma labiau suklaidinti atakuotoją.

```
      8048000 | 55              push   %ebp
      8048001 | 89 e5           mov    %esp, %ebp
              |
      8048003 | e8 00 00 74 11  call   19788008 <branch fnct>
      8048008 | 0a 05           (junk)
              |
      804800a | 3c 00           cmp    0, %eax
      804800c | 75 06           jne    8048014 <L1>
      804800e | b0 00           mov    0, %eax
      8048010 | eb 07           jmp    8048019 <L2>
      8048012 | 0a 05           (junk)
L1:   8048014 | a1 00 00 74 01   mov    (1740000), %eax
              |
L2:   8048019 | 89 ec           mov    %ebp, %esp
      804801b | 5d              pop    %ebp
      804801c | c3              ret
      804801d | 90              nop
```

7 pav. Beverčio kodo pavyzdys [12].

Du beverčio kodo baitai yra įterpiami 0x8048012 adresu. Toks kodas niekada nebus vykdomas nes jis yra įterpiamas po šuolio instrukcijos 0x8048010. Toks metodas gali patikimai apsaugoti nuo linijinio nuskaitymo.

2.2. 3.2. Valdymo srauto slėpimas (*Control flow hidding*)

Šis metodas yra panašus į aukščiau aprašytą programos valdymo srauto supainiojimą. Dažniausiai yra naudojami du valdymo srauto slėpimo būdai: šuolių lentelės apgaulės (*Jump table cheat*) ir nukreipimo funkcija (*Branch function*). Šuolių lentelės apgaulės metodu yra bandoma padaryti kaip įmanoma sunkiau suprantamą programos veikimą. Tai atliekama nurodant netikrus programos šuolius iš vienos jos dalies į kitą.

Nukreipimo funkcijos technika pakeičia procedūrų kvietimo tvarką. Normalus paprogramės vykdymas yra pakeičiamas nukreipimo funkcija, kuri paprogramės funkcijas vykdo netiesiogiai. Be to yra pridedama papildomų grįžimo adresų. Kai paprogramė yra įvykdyta, kontrolė nėra iškart perduodama adresui, po instrukcijos iškvietimo. Vietoj to, yra kviečiama instrukcija, kuri kompensuoja bitų postūmį.

```
8048000   55                 push  %ebp
8048001   89 e5              mov   %esp, %ebp

8048003   e8 00 00 74 11     call  19788008 <branch fnct>
8048008   0a 05 3c 00 75 06  or    675003c, %al

804800a
804800c
804800e   b0 00              mov   0, %eax
8048010   eb 07              jmp   8048019
8048012   0a 05 a1 00 00 74  or    740000a1, %al
8048014

8048018   01 89 ec 5d c3 90  adc   %ecx, 90c35dec(%ecx)
8048019
804801b
804801c
804801d
```

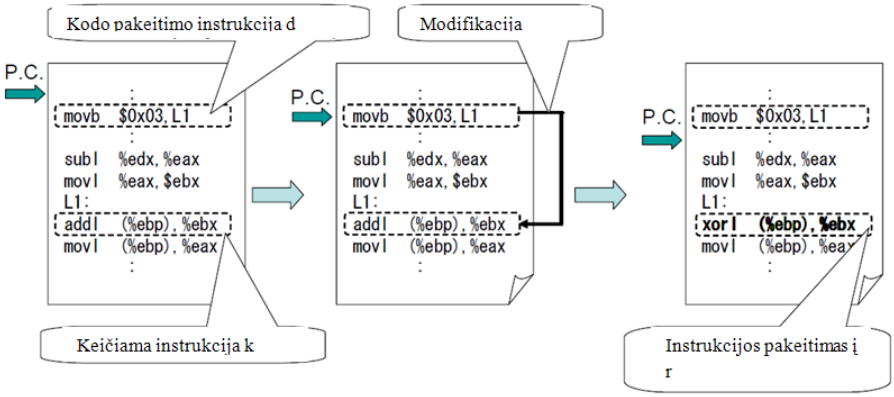
8 pav. Valdymo srauto slėpimas [12].

Kaip matyti 8 pav. adresu 0x8048008 yra įterpti du beverčiai bitai, kaip **or** instrukcija, kurie atlieka **cmp** ir **jne** instrukcijas. Panašiai yra padaryta ir 0x8048012 adresu. Dėl tokių pakeitimų yra įvykdoma 12 instrukcijų iš kurių tik 5 yra teisingos, kas labai apsunkina atakuotojo bandymus panaudoti automatines atvirkštinės inžinerijos priemones ir įveikti taikomosios programos apsaugą.

2.2. 3.3. Kodo šifravimo technologija (*Code encryption technology*)

Kodo šifravimas – programinio kodo šifravimas taikant duomenų šifravimo technologijas. Toks apsaugos mechanizmas labai patikimai apsaugo nuo statinio programų išskaidymo į dalis. Žinomiausias programinio kodo šifravimo metodas yra savarankiškai besikeičiantis kodas (*Self modifying code*). Jis paslepia tikruosius programos kodus, naudodamas niekam naudingas instrukcijas (*Dummy instructions*). Kiekvieną kartą paleidus programą, slepiami programiniai

kodai yra vis kiti. Skaidymo metu bus gaunami klaidų pranešimai, jei bus bandoma vykdyti instrukcijos, kai jos yra šifruotos formos.



9 pav. Savarankiškai besikeičiančio kodo pavyzdys.

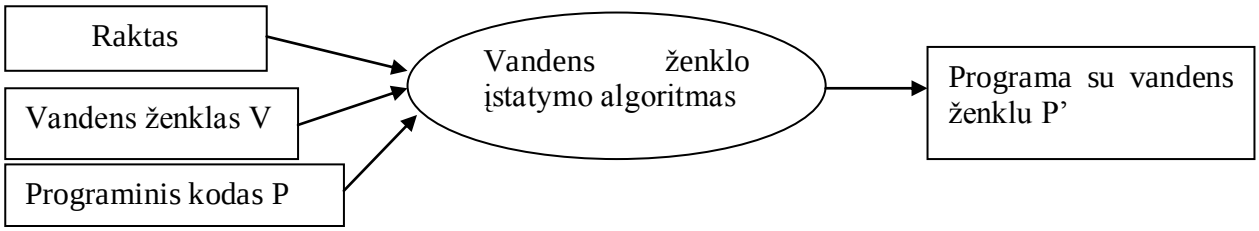
9 pav. pateiktas savarankiškai besikeičiančio kodo pavyzdys, asemblerio programavimo kalboje. P.C. parodo programos skaitliuką. Pavyzdyje instrukcijos „movb $0x03,L1“, „addl (%ebp),%ebx” ir „xorl (%ebp),%ebx”, atitinka d, k, ir r. Instrukcija „addl (%ebp),%ebx“ yra pakeičiama instrukcija „xorl (%ebp),%ebx“, tai yra atliekama programos paleidimo metu vykdant instrukciją „xorl (%ebp),%ebx“. Kaip matome po pakeitimo pakeistoji instrukcija yra visai nepanaši į pradinę [16]. Tokie pakeitimai labai apsunkina atakuotojų galimybės atlikti programos nulaužimą. Geresnė programos apsauga bus gauta, jei šis metodas bus naudojamas su kitais apsaugos būdais.

2.2.4. Programų vandens ženklinimas

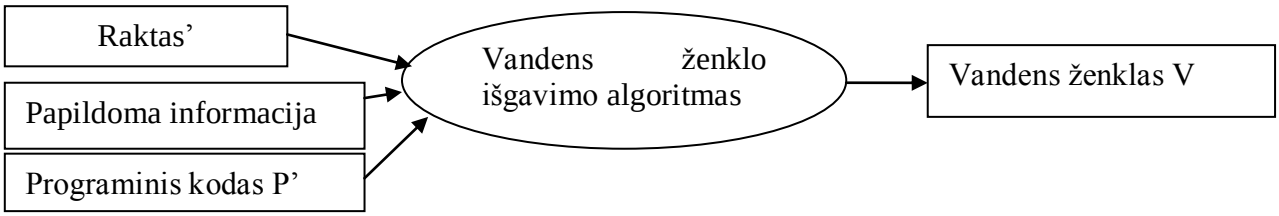
Programų apsaugoje vandens ženklai naudojami įterpti slaptą informaciją apie legalų programos vartotoją ar programos autorinės teises. Tokia programų apsauga gali būti panaudota nelegalios programinės įrangos naudotojų sekimui, atsekti programinio kodo panaudojimą kitose programose, apsaugai nuo nelegalaus programų kopijavimo ir naudojimo.

Taikomųjų programų apsaugai naudojami vandens ženklai yra skirstomi į dvi grupes[4]: statiniai ir dinaminiai. Prie statinių metodų būtų galima priskirti Moskowitz [33] pasiūlytą metodą. Jis siūlo vandens ženklą įdėti pvz. į paveiksliuką naudojant vieną iš laikmeninių vandens ženklinimų. Ir tą paveiksliuką patalpinti į programos statinių duomenų sritį, tačiau toks programos ženklinimas nėra saugus, nes atakuotojas gali lengvai aptikti tokį vandens ženklinimą. Dinaminiai programų vandens ženklai yra patikimesni, nes juos sunkiau aptikti. Dinaminiai

vandens ženklinimo metodai dažniausiai yra paremti grafų ir išplėstinio spektro teorijomis. Bendrai vandens ženklų uždėjimo ir nuėmimo veiksmus galima pavaizduoti, tokiomis schemomis [4]:



10 pav. Vandens ženklo uždėjimas.



11 pav. Vandens ženklo išgavimas.

Panaudojus vandens ženklo išgavimo algoritmą, išgautas ženklas yra tikrinamas ar jis toks, koks turi būti. Jeigu ne, tai galimas daiktas, kad buvo pakeistas programos „nulaužėjų". Tikrinant tokį ženklą galima aptikti nelegalaus programinio kodo panaudojimo atvejus ar nelegalios programinės įrangos plitimo šaltinį.

2.2.4.1. Dinaminiai, taikomųjų programų, vandens ženklinimo metodai

Dinaminiai vandens ženklai, apžvelgti pasirinkti dėl to, kad užtikrina patikimesnę apsaugą nei statiniai, nes yra sunkiau aptinkami. Visi dinaminiai vandens ženklinimo metodai, taikomi taikomųjų programų apsaugai yra skirstomi į tris grupes: Velykų kiaušinio (*Easter egg*), duomenų struktūrų ir vykdymo eigos (*Execution trace*).

1) **Velykų kiaušinis** – programinis kodas, kuris aktyvuojamas suvedus nenumatytas reikšmes ar atlikus netinkamus veiksmus. Kai taip nutinka vartotojui yra parodomas pranešimas, paprastai tai yra informacija apie autorines teises. Pagrindinė problema su Velykų kiaušinio vandens ženklinimu yra tai, kad jis lengvai randamas. Yra net kelios tokių vandens ženklų radimo svetainės, kuriose aprašyta kaip ir kur reikia/galima rasti vandens ženklą.

2) **Duomenų struktūrų** – slapta informacija yra įdedama į programą pasinaudojant: krūva, steko duomenimis. Vandens ženklas yra išgaunamas tikrinant einamuosius programos kintamuosius. Tai atliekama pasinaudojus ženklo išgavimo funkcijomis. Kadangi naudojant šį metodą nėra atliekamas joks išvedimas, tokį vandens ženklą yra sunkiau aptikti nei Velykų kiaušinio.
3) **Vykdymo eigos** – vandens ženklas yra uždedamas vykdant programą paeiliui, naudojant tam skirtą įvesties kintamąjį. Išgavimas – stebima kai kurie nustatyti adresai ir/arba operacijų vykdymo tvarka.

2.2.4.2. Atakos prieš vandens ženklus

Dėl to, kad taikomųjų programų vandens ženklinimas nepadaro programos kodo sunkiau suprantamo naudojant atvirkštinės inžinerijos priemones, o tik įterpia slaptą informaciją į programą, prieš vandens ženklinimą yra naudojamos specifinės atakos [34]:

1) Atėmimo ataka (*Subtractive attack*) – „nulaužėjas“ aptikęs vandens ženklo buvimo vietą, gali bandyti pašalinti jį iš programos. Jei ši ataka pavyksta, gaunama programa, kuri vis dar pakankamai tiksliai atitinka savo pradinį variantą ir veikia taisyklingai.
2) Iškraipymo ataka (*Distortive attack)* – jeigu „nulaužėjas“ negali aptikti vandens ženklo ir yra pasirengęs susitaikyti su kai kuriais programos kokybės blogėjimo atvejais, gali vykdyti įvairias iškreipiančias transformacijas, apimančias visą programą tokiu būdu išgadinant vandens ženklą. Įvykdžius tokią iškraipančią ataką, nebus galima aptikti iškraipyto vandens ženklo ir įrodyti teisių į programą. O „nulaužėjas“ turės veikiančią programą, su tam tikrais jos kokybės pablogėjimais.
3) Pridedamoji ataka (*Additive attack*) – „nulaužėjas“ gali papildyti programą savo sugalvotu vandens ženklu (arba keletu tokių ženklų). Veiksmingas tokios atakos atlikimo rezultatas, „nulaužėjo“ vandens ženklas paslepia originalų ženklą, taip kad jo būtų neįmanoma išgauti arba neįmanoma nustatyti, kad originalus vandens ženklas buvo uždėtas pirma „nulaužėjo“.

2.2.5. Programinio kodo glaudinimas ar šifravimas

Naudojant šią technologiją programos kodo dalis, atskiras modulis (pvz.: biblioteka) ar visas vykdomasis failas yra suspaudžiami arba užkoduojami, taip pakeistas kodas ar kitos programos dalys – susiejama su atstatymo veiksmais. Suglaudintos ar užšifruotos programos paleidimo metu

atstatymo veiksmai, atstato pradinius programos kodus ar duomenis ir juos įvykdo.

Originaliai glaudinimas buvo naudojamas sumažinti vykdomojo failo dydį. Šiandien glaudinimas yra naudojamas kaip apsauga nuo atvirkštinės inžinerijos, nes prieš suspaustą failą negali būti panaudotos atvirkštinės inžinerijos priemonės ir failas paverčiamas į asemblerio kalbą. Pirmiausia tokį suspaustą kodą reikia išspausti, suspaustas failas gali būti apsaugomas raktu. Toks suspaustas kodas gali būti laikomas savaime išsispaudžiančiame (*Selft-extracting*) archyve, kur duomenys yra supakuojami kartu su atitinkamu išspaudimo raktu. Keletas suspaudimo programų: ASPack, UPX, PECompact2.

Panašiai yra naudojama ir šifravimas. Šiuo atveju programos kodas ar tam tikros jos dalys yra saugomos šifruotoje formoje ir iššifruojamos jas paleidžiant. O šifravimo/iššifravimo raktas būtų saugomas programoje arba sugeneruojamas kiekvieną kartą paleidžiant šifruotą programos dalį. Šifravimo/iššifravimo metodui yra reikalinga daugiau skaičiavimo resursų, nei pakavimui.

Kaip parodė atliktas tyrimas [26], paleidimo failas, suspaustas Secure UPX programa pasileido beveik taip pat greitai kaip ir originalus, o panaudojus kriptografinį AES suspaudimo algoritmą, paleidimas užtruko iki 20 kartų ilgiau. Bandymo rezultatai pateikti 1 lentelėje.

1 lentelė. Programų paleidimo laikai milisekundėmis.

Paleista programa	Originalus paleidimo failas	Suspaustas su Secure UPX	Suspaustas su AES
MD5	55,450	59,836	1139,100
AES	50,730	53,786	853,700

Tokie rezultatai buvo gauti dėl to, kad AES išskleidimui reikia atlikti nemažai papildomų dešifravimo operacijų. Taip pat reikia atsižvelgti į tai, kad greitaveika buvo tiriama įterptinėse sistemos, kur skaičiavimo resursai yra riboti. Tokio žymaus laiko padidėjimo neturėtų būti sistemose su didesniais skaičiavimo resursais, tokiose kaip šiuolaikiniai personaliniai kompiuteriai.

Vien programos kodo suspaudimas ar užšifravimas apsunkina statinę atvirkštinę inžineriją. Kadangi toks kodas negali būti tiesiogiai išskaidytas į asemblerio kalbą. Be to šią technologiją galima lengvai panaudoti kartu su aukščiau aprašytomis apsaugos priemonėmis, todėl panaudojus įvairias jų kombinacijas galima pasiekti gerų rezultatų kovoje prieš atvirkštinę inžineriją. Tačiau toks apsaugos metodas, kad ir papildytas kitais, turi problemų – kaip generuoti ir kur saugoti reikalingus raktus.

2.2.5.1. Raktų generavimas

Raktų generavimas yra sunkus uždavinys. Dažniausiai raktai generuojami naudojant įvairius pseudoatsitiktinius generatorius, kurie esant tiems patiems įėjimo duomenims pateikia tuos pačius išėjimo duomenis [27]. Pseudoatsitiktiniai generatoriai gali būti tiek programiniai, tiek aparatūriniai. Nesvarbu koks generatorius ar programinis, ar aparatūrinis – tai yra sudėtinga sistema, kuri turi generuoti determinstines sekas su statistinėmis ypatybėmis kaip įmanoma artimesnėms į atsitiktines[28]. Sukurti generatorių, kuris tenkintų šiuos reikalavimus yra sudėtingas ir daug laiko reikalaujat procesas.

Be pseudoatsitiktinių generatorių yra pasiūlymų raktus generuoti panaudojant elektroninių prietaisų fizikines savybes, pvz.: sistemos įrenginių veikimo ar realaus laiko laikrodžio dažnius. Raktus galima sudaryti ir iš ROM, RAM, SRAM ir kitų atminties modulių gamybos ypatumų, kadangi gamybos proceso metu nėra gaunama visiškai vienodi atminties modulio tranzistoriai, taip yra dėl silicio savybių [29, 30]. Pastarieji metodai dažniausiai yra naudojami įterptinėse sistemose. Naudojant papildomą aparatūrinę įrangą raktus galima generuoti iš tam tikrų žmogaus fiziologių savybių – pirštų atspaudai, veido bruožai, akies rainelė ir t.t. Tačiau tokiam generavimui reikalinga aparatūrinė įranga, kuri yra pakankamai brangi [31].

2.3. Aparatūriniai apsaugos metodai

Be programinių taikomųjų programų apsaugos būdų yra ir aparatūriniai. Aparatūriniai apsaugos metodai, kurie taikomi programų apsaugai, yra kompiuterinių sistemų apsaugos objektai, kurie priskiriami techninėms apsaugos priemonėms ir apriboja prieigą prie taikomosios programos. Vartotojas norėdamas pasinaudoti programa turi turėti apsaugai priskirtą aparatinį įrenginį. Aparatūriniai metodai gali žymiai padidinti saugumo lygį ne linijiniu būdų, nes tai išorinis įtaisas, kurio saugumo lygį nustato programinės įrangos tiekėjas, bet ne galutinis vartotojas [16]. Tačiau tokie apsaugos mechanizmai yra palyginti brangūs ir taikomi tik tokioms programoms, kurios turi didelę komercinę vertę.

2.3.1. Apsaugos raktai

Dabar labiausia tyrinėjami aparatūriniai apsaugos metodai yra paremti apsaugos raktais. Apsaugos raktas – aparatūrinis įrenginys, kuris jungiamas prie kompiuterio, kad pasileistų programa ir be kurio programa iš vis nepasileistų arba prarastų dalį funkcionalumų. Pats apsaugos raktas yra panašus į USB (*Universal serial bus*) raktą, prie kompiuterio jungiamas

naudojant LPT, USB, RS232C, PCMCIA prievadus. Pats paprasčiausias apsaugos raktas saugo programos identifikacinį numerį, kuris yra patikrinamas prieš programos paleidimą. Sudėtingesni apsaugos raktai susideda iš: simetrinės kriptografijos variklio (*Engine*) ir simetrinių raktų saugyklos; nuolatinės atminties, kurią programa gali skaityti ir rašyti; unikalaus serijinio rakto; unikalaus nepriklausomo programinės įrangos tiekėjo identifikacinio skaičiaus. Tokiuose raktuose jau galima atlikinėti funkcijų reikšmių skaičiavimus, daryti šifravimus/iššifravimus.

Sudėtingesniuose apsaugos raktuose galima saugoti dalį programos ar norimą programos algoritmą. Tokie raktai yra paremti šifravimo procesoriaus veikimu (*Cryptoprocessor*). Jo pagalba programos paleidimo metu, jame saugoma informacija yra atkoduojama ir leidžiama dirbti su ja.

Yra programinės įrangos gamintojų, kurie vietoj apsaugos rakto naudoja įprastus USB raktus. Kurie atlieka tuos pačius veiksmus kaip ir apsaugos raktai. Rakte saugoma informacija yra užšifruota. Tačiau ne visų gamintojų USB raktai tinka, kaip apsaugos rakto pakaitalas, nes gamintojas jiems nepriskiria unikalaus identifikacinio numerio.

Vienas iš tokių pasiūlymų yra aprašytas [17] straipsnyje. Autoriai pateikia kaip reikia „išmokinti“ bendrauti programą ir USB raktą. Pirmiausia, kad programa galėtų bendrauti su raktu, vartotojas suveda savo PIN kodą. Jeigu kodas teisingas tai programa ir raktas apsikeičia tam tikrais duomenimis, kurių pagalba yra patvirtinama vartotojo teisė naudotis programa.

2.4. Analizės išvados

1. Atlikus programinių apsaugos metodų analizę išsiaiškinta, kad didžiausią grėsmę taikomųjų programų saugumui kelia atvirkštinė inžinerija.
2. Atlikus analizę išsiaiškinta, kad programinio kodo supainiojimo ir programinio kodo glaudinimo ar šifravimo metodai pasižymi gera apsauga nuo atvirkštinės inžinerijos, tačiau jie reikalauja daugiau sistemos išteklių, o apsauga nuo derinimo ir programinio kodo išskaidymo, pasižymi prastesne apsauga, bet reikalauja mažiau sistemos išteklių.
3. Programų vandens ženklinimas neapsaugo programos nuo atvirkštinės inžinerijos, o tik įterpia informaciją apie autorines teises.
4. Aparatūriniai apsaugos metodai dėl savo kainos yra naudojami rečiau.
5. Analizė parodė, kad nei vienas iš nagrinėtų metodų neužtikrina visiškos apsaugos, todėl dažniausiai yra naudojami keli apsaugos metodai vienu metu. Lengviausiai su kitais apsaugos metodais yra suderinamas programos glaudinimo ar šifravimo apsaugos

metodas. Panaudojus šį ir kitų apsaugos metodų įvairius variantus galima pasiekti gerų rezultatų kovoje prieš atvirkštinę inžineriją. Tačiau programos glaudinimo ar šifravimo apsaugos metodas turi problemų – kaip generuoti ir kur saugoti reikalingus raktus.

2.5. Uždaviniai

Atlikus taikomųjų programų apsaugos metodų analizę, buvo iškelti uždaviniai, pasiūlyti apsaugos metodą paremtą atskirų programos modulių šifravimu, nes šis metodas patikimai apsaugo nuo atvirkštinės inžinerijos ir gali būti naudojamas su kitais apsaugos metodais. Pasiūlytas apsaugos metodas nereikalautų papildomų išorinių prietaisų ir infrastruktūrų raktų generavimui, saugojimui ir valdymui. Programos kodas saugomas šifruotoje formoje, šifravimo raktai generuojami realiame laike, pagal poreikį, prieš vykdant šifruotą programos modulį.

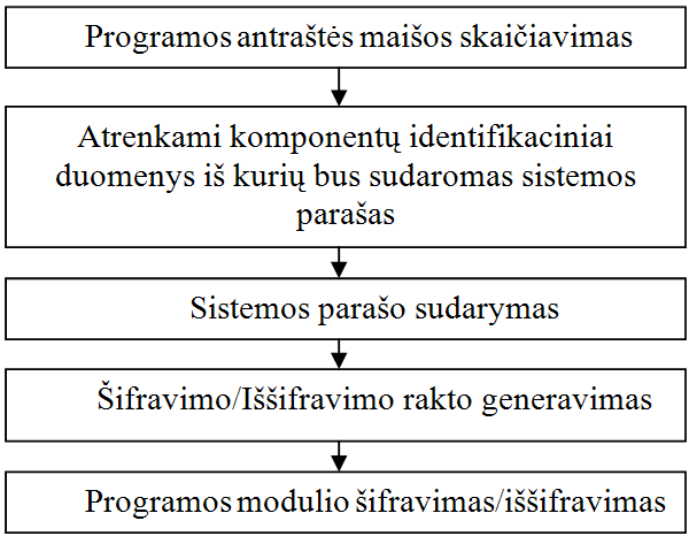
Sumodeliavus apsaugos metodą, bus tiriama metodo generuojamų šifravimo raktų stiprumas. Bus tiriama kiek laiko užtrunka šifravimo raktų generavimas siūlomu metodu, kokią įtaką programos veikimui (kiek pailgės programos vykdymas) turės šifruotų modulių naudojimas, bei naudojamas šifravimo algoritmas.

3. PROGRAMŲ APSAUGOS METODAS, NAUDOJANT SISTEMOS PARAŠĄ

Atlikus taikomųjų programų apsaugos metodų analizę, nustatyta, kad programos glaudinimas ir/ar šifravimas patikimai apsaugo nuo statinės atvirkštinės inžinerijos. Nes nulaužėjui, pirmiausia reikia išskleisti ar iššifruoti programą, šifravimo atveju norint atlikti tokius veiksmus reikia žinoti ir iššifravimo raktą. Siūlomas programinis apsaugos modelis, paremtas programos modulių šifravimu, kur šifravimo raktas generuojamas kiekvieną kartą paleidžiant šifruotą modulį, kuris užkraunamas į atmintį ir po naudojimo ištrinamas.

3.1. Apsaugos metodas

Siūlomas apsaugos metodas yra pagrįstas atskirų programos modulių šifravimu. Programos modulių apsaugos raktas generuojamas pagal apsaugomos programos antraštės, sistemos aparatinių ir programinių komponentų: CPU, RAM, ROM, BIOS, OS, ir t.t., identifikacinius duomenis: modelis, gamintojas, serijinis numeris ir t.t., naudojant logines komandas ir maišos funkcijas. Tokiu būdu sudarant ***sistemos parašą*** (*SP*). Šifravimui yra naudojami simetriniai algoritmai, nes jie greitesnis už asimetrinius. Šifravimo algoritmas parenkamas atsižvelgiant į jo greitaveiką ir reikiamą programos apsaugos lygį.



12 pav. Kuriamo modelio schema.

Apsaugos modelis paremtas tuo, kad ne viskas taikomosiose programose turi būti apsaugoma, tai gali būti programos antraštės, tam tikros bibliotekas, nesvarbūs moduliai. Iš 12 pav. matyti, kad iš neapsaugotų programos dalių, metode yra naudojama programos antraštė(-ės). Pirmiausia yra suskaičiuojama programos antraštės(-čių) maišos reikšmė(-ės). Naudojantis šia

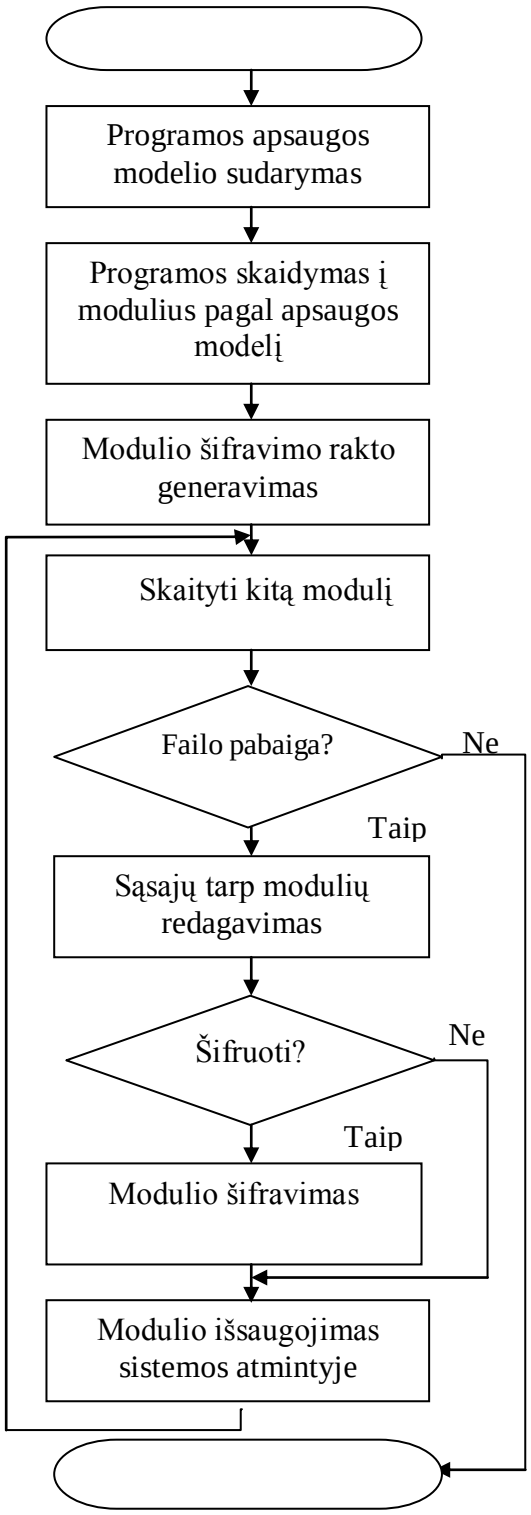
maišos reikšme bus atrenkami (pseudoatsitiktinai) sistemos komponentų identifikaciniai duomenys, kurie sudarys sistemos parašą. Sistemos parašas bus sudaromas, atrinktus komponentų duomenis apdorojant, apdorojimui naudojant logines operacijas ar jų įvairius variantus. Iš tokiu būdu gauto sistemos parašo bus sudaromas šifravimo/iššifravimo raktas, suskaičiuojant parašo maišos reikšmę. Maišos skaičiavimas naudojamas norint gauti fiksuoto ilgio raktus, nes sistemos parašo ilgis priklauso nuo atrinktų komponentų identifikacinių duomenų ilgio. Priklausomai nuo šifravimo algoritmo ir naudojamo rakto ilgio yra naudojama ir atitinkama maišos funkcija (MD5, SHA, SHA-2 ir t.t.). Gavus šifravimo raktą yra vykdomas programos modulio šifravimas arba iššifravimas.

3.1.1. Apsaugos uždėjimas

Apsaugos metodu, programos moduliai būtų apsaugomi diegimo į sistemą metu. Tam reiktų naudoti specialų programinės įrangos įdiegėją (*Installer*)[40]. Pagrindiniai įdiegėjo procesai:

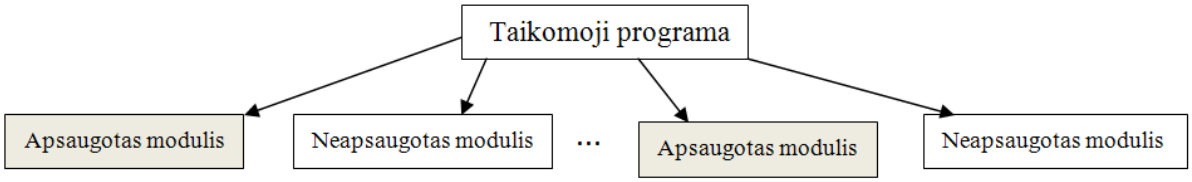
- Programos apsaugos modelio sudarymas. Modelio sudarymas – apsaugos parametrų nurodymas. Nurodoma: maišos ir šifravimo algoritmai, rakto ilgis, kiek (1,2,...,n) sistemos komponentų duomenų sudarys sistemos parašą, saugomų modulių pavadinimai. Priklausomai nuo programinės įrangos įdiegėjo, apsaugos parametrai gali būti nurodomi specialiame faile arba įdiegimo proceso pradžioje.
- Programos skaidymas į modulius pagal apsaugos modelį. Šio proceso metu yra nustatomi moduliai, kuriuos reikia apsaugoti, kurių antraščių maišos reikšmės turi būti skaičiuojamos.
- Šifravimo rakto skaičiavimas.
- Programos modulio šifravimas.
- Sąsajų tarp modulių redagavimas ir modulių išsaugojimas sistemos atmintyje.

Programinės įrangos įdiegėjo schematinis veikimas pateiktas 13 pav.:



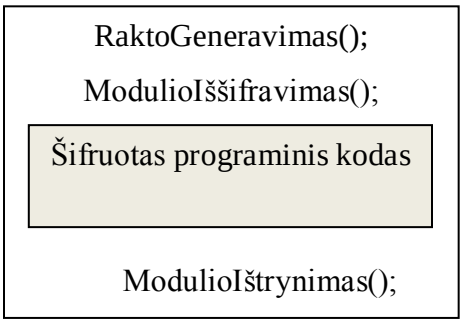
13pav. Programinės įrangos įdiegėjas.

Po apsaugos uždėjimo, gauname programos struktūrą 14 pav., kuri yra sudaryta iš šifruotų ir nešifruotų modulių. Moduliai tarpusavyje susieti.



14 pav. Apsaugotos taikomosios programos struktūra.

Apsaugotas modulis atrodytų taip:

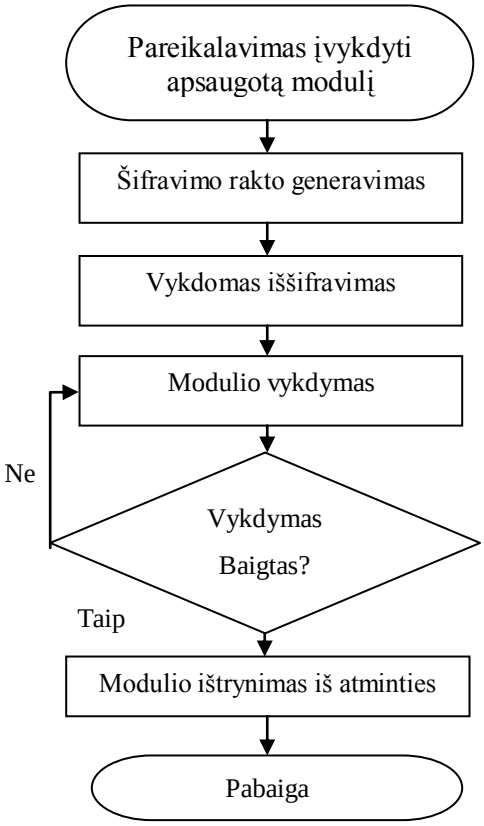


15 pav. Apsaugoto modulio struktūra.

Programos vykdymo metu norint įvykdyti apsaugotą modulį, pirmiausia vykdomi funkciniai kreipiniai: rakto generavimas ir iššifravimas. Be šių funkcinių kreipinių, apsaugotame modulyje yra dar vienas: įvykdyto molio ištrynimas iš sistemos atminties. Šie funkciniai kreipiniai yra visuose apsaugotuose moduliuose.

3.1.2. Apsaugoto modulio vykdymas

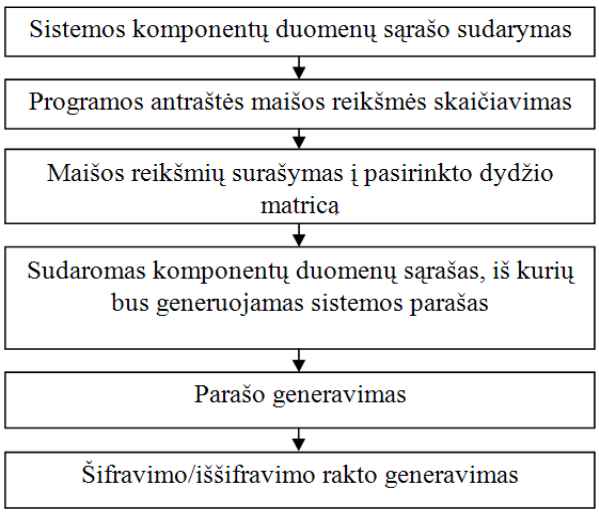
Programos vykdymo metu pareikalavus įvykdyti apsaugotą modulį, pirmiausia sugeneruojamas iššifravimo raktas. Iššifravimo rakto generavimas yra pateiktas 3.1.3 skyriuje. Sugeneravus raktą vykdomas šifruoto programos modulio iššifravimas. Po to modulis yra įvykdomas, jeigu modulio daugiau nebereikia vykdyti, iššifruotas modulis ištrinamas iš sistemos atminties. Schematiškai apsaugoto modulio vykdymas pateiktas 16 pav.



16 pav. Apsaugoto modulio vykdymas.

3.1.3. Raktų generavimas

Buvo minėta, kad programos modulių šifravimo/iššifravimo raktų generavimas susietas su sistemą sudarančių komponentų duomenimis. Šie duomenys parenkami pseudoatsitiktinai, naudojant apsaugomos programos antraštės maišos reikšmę. Raktų generavimo algoritmas pateiktas 17 paveiksle[39].



17 pav.Rakto generavimo schema.

Slaptų raktų generavimo procesas susideda iš 6 žingsnių, kurie paaiškinti žemiau. Proceso aprašymui, pateiksime pavyzdį, kai sistemos parašas bus sudaromas iš 4 komponentų identifikacinių duomenų, juos apdorojant XOR logine operacija. Sistemos parašą sudarančių komponentų duomenų skaičius gali būti koks nori taip pat komponentų duomenys gali kartotis, sudarant sistemos parašą. Pirmame – ketvirtame žingsniuose nustatoma kokių sistemos komponentų parašai bus naudojami slaptojo šifravimo rakto generavimui. Penktame – šeštame žingsniuose apskaičiuojamas raktas. Slapto rakto generavimo procesas:

1) Sudaromas sistemos komponentų duomenų sąrašas:

$$KDS = \{KDS_i\},\ i=,\ldots,n;$$

2) Suskaičiuojama apsaugomos programos antraštės maišos reikšmė:

$$PH = h(PA),$$

kur *PH* gauta maišos reikšmė, *h()* – maišos funkcija, *PA* – programos antraštė.

3) Priklausomai nuo to kiek sistemos komponentų duomenų naudojama sistemos parašo sudarymui, sudaroma atitinkamo dydžio matrica. Kadangi parašo sudarymui naudosime 4 komponentų duomenis, sudarysime matricą 4x4. Ji užpildoma 2 žingsnyje gautos maišos reikšmėmis. Tarkime, kad maišos reikšmė buvo gauta tokio *PH= 8eeba89186027b5630c3b122d8cf746f* (čia ir toliau skaičiai pateikti šešioliktainėje skaičiavimo sistemoje), šiuo atveju maišos reikšmė buvo gauta naudojant MD5 algoritmą, tada užpildyta matrica atrodys taip:

8	e	e	b
a	8	9	1
8	6	0	2
7	b	5	6

Matrica buvo užpildyta pirmomis 16 maišos funkcijos reikšmėmis ir pildymas buvo atliekamas eilutėmis. Siūlomas metodas neriboja kaip turi būti užpildyta matrica. Pvz. tai gali būti padaryta imant paskutines maišos reikšmes ar imant kas antrą ir jas surašant stulpeliais ar eilutėmis.

4) Matricos reikšmes sudedame. Sudėti galima stulpeliais arba eilutėmis. Sudėjus stulpeliais gauname:

8	e	e	b
a	8	9	1
8	6	0	2
7	b	5	6
21	27	1c	14

Gautas sumas padaliname moduliu (%) iš pasirinkto skaičiaus. Šis skaičius gali būti pasirenkamas ar sugeneruojamas pagal poreikį. Pavyzdžio atveju tegu šis skaičius bus *8*, tai yra pirma maišos reikšmė. Atlikus veiksmus gauname:

$$21 \,\%\, 8 = 1$$

$$27 \,\%\, 8 = 7$$

$$1c \,\%\, 8 = 4$$

$$14 \,\%\, 8 = 4$$

5) Ketvirtame žingsnyje gavome, kad sistemos parašas bus sudaromas iš komponentų parašų, kurie *KDS* sąraše yra įrašytos 1,7 ir 4 numeriu. Nustačius komponentų parašus, reikalingus sudaryti sistemos parašą, juos apdorojame pasirinktomis loginėmis operacijomis ar pasirinktu jų išdėstymo variantu. Sistemos parašas *SP* bus:

$$SP = KSS_1 \oplus KSS_7 \oplus KSS_4 \oplus KSS_4;$$

6) Galutinis šifravimo raktas *GSR* yra gaunamas sistemos parašui suskaičiuojant maišos reikšmę:

$$GSR = h(SP);$$

Sugeneruotas raktas GSR naudojamas apsaugomos programos modulių šifravimui ir iššifravimui programos vykdymo metu. Gautas raktas gali būti sustiprintas naudojant rantų stiprinimo metodus [32].

Priklausomai nuo naudojamo šifravimo algoritmo (3 lentelė) gautas raktas gali būti per trumpas arba per ilgas. Todėl esant reikalui jis gali būti pailgintas ar sutrupintas. Vienas iš būdų kaip tai galima padaryti pateiktas 2 lentelėje. 2 letelėje pateikti galimi raktų ilginimo ar trumpinimo metodai tinka tuo atveju, kai sugeneruotas šifravimo/iššifavimo raktas yra 128 bitų ilgio.

2 lentele. Reikia ilgio rakto gavimas.

Reikiamas rakto ilgis, bitais	Gavimas
64	Paimti tik 16 reikšmių pradinio rakto
80	Paimti tik 20 reikšmių pradinio rakto
128	-
192	Prie gauto rakto 32 reikšmių pridėti to pačio rakto 16 reikšmių
256	Raktą pakartoti du kartus

3.1.4. Sistemos parašo sudarymui naudojamos funkcijos

Sistemos parašui sudaryti iš sistemą sudarančių komponentų duomenų, naudosime logines operacijas XOR ir OR, bei įvairius jų variantus. Sistemos parašo sudarymui yra siūlomos 4 funkcijos. Funkcijų aprašymas ir formalus jų aprašymas pateiktas žemiau:

1. Funkcija ***parasas1().*** Komponentų duomenų baitams apdoroti naudojant XOR operaciją:

```
input: KDS, m // m komponentų sąrašas
output: SP // sistemos parašas
l = maxlength KDS // ilgiausias komponento duomuo
for j = 1 to l do
  SP (j) = KDS (1, j)
end for
for i = 2 to m do
  for j = 1 to l do
    SP (j) = SP (j) XOR KDS (i, j)
  end for
end for
```

18 pav. formalus funkcijos paprasas1() aprašymas.

2. Funkcija ***paprasas2()***. Komponentų duomenų apdorojimui naudojant OR operaciją:

```
input: KDS, m // m komponentų sąrašas
output: SP // sistemos parašas
l = maxlength KDS // ilgiausias komponento duomuo
for j = 1 to l do
  SP (j) = KDS (1, j)
end for
for i = 2 to m do
  for j = 1 to l do
    SP (j) = SP (j) OR KDS (i, j)
  end for
end for
```

19 pav. formalus funkcijos paprasas2() aprašymas.

3. Funkcija ***paprasas3()***. Komponentų duomenys apdorojami naudojat XOR arba OR operacijas pakaitom. Kas antras duomuo apdorojamas XOR arba OR operacija, pradedant XOR:

```
input: KDS, m // m komponentų sąrašas
output: SP // sistemos parašas
l = maxlength KDS // ilgiausias komponento duomuo
for j = 1 to l do
  SP (j) = KDS (1, j)
end for
for i = 2 to m-1 step 2 do
  for j = 1 to l do
    SP (j) = SP (j) XOR KDS (i, j)
  end for
  for j = 1 to l do
    SP (j) = SP (j) OR KDS (i+1, j)
  end for
end for
```

20 pav. formalus funkcijos paprasas3() aprašymas.

4. Funkcija ***parasas4()***. Komponentų duomenys apdorojami naudojat XOR arba OR operacijas pakaitom. Kas antras komponento duomens baitas apdorojams su sekančiu baitu naudojant XOR arba OR operacijas. Pirmi baitai sudedami XOR operacija:

```
input: KDS, m // m komponentų sąrašas
output: SP // sistemos parašas
l = maxlength KDS // ilgiausias komponento duomuo
for j = 1 to l do
  SP (j) = KDS (1, j)
end for
for i = 2 to m do
  for j = 1 to l-1 step 2 do
    SP (j) = SP (j) XOR KDS (i, j)
    SP (j+1) = SP (j+1) OR KDS (i, j+1)
  end for
 end for
```

21 pav. formalus funkcijos paprasas4() aprašymas.

Aukščiau pateikti funkcijų pavadinimai: *parasas1()*, *parasas2()*, *parasas3()*, *parasas4()*, bus naudojami ir tolimesniame darbo aprašyme.

3.1.5. Šifravimas/iššifravimas

Programos modulių šifravimui naudojami simetriniai šifravimo algoritmai. Simetriniai algoritmai naudojami dėl to, kad jų greitaveika yra didesnė, kuri kartais gali siekti ir iki 1000 kartų. Galimi naudoti algoritmai yra pateikti 3 lentelėje.

3 lentelė. Galimi naudoti šifravimo algoritmai ir jų raktų ilgiai.

Eil. Nr.	Algoritmas	Rakto dydis (bitais)
1	AES(RijnDael)	128, 192, 256
2	DES	64
3	TripleDES	192
4	RC6	128, 192, 256
5	MISTY1	128
6	IDEA	128

3.2. Išvados

1. Sumodeliuotas taikomųjų programų apsaugos metodas, kuris pagrįstas programos modulių šifravimu. Apsaugą uždedant programos diegimo į sistemą metu.
2. Šifravimo/iššifravimo rakto generavimui naudojant sistemos parašą ir maišos algoritmus. Sistemos parašo sudarymas paremtas [20] straipsniu, kur bendravimui internete siūloma naudoti genetinį kompiuterio kodą, sudarytą iš 7 fiksuotų kompiuterio komponentų

duomenų. Sumodeliuotame metode komponentų skaičius nėra fiksuotas ir komponentų identifikaciniai duomenys parenkami speudoatsitiktinai.

3. Sumodeliuotame apsaugos metode yra įgyvendinti po taikomųjų programų apsaugo metodų analizės iškelti uždaviniai: šifravimo/iššifravimo raktų generavimui, saugojimui ir valdymui nenaudoti jokių papildomų išorinių įrenginių ir infrastruktūrų.

4. SISTEMOS PARAŠU PAGRĮSTO PROGRAMŲ APSAUGOS METODO EKSPERIMENTINIS TYRIMAS

Remiantis sumodeliuotu apsaugos metodu buvo sukurtas eksperimentinis taikomųjų programų apsaugos metodo prototipas. Apsaugos metodo prototipas parašytas naudojant C programavimo kalbą ir OpenSSL biblioteką. Naudojantis šiuo prototipu buvo tiriama: kokias logines operacijas ar jų variantus naudoti, sudarant sistemos parašą, sistemos parašų ir šifravimo raktų stiprumas, stiprumą vertinant entropija [38], kur dižiausia galima reikšmė lygi 1. Kuo gautos reikšmės artimesnės 1 tuo geriau. Taip pat buvo tiriamas šifravimo rakto generavimo laikas, bei apsaugoto modulio vykdymo laikas. Eksperimentai buvo vykdomi su personaliniu kompiuteriu, kurio modelis Dell Inspiron 1520 turintis tokius parametrus: Intel Core 2 Dou T7250 (2 GHz), 2 GB RAM, Windows XP operacinė sistema.

4.1. Sistemos parašo sudarymas

Darant eksperimentą buvo tikrinama, kokią įtaka sistemos parašo stiprumui turi loginės operacijos XOR ir OR, bei įvairūs jų variantai ir ar tinka sistemos parašai naudoti kaip šifravimo/iššifravimo raktai. Eksperimente buvo naudojamos 3.1.4 skyriuje aprašytos sistemos parašų sudarymo funkcijos. Norint kuo geriau atkartoti realias sąlygas, programų antraštės ir sistemos komponentų duomenys buvo generuojami naudojant programinius atsitiktinius eilučių ir skaičių generatorius. Eksperimentui atlikti buvo sugeneruoti 7 pseudoatsitiktiniai komponentų duomenys, kurie buvo naudojami sistemos parašui sudaryti. Naudojant sugeneruotas antraštes, sistemos parašai buvo sudaromi naudojant nuo 2-jų iki 7-nių komponentų duomenų. Kiekviename intervale buvo atliekama po 20 skaičiavimų su kiekviena sistemos parašui sudaryti naudojama funkcija, kiekvieną kartą sugeneruojant naują programos antraštę, kurią panaudojus buvo atrenkami sitemos parašą sudarantys komponentų duomenys. Sitemos parašų sudarymui, šią antraštę naudojant su kiekviena 3.1.4 skyriuje aprašyta funkcija. Tokiu būdu gavome, kad naudojant skirtingas sistemos parašų sudarymo funkcijas, sistemos parašai buvo sudaromi iš tų pačių komponentų duomenų.

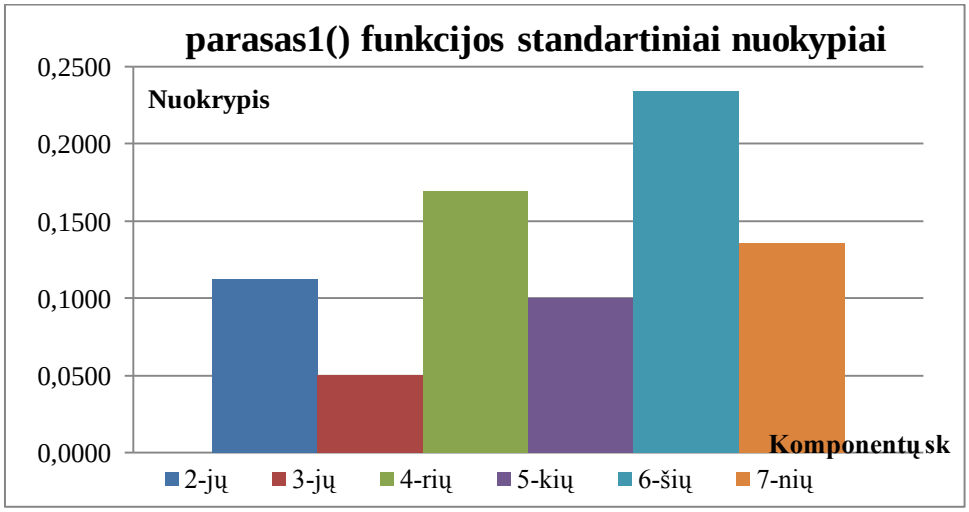
Sugeneruotų sistemos parašų entropijos ir ilgiai pateikti priede 8.1, 1-4 lentelėse. Sistemos parašo ilgis priklauso nuo pseudoatsitiktinai parinktų komponentų duomenų ilgių, parašo ilgis bus toks koks yra ilgiausias komponento duomuo bitais. Pagal eksperimentui sugeneruotus komponentų duomenis, ilgiausias sistemos parašas gali būti 256 bitų.

Gauti duomenys buvo apdoroti, naudojant standartinį nuokrypį. Siekiant parodyti kaip gauti rezultatai pasiskirstę apie vidurkį. Kuo standartinis nuokrypis mažesnis tuo reikšmes artimesnės vidurkiui, bei viena kitai.

4 lentelėje pateikta funkcijos *parasas1()* standartiniai nuokrypiai, priklausomai nuo sistemos parašui naudotų komponentų duomenų kiekio, bei galimas entropijų išsidėstymo intervalas. Lentelėje yra pateikta entropijos vidurkiai, atlikus 20 skaičiavimų, naudojant atitinkamą komponentų duomenų skaičių. Kaip matyti iš 4 lentelėje pateiktų duomenų, *parasas1()* funkcija, sudaryti sistemos parašai nepasižymi mažu entropijos standartiniu nuokrypiu, vidutinis nuokrypis 0,1338. Skirtumas tarp didžiausio ir mažiausio standartinio nuokrypio yra 0,1834. Iš to galima daryti išvadą, kad sistemos parašai nepasižymi stabiliomis entropijomis.

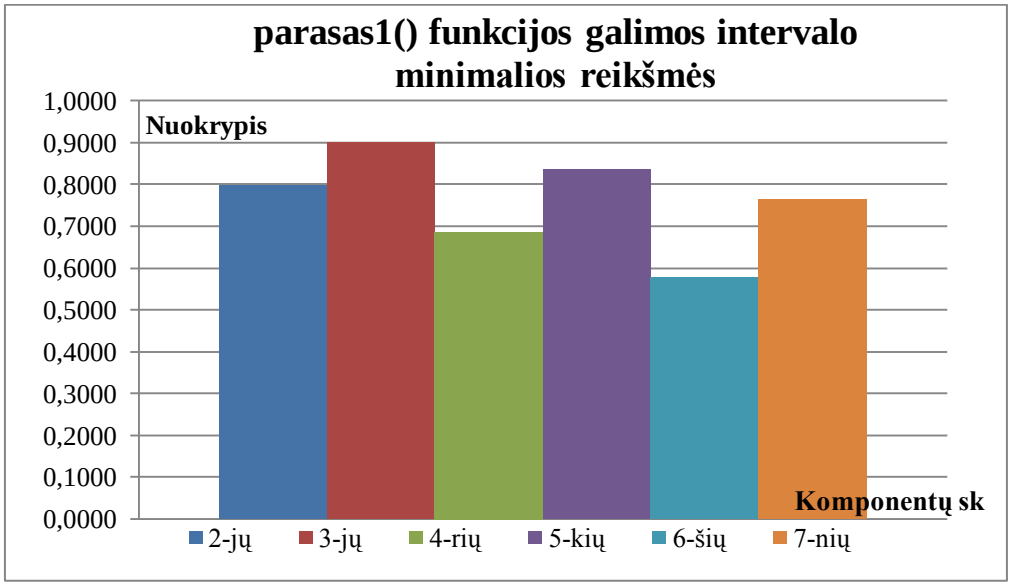
4 lentelė.funkcijos parasas1()standartinis nuokrypis.

Komponentų sk.	2	3	4	5	6	7	Vidurkis
Entropijos vidurkis	0,9095	0,9492	0,8553	0,9358	0,8126	0,9010	0,8939
Standartinis nuokrypis	0,1127	0,0507	0,1693	0,1002	0,2341	0,1358	0,1338
	Galimas reikšmių intervalas						
Minimali	0,7968	0,8986	0,6861	0,8357	0,5785	0,7652	0,7601
Maksimali	1,0000	0,9999	1,0000	1,0000	1,0000	1,0000	1,0000



22 pav.parasas1()funkcijos standartinių nuokrypių išsidėstymas

Kaip matyti iš 22 paveikslo sistemos parašams, sudarytiems naudojant *parasas1()* funkciją yra būdingas ciklinis entropijos standartinio nuokrypio padidėjimas ir sumažėjimas. Cikliškumas priklauso nuo sistemos parašui naudojamų komponentų duomenų skaičiaus. Jeigu yra naudojamas nelyginis komponentų duomenų skaičius yra stebimas ryškus standartinio nuokrypio sumažėjimas. Tai aiškiai matosi kai yra naudojami 5-kių ir 6-šių komponentų duomenys. Standartinio nuokrypio skirtumas tarp taip sudarytų sistemos paršų yra 1,339. Iš tokių gautų duomenų galima spręsti, kad sistemos parašui, sudaryti naudojant *parasas1()*funkciją, reikia pasirinkti nelyginį komponentų duomenų kiekį. Tą galima matyti iš 4 lentelės ir 23 paveikslo, jeigu yra naudojama nelyginis komponentų duomenų kiekis galimas entropijų intervalas yra mažesni. Naudojant 5-kių komponentų duomenis, sistemos parašų entropijų intervalas yra [0,8357; 1,000], o naudojant 6-šių – [0,5785;1,000]. Matome, kad intervalas naudojant 6-šių komponentų duomenis yra 0,2572 didesnis, iš ko sprendžiame, kad gautos sistemų parašų entropijų reikšmės yra mažiau stabilios.



23 pav. Funkcijos parasas1() mažiausios spėjamos intervalų reikšmės.

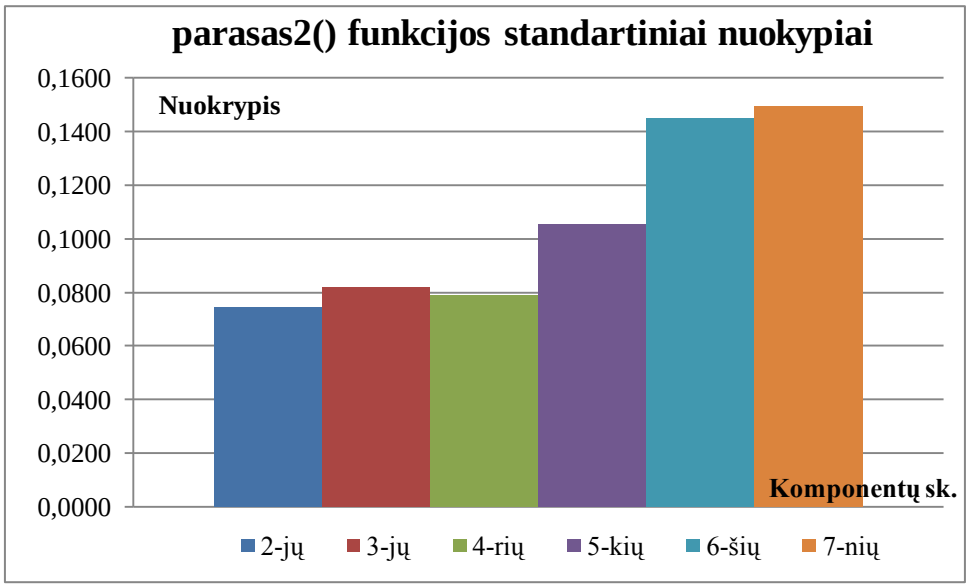
Funkcijos *parasas2()*, sudarytų sistemos prašų, rezultatai pateikti 5 lentelėje. Palyginus gautus rezultatus su funkcijos *parasas1()*, matome kad šia funkcija sudaryti sistemos parašai pasižymi mažesniu standartiniu nuokrypiu. *parasas1()* funkcijos atveju blogiausias standartinis nuokrypis buvo 0,2341, o *parasas2()* – 0,1493. Tai yra 0,0848 mažiau. Be to ir skirtumas tarp mažiausio ir didžiausio nuokrypio yra 0,0747, kai *parasas1()* funkcijos atvejus – 0,1834. Iš to matome, kad naudojant *parasas2()* funkciją sudaryti sistemos parašai pasižymi stabilesnėmis entropijos reikšmėmis. Tai parodo ir vidutinis galimas entropijos reikšmių intervalas, funkcijos

parasas1() reikšmių intervalo dydis yra 0,2399, o *parasas2()* – 0,2117. Tačiau šia funkcija sudaryti sistemos parašai pasižymi mažesnėmis entropijos reikšmėmis, vidutiniškai – 0,0416, entropijos atžvilgiu tai nėra gerai.

5 lentelė. Funkcijos parasas2()standartinis nuokrypis.

Komponentų sk.	2	3	4	5	6	7	Vidurkis
Entropijos vidurkis	0,8994	0,8897	0,8731	0,8526	0,7958	0,8030	0,8523
Standartinis nuokrypis	0,0745	0,0819	0,0787	0,1055	0,1450	0,1493	0,1058
	Galimas reikšmių intervalas						
Minimali	0,8249	0,8078	0,7944	0,7471	0,6508	0,6537	0,7464
Maksimali	0,9740	0,9716	0,9518	0,9580	0,9408	0,9522	0,9581

Kaip matyti iš 24 paveikslo, naudojant *parasas2()* funkciją, sistemos parašo entropijos standartinis nuokrypis priklauso nuo komponentų duomenų skaičiaus, naudojamų jam sudaryti. Kuo didesnis duomenų skaičius tuo didesnis ir nuokrypis. Tokį entropijos standartinio nuokrypio didėjimą galėjo įtakoti *parasas2()* funkcijoje naudojama OR loginė operacija bei pseudoatsitiktinai sugeneruotos komponentų duomenų eilutės. OR operacija pasižymi savybe, kuo daugiau bitų eilučių apdorojama, naudojant ją, tuo didesnė tikimybė, kad galutinėje bitų eilutėje bus žymiai daugiau 1. Todėl naudojant OR loginę operaciją būtina tinkamai pasirinkti apdorojamas bitų eilutes, kas eksperimento metu nebuvo daroma.



24 pav.parasas2() funkcijos standartinių nuokrypių išsidėstymas.

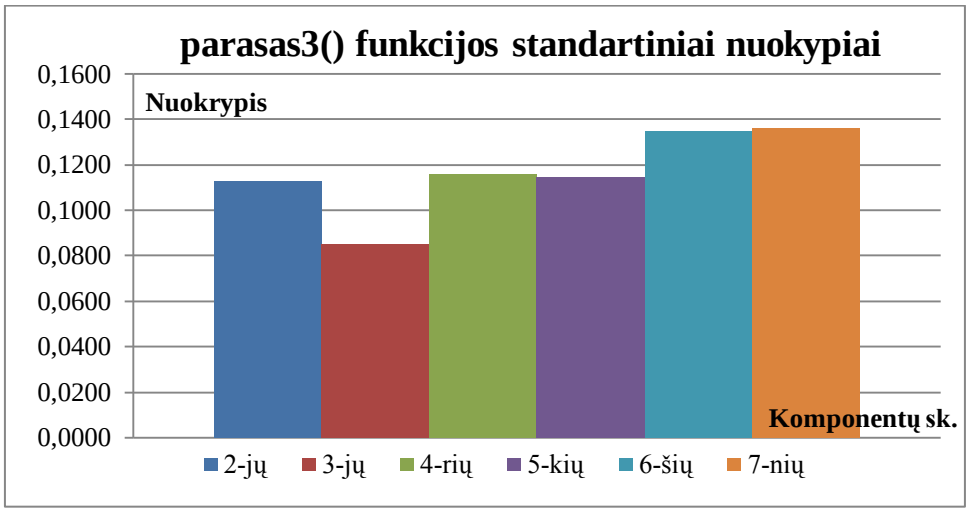
Funkcijos *parasas3()*, sudarytų sistemos prašų entropijų rezultatai pateikti 6 lentelėje. Entropijos atžvilgiu, ši funkcija yra panaši į *parasas1()*. Tačiau pasižymi mažesniu vidutiniu nuokrypiu, kuris yra panašus į *parasas2()* funkcijos. Net ir galimas entropijos reikšmių intervalo dydis yra panašus. Iš ko galima spręsti, kad šiomis funkcijomis sudaryti sistemos parašai pasižymės panašiomis savybėmis. Tik *parasas3()* funkcijos atveju galima tikėtis didesnių, sistemos parašų entropijų.

6 lentelė. Funkcijos parasas3()standartinis nuokrypis.

Komponentų sk.	2	3	4	5	6	7	Vidurkis
Entropijos vidurkis	0,9095	0,9108	0,8737	0,9235	0,8701	0,9010	0,8981
Standartinis nuokrypis	0,1127	0,0853	0,1156	0,1143	0,1347	0,1358	0,1164
	Galimas reikšmių intervalas						
Minimali	0,8249	0,8078	0,7944	0,7471	0,6508	0,6537	0,7817
Maksimali	1,0000	0,9962	0,9893	1,0000	1,0000	1,0000	0,9976

Palyginus 22 – 25 paveikslus matome, kad sistemos parašams sudaryti, naudojant *parasas3()* funkciją, standartinių nuokrypių išsidėstymas yra stabilesnis. Nėra tokio ciklišumo kaip *parasas1()* funkcijos atveju (22 pav.), ar tokios aiškios standartinio nuokrypio priklausomybės nuo sistemos parašui sudaryti, naudojamų komponentų duomenų skaičiaus kaip *parasas2()*

funkcijos atveju (24 pav.). Tokius rezultatus galėjo įtakoti mišrus XOR ir OR loginių operacijų naudojimas.



25 pav.Funkcijos parasas4()standartinių nuokrypių išsidėstymas.

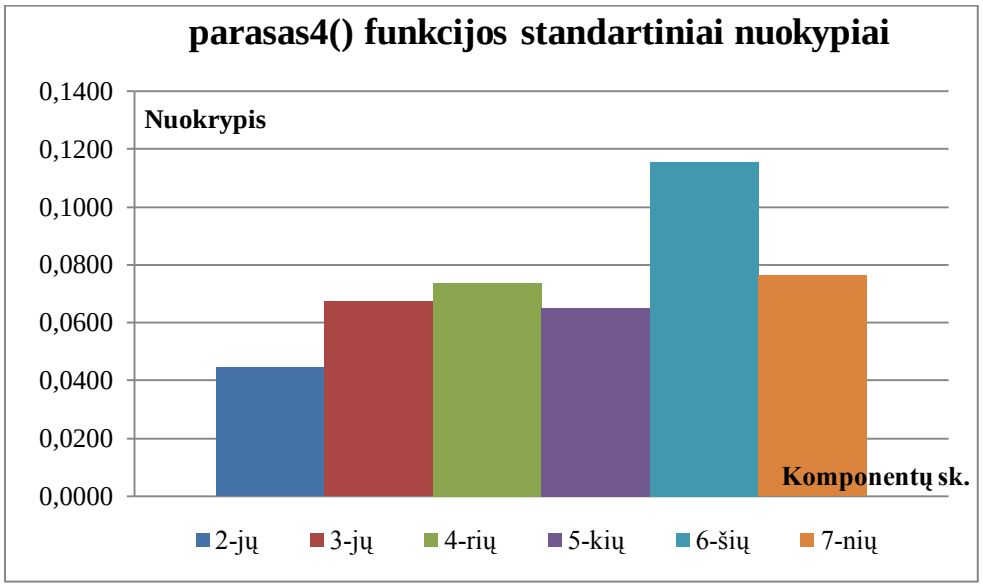
Kaip matyti iš 7 lentelės, *parasas4()* funkcija sudaryti sistemos parašai pasižymi didžiausia entropija. Bei mažiausiu galimų reikšmių intervalų dydžiais. Didžiausias reikšmių intervalas yra 0,2014, kaip *parasas1()* funkcijos – 0,4215. Tai yra beveik dvigubai mažesnis. Kitų dviejų funkcijų atžvilgiu šis skirtumas taip yra panašus.

7 lentelė. Funkcijos parasas4()standartinis nuokrypis

Komponentų sk.	2	3	4	5	6	7	Vidurkis
Entropijos vidurkis	0,9792	0,937	0,9183	0,9535	0,9140	0,9323	0,9391
Standartinis nuokrypis	0,0443	0,067	0,0737	0,0649	0,1154	0,0762	0,0737
	Galimas reikšmių intervalas						
Minimali	0,9349	0,870	0,8447	0,8885	0,7986	0,8561	0,8654
Maksimali	1,0000	1,0000	0,9920	1,0000	1,0000	1,0000	0,9987

Kaip matyti iš 26 paveikslo, *parasas4()* funkcija sudarytų sistemos parašų entropijos standartiniam nuokrypiui, mažiausiai įtakos turėjo komponentų skaičiaus kitimas. *parasas2()* ir *parasas3()* funkcijų atvejų, matome kad komponentų duomenų skaičiaus didėjimas didina ir standartinį nuokrypį. Kai naudojant funkciją *parasas4()*, tokios priklausomybės nėra. Ir šios funkcijos atveju standartinis nuokrypis priklauso tik nuo sistemos parašui sudaryti, naudojamų komponentų duomenų ir jų apdorojimo eiliškumo. Tai aiškiai matyti sistemos parašo sudarymui

naudojant 5-kis, 6-šis ir 7-nis komponentų duomenis. Panaudojus 6-šis, standartinis nuokrypis padidėja ~0,04, palyginus su 5-kiais ir 7-niais komponentais.



26 pav.Funkcijos paprasas4() standartinių nuokrypių išsidėstymas.

Apibendrinti sistemų parašų generavimo rezultatai pateikti 8 lentelėje.

8 lentelė. Entropijos priklausomybė nuo sistemos parašo sudarymo funkcijos.

Funkcija	Vidurkis	Standartinio nuokrypio vidurkis	Galimas reikšmių intervalas	
			Min.	Mak.
parasas1()	0,8939	0,1338	0,7601	1,000
parasas2()	0,8523	0,1058	0,7464	0,9581
parasas3()	0,8981	0,1164	0,7817	0,9976
parasas4()	0,9391	0,0737	0,8654	0,9987

Kaip matyti iš 8 lentelės, sistemos parašai sugeneruoti naudojant *parasas4()* pasižymėjo geriausiomis savybėmis. Galimas reikšmių intervalas ([0,8654; 0,9987]) ir standartinis nuokrypis (0,0737) yra mažiausi. Iš ko galima spręsti, kad šia funkcija sudaryti sistemos parašai pasižymi geriausiomis savybėmis ir ją reiktų naudoti sistemos parašų generavimui. Blogiausius rezultatus parodė *parasas1()* funkcija. Ja sudaryti sistemos parašai pasižymėjo didžiausiu standartiniu nuokrypiu (0,1338) ir dideliu galimu reikšmių intervalu ([0,7601; 1,000]).

Visomis funkcijomis sugeneruoti sistemos parašai buvo gauti vienodo ilgio, nes visų funkcijų atveju, sistemos parašai buvo generuojami naudojant tuos pačius komponentų duomenis. Kur sistemos parašo ilgis priklauso nuo ilgiausio komponento duomens ilgio, bitais, kuris naudojamas jį sudaryti. Sistemos parašų ilgiai buvo įvertinti naudojant standartinį nuokrypį. Rezultatai pateikti 9 lentelėje.

9 lentelė. Sistemos parašų priklausomybė nuo komponentų duomenų skaičiaus.

Komponentų sk.	2	3	4	5	6	7
Ilgio vidurkis, bitais	161,2	183,6	190,4	190,4	197,2	213,6
Standartinis nuokrypis	90,4454	84,3242	84,409	84,409	83,92	76,6499
	Galimas reikšmių intervalas					
Minimali	70,7546	99,2758	105,99	105,991	113,3	136,95
Maksimali	231,955	256	256	256	256	256

Kaip matyti iš 9 lentelės, sistemos parašų ilgiui komponentų duomenų skaičius įtakos turi, bet nedaug. Kuo daugiau komponentų duomenų naudojama sistemos paprašo sudarymui, tuo didesnė tikimybė, kad bus sugeneruotas ilgesnis parašas. Tačiau išlieka galimybė, kad panaudojus 7-nių komponentų duomenis, sugeneruotas sistemos parašo ilgis nebus didesnis nei panaudojus 2-jų. Išlieka labai didelis galimų reikšmių intervalas, nes sugeneruoto sistemos parašo ilgį apsprendžia naudojamų komponentų duomenų ilgis, o ne jų kiekis.

Apibendrinus gautus rezultatus galima teigti, kad sistemos parašai netinka būti naudojami kaip šifravimo/iššifravimo raktai. Pirmiausia dėl to, kad jų entropijos yra nestabilios, net ir geriausius rezultatus parodžiusios *parasas4()* funkcijos, galimas reikšmių intervalas yra didelis 0,1333. Iš kurio galima spręsti, kad nemažai sudarytų sistemos parašų nebus pakankamai arti maksimalios galimos reikšmės 1. Tai nėra gerai, jeigu norima sistemos parašus naudoti kaip šifravimo/iššifravimo raktus. Antra, sistemos parašų ilgių nevienodumas, kurio kontroliuoti negalima dėl pseudoatsitiktinai parenkamų komponentų duomenų, parašų generavimui.

4.2. Šifravimo/iššifravimo rakto generavimas, naudojant sistemos parašą

Šifravimo/iššifravimo raktų generavimui iš sistemos parašo buvo naudojamos trys maišos funkcijos: MD5, SHA, SHA-2. Šifravimo/iššifravimo raktų sudarymui, sistemos parašai yra naudojami tie, kurių generavimui buvo naudojamos *parasas1()* ir *parasas4()* funkcijos tai yra tos funkcijos, kuriomis sudaryti sistemos parašai parodė blogiausius ir geriausius rezultatus. Be entropijos šiame eksperimente buvo stebima ir šifravimo/iššifravimo rakto generavimo laiko priklausomybė nuo maišos funkcijos ir naudojamų komponentų duomenų skaičiaus, sudarančių sistemos parašą. Eksperimento rezultatai pateikti priede 8.1, 5 lentelėje, apibendrinti rezultatai pateikti 10 lentelėje.

Eksperimentas parodė, kad visos maišos funkcijos generavo raktus su žymiai stabilesnėmis entropijos reikšmėmis, palyginus juos su sistemos parašais. Kaip matyti iš 10 lentelės, sistemos parašui sudaryti naudojant *parasas1()* funkciją, standartinio nuokrypio vidurkis MD5 – algoritmo atveju sumažėjo 93,8%, SHA – 96,5%, SHA-2 – 96,5%. Iš ko matyti, kad šifravimo/iššifravimo raktų entropijos tapo žymiai stabilesnės. Sistemos parašų, sudarytų *parasas1()* funkcija, entropijų spėjamų reikšmių intervalas sumažėjo nuo [0,760;1,000] iki [0,985;1,000] (MD5 algoritmo, nes kitų algoritmų intervalai yra dar mažesni), tai yra entropijos reikšmių intervalo dydis sumažėjo nuo 0,24 iki 0,015 arba 94,2%. Toks galimų reikšmių intervalo dydžio sumažėjimas rodo, kad šifravimo/iššifravimo raktų entropijų reikšmės tapo žymiai didesnės ir artimesnės maksimaliai galimai reikšmei, palyginus jas su sistemos parašais.

Panašūs rezultatai buvo gauti ir su *parasas4()* funkcija. Funkcijos *parasas4()* atveju gavome, kad standartinio nuokrypio vidurkis MD5 – algoritmo atveju sumažėjo 89%, SHA – 91,3%, SHA-2 – 95,7%. Spėjamų reikšmių intervalo dydis sumažėjo nuo 0,766 iki 0,014 arba 89,5%.

10 lentelė. Sistemos parašų ir Šifravimo/iššifravimo raktų entropijų palyginimas

Komponentų sk. / Naudotas metodas	2	3	4	5	6	7	Vidurkis	Vidutinis standartinis nuokrypis	Galimas vidutinis reikšmių intervalas
parasas1()	0,909	0,949	0,855	0,935	0,812	0,901	0,894	0,1338	[0,760;1,000]
parasas1()+MD5	0,995	0,989	0,993	0,992	0,995	0,996	0,993	0,0083	[0,985;1,000]
parasas1()+SHA	0,998	0,996	0,996	0,995	0,998	0,996	0,997	0,0047	[0,992;1,000]
parasas1()+SHA2	0,997	0,998	0,998	0,998	0,998	0,997	0,998	0,0032	[0,995;1,000]
parasas4()	0,979	0,937	0,918	0,953	0,914	0,932	0,939	0,0737	[0,8654;0,999]
parasas4()+MD5	0,995	0,994	0,994	0,991	0,995	0,993	0,994	0,0081	[0,986;1,000]
parasas4()+SHA	0,997	0,995	0,994	0,993	0,997	0,994	0,995	0,0064	[0,989;1,000]
parasas4()+SHA2	0,998	0,998	0,996	0,998	0,998	0,998	0,998	0,0032	[0,995;1,000]

Abiejų funkcijų atveju gavome, kad geriausius entropijos rezultatus rodė tie šifravimo/ iššifravimo raktai, kurie buvo sugeneruoti SHA-2 algoritmu. Apibendrinus gautus rezultatus matome, kad maišos algoritmų naudojimas žymiai pagerino sistemos parašų savybes, entropijos atžvilgiu. Jos tapo stabilesnės ir didesnės, bei artimesnės maksimaliai galimai reikšmei 1. Taip pat išsprendė sistemos parašų ilgių nestabilumo problemas. Todėl šifravimui reiktų naudoti, šifravimo/iššifravimo raktus, sudarytus iš sistemos parašo ir maišos algoritmų, o ne sistemos parašus.

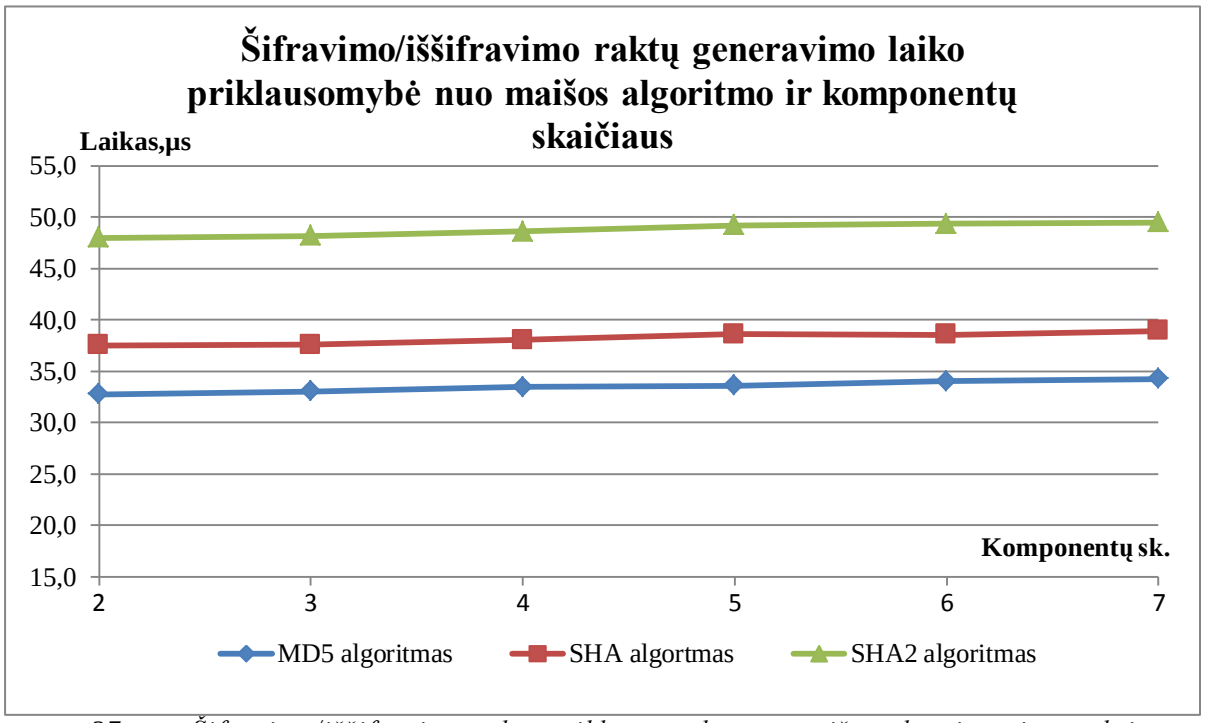
4.2.1. Šifravimo/iššifravimo rakto generavimo laikas

11 lentelėje ir 27 paveiksle pateikta šifravimo/iššifravimo raktų generavimo laikai. Į laiko skaičiavimą įėjo: apsaugomos programos antraštės maišos reikšmės skaičiavimas MD5 algoritmu, sistemos parašo sudarymas ir sistemos parašo maišos reikšmės skaičiavimas, naudojant *parasas4()* funkciją. Iš 27 paveikslo matyti, kad sistemos parašo sudarymui, naudojamų komponentų duomenų skaičius bei sistemos parašo ilgis, šifravimą/iššifravimo rakto generavimo laikui turi mažai įtakos. Vienu komponento duomeniu padidinus sistemos parašo sudarymą, vidutiniškai rakto generavimo laikas pailgėjo ~0,3 µs. Rakto generavimo pailgėjimas, sistemos parašui sudaryti naudojant 2 ir 7 ID, yra ~1,5 µs. Taigi rakto generavimo laikas labiausiai priklauso nuo naudojamo maišos algoritmo.

11 lentelė. Šifravimo/iššifravimo raktų generavimo laikai.

	Komponentų sk.	2	3	4	5	6	7	Vidurkis
MD5	Vidutinis Vykdymo laikas, µs	32,750	33,040	33,453	33,586	34,020	34,242	33,515
	Nuokrypis	0,482	0,433	0,615	0,880	0,625	0,616	0,608
	Galimas vykdymo laiko intervalas (min.;maks.)	32,268	32,607	32,838	32,706	33,395	33,627	32,907
		33,231	33,473	34,069	34,466	34,645	3`4,858	34,123
SHA	Vidutinis Vykdymo laikas, µs	37,544	37,577	38,047	38,580	38,571	38,933	38,209
	Nuokrypis	0,342	0,675	0,485	0,594	0,721	0,702	0,586
	Galimas vykdymo laiko intervalas (min.;maks.)	37,202	36,901	37,562	37,987	37,850	38,231	37,622
		37,886	38,252	38,532	39,174	39,292	39,635	38,795
SHA-2	Vidutinis Vykdymo laikas, µs	47,976	48,203	48,620	49,194	49,358	49,479	48,805
	Nuokrypis	0,419	0,691	0,537	0,689	0,853	0,681	0,645
	Galimas vykdymo laiko intervalas, (min.;maks.)	47,557	47,512	48,083	48,504	48,505	48,798	48,160
		48,395	48,893	49,156	49,883	50,212	50,160	49,450

Kaip matyti iš 27 paveikslo šifravimo raktai greičiausiai generuojami naudojant MD5 maišos algoritmą, lėčiausiai – SHA-2. Vidutinis laiko skirtumas tarp šių dviejų algoritmų yra 45,6%, tai yra MD5 algoritmas beveik dvigubai greičiau sugeneruoja šifravimo/iššifravimo raktus, nei SHA-2. Kaip parodė tyrimas entropijos atžvilgiu MD5 algoritmo sugeneruoti raktai mažai atsilieka nuo SHA-2 algoritmu sugeneruotų. Įvertinus maišos algoritmų sugeneruotų šifravimo/iššifravimo raktų entropijas (11 lentelė) ir vykdymo laikus (12 lentelė ir 27 pav.), raktų generavimui geriau naudoti MD5 algoritmą. Nes skirtumas tarp maišos algoritmų sugeneruotų raktų entropijų yra tik ~0,4%, o laiko atžvilgiu skirtumas 45,6%.



27 pav. Šifravimo/iššifravimo raktų priklausomybė nuo maišos algoritmo ir naudojamų komponentų duomenų skaičiaus.

Į aukščiau pateiktas savybes reikia atsižvelgti, jeigu yra naudojamas 128 bitų ar trumpesnis šifravimo/iššifravimo raktas. Tačiau esant ilgesniam raktui, tarkim 256 bitų, reiktų įvertinti kiek laiko pailgėja rakto generavimas naudojant MD5 algoritmą, kai jį reikia pailginti, nes atsiranda papildomų veiksmų, kurie reikalauja papildomo laiko. Atliktame eksperimente nebuvo stebima šifravimo/iššifravimo raktų ilginimo laikų sąnaudos.

Be naudojamų raktų ilgių ir jiems sugeneruoti reikalingų maišos algoritmų, reiktų įvertinti sistemas, kuriose naudojamas apsaugos metodas. Kad šifravimo/iššifravimo rakto generavimas neužtruktų per ilgai, nes šifravimo/iššifravimo raktų generavimas skirtingomis maišos funkcijomis, laiko atžvilgiu, skiriasi labai.

4.3. Apsaugoto modulio vykdymas

Eksperimento metu buvo stebimas apsaugoto modulio vykdymo laikas. Modulio apsaugai buvo naudoti 5 šifravimo algoritmai: DES, 3DES, AES, IDEA ir Blowfish. Iššifravimo raktas buvo generuojamas funkcija *parasas4()* ir MD5 maišos algoritmu, kai sistemos parašą sudarė 7-ių komponentų duomenys. AES, IDEA ir Blowfish šifravimo algoritmams buvo naudojami 128 bitų raktai. DES ir 3DES algoritmų raktai buvo pritaikomi pagal 3 lentelėje pateiktus metodus. Norit nustatyti kokią įtaką modulio vykdymui turi šifravimas, pirmiausia nustatytas vidutinis modulio vykdymo laikas jį vykdant 20 kartų. Apsaugotas modulis su kiekvienu šifravimo algoritmu buvo vykdomi po 20 kartų. Eksperimente naudotame modulyje suprogramuotas diferencialinės lygties Y'=F(x,y) sprendimas, naudojant Runge-Kutta metodą. Modulio dydis 100 kB. Gauti rezultatai pateikti 12 lentelėje.

12 lentelė. Apsaugoto modulio vykdymo laikai.

	Neapsaugotos programos	DES	3DES	AES	IDEA	Blowfish
Matavimo nr.	Laikas, ms					
1	26,634	29,963	32,721	30,120	29,622	30,289
2	26,860	30,217	33,301	31,445	30,506	30,556
3	26,045	29,333	32,083	30,416	29,345	29,295
4	26,914	30,658	33,239	31,024	30,157	30,297
5	29,070	32,342	35,221	32,886	32,085	32,482
6	24,345	27,831	30,823	28,967	27,328	27,656
7	26,583	30,422	33,408	30,471	29,864	29,777
8	24,396	27,564	30,417	28,057	27,465	27,512
9	26,630	30,327	32,481	30,781	29,924	30,182
10	26,323	29,748	33,117	29,880	29,733	29,583
11	27,286	31,031	34,244	31,743	30,856	30,840
12	26,235	29,716	33,160	30,233	29,640	30,060
13	24,289	28,149	31,136	28,433	27,399	27,417
14	28,804	32,748	35,486	33,458	32,243	32,504
15	26,800	30,584	32,850	31,536	30,291	30,775
16	25,742	28,814	32,526	29,454	29,136	29,016
17	28,255	31,602	34,114	33,011	31,835	31,343
18	26,194	29,482	32,700	30,478	29,532	29,786
19	26,842	29,898	32,909	30,960	30,263	29,973
20	26,267	29,919	32,484	30,577	29,111	29,562
Vidurkis	26,526	30,017	32,921	30,697	29,817	29,945
Vidutinis laiko padidėjimas	____	3,492	6,395	4,171	3,9291	3,420

Kaip matyti iš 12 lentelės, geriausiomis laikinėmis charakteristikomis pasižymėjo Blowfish, IDEA ir DES algoritmai. Apsaugoto modulio vykdymas labiausiai padidėjo naudojant 3DES šifravimo algoritmą. Tai yra ~9,4% ilgiau nei IDEA šifravimo algoritmo atveju. Nes pagal gautus rezultatus šifravimas 3DES šifravimo algoritmas užtruko 48,5% ilgiau nei IDEA. Tokį laiko padidėjimą lėmė 3DES algoritmo savybės.

Iš geriausius rezultatus parodžiusių šifravimo algoritmų, DES naudoja trumpiausius raktus ir yra mažiausiai atsparus kriptoanalizės atakoms. Todėl, jeigu yra reikalingas didesnis apsaugos lygis, šio algoritmo naudoti nereiktų. Lyginant IDEA ir Blowfish algoritmus, Blowfish algoritmas pasižymi geresnėmis duomenų šifravimo savybėmis. Todėl taikomųjų programų modulių apsaugai reiktų rinktis šį šifravimo algoritmą. Be to šis algoritmas gali naudoti šifravimo raktu nuo 32 iki 448 bitų, kas palengvina jo naudojimą su įvairiomis maišos funkcijomis, nes nereikalingas papildomas šifravimo/iššifravimo rakto ilginimas ar trumpinimas.

13 lentelė. Šifravimo/iššifravimo rakto generavimo laikas,modulio šifravimo procese.

	Rakto generavimas, naudojant MD5 algoritmą	Rakto generavimas, naudojant SHA algoritmą	Rakto generavimas, naudojant SHA-2	DES	3DES	AES	IDEA	Blowfish
Vidutinis laikas, ms	0,0335	0,0382	0,0488	3,492	6,395	4,171	3,291	3,420
Rakto generavimo laikas MD5 algoritmu, %	_____	_____	_____	0,96	0,52	0,8	1,02	0,98

Kaip matyti iš 13 lentelės, šifravimo/iššifravimo rakto generavimas, modulio šifravimo procese užima mažai laiko. Atliktame eksperimente rakto generavimo laikas sudarė ~1% modulio šifravimo laiko, MD5 algoritmo atveju. Įvertinus 11 lentelėje pateiktus duomenis galima numatyti, kad ir kiti (SHA ir SHA-2) maišos algoritmai, naudojami sistemos parašui sudaryti, šifravimo/iššifravimo proceso, laiko atžvilgiu, labai neįtakos. Be to laikas reikalingas rakto generavimui mažėja, didėjant šifravimo laikui, kaip tai yra DES algoritmo atveju. Kaip matyti iš gautų rezultatų, pasiūlytas raktų generavimo algoritmas turi mažai įtakos modulio iššifravimo laikui, o modulio šifravimo laikas labiausiai priklauso nuo naudojamo šifravimo algoritmo. Apibendrinus gautus rezultatus galima teigti, kad pasiteisino siūlymas šifravimo/iššifravimo

raktus generuoti naudojant maišos funkcijas ir sistemos parašą, sudarytą iš sistemos komponentų duomenų ir loginių operacijų XOR ir OR.

4.4. Išvados

1. Eksperimentai parodė, kad sistemos parašai netinkami naudoti kaip šifravimo/iššifravimo raktai. Pirmiausia dėl to, kad sudarytų sistemos parašų entropijų galimų reikšmių intervalai yra dideli net ir geriausios naudotos funkcijos atveju jis yra [0,8654;0,999]. Antra sugeneruoti sistemos parašai nėra fiksuoto ilgio, nes jų ilgis priklauso nuo ilgiausio sistemos komponento duomens, naudoto sistemos parašui sudaryti.
2. Šias problemas išsprendė sistemos parašo maišos skaičiavimas ir jos reikšmės naudojimas kaip šifravimo/iššifravimo rakto. Kaip parodė eksperimentai taip sudaryti šifravimo/iššifravimo raktai pasižymėjo geromis entropijomis, kurios buvo artimos maksimaliai reikšmei. Geriausiomis charakteristikomis pažymėjo tie raktai, kurie buvo sugeneruoti SHA-2 maišos algoritmu. Tačiau rakto generavimo, laiko atžvilgiu, ši funkcija buvo lėčiausia. Ji net 45,6% lėtesnė už MD5 algoritmą. O šiomis funkcijomis sudarytų šifravimo/iššifravimo raktų entropijų skirtumas yra ~0,4%. Šifravimo/iššifravimo raktų generavimui rekomenduojama naudoti MD5 maišos algoritmą, bet tik tuo atveju jeigu reikalingas rakto ilgis yra 128 bitai arba trumpesnis.
3. Apsaugotos programos vykdymo laiko pailgėjimas labiausiai priklauso nuo naudojamo šifravimo algoritmo. Geriausiomis laikinėmis charakteristikomis pažymėjo Blowfish algoritmas. Modulis užšifruotas juo buvo vykdomas greičiausiai. Iššifravimo rakto generavimas, sukurtu metodu, užtruko apie ~1% iššifravimo laiko. Tai yra beveik neįtakojo modulio vykdymo pailgėjimo.

5. IŠVADOS

1. Atlikus taikomųjų programų apsaugos nuo nelegalaus naudojimo ir intelektinės nuosavybes vagysčių analizę išsiaiškinta, kad didžiausią grėsmę apsaugos metodams kelia atvirkštinė inžinerija.
2. Atlikus programų apsaugos programiniais ir aparatūriniais metodais analizę išsiaiškinta, kad programos ar jos dalies glaudinimas ar/ir šifravimas gerai apsaugo nuo atvirkštinės inžinerijos, nes norint atlikti ją pirmiausia reikia programą iškleisti. Išskleidimo veiksmus, atakuotojui, apsunkina naudojamas apsaugos raktus. Be to, šią technologiją galima lengvai panaudoti kartu su kitomis aprašytomis apsaugos priemonėmis. Todėl panaudojus įvairias jų kombinacijas galima pasiekti gerų rezultatų kovoje prieš atvirkštinę inžineriją.
3. Atsižvelgus į analizės rezultatus buvo pasiūlytas programų apsaugos metodas, paremtas atskirų programos modulių šifravimu. Kur šifravimo raktai generuojami realiuoju laiku, apsaugoto modulio paleidimo metu. Šifravimo/iššifravimo raktai generuojami naudojant sistemos parašą ir maišos algoritmus.
4. Darbe sukurto apsaugos metodo, eksperimentinis įvertinimas paradė, kad pasiūlytu metodo yra generuojami aukštos entropijos raktai, be jokios papildomos aparatūrinės įrangos ir infrastruktūros. Kaip parodė eksperimentai, šifravimo/iššifravimo raktus geriausia generuoti naudojant SHA-2 maišos algoritmą. Tačiau sistemose, kurios turi ribotus skaičiavimo resursus, gėriau naudoti MD5 algoritmą, nes jis yra ~45% greitesnis už SHA-2, o raktų entropija yra ~0,4% mažesnė.
5. Apsaugotos programos vykdymas parodė, kad programos vykdymo pailgėjimas labiausiai priklauso nuo naudojamo šifravimo algoritmo. Geriausius rezultatus parodė Blowfish algoritmas, mažiausiai pailgėjo programos vykdymas. Eksperimentai parodė, kad raktų generavimas pasiūlytu metodu, iššifravimo procese užėmė ~1% laiko, tai yra beveik neįtakojo programos veikimo pailgėjimo.
6. Magistrinio darbo tematika parašyti trys moksliniai straipsniai išspausdinti: „Informacinės technologijos ir valdymas“ 2012 m. [39], „Mechanika“ 2012 m, Nr. 18 (2) [40] ir 17-osios Tarpuniversitetinės magistrantų ir doktorantų mokslinės konferencijos "Informacinės technologijos 2012" pranešimų medžiagoje[41]. Taip pat perskaityti pranešimai 17 tarptautinėje konferencijoje „Mechanika 2012“ vykusioje 2012-04-12/13 Kaune bei 17 tarpuniversitetinė magistrantų ir doktorantų konferencija „Informacinės technologijos 2012“ vykusioje 2012-04-20 Kaune.

6. LITERATŪRA

1. Bisness Software Alliance. 2010. Seventh Annual BSA and IDC Global Software Piracy Study. p. 18.
2. Sandra Parker. Types of Software Piracy. Iš *ehow* [interaktyvus]. 2010, Sausis. [žiūrėta 2011-01-24]. Prieiga per internetą: http://www.ehow.com/list_5911080_types-software-piracy.html
3. P. Červen. Crackproof Your Software – The Best Ways to Protect Your Software Against Crackers. V.: No Starch Press, 2001, p. 170.
4. T. Jiutao, L. Guoyuan. Research of Software Protection // International Conference on Educational and Network Technology (ICENT 2010), IEEE Computer Society, 2010, p. 410-413.
5. M. Ceccato, M. Di Penta, J. Nagra ir kt.The Effectiveness of Source Code Obfuscation: an Experimental Assessment“, ICPC 2009, IEEE Computer Society, 2009, p. 178-187.
6. Hsiang-Yang Chen, Ting-Wei Hou.Changing Data Type Method of Data Obfuscation on Java Software // International Computer Symposium, 2004, p. 439 – 442.
7. Ming-Hsiu Tsai, Hsiang-Yang Chen, Ting-Wei Hou. Three methods of control flow obfuscation on Java software// International Computer Symposium, 2004, p. 318 – 324.
8. Michael N. Gagnon, S. Taylor, Anup K. Ghosh. Software protection through anti-debugging // Security & Privacy, IEEE Computer Society, 2007, p. 82-84.
9. G. Bing, L. Guoyuan, T. Jiutao. Application of structure exception handling in software anti-debugging // 3rd International Conferance on Advance Computer Theory and Engineering (ICACTE), 2010, p. 133-136.
10. H. Chen, L. Yuan, B. Huang, P. Yew. Control Flow Obfuscation with Information Flow Tracking // 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009, p. 391-400.
11. A. Balakrishnan, C. Schulze. Code Obfuscation Literature Survey // Computer Sciences Department University of Wisconsin, Madison, 2005.
12. C. Kruegel, W. Robertson, F. Valeur, G. Vigna, Static disassembly of obfuscated binarines // Proceedings of the 13th USENIX Security Symposium, 2004.
13. Y. Kanzaki, A. Monden, M. Nakamura and K. Matsumoto. Exploiting self-modification mechanism for program protection, Proc. the 27th Annual International Computer

Software and Applications Conference, Washington: IEEE Computer Society, 2003, p. 170-179

14. Z. Jian-qi, L. Yang-heng, Y. Ke, Y. Ke-xin. A novel method-based software watermarking scheme, Sixth International Conference on Information Technology: New Generation, IEEE Computer Society, 2009, p. 338-392.
15. M. Kim, J.Lee, H. Chang ir kt. Desing and Performance Evaluation of Binary Code Packing for Protecting Embedded Software Against Reverse Engineering // 13th IEEE International Symposium on Object/Component/ Service-Oriented Real-Time Distributed Computing, 2010, p. 80-86.
16. I. J. Jozwiak, A. Liber, K. Marczak. A Hardware-Based Software Protection Systems-Analysis of Security Dongles With Memory // Proc. the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07), Washington: IEEE Computer Society, 2007, p. 28-28.
17. L. MeiHong, L. JiQiang. USB Key-Based Approach for Software Protection // International Conference on Industrial Mechatronics and Automation, 2010, p. 151-153.
18. Business Software Alliance // Sixth Annual BSA and IDC Global Software Piracy Study. 2009.
19. Ruirui H., Daniel Y. Deng, G. Edward S. Orthus:Efficient Software Integrity Protaction on Multi-Cores// ASPLOS '10 Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, 2010.
20. Miliefsky G. Guarding against identity theft//Hakin9. ISSN 1733-7186. 2011, Nr. 3, p. 32.
21. Bing G., Guoyuan L., Jiutao T. Application of Structured ExectionHandling in Software Anti-Debugging// 3rd International Canference on Advanced Computer Theory and Engineering (ICACTE), 2010, p. 133-136.
22. Popov I.V., Debray S. K.,Andrews G.R.Binary Obfuscation Using Signal // 16th USENIX Security Symposium, 2007, p. 275-290.
23. Aycock J., Cardenas J. M., de Castro D. M. Code Obfuscation using Pseudo-Random Number Generators // International Conference on Computational Science and Engineering, 2009, p. 418-422.
24. Wu Z., Gianvecchio S., Xie M. ir kt. Mimimorphism: a new approach to binary code obfuscation // 17th ACM conference on Computer and communications security, 2010.

25. Borello J. M., Me L.Code Obfuscation techniques for metamorphic viruses // Journal in Computer Virology. SSN: 1772-9904. 2008, Nr. 3.
26. Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations // Technical Report 148 Department of Computer Science University of Auckland. ISSN: 1173-3500, 1997, p. 36.
27. Arnault F., Berge T. P. Design and Properties of a New Pseudorandom Generator Based on a Filtered FCSR Automaton // *IEEE Transactions on Computers*, 2005, p. 1374-1383.
28. Gonzalez-Diaz V. R., Setti G. A Pseudorandom Number Generator Based on Time-Variant Recursion of Accumulators // IEEE Transactions on Circuits and Systems Part II: Express Briefs, ISSN: 1549-7747, 2011, Nr. 9.
29. Suh G. E., Devadas S. Physical Unclonable Functions for Device Authentication and Secret Key Generation // Design Automation Conference (DAC 2007), 2007, p. 9-14.
30. Kursawe K., Sadeghi A. R., Schellekens D., Škoric B., Tuyls P. Reconfigurable Physical Unclonable Functions Enabling Technology for Tamper-Resistant Storage // Hardware-Oriented Security and Trust, 2009, p. 22-28.
31. You L., Zang G., Zhang F. A fingerprint and threshold scheme-based key generation method // Computer Sciences and Convergence Information Technology (ICCIT), 2010, p. 615- 619.
32. International Organization for Standardization. ISO/IEC FCD 18033–2, IT Security techniques – Encryption Algorithms – Part 2: Asymmetric Ciphers, 2004.
33. Moskowitz S., Cooperman M. Method for stega-cipher protection of computer code. US Patent 5,745,569, January 1996.
34. Zhang X., He F., Zou W. Hash Function Based Software Watermarking // Advanced Software Engineering and Its Applications, 2008, p. 95-98.
35. Mumtaz S., Iqbal S., Hameed E. I. Development of a Methodology for Piracy Protection of Software Installations// 9th International Multitopic Conference, IEEE INMIC 2005, 2005, p. 1-7.
36. Birrer B., Raines R., Baldwin R., Mullins B., Bennington R. R. Program Fragmentation as a Metamorphic Protection// Third International Symposium of Information Assurance and Security, 2007, p. 369-374.
37. Moser A., Kruegel C., Kirda E. Limits of Static Analysis of Malware Detection// Computer Security Applications Conference, 2007.,ACSAC 2007, 2007, p. 421 – 430

38. Carrera E. Scanning data for entropyanomalies. 2007, Gegužė. [žiūrėta 2011-01-24]. Prieiga per internetą: http://blog.dkbza.org/2007/05/scanning-data-for-entropy-anomalies.html
39. Venčkauskas A., Jusas N., Mikuckienė I., Butleris R. Secret encryption key generation using signature of embedded systems// Information Technology and Control. ISSN: 1392-1215, 41(xx),–. (Submitted).
40. Venčkauskas A., Jusas N., Kižauskienė L., Kazanavičius E., Kazanavičius V. Security method of embedded software for mechatronic systems// Mechanika. ISSN 1392 – 1207, 2012, Nr. 18(2), p. 196-202.
41. Venčkauskas A., Jusas N. Programos apsaugos metodas, naudojant sistemos parašą// 17-osios Tarpuniversitetinės magistrantų ir doktorantų mokslinės konferencijos "Informacinės technologijos" pranešimų medžiaga. ISSN 2029-249X, 2012, p. 153-156.

7. SUMMARY

Software security – protection of programs from illegal use, programming code integrity and intellectual property – is an important problem of nowadays. According to a study by the Business Software Alliance, software creators lost USD 51.4 billion and pirated software accounted for 43% of all software, observed in approximately 2% annual growth trend of piracy. High piracy rate is in Lithuania, were 54% of software are illegal. This brings software-based protection to be one of the most important defenses against illegal software usage. The analysis showed that the biggest problem for software protection is reverse engineering. One of the best measures against reverse engineering is software packaging or/and encryption. Considering to it was proposed software protection method where separate programs modules are encrypted. And encryption keys are generated using a hash function from a system's signature, which is composed of protecting software headers and system hardware and software components identification data. Experiments showed that proposed method generated high entropy encryption keys using system signature. The best function for the generation of key is SHA-2, its generated highest entropy keys. For the encryption should be used Blowfish algorithms, because the best time characteristics were obtained by it. Experiments showed that key generation is very fast comparing to encryption, it takes about 1% of encryption time for 100kB file. To summarize all experiments, the proposed method effectively generates high entropy keys without any additional hardware and infrastructure cost, what is very important for software protection.

8. PRIEDAI

Prieduose yra pateikiami eksperimentų rezultatai, kuriouse buvo skaičiuojama sistemos parašų ir šifravimo/iššifravimo raktų entropijos. Taip pat pateikiami moksliniai straipsniai, parašyti magistrinio darbo tematika.

8.1. Sistemos parašų ir šifravimo/iššifravimo raktų entropijos

1lentelė. Sistemos parašų entropija, naudojant parasas1()funkciją.

Bandymo nr.	2-jų ID		3-jų ID		4-rių ID		5-kių ID		6-šių ID		7-nių ID	
	Bitų skaičius	Entropija	Bitų Skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija
1	120	1,0000	120	0,9544	120	0,9544	120	0,9544	256	1,0000	256	0,5436
2	120	0,8113	120	0,9544	120	0,5436	120	0,9544	120	1,0000	256	0,9544
3	256	0,9544	256	0,8113	256	0,9544	256	0,9544	256	0,8113	256	1,0000
4	64	0,8113	64	0,9544	64	0,9544	64	0,9544	64	0,5436	120	0,8113
5	256	0,9544	256	0,9544	256	0,9544	256	0,9544	256	0,9544	256	1,0000
6	64	0,9544	64	0,9544	64	0,5436	64	0,5436	64	0,5436	64	1,0000
7	64	0,8113	64	0,9544	64	0,5436	64	0,9544	64	0,8113	64	0,9544
8	120	0,8113	120	0,9544	256	0,9544	256	0,9544	256	1,0000	256	0,8113
9	64	0,9544	64	1,0000	64	1,0000	64	1,0000	64	0,8113	64	1,0000
10	120	0,9544	120	1,0000	120	0,8113	120	1,0000	120	1,0000	256	0,9544
11	64	0,9544	256	0,9544	256	0,9544	256	0,9544	256	0,9544	256	0,9544
12	256	0,9544	256	0,9544	256	0,9544	256	0,9544	256	0,8113	256	0,9544
13	256	1,0000	256	0,9544	256	0,5436	256	0,9544	256	0,8113	256	0,9544
14	256	1,0000	256	0,9544	256	0,9544	256	0,9544	256	0,8113	256	0,8113
15	256	1,0000	256	1,0000	256	1,0000	256	1,0000	256	0,8113	256	0,9544
16	256	0,9544	256	0,8113	256	0,8113	256	1,0000	256	0,8113	256	0,9544
17	64	0,8113	120	0,9544	120	0,9544	120	0,9544	120	0,0000	120	0,9544
18	56	0,5436	256	0,9544	256	0,9544	256	0,9544	256	0,9544	256	0,9544
19	256	1,0000	256	0,9544	256	0,9544	256	0,9544	256	0,8113	256	0,9544
20	256	0,9544	256	1,0000	256	0,8113	256	0,8113	256	1,0000	256	0,5436
	Entropijos vidurkis	0,9095		0,9492		0,8553		0,9358		0,8126		0,9010

2 lentelė. Sistemos parašų entropija, naudojant parasas2()funkciją

Bandymo nr.	2-jų ID		3-jų ID		4-rių ID		5-kių ID		6-šių ID		7-nių ID	
	Bitų skaičius	Entropija	Bitų Skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija
1	120	0,8113	120	0,8113	120	0,8113	120	0,8113	256	0,811278	256	0,8113
2	120	0,8113	120	0,8113	120	0,8113	120	0,8113	120	0,811278	256	0,8113
3	256	0,9544	256	1,0000	256	0,9544	256	0,9544	256	0,954434	256	0,9544
4	64	1,0000	64	1,0000	64	1,0000	64	1,0000	64	0,954434	120	1,0000
5	256	0,9544	256	0,8113	256	0,8113	256	0,8113	256	0,543564	256	0,5436
6	64	0,9544	64	0,9544	64	0,9544	64	0,9544	64	0,811278	64	0,9544
7	64	0,9544	64	0,9544	64	0,9544	64	0,9544	64	0,811278	64	0,8113
8	120	0,8113	120	0,8113	256	0,8113	256	0,8113	256	0,811278	256	0,8113
9	64	0,8113	64	0,8113	64	0,8113	64	0,8113	64	0,811278	64	0,8113
10	120	0,8113	120	0,9544	120	0,9544	120	0,9544	120	0,954434	256	0,9544
11	64	0,9544	256	0,8113	256	0,8113	256	0,8113	256	0,811278	256	0,8113
12	256	0,9544	256	0,9544	256	0,8113	256	0,8113	256	0,811278	256	0,8113
13	256	0,9544	256	0,9544	256	0,9544	256	0,8113	256	0,543564	256	0,5436
14	256	0,8113	256	0,9544	256	0,9544	256	0,9544	256	0,954434	256	0,9544
15	256	0,8113	256	0,8113	256	0,8113	256	0,8113	256	0,811278	256	0,5436
16	256	0,9544	256	0,8113	256	0,8113	256	0,8113	256	0,811278	256	0,8113
17	64	0,9544	120	0,8113	120	0,8113	120	0,8113	120	0,543564	120	0,8113
18	56	0,9544	256	0,9544	256	0,8113	256	0,8113	256	0,811278	256	0,8113
19	256	0,8113	256	0,8113	256	0,8113	256	0,5436	256	0,543564	256	0,5436
20	256	0,9544	256	1,0000	256	1,0000	256	1,0000	256	1,0000	256	0,9544
	Entropijos vidurkis	0,8994		0,8897		0,8731		0,8526		0,7958		0,8030

3 lentelė. Sistemos parašų entropija, naudojant parasas3()funkciją.

Bandymo nr.	2-jų ID		OR, 3-jų ID		4-rių ID		5-kių ID		6-šių ID		7-nių ID	
	Bitų skaičius	Entropija	Bitų Skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija
1	120	1,0000	120	0,8113	120	1,0000	120	0,8113	256	0,9544	256	1,0000
2	120	0,8113	120	1,0000	120	0,8113	120	1,0000	120	0,9544	256	0,8113
3	256	0,9544	256	0,9544	256	0,8113	256	1,0000	256	0,9544	256	0,9544
4	64	0,8113	64	1,0000	64	0,8113	64	1,0000	64	0,8113	120	0,9544
5	256	0,9544	256	0,8113	256	1,0000	256	0,8113	256	0,8113	256	0,5436
6	64	0,9544	64	1,0000	64	0,8113	64	0,9544	64	0,9544	64	0,9544
7	64	0,8113	64	1,0000	64	0,8113	64	1,0000	64	0,9544	64	0,9544
8	120	0,8113	120	0,8113	256	0,8113	256	0,9544	256	0,8113	256	1,0000
9	64	0,9544	64	0,9544	64	0,9544	64	0,9544	64	0,9544	64	0,9544
10	120	0,9544	120	0,9544	120	0,9544	120	1,0000	120	1,0000	256	0,9544
11	64	0,9544	256	0,8113	256	1,0000	256	0,8113	256	1,0000	256	0,9544
12	256	0,9544	256	0,9544	256	0,9544	256	0,9544	256	0,8113	256	1,0000
13	256	1,0000	256	0,9544	256	0,5436	256	0,9544	256	0,8113	256	0,9544
14	256	1,0000	256	1,0000	256	0,8113	256	1,0000	256	0,5436	256	1,0000
15	256	1,0000	256	0,8113	256	1,0000	256	0,8113	256	0,8113	256	0,5436
16	256	0,9544	256	0,8113	256	0,8113	256	1,0000	256	0,8113	256	0,8113
17	64	0,8113	120	0,8113	120	0,8113	120	1,0000	120	0,5436	120	0,9544
18	56	0,5436	256	0,9544	256	0,9544	256	0,9544	256	0,9544	256	0,8113
19	256	1,0000	256	0,8113	256	1,0000	256	0,5436	256	1,0000	256	0,9544
20	256	0,9544	256	1,0000	256	0,8113	256	0,9544	256	0,9544	256	0,9544
	Entropijos vidurkis	0,9095		0,9108		0,8737		0,9235		0,8701		0,9010

4 lentelė. Sistemos parašų entropija, naudojant parasas4()funkciją.

Bandymo nr.	2-jų ID		3-jų ID		4-rių ID		5-kių ID		6-šių ID		7-nių ID	
	Bitų skaičius	Entropija	Bitų Skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija	Bitų skaičius	Entropija
1	120	1,000	120	0,954	120	0,811	120	0,954	256	0,954	256	0,954
2	120	1,000	120	1,000	120	0,954	120	1,000	120	1,000	256	0,811
3	256	1,000	256	0,954	256	0,954	256	1,000	256	1,000	256	1,000
4	64	0,954	64	1,000	64	0,954	64	1,000	64	0,811	120	0,811
5	256	1,000	256	0,811	256	0,954	256	0,811	256	0,811	256	0,811
6	64	0,954	64	0,954	64	0,811	64	0,954	64	0,954	64	0,954
7	64	1,000	64	0,954	64	0,954	64	1,000	64	1,000	64	0,954
8	120	0,954	120	0,954	256	1,000	256	0,954	256	0,954	256	0,954
9	64	1,000	64	0,954	64	0,811	64	0,954	64	0,954	64	0,954
10	120	0,811	120	0,954	120	0,954	120	1,000	120	1,000	256	1,000
11	64	1,000	256	0,811	256	0,954	256	0,811	256	1,000	256	1,000
12	256	1,000	256	1,000	256	0,954	256	0,811	256	1,000	256	1,000
13	256	1,000	256	0,954	256	0,811	256	1,000	256	1,000	256	0,811
14	256	1,000	256	0,954	256	1,000	256	0,954	256	0,954	256	0,954
15	256	1,000	256	1,000	256	0,954	256	1,000	256	0,811	256	0,811
16	256	1,000	256	0,811	256	0,954	256	1,000	256	0,811	256	1,000
17	64	1,000	120	0,811	120	0,954	120	1,000	120	0,811	120	1,000
18	56	0,954	256	0,954	256	1,000	256	0,954	256	0,954	256	0,954
19	256	1,000	256	0,954	256	0,811	256	0,954	256	0,544	256	0,954
20	256	0,954	256	1,000	256	0,811	256	0,954	256	0,954	256	0,954
	Entropijos vidurkis	0,9792		0,9372		0,9183		0,9535		0,9140		0,9323

5 lentelė.Šifravimo/iššifravimo raktų entropijų priklausomybė nuo maišos algoritmo.

Bandymo nr.	2-jų ID			3-jų ID			4-rių ID			5-kių ID			6-šių ID			7-nių ID		
	Md5	SHA	SHA2	Md5	SHA	SHA2	Md5	SHA	SHA2	Md5	SHA	SHA2	Md5	SHA	SHA2	Md5	SHA	SHA2
1	1,000	1,000	1,000	1,000	1,000	1,000	0,978	0,997	0,997	0,999	0,991	0,998	0,989	0,998	0,999	1,000	0,991	1,000
2	0,991	0,999	0,995	0,961	0,995	0,996	0,993	0,996	0,999	0,997	0,989	0,999	0,989	0,993	0,998	0,997	0,991	0,996
3	0,993	1,000	0,975	0,990	0,997	0,998	0,999	1,000	1,000	0,993	0,996	0,999	0,999	1,000	1,000	1,000	0,997	1,000
4	0,989	0,993	0,998	0,987	0,996	0,993	0,987	0,990	1,000	0,990	1,000	0,996	1,000	1,000	0,999	0,996	0,998	0,999
5	0,999	0,999	0,997	1,000	1,000	0,997	0,998	0,996	0,990	0,985	1,000	0,999	0,989	0,997	0,997	0,982	1,000	1,000
6	1,000	0,996	1,000	0,965	0,998	1,000	0,998	0,989	1,000	0,996	0,974	0,999	1,000	0,994	0,998	1,000	1,000	0,999
7	0,998	0,998	0,999	0,998	0,996	1,000	0,986	1,000	0,996	0,972	0,999	0,994	0,993	0,990	0,999	1,000	0,971	0,998
8	0,998	0,996	1,000	0,999	0,997	0,999	1,000	0,999	0,999	0,992	0,991	0,997	0,999	0,999	1,000	1,000	0,999	1,000
9	0,989	0,991	0,998	0,998	0,993	0,997	0,998	1,000	0,999	0,992	1,000	0,998	0,989	0,996	1,000	1,000	0,999	0,999
10	0,999	0,999	0,998	0,986	0,991	0,999	1,000	1,000	0,997	0,994	1,000	1,000	0,986	0,998	0,999	0,986	0,998	0,999
11	1,000	0,994	0,999	0,996	0,998	1,000	0,974	0,996	0,999	1,000	0,999	1,000	0,996	0,997	1,000	0,998	0,996	0,997
12	0,999	0,999	0,997	0,986	0,989	0,999	0,992	1,000	0,990	1,000	0,995	1,000	1,000	1,000	0,993	0,996	1,000	0,999
13	1,000	1,000	1,000	0,997	0,987	1,000	0,999	0,995	0,998	0,986	0,994	0,999	1,000	0,998	0,998	0,999	0,991	0,999
14	1,000	1,000	1,000	0,998	1,000	0,994	0,994	0,993	0,996	1,000	1,000	0,999	0,987	1,000	1,000	1,000	1,000	0,998
15	0,978	1,000	1,000	1,000	1,000	0,999	1,000	0,998	1,000	0,996	0,999	0,995	0,999	1,000	1,000	0,998	0,995	1,000
6	0,999	0,999	0,997	0,994	1,000	0,995	0,998	0,996	0,999	0,989	0,991	0,996	1,000	0,999	0,999	1,000	0,999	0,999
17	0,998	0,998	0,999	0,954	0,994	0,993	0,997	0,976	1,000	0,999	0,999	1,000	1,000	0,999	0,994	0,998	0,997	0,978
18	0,997	0,991	1,000	0,999	1,000	0,998	0,967	1,000	1,000	0,976	0,999	0,999	0,995	1,000	0,999	0,997	0,999	0,999
19	0,978	1,000	1,000	0,982	0,993	1,000	0,999	0,999	1,000	0,997	0,986	0,997	1,000	0,997	0,997	0,986	1,000	0,993
20	1,000	0,998	0,998	1,000	0,996	0,999	1,000	0,997	0,999	0,996	0,999	1,000	1,000	1,000	0,997	0,985	0,991	0,996
Entropijos vidurkis	0,995	0,998	0,997	0,989	0,996	0,998	0,993	0,996	0,998	0,992	0,995	0,998	0,995	0,998	0,998	0,996	0,996	0,997

8.2. Publikacija „Security method of embedded software for mechatronic systems“

Šiame priede yra pateikiamas mokslinis straipsnis išspausdintas žurnale „MECHANIKA“. 2012 m, 18(2): 196-202.

ISSN 1392 - 1207. MECHANIKA. 2012. 18(2): 196-202

Security method of embedded software for mechatronic systems

A. Venčkauskas*, N. Jusas, L. Kižauskienė***, E. Kazanavičius****, V. Kazanavičius *****

**Kaunas University of Technology, Studentų 50, LT-51368, Kaunas, Lithuania, E-mail: algimantas.venckauskas@ktu.lt*

*** Kaunas University of Technology, Studentų 50, LT-51368, Kaunas, Lithuania, E-mail: nerijus.jusas@stud.ktu.lt*

**** Kaunas University of Technology, Studentų 50, LT-51368, Kaunas, Lithuania, E-mail: losta@ifko.ktu.lt*

***** Kaunas University of Technology, Studentų 50, LT-51368, Kaunas, Lithuania, E-mail: ekaza@ifko.ktu.lt*

****** TEO LT, AB, Lvovo g. 25, LT-09320, Vilniaus, Lithuania, E-mail: vkaza@ifko.ktu.lt*

1. Introduction

Mechatronic systems are widespread in various areas of life – home, office, manufacturing, and transport. They are widely used in robots, digitally controlled machines, „smart machine tool" and so on. The typical view of mechatronics is as a combination of mechanical and electrical systems controlled by an embedded control system [1] (Fig. 1).

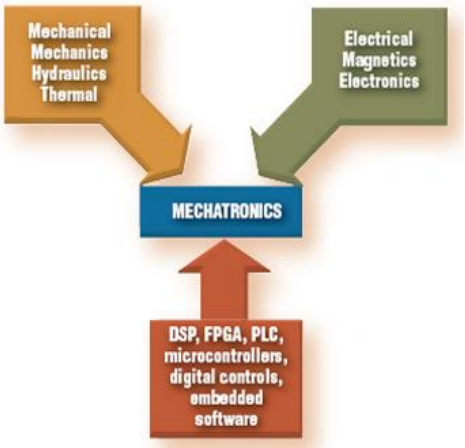


Fig. 1. Mechatronics is a synergy of mechanical and electrical systems controlled by an embedded system.

Machining is a process that removes a lager of material from a workpiece in the form of chips to obtain the desired product shape, size, accuracy, and surface quality. Conventional machining operations, which include turning, milling, grinding, and drilling are among the most common activities in the manufacturing industry (US industries spend US $100 billion annually to machine metals). Experimental structure of smart machine tool is presented in Fig. 2.

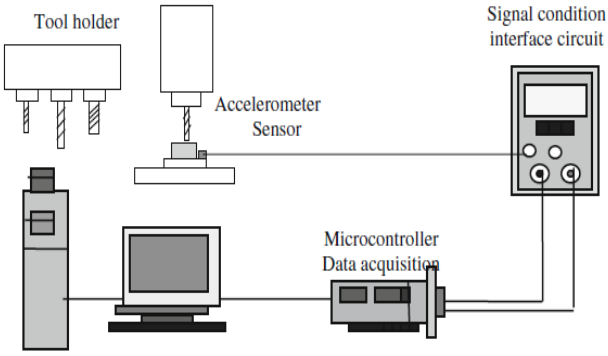


Fig. 2. Experimental structure of smart machine tool.

The complex interaction between machines, tools, workpieces, fluids, measurement systems, material handling systems, humans and the environment in cutting operations requires the application of sensors or embedded systems to ensure efficient production identify the needs for maintenance, protect workers and the environment [2]. Standard approaches of process monitoring are the measurement or identification of the interaction between the process and machine structure.

In a "smart machine tool" the objective is to maintain an optimized cutting performance by using sensors and control systems with knowledge accumulation capability for use in future production. Vibrational behavior of the tool is of utmost importance since it significantly affects the workpiece [3]. For example, measurement of vibrations on the tool fixture is one of the indirect methods to evaluate the effects of the cutting force.

Vibration sensor signals are very sensitive to the change of workpiece dynamics, which reflects the change of cutting force due to the tool wear. During machining operation the sensors collect tool vibration signals in real-time, which are transmitted to the machine control system via feedback loop, which adjusts cutting parameters, if required, in order to reduce excessive unwanted vibrations in machine-tool-workpiece system thereby ensuring high machining quality and higher productivity. These cutting parameters may include feed rate, depth of cut, spindle speed, etc. As the sensors need to be installed near the cutting area inside the machining chamber, the wiring is an obstacle to the application of vibration measuring device in machining centers, in particular in milling machines, where the cutter and workpiece are always moving. Therefore,

wireless data transmission is an attractive solution for vibration monitoring in machining operation.

Sensor systems must be able to be interfaced with open system architecture controllers for machines and systems must be designed to accommodate needs of so called “reconfigurable” systems. Activity in both of these areas is still predominately in the research stage with few industrial applications. Accordingly, one of the main challenges in future machining process monitoring systems is the development of algorithms and paradigms that are truly autonomous from machine tool operators with signal feature extraction and decision making performed without intervention of the operator, who should provide only very simple (the lesser, the better) input and information.

Integral parts of mechatronic systems, which often determine the system's functionality and vitality, are the embedded control systems – digital hardware and software subsystem. As an integral part, mechatronic systems and embedded systems face significant challenges in information security; these systems usually have very limited resources and function in an unsafe environment. Embedded systems usually perform critical functions – control important real time objects, process important information, therefore its work can be sabotaged.

Security requirements of an embedded system’s depend on specific areas of application [4]. The following requirements are related to the general requirements for information security: integrity, availability and confidentiality. However, the specificity of mechatronic systems, their mobility and work in real time, typically have certain limitations such as processing gap, energy gap, flexibility, tamper resistance, assurance gap and cost, largely due to limited resources, performance and security requirements.

An important component of embedded systems, which often determines the system's performance and vitality is software. Software security has two aspects: secure program and program protection. We will explore the protection aspect of the program security. The main program protection vulnerabilities are [5]: Violation of intellectual property – illegal copying and distribution, improper use of licenses, and reverse engineering – disclosure of software code, theft of algorithms and falsification of software codes.

According to a study by the Business Software Alliance (BSA) [6], software creators lost 51.4 billion dollars and pirated software accounted for 43% of all software, observing approximately 2% annual growth trend of piracy.

No matter from what threats software is protected, for example copying or stealing algorithms, attackers attempt to crack the protection by several methods including reverse engineering, including disassembly and decompilation, debuggers, disassemblers, decompilers, emulators, simulators and spoofing attacks [7].

There are many software protection methods, which are divided into software-based and hardware-based.

Software-based protection mechanisms are installed into software or algorithms that are protected and can be added to software code - code and date obfuscation [8], anti-debugging method [9], code encryption technology, self-modifying code and self-extracting code [10].

Hardware-based methods can significantly increase the level of security, because it is external device in which the level of security is controlled by the software provider and not by the end-user [11, 12]. By using additional hardware (commonly Dongle or USB keys), part of the program code or data (encryption keys) required to run the program, can be stored. However, this protection mechanism is relatively expensive and is generally only used for those programs that are of great commercial value.

Intermediate software/hardware methods are also used – tethering a program to a computer or devices signatures (CPU, RAM, ROM, BIOS, OS and etc. serial numbers, model ID and so on) [13, 14, 15]. Firewalls are used for the protection of internet programs [16]. These methods are usually used for anti-piracy in personal computers.

In assessing the limitations of embedded systems [17], one of the most acceptable software protection methods is encryption of a code. However, one needs to take into account the key’s management issues; external storage medium, network – transfer must be secure, using SSL protocol and the encryption key entered manually.

Software development is one of the most challenging tasks during the design of a mechatronic system. Mechatronic system software is related to and dependent on the other system components; mechanics, electronics, controllers etc. Therefore, ranges of techniques are used for the development of mechatronic system software.

Model driven architecture is an approach to increase the quality of complex software systems based on creating high level system models that represent systems at different abstract levels and automatically generating system architectures from the models. In the papers [18, 19] is proposed a model-driven (model-based) approach to design the software part of a mechatronic system, which consists of two major parts; systematic modeling and correctness-preserving synthesis. In the paper [20] is presented an agent-based embedded control system design methodology for mechatronic systems. The paper [21] puts forward a component-based development method for increasingly complex embedded systems. Most methods used the UML *(Unified Modeling Language)* for the description of mechatronic systems.

Protection of programs is not directly related to mechatronic system functionality. In order for the developer to concentrate on the functionality, he should be free from issues related to program protection. Protection of programs must be automatically included in the system during the realization. For this it is necessary to describe the program protection requirements at a high level of mechatronic system design (UML).

Model-based approach is also widely used to create *secure software*. In the paper [22] are described processed data security and an access control requirement in the UML and OCL *(Object Constraint Language)*, each vulnerability defined by its own *stereotype*. In the paper [23] is proposed an approach to the security model as a

separate concern by augmenting UML with separate and new *diagrams* for role-based, discretionary and mandatory access controls; collectively, these diagrams provide visual access-control aspects. In the paper [24] is proposed security *primitives* (Authentication, data Integrity, data Confidentiality …) for UML; [25] defines User rights as UML and OCL *context*. The Secure UML meta-model [26] introduces *the concepts of User*, *Role*, and *Permission* to annotate UML diagrams with information pertaining to access control. In the paper [27] are described security criteria, such as confidentiality and integrity. He also defines in UMLSec a UML profile extension using *stereotypes*, *tagged values* and *constraints*.

As we can see, the UML is extended in various ways and is mainly used for creating secure software.

Our goal is to extend the model-driven embedded system development methodology measures to describe the requirements for the program protection to create a mechatronic system embedded software protection method. This method should implement a sufficient level of protection and not require additional hardware and security infrastructure.

In the following sections we describe the proposed security method of embedded software for mechatronic systems and investigate its characteristics and the possibilities of using for protection of embedded software.

2. Embedded software protection method

Protection method for mechatronic systems embedded software core is:

- Protection requirements of the program modules are described in the UML diagram by using OCL constraints.
- Installation procedure of mechatronic systems embedded software automatically integrates program protection.
- Program data and code modules are stored separately.
- Critical program modules are encrypted by symmetric algorithms independently of each other.
- Encryption keys are not stored; they are generated from the system component's signature on demand before encryption or decryption.
- Code modules are decrypted just before the execution (runtime decryption). After execution they are destroyed.

To describe the program module requirements for the protection, we extended the UML diagrams by special OCL *constraints*. These requirements, we describe in the UML class and components level, use these types of OCL constraints:

<< protectionRequirements >>
context *programModule : ProgramModule:*
self.ProtectionLayer = {1…3}
self.TimeRestrictions = real
self.SignaturesNumber = {1…7}
self.KDFfunction = {MD, SHA, SHA-2}
self.encryption = {DES, AES, Blowfish}

In the constraints there may be specified a necessary level of protection, time limitations, encryption key, the number of signatures and the generation function and the encryption algorithm. If the protections settings are not specified, then the default level of the program protection is applied.

A representation of program protection requirements in UML diagram format is shown in Fig. 3.

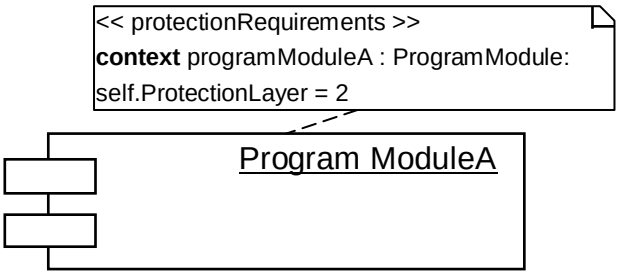


Fig. 3. Representation of program protection requirements in UML

By installing embedded software of a mechatronic system, according to a description of the UML, a special install program automatically adds the security measures, created by protection templates.

Secret keys are generated in our proposed method [28]. Secret key generation process is shown in Fig. 4.

Protection key of software module is generated according to the protecting software headers and mechatronic system hardware and software components (controller, CPU, RAM, ROM, BIOS, OS, and etc.) signatures, using the fastest and simplest logical commands (XOR, OR).

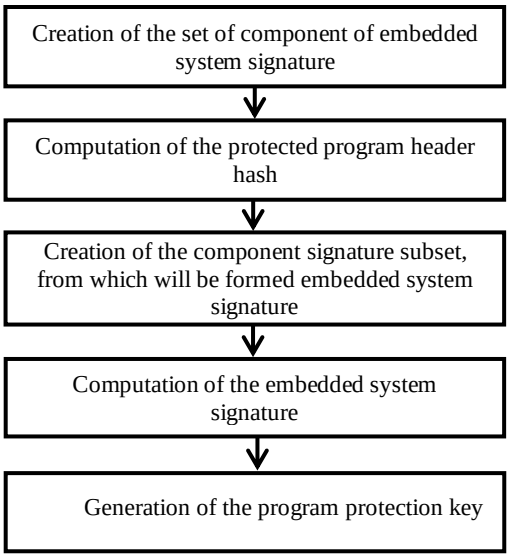


Fig. 4. Secret key generation process

The encryption key must be a fixed length and must have sufficient value of entropy. The strings of an embedded system signature are variable in length. Key Derivation Functions [29] and hash functions MD5, SHA, SHA-2 [30] are used to format fixed-length and high entropy secret keys from the variable-length strings.

The structure of the protected program is presented in Fig. 5. To increase effectiveness of the program, only critical code modules are encrypted and other modules – the program header, the data segments and non-critical modules are not encrypted.

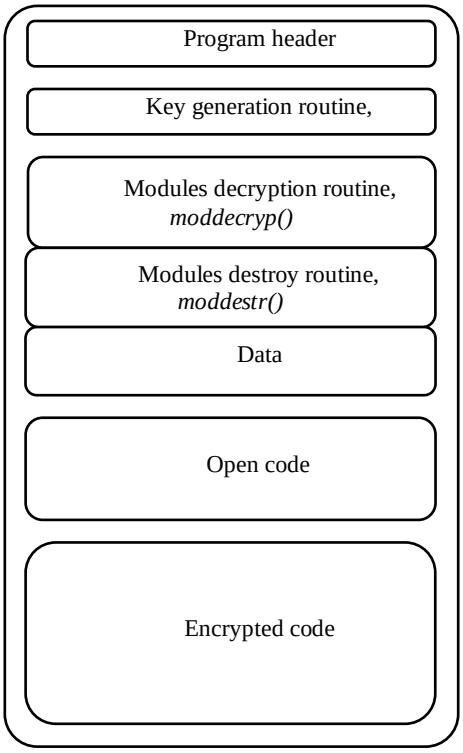


Fig. 5. The structure of the protected

Encrypted code modules are decrypted in execution time automatically. Therefore, each module includes calls to key generation and decryption routines (Fig. 6.).

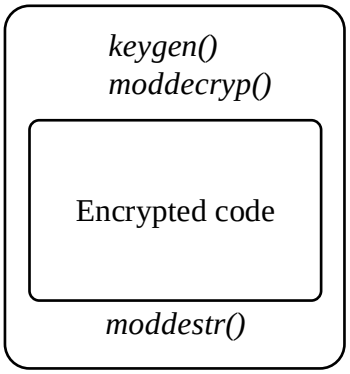


Fig. 6. The structure of the protected module

The program is protected (the required modules are encrypted) during installation in mechatronic systems by using a special software installer, whose functioning is shown schematically in Fig. 7.

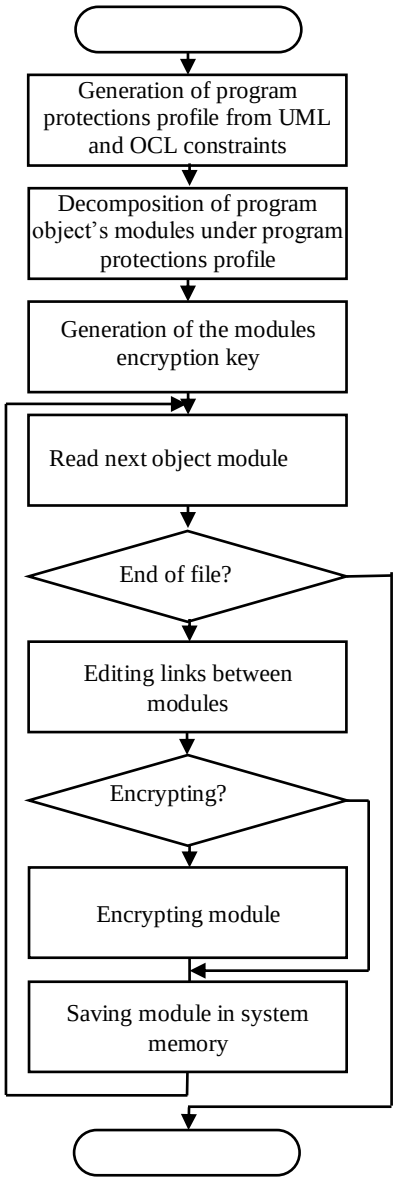


Fig. 7. The software installer operation scheme

The main steps of the installation process:

- Generation of program protections profile from UML and OCL constraints.
- Decomposition of program object's modules under program protections profile.
- Generation of the modules encryption key. Editing links between modules, encrypting and saving modules in system memory.

The next section will investigate the created method of the program protection characteristics.

3. Evaluation of embedded software protection method

For evaluation of the proposed method, we created a prototype of mechatronic system software installer that realizes the described options. We investigated the secret encryption key entropy and its dependence on the signature creation and the hash function, and the formation time. We also estimated the

impact of various encryption algorithms to operation speed of protection mechanisms; this is vital to mechatronic systems operating in real time.

The experiments were performed on the PDA *(Personal Digital Assistant)* of the model ASUS P750 (Pocket PC platform, Intel PXA270 520 MHz CPU, 256 MB RAM, Windows Mobile © 6 Professional CE OS 5.2). We simulated the software of a mechatronic system by programming discrete mathematical methods. The experiment's initial data – header of the program to be protected, mechatronic system hardware and software components signatures elements *(Vendor ID, Type ID, Model ID and Serial Number)*, their lengths and numbers generated with programmable random strings and numbers generators. 20 sets of signatures (from 2 to 7 elements) were generated.

Secret encryption keys are generated from the embedded system signature using *Key Derivation Function*. These functions use hash functions, such as MD5, SHA, SHA-2 etc. Furthermore, we investigated the influence of the hash function algorithm for the value of entropy. Since the embedded system signature, which was formatted using *sign4* function, based on OR and XOR operations [28], has the best entropy, we investigated the key generated by this function. Fig. 8 displays the entropy of keys, which was formatted from 7 component signatures, using *sign4* function and MD5, SHA and SHA-2 hash functions.

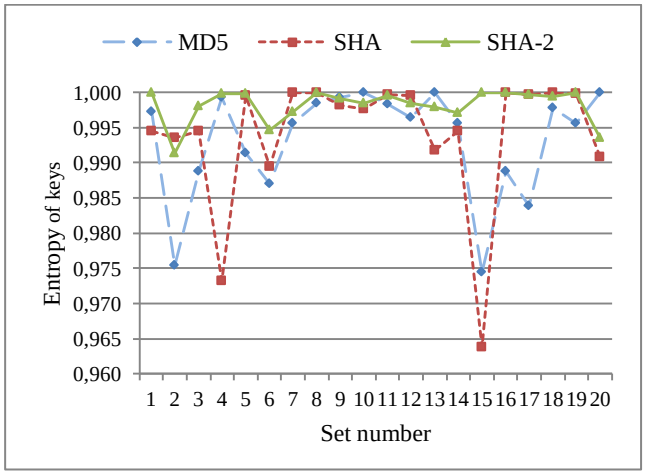


Fig. 8. Keys entropy depend on the hash functions

Entropy estimates – average, standard deviation and prediction interval depending on the hash function are shown in Table 1.

Table 1

Secret keys entropy depend on the function

Function	Average	Standard deviation	Prediction interval	
			min	max
MD5	0.994	0.008	0.985	1.000
SHA	0.995	0.007	0.988	1.000
SHA-2	0.998	0.003	0.994	1.000

All hash functions generate high-entropy cryptographic keys, however the least standard deviation (0.003) and the lower limit of prediction interval (0.994) contain keys generated using function SHA-2.

The computing time *(ms)* of the keys, which was formatted from 7 component signatures, using *sign4* and MD5, SHA and SHA-2 hash functions is shown in Fig. 9.

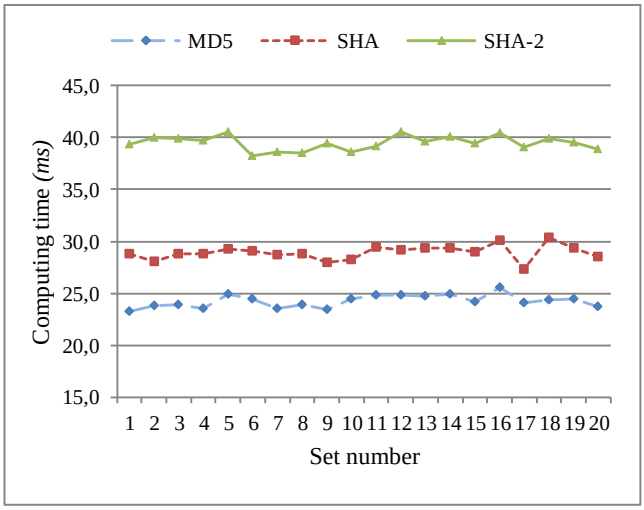


Fig. 9. Keys generation time *(ms)* dependence on the hash functions

Key computing time estimates – average, standard deviation and prediction interval, depending on the hash function are shown in Table 2.

Table 2

Keys computing time *(ms)* dependence on the function

Function	Average	Standard deviation	Prediction interval	
			min	max
MD5	23.515	0.802	22.713	24.317
SHA	28.209	0.791	27.418	29.000
SHA-2	38.805	0.867	37.938	39.672

As can be seen from Table 2, the best time characteristics were obtained by using the MD5 hash function, 65% faster than SHA-2. In the assessment of the generated key entropy (Table 1) and the generation time (Table 2), it is clear that for key generation it is better to use MD5, as the entropy is high enough, only 0.4% lower than the SHA-2, but with a much shorter generation time.

To investigate the impact of encryption algorithms to characteristics of program protection method, the simulated module solved the system of differential equations by using the Runge-Kutta method. The experiment was repeated 20 times and different algorithms were used to encrypt the module. Program execution times average and encryption module size (kB) are presented in Table 3.

As can be seen from Table 5, the best time characteristics were obtained by using the Blowfish, DES and IDEA algorithms. Blowfish are known to have better encryption (i.e. stronger against data attacks) than the other two. The Blowfish algorithm is the smallest size at 7.2 kB. It is therefore proposed to use the Blowfish algorithm to protect programs.

Table 3

Module execution time *(ms)* dependence on the encryption algorithm

	Unprotected	Encryption algorithm					
		DES	TR-DES	AES CBC	AES CFB	IDEA	Blow-fish
Average	26.5	37.9	80.0	48.1	49.3	38.6	37.6
Increase		11.4	53.5	21.6	22.8	12.1	11.1
Size kB		15	12.9	11.9	12.2	12.1	7.2

4. Conclusions

In this paper we have presented security method of embedded software for mechatronic systems. This method is based on encryption and decryption code of critical program modules during execution.

We proposed to describe protection requirements of the program modules in the UML diagram by using OCL constraints.

The proposed method effectively generates high entropy keys using the embedded system signature.

The Blowfish algorithm is the fastest and has better encryption: it is therefore proposed to use the Blowfish algorithm to protect programs.

References

1. **Lennon L.; Mass N.** 2008. Model-based Design for Mechatronics Systems. Machine Design, Embedded Systems Industry Focus – Electronics World, 23-26.
2. **Bargelis A.; Mankute R.** 2010. Impact of manufacturing engineering efficiency to the industry advancement. Mechanika, Kaunas: Technologija, Nr. 4(84): 38-44.
3. **Ubartas M.; Ostasevicius V.; Samper S.; Jurenas V.; Dauksevicius R.** 2011. Experimental investigation of vibrational drilling. Mechanika, Kaunas: Technologija, T. 17, nr. 4: 368-373.
4. **Kocher P.; Lee R.; McGraw G.; Raghunathan A.** 2004. Security as a new dimension in embedded system design. In Proceedings of the 41st annual Design Automation Conference (DAC '04). ACM, New York, NY, USA, 753-760.
5. NIST. National Vulnerability Database Version 2.2, http://nvd.nist.gov/home.cfm.
6. BSA. 2010. Seventh Annual BSA and IDC Global Software Piracy Study. 18 p.
7. **Main A.; van Oorschot P.C.** 2003. Software protection and application security: Understanding the battleground. International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography, Heverlee, Belgium. 19 p.
8. **Collberg C.; Thomborson C.; Low D.** 1997. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, the University of Auckland. 36 p.
9. **Gagnon M. N.; Taylor S.; Ghosh A. K.** 2007. Software Protection through Anti-Debugging. IEEE Security and Privacy, v.5 n.3: 82-84.
10. **Kanzaki Y.; Monden A.; Nakamura M.; Matsumoto K.** 2003. Exploiting self-modification mechanism for program protection. Proc. the 27th Annual International Computer Software and Applications Conference, Washington: IEEE Computer Society: 170-179.
11. **Jozwiak I. J.; Liber A.; Marczak K.** 2007. A Hardware-Based Software Protection Systems-Analysis of Security Dongles With Memory. Proc. the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07), Washington: IEEE Computer Society: 28-38.
12. **MeiHong L.; JiQiang L.** 2010. USB Key-Based Approach for Software Protection. International Conference on Industrial Mechatronics and Automation: 151-153.
13. **Mumtaz S.; Iqbal S.; Hameed I.** 2005. Development of a Methodology for Piracy Protection of Software Installations. 9th International Multitopic Conference, IEEE INMIC: 1-7.
14. **Liutkevičius A.; Vrubliauskas A.; Kazanavičius E.** 2011. Assessment of Dongle-based Software Copy Protection Combined with Additional Protection Methods. Electronics and Electrical Engineering. – Kaunas: Technologija – No. 6(112): 111–116.
15. PC GUARD. Professional software protection and licensing system. http://www.sofpro.com.
16. **Kazanavičius E.; Paškevičius R.; Venčkauskas A.; Kazanavičius V.** 2012. Securing Web Application by Embedded Firewall. Electronics and Electrical Engineering. – Kaunas: Technologija – No. 3(119): 65–68.
17. **Babar S.; Stango A.; Prasad N.; Sen J.; Prasad R.** 2011**.** Proposed embedded security framework for Internet of Things (IoT). Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference: 1-5.
18. **Barner S.; Geisinger M.; Buckl C.; Knoll A.** 2008. EasyLab: Model-Based Development of Software for Mechatronic Systems. Mechtronic and Embedded Systems and Applications. MESA 2008. IEEE/ASME International Conference: 540-545.
19. **Huang Jinfeng; Voeten J.; Groothuis M.; Broenink J.; Corporaal H.** 2007. A model-driven design approach for mechatronic systems. Application of Concurrency to System Design. ACSD 2007. Seventh International Conference: 127-136.

20. **Kižauskienė L.; Kazanavičius E.; Gaidys R.** 2011. Agent-based methodology for developing mechatronic systems software. Mechanika, Kaunas: Technologija, T. 17, nr. 5: 551-556.

21. **Torngren M.; DeJiu Chen; Crnkovic I.** 2005. Component-based vs. model-based development: a comparison in the context of vehicular embedded systems. Software Engineering and Advanced Applications. 31st EUROMICRO: 432- 440.

22. **Peralta K. P.; Orozco A. M.; Zorzo A. F.** 2008. Specifying Security Aspects in UML Models. In ACM/IEEE 11th International Conference on Model Driven Engineering Languages and System. Toulouse, França. Proceedings of the Workshop on Modeling Security (MODSEC08), v. 1: 1 - 10.

23. **Pavlich-Mariscal J.; Michel L.; Demurjian S.** 2007. Enhancing UML to Model Custom Security Aspects. Proceedings of the 11th International Workshop on Aspect-Oriented Modeling (AOM@AOSD'07).

24. **Nakamura Y.; Tatsubori M.; Imamura T.; Ono K.** 2005. Model-driven security based on Web services security architecture. Services Computing, 2005 IEEE International Conference, vol.1, no.: 7- 15.

25. **Alam M.M.; Breu R.; Breu M.** 2004. Model driven security for Web services (MDS4WS). Multitopic Conference. Proceedings of INMIC 2004. 8th International: 498- 505.

26. **Basin D.; Doser J.; Lodderstedt T.** 2006. Model Driven Security: from UML Models to Access Control Infrastructure. ACM Transactions on Software Engineering and Methodology (TOSEM) 15 (1): 39-91.

27. **Jürjens J.** 2002. Using UMLsec and goal trees for secure systems development, Proceedings of the 2002 ACM symposium on applied computing: 1026–1030.

28. **Venčkauskas A.; Jusas N.; Mikuckienė I.; Butleris R.** 2012. Secret encryption key generation using signature of embedded systems. Information Technology and Control. 41(xx),–. (Submitted).

29. International Organization for Standardization. 2004. ISO/IEC FCD 18033–2, IT Security techniques — Encryption Algorithms — Part 2: Asymmetric Ciphers.

30. **Henke Ch.; Schmoll C.; Zseby T.** 2008. Empirical evaluation of hash functions for multipoint measurements. SIGCOMM Comput. Commun. Rev. 38, 3: 39-50.

A. Venčkauskas, N. Jusas, L. Kižauskienė,
E. Kazanavičius, V. Kazanavičius

MECHATRONINIŲ SISTEMŲ ĮTERPTOSIOS PROGRAMINĖS ĮRANGOS SAUGOS METODAS

R e z i u m ė

Straipsnyje pateiktas mechatroninių sistemų įtertosios programinės įrangos saugos modelis, paremtas kritinių programos modulių šifravimu ir dešifravimu kodo vykdymo metu. Slaptieji šifravimo raktai nesaugomi, o generuojami pagal mechatroninės sistemos komponentų signatūras. Darbe eksperimentiškai ištirta šifravimo raktų entropija, įvairių simetrinių kriptografinių algoritmų taikymo galimybės ir apsaugos priemonių įtaka įterptosios programinės įrangos charakteristikoms – greitaveikai ir papildomos atminties sąnaudoms.

A. Venčkauskas, N. Jusas, L. Kižauskienė,
E. Kazanavičius, V. Kazanavičius

SECURITY METHOD OF EMBEDDED SOFTWARE FOR MECHATRONICS SYSTEMS

S u m m a r y

This paper proposes embedded software of mechatronic system protection method based on encryption and decryption code of critical program modules during runtime. Secret keys are not stored, but generated by the signature of mechatronic system components. This paper experimentally researches the application of symmetric cryptographic algorithms and the influence of security mechanisms on characteristics (value entropy of secret key, operating speed, and amount of memory) of embedded software.

Keywords: mechatronic system, embedded software, security, secret key generation.

8.3. Publikacija „Programos apsaugos metodas, naudojant sistemos parašą“

1. Šiame priede yra pateikiamas mokslinis straipsnis išspausdintas 17-osios Tarpuniversitetinės magistrantų ir doktorantų mokslinės konferencijos "Informacinės technologijos" pranešimų medžiagoje. ISSN 2029-249X, 2012, p. 153-156.

PROGRAMOS APSAUGOS METODAS, NAUDOJANT SISTEMOS PARAŠĄ

Nerijus Jusas[1], Algimantas Venčkauskas[2]

[1]*Kauno Technologijos Universitetas, Kompiuterių katedra, Studentų g. 50, Kaunas, Lietuva, nerijus.jusas@stud.ktu.lt*
[2]*Kauno Technologijos Universitetas, Kompiuterių katedra, Studentų g. 50, Kaunas, Lietuva, algimantas.venckauskas@ktu.lt*

Santrauka (abstract). Svarbi taikomųjų programų saugos problema yra pačios programos apsauga nuo nelegalaus jos naudojimo ir platinimo. Šiame straipsnyje aprašytas programų apsaugos metodas –atskirų programos modulių šifravimas. Šifravimo raktai generuojami naudojant maišos funkcijas iš sistemos parašo, kuris sudaromas iš sistemos komponentų identifikacinių duomenų.

Raktiniai žodžiai: programų apsauga, slaptų raktų generavimas, kriptografija.

Įvadas

Taikomųjų programų kūrimui reikia investuoti daug laiko, pinigų ir pastangų. Nelegalus programinės įrangos naudojimas ir kopijavimas, žinomas kaip „programų piratavimas“, yra viena [10] didžiausių problemų su kuria susiduria taikomųjų programų kūrėjai. BSA tyrimai [1] parodė, kad pasauliniu mastu 43% programinės įrangos yra naudojama nelegaliai ir kasmet piratavimo mastai auga 2%. Be piratavimo, programų kūrėjai kenčia ir nuo intelektinės nuosavybės pažeidimų, algoritmų vagysčių.

Programinės įrangos apsaugos nuo kopijavimo ir algoritmų vagysčių mechanizmų sutrikdymui naudojami atvirkštinės inžinerijos, programų kodo vykdymo eigos sekimo (derinimo), mašininio kodo vertimo į asemblerio kalbą metodai [11]. Iš išvardintų nulaužimo metodų dažniausiai yra naudojama atvirkštinė inžinerija ją papildant kitais dviem. Atvirkštinės inžinerija – nagrinėja dvejetainį failą, paversdama mašininį kodą į asemblerio kalbos kodą, kad būtų lengviau nustatyti kaip veikia programa [12]. Atlikus tokius veiksmus gali būti rasti programų apsaugos apėjimo būdai ar net sugeneruotas pradinis programos kodas.

Programų apsaugai yra sukurta daug teorinių ir praktinių apsaugos metodų. Visi šie metodai yra skirstomi į dvi dideles grupes: programiniai ir aparatūriniai.

Aparatūriniai apsaugos metodai naudoja papildomus įrenginius. Tokiai apsaugai dažniausiai yra naudojami USB atmintinės, apsaugos *(Dongle)* raktai. Dongle raktai gali žymiai padidinti programos apsaugą todėl, kad tai yra išorinis įrenginys, kurio teikiamos apsaugos sprendimai yra valdomi programinės įrangos kūrėjo, o ne galutinio vartotojo [3]. Tačiau tokie apsaugos mechanizmai yra naudojami taikomosioms programoms, kurios turi didelę komercinę vertę, nes šių įrenginių kaina didelė.

Programiniai apsaugos mechanizmai yra įdiegiami į taikomąsias programas. Taikomųjų programų apsaugai dažniausiai yra taikomi programinio kodo supainiojimo, derintuvų aptikimo, apsaugos nuo programinio kodo išskaidymo ir programinio kodo glaudinimo ir šifravimo metodai.

Programinio kodo supainiojimo metodas naudoja tam tikras transformacijas [4], kurios pradinį programos kodą pertvarko į „supainiotą“ kodą, kuri yra sunkiau suprantamas ir programa tampa atsparesnė atvirkštinei inžinerijai. Atlikus tokias transformacijos pasikeičia tik taikomosios programos kodo išvaizda, tačiau veikimas išlieka toks pats. Kodo supainiojimo technologijos dažniausiai yra naudojamos programoms sukurtoms .NET, Java ir kitomis interpretacinėmis programavimo priemonėmis.

Apsauga nuo programų kodo vykdymo eigos sekimo (derinimo) apima metodus ir gudrybes, kurios apsunkina derintuvų *(debugger)* ir atvirkštinės inžinerijos naudojimą [5]. Šiomis priemonėmis siekiama uždrausti naudotis derintuvais. Programos gali būti sukurtos taip, kad aptiktų techninio, programinio ir vartotojo lygio derintuvus. Derintuvai yra aptinkami, nes tai yra programos veikiančios darbinėje atmintyje ir paliekančios tam tikrus pėdsakus, pakeičia atitinkamas procesoriaus vėliavėles, jei programa yra derinama. Taikomoji programa, apsaugota nuo derinimo priemonių, aptikusi derintuvą gali imtis įvairių veiksmų: išjungti derintuvą, blokuoti ar sunaikinti programą.

Dažnai naudojamas programų „nulaužimo“ metodas yra vykdomojo kodo skaidymas į dalis ir pavertimas į asemblerio kalbą. Toks pavertimas dažniausiai yra atliekamas naudojant linijinio nuskaitymo *(liner-scanning)* ir rekursijos žingsnių *(recursion-marching)* technologijas [6]. Apsaugai nuo tokio programos nulaužimo yra naudojami metodai, kurie padaro programinį kodą, paverstą į asemblerio kalbą, sunkiai arba visai nesuprantamą. Tam tikslui yra naudojami du metodai: bevertis kodas [7] ir valdymo srauto slėpimas [8].

Naudojant programinio kodo glaudinimo ir šifravimo technologijas pradinis programos kodas yra suglaudinamas arba / ir užšifruojamas. Pertvarkytas kodas ir duomenys susiejami su atstatymo veiksmais. Taip

pertvarkytos programos kodas ir duomenys vykdymo metu yra atstatomi į pradinį kodą. Toks suglaudintas kodas gali būti laikomas savaime išsiskleidžiamame *(selft-extracting)* archyve, kur duomenys yra suglaudinami kartu su atitinkamu iššifravimo raktu. Vien programos kodo suspaudimas ar užšifravimas apsunkina atvirkštinę inžineriją, kadangi toks kodas negali būti tiesiogiai paverstas į asemblerio kalbą. Be to, šią technologiją galima lengvai panaudoti kartu su aukščiau aprašytomis apsaugos priemonėmis. Todėl panaudojus įvairias jų kombinacijas galima pasiekti gerų rezultatų kovoje prieš atvirkštinę inžineriją. Tačiau toks apsaugos metodas turi ir problemų – kaip generuoti ir kur saugoti reikalingus raktus.

Raktų generavimas yra sunkus uždavinys. Dažniausiai raktai generuojami naudojant įvairius pseudoatsitiktinius generatorius, kurie esant tiems patiems įėjimo duomenims pateikia tuos pačius išėjimo duomenis [14]. Taip pat raktai generuojami panaudojant elektroninių prietaisų fizikines savybes, pvz., sistemos įrenginių veikimo ar realaus laiko laikrodžio dažniai. Raktus galima sudaryti ir iš ROM, RAM, SRAM ir kitų atminties modulių gamybos ypatumų, kadangi gamybos proceso metu nėra gaunama visiškai vienodi atminties modulio tranzistoriai, taip yra dėl silicio savybių [13, 15]. Pastarieji metodai dažniausiai yra naudojami įterptosiose sistemose. Naudojant papildomą aparatūrinę įrangą raktus galima generuoti iš tam tikrų žmogaus fiziologių savybių – pirštų atspaudai, veido bruožai, akies rainelė ir t.t. Tačiau tokiam generavimui reikalinga aparatūrinė įranga, kuri yra pakankamai brangi [16].

Mūsų tikslas sukurti pakankamo lygio programinės įrangos apsaugos metodą, nereikalaujantį papildomos aparatinės įrangos ir infrastruktūros apsaugos raktų generavimui ir saugojimui. Programos kodas saugomas šifruotoje formoje, šifravimo raktai generuojami realiame laike, pagal poreikį, prieš vykdant šifruotą programos modulį.

Programos modulių šifravimo raktų generavimas, naudojant sistemos parašą

Siūlomas apsaugos metodas yra pagrįstas atskirų programos modulių šifravimu. Programos modulių apsaugos raktas generuojamas pagal apsaugomos programos antraštės ir sistemos aparatinių ir programinių komponentų, CPU, RAM, ROM, BIOS, OS, ir t.t., identifikacinius duomenis: modelis, gamintojas, serijinis numeris ir t.t., naudojant logines komandas ir maišos funkcijas. Šifravimui yra naudojama simetriniai algoritmai, nes jie greitesnis už asimetrinius. Šifravimo algoritmas parenkamas atsižvelgiant į jo greitaveiką ir reikiamą programos apsaugos lygį.

Slaptų raktų generavimo procesas susideda iš 6 žingsnių, kurie yra paaiškinti žemiau. Proceso aprašymui, pateiksime pavyzdį, kai sistemos parašas bus sudaromas iš 4 komponentų identifikacinių duomenų, naudojant XOR logines operacijas. Parašą sudarančių komponentų skaičius gali būti koks nori. Pirmame – ketvirtame žingsniuose nustatoma kokių sistemos komponentų parašai bus naudojami slaptojo šifravimo rakto generavimui. Penktame – šeštame žingsniuose apskaičiuojamas raktas.

1. Sudaromas sistemos komponentų parašų sąrašas:
$$KSS = \{KSS_i\},\ i=,\ldots,n;$$
2. Suskaičiuojama apsaugomos programos antraštės maišos reikšmė:
$$PH = h(PA),$$
kur *PH* gauta maišos reikšmė, *h()* – maišos funkcija, *PA* – programos antraštė.
3. Priklausomai nuo to kiek sistemos komponentų parašų naudojama sistemos parašo sudarymui, sudaroma atitinkamo dydžio matrica. Kadangi parašo sudarymui naudosime 4 komponentų parašus, tai sudarysime matricą 4x4. Ji užpildoma 2 žingsnyje gautos maišos reikšmės. Tarkime, kad maišos reikšmė buvo gauta tokio *8eeba89186027b5630c3b122d8cf746f* (čia ir toliau skaičiai pateikti šešioliktainėje skaičiavimo sistemoje), tada užpildyta matrica atrodys taip:

8	e	e	b
a	8	9	1
8	6	0	2
7	b	5	6

4. Matricos reikšmes sudedame. Sudėti galima stulpeliais arba eilutėmis. Sudėjus stulpeliais gauname:

8	e	e	b
a	8	9	1
8	6	0	2
7	b	5	6
21	27	1c	14

Gautas sumas padaliname moduliu (%) iš pasirinkto skaičiaus. Šis skaičius gali būti pasirenkamas ar sugeneruojamas pagal poreikį. Šiuo atveju tegu šis skaičius bus *8*, tai yra pirma maišos reikšmė. Atlikus veiksmus gauname:

$$21 \,\%\, 8 = 1$$
$$27 \,\%\, 8 = 7$$
$$1c \,\%\, 8 = 4$$
$$14 \,\%\, 8 = 4$$

5. Ketvirtame žingsnyje gavome, kad sistemos parašas bus sudaromas iš komponentų parašų, kurie *KSS* sąraše yra įrašytos 1,7 ir 4 numeriu. Nustačius komponentų parašus, reikalingus sudaryti sistemos parašą, juos sudedame pasirinktomis loginėmis operacijomis ir pasirinktu jų išdėstymu. Sistemos parašas *SP* bus:
$$SP = KSS_1 \oplus KSS_7 \oplus KSS_4 \oplus KSS_4$$
6. Galutinis šifravimo raktas *GSR* yra gaunamas sistemos parašui suskaičiuojant maišos reikšmę:
$$GSR = h(SP);$$

Sugeneruotas raktas GSR naudojamas apsaugomos programos modulių šifravimui ir iššifravimui programos vykdymo metu.

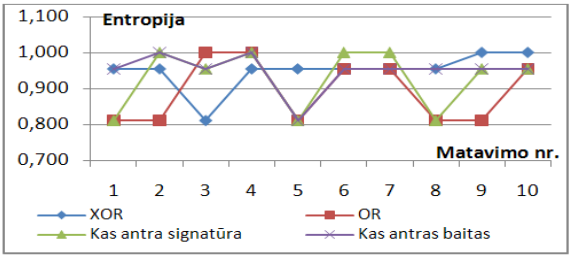
Eksperimentas

Tyrimo metu pradinius duomenis – saugomos programos antraštes, sistemos aparatinių ir programinių komponentų identifikacinius duomenis *(Gamintojo ID, Tipo ID, MOdelio ID ir Serijinius Numerius)*, jų ilgius ir skaičių generavome atsitiktinių skaičių (baitų eilučių) programiniais generatoriais.

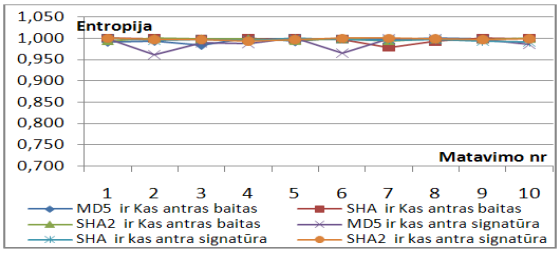
Eksperimento metu buvo generuojama programos antraštė ir komponentų parašai, sistemos parašas buvo formuojamas iš sugeneruoto komponentų parašų sąrašo atsitiktinai parenkant 3 elementus. Sudarytas sistemos parašas ir sugeneruotas slaptasis šifravimo raktas buvo vertinami skaičiuojant entropiją [17]. Eksperimentas buvo kartojamas 10 kartų. Gauti rezultatai pateikti lentelėje Nr. 1 ir 1 ir 2 pav.

Lentelė Nr. 1. Sistemos parašo ir šifravimo rakto entropija

	Loginių operacijų naudojimo variantai					Šifravimo rakto entropija					
						XOR			Kas antra baitas		
Matavimo nr.	Bitų sk.	XOR	OR	Kas antra signatūra	Kas antras baitas	MD5	SHA	SHA2	MD5	SHA	SHA2
1	120	0,954	0,811	0,811	0,954	0,991	1,000	0,994	1,000	1,000	1,000
2	120	0,954	0,811	1,000	1,000	0,993	1,000	1,000	0,961	0,995	0,996
3	256	0,811	1,000	0,954	0,954	0,984	0,996	0,999	0,990	0,997	0,998
4	64	0,954	1,000	1,000	1,000	1,000	1,000	0,999	0,987	0,996	0,993
5	256	0,954	0,811	0,811	0,811	0,992	0,999	0,995	1,000	1,000	0,997
6	64	0,954	0,954	1,000	0,954	1,000	0,998	0,999	0,965	0,998	1,000
7	64	0,954	0,954	1,000	0,954	0,998	0,979	0,994	0,998	0,996	1,000
8	120	0,954	0,811	0,811	0,954	1,000	0,993	0,998	0,999	0,997	0,999
9	64	1,000	0,811	0,954	0,954	0,998	1,000	0,998	0,998	0,993	0,997
10	120	1,000	0,954	0,954	0,954	0,999	1,000	1,000	0,986	0,991	0,999
	Vidurkis	0,949	0,892	0,930	0,949	0,996	0,996	0,998	0,988	0,996	0,998



1 pav. Sistemos parašo entropijos kitimas



2 pav. Sistemos šifravimo rakto entropijos kitimas

Kaip matyti iš lentelės Nr. 1 ir 1 pav., mažiausia entropija sistemos parašų, kurie buvo sudaryti naudojant OR loginę operaciją, naudojant ją gauta daug reikšmių, kurios yra lygios 0,811. Aukšta entropija sistemos parašų, sudarytų naudojant XOR operaciją ir sumuojant kas antras baitą operacijomis XOR ir OR.

Taip pat įvertinome kokią reikšmę turi maišos algoritmas generuojant šifravimo raktą. Tyrėme du sistemos parašo sudarymo metodus: XOR operacija ir kas antrą baitą sumavome XOR ir OR operacijomis. Kaip matyti iš lenteles Nr. 1 ir 2 pav., maišos funkcija pagerina šifravimo rakto entropiją: pirma, entropijos tapo stabilesnės – visos

reikšmės telpa intervale [0,961; 1]; antra, padidėjo šifravimo rakto entropija palyginus su sistemos parašu. Eksperimentas parodė, kad šifravimo rakto entropijai naudojamas maišos algoritmas turi nedidelę reikšmę, tačiau geriausi rezultatai gauti naudojant algoritmą SHA2.

Išvados

Šiame straipsnyje pristatytas slaptų raktų generavimo pagal sistemos parašą metodas, įvertinta raktų entropija. Pasiūlytas metodas generuoja aukštos entropijos raktus be papildomų aparatinių ir infrastruktūros sąnaudų, tai svarbu taikomųjų programų apsaugai sistemose su ribotais ištekliais.

Ateityje, naudojant pasiūlytą slaptų raktų generavimo metodą, ištirsime jo pritaikymo galimybes įterptųjų sistemų, kuriose yra specifiniai ribojimai ištekliams – atminčiai, greitaveikai ir panašiai, programinės įrangos apsaugai.

Literatūros sąrašas

[1] **BSA.** *Seventh Annual BSA and IDC Global Software Piracy Study.* May 2010. 18 p.

[2] **Jiutao T., Guoyuan L.** Research of Software Protection. *International Conference on Educational and Network Technology (ICENT 2010)*, IEEE Computer Society, 2010, p. 410- 413.

[3] **Jozwiak I. J., Liber A., Marczak K.** A Hardware-Based Software Protection Systems-Analysis of Security Dongles With Memory. *Proc. the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07)*, Washington: IEEE Computer Society, 2007, 28-38.

[4] **Chen H., Yuan L., Huang B., Yew P**. Control Flow Obfuscation with Information Flow Tracking. *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, p. 391-400.

[5] **Ceccato M., Di Penta M., Nagra J.** The Effectiveness of Source Code Obfuscation: an Experimental Assessment. *ICPC 2009, IEEE Computer Society*, 2009, p. 178-187.

[6] **Chen H. Y., Hou T.W.** Changing Data Type Method of Data Obfuscation on Java Software. *International Computer Symposium*, 2004, p. 439 – 442.

[7] **Tsai M. H., Chen H. Y., Hou T. W.** Three methods of control flow obfuscation on Java software. *International Computer Symposium*, 2004, p. 318 – 324.

[8] **Gagnon M. N., Taylor S., Ghosh A. K.** Software protection through anti-debugging. *Security & Privacy, IEEE Computer Society*, 2007, p. 82-84.

[9] **Kim M.J., Lee J. Y., Chang H. Y.** Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering. *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, IEEE*, 2010, 80-86.

[10] **Mumtaz S., Iqbal S., Hameed E. I.** Development of a Methodology for Piracy Protection of Software Installations. *9th International Multitopic Conference, IEEE INMIC 2005*, 24-25 Dec. 2005, 1-7.

[11] **Main A., van Oorschot P.C.** Software protection and application security: Understanding the battleground. *International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography, Heverlee, Belgium*, June 2003. 19.

[12] **Birrer B., Raines R., Baldwin R., Mullins B., Bennington R.** Program Fragmentation as a Metamorphic Software Protection. *Third International Symposium of Information Assurance and Security*, 2007, 369-374.

[13] **Suh G. E., Devadas S.** Physical Unclonable Functions for Device Authentication and Secret Key Generation. *Design Automation Conference (DAC 2007)*, 2007, 9-14.

[14] **Arnault F., Berge T. P.** Design and Properties of a New Pseudorandom Generator Based on a Filtered FCSR Automaton. *IEEE Transactions on Computers*, 2005, 1374-1383.

[15] **Kursawe K., Sadeghi A. R., Schellekens D., Škoric B., Tuyls P.** Reconfigurable Physical Unclonable Functions Enabling Technology for Tamper-Resistant Storage. *Hardware-Oriented Security and Trust*, 2009, 22-28.

[16] **You L., Zang G., Zhang F.** A fingerprint and threshold scheme-based key generation method. *Computer Sciences and Convergence Information Technology (ICCIT)*, 2010, 615- 619.

[17] **Wanli Ma, Campbell J., Tran D., Kleeman D**. Password Entropy and Password Quality. *Network and System Security (NSS), 2010 4th International Conference on* 1-3 Sept. 2010, 583-587.

Program Protection Method Using Signature of System

Important programs security problem is the protection of program from illegal use and distribution. This article describes the method of protection program – separate program modules are encrypted. Encryption keys are generated using a hash function from a system's signature, which is composed of components of the system identification data.

8.4. Publikacija „Secret encryption key generation using signature of an embedded system“

Šiame priede yra pateikiamas mokslinis straipsnis, kuris priimtas spausdinimui žurnale „Informacinės technologijos ir valdymas“. 2012 m, 41(xx).

SECRET ENCRYPTION KEY GENERATION USING SIGNATURE OF AN EMBEDDED SYSTEM

Algimantas Venčkauskas[1], Nerijus Jusas[1],
Irena Mikuckienė[2], Rimantas Butleris[3]
[1]Computer Department, Kaunas University of Technology,
Studentų str. 50-209, LT-51368, Kaunas, Lithuania
E-mail: algimantas.venckauskas@ktu.lt, nerijus.jusas@stud.ktu.lt
[2]System Analysis Department, Kaunas University of Technology,
Studentų str. 50-211, LT-51368, Kaunas, Lithuania
E-mail: irena.mikuckiene@ktu.lt
[3]Department of Information Systems, Kaunas University of Technology,
Studentų str. 50-311, LT-51368, Kaunas, Lithuania
E-mail:rimantas.butleris@ktu.lt

Abstract. Program protection, programming code integrity and intellectual property protection are important problems in embedded systems. Security mechanisms for embedded systems have some specific restrictions related to limited resource, bandwidth requirements and security. In this paper we develop a secret encryption key generation algorithm using the signature of an embedded system. We have explored the qualitative characteristics of the generated keys - the entropy. Experiments showed that generated secret keys have high entropy.

Keywords: embedded system, program protection, secret encryption key, hash function, entropy.

Introduction

It is hard to imagine most of today's appliances and devices, electronics, telecommunications, mechatronics and so on, without embedded systems. These systems face significant challenges in information security; on one hand they usually have very limited resources and on the other hand they function in a physically unsafe environment. Embedded systems usually perform critical functions: controlling important real time objects and processing important information. Therefore, their work is open to sabotage.

Security requirements of embedded systems depend on specific areas of application [16, 26]. The following requirements are related to the general requirements for information security: integrity, availability and confidentiality. However, the specificity of embedded systems, their mobility and operation in real time, typically have certain limitations such as processing gap, energy gap, flexibility, tamper resistance, assurance gap and cost. This is largely due to limited resources, performance and security requirements.

An important component of an embedded system that also influences its performance and vitality is software. Software security has two aspects: secure program and program protection [19]. We will explore the protection aspect of program security. The main program protection vulnerabilities are the following [23]: violation of intellectual property (illegal copying and distribution, improper use of licenses, reverse engineering), disclosure of software code, theft of algorithms and falsification of software code.

According to a study by the Business Software Alliance (BSA) [4], software creators lost USD 51.4 billion and pirated software accounted for 43 % of all software with piracy growing by around 2 percent annually.

No matter what threats software is protected from, for example copying or the theft of algorithms, attackers use a wide range of means to crack the protection: reverse engineering, including disassembly and decompilation, debuggers, disassemblers, decompilers, emulators, simulators and spoofing attacks [18].

There are many software protection methods, which are divided into software and hardware-based. Software-based protection mechanisms are integrated into software or an algorithm, which is protected and can be added to the software code: code and date obfuscation [6], anti-debugging method [7], code encryption technology, self-modifying code and self-extracting code [13]. Hardware-based methods can significantly increase the level of protection, largely due to the fact that they are external devices in which the level of protection is controlled by a software provider and not by the end-user [12, 17, 20]. Part of the program code or data (encryption keys) required to run a program can be stored in additional hardware (commonly Dongle or USB keys). However, this protection mechanism is relatively expensive and is generally only used for those programs which are of great commercial value.

Intermediate software/hardware methods are also used; tethering a program to a computer or device signatures (CPU, RAM, ROM, BIOS, OS etc. serial numbers, model ID and so on) [21, 25]. Firewalls also are used for the protection of internet programs [14]. These methods are usually used for anti-piracy in personal computers.

In paper [8] a joint compiler/hardware infrastructure for embedded system software protection for fully encrypted execution in which both program and data are in encrypted form in memory is proposed. The processor is supplemented with a Field Programmable Gate Array (FPGA) based secure hardware component. In paper [1] architecture for hardware-assisted run-time monitoring is presented, wherein the embedded processor is augmented

with a hardware monitor that observes the processor's dynamic execution trace and checks whether the execution trace falls within the allowed program behaviour.

In assessing the limitations of embedded systems [2], one of the most acceptable software protection methods is code encryption. However, key management faces additional issues: it requires an additional storage medium, the encryption keys have to be entered manually, key transfer via the network must be protected using SSL protocol and so on.

Secret encryption keys are used for various purposes in embedded systems, such as communication, data encryption etc. Physical embedded system characteristics such as physical unclonable functions (PUF) are used for the generation of keys [24, 27].

Our goal is to create an embedded system software protection method that does not require external hardware and infrastructure for key generation, storage and management and allows a sufficient level of security. The embedded system software code is stored in an encrypted form; secret encryption keys are generated in real time, on demand, before execution of an encrypted software module.

In the following sections we describe the proposed method of secret key generation using the signature of an embedded system and investigate its characteristics and possibilities of its application for the protection of embedded system software.

Method of secret key generation using the signature of an embedded system

The secret encryption key of a software module is generated from the headers of programs to be protected and embedded system hardware and software component (CPU, RAM, ROM, BIOS, OS etc.) signatures, using the fastest and simplest logical commands (XOR, OR, SHIFT). For convenience of description, terminology and notations used in the paper are summarized as follows:

– $K = \{k_i\}$: program encrypting key.
– $P = \{p_i\}$: header of program to be protected. An executable or shared object file's program header is an array of structures, each describing a segment or other information the system needs to prepare the program for execution.
– $PSN = \{psn_i\}$: serial number of program to be protected.
– $PH = \{ph_i\}$: hash of header of program to be protected.
– $SS = \{ss_i\}$: signature of embedded system.
– $ES = \{es_i\}$: signature of embedded system components.
– $CV = \{cv_i\}$: component vendor identifiers (ID).
– $CT = \{ct_i\}$: component type ID.
– $CM = \{cm_i\}$: component model ID.
– $CSN = \{csn_i\}$: component serial number.
– $\|$: string catenation operation.
– $\vee$: the bitwise OR operation.
– $\oplus$: the bitwise exclusive OR (XOR) operation.
– *mod n*: division by module *n* operation.
– *h(.)*: a cryptographic one-way hashing function (MD5, SHA-1…).
– *eb(s, k)*: the *k-th* byte extraction from the string *s function.*
– *sign(.)*: function for creating the signature of embedded system (defined below).
– *key(.)*: function for generation of secret encryption key (defined below).

Secret key generation process consists of 5 steps (Fig. 1).

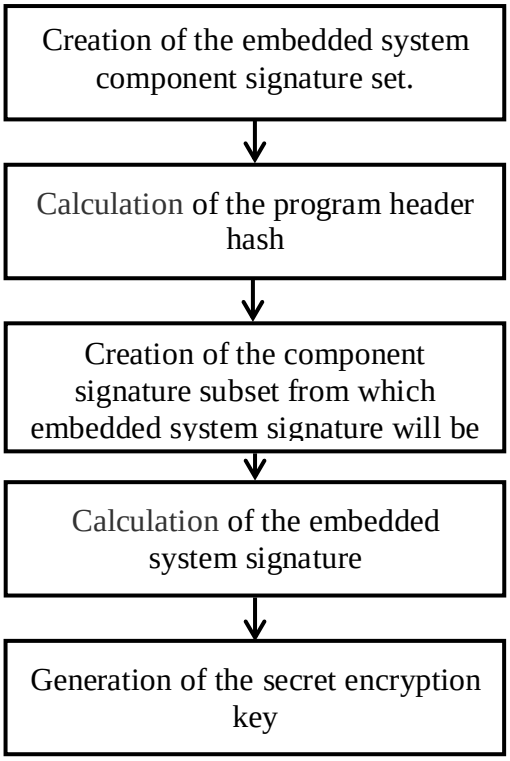


Figure 1. Secret keys generation

Further, the method of generating a secret key using the signature of an embedded system is described in detail.

1. Creation of $ES = \{es_i\}, i = 1, \dots, n$ – set of signatures of embedded system components. The component signature is created by the string catenation of the following components: *Vendor ID, Type ID, Model ID* and *Serial Number*:

$$es_i = cv_i \| ct_i \| cm_i \| csn_i.$$

Steps 2 – 6 of the algorithm define a subset of the component signatures, from which embedded system signatures will be computed.

2. The program header hash is calculated:

$$ph = h(p \| psn).$$

3. The matrix $MH = \{mh_{i,j}\}, i = 1, \dots, n;\ j = 1, \dots, m,$ from program header hash bytes is created:

$mh_{i,j} = eb(ph, (i-1) \times j + i)$, where n- amount of embedded system signatures, m – is calculated by: $m = eb(ph, n)\ mod\ n$.

4. The sum of columns in the matrix MH $s_j, j = 1, \dots, m$ is calculated:

$$s_j = \sum_{i=1}^{n} mh_{ij}$$

5. The index array of component signatures is created, repetitive index is deleted.

$IND = \{ind_j\},\ kur\ ind_j = s_j\ mod\ n$ and $ind_j \neq ind_i, \forall\ i \in \{1 \dots j-1\}$.

6. The component signatures subset, from which the embedded system signature will be formed, is created:

$$\widetilde{ES} \subseteq ES, \widetilde{es_\iota} = es_j, where\ j = ind_k,$$
$$\forall\ ind_k \in IND,\ \ k = 1, \dots, m.$$

7. The embedded system signature is created:

$$ss = sign(\widetilde{ES})$$

8. The program protection key is generated:

$k = key(ss, p1, p2)$, *p1* and *p2* defined below.

Further, the set of *sign* functions are described. Embedded system signatures are created by processing byte strings of component signatures. All *sign* functions can be performed using bit operations (bitwise OR, bitwise XOR, bitwise AND, SHIFT).

Function *sign1*. Bitwise XOR is used to create an embedded system signature (Fig. 2).

```
input: ess, m // subset of m component
                  // signatures
output: ss // embedded system signature
l = maxlength ess // max of component signatures
                  // length
for j = 1 to l do
  ss (j) = ess (1, j)
end for
for i = 2 to m do
  for j = 1 to l do
    ss (j) = ss (j) XOR ess (i, j)
  end for
end for
```

Figure 2. Function ***sign1***

Function *sign2*. Bitwise OR is used to create an embedded system signature (Fig. 3).

Function *sign3*. Bitwise OR and XOR is used to create an embedded system signature (every other component signature uses XOR or OR operation, started from XOR, Fig. 4).

```
input: ess, m // subset of m component
                  // signatures
output: ss // embedded system signature
l = maxlength ess // max of component signatures
                  // length
for j = 1 to l do
  ss (j) = ess (1, j)
end for
for i = 2 to m do
  for j = 1 to l do
    ss (j) = ss (j) OR ess (i, j)
  end for
end for
```

Figure 3. Function ***sign2***

```
input: ess, m // subset of m component
                  // signatures
output: ss // embedded system signature
l = maxlength ess // max of component signatures
                  // length
for j = 1 to l do
  ss (j) = ess (1, j)
end for
for i = 2 to m-1 step 2 do
  for j = 1 to l do
    ss (j) = ss (j) XOR ess (i, j)
  end for
  for j = 1 to l do
    ss (j) = ss (j) OR ess (i+1, j)
  end for
end for
```

Figure 4. Function ***sign3***

Function *sign4*. Bitwise OR and XOR are used to create an embedded system signature (every other byte of component signature uses XOR or OR operation, started from XOR, Fig. 5).

```
input: ess, m // subset of m component
                  // signatures
output: ss // embedded system signature
l = maxlength ess // max of component signatures
                  // length
for j = 1 to l do
  ss (j) = ess (1, j)
end for
for i = 2 to m do
  for j = 1 to l-1 step 2 do
    ss (j) = ss (j) XOR ess (i, j)
    ss (j+1) = ss (j+1) OR ess (i, j+1)
  end for
end for
```

Figure 5. Function ***sign4***

The length of signatures created using *sign* functions is equal to the maximum length of the component's signature:

$$l = maxlength(es_i), es_i \in \widetilde{ES}\ ,\ \ i = 1, \dots, m.$$

The strings of embedded system signatures are of variable length. The encryption key should contain fixed length strings. The length of these strings depends on the requirements of encryption key length. The encryption key must have a high value of entropy. Hash functions such as MD5 and SHA are used to format fixed length secret keys from the variable length strings [9]. However, the keys generated only with a hash function are not strong enough against brute force attacks [11, 29]. *Key Derivation Functions* [10] generate strong keys, where besides bytes strings, extra data (*random salt* and *an iteration count)* is also used. The main problem of these functions is how to define this additional data: *salt* and *iteration count* [15]. Our proposed method as a *salt* use a number of columns of matrix *MH*, and as a *count iteration*, the count of component signatures subset $(\widetilde{ES})$:

$$salt = eb(ph, n)\ mod\ n,$$
$$count\ iteration = count(\widetilde{ES}).$$

The assessment of what is stated above, the function of generation of the program protection key is defined as follows:

$$k = h\ (ss \| salt \| count\_iteration, keylength).$$

In the following section, we investigate the characteristics of the method of secret key generation that uses the signature of an embedded system.

Evaluation of the method of encryption secret key generation using the signature of embedded systems

Encryption keys need to be generated truly *randomly* and to contain sufficient *entropy* and *length* [28]. Since our problem is embedded systems, the generation process must satisfy the additional constraints: *processing gap, battery gap, flexibility, tamper resistance, assurance gap and cost* [2, 26]. The proposed method generates the secret keys only from the signature of the embedded system. Further, we investigate whether the entropy of a key is high enough because the value of entropy must be close to 1. The entropy of a bit string is calculated so the maximum value can be 1 [28, 5, 3].

The initial data of the experiment (the header of the program to be protected, embedded system hardware and software component signature elements (*Vendor ID, Type ID, Model ID* and *Serial Number*), their lengths and numbers) is generated with programmable random string and number generators. 20 sets of signatures (from 2 to 7 elements) are generated. We evaluated the signatures of a system that were formed from 2 to 7 elements, for the reason that the entropy changes marginally when the number of elements increases.

The entropy of these signature sets is presented in tables 1-4, (number of component signatures, length of system signature and entropy).

Table 1. Signatures entropy, *sign1* function

	n=2		*n=3*		*n=4*		*n=5*		*n=6*		*n=7*	
Test N.	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy
1	120	1,000	120	0,954	120	0,954	120	0,954	256	1,000	256	0,544
2	120	0,811	120	0,954	120	0,544	120	0,954	120	1,000	256	0,954
3	256	0,954	256	0,811	256	0,954	256	0,954	256	0,811	256	1,000
4	64	0,811	64	0,954	64	0,954	64	0,954	64	0,544	120	0,811
5	256	0,954	256	0,954	256	0,954	256	0,954	256	0,954	256	1,000
6	64	0,954	64	0,954	64	0,544	64	0,544	64	0,544	64	1,000
7	64	0,811	64	0,954	64	0,544	64	0,954	64	0,811	64	0,954
8	120	0,811	120	0,954	256	0,954	256	0,954	256	1,000	256	0,811
9	64	0,954	64	1,000	64	1,000	64	1,000	64	0,811	64	1,000
10	120	0,954	120	1,000	120	0,811	120	1,000	120	1,000	256	0,954
11	64	0,954	256	0,954	256	0,954	256	0,954	256	0,954	256	0,954
12	256	0,954	256	0,954	256	0,954	256	0,954	256	0,811	256	0,954
13	256	1,000	256	0,954	256	0,544	256	0,954	256	0,811	256	0,954
14	256	1,000	256	0,954	256	0,954	256	0,954	256	0,811	256	0,811
15	256	1,000	256	1,000	256	1,000	256	1,000	256	0,811	256	0,954
16	256	0,954	256	0,811	256	0,811	256	1,000	256	0,811	256	0,954
17	64	0,811	120	0,954	120	0,954	120	0,954	120	0,000	120	0,954
18	56	0,544	256	0,954	256	0,954	256	0,954	256	0,954	256	0,954
19	256	1,000	256	0,954	256	0,954	256	0,954	256	0,811	256	0,954
20	256	0,954	256	1,000	256	0,811	256	0,811	256	1,000	256	0,544

Table 2. Signature entropy, *sign2* function

	n=2		*n=3*		*n=4*		*n=5*		*n=6*		*n=7*	
Test N.	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy
1	120	0,811	120	0,811	120	0,811	120	0,811	256	0,811	256	0,811
2	120	0,811	120	0,811	120	0,811	120	0,811	120	0,811	256	0,811
3	256	0,954	256	1,000	256	0,954	256	0,954	256	0,954	256	0,954
4	64	1,000	64	1,000	64	1,000	64	1,000	64	0,954	120	1,000
5	256	0,954	256	0,811	256	0,811	256	0,811	256	0,544	256	0,544
6	64	0,954	64	0,954	64	0,954	64	0,954	64	0,811	64	0,954
7	64	0,954	64	0,954	64	0,954	64	0,954	64	0,811	64	0,811
8	120	0,811	120	0,811	256	0,811	256	0,811	256	0,811	256	0,811
9	64	0,811	64	0,811	64	0,811	64	0,811	64	0,811	64	0,811
10	120	0,811	120	0,954	120	0,954	120	0,954	120	0,954	256	0,954
11	64	0,954	256	0,811	256	0,811	256	0,811	256	0,811	256	0,811
12	256	0,954	256	0,954	256	0,811	256	0,811	256	0,811	256	0,811
13	256	0,954	256	0,954	256	0,954	256	0,811	256	0,544	256	0,544
14	256	0,811	256	0,954	256	0,954	256	0,954	256	0,954	256	0,954
15	256	0,811	256	0,811	256	0,811	256	0,811	256	0,811	256	0,544
16	256	0,954	256	0,811	256	0,811	256	0,811	256	0,811	256	0,811
17	64	0,954	120	0,811	120	0,811	120	0,811	120	0,544	120	0,811
18	56	0,954	256	0,954	256	0,811	256	0,811	256	0,811	256	0,811
19	256	0,811	256	0,811	256	0,811	256	0,544	256	0,544	256	0,544
20	256	0,954	256	1,000	256	1,000	256	1,000	256	1,000	256	0,954

Table 3. Signature entropy, *sign3* function

	n=2		*n=3*		*n=3*		*n=5*		*n=6*		*n=7*	
Test N.	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy
1	120	1,000	120	0,811	120	1,000	120	0,811	256	0,954	256	1,000
2	120	0,811	120	1,000	120	0,811	120	1,000	120	0,954	256	0,811
3	256	0,954	256	0,954	256	0,811	256	1,000	256	0,954	256	0,954
4	64	0,811	64	1,000	64	0,811	64	1,000	64	0,811	120	0,954
5	256	0,954	256	0,811	256	1,000	256	0,811	256	0,811	256	0,544
6	64	0,954	64	1,000	64	0,811	64	0,954	64	0,954	64	0,954
7	64	0,811	64	1,000	64	0,811	64	1,000	64	0,954	64	0,954
8	120	0,811	120	0,811	256	0,811	256	0,954	256	0,811	256	1,000
9	64	0,954	64	0,954	64	0,954	64	0,954	64	0,954	64	0,954
10	120	0,954	120	0,954	120	0,954	120	1,000	120	1,000	256	0,954
11	64	0,954	256	0,811	256	1,000	256	0,811	256	1,000	256	0,954
12	256	0,954	256	0,954	256	0,954	256	0,954	256	0,811	256	1,000
13	256	1,000	256	0,954	256	0,544	256	0,954	256	0,811	256	0,954
14	256	1,000	256	1,000	256	0,811	256	1,000	256	0,544	256	1,000
15	256	1,000	256	0,811	256	1,000	256	0,811	256	0,811	256	0,544
16	256	0,954	256	0,811	256	0,811	256	1,000	256	0,811	256	0,811
17	64	0,811	120	0,811	120	0,811	120	1,000	120	0,544	120	0,954
18	56	0,544	256	0,954	256	0,954	256	0,954	256	0,954	256	0,811
19	256	1,000	256	0,811	256	1,000	256	0,544	256	1,000	256	0,954
20	256	0,954	256	1,000	256	0,811	256	0,954	256	0,954	256	0,954

Table 4. Signature entropy, *sign4* function

	n=2		*n=3*		*n=4*		*n=5*		*n=6*		*n=7*	
Test N.	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy
1	120	1,000	120	0,954	120	0,811	120	0,954	256	0,954	256	0,954
2	120	1,000	120	1,000	120	0,954	120	1,000	120	1,000	256	0,811
3	256	1,000	256	0,954	256	0,954	256	1,000	256	1,000	256	1,000
4	64	0,954	64	1,000	64	0,954	64	1,000	64	0,811	120	0,811
5	256	1,000	256	0,811	256	0,954	256	0,811	256	0,811	256	0,811
6	64	0,954	64	0,954	64	0,811	64	0,954	64	0,954	64	0,954
7	64	1,000	64	0,954	64	0,954	64	1,000	64	1,000	64	0,954
8	120	0,954	120	0,954	256	1,000	256	0,954	256	0,954	256	0,954
9	64	1,000	64	0,954	64	0,811	64	0,954	64	0,954	64	0,954
10	120	0,811	120	0,954	120	0,954	120	1,000	120	1,000	256	1,000
11	64	1,000	256	0,811	256	0,954	256	0,811	256	1,000	256	1,000
12	256	1,000	256	1,000	256	0,954	256	0,811	256	1,000	256	1,000
13	256	1,000	256	0,954	256	0,811	256	1,000	256	1,000	256	0,811
14	256	1,000	256	0,954	256	1,000	256	0,954	256	0,954	256	0,954
15	256	1,000	256	1,000	256	0,954	256	1,000	256	0,811	256	0,811
16	256	1,000	256	0,811	256	0,954	256	1,000	256	0,811	256	1,000
17	64	1,000	120	0,811	120	0,954	120	1,000	120	0,811	120	1,000
18	56	0,954	256	0,954	256	1,000	256	0,954	256	0,954	256	0,954
19	256	1,000	256	0,954	256	0,811	256	0,954	256	0,544	256	0,954
20	256	0,954	256	1,000	256	0,811	256	0,954	256	0,954	256	0,954

The summarized results of the signature entropy dependence on each function used are presented in table 5. As one can see in table 5, the best quality of an embedded system signature generating function is sign4: the maximum values of entropy (1,000), the least standard deviation (0,078) and the lowest limit of prediction interval (0,861). The quality of signatures generated by function sign3 is also relatively high; standard deviation is 0,117 and in prediction interval the lower limit is 0,781. Thus, the function of signature generation has to be constructed by using operations XOR and OR.

Table 5. Signature entropy depend on the function

Function	Average	Standard deviation	Prediction interval	
			min	max
sign1	0,894	0,150	0,744	1,000
sign2	0,852	0,115	0,737	0,967
sign3	0,898	0,117	0,781	1,000
sign4	0,939	0,078	0,861	1,000

Secret encryption keys are generated from an embedded system signature using *Key Derivation Function*. These functions use hash functions, such as MD5, SHA, SHA-2 etc. Further, we investigated the influence of the hash function algorithm on the value of entropy. Since an embedded system signature formatted using *sign4* function has the best entropy, we investigated the key generated by this function. The entropy of keys formatted from 2 to 7 component signatures using *sign4* function and MD, SHA and SHA-2 hash functions is presented in table 6.

Table 6. Keys entropy, used hash functions and *sign4* function

	n=2			*n=3*			*n=4*			*n=5*			*n=6*			*n=7*		
Test N.	MD5	SHA	SHA2	MD5	SHA	SHA2	MD5	SHA	SHA2	MD5	SHA	SHA2	MD5	SHA	SHA2	MD5	SHA	SHA2
1	1,000	1,000	1,000	1,000	1,000	1,000	0,978	0,997	0,997	0,999	0,991	0,998	0,989	0,998	0,999	1,000	0,991	1,000
2	0,991	0,999	0,995	0,961	0,995	0,996	0,993	0,996	0,999	0,997	0,989	0,999	0,989	0,993	0,998	0,997	0,991	0,996
3	0,993	1,000	0,975	0,990	0,997	0,998	0,999	1,000	1,000	0,993	0,996	0,999	0,999	1,000	1,000	1,000	0,997	1,000
4	0,989	0,993	0,998	0,987	0,996	0,993	0,987	0,990	1,000	0,990	1,000	0,996	1,000	1,000	0,999	0,996	0,998	0,999
5	0,999	0,999	0,997	1,000	1,000	0,997	0,998	0,996	0,990	0,985	1,000	0,999	0,989	0,997	0,997	0,982	1,000	1,000
6	1,000	0,996	1,000	0,965	0,998	1,000	0,998	0,989	1,000	0,996	0,974	0,999	1,000	0,994	0,998	1,000	1,000	0,999
7	0,998	0,998	0,999	0,998	0,996	1,000	0,986	1,000	0,996	0,972	0,999	0,994	0,993	0,990	0,999	1,000	0,971	0,998
8	0,998	0,996	1,000	0,999	0,997	0,999	1,000	0,999	0,999	0,992	0,991	0,997	0,999	0,999	1,000	1,000	0,999	1,000
9	0,989	0,991	0,998	0,998	0,993	0,997	0,998	1,000	0,999	0,992	1,000	0,998	0,989	0,996	1,000	1,000	0,999	0,999
10	0,999	0,999	0,998	0,986	0,991	0,999	1,000	1,000	0,997	0,994	1,000	1,000	0,986	0,998	0,999	0,986	0,998	0,999
11	1,000	0,994	0,999	0,996	0,998	1,000	0,974	0,996	0,999	1,000	0,999	1,000	0,996	0,997	1,000	0,998	0,996	0,997
12	0,999	0,999	0,997	0,986	0,989	0,999	0,992	1,000	0,990	1,000	0,995	1,000	1,000	1,000	0,993	0,996	1,000	0,999
13	1,000	1,000	1,000	0,997	0,987	1,000	0,999	0,995	0,998	0,986	0,994	0,999	1,000	0,998	0,998	0,999	0,991	0,999
14	1,000	1,000	1,000	0,998	1,000	0,994	0,994	0,993	0,996	1,000	1,000	0,999	0,987	1,000	1,000	1,000	1,000	0,998
15	0,978	1,000	1,000	1,000	1,000	0,999	1,000	0,998	1,000	0,996	0,999	0,995	0,999	1,000	1,000	0,998	0,995	1,000
16	0,999	0,999	0,997	0,994	1,000	0,995	0,998	0,996	0,999	0,989	0,991	0,996	1,000	0,999	0,999	1,000	0,999	0,999
17	0,998	0,998	0,999	0,954	0,994	0,993	0,997	0,976	1,000	0,999	0,999	1,000	1,000	0,999	0,994	0,998	0,997	0,978
18	0,997	0,991	1,000	0,999	1,000	0,998	0,967	1,000	1,000	0,976	0,999	0,999	0,995	1,000	0,999	0,997	0,999	0,999
19	0,978	1,000	1,000	0,982	0,993	1,000	0,999	0,999	1,000	0,997	0,986	0,997	1,000	0,997	0,997	0,986	1,000	0,993
20	1,000	0,998	0,998	1,000	0,996	0,999	1,000	0,997	0,999	0,996	0,999	1,000	1,000	1,000	0,997	0,985	0,991	0,996

Entropy estimates, average, standard deviation and prediction interval depending on the hash function are presented in table 7.

Table 7. Secret keys entropy dependence on function

Function	Average	Standard deviation	Prediction interval	
MD5	0,994	0,008	0,985	1,000
SHA	0,995	0,007	0,988	1,000
SHA-2	0,998	0,003	0,994	1,000

All hash functions generate high-entropy cryptographic keys. However, the least standard deviation (0,003) and the lowest limit of prediction interval (0,994) contain keys generated using function SHA-2.

To summarize, it can be stated that the entropy of secret encryption key generation by using signature formation functions based on OR and XOR operations and SHA-2 hash function is high.

Conclusions

In this paper, we present the method for secret encryption key generation using an embedded system signature and evaluate the entropy of keys and the efficiency of hash functions.

The proposed method effectively generates high entropy (value close to 1) keys without any additional hardware and infrastructure cost, which is vital for embedded systems with limited resources.

The secret encryption keys with highest entropy are generated using the signature generation function developed from XOR and OR operations and using the hash function SHA-2.

In future, we will develop a prototype tool of embedded system software protection using the proposed secret key generation method and investigate the possibilities of its use.

References

1. **Arora D., Ravi S., Raghunathan A., Jha N.J.** Secure Embedded Processing through Hardware-assisted Run-time Monitoring. *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'05), IEEE Computer Society*. Washington, DC, 1. 2005, 178-183.
2. **Babar S., Stango A., Prasad N., Sen J., Prasad, R.** Proposed embedded security framework for Internet of Things (IoT). *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronics Systems Technology (Wireless VITAE), 2011 2nd International Conference on, vol., no., pp.1-5, Feb. 28 2011-March 3* 2011.
3. **Bedrune J. B., Éric F., Frédéric R.** Cryptography: all-out attacks or how to attack cryptography without intensive cryptanalysis. *Journal in Computer Virology*, Springer Paris, Vol. 6, Issue 3. 2010, 207–237.
4. BSA. Seventh Annual BSA and IDC Global Software Piracy Study. May 2010. 18 p.
5. **Carrera E.** Scanning data for entropy anomalies. 2007. http://blog.dkbza.org/2007/05/scanning-data-for-entropy-anomalies.html.
6. **Collberg C., Thomborson C., Low D.** A taxonomy of obfuscating transformations. *Technical Report 148, Department of Computer Sciences, the University of Auckland, July 1997*, 36 p.
7. **Gagnon M. N., Taylor S., Ghosh A. K.** Software Protection through Anti-Debugging. *IEEE Security and Privacy, v.5 n.3, May* 2007, 82-84.
8. **Gelbart O., Ott P., Narahari B., Simha R., Choudhary A., Zambreno J.** CODESSEAL: Compiler/FPGA Approach to Secure Applications. *Proceedings of IEEE International Conference on Intelligence and Security Informatics*. Springer-Verlag, Heidelberg, Germany. 2005, 530-536.
9. **Henke Ch., Schmoll C., Zseby T.** Empirical evaluation of hash functions for multipoint measurements. *SIGCOMM Comput. Commun. Rev.* 38, 3 (July 2008), 39-50.
10. International Organization for Standardization. ISO/IEC FCD 18033–2, IT Security techniques — Encryption Algorithms — Part 2: Asymmetric Ciphers, 2004.
11. **Joux A., Peyrin T.** Hash functions and the (amplified) boomerang attack. *In Proceedings of the 27th annual international cryptology conference on Advances in cryptology (CRYPTO'07), Alfred Menezes (Ed.). Springer-Verlag, Berlin, Heidelberg*, 244-263.
12. **Jozwiak I. J., Liber A., Marczak K.** A Hardware-Based Software Protection Systems-Analysis of Security Dongles With Memory. *Proc. the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07), Washington: IEEE Computer Society*, 2007, 28-38.
13. **Kanzaki Y., Monden A., Nakamura M., Matsumoto K.** Exploiting self-modification mechanism for program protection. *Proc. the 27th Annual International Computer Software and Applications Conference, Washington: IEEE Computer Society*, 2003, 170-179.
14. **Kazanavicius E., Paskevicius R., Venckauskas A., Kazanavicius V.** Securing Web Application by Embedded Firewall. *Electronics and Electrical Engineering. – Kaunas: Technologija*, 2012. – No. 3(119).
15. **Kent A.D., Liebrock L.M.** Secure Communication via Shared Knowledge and a Salted Hash in Ad-Hoc Environments. *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, 18-22 July 2011, 122-127.
16. **Kocher P., Lee R., McGraw G., Raghunathan A.** Security as a new dimension in embedded system design. *In Proceedings of the 41st annual Design Automation Conference (DAC '04). ACM, New York, NY, USA*, 753-760.
17. **Liutkevicius A., Vrubliauskas A., Kazanavicius E.** Assessment of Dongle-based Software Copy Protection Combined with Additional Protection Methods. *Electronics and Electrical Engineering. – Kaunas: Technologija*, 2011. – No. 6(112), 111–116.
18. **Main A., van Oorschot P.C.** Software protection and application security: Understanding the battleground. *International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography, Heverlee, Belgium, June* 2003. 19.
19. **McGraw G.** Software security. *Security & Privacy, IEEE , vol.2, no.2*, 2004, 80- 83.
20. **MeiHong L., JiQiang L.** USB Key-Based Approach for Software Protection. *International Conference on Industrial Mechatronics and Automation*, 2010, 151-153.
21. **Mumtaz S., Iqbal S., Hameed I.** Development of a Methodology for Piracy Protection of Software Installations. *9th International Multitopic Conference, IEEE INMIC 2005, 24-25 Dec.* 2005, 1-7.
22. **Navidi W.** Statistics for engineers and scientists. New York: McGraw-Hill, 2011, 908 p.
23. NIST. National Vulnerability Database Version 2.2, *http://nvd.nist.gov/home.cfm.*
24. **Papoutsis E., Howells G., Hopkins A., McDonald-Maier K.** Key Generation for Secure Inter-satellite Communication. *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on 5-8 Aug.* 2007, 671-681.
25. PC GUARD. Professional software protection and licensing system. *http://www.sofpro.com.*
26. **Ravi S., Raghunathan A., Kocher P., Hattangady S.** Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst. 3, 3, August 2004*, 461-491.
27. **Suh G. E., Devadas S.** Physical unclonable functions for device authentication and secret key generation. *Proceedings of the 44th annual Design Automation Conference, June 04-08, 2007, San Diego, California.*
28. **Tilborg H. C.** (Ed). Encyclopedia of Cryptography and Security. Springer, 2005, 697 p.
29. **Wang X Y, Yu H B.** How to break MD5 and other hash functions. *Advances in Cryptology, EUROCRYPT 2005, LNCS 3494. Berlin: Springer-Verlag*, 2005: 19–35.