Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a Informatiky
Katedra Informatiky

Ochrana spustiteľného kódu

Ochrana spustiteľného kódu pomocou samoopravných kódov

(diplomová práca)

Michal Šikerle

Školiteľ: Mgr. Miroslav Demeter

Bratislava, 2007

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry.

V Bratislave, 6.Mája 2007 .....

Michal Šikerle

Ďakujem môjmu diplomovému vedúcemu Mgr. Miroslavovi Demeterovi a tiež konzultantovi Mgr. Radovanovi Mikušovi za cenné rady a pripomienky pri vypracovávaní tejto diplomovej práce. Chcel by som tiež poďakovať mojej rodine za podporu počas môjho štúdia.

Obsah

1 Úvod **1**

1.1 Úvod 1
1.2 Cieľ práce 2
1.3 Členenie práce 2

2 Prehľad ochrany softvéru **3**

2.1 Code obfucation 4
2.2 Software Watermarking 5
2.3 Software diversity 6
2.4 Software Tamperproofing 6
2.4.1 Remote Verification 6
2.4.2 Hardware assisted tamper resistance 7
2.4.3 Encryption 7
2.4.4 Multi-blocking encryption 7
2.4.5 Self-checking tamper resistance 7

3 Ochrana spustiteľného kódu **9**

3.1 Hešovanie 9
3.2 Ochrana hešovacieho algoritmu 10
3.2.1 IVK 10
3.2.2 Sieť hešovacích funkcií 11
3.2.3 Ďalšie metódy 12

4 Útoky na samokontrolné mechanizmy **13**

4.1 Základné útoky 14
4.1.1 Základné predpoklady odolnosti 15
4.2 Duplikácia pamätových stránok 16

5 Úvod do problematiky samoopravných kódov **18**
5.1 Základné pojmy . . . 18
5.1.1 Jednoduché kódy opravujúce/odhaľujúce chyby . . . 19
5.2 Lineárne kódy . . . 20
5.2.1 Hammingov kód . . . 20
5.3 Ďalšie samoopravné kódy . . . 21
5.3.1 Cyklické kódy . . . 22
5.3.2 BCH kódy . . . 22
5.4 Konvolučné kódy . . . 23
5.5 Modifikácie samoopravných kódov . . . 23
5.5.1 Predlžovanie a skracovanie . . . 23
5.5.2 Zväčšovanie a zmenšovanie . . . 24
5.5.3 Rozšírenie a zúženie . . . 24

6 Náš Návrh **25**
6.1 Hešovacie stromy . . . 25
6.1.1 Hešovací zoznam . . . 26
6.1.2 Hešovací strom . . . 26
6.2 Návrh mechanizmu . . . 26
6.3 Základné jednotky mechanizmu . . . 27
6.3.1 Spôsoby vkladania . . . 27
6.3.2 Popis Jednotiek . . . 28
6.4 Vkladanie Jednotiek do programu . . . 29
6.4.1 Popis základného algoritmu vkladania . . . 30
6.5 Vylepšené algoritmy vkladania . . . 34
6.6 Beh mechanizmu . . . 35

7 Riešenia narušenia integrity **37**
7.1 Klasifikácia prístupov . . . 37
7.1.1 Logovanie . . . 37
7.1.2 Samodeštrukcia . . . 38
7.1.3 Opravné mechanizmy . . . 38
7.2 Samoopravné mechanizmy . . . 39
7.3 Reakcia nášho mechanizmu na narušenie integrity . . . 42

8 Metriky, odolnosť mechanizmu **44**
8.1 Metriky 44
8.1.1 Nárast veľkosti programu 45
8.1.2 Spomalenie behu programu 45
8.1.3 Opravná schopnosť 46
8.1.4 Odolnosť pred odstránením 46

9 Záver **48**

Kapitola 1

Úvod

1.1 Úvod

Každý rok čelí softvérový priemysel na celom svete miliardovým stratám v dôsledku softvérového pirátstva. Od chvíle, ako sa počítače stali populárnymi a začali sa masovo šíriť medzi ľudí, problém nelegálneho šírenia softvéru začal naberať na intenzite. Veľkú zásluhu na rozkvete softvérového pirátstva mal taktiež obrovský nárast celosvetovej siete Internet, neustále sa zväčšujúci počet pripojených počítačov a obrovský nárast komunikácie. No až technológie rýchleho a pritom cenovo dostupného pripojenia priniesli so sebou zrejme najväčší problém v podobe peer to peer sietí. Vďaka týmto sieťam je dnes možné dostať sa k nelegálnym kópiam ľubovoľného softvéru priamo z pohodlia vlastného domova a v prakticky zanedbateľnom čase. Vývojári softvéru vkladajú nemalé prostriedky na zabezpečenie ochrany pre svoje produkty, no napriek tejto snahe sa zdá, že vynaliezavosť ľudí, ktorí tieto ochrany lámu, je ešte väčšia. Bohužiaľ, dôsledkom softvérového pirátstva, respektíve ním spôsobených ušlých ziskov, sú vyššie ceny softvéru pre legálnych zákazníkov. Existujú dva hlavné názorové smery ako sa s týmto problémom vyrovnať. Ten prvý je založený na zákonných úpravách a postihovaní pirátstva. Čoraz tvrdšie tresty pre nelegálnych užívateľov softvéru sú síce realitou, no postihujú len firmy a organizácie. Zrejme nie je v silách žiadneho úradu bojovať so šírením nelegálneho softvéru medzi súkromnými osobami. Práve tu prichádza druhý názorový smer v riešení pirátstva čoraz sofistikovanejšími ochrannými mechanizmami, ktoré vývojári vo svojich produktoch implementujú. Úlohou týchto mechanizmov je ochrana programov pred spätným inžinierstvom ako aj manipuláciami s programom,

ktoré sú nevyhnutné pre tzv. cracknutie softvéru. Ak by tieto mechanizmy fungovali spoľahlivo, znamenalo by to koniec softvérového pirátstva. Bohužiaľ, realita je iná a ochrana ľubovoľného generického produktu je veľmi rýchlo po jeho vydaní prelomená, čo znamená okamžitú distribúciu jeho nelegálnych kópií do sveta. Je zrejme nemožné zabezpečiť neprelomiteľnú ochranu pre ľubovoľný softvér. Cieľom vývoja v tejto oblasti je urobiť ochranný mechanizmus tak odolný, aby pokusy o jeho odstránenie boli z finančného a časového hľadiska nerentabilné. Ak totiž vyjde produkt a jeho pirátska verzia sa objaví až po dostatočne dlhom čase, množstvo ľudí sa zrejme radšej rozhodne si daný produkt zakúpiť ako ďalej čakať.

Veľký problém vo výskume ochrany softvéru spočíva v nedostupnosti informácií o ochrane implementovanej v súčasných programoch. To je samozrejme aj pochopiteľné, keďže zverejnenie informácií o mechanizme, ktorý chráni produkt pred manipuláciou by malo za následok jeho ľahké odstránenie. Výskum sa preto odohráva na akademickej scéne, predovšetkým vo forme rôznych návrhov a článkov, ktoré sa venujú tejto problematike.

1.2 Cieľ práce

Cieľom tejto práce je ďalší návrh ochranného mechanizmu. Ako ďalej ukážeme, je to mechanizmus spadajúci do kategórie ochrany integrity programu, ktorý je zabudovaný priamo do programu. Zvláštnosťou tohto návrhu je jednak použitie istej špeciálnej dátovej štruktúry a najmä aplikácia samoopravných kódov do ochrany softvéru, ktorá sa zatiaľ v žiadnom návrhu neobjavila.

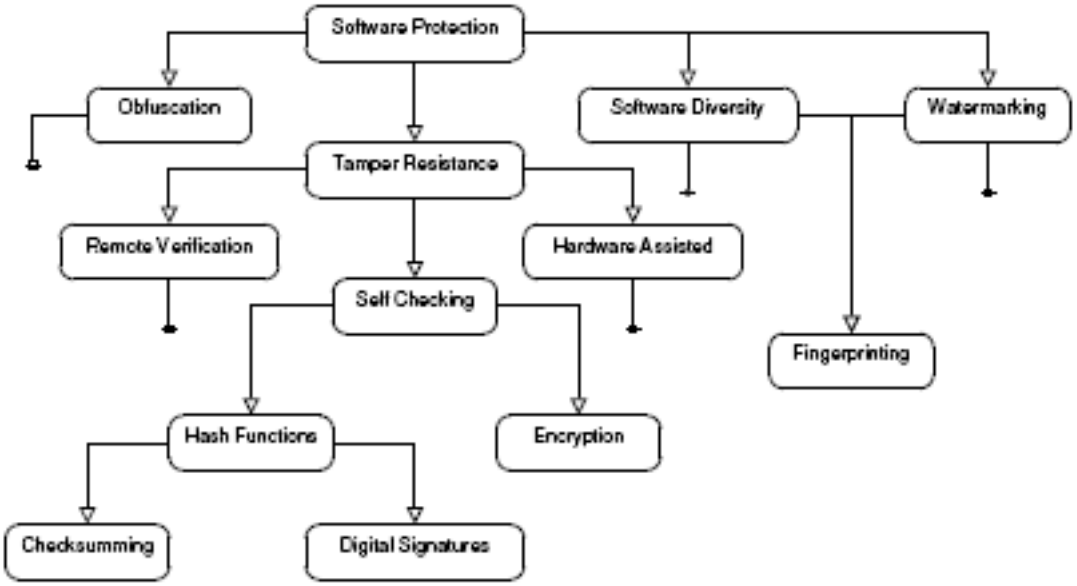
1.3 Členenie práce

Práca je rozdelená na niekoľko kapitol. Kapitola 2 sa venuje základnému prehľadu v ochrane softvéru. V kapitole 3 hovoríme o ochrane spustiteľného kódu, špeciálne o jeho ochrane pomocou samokontrolných mechanizmov. Kapitola 4 sa zameriava na útoky proti týmto mechanizmom. V kapitole 5 sú základy teórie samoopravných kódov, ktoré využívame v našom návrhu. Kapitoly 6 a 7 rozoberajú samotný návrh a kapitola 8 je venovaná metrikám skúmaným na našom návrhu.

Kapitola 2

Prehľad ochrany softvéru

Pre pochopenie ľubovoľného problému potrebujeme najskôr poznať jeho pozadie. Preto sa na úvod práce pozrieme na všeobecný prehľad v ochrane softvéru. Ako ukazuje obrázok nižšie, táto disciplína má niekoľko základných kategórií, ktoré si v tejto kapitole zbežne predstavíme.



Obrázok 2.1: Klasifikácia ochrany softvéru

2.1 Code obfucation

Primárny cieľ zahmlievania kódu (voľný preklad code obfuscation) je ochrana kódu proti spätnému inžinierstvu (revers enginering), teda získaniu zdrojového kódu programu zo skompilovaného. Táto ochrana je dôležitá pre ochránenie intelektuálneho vlastníctva vývojárov daného programu, teda kľúčových algoritmov a postupov. Vo všeobecnosti je to teda ochrana pred nelegálnym používaním softvéru. Zahmlievanie kódu je však nutnou podmienkou dobrého fungovania akejkoľvek ochrany integrity programu. Podstatou zahmlievania kódu je transformácia kódu na funkčné ekvivalentný, ale pre útočníka po dekompilácii ťažšie zrozumiteľný tvar. Útočníkovi, ktorý sa snaží z programu vyextrahovať pre neho užitočné informácie sa snaží zabrániť pochopeniu fungovania a kontroly riadenia v programe. Vo všeobecnosti sa nedá zaručiť, že sa útočníkovi napriek našej snahe nepodarí zistiť funkcionalitu programu. Zahmlievanie sa však snaží o to, aby to bolo z finančného hľadiska neefektívne.

Zahmlievanie kódu môžeme charakterizovať niekoľkými metrikami.

- Účinnosť

  Účinnosť charakterizuje úroveň zmätenia pre útočníka. Účinná metóda proti ľudskému útočníkovi už ale nemusí byť tak účinná proti automatickým nástrojom.

- Odolnosť

  Odolnosť zahmlievania je schopnosť odolať automatickým nástrojom.

- Cena

  Čiže nakoľko sa program po zahmlievaní spomalí, prípadne o koľko viac zdrojov spotrebuje.

- Neviditeľnosť

  Táto metrika hovorí o tom, či je hneď jasné, že kód bol zahmlievaný.

Techniky zahmlievania môžeme rozdeliť do niekoľkých kategórií.

- Lexikálne transformácie

  Zahŕňa to napríklad premenovanie premenných, tak aby nebolo jasné na čo slúžia. Odstránenie komentárov a formátovania(to už robí kompilátor) atď.

- Transformácie toku riadenia

  Transformácie behu programu, teda vykonávanie inštrukcií za behu programu. Tento typ transformácií sa snaží o zahmlenie behu programu napríklad vsunutím falošného vetvenia programu alebo nefunkčného kódu, pomocou predikátov, ktoré sú vždy true alebo vždy false.

- Dátove transformácie

  Transformácie statických a dynamických dátových štruktúr v programe. Existuje množstvo rôznych prístupov, napríklad rozdelenie tried alebo premenných, preusporiadanie dát, zašifrovanie reťazcov atď.

- Preventívne transformácie

  Špeciálny typ transformácií, ktorý sa zameriava na nedostatky súčasných nástrojov používanych pri odzahmlievaní. Keď poznáme metódy, akými tieto nástroje analyzujú kód, môžeme to využiť proti nim.

Na záver už len spomenieme, že ďalšie informácie o zahmlievaní kódu je možné nájsť napríklad v[14].

2.2 Software Watermarking

Watermarking vo všeobecnosti, je informácia vložená do nejakého objektu. Software Watermarking(softvérový vodoznak) je informácia zabudovaná do spustiteľného kódu. Najčastejšie je vkladaná informácia o autorovi programu, prípadne o vlastníkovi licencie. Úlohou watermarkingu je teda dokázať vlastníctvo programu. Tento mechanizmus musí byť dostatočne odolný proti odstráneniu automatickými nástrojmi ako aj proti úmyselnému pokusu o jeho odobratie z programu. Ďalšie informácie možno nájsť v[26, 27]. Uvedieme aspoň základné rozdelovanie watermarkov.

Statické metódy

Takýto watermark sa dá rozpoznať bez spustenia programu. Informácia môže byť uložená buď v statických dátach programu alebo v samotnom kóde(graf vetvenia, frekvencia použitia inštrukcií).

Dynamické metódy

Watermark nie je v samotných dátach, nedá sa rozoznať bez spustenia. Pre jeho rozpoznanie sa program musí spustiť z nejakým špeciálnym vstupom. Ester-egg je typickým príkladom takéhoto watermarku.

2.3 Software diversity

Všetky kópie jednej aplikácie sú zvyčajne absolútne identické. Akonáhle sa podarí zlomiť ochranu jednej, ten istý postup sa použije na všetky ostatné. Software diversity (voľne preložené ako rôznorodosť softvéru) znamená odlišnosť jednotlivých inštancií produktu. V prípade, že každá kópia programu by bola čo i len trochu odlišná od ostatných, útočníkovi by nestačilo odstrániť ochranu na jednej kópii a na ostatné použiť ten istý postup.

2.4 Software Tamperproofing

Software Tamperpoofing (ochrana spustiteľného kódu) je disciplína, ktorá narozdiel od zahmlievania kódu, ktoré sa snaží zabrániť porozumeniu programu, sa snaží zabrániť manipulácii s programom. V princípe je to podobné metódam na zisťovanie chýb pri prenose údajov (napríklad CRC), s tým rozdielom, že prúd údajov tu predstavuje binárny program. Manipulácia s programom predstavuje akýkoľvek zásah do programu, ktorý nie je v súlade z licenčnými podmienkami. Väčšina týchto mechanizmov sa implementuje do programov ako ochrana pred odstránením mechanizmu, ktorý má chrániť program pred nelegálnym kopírovaním. Samozrejme overovanie integrity sa dá rovnako aplikovať na ľubovoľnú časť programu, ktorú si želá jej vlastník ochrániť pred neželanými zmenami. Problémom týchto mechanizmov je, že oproti zisťovaniu chýb pri prenose údajov, tu musíme zabezpečiť oveľa väčšiu robustnosť, keďže proti nám nestojí šum v prenosovom kanáli, ale ľudský útočník.

2.4.1 Remote Verification

Prvá možnosť ako zabrániť manipulácii s programom, je verifikácia jeho integrity iným programom. Ten sa môže nachádzať na tom istom počítači, prípadne

sa môže verifikácia odohrať cez sieť. Je niekoľko možných spôsobov na vykonanie tejto verifikácie, ale všetky majú veľké nedostatky v podobe komunikácie medzi programami, ktorá môže byť zo siete zachytená a modifikovaná. V prípade, že program ktorý vykonáva verifikáciu je na tom istom počítači, môže byť pozmenený najskôr on tak, aby túto kontrolu nevykonal.

2.4.2 Hardware assisted tamper resistance

Táto metóda predpokladá hardvérovú pomoc pri zabezpečení programov. Je predpoklad, že manipulovať z hardvérom je ďaleko náročnejšie ako so softvérom a zvládnu to len najodhodlanejší útočníci. Je niekoľko možných prístupov cez zabezpečenie pamäte počítača pred manipuláciou pomocou kryptovania, prípadne zavedenie takzvanej Execute Only Memory, t.j. modulu na ktorom je uložený program, je ho možné odtiaľ spustiť, ale nie je možné ho čítať. Návrhy týkajúce sa tejto metódy možno nájsť napríklad tu[6, 7].

2.4.3 Encryption

Ďalšou altrenatívou obrany softvéru pred manipuláciou, je držať celý program zakryptovaný a odkryptovať ho až pred spustením. Útočník potom nie je schopný manipulovať s programom, pokiaľ nenájde kryptovací kľúč. Problém je, že akonáhle sa program spustí, nachádza sa v pamäti v nezakryptovanom stave a je možné ho odtiaľ získať. Taktiež rutina, ktorá vykonáva samotné kryptovanie, môže byť útočníkom v programe odhalená.

2.4.4 Multi-blocking encryption

Táto forma kryptovania rozdelí binárny program na individuálne zakryptované bloky. Počas spustenia programu sa odkryptujú vždy len tie bloky, ktoré sú momentálne potrebné k behu programu. Keď dokončia svoju činnosť, vrátia sa do zakryptovaného stavu. Podrobnejšie je táto technika rozoberaná v [25].

2.4.5 Self-checking tamper resistance

Self-checking tamper resistance, alebo aj samokontrolne overovanie integrity je obranný mechanizmus, ktorý sa nespolieha na vonkajšie prostriedky, ale obranu pred zasahovaním má zabudovanú priamo do tela programu. Tento typ ochrany

softvéru je hlavným cieľom tejto práce, preto sa naň pozrieme podrobnejšie v ďalšej kapitole.

Kapitola 3

Ochrana spustiteľného kódu

Ako sme povedali, táto práca sa zaoberá hlavne samokontrolnými mechanizmami, preto sa teraz pozrieme podrobnejšie na pozadie potrebné k pochopeniu týchto metód ako aj doposiaľ navrhované spôsoby jeho implementácie.

Základom väčšiny samokontrolných mechanizmov je hešovanie kódu. Hešovanie kódu je založené na každému dobre známych hešovacích funkciách.

3.1 Hešovanie

Hešovacia funkcia je definovaná ako funkcia, ktorá mapuje vstup x konečnej dĺžky na výstup $h(x)$ konečnej dĺžky n, a je ľahké určiť $h(x)$ pre dané h a vstup x. Kryptograficky bezpečná hešovacia funkcia navyše musí spĺňať:

- Pre ľubovoľný výstup je nemožné nájsť vstup, ktorý sa hešuje na daný výstup.
- Je nemožné nájsť druhý vstup, ktorý má rovnaký výstup ako iný špecifikovaný vstup.

Hešovacia funkcia počíta heš strojového kódu dynamicky, teda po začatí vykonávania programu. Následne je výstup porovnaný s dopredu známou správnou hodnotou hešu. Ak sa hodnoty nezhodujú, je zrejmé, že s programom sa manipulovalo. Vychádza to z predpokladu, že ak používame kryptograficky bezpečné hešovacie funkcie nie je možné pozmeniť program tak, aby jeho výsledný heš sa zhodoval s originálom.

Je teda zrejmé, že nie je vhodné ako hešovaciu funkciu použiť funkciu, ktorá bola prelomená, teda sú známe kolízie. Takými sú dnes napríklad MD5, SHA-0 a

iné. V súčasnosti často používaná funkcia SHA-1 je ešte stále pomerne bezpečná (je známa kolízia na ktorú je ale potrebných 2^63 výpočtov hešu) a zároveň spĺňa tzv. lavínový efekt, teda zmena jedného bitu na vstupe má za nasledok s veľkou pravdepodobnosťou úplne inú hodnotu hešu.

Toto je akási základná metóda samokontrolných mechanizmov, ktorá má však evidentné nedostatky a je možné ju jednoducho prelomiť. Pozrieme sa teraz na spôsoby zabezpečenia robustnosti hešovacích algoritmov.

3.2 Ochrana hešovacieho algoritmu

Náš predpoklad je, že útočník pokúšajúci sa o manipuláciu, má úplný prístup k binárnemu programu. Keďže hešovací algoritmus aj predpočítaná hodnota sú uložené spolu s algoritmom, je taktiež potrebné zabezpečiť ich ochranu. Hešovcia funkcia síce chráni kód programu, ale samotná funkcia a predpočítaná hodnota chranené nie sú. Bolo navrhnutých niekoľko riešení tohto problému.

3.2.1 IVK

IVK (Integrity Verification Kernel) je mechanizmus navrhovaný Davidom Aucsmitom[1]. IVK definuje ako malé segmenty kódu, ktoré sú odolné proti manipulácii. Činnosti, ktoré IVK vykonáva, sú:

- Overuje integritu kľúčových častí programu.IVK preto obsahuje hešovací algoritmus tak ako aj predpočítanú správnu hodnotu.
- Vykonáva kľúčové časti programu, ako napríklad inicializácia globálnych premenných. Takto je zabezpečené, že IVK sa nedá z programu jednoducho odstrániť.

IVK využíva niekoľko techník, ktoré zabezpečujú, že všetky funkcie vykonávané vo vnútri IVK sú chránené pred manipuláciou. Spomeňme niektoré:

- Prekladanie úloh

  Všetky úlohy vo vnútri IVK sú vykonávané po malých častiach. Ak úlohy A, B, C sa skladajú z podúloh $a_1, a_2, ..b_1, b_2, ...$ IVK ich vykonáva ako $a_1, b_1, c_1, a_2, b_2, ...$ namiesto $a_1, a_2, .., b_1, b_2,$. Takto je zabezpečené, že sa nevykoná jedna funkcia IVK bez toho, aby sa nevykonali ostatné.

- Šifrovanie kódu

  IVK používa šifrovanie, navyše keď je jedna časť kódu odšifrovaná, zabezpečí, aby sa iné opätovne zašifrovali.

- Jedinečné modifikácie

  Každé IVK, dokonca i pre ten istý program, obsahuje jedinečné modifikácie pre jeho inštancie. Takto je zabezpečená odolnosť proti tzv. Class útoku, teda ak zlomím ochranu jednej inštancie programu, môžem ten istý spôsob aplikovať aj na ostatné.

Celé IVK je rozdelené do malých buniek, ktoré obsahujú malú časť nejakej úlohy IVK plus príkaz skoku na ďalšiu bunku. Keď skončí vykonávanie nejakej bunky, ďalšia bunka v poradí je odšifrovaná a následne sa začne vykonávať. Predchádzajúca bunka sa opätovne zašifruje. Popis celého mechanizmu je pomerne obsiahly a v tejto práci preň nie je mieto, v prípade záujmu odporúčam[1].

3.2.2 Sieť hešovacích funkcií

Ďalšie možné riešenie tohto problému spočíva v zakomponovaní siete hešovacích kódov pracujúcich spoločne. Telo hešovacej funkcie tak ako aj predefinovaná hodnota hešu sú taktiež reťazce strojového kódu, môžu byť aj oni súčasťou vstupu inej hešovacej funkcie. Takto vzniká sieť hešovacích funkcií, v ktorých časť kódu alebo hodnota hešu sú vždy vstupom pre inú hešovaciu funkciu. Nedostatkom je, že ak v závislostiach hešovacích funkcií nie sú cykly, tak vždy aspoň jedna hešovacia funkcia je nechránená. Útočník môže nájsť toto miesto a následne znefunkčniť celú sieť. Tento nedostatok sa samozrejme dá odstrániť vytvorením cyklov. No v tomto prípade buď musia byť predefinované hodnoty hešu vynechané zo vstupov hešovacích funkcií alebo sa nepoužijú kryptograficky bezpečné hešovacie funkcie. Hešovacie funkcie, ktoré nie sú kryptograficky bezpečné sa nazývajú Checksumy.

Testers

Táto idea pochádza od B. Horneho[30]. Tento mechanizmus je založený na sieti checksumov, ktoré overujú integritu kľúčovej časti kódu. Porovnávanie so správnou hodnotou je však naviazané na samotný beh programu, napríklad je

súčasťou podmienkového príkazu. Ak sa teda s programom manipulovalo, má to za následok chybné správanie niektorých častí aplikácie.

Guards

Mechanizmus navrhovaný H. Changom a M. Atallahom[2]. Malé programové jednotky, ktoré však okrem vlastného rátania checksumu na časti kódu vykonávajú ďalšie činnosti. Mimoriadne dôležitá vlastnosť je, že ak niektorá z Guradov zistí manipuláciu, nespôsobí okamžité prerušenie v programe, ktoré by bolo ľahko odhaliteľné pomocou dynamickej analýzy programu. Jedna z možností je pozmeniť program, aby síce fungoval, ale neprodukoval platné výsledky. Ďalšia možnosť je, že obsahuje kód, ktorý miesto v programe s ktorým sa manipulovalo prepíše nezmeneným kódom. Toto ale jednak znamená zväčšenie veľkosti programu a navyše, ak útočník odhalí miesto na ktorom sú tieto časti kódu uložené, môže ich zmeniť tam a keď guard zaznamená manipuláciu, prepíše to aj tak pozmeneným kódom.

3.2.3 Ďalšie metódy

Tu spomenuté metódy sú len akýsi výber z potenciálnych návrhov, ktoré sa za posledné obdobie objavili. Môžeme ďalej spomenúť metódy ako Oblivious hashing[3], ktorý je založený na hešovaní hodnôt premenných počas aktuálneho spustenia a množstvo iných, ich popisovanie však taktiež nie je možné v rozsahu tejto práce a preto len odporúčam ďalšiu literatúru, napríklad [28, 19].

Kapitola 4

Útoky na samokontrolné mechanizmy

Aby sme mohli ďalej uvažovať o návrhu nášho samokontrolného mechanizmu, potrebujeme spomenúť, aké typy útokov boli navrhnuté na ich prekonanie.

Na začiatok sa pozrieme na klasifikáciu metodík, ktoré sa používajú pri útokoch.

- Statický alebo dynamický

  Rozdeľuje útoky podľa informácií, ktoré používajú. Statický útok je založený čisto na skúmaní kódu bez jeho spustenia. Dynamický útok naproti tomu vychádza z informácií zistených pri spustení programu a pozorovaní vykonávania pri jednotlivých spusteniach.

- Konzervatívny alebo aproximatívny

  V ideálnom prípade sú informácie, ktoré máme o programe presné. Presné informácie o programe by sme získali, ak by sme dokázali zistiť všetky možné stavy a priebehy vykonávania programu pre všetky možné vstupy. Tiež by sme museli zistiť aj všetky možné interakcie s prostredím v ktorom sa program nachádza. Toto však takmer vo všetkých prípadoch nie je možné. Konzervatívny a aproximatívny prístup rozdeľujú útoky podľa toho, ako narábame s chýbajúcimi infomáciami. Konzervatívny útok je navrhnutý tak, aby fungoval nehľadiac na chýbajúce informácie. Ak útok môže za istých podmienok z chýbajúcich informácií zlyhať, hovoríme o aproximatívnom útoku.

- Úplný alebo závislý na prostredí

  O úplnom útoku hovoríme, ak ochranný mechanizmus bol úplne znefunkčnený a na následné spúšťanie programu už nie sú kladené žiadne ďalšie požiadavky. Útok je závislý od prostredia, ak pre spustenie manipulovaného programu sú potrebné ešte ďalšie zásahy(napríklad do operačného systému ...) pri každom jeho spustení.

4.1 Základné útoky

Útočník má v súčasnosti obrovské množstvo nástrojov, ktoré mu napomáhajú v snahe zlomiť ochranu softvéru, spomenieme najdôležitejšie z nich.

- Hex editory

  Slúžia na editáciu binárnych súborov v hexadecimálnom formáte.

- Disassembler

  Najdôležitejší nástroj útočníkov. Číta prúd bitov a prekladá ho do skupín inštrukcií. Tieto inštrukcie následne prekladá do, pre ľudí zrozumiteľného, jazyka assembléra.

- Debugger

  Debuger je nástroj určený tak, ako pri bežnom programovaní, na krokovanie kódu a sledovanie hodnôt premenných. Debugery určené pre spätné inžinierstvo dokážu krokovať binárny program a za behu meniť hodnoty registrov, pamäte, premenných.

- Systémové monitory

  Tieto nástroje slúžia na sledovanie interakcií súborov s prostredím, teda zápisy do registrov, volania, atď.

- Unpackery

  Binárne súbory alebo ich časti sa často držia v skomprimovanom stave. Predtým ako útočník môže začať svoju prácu potrebuje súbor v kompletne rozbalenom stave. Práve na to slúžia nástroje zvané Unpackery.

Samozrejme existuje oveľa väčšie množstvo nástrojov napomáhajúcich útočníkom. Ich analýza však nie je náplňou tejto práce.

Ďalej sa pozrieme na základné predpoklady, ktoré musí dobrý samokontrolny mechanizmus spĺňať, aby predstavoval pre útočníka vážnejšiu prekážku.

4.1.1 Základné predpoklady odolnosti

- Statická a dynamická samokontrola

  O statickej sammokontróle hovoríme, ak ochranný mechanizmus vykoná kontrolu integrity len raz a to hneď po spustení programu. V prípade, že program využíva len statickú samokontrolu, teda kontrola sa vykoná len raz, hneď po začatí vykonávania, je možné ho napadnúť tak, že ho zmeníme počas behu a následne zmeny vrátime, takže keď sa program znovu spustí, samokontrolny mechanizmus nezaznamená nijakú zmenu. Takémuto útoku hovoríme dočasné zmeny programu. O dynamickej samokontrole hovoríme teda ak kontrola integrity je vykonávaná neustále počas behu programu. Dočasné zmeny sa dajú aplikovať aj v prípade dynamickej samokontroly, ktorý vykonáva kontrolu neustále počas behu programu. Útočník by však musel vedieť presne, kedy sa tento mechanizmus spúšťa a neustále vymienať správnu verziu za manipulovanú, čo by mu výrazne sťažilo situáciu.

- Zahmlievanie kódu

  Základný predpoklad úspešnosti samokontrolných mechanizmov je, že útočník po dekompilácii programu nesmie odhaliť pomocou statickej analýzy dekompilovaného programu všetky časti kódu, ktoré vykonávajú príslušnú ochranu. Toto je možné dosiahnuť jedine ak útočník nedokáže dostatočne porozumieť kódu. To má za úlohu zahmlievanie kódu pred jeho skompilovaním.

- Viacnásobna ochrana

  Keďže má útočník nad programom neobmedzenú moc, môže ho spúšťať a sledovať jeho správanie pomocou rôznych emulátorov. Môže takto porovnávať jeho správanie bez zásahu a po ňom. Ak teda nejaké miesto v programe chránime len raz, útočník môže takto ľahko vystopovať, kde sa

toto miesto nachádza. Ďalším predpokladom dobrého samokontrolného mechanizmu je teda schopnosť chrániť každé miesto v programe viackrát.

- Sigle-point of failure

  Single-point of failure znamená, že celý ochranný mechanizmus može byť odstránený jedným zásahom do programu. Jedná sa väčšinou o jednoduché spôsoby ochrany ako napríklad jedno miesto v kóde, ktoré počíta heš. Ďalšou dôležitou vlastnosťou dobrého samokontrolneho mechanizmu je rozdelenie mechanizmu, teda jeho distribúcia po celom programe.

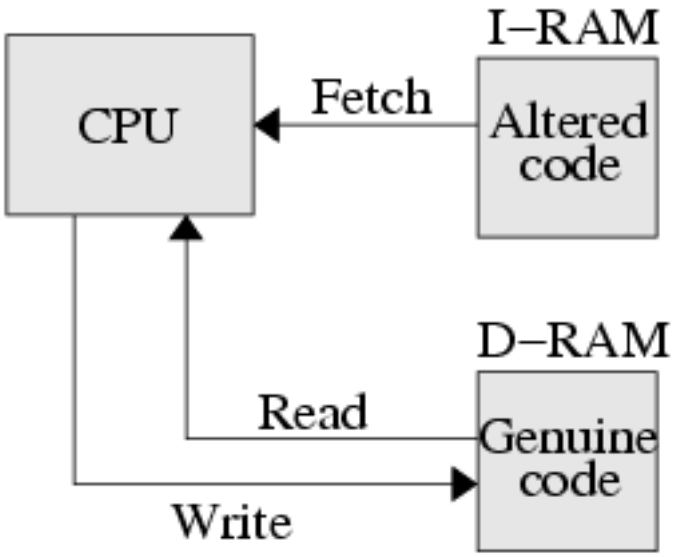
4.2 Duplikácia pamätových stránok

Predstavme si teraz jednu špeciálnu metódu útoku podrobnejšie. Touto metódou je duplikácia stránok v pamäti na ktorých sa nachádza časť kódu, ktorú si želá útočník zmeniť. Táto metóda bola navrhnutá Glenom Wursterom[31]. Súčasné počítače využívajú Von Nuemanovsku architektúru pamäti, kde inštrukcie a dáta pristupujú k tej istej fyzickej pamäti. Existujú prostriedky, ktoré zabezpečia rozdelenie pamäti na dátovú a inštrukčnú časť (takzvaný Harvardský model pamäte), takže hodnoty zapisované do pamäti menia iba dátovú časť a inštrukčnú ponechávajú nezmenenú. Útočník zduplikuje stránky pamäti, ktoré si želá meniť a umiestni ich do dátovej časti aj do inštrukčnej časti. Kópia umiestnená v dátovej časti ostáva nezmenená, takže hešovanie nezistí žiadnu manipuláciu. Ale inštrukcie sa nahrávajú na vykonanie z inštrukčnej časti. Útočník je teda schopný nepozorovane zmeniť stránky pamäte uložené v inštrukčnej časti.

s

Tento útok sa však podarilo úspešne zdolať pomocou samomodifikovateľného kódu. Myšlienka je založená na modifikácii inštrukcie v programe a jej následnom spustení. Proces funguje nasledovne.

1. Prepíše existujúcu inštrukciu I_1 novou inštrukciou I_2. Zmena tejto inštrukcie sa musí dať jednoznačne odpozorovať.
2. Načíta naspäť inštrukciu z pamäte.
3. Vykoná inštrukciu.



Obrázok 4.1: Replikácia pamäte

Ak pamäť je von Neumannovskeho typu, inštrukcia načítaná v kroku dva a inštrukcia vykonaná v kroku tri, bude tá istá inštrukcia I_2. Ak ale pamäť nie je von Neumannovskeho typu, krok dva prečíta I_2, ale krok tri vykoná pôvodnú inštrukciu I_3. Keďže vykonanie inštrukcie I_2 sa dá jednoznačne odpozorovať, zistíme takto jednoznačne typ pamäťovej architektúry na ktorej program beží.

Kapitola 5

Úvod do problematiky samoopravných kódov

Súčasťou obranného mechanizmu navrhovaného v tejto práci je aj schopnosť programu opraviť sa (teda svoje poškodené, respektíve pozmenené časti). Práve za účelom opravy dát boli navrhnuté isté mechanizmy, súborne označované ako samoopravné kódy. Táto kapitola ponúka mimoriadne stručný úvod do tejto problematiky a ukazuje niekoľko základných samoopravných kódov, ktoré sa najčastejšie objavujú v bežnej praxi.

Samoopravné kódy sú mimoriadne dôležitou časťou informačnej teórie. Základnou myšlienkou samoopravných kódov je schopnosť odhaliť, lokalizovať a opraviť chyby vznikajúce pri prenose dát. Rovnaký mechanizmus je však možné použiť aj k lokalizácii a oprave chýb na statických dátach, kedy pre tieto kódy je v princípe jedno či ide o transformácie v čase alebo v priestore. Vo všeobecnosti je podstatou fungovania týchto kódov vkladanie určitej redundantnej informácie do dát, vytvárajúc tak určitú algebraickú, prípadne geometrickú reláciu, ktoré umožňujú naslednú opravu dát. Samoopravné kódy môžeme v zásade rozdeliť na dve veľké triedy a to kódy Blokové a kódy Konvolučné. Všetky kódy spomínané v tejto kapitole pochádzajú z [4, 10, 11].

5.1 Základné pojmy

Jedným zo základných pojmov pre samoopravné kódy je samotný pojem kód. Kód nad abecedou Σ je ľubovoľná množina $C \subset \Sigma^n$ kde n je ľubovoľné a je

označené ako dĺžka kódu. Veľkosť kódu je počet jeho prvkov teda $||C|$. Prvky množiny Σ^n sú slová a prvky množiny C sú potom kódové slová.

Kódom nad dvojprvkovou abecedou sa hovorí binárne. Vo všeobecnosti, kódom nad abecedou veľkosti q sa hovorí $q - arne$ kódy.

Ďalší dôležitý pojem v teórii samoopravných kódov je Hammingova vzdialenosť kódových slov(alebo kódových vektorov). Nech sú $X = (x_1, ..., x_n)$ a $Y = (y_1, ..., y_n)$ prvky Σ^n. Potom ich Hammingova vzdialenosť $d(x, y)$ je počet súradníc, kde sa tieto prvky líšia, teda počet i pre ktoré je $x_i \neq y_i$.

Minimálna vzdialenosť kódu C je definovaná ako

$$\triangle(C) = min_{x,y \in C} d(x, y)$$

Je to teda aj minimálny počet substitúcií, ktoré sú potrebné, aby z jedného kódového slova vzniklo iné kódové slovo. Čím väčšia je minimálna vzdialenosť kódu, tým väčší počet chýb dokáže opravovať.

Celkovo má teda každý samoopravný kód 3 základné vlastnosti a to dĺžka n, veľkosť $||C|$ namiesto ktorej sa používa zvyčajne jej logaritmus so základom $q = |\Sigma|$, teda $k = \log_q |C|$ a minimálna vzdialenosť $d = \triangle(C)$. O kóde s týmito parametrami hovoríme ako o $(n, k, d) - kde$.

Posledný dôležitý pojem je hustota kódu C, čo je podiel $\alpha(C) = k/n$.

5.1.1 Jednoduché kódy opravujúce/odhaľujúce chyby

Najjednoduchším príkladom kódu je Totálny kód nad Σ^n, ktorý je tvorený všetkými slovami dĺžky $n$nad abecedou Σ.

Ďalším jednoduchým príkladom kódu je takzvaný paritný kód. Máme slová dĺžky n. Pridáme ku každému slovu n+1 bit tak, aby počet jednotkových bitov v slove dĺžky n+1 bol párny. Kódové slová budú vyzerať takto

0110101000

1011001101

Doplnený bit sa nazýva paritný bit. Takýto kod dokáže odhaliť chybu nepárnej váhy, keďže potom bude v kódovom slove nepárny počet jednotkových bitov. Nedokáže však určiť, kde táto chyba nastala.

5.2 Lineárne kódy

Veľká množina samoopravných kódov je vytvorená nad nejakou vhodnou algebraickou štruktúrou, teda využíva jej prvky ako kódové slová.

Máme konečné pole $GF(q)$, kde q je mocnina prvočísla (najčastejšie 2). Množina $GF(q)^n$ n-tic nad poľom $GF(q)$ s aditívnou a multiplikatívnou operáciou tvorí vektorový priestor.

Lineárnym kódom nad abecedou $GF(q)$ je ľubovoľný vektorový podpriestor vektorového priestoru $GF(q)^n$. Ak má lineárny kód dimenziu = k potom $|C| = q^k$. Jednou z veľkých výhod lineárneho kódu je úsporný popis. Namiesto q^k prvkov kódu stačí uviesť len k prvkov niektorej jeho bázy. Obvyklým tvarom takéhoto zápisu je *generujcamatica* kódu.

Generujúca matica lineárneho kódu sa dá upraviť do tvaru v ktorom prvých k stĺpcov tvorí jednotkovú maticu. Riadky výslednej matice sú naďalej bázou kódu a pre slovo $c \in F_q^k$ dĺžky k existuje práve jedno slovo z kódu, ktoré má na prvých k súradniciach práve slovo c. O prvých k symboloch potom hovoríme ako o informačných a o zvyšných ako o kontrolných.

Poslednou mimoriadne dôležitou vecou pri lineárnych kódoch je Duálny kód k lineárnemu kódu, čo je jeho ortogonálny doplnok teda

$$C^{\perp} = \{x :< x, y >= 0 \text{ pre vsetky } y \in C\}$$

Generujúca matica tohto duálneho kódu sa nazýva aj matica kontrolná, pretože jej riadky určujú lineárne rovnice, ktoré musí spĺňať každé slovo z kódu. Takže zohráva kľúčovú úlohu pri dekódovaní prijatého slova.

5.2.1 Hammingov kód

Pozrieme sa teraz podrobnejšie na jeden špeciálny lineárny kód, ktorý ako neskôr ukážeme, má pre nás mimoriadne využitie. Hammingov kód pochádza od amerického matematika Richarda Hamminga a bol publikovaný už v roku 1950.

Hammingov kód určený dvojicou parametrov (n, k), kde $n = 2^m - 1$, $m >= 3$, $minN$ a $k = 2^m - 1 - m$, n je dĺžka kódového slova, k je dĺžka informácie a $n - k$ je počet kontrolných bitov.

Pozrieme sa teraz na princíp vytvárania hammingových kódov na Hammingovom kóde $(7, 4)$. Predpokladáme, že sme vytvorili kódové slovo $v = (v_1, ..., v_7)$.

Z jednotlivých komponentov kódového slova vytvoríme tri kontrolné sumy s_0, s_1, s_2, pomocou ktorých sme schopný rozlíšiť 8 rôznych udalostí. Sú to: nenastala žiadna chyba, nastala chyba v prvom komponente, druhom, ..., siedmom.

Nech $\sigma(i, n)$ označuje n-bitový vektor reprezentujúci číslo i. Nech $u = (u_1, ..., u_n)$n a $v = (v_1, ..., v_n)$ su binárne vektory, symbolom $u\&v$ budeme označovat vektor $u\&v = (u_1v_1, ..., u_nv_n)$. Potom pre kontrolné sumy platí: $s_j = \oplus_{\sigma(i,3)\&\sigma(2^j,3)=\sigma(2^j,3)} v_i$, v našom prípade je to $s_0 = v_1 \oplus v_3 \oplus v_5 \oplus v_7$, $s_1 = v_2 \oplus v_3 \oplus v_6 \oplus v_7$, $s_2 = v_4 \oplus v_5 \oplus v_6 \oplus v_7$.

Komponent v_i sa vyskytuje v toľkých kontrolných sumách, ako je Hammingova váha binárneho zápisu $\sigma(i, 3)$. Keďže existujú práve 3 binárne vektory dĺžky tri s Hammingovou vahou 1, reprezentujúce čísla 1, 2, 4, každý s komponentov v_1, v_2, v_3 vystupuje v jednej kontrolnej sume. To znamená, že pri ľubovoľnej voľbe komponentov v_3, v_5, v_6, v_7 kódového slova a vhodnou voľbou komponentov v_1, v_2, v_3 dosiahneme, že kontrolné sumy budú pre kódove slovo nulové. Stačí položiť $v_1 = v_3 \oplus v_5 \oplus v_7$, $v_2 = v_3 \oplus v_6 \oplus v_7$, $v_4 = v_5 \oplus v_6 \oplus v_7$.

Kódovanie správ pomocou Hammingovho $(7, 4)$ kódu prebieha tak, že sa správa najprv rozdelí na bloky dĺžky 4 a tie sa doplnia tromi kontrolnými symbolmi na kódové slovo.

Dekódovanie Hammingovho $(7, 4)$ kódu je nasledovné. Predpokladajme, že pri prenose nastala chyba v i-tom komponente kódového slova. Chyba spôsobí, že všetky kontrolné sumy, ktoré obsahujú komponent v_i nadobudnú hodnotu jedna. To sú práve tie sumy s_j , pre ktoré platí $\sigma(i, 3)\&\sigma(2^j, 3) = \sigma(2^j, 3)$, teda binárny vektor $s = (s_2, s_1, s_0)$ predstavuje číslo $\sigma(i, 4)$. Vektor hodnôt jednotlivých kontrolných súm sa nazýva syndróm chyby. V tomto prípade syndróm chyby predstavuje pozíciu, na ktorej chyba váhy jedna v kódovom slove vznikla. Ak chyba pri prenose nevznikla, hodnota syndrómu chyby je rovná nule.

5.3 Ďalšie samoopravné kódy

Pozrieme sa teraz este stručne na niekoľko ďalších významných samoopravných kódov.

5.3.1 Cyklické kódy

Cyklické kódy sú podtriedou lineárnych kódov, ktoré fungujú na, o niečo silnejšej algebraickej štruktúre. Čiastočne tak odstraňujú nevýhody čisto lineárnych kódov a to najmä výpočtovo veľmi náročné dekódovanie.

Lineárny kód C sa nazýva cyklický, ak pre ľubovoľné kódové slovo $X = (x_0, ..., x_n - 1) \in C$ platí, že $X^{'} = (x_n - 1, x_0, ..., x_n - 2) \in C$

Cyklické kódy sú teda lineárne kódy, ktoré sú uzavreté na cyklický posun kódových slov. Vhodná reprezentácia cyklických kódov je pomocou polynómov z okruhu $GF(q)[x]/x^n - 1$. Každú n-ticu $a = (a_0, a_1, ..., a_n)$ stotožníme z polynómom $\sum_{i=0}^{n-1} a_i x^i \in F_q^n[x]$

Cyklický kód sa dá generovať pomocou polynómu minimálneho stupňa z tohto kódu. Každý cyklický kód obsahuje takýto polynóm. Z tohto polynómu je možné odvodiť jeho generujúcu maticu. Nech stupeň generujúceho polynómu β je k. Potom množina $B = (\beta, x\beta, ..., x^{n-k-1}\beta)$ je bázou nášho kódu.

5.3.2 BCH kódy

BCH kódy sú mimoriadne rozsiahla podmnožina cyklických kódov, ktoré majú niekoľko dobrých vlastností a sú vďaka tomu často používané v praxi. BCH kódy existujú pre veľké množstvo parametrov a existujú pre ne efektívne metódy kódovania a dekódovania.

BCH kód je určený niekoľkými hodnotami. Veľkosťou abecedy q, dĺžkou n a zadanou vzdialenosťou δ. Kód s týmito parametrami označíme $BCH_{q,n,\delta}$. Je to cyklický kód dĺžky $n = q^m - 1$ nad F_q tvorený všetkými polynómami s koreňmi $\alpha, \alpha^2, ..., \alpha^{\delta-1}$ kde α je primitívny prvok telesa F_{g^m}.

Dôležitá vlastnosť BCH kódov je, že minimálna vzdialenosť kódu $BCH_{q,n,\delta}$ je väčšia alebo rovná zadanej vzdialenosti δ.

Reed-Solomonove kódy

Reed-Solomonove kódy je špeciálna podmnožina BCH kódov.

RS kód je primitívny BCH kód dĺžky $n = q - 1$ nad poľom $GF(q)$. RS kód opravujúci t chýb možno zadať polynómom $g(x)$ s koreňmi $\alpha^{j_0+1}, \alpha^{j_0+2}, ..., \alpha^{j_0+2t}$ kde α je primitívny prvok poľa $GF(q)$.

Pre RS kódy platí, že pre zadané n,k neexistuje kód ktorého minimálna vzdialenosť by bola väčšia ako minimálna vzdialenosť RS kódu. Bohužiaľ, RS-

kód neexistuje pre všetky možné parametre n a k takže musíme často hľadať iný dostatočne dobrý kód.

RS kódy sú dnes široko používané napríklad aj na opravu dát uložených na Kompaktných diskoch(CD).

5.4 Konvolučné kódy

Blokové kódy dopĺňajú istú redundantnú informáciu(kontrolné bity) do každého z blokov pevnej dĺžky nezávisle od seba. Konvolučné kódy pracujú s prúdom blokov, obvykle malej dĺžky do ktorých dopĺňajú kontolné bity na základe informačných symbolov daného bloku a niekoľkých predchádzajúcich blokov. Zovšeobecnenie takéhoto kódovania je stromový kód, ktorý je generovaný ako stav-meniaci proces. Podľa vstupných symbolov generátor prechádza zo stavu do stavu a pri každom takomto prechode generuje výstupné symboly.

5.5 Modifikácie samoopravných kódov

Modifikácie samoopravných kódov riešia otázku, čo ak potebujeme kód z určitými parametrami a vieme, že existuje dobrý kód s veľmi podobnými parametrami. Takéto transformácie samoopravných kódov skutočne existujú.

Budeme zvažovať tri základné parametre kódu a to dĺžku n, veľkosť(počet informačných bitov) k a počet kontrolných bitov $n - k$.

5.5.1 Predlžovanie a skracovanie

Predlžovanie a skracovanie mení dĺžku a veľkosť, počet kontrolných bitov ostáva zachovaný. Predĺženie zväčšuje dĺžku kódu a pridáva nové kódové slová. Pre lineárne kódy je to pridávanie rovnakého počtu riadkov a stĺpcov ku generujúcej matici kódu (pridáme nulový stĺpec a následne nulový riadok, ktorý bude mať nenulovú hodnotu v poslednom doplnenom stĺpci). Skrátenie je inverzné k predĺženiu.

5.5.2 Zväčšovanie a zmenšovanie

Zväčšovanie a zmenšovanie zachováva dĺžku kódu, ale mení počet informačných a kontrolných symbolov. Pri zväčšovaní rastie počet informačných bitov a zároveň sa zväčšuje aj počet kódových slov. Zároveň však klesá počet kontrolných bitov a teda sa zmenšuje aj minimálna vzdialenosť a tým opravná schopnosť kódu. V prípade zmenšovania je to naopak. Na lineárnych kódoch sa tieto metódy realizujú jednoducho tak, že do generujúcej matice pridávame, respektíve odoberáme riadky.

5.5.3 Rozšírenie a zúženie

Pri týchto metódach si kód zachováva počet informačných symbolov a mení sa dĺžka a počet kontrolných bitov. Rozširovanie pridáva nové kontrolné symboly (pre lineárne kódy znamená pridanie stĺpca do generujúcej matice), Zúženie naopak kontrolné symboly odberá (teda odoberá stĺpce z generujúcej matice).

Kapitola 6

Náš Návrh

Momentálne existujúce techniky a výskum v oblasti ochrany kódu sa zameriavajú predovšetkým na čo najväčšiu odolnosť mechanizmu pred odstránením z programu útočníkom.

Táto práca ponúka návrh systému ochrany kódu, ktorý by mal byť nielen pomerne robustný, ale taktiež vďaka svojej špeciálnej štruktúre poskytuje široké možnosti reakcie na zistené narušenie integrity. Systém dokáže presne lokalizovať miesto v programe kde došlo k manipulácii, vďaka čomu sme schopní sa ľahšie rozhodovať ako na manipuláciu zareagovať.

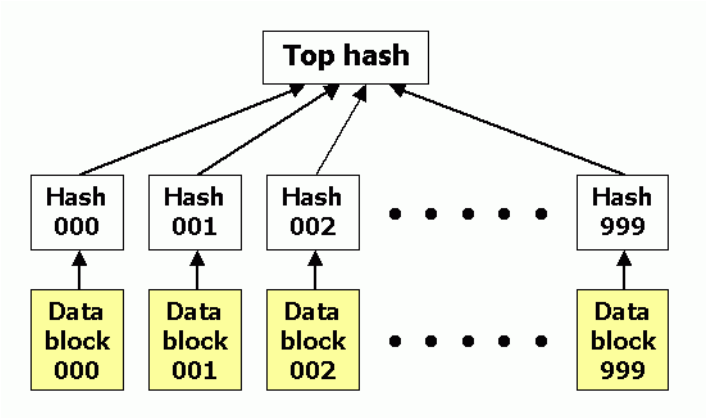
Navrhujeme systém ochrany integrity programu určený pre spustiteľné súbory pod platformou Win32. Zameriavame sa teda na formát spustiteľného súboru PE (Portable Executable). Náš návrh popisuje jednotlivé časti systému, ich požadované funkcie a význam. Nie je to však návod na implementáciu tohto systému, teda v celej práci sa nebudeme zaoberať implementačnými problémami pri jeho realizácii.

6.1 Hešovacie stromy

Základ fungovania nášho návrhu je postavený na štruktúre známej ako hešovací strom alebo tiež Merkleho strom. Pomenovaný podľa svojho objaviteľa Ralpha Merkleho v roku 1979. Tieto stromy sú rozšírením jednoduchšej štruktúry nazývanej hešovací zoznam. Idea použiť túto štruktúru pri overovaní integrity programu pochádza s krátkeho článku od Bena Mossa a Helen Ashman[18].

6.1.1 Hešovací zoznam

Hešovací zoznam je zoznam hešov blokov dát súboru alebo súborov. Táto štruktúra sa v súčasnosti využíva najmä v peer to peer sieťach, kde zabezpečuje integritu blokov dát posielaných od rôznych klientov. K samotnému zoznamu sa zvykne pridavat ešte jeden heš, nazývaný Top heš, ktorý hešuje samotný zoznam.



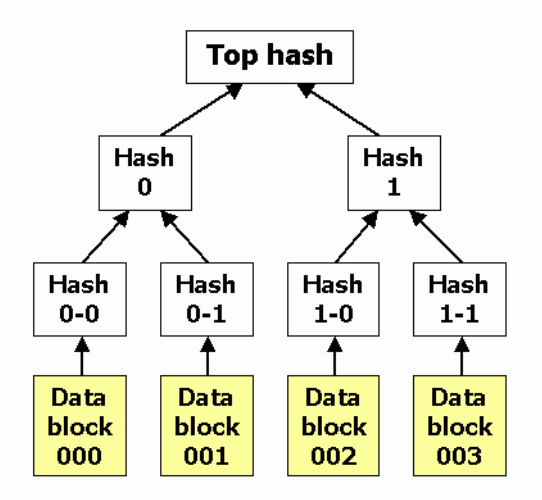
Obrázok 6.1: Hešovací zoznam

6.1.2 Hešovací strom

Samotný hešovací strom je strom hešov, v ktorom listy sú heše konkrétnych blokov dát. Uzly vyššie v strome sú heše blokov dát svojich potomkov a koreňový uzol je heš celého bloku dát. Implementácie týchto stromov sú väčšinou binárne, no koncept je rovnako funkčný pre ľubovoľné stromy, aj stromy s variabilným počtom synov pre každý vrchol.

6.2 Návrh mechanizmu

Hešovanie kódu je základom množstva už existujúcih metód na ochranu softvéru. Náš návrh odkrýva ďalšiu možnosť využitia hešovacích funkcií opierajúcu sa práve o spomínané hešovacie stromy. Táto štruktúra ponúka oproti klasickému hešovaniu kódu množstvo výhod. V dátach rozdelených hešovacím stromom na jednotlivé bloky je jednoduché presne určiť rozsah a miesto manipulácie, uľahčuje ich nápravu a má množstvo ďalších výhod.



Obrázok 6.2: Hešovací strom

6.3 Základné jednotky mechanizmu

V tejto časti načrtneme návrh základných stavebných jednotiek mechanizmu, pomenujme ich jednoducho Jednotky, ktoré budeme vkladať do cieľového programu. Tieto jednotky predstavujú uzly stromovej štruktúry, ktorú v programe vytvoríme. Ich vytváranie a aplikáciu na cieľový program si najskôr neformálne popíšeme a následne uvedieme algoritmus ich vytvárania a vkladania v pseudokóde.

6.3.1 Spôsoby vkladania

Ešte pred tým ako začneme popisovať Jednotky, musíme sa zamyslieť nad tým, ako budeme do cieľového programu vkladať celý mechanizmus. Sú možné dva prístupy a to vkladanie do zdrojových kódov programu alebo do už skompilovaného programu. Náš mechanizmus je navrhnutý tak, že sa vkladá priamo do zdrojových kódov, čo má oproti vkladaniu do skompilovaného programu výhodu v podobe výrazne jednoduchšej implementácie. Ale prináša to aj nevýhody, ako potreba počas vkladania opätovne program kompilovať a takisto nemožnosť aplikovať mechaniznus na program bez zdrojových kódov. Vkladanie podobného mechanizmu do už skompilovaného programu je spomínane napríklad v práci[2]. Nami preferované vkladanie do programových kódov sa vyskytuje napríklad v práci[3].

6.3.2 Popis Jednotiek

Malé časti programového kódu, ktoré budú tvoriť jednotlivé uzly stromu, budeme nazývať Jednotky. Všetky Jednotky, ktoré budeme vkladať do programu majú spoločnú základnú štruktúru. Tá by sa dala rozdeliť na povinné časti, ktoré obsahuje každá jednotka a voliteľné, ktoré jednotka môže a nemusí obsahovať.

- Základné údaje

  Tieto údaje sú povinné, teda musí ich obsahovať každá jednotka. Zahŕňajú, jednoznačný identifikátor uzla, ktorý je dôležitý najmä pre systém výmeny správ v strome a typ uzlu (koreň, list, vnútorný uzol). V prípade nelistového uzla identifikátory synov. Pre každý uzol okrem koreňového potrebujeme aj identifikátor rodiča. Ďalej tu potrebujeme hranice bloku dát ktorý hešujeme a v neposlednom rade predpočítanú správnu hodnotu hešu relevantného bloku dát.

- Hešovací algoritmus

  Základom činnosti každej jednotky je dobre známy princíp hešovania kódu, takže každá jednotka musí vedieť spočítať heš bloku, ktorý pokrýva. Samotný algoritmus však nemusí byť priamo jej súčasťou, môže mať v sebe aj volanie tohto algoritmu ako funkcie z parametrami hraníc bloku dát, ktorý hešuje z iného miesta v kóde (napríklad z inej Jednotky). Túto časť teda nemusí obsahovať každá Jednotka. Nikdy však nesmie byť volany len z jedného miesta, znamenalo by to porušenie vlastnosti single-point of failure spomínanej v predchádzajúcej kapitole.

- Systém na posielanie správ

  Ako neskôr ukážeme, pre fungovanie mechanizmu je nevyhnutná komunikácia medzi jednotkami. Preto každá jednotka musí obsahovať systém na posielanie správ. Systém musí byť schopný poslať správu z ľubovoľného miesta v strome na ľubovoľné, na základe identifikátora. To znamená, že potrebujeme funkciu SEND s parametrom identifikátor, ďalej potrebujeme jednoduchšie funkcie na posielanie správ synom, rodičom a súrodencom. Tu nám stačia identifikátory, ktoré už máme v jednotke, respektíve na poslanie správy súrodencom sú identifikátory uložené v ich spoločnom otcovi. Tak ako funkcie na hešovanie aj funkcia na výmenu

správ nemusí byť v každej Jednotke (dokonca nemusí byť v žiadnej, môže byť uložená úplne mimo stromu, avšak tak ako platí pre hešovací algoritmus aj tu nikdy nie len na jednom mieste).

- Systém na spracovanie správ

  Výmena správ má za cieľ doručovať správy medzi uzlami, no tie je potrebné ďalej spracovať. Ak uzol zachytí správu a táto je určená inému uzlu, posiela správu ďalej. Ak mu je určena, pozrie si obsah správy. Najbežnejším obsahom správy je v systéme požiadavka na overenie integrity (čiže spočítanie hešu), ďalej sú to potrdzovacie správy o úspešnosti hešovania a iné. Tato časť je pre každú Jednotku povinná.

- Ďalšie voliteľné údaje

  Tieto údaje nemusí obsahovať každá jednotka (respektíve žiadna). Sú to informácie o bloku dát(ak sú známe), ktorý Jednotka pokrýva, teda informácie o postupe pri zistení manipulácie práve na tejto časti dát. Týmto sa budeme zaoberať podrobnejšie v ďalšej kapitole.

- Polymorfizmus

  Veľmi dôležitá vlastnosť Jednotiek je polymorfizmus, teda ak aj dve Jednotky vykonávajú tú istú funkciu, ich kód je odlišný. Takto by totiž v programe vznikal istý opakujúci sa, ľahko odhaliteľný vzor(pattern).

6.4 Vkladanie Jednotiek do programu

Ako sme povedali, budeme Jednotky vkladať do zdrojových kódov programu. Všeobecne toto vloženie je realizované jednoducho, identifikovaním miesta v programe kde jednotku chceme vložiť a vsunutím jej kódu, makra, respektíve volania jej kódu do cieľového kódu.

Vytváranie a vkladanie jednotiek je možné robiť ručne, poloautomaticky alebo aj úplne automaticky. V prípade ručného vkladania sami vytvoríme Jednotku a ručne ju vložíme do kódu. Avšak vďaka veľmi podobnej štruktúre každej jednoky je oveľa praktickejšie mať pripravený automatický nástroj, akýsi generátor jednotiek, ktorý vytvorí kód Jednotky. My už len špecifikujeme miesto v kóde, kde sa má Jednotka vložiť. V prípade úplne automatického vkladania, môžeme

nástrojom tieto jednotky vkladať úplne samostatne. Môže ich do kódu umiestňovať náhodne, prípadne vytvoriť graf toku riadenia programu a Jednotky rovnomerne distribuovať po vetvách grafu.

Predpokladáme však, že sú určité časti programu, kde sa nám príliš nehodí vloženie Jednotky, ktorá by danú časť kódu spomalila. Vhodnou alternatívou plne automatického vkladania je teda automatické vkladanie s vyznačením obmedzenia časti kódu, kde Jednotky nemôžu byť vkladané.

6.4.1 Popis základného algoritmu vkladania

Proces vloženia mechanizmu do kódu má iteratívny charakter. Prvá fáza vkladania je nastavenie (deje sa v nástroji na vkladanie) parametrov pre vkladaný strom.

- Ako prvé potrebujeme určiť úseky dát, ktoré chceme chrániť a tie následne rozdeliť na bloky. Jeden spôsob je povedať akú percentuálnu časť úseku má jeden blok pokrývať a podľa toho určiť počet jednotiek. Čím viac blokov, tým podrobnejšie delenie. Viac uzlov čiže vyššia bezpečnosť, ale taktiež väčší nárast na veľkosti a väčšie spomalenie programu. Je preto potrebné dobre zvážiť a zvoliť kompromis medzi bezpečnosťou a zväčšením nárokov programu. Ďalšia možnosť je vytváranie blokov s ohľadom na presnú funkciu kódu vo vnútri bloku. Takéto delenie nám následne poskytuje oveľa väčšie možnosti pri reakcii na manipuláciu. Bohužiaľ je ale veľmi problematické to presne určiť. Je dobré podotknúť, že jednotlivé bloky dát nemusia mať nutne tú istú veľkosť.

  Ako uvedieme v ďalšej kapitole, na jednotlivé bloky dát pokrývané Jednotkami budeme aplikovať istý špeciálne upravný samoopravný kód, ktorý dokáže opraviť istý počet súvislých chýb v bloku dát. Tento počet je presné percento z veľkosti bloku, ak teda rozdelíme program na príliš veľký počet blokov, strácame na opravnej schopnosti pre súvislé chyby. Je preto odporúčané, aby Jednotky nepokrývali príliš malú časť programu(závisí od veľkosti, ale táto hodnota by nemala ísť pod 5% z veľkosti pokrývanej časti programu).

- Ďalší krok je určenie štruktúry, ktorú má nástroj v programe vytvárať. Teda či pôjde o strom binárny alebo n-árny, prípadne môžeme požadovať variabilné vetvenie.

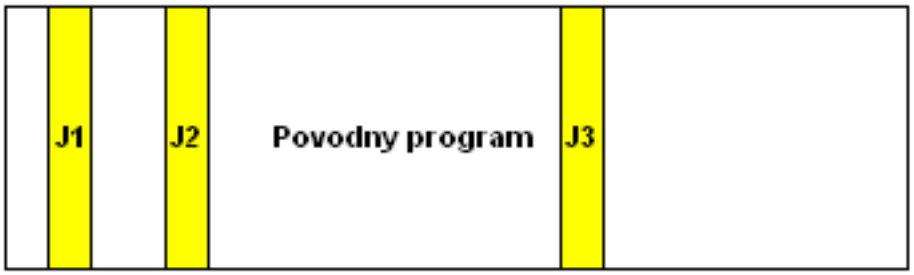
- Na základe týchto údajov program vypočíta počet jednotiek, ktoré sa majú do cieľového programu vložiť a môže začať samotné vkladanie.

Ďalej je dôležité poznamenať, že bloky, na ktoré rozdelíme program nemusia pokrývať celý program, respektíve celý program ani nikdy pokrývať nemôžu (keď vypočítame heš celého programu vrátane dát, nedokážeme ho už do neho vložiť, keďže po jeho vložení by sa hodnota hešu zmenila). Samozrejme, že hešmi môžeme pokrývať len časť programu obsahujúcu kód a read-only dáta. Dáta, ktoré menia svoju hodnotu a ukladajú sa v programe pokrývať nemožno. Medzi jednotlivými blokmi môžu byť medzery, tak ako aj bloky môžu pokrývať len nami určenú časť programu. Na fungovanie samotného mechanizmu to nemá vážnejší dosah. Každá jednotka má v sebe zoznam adries blokov na ktorých má vykonať heš a hešovací algoritmus, nevyžaduje ani, aby dáta na ktorých počíta heš boli súvisle za sebou.

Teraz si ukážeme základný algoritmus vkladania a nižšie ho podrobnejšie rozpíšeme.

```
početuzlov := n;
vsun n prázdnych Jednotiek do kódu;
for i := 1 to hĺbka stromu do begin
    for j := 1 to počet uzlov v hĺbke i do begin
    skompiluj program, vyber blok, vypočítaj heš,
    vyber ľubovoľnú voľnú jednotku, vlož do nej adresy bloku, vypočítaný heš a ID;
    podľa vetvenia stromu si pamätaj ID súrodeneckých Jednotiek;
    podľa potreby nastav Jednotke ID ostatných Jednotiek, ktoré má poznať;
    end;
end;
```

Proces samotného vkladania začneme vložením prázdnych Jednotiek do kódu. Hešovací strom budeme vytvárať po jednotlivých úrovniach, začínajúc od najhlbšej (teda od listov). Ako prvé potrebujeme vedieť adresu bloku dát a správnu hodnotu hešu pre daný blok, následne musíme program skompilovať. Bloky dát vyberáme zaradom od začiatku alebo konca súboru (teda začíname vytvárať najnižšiu úroveň stromu zľava alebo sprava). Tieto údaje následne vložíme do nami vybranej Jednotky a tejto následne pridelíme identikátor. Pri inicializácii Jednotiek im taktiež potrebujeme vložiť identifikátor



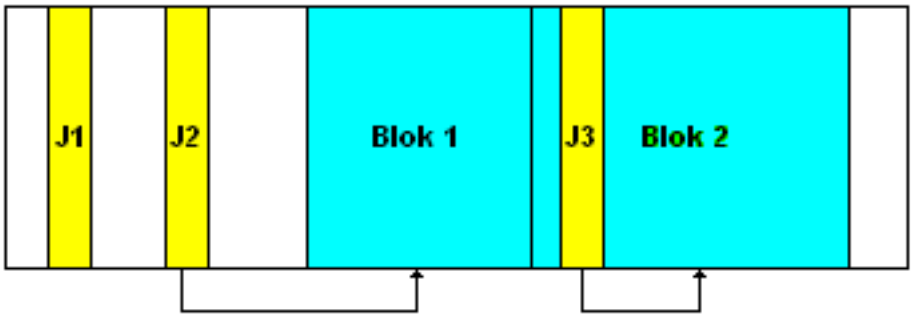
Obrázok 6.3: Vloženie prázdnych Jednotiek

ich otca, respektíve v prípade nelistových vrcholov identifikátory synov. Otec práve inicializovanej Jednotky má teda identifikátor dopredu určený, aj keď ešte nie je inicializovaný. Je to potrebné, aby sme po inicializácii jednotky už do nej nemuseli zasahovať.

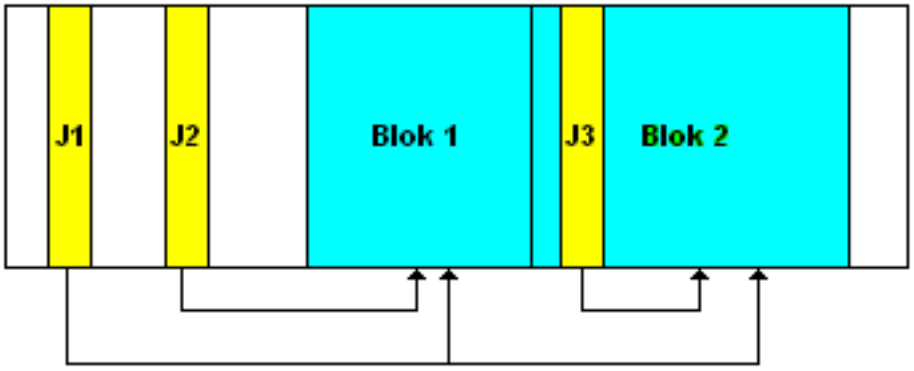
Pri výbere prázdnych jednotiek, ktoré ideme zapájať do mechanizmu, nevyberáme Jednotku, ktorá bola vsunutá hneď na začiatok programu (teda na začiatok grafu toku riadenia). Táto jednotka je vyhradená pre koreň stromu.

Obrázok 6.4: Určenie úseku v programe, ktorý chceme chrániť a jeho rozdelenie na bloky.

Ak hešmi pokrývame nielen kód, ale aj dáta programu pri celom vytvaraní stromu, potrebujeme zabezpečiť, aby premenné do ktorej vkladáme údaje o správnej hodnote hešu, adresu bloku a ID (respektíve všetky údaje, ktoré do jednotiek pridávame počas procesu vkladania), sa nenachádzali v práve hešovanom bloku dát. Taktiež v už hešovanom bloku dát sa nemôžu nachádzať ani tieto premenné ešte neinicializovaných Jednotiek.



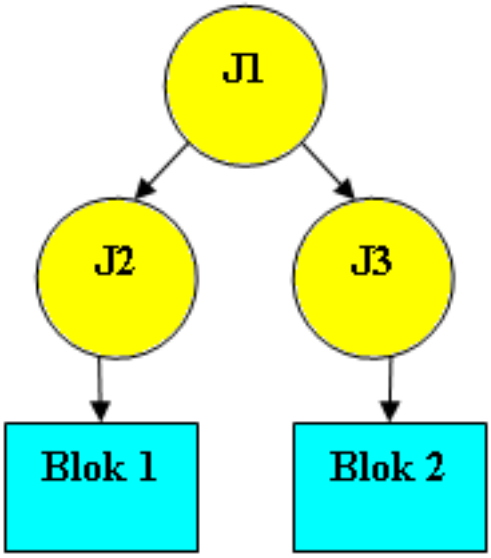
Obrázok 6.5: Priradenie listových Jednotiek k blokom a ich inicializácia. Pre jednoduchosť predpokladáme, že hešujeme len kódovu časť a teda nemusíme riešiť problémy súvisiace s hešovaním dátovej časti programu.



Obrázok 6.6: Priradenie koreňovej jednotky k celému úseku dát. Proces vkladania je týmto ukončený.

Pri vytváraní tohto stromu si na každej úrovni potrebujeme pamätať identifikátory súrodeneckých vrcholov, aby sme pri inicializácii jednotiek nasledujúcej vrstvy vedeli určiť, ktoré bloky dát má hešom pokrývať ich otec. Touto postupnou iteráciou dokážeme zabezpečiť, že aj hodnoty hešov, respektíve ostatné dynamicky vkladané informácie, môžu byť pokryté ostatnými Jednotkami, môžu to byť ich rodičia ale aj jej súrodenci. Jedine dynamicky prideľované hodnoty, ktoré nemôžu byť ničím pokryté, patria koreňovej jednotke.

Časová zložitosť(teda počet kompilácií) tohto vkladania je priamo úmerná počtu uzlov, čiže $O(n)$, kde n je počet uzlov(Jednotiek) stromu. Sú však možnosti ako túto zložitosť znížiť.



Obrázok 6.7: Štruktúra hešovacieho stromu ktorá vznikla.

6.5 Vylepšené algoritmy vkladania

Počet kompilácií priamo úmerný počtu uzlov je akcepovateľný len pri menších programoch, ale zavedenie tohto mechanizmu do veľkého systému, by bolo takto neakceptovateľne pomalé. Preto sa snažíme nájsť spôsob, ako algoritmus vkladania čo najviac skrátiť.

Naše možnosti závisia od toho, čo z programu pokrývame hešmi. V prípade, že pokrývame čisto len kódovú časť programu a nepokrývame žiadne dáta, teda ani počas procesu inicializácie jednotiek vkladané údaje, môžeme celý proces vkladania výrazne zrýchliť. V takomto prípade je totiž možné skonštruovať celý strom v jednej iterácii, keďže všetky dynamické údaje Jednotiek sú vkladané do dátovej časti programu, ktorú nepokrývame. Stačí nám preto jedna kompilácia v ktorej určíme heše pre všetky jednotky a následne ich aj s ostatnými údajmi jednoducho vložíme do programu. Výsledná časová zložitosť je teda $O(1)$.

Takéto riešenie je však výrazne na úkor bezpečnosti, keď kľúčové údaje celého mechanizmu zostávaju nechránené. Preto sa pokúsime načrtnúť spôsob zníženia počtu kompilácie, ktorý počíta aj s hešovaním dátových (read-only) častí programu.

Keď chceme pokryť aj dátovu časť programu(dáta, ktoré sú nemenné počas exekúcie programe teda konštanty), kde sú uložené aj naše hodnoty hešov, ID

Jednotiek a ostatné údaje nemôžeme vytvoriť celý strom naraz, keďže by sme po skompilovaní programu a vypočítaní hešov ich následne určité vkladali do časti programu, ktorú už pokrývajú niektoré heše v našom strome (je to cesta v strome začínajúca vrcholom a končiaca v liste alebo niekoľkých listoch, ktoré hešujú daný blok dát). Je teda zrejmé, že nemôžeme inicializovať naraz dokonca ani len všetky uzly v jednej hĺbke, keďže nevieme presnú pozíciu v programe, kde sa dáta ukladajú (pred tým ako sa tam uložia). Ďalší problém vzniká, že v každej hĺbke stromu musí zostať jedna hodnota hešu nechránená (nie je možné vytvoriť cyklus). Nie je však nutné inicializovať len jednu Jednotku naraz.

Môžeme znížiť počet kompilácií tak, že počas jednej iterácie inicializujeme viac ako jednu Jednotku. Pri tejto aktivácii vždy sledujeme či hodnoty jednej z Jednotiek nie sú práve v časti programu, ktorú hešuje druhá jednotka(respektíve, či hodnoty nevkladá do časti programu, ktorú sama hešuje). Ak sa nám niektorú z jednotiek nepodarí inicializovať vyberieme v ďalšej iterácii nejakú inú. Ak pokrývame celú dátovú časť programu, na konci inicializácie ostane vždy jedna Jednotka, ktorú už nebude možné inicializovať. Tú budeme musieť vynechať. V optimálnom prípade by sa nám takto na každej úrovni stromu znížil počet potrebných kompilácií $1/n$, pričom n je počet Jednotiek inicializovaných pri jednej iterácii.

Zaujímavou možnosťou, ktorá by nám umožnila vkladanie na jeden krok je taktiež zabudovanie podpory procesu vkladania Jednotiek priamo do kompilátora. Samotné iterácie vkladania by potom mohli bežať vo vnútri procesu kompilácie, pričom kompilátor by nemusel znovu a znovu kompilovať časti programového kódu, ktoré sú už od prvej iterácie nemenné.

6.6 Beh mechanizmu

Pozrieme sa teraz na typický beh celého mechanizmu po spustení programu do ktorého bol aplikovaný. Po spustení programu sa aktivuje koreňová bunka. Ak odhalí manipuláciu, predáva riadenie svojim deťom. Ten syn, ktorý odhalí manipuláciu, pokračuje a predáva riadenie svojim deťom. Toto je prvá fáza kontroly. Ak sa zistí manipulácia, postupuje sa podľa údajov v najspodnejšej Jednotke(Jednotkách), ktorá odhalila manipuláciu (reakciou na takéto zistenie sa zaoberáme v ďalšej kapitole).

Ak by však celý systém nebol prepojený s programom všetkými jednotkami,

ale celý strom by sa počítal na jednom mieste, respektíve fakt, že nemôžeme pokryť hešom koreňovú jednotku, bol by to jasný "single-point of failure".

V takomto návrhu sú však jednotlivé jednotky prepojené s kódom, takže aj po odstránení koreňovej jednotky sú počas vykonávania programu aktivované ďalšie Jednotky, ktoré následne začnú kontrolovať integritu dát vo svojom podstrome. Keď sa beh programu dostane na miesto, kde je vložená nejaká Jednotka, táto pokiaľ nemá implicitne nastavené, že má spustiť detekciu manipulácie len na požiadavku od inej Jednotky, samočinne začne s kontrolou. Spomalenie je však akceptovateľné, pretože ak ku žiadnej manipulácii nedošlo, počítať sa bude len jeden heš.

Kapitola 7

Riešenia narušenia integrity

Mechanizmus zabudovanej ochrany softvéru sa skladá z dvoch hlavných častí.

- Detekcia narušenia integrity.
- Náprava narušenia integrity.

Detekcia narušenia integrity by bola sama o sebe zbytočná, keby po nej nenasledovalo isté, nami zvolené riešenie tejto manipulácie. Existuje množstvo postupov a metód, ktoré sa po vzniknutí manipulácie môžu zvoliť a boli už v minulosti navrhnuté. Táto kapitola ponúka ich prehľad a zároveň prináša návrh metódy založenej na teórii samoopravných kódov, s ktorou sa v súvislosti s ochranou integrity softvéru zatiaľ veľmi nezaoberalo.

7.1 Klasifikácia prístupov

7.1.1 Logovanie

Prvá možnosť, ktorá nás napadne, je samozrejme vytvoriť o detekovanej manipulácii záznam. Najjednoduchšia možnosť je, že tento záznam je vlastne len boolovská hodnota, teda ÁNO došlo k manipulácii alebo NIE nedošlo k manipulácii. Samozrejme je ale rozumné, aby tento záznam obsahoval čo najviac informácií, ktoré nám prípadne v budúcnosti môžu pomôcť. Je sem možné uložiť rozsah manipulácie, podľa počtu blokov, ktoré boli poškodené, prípadne aj presné identifikátory týchto blokov. Taktiež je možné uložiť informácie o prostredí, respektíve počítači, v ktorom sa program nachádza. Taktiež pri každom novom zistení zásahu do kódu môžeme inkrementovať počítadlo, táto

informácia nám môže poslúžiť ako indikátor či bola manipulácia náhodná alebo cielená.

Ak si dáme prácu s vytváraním takéhoto logu, prirodzene chceme, aby sa k nám tieto informácie dostali alebo aspoň aby ich útočník v programe príliš jednoducho nenašiel. Pridávať ich priamo do programu evidentne môžeme len na miesto v programe, ktoré nie je pokryté žiadnym hešom, ak teda zvolíme túto možnosť musíme s tým počítať dopredu. Ďalšou možnosťou je zapísať údaje niekam na disk v ktorom sa program nachádza, to však zvyšuje šancu na odhalenie logovania a nakoniec aj na odstránenie mechanizmu, keďže útočník môže údaje z logu využiť. Ak je počítač pripojený k sieti a spojenie nie je ničím blokované, môže sa program pokúsiť poslať log na dopredu známu adresu v sieti, kde môžeme log zachytiť. Pokus o nadviazanie takéhoto spojenia však tiež obnáša zvýšené riziko odhalenia.

7.1.2 Samodeštrukcia

Samodeštrukcia programu znamená, že po zistení manipulácie sa program sám nejakým spôsobom poškodí. Triviálna možnosť je vymazať časť programu alebo celý program. Toto však v prípade pokusu útočníka o cracknutie programu určite neprinesie žiadny osoh. Rafinovanejšia možnosť je pozmeniť niektoré časti programu tak, aby produkovali chybné výsledky, v prípade hier to môže byť napríklad neúmerné zvýšenie obtiažnosti a podobne. Mechanizmus môže byť nastavený tak, aby po zistení cielenej manipulácie ďalej nereagoval na zásahy a tváril sa akoby sa útočníkovi podarilo zlomiť jeho ochranu...

7.1.3 Opravné mechanizmy

V niektorých špecifických prípadoch je vhodné, aby škody vzniknuté úmyselným či neúmysleným zásahom boli napravené do pôvodného stavu. Je niekoľko možností ako to docieliť. Triviálny spôsob je držať niekde v programe kópiu dát, ktoré sa snažíme ochrániť. Po detekovaní manipulácie sa poškodené časti programu jednoducho prepíšu touto funkčnou kópiou. Vylepšením tohto prístupu je držať dáta v zakryptovanej podobe, čo útočníkovi mierne sťaží prácu. Podobne ako pri logovaní aj tu je možnosť pripojenia na sieť a vyžiadania si dát z iného zdroja.

Všetky tieto prístupy sú však pomerne málo odolné pred odstránením, prípadne kladú na program špecifické požiadavky. Navrhujeme preto systém, ktorý dokáže istú rozumnú časť dát opraviť pomocou pomerne malej množiny redundantných dát.

7.2 Samoopravné mechanizmy

Naším návrhom je použitie samoopravných kódov a ich zabudovanie do programu. V teórii ochrany softvéru sa touto metódou až doposiaľ nikto nezaoberal a preto je vhodné načrtnúť akýsi základ kadiaľ by mohla viesť cesta.

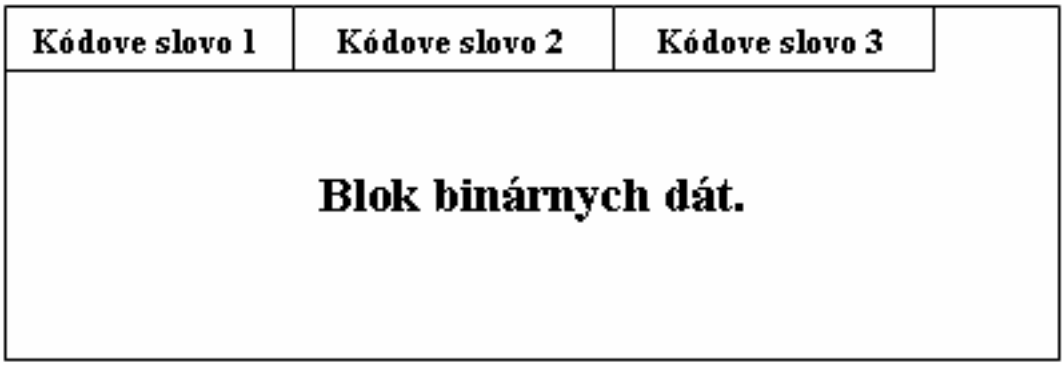
Nech už na opravu použijeme ľubovoľný kód, pri implementovaní nášho mechanizmu budeme potrebovať do jednotiek doplniť dve veci. Algoritmus na dekódovanie kódu a samotné redundantné dáta. Algoritmus je možné do jednotky vložiť rovnako ako algoritmus na výpočet hešu. Redundantné dáta musíme vkladať rovnakým postupom, respektíve zároveň popri vkladaní hešov, keďže sú taktiež závislé na výslednom skompilovanom súbore. Pri výbere prvého bloku vypočítame okrem hešu aj redundantné dáta prislúchajúce k danému bloku pre nami zvolený samoopravný kód a následne tieto dáta vložíme do programu. Táto jednotka je teda už kompletná, čiže ďalej vytvárané jednotky ju môžu pokrývat hešmi.

Na tento účel sa však nedajú použiť všetky samoopravné kódy. Väčšina kódov realizuje transformáciu, ktorá z istého binárneho vektora vytvorý iný, dlhší, ktorý tým získa konštantnú opravnú schopnosť. Avšak aby sme vedeli takýto kód aplikovať na binárne dáta, potrebovali by sme z výsledného binárneho vektora určiť, ktoré bity sú informačné a ktoré sú redundantné. Redundantné bity by sme si následne mohli odložiť a zapamätať si k nim pozície, na ktorých sa nachádzali v kódovom slove. To však nevieme.

Preto sa na účely opravy dát v binárnych programoch dajú použiť len kódy, ktoré majú vlastnosť oddeliteľnosti informačných a redundantných bitov a zároveň informačné bity pôvodného kódového slova zostávajú nemenné.

Takýmto kódom je napríklad Hammingov kód. Jeho popis a fungovanie sme si ukázali v kapitole o samoopravných kódoch. Pre naše účely je najvhodnejšie jeho rozšírenie $(7, 4)$. Je tak preto, lebo táto jeho varianta má v našom prípade najväčšiu opravnú schopnosť. Máme teda kódové slovo dĺžky 7, z toho 4

informačné bity. Ak by sme ho však aplikovali klasickým spôsobom, dokázali by sme opravovať iba isté percento náhodných chýb. My však chceme opravovať hlavne takzvané zhlukové chyby, teda množstvo súvislých chýb po sebe, čo najviac zodpovedá charakteru manipulácie s určitou časťou kódu. Aby sme to dosiahli, použijeme techniku zvanú interleaving. Táto technika funguje tak, že kódové slová v bloku dát neberieme za sebou, ale ich nejakým spôsobom prehadzujeme. Výsledok je taký, že po sebe nasledujúce bity sú súčasťou rôznych kódových slov.



Obrázok 7.1: Klasické vytváranie kódových slov na bloku dát

Obrázok 7.2: Vytváranie kódových slov z použitím interleavingu

Pomocou tejto metódy sme schopní aplikovať kód, ktorý na bloku dát dokáže odstrániť zhlukovú chybu v istom ohraničenom rozsahu. Aby sme však pomocou takýchto samoopravných kódov dokázali napraviť úmyselnú

manipuláciu s programom, potrebujeme opravovať pomerne veľké množstvo chýb za sebou. Preto musíme ísť v našich úvahách o aplikácii interleavingu ešte ďalej. Povedzme, že blok dát bude mať 40000 bitov. Ak teda použijem Hammingov kód $(7, 4)$ na tomto bloku dát vytvoríme 10000 kódových slov a celková veľkosť sa zväčší na 70000 bitov. Môžeme teda tieto kódové slová vytvárať aj tak, že prvé kódové slovo bude mať svoj prvý informačný bit na pozíci 1, druhý na pozícii 10001, tretí na pozícii 20001, atď. Druhé kódové slovo bude mať potom svoje bity na pozíciách 2, 10002, 20002, 30002. Samozrejme algoritmus dekódovania kódových slov musí mať údaje o tom, ako sú na danom bloku dát vytvorené kódové slová k dispozícii. Pri takomto rozmiestnení kódových slov dokážeme opraviť zhlukovú chybu o veľkosti 10000 bitov, teda približne 14 % súvislých dát vo výslednom bloku (aj s redundantnými bitmy). Hammingov kód $(7, 4)$ má teda pomerne dobrú opravnú schopnosť, vyznačuje sa však veľkou redundanciou($R_H = ((n - k)/n) = 0,428$). V prípade, že zvolíme jeho rozšírenie $(15, 11)$, zmenšíme redundanciu($R_H = 0,266$) no zároveň kód stratí na opravnej schopnosti. Je to práve pre spôsob akým ho používame(interleaving), teda nám vyhovujú čo najkratšie kódové slová. Aký kód sa použije, závisí na konkrétnom programe a potrebách vývojára (teda či potrebuje čo najväčšiu opravnú schopnosť alebo obmedziť redundanciu), ale používať kódy z $n > 15$ už pre naše účely nemá zmysel.

Obrázok 7.3: Vytváranie kódových slov s preskakovaním.

7.3 Reakcia nášho mechanizmu na narušenie integrity

Predstavili sme si niekoľko možností, ktoré máme po zistení narušenia integrity. Ako bude naň reagovat náš mechanizmus je ponechané vo väčšej miere na užívateľa, ktorý do svojho programu mechanizmus aplikuje. Každá individuálna Jednotka môže mať v sebe zabudovaný systém na zápis informácií do logu, poslanie správy cez sieť alebo kód, ktorý znehodnotí program. Jednotky nachádzajúce sa vyššie v strome môžu mať komplexnejší systém na vyhodnotenie manipulácie na základe informácií od svojich detí. Napríklad môže vyzerať takto.

1. Bolo zistené narušenie integrity, pošli požiadavku na svojich synov o ďalšie vyhodnotenie.
2. IF heš ľavého syna nesedí AND heš pravého syna nesedí DO Scenar1;
3. IF heš ľavého syna nesedí AND heš pravého syna sedí DO Scenar2;
4. ...

Konkrétnymi scenármi ako ktorá jednotka reaguje na zistenú manipuláciu sa tu nezaoberáme, keďže to je závislé na konkrétnom programe, na ktorý je mechanizmus aplikovaný. Stačí povedať, že všetky vyššie spomínané spôsoby môžu byť zabudované do ľubovoľnej Jednotky a keďže súčasťou nášho návrhu je prítomnosť systému na posielanie správ medzi Jednotkami, do konkrétnej Jednotky už je potrebné zabudovať len algoritmus na vyhodnotenie manipulácie.

Ukážeme si ešte jeden špeciálny prípad, ktorý nastáva, ak sme sa rozhodli použiť samoopravný mechanizmus. V takom prípade je vhodne zabezpečiť niekoľko vecí.

1. Ak blok dát A_i, na ktorý ideme aplikovať samoopravný mechanizmus má susedov $A_i - 1$ a $A_i + 1$ a tieto bloky v programe sú priamy susedia bloku A_i(môžu byť susedia v strome ale v programe medzi nimi môže byť ponechané miesto, ktoré sme sa rozhodli nepokryť), musíme preveriť aj integritu blokov $A_i - 1$ a $A_i + 1$.

2. Ak zistíme narušenie integrity v týchto susedoch, potrebujem znova preveriť ich susedov až kým nenájdem celý úsek ,na ktorom je potrebné vykonať nápravu.

3. Úspešnosť nápravy overíme jednoducho, znovuspočítaním hešu prislúchajúceho bloku dát. Robíme tak preto, že ak aplikujeme ľubovoľný samoopravný kód, ten má isté maximum chýb, ktoré dokáže odhaliť. Takže ak vzniká v kódovom slove chýb viac, toto náš kód nedokáže zistiť. Napríklad spomínaný Hamming $(7, 4)$ dokáže opraviť chybu váhy 1 ale aj odhaliť len chybu váhy 1. Preto pre overenie či sme boli schopní úspešne napraviť relevantný blok dát, vypočítame na ňom znovu prislúchajúci heš.

Kapitola 8

Metriky, odolnosť mechanizmu

Na záver práce sa pozriem na jednotlivé metriky, ktoré sa dajú pri našom mechanizme hodnotiť a pozrieme sa na jeho odolnosť pred útokmi.

8.1 Metriky

V zásade môžeme pri všetkých samokontrolných mechanizmoch uvažovať o týchto základných metrikách.

- Nárast veľkosti programu

  O koľko sa program zväčší po aplikácii samokontrolného mechanizmu.

- Spomalenie behu programu

  Priemerná doba spomalenia behu programu.

- Odolnosť pred Odstránením

  Odolnosť mechanizmu pred odstránením útočníkom.

- Opravná schopnosť

  Schopnosť mechanizmu napraviť vzniknuté škody.

Preskúmame teraz tieto metriky na našom návrhu.

8.1.1 Nárast veľkosti programu

Nárast veľkosti programu v prípade nášho mechanizmu závisí od niekoľkých premenných. Prvou premennou je počet Jednotiek k vložených do programu. Ďalšou je veľkosť každej Jednotky, označme ju j, ktorá závisí od miery zdieľania prostriedkov medzi Jednotkami (volanie jednej hešovacej funkcie dvoma rozličnými Jednotkami) a taktiež podľa veľkosti algoritmov na riešenie manipulácie. Je však zrejmé, že veľkosť jednej Jednotky bude v porovnaní s veľkosťou priemerného programu zanedbateľná (závisí však od konkrétnej implementácie, preto ju tu nebudeme presne vyjadrovať). Do programu pridáme konštantné množstvo týchto jednotiek, takže celkový nárast vyjadríme teda ako $k * j$. Poslednou premennou, ktorá vplýva na nárast veľkosti je množstvo redundantných dát. To je závislé od použitého samoopravného kódu a počtu Jednotiek ktoré majú opravnú schopnosť, označme ho ako $k^{'}$. Napríklad, ak by sme zobrali do úvahy Hammingov kód $(15, 11)$, kde na každých 11 informačných bitov by pripadli 4 redundantné a aplikovali by sme ho na celý program, horná hranica nárastu by bola asi 27% v prípade, že by sme samoopravným kódom pokrývali celý program. V prípade nami preferovaného kódu $(7, 4)$ by horná hranica nárastu predstavovala 75%. Celkový nárast veľkosti programu teda vyjadríme ako $k * j + k^{'} * r$, kde r je objem redundantných dát(závislý od zvoleného kódu) prislúchajúcich k danej Jednotke.

8.1.2 Spomalenie behu programu

Spomalenie programu je závislé od konkrétneho vykonávania programu, teda na koľko Jednotiek sa pri toku riadenia narazí. Vždy pri začiatku vykonávania sa však aktivuje koreňová jednotka, ktorá spustí hešovací algoritmus, tie majú však lineárnu časovú zložitosť, takže spomalenie v prípade, že nedošlo k porušeniu integrity bude zodpovedať zbehnutiu jedného hešovacieho algoritmu a jednému porovnaniu hešu. Samozrejme k rovnakému spomaleniu dôjde pri každom spustení Jednotiek napojených na beh programu. Ak však k narušeniu došlo, Jednotka posunie riadenie o úroveň nižšie a tu sa už za predpokladu binárneho stromu počítajú dva heše. V najhoršom prípade, teda ak bolo manipulované s každým blokom dát, ktoré pokrývajú listové Jednotky, bude sa počítať toľko hešov, koľko má celý hešovací strom uzlov. Teda $k * O(n)$, kde k je počet uzlov stromu a $O(n)$ je časová zložitosť hešovacieho algoritmu.

Presne vyjadríme spomalenie behu programu bez manipulácie ako $k * O(n)$, kde k je počet Jednotiek aktivovaných počas behu programu a $O(n)$ je časová zložitosť hešovacieho algoritmu. Premenná k sa dá stanoviť presne pre daný beh programu, kedže presne vieme kde v programe sa Jednotky nachádzajú (vkladali sme ich tam my alebo automatický nástroj, ktorý si ich pozíciu vie zapamätať).

8.1.3 Opravná schopnosť

Táto metrika je závislá od zvoleného samoopravného kódu a veľkosti častí programu na ktorú kód aplikujeme. Využívame techniky, ktoré sa zameriavajú na čo najlepšie výsledky pri odstraňovaní zhlukových chýb, no tým strácajú isté schopnosti pri odstraňovaní náhodných chýb. Ako sme ukázali v predchádzajúcej kapitole pri využití Hammingovho kódu $(7, 4)$ dokážeme opraviť 14% súvislých poškodených dát.Ak by sme teda aplikovali tento kód na program ako celok, sme schopný napraviť zásah útočníka, ktorý pozmenil súvislých 14% programu (programu do ktorého sme v prípade tohto kódu pridali 75% redundantných dát). Hornú hranicu opravnej schopnosti môžeme teda vyjadriť ako $k * s/n$, kde k je veľkosť časti programu, ktorú pokrývame Jednotkami s opravnou schopnosťou, s je opravná schopnosť zvoleného kódu a n je veľkosť programu. Tu spomenutý Hamming $(7, 4)$ má, z kódov ktoré sme skúmali, najväčšiu opravnú schopnosť, preto do uvedeného vzorca môžeme za s dosadiť hodnotu 14%.

8.1.4 Odolnosť pred odstránením

Odolnosť pred odstranením v podstate nie je metrikou v pravom slova zmysle, keďže ju nemôžeme stanoviť jednoznačne. Nestojíme totiž len proti automatickým nástrojom, ale stojíme proti ľudskému protivníkovi. Môžeme preto len zhodnotiť niekoľko vecí o ktorých vieme, že útočníkom znepríjemňujú život pri pokusoch o zlomenie ochrany softvéru. V kapitole venovanej útokom sme spomínali základné charakteristiky, ktoré by dobrý samokontrolný mechanizmus mal spĺňať. Pozrieme sa ako je z hľadiska týchto charakteristík na tom náš návrh.

- Kontrola počas celého behu

Jednou zo zvláštností dobrého samokontrolného mechanizmu bola kontrola integrity počas celého behu. Túto podmienku náš mechanizmus spĺňa, keďže jednotlivé Jednotky sú previazané s tokom riadenia v programe a priebežne počas exekúcie vykonávajú overovanie integrity.

- Single-point of failure

  Taktiež vďaka previazaniu s behom programu a distribúcii Jednotiek po celom programe neobsahuje jedno miesto manipulovanie, s ktorým by vyradilo z činnosti celý systém.

- Viacnásobná ochrana

  Mechanizmus spĺňa aj túto požiadavku, keďže jedno miesto v programe hešuje až n Jednotiek, pričom n je výška nami zvoleného stromu.

Kapitola 9

Záver

V tejto práci sme navrhli možný spôsob ochrany softvéru pred neželanou manipuláciu. Využíva vedomosti a postupy predchádzajúcich návrhov a navyše pridáva niekoľko nápadov doteraz v žiadnej práci nepreskúmaných. Ide najmä o aplikáciu samoopravných kódov, tak ako aj podrobnejší návrh použitia hešovacích stromov pre ochranu integrity programov. V kapitolách 2, 3 a 4 sme sa najskôr pozreli na celkový prehľad ochrany softvéru podrobnejšie sme sa pritom zamerali na samokontrolné mechanizmy. V kapitole 5 sme si ukázali základy samoopravných kódov, ktoré v našom návrhu využívame. Kapitoly 6, 7 a 8 sa už venujú samotnému návrhu, menovite kapitola 6 hovorí o návrhu mechanizmu na overovanie integrity a jeho vkladanie do programov. Navrhli sme základnú štruktúru mechanizmu, popísali sme algoritmus jej vloženia do kódu, ako aj možnosti zlepšenia jeho časovej zložitosti. Kapitola 7 hovorí o časti mechanizmu, ktorá má na starosti riešenie narušenia integrity. Tu sme najskôr zhrnuli možnosti predostreté v inych prácach ktoré sú využiteľné aj v našom mechanizme. Tažiskom tejto kapitoly je potom náš návrh využitia samoopravných kódov ako jednej z možností riešenia narušenia integrity programu. Kapitola 8 nakoniec pojednáva o metrikách skúmaných na našom návrhu.

Literatúra

[1] D. Aucsmith. Tamper resistant software: An implementation. Proceedings of the First International Workshop on Information Hiding, volume 1174 of Lecture Notes in Computer Science, pages 317–333. Springer- Verlag, 1996.

[2] H. Chang and M. Atallah. Protecting software code by guards. In Proceedings of the 1st ACM Workship on Digital Rights Management (DRM 2001), volume 2320 of Lecture Notes in Computer Science, pages 160–175. Springer-Verlag, 2002.

[3] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinba, and M. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In Proc. 5th Information Hiding Workship (IHW), volume 2578 of Lecture Notes in Computer Science, pages 400–414, Netherlands, Springer-Verlag, 2002.

[4] Daniel Olejár, Martin Stanek. Úvod do teórie kódovania. FMFI, 2006.

[5] C. S. Collberg, C. Thomborson. Watermarking, tamper-proofing, and obfuscation: Tools for software protection. IEEE Transactions on Software Engineering, 28(8):735–746, 2002.

[6] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz. Specifying and verifying hardware for tamper-resistant software. In Proceedings of the 2003 IEEE Symposium on Security and Privacy, page 166. IEEE Computer Society, 2003.

[7] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz. Architectural support for copy and tamper resistant software. In

Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 168–177. ACM Press, 2000.

[8] Cloakware Corporation. Introduc- tion to cloakware tamper-resistant software (trs) technology

[9] Masahiro Mambo, Takanori Murayama, and Eiji Okamoto. A tentative approach to constructing tamper-resistant software. New Security Paradigms Workshop, Proceedings of the 1997 workshop on New security paradigms. ACM Press, 1998.

[10] W. Cary Huffman, Vera Ples. Fundamentals of error-correcting codes. Cambridge University Press, 2003.

[11] Michael Purser. Introduction to error-correcting codes. Artech house, 1995.

[12] Mikhail J. Atallah, Eric D. Bryant, Martin R. Stytz. A Survey of Anti-Tamper Technologies. www.stsc.hill.af.mil/CrossTalk/2004/11/0411Atallah.html

[13] Andrew Griffiths. Binary protection schemes. 2006, www.codebreakers-journal.com

[14] Mayur Kamat, Nishant Kumat. Code Obfuscation. logic.pdmi.ras.ru/~yura/en/stuttgart-talk.pdf

[15] Greg Hoglund, Gary McGraw. Exploiting Software: How to Break Code, chapter 3: Reverse Engineering and Program Understanding. Addison-Wesley Professional, 2004.

[16] Elliot J. Chikofsky, James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, Volume 7, pages 13 - 17, ACM Press, 1990.

[17] Ben Moss, Helen Ashman. Hash-Tree Anti-Tampering Schemes. n Proceedings IEEE International Conference on Information Technology and Applications, Bathurst, Australia, 2002.

[18] M. Madou, B. Anckaert, B. De Sutter, K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In Proceedings of the 5th ACM Workshop on Digital Rights Management, pages 75 - 82. ACM Press, 2005.

[19] G. Myles, S. Nusser. Content protection for games. IBM Systems Journal Volume 45, Pages: 119 - 143, IBM Corp., 2006.

[20] Mikhail J. Atallah, Victor Raskin, Christian F. Hempelmann, Mercan Karahan, Radu Sion, Umut Topkara, Katrina E. Triezenberg. Natural Language Watermarking and Tamperproofing. Lecture Notes In Computer Science; Vol. 2578, Revised Papers from the 5th International Workshop on Information Hiding, ACM Press, 2002.

[21] Gleb Naumovich, Nasir Memon. Preventing Piracy, Reverse Engineering, and Tampering. Computer Volume 36, Pages: 64 - 71, IEEE Computer Society Press, 2003.

[22] P.C. van Oorschot. Revisiting Software Protection. Information Security, Volume 2851/2003, pages 1 - 13, Springer-Verlag, 2003.

[23] Chenxi Wang, Jonathan Hill, John Knight, Jack Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report: CS-2000-12, ACM Press, 2000.

[24] Ross Anderson, Markus Kuhn. Tamper Resistance - a Cautionary Note. Proceedings of the Second Usenix Workshop on Electronic Commerce, pages 1 - 11, 1996.

[25] Ping Wang. Tamper Resistance for Software Protection. A Thesis for the Degree of Master of Science, School of Engineering Information and Communications University, 2005.

[26] Deepa Kundur, Dimitrios Hatzinakos. Digital Watermarking for Telltale Tamper Proofing and Authentication. Proceedings of the IEEE volume 87, pages 1167 - 1180, IEEE Computer Society Press, 1999.

[27] Christian Collberg, Clark Thomborson. Software Watermarking: Models and dynamic embeddings. Proceedings of the 26th ACM SIGPLAN-

SIGACT symposium on Principles of programming languages, pages 311 - 324, ACM Press, 1999.

[28] Toshio Ogiso, Yusuke Sakabe, Masakazu Soshi, Atsuko Miyaji. Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis. Technical Report CS-2000-12, 2000.

[29] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. Malware Detection, Part V, pages 253 - 289, Springer-Verlag, 2007.

[30] B. Horne, L. Matheson, C. Sheehan, R. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In Proceedings of the 1st ACM Workshop on Digital Rights Management (DRM 2001), volume 2320 of Lecture Notes in Computer Science, pages 141–159. Springer-Verlag, 2002.

[31] G. Wurster. A generic attack on hashing-based software tamper resistance. Master's thesis, Carleton University, Canada, 2005.

[32] Wikipedia, the free encyclopedia. www.wikipedia.org