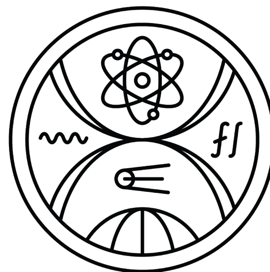
UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



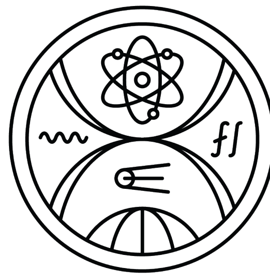
FORMÁLNE METÓDY A INFORMAČNÝ TOK

Diplomová práca

2022

Bc. Michal Pázmány

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



FORMÁLNE METÓDY A INFORMAČNÝ TOK

Diplomová práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	2511 Aplikovaná informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	doc. RNDr. Damas Gruska, PhD.

Bratislava, 2022 Bc. Michal Pázmány

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Michal Pázmány
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Formálne metódy a informačný tok
Formal methods and information flow

Anotácia: Cieľom práce je výskum (prípadne vyvoj softvérového nástroja) v oblasti bezpščnosti založenej na absencii informačného toku.

Cieľ: Cieľom práce je výskum (prípadne vyvoj softvérového nástroja) v oblasti bezpščnosti založenej na absencii informačného toku.

Vedúci: doc. RNDr. Damas Gruska, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 01.12.2020

Dátum schválenia: 03.12.2020

prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.... študent

.... vedúci práce

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne len s použitím uvedenej literatúry a za pomoci konzultácií u môjho školiteľa.

. . . .

Bratislava, 2022 Bc. Michal Pázmány

Poďakovanie

Poďakovanie

Abstrakt

Táto diplomová práca sa venuje otázke dôveryhodnosti programov a možným únikom informácií v zdrojovom kóde. Ako jedno z riešení sa uvažuje o použití typových systémov s definovanou politikou dôvernosti. Obsahuje analýzu informačných tokov v oblasti objektovo orientovaných jazykov, úvod do teórie typových systémov a príklady únikov informácií. Popisuje prehľad teórie a riešení v oblasti objektovo orientovaných jazykov a jasne informuje čitateľa o postupoch takéhoto analýzy. Ďalej sa opisuje návrh verifikátora informačného toku pre jazyk Java, lexikálna analýza potrebná pre spracovanie zdrojového kódu a implementácia webovej aplikácie. Na príkladoch sa overuje jej funkčnosť.

Kľúčové slová: informačný tok, typovací systém, noninterferencia, lexikálna analýza, objektovo orientované programovanie, Java

Abstract

This thesis focuses on the issue of program trustworthiness and possible information leaks in source code. One possible solution to this problem is the use of type systems with a defined privacy policy. It includes an analysis of information flows in object-oriented languages, an introduction to type system theory, and examples of information leaks. It provides an overview of the theory and solutions in the field of object-oriented languages and clearly informs the reader of the procedures for such analysis. The thesis also describes the design of an information flow verifier for the Java language, the lexical analysis required for processing source code, and the implementation of a web application. Its functionality is verified using examples.

Keywords: information flow, type system, noninterference, lexical analysis, object oriented programming, Java

Obsah

1 Úvod **1**

2 Analýza informačného toku **4**

2.1 Základný princíp 5

2.2 Noninterferencia 7

2.3 Princípy typovania 9

2.4 Konkurencia 13

2.4.1 Typovací systém pre konkurentné programy 18

3 Analýza existujúcich riešení **24**

3.1 JIF 24

3.2 SPARK Examiner 24

4 Lexikálna analýza **25**

4.1 ANTLR 25

5 Implementácia riešenia **26**

5.1 Programovací jazyk 26

5.2 Postup 26

5.3 Testovanie 26

6 Záver **27**

Kapitola 1

Úvod

Dnes žijeme v rýchlej dobe, v dobe internetu, v dobe smartfónov, v dobe, v ktorej neustály prúd informácií nás obklopuje. Informácie sú ľahšie dostupné ako kedykoľvek predtým, ale vďaka tejto dostupnosti dát vznikajú čoraz väčšie problémy s únikmi informácií, ktoré by mali zostať tajné. Tento jav sa môže vidieť takmer každý deň, pre ľudí sú ale najdôležitejšie úniky ich vlastných informácií, ktoré poskytli webovým stránkam, ktoré ich ale nedokázali zabezpečiť. Ako by sa malo postupovať pri navrhovaní používateľských aplikácií, aby podobné riziká boli čo najviac minimalizované? Existujú rôzne pohľady na túto otázku. Známym prístupom je napríklad systém bezpečnostných práv, ale ten nedokáže úplne zabezpečiť, že informácia neprenikne na miesta, kde by mohla byť sponzorovaná útočníkom, ktorý by na základe hierachie práv nemal k takejto informácii prístup. Ako jedným z možných riešení tohto problému sa ponúka statická analýza informačného toku.

Na začiatok je potrebné definovať politiku ochrany súkromia, na základe ktorej jednoznačne určíme, či došlo k možnému úniku dôveryhodných informácií alebo nie. Touto politikou určíme, kto má práva s akými informáciami nakladať a aké operácie s nimi sú povolené, aby sa predišlo porušeniu tejto politiky.

Analýza toku informácií bude nástrojom na overenie dodržania tejto politiky a na základe jej výsledku budeme môcť povedať, či program splňa všetky jej požiadavky a je bezpečný proti úniku alebo nie. Pripomeňme si nasledujúci príklad žiadosti o vrátenie daní cez internetovú aplikáciu. Používateľ si stiahne od predajcu program na vyplnenie formulára, pričom zadáva súkromné finančné informácie, ktoré by mali zostať tajné. Program môže tieto údaje odoslať priamo inštitúciám zodpovedným za danú problematiku, pričom sa použije šifrovanie na zabezpečenie diskrétnosti. Avšak program môže tiež odosielať späť predajcovi fakturačné údaje za použitie programu. Ako sa môže používateľ uistiť, že tieto fakturačné informácie neobsahujú žiadne súkromné finančné informácie používateľa.

Statická analýza by v tomto prípade dokázala označiť, či program dodržiava alebo nedodržiava politiku ochrany dôvernosti, a preto je možné, že došlo k nepriamemu úniku informácií. Systém je bezpečný z hľadiska toku informácií, ak vonkajší útočník nemôže získať informácie o utajených informáciách iba vďaka interakcii s daným systémom. V zásade programovacích jazykov sa teda zaoberáme problémom, ktorým je rozdelenie premenných (údajov) do skupín tajných a prístupných, kde by používateľ nemal prístup k tajným premenným a môže manipulovať s prístupnými, ako mu to umožňuje program. Cieľom je zabezpečiť, aby po manipulácii s prístupnými premennými nebolo možné zistiť nič o tajných (v ďalšej kapitole sú možné konkrétne príklady úniku tajných informácií iba vďaka manipulácii s prístupnými premennými). Politika ochrany dôvernosti, ktorú chceme skontrolovať, je formulovaná ako politika toku informácií a mechanismus na jej zaručenie. Tieto mechanismy sú v podobe ovládacích prvkov.

Táto myšlienka sa pôvodne vynára do 70. rokov minulého storočia, keď boli publikované prvé práce. Za predstaviteľov tejto myšlienky považujeme D. a

P. Denningovcov a ich prácu "Certifikácia programov pre bezpečný informačný tok". Od týchto skorých časov bola daná problematika dosť skúmaná, čo nasvedčuje jednému z najznámejších článkov zhrnujúcich tento výskum, ktorý napísali A. Sabelfeld a A. C. Myers v "Jazykový bezpečnostný informačný tok", citujúc okolo 160 ďalších prác.

V kapitole 2 sa bližšie vysvetľuje informačný tok, jeho vlastnosti, ukážky únikov a zadefinovaná noninterferencia. Ďalej sa práca venuje objektovo orientovanému pohľadu na túto problematiku. Nasleduje ukážka jedného z prvých zásadných riešení od Jens Palsberga a Michael I. Schwartzbacha, ktoré je postavené na teoretickom jazyku vychádzajúcom z jazyka Smaltalk. Ďalej nasleduje podrobnejší prehľad jazyka JFlow, ktorý bol predchodcom najväčšieho riešenia v tejto problematike a jazyka JIF. Ďalej sa práca okrajovo venuje existujúcim nástrojom, vyvinutým vo svete aj na Univerzite Komenského.

V kapitole 5 čitateľa oboznámi s potrebnou lexikálnou analýzou a nasledne sa čitateľ môže oboznámiť s implementáciou verifikátora tokov na jazyku Java, ktorý je funkčný nad rozumnou podmnožinou konštruktov tohto jazyka. Nasledne je na príkladoch overená jeho funkčnosť.

Kapitola 2

Analýza informačného toku

V tejto kapitole sa budeme venovať informačnému toku, jeho únikom, nebezpečným konštrukciám a vlastnostiam ktoré definujú jeho bezpečnosť. V neskoršej časti sa dostaneme k typovacím systémom, ich aplikácií a fungovaniu na nekonkurentných, neskôr na konkurentných programoch.
Ako sme si v úvode naznačili, to čo nám nemôže zabezpečiť inak bezpečný systém prístupových práv, je dôveryhodnosť programu. V tejto práci budeme často používať výraz dôvernosť ako vlastnosť nejakej entity. Dôvernosť sa v tomto kontexte vzťahuje k zabráneniu úniku (kompletných alebo len častí) informácií k neautorizovaným osobám alebo systémovým premenným. Podľa čoho vieme usúdiť kedy je program dôverný? Ako zistiť kedy nám unikajú informácie?
Odpovedí môže byť veľa a od konkrétnych prípadov sa môžu líšiť. Riešenie je však intuitívne, zavedieme si politiku dôvernosti, ktorá povie, čo sa môže, čo nie a kto je autorizovnaý na prácu s ktorými dátami. Aby sme teda mohli prehlásiť, že program si drží vlastnosť dôvernosti, je potrebné sa uistiť, že informácie v danom programe boli použité len v súlade s príslušnou politikou dôvernosti.

Či je politika dodržaná nám povie analýza toku informácií v danom programe. Politika dôvernosti, ktorú chceme presadiť je formovaná ako politika informačného toku a mechanizmy, ktoré ju zaručujú sú ovládacie prvky informačného toku (neskôr si ukážeme ako presne vyzerá). Analýza informačného toku pozostáva zo statickej analýzy zdrojového kódu programu ešte pred jeho vykonaním, ktorá overí legálnosť informačného toku na samotnom zdrojovom kóde programu a teda odsleduje či sa nedostávajú tajné alebo iné, vysoko chránené informácie, do menej chránených oblastí alebo sa nedajú iným spôsobom odvodiť. Nakoľko je z dôvodu komplexnosti moderných výpočtových systémov manuálna analýza nemožná, môže byť takáto analýza formulovaná ako typovací systém. Táto voľba má hneď niekoľko výhod:

- Typy môžu slúžiť ako formálne špecifikovaný jazyk, a tak poskytnúť automatickú verifikáciu kódu, napr. typovou inferenciou

- Navyše, pretože celá analýza sa dá vykonať počas kompilácie, nestojí nás nič počas runtimu

Typovací systém je teda realizácia politiky informačného toku (politiky dôvernosti) a jeho jednotlivé konštrukcie, sú ovládacie prvky informačného toku.
Štandardnou formalizáciou bezpečnosti informačného toku je noninterferencia. Program má vlastnosť noninferencie, pokiaľ pri rovnakých low level vstupoch produkuje vždy rovnaké low level výstupy, bez ohľadu na hodnoty high level premenných.

2.1 Základný princíp

Základom princípu analýzy informačného toku je rozdelenie programových premenných do rôznych úrovní bezpečnosti. Najnižšia úroveň je rozdelená

do dvoch tried - L (nízka bezpečnosť) a H (vysoká bezpečnosť). Cieľom je zabrániť pretečeniu alebo zverejneniu dát označených ako H do L. Princíp bezpečnostných levelov znamená, že informačný tok môže prejsť len na vyššej úrovni. Typový systém je lepšou alternatívou oproti manuálnej analýze, ktorá by pri súčasných zložitých programových riešeniach nebola možná. Výhodou typového systému je, že formálna špecifikácia môže byť súčasťou jazyka a kontrola kódu tak môže byť automatická. Najznámejším príkladom je implementácia jazyka JIF, ktorý rozširuje Java o analýzu informačného toku a kontrolu bezpečnostných levelov. Typový systém slúži ako nástroj na kontrolu dôvernosti a politík pomocou analýzy informačného toku.

Našim cieľom je zabrániť nevhodnému zverejneniu alebo pretečeniu H premenných. Foriem, takéhoto zverejnenia je mnoho, avšak určite potrebujeme zabrániť toku informácií z H premenných do L premenných. Typovací systém je teda taký systém, nástroj, algoritmus, ktorý rozširuje daný program/programovací jazyk o bezpečnostné typy vzhľadom na informačný tok a pomocou nich kontroluje prechod citlivých dát do menej citlivých oblastí, kde môže dôjsť k ich zverejneniu. Typovací systém má o to silnejšiu bezpečnostnú politiku, pokiaľ vie zamedziť aj nepriamy prúd informácií. V tomto prípade ide hlavne o možnosť odvodiť hodnotu high level premennej bez priameho priradenia a to na základe rôznych konštrukcií programu alebo logického vyvodenia premennej na základe znalosti správania sa systému na určitých častiach kódu. Príklady odvodeného informačného toku:

```
1. if h >= 0 then l := 1 else l := 3;
2. l := 0; while l < h do l := l + 1;
3. while h >= 0 do skip;
```

```
4. while h > 0 do h := (h − 1);
```

V príkladoch 1 a 2 vieme hodnotu premennej h zistiť alebo aspoň priblížiť aj bez priameho priradenia. V príklade 1 vieme podľa hodnoty l či je h kladné alebo záporné. V príklade 2 vieme zistiť presnú hodnotu h za predpokladu, že l je menšie ako h. Tajnú informáciu nám vie odhaliť aj ukončenie programu. Program 3 skončí, iba ak je hodnota h záporná. Ďalšie správanie, ktoré nám vie odhaliť nejaké informácie je dĺžka behu programu. V príklade 4 závisí dĺžka behu programu od iniciálnej hodnoty premennej h.

2.2 Noninterferencia

V tejto kapitole sa bližšie pozrieme na noninterferenciu. Program, ktorý dodržiava noninterferenciu, je bezpečný ohľadom na informačný tok. Noninterferencia je sformalizovaný model striktnej viacúrovňovej politiky, opísaný Goguenom a Meseguerom v roku 1982 a zosilnený v roku 1984 [GM82], ktorý zabezpečí, že program neprepustí informáciu z vyššieho bezpečnostného levelu do nižšieho. Máme program C, ktorý má svoje vstupy a výstupy klasifikované do bezpečnostných levelov L a H. Program C má vlastnosť noninterferencie, ak pri každom z behov tohto programu s rovnakými L vstupmi vracia rovnaké L výstupy, nech je hodnota vstupných H premenných akákoľvek. Takýmto spôsobom útočník, ktorý má prístup len k premenným typu L počiatočného a koncového stavu, sa daný program C správa plne deterministicky. Táto vlastnosť abstrahuje od podrobností spustenia programu, vývojovej platformy, jazyka a architektúry prostredia, v ktorom je program spúšťaný.

Všimnime si, že noninterferencia je splnená len pre deterministické prog-

ramy. Neskôr si rozoberieme noninterferenciu, ktorá bude platiť aj pre nedeterministické programy.

Samozrejme, zverejnenie H informácie L premenným nie je jediný spôsob ako môže informácia uniknúť. Majme nasledovný program:

```
while (secret != 0);
```

Pokiaľ je secret nenulové, program sa cyklí. Takže útočník, ktorý môže pozorovať ukončenie programu môže dedukovať hodnotu secret. Podobne, aj čas behu programu môže závisieť na H informáciach. Takýmto časovým únikom sa dá ťažko zabrániť, pretože môžu využívať nízkoúrovňové implementačné detaily, ktoré sa nedajú nijako ovplyvniť. Napríklad:

```
int i, count, xs[4096], ys[4096];
for (count = 0; count < 100000; count++) {
        if (secret != 0)
                for (i = 0; i < 4096; i += 2)
                        xs[i]++;
        else
                for (i = 0; i < 4096; i += 2)
                        ys[i]++;
        for (i = 0; i < 4096; i += 2)
                xs[i]++;
}
```

Na abstraktnej úrovni, keď sa pozrieme na tento kód, tak tam nevidíme žiadnu závislosť k hodnote secret. Ak však tento kód spustíme na lokálnom Sparc serveri s 16K data cache pamäťou, tak beží dvakrát tak dlho keď je

secret 0 ako keď je nenulové (keď je secret nenulové, pole xs ostáva v cache počas vykonávania programu, keď je secret 0, cache si drží xs, aj ys). Pretože vonkajšie pozorovanie behu programu tak veľmi sťažuje kontrolu informačného toku, väčšina práce na bezpečnosť informačného toku sa sústreďuje na zabránenie toku informácií z H do L, čo je zachytené nejakým typom noninterferencie.

2.3 Princípy typovania

V tejto časti si popíšeme, ako môžu byť typovacie systémy použité na zaistenie noninterferencie. Pre jednoduchosť predpokladajme len bezpečnostné úrovne H a L. Uvažujme jednoduchý imperatívny jazyk[17], ktorého abstraktná syntax je definovaná nasledovne:

```
(fr  zy)                 p ::= e | c
(v razy)          e ::= x | n | e1 op e2
(pr kazy)         c ::= x := e |
                        skip |
                        if e then c1 else c2 |
                        while e do c |
                        c1; c2
```

Metapremennú x používame pre identifikátory a n pre integrové literály. Hodnota premennej môže byť len integer, 0 sa používa pre f alse a nenulový integer pre true. Program c sa vykonáva pod pamäťou μ, ktorá mapuje identifikátory na hodnoty. Predpokladáme, že výrazy sú konečné a vykonávajú sa ihneď. Hodnotu výrazu e v pamäti μ značíme μ(e). Vykonanie príkazu je definované cez štrukturálnu operačnú sémantiku znázornenú nižšie. Jej pravidlá definujú

tranzičnú reláciu → na konfiguráciách. Konfigurácia je buď pár (c, μ) alebo iba pamäť μ. Konfigurácia (c, μ) vyjadruje, že sa akurát ide vykonať príkaz c. Konfigurácia μ značí len pamäť μ. Označenie → vyjadruje reflexívny tranzitívny uzáver (konečný prechod) →. Prechody a zmeny v konfiguráciách sú definované cez štrukturálnu operačnú sémantiku nasledovne:

```
(UPDATE)        x      dom(u)
                       (x := e, u)      u[x := u(e)]
(NO–OPT)        (skip,     )      u
(BRANCH)        u(e) = 0
                       (if e then c1 else c2, u)      (c1, u)

                       u(e) = 0
                       (if e then c1 else c2, u)      (c2, u)
(LOOP)          u(e) = 0
                       (while e do c, u)   u
                       u(e) = 0
                       (while e do c,     )   (c; while e do c, u)

(SEQUENCE)      (c1, u)      u
                       (c1; c2, u)      (c2, u0)
                       (c1, u)      (c1, u0)
                       (c1; c2, u)      (c1; c2, u0)
```

Obmedzenia na takomto jazyku, resp. čo je povolené, môžeme vyjadriť typovacím systémom[13]. Pre typovací systém majme nasledovné bezpečnostné typy dát a fráz:

(typy d t) t ::= L | H
(typy fr z) p ::= t | t var | t cmd

Intuícia je taká, že výraz e typu obsahuje len premenné úrovne a menej, ďalej, príkaz c typu cmd robí priradenia len do premenných úrovne alebo vyššej.
Potom majme identifikátor , ktorý mapuje každú premennú na typ t var, vyjadrujúci jej bezpečnostnú úroveň. Úsudok R = p : p, čítame "z R je dokázateľné, že p je typu p". Potom nech náš typovací systém obsahuje typovacie pravidlá:

(R–VAL) R(x) = t var
R = x : t

(INT) R = n : L

(PLUS) R = e1 : t , R = e2 : t
R = e1 + e2 : t

(ASSIGN) R(x) = t var , R = e :
R = x := e : t cmd

(SKIP) R skip : H cmd

(IF) R = e : t
R = c1 : t cmd
R = c2 : t cmd
R = if e then c1 else c2 : t cmd

(WHILE) R = e : t
R = c : t cmd
R = while e do c : cmd

(COMPOSE) R = c1 : t cmd
R = c2 : t cmd
R = c1 ; c2 : t cmd

a pravidlá pre podtypy, ktoré nám umožňujú pretypovať výrazy a príkazy, aby nám sedeli do predpokladov typovacích pravidiel. Pravidlá pre podtypy:

(BASE) L : H

(C M D) t : t
t cmd : t cmd0

(REFLEX) p : p

(TRANS) p1 : p2 , p2 : p3
p1 : p3

(SUBSUMP) R = p : p1 , p1 : p2
R = p : p2

Väčšina pravidiel je intuitívna. Pravidlo CMD hovorí, že príkaz môžeme podtypovať, keďže z definície vieme, že typ cmd robí priradenia do premenných úrovne alebo vyššej. Odvodiť nový typ nám umožňuje pravidlo SUBSUMP. Vďaka pravidlu SUBSUMP zase vieme nadtypovať výrazy, ktoré zase z definície obsahujú premenné danej úrovne a menej.

Tieto pravidlá zabezpečia, že pokiaľ je program validný tak sa na jeho konštrukcie budú dať aplikovať pravidlá, až dôjdeme k jednému výslednému typu. To bude typ pre celý program. Pokiaľ v nejakom stave nebudeme môcť aplikovať žiadne pravidlo, tak tam dochádza k leaku. Programy, ktoré sú dobre otypované týmto systémom určite spĺňajú noninterferenciu. Povieme si ako je to zabezpečené.

2.4 Konkurencia

V tejto časti sa budeme venovať bezpečnosti konkurentných programov. Ukážeme si ďalšie nebezpečné príklady v konkurentných programoch. Potom si zavedieme nové, ešte silnejšie typy noninterferencie a popíšeme si typovací systém pre konkurentné programy.

Predpokladajme, že náš jazyk rozšírime na viac threadov so zdieľanou pamäťou. Týmto zavádzame nedeterminizmus, ktorý robí definíciu noninterferencie nevhodnou - teraz spustenie programu dvakrát na tej istej pamäti môže vyprodukovať dve pamäte, ktoré sa nezhodnú na L premenných. Takto prichádzame k vlastnosti posibilistickej noninterferencie[18], ktorá hovorí, že zmenou iniciálnych hodnôt H premenných sa nemôže zmeniť množina možných konečných hodnôt L premenných.

Definícia 1

(Posibilistická noninterferencia). Program c spĺňa posibilistickú noninterferenciu, ak pre ľubovoľnú pamäť μ a pamäť , zhodujúce sa na L premenných, beh c na μ môže vyprodukovať konečnú pamäť μ, potom beh c na môže vyprodukovať pamäť takú, že μ a sa zhodujú na L premenných.

Majme daný nižšie uvedený kód s premennými inicializovanými na 0, mask na mocninu 2 a secret s ľubovoľnou hodnotou. Ďalej majme dobre otypované premenné: secret, trigger0, trigger1 typu H a leak, maintrigger, mask typu L. Spomínané typovacie pravidlá nepostačujú na zabezpečenie posibilistickej noninterferencie - vidíme, že pod férovým schedulerom program vždy skopíruje secret do leak.

```
Thread :
        while (mask != 0) {
                while (trigger0 == 0);
                leak = leak | mask; // bitwise    or
                trigger0 = 0;
                maintrigger = maintrigger+1;
                if (maintrigger == 1)
                        trigger1 = 1;
        }
Thread :
        while (mask != 0) {
                while (trigger1 == 0);
                leak = leak & ~mask; // bitwise    and
                                        with complement of mask
                trigger1 = 0;
                maintrigger = maintrigger+1;
                if (maintrigger == 1)
                        trigger0 = 1;
        }
```

```
Thread     :
          while (mask != 0) {
                    maintrigger = 0;
                    if (secret & mask == 0)
                              trigger0 = 1;
                    else
                              trigger1 = 1;
                    while (maintrigger != 2);
                    mask = mask/2;
}
          trigger0 = 1;
          trigger1 = 1;
```

Takže musíme pridať dodatočné obmedzenia na multithreadové programy. Najskôr si však uzrejmime požiadavky na scheduler. Posibilistická noninterferencia je splnená, len ak predpokladáme čisto nedeterministický scheduler, ktorý si v každom kroku môže vybrať ľubovoľný thread, ktorý bude aktuálne bežať. V takomto modeli nie je žiadny vzťah s pamäťou, ktorá je výsledkom behu programu.

No skutočný scheduler je oveľa viac predvídateľný (ROUND ROBIN, FIFO). Napríklad, scheduler si môže hodiť mincou v každom kroku, aby si vybral ktorý thread pobeží. Pod takýmto pravdepodobnostným schedulerom je posibilistická noninterferencia ako bezpečnostná vlastnosť nedostačujúca! Majme nasledovný program s premennými secret : H, leak : L, kde secret môže nadobúdať hodnotu 1 až 100 a rand vracia náhodný integer od 1 do 100:

```
Thread A:
leak := secret
```

```
Thread B:
leak := rand(100)
```

Tento program spĺňa posibilistickú noninterferenciu, pretože konečná hodnota premennej leak môže byť od 1 do 100 bez ohľadu na iniciálnu hodnotu premennej secret. Program ale nebude bezpečný. Keď si ho pustíme viackrát tak môžeme dostať postupnosť leakov napr:

75, 22, 12, 22, 22, 93, 4, 22, ...

z čoho môžeme ľahko usúdiť, že hodnota secret je 22. Dá sa to vysvetliť nasledovne:
Keďže každý thread má rovnakú pravdepodobnosť pustenia, tak dostávame pravdepodobnosť, že leak bude obsahovať hodnotu premennej secret 101/200 a že tam bude hocijaké iné číslo od 1 do 100 je 1/200. Teda si vieme byť istí, že keď viackrát pustíme program a vyberieme najčastejšiu hodnotu, dostaneme hodnotu secret. Z čoho nám vyplýva, že napriek splneniu posibilistickej noninterferencie, je daný program nebezpečný. Tento príklad nás motivuje pre ešte silnejšiu bezpečnostnú vlastnosť, pravdepodobnostnú noninterferenciu[19], ktorá hovorí, že iniciálna hodnota H premenných nemôže ovplyvniť distribúciu vzájomnej pravdepodobnosti na konečných hodnotách L premenných. Potom by tento program nespĺňal pravdepodobnostnú noninterferenciu, pretože zmeny iniciálnej hodnoty secret by menili distribúciu konečnej hodnoty leak.
Takže môžeme vidieť, že vhodná formulácia noninterferencie závisí od typu jazyka, ktorý uvažujeme. Vo všetkých prípadoch je idea taká, že konečná hodnota L premenných je nezávislá od vstupnej hodnoty H premenných. Pre deterministický jazyk to znamená, že iniciálna hodnota H premenných ne-

môže zmeniť konečnú hodnotu L premenných. Pre nedeterministický jazyk to znamená, že zmena iniciálnych hodnôt H premenných nemôže zmeniť množinu možných konečných hodnôt L premenných. A pre pravdepodonostný jazyk to znamená, že zmena iniciálnych hodnôt H premenných sa nemôže zmeniť distribúcia možných konečných hodnôt L premenných.
Pre splnenie pravdepodobnostnej noninterferencie bol navrhnutý systém v D. Volpano a G. Smith[19] a obdobný v G. Boudol a I. Castellani[3], ktorý sa dá zhrnúť do nasledovných obmedzení:

1. Výraz e je H, ak obsahuje nejakú H premennú
2. Len L výrazy môžu byť priradené do L premenných
3. Strážený príkaz s H strážou sa nemôže priradiť do L premenných (podmienený príkaz s H v podmienke)
4. Stráž while cyklu musí byť L
5. If podmienka s H strážou musí byť chránená (aby sa vykonala automaticky, bez akejkoľvek závislosti) a nesmie obsahovať žiadne while cykly vo vnútri tela

Obmedzenia 2 a 3 zabraňujú priamemu a nepriamemu toku. V jazyku bez konkurencie aj úspešne spĺňajú noninterferenciu. Obmedzenia 4, 5 boli zavedené na zabránenie časovo založeným tokom(v závislosti od použitej architektúry a dĺžky behu niektorých operácií sa vonkajším pozorovaním dajú odhaliť určité vlastnosti systému). Tento systém nepovolí vyššie spomínaný program s threadmi , , , keďže trigger0 a trigger1 sú H. Tento systém však nanešťastie vo veľkom obmedzuje aj množinu povolených programov. Nový, lepší systém bol prezentovaný Smithom[15]. Tento systém povoľuje strážam while cyklov obsahovať H premennú, ale na zamedzenie časovým tokom, po-

užíva inú metodiku. Daný systém, si bližšie popíšeme v ďalšej časti.

2.4.1 Typovací systém pre konkurentné programy

Ako sme si vyššie spomenuli, pre kontrolu toku v multithreadovom programe je tiež dôležitá otázka časových leakov, a teda do akých premenných príkaz priraďuje a ako dlho beží. Nasledujúci typovací systém pre konkurentné programy[15] sa uberá práve týmto smerom. Teraz si ho zadefinujeme a bližšie si ukážeme ako funguje. Tento typovací systém je založený na nasledujúcich typoch:

(typy d t) t ::= L | H
(typy fr z) p ::= t | t var | t1 cmd t2 | t cmd n

Nové typy príkazov majú nasledovnú intuíciu:
• Príkaz c je klasifikovaný ako 1 cmd 2, ak priraďuje len do premenných typu 1 alebo viac a jeho čas behu závisí od premenných typu 2 alebo menej.
• Príkaz c je klasifikovaný ako 1 cmd n, ak priraďuje len do premenných typu 1 alebo viac a je garantované, že skončí presne po n krokoch.
Ďalej máme nasledovné typovacie pravidlá pre multithreadové programy:

(R–VAL) R(x) = t var
R= x : t
(INT) R = n : L
(PLUS) R = e1 : t, R = e2 : t
R = e1 + e2 : t
(ASSIGN)R(x) = t var, R= e : t

R = x := e : t cmd 1

(SKIP) R skip : H cmd 1

(IF) R = e : t

R = c1 : t cmd n

R = c2 : t cmd n

R = if e then c1 else c2 : t cmd n + 1

R = e : t1

t1 t2

R = c1 : 2 cmd 3

R = c2 : 2 cmd 3

R = if e then c1 else c2 : t2 cmd t1 t3

(WHILE) R = e : t1

t1 t2

t3 t2

R = c : t2 cmd t3

R = while e do c : t2 cmd t1 t3

(COMPOSE) R = c1 : t cmd m

R = c2 : t cmd n

R = c1; c2 : t cmd m + n

R = c1 : t1 cmd 2

t2 t3

R = c2 : t3 cmd 4

R = c1; c2 : t1 t3 cmd t2 4

(PROTECT) R = c : t1 cmd t2

c contains no while loops

R = protect c : t1 cmd 1

Pre menej intuitívne pravidlá si teraz povieme, čo tieto pravidlá vyjadrujú a

prečo sú takto navrhnuté. Pravidlo IF má pre otypovanie príkazu if e then c1 else c2 dva varianty. Prvý hovorí, že ak výraz e zo stráže je úrovne a príkazy c1 a c2 sú úrovne , ktoré sa oba vykonajú v n krokoch, vieme odvodiť, že cieľový príkaz je úrovne , ktorý sa vykoná v n+1 krokoch (jeden zvolený blok + vyhodnotenie stráže). Druhý variant tiež požaduje rovnakú úroveň príkazov 2, avšak vykonanie neprebehne v presnom počte krokov, ale závisí od úrovne 3. Ďalej tento variant vyžaduje, aby 1 2, čo nám zabezpečí, že zo stráže vyššej úrovne nám nepretečie informácia do tela nižšej úrovne. Takto vieme odvodiť, že cieľový príkaz je úrovne 2 a jeho vykonanie závisí od vyššej z úrovní t1 a t3 (keďže beh podľa definície závisí od danej úrovne alebo nižšej). V pravidle využívame operáciu , ktorá značí hornú hranicu. Pravidlo WHILE pre cieľový príkaz while e do c požaduje, aby pre výraz e : 1 a príkaz c : 2 cmd 3 platilo 1 2, čo zabraňuje toku vyššej úrovne zo stráže do tela nižšej úrovne a 3 2, zabraňuje vzniku iteračného cyklu priraďujúceho do premennej nižšej úrovne, ktorého beh závisí od vyššej úrovne. Potom náš výsledný príkaz má úroveň 2 a jeho beh závisí od väčšieho z dvojice 1 a 3. Pravidlo COMPOSE umožňuje zložiť dva sekvenčne nasledujúce príkazy do jedného. Má dva varianty. Prvý nám umožňuje zložiť príkazy úrovne , jeden bežiaci m krokov a druhý n krokov na jeden príkaz úrovne bežiaci m + n krokov. Druhý variant vyžaduje príkaz c1 s typom 1 cmd 2 a c2 s typom 3 cmd 4. Splnením podmienky 2 3 je zabezpečené, že po príkaze, ktorého beh závisí od vyššej úrovne nebude nasledovať príkaz nižšej úrovne. Nebezpečenstvo takejto konštrukcie si vysvetlíme po výklade pravidiel. Takto nám potom vznikne zložený príkaz c1; c2 úrovne nižšej z dvojice 1 a 3 (značíme operáciou), ktorého beh závisí od vyššieho z dvojice 2 a 4. Ďalej máme pravidlá pre podtypy v multithreadových programoch, ktoré nám umožňujú pretypovať výrazy a príkazy, aby sme mohli na ne aplikovať vyššie uvedené

pravidlá. Pravidlá pre podtypy:

(BASE) L : H

(CMD) t1 : t1, t2 : t2

t1 cmd t2 : t1

cmd t2

t : t

t cmd n : t

cmd n

t cmd n : t cmd L

(REFLEX) p : p

(TRANS) p1 : p2, p2 : p3

p1 : p3

(SUBSUMP) R = p: p1, p1 : p2

R = p : p2

Pravidlo CMD má 3 varianty, prvé dva hovoria, že príkaz môžeme podtypovať (podobne ako pri prvom typovacom systéme) a prvý ešte hovorí, že závislosť behu môžeme nadtypovať. Keďže príkaz sa nikdy nenachádza v stráži (a teda z neho nepotečie informácia nižšie), tak podtypovaním nič nepokazíme. Tretie pravidlo hovorí, že konštantný počet krokov vykonávania programu vieme pretypovať na L závislosť.

Tento multithreadový systém na rozdiel od predchádzajúceho povoľuje while cyklu obsahovať H premenné, ale na zamedzenie časovým tokom vyžaduje, aby za príkazom, ktorého dĺžka behu závisí od H premenných, sekvenčne nenasledovalo priradenie do L premennej. Intuícia je taká, že takéto priradenie do L premennej je v multithreadovom programe nebezpečné, lebo za predpokladu, že nejaký iný thread tiež priraďuje do tej istej L premennej, poradie,

v ktorom sa tieto priradenia vykonajú, a teda aj konečná hodnota tejto L premennej závisí na H informácii. Teraz, keď už vieme ako typovať program, poďme si ukázať zopár otypovaných príkazov a overiť na nich, či sú validné. Majme x : H a y : L:

```
1. x := 0 : H cmd 1
2. y := 0 : L cmd 1
3. if x := 0 then x := 5 else skip : H cmd 2
4. while y = 0 do skip : H cmd L
5. while y = 0 do y := y    1 : L cmd L
6. y := 5; while x + 1 do skip : L cmd H
7. (while x = 0 do skip); y := 5 : illegal
```

Posledný príklad je nelegálny, pretože čas behu while cyklu závisí od H premennej x a pokiaľ neplatí H L, nemôže za týmto príkazom nasledovať príkaz priraďujúci do L premennej. Tento príklad je nebezpečný, lebo ľubovoľný iný thread sa môže dozvedieť či x je 0 jednoduchým spôsobom: Chvíľu počká, aby dal šancu scheduleru pospúšťať všetky thready, a potom sa iba pozrie či je hodnota y rovná 5. Pri typovaní je ešte dôležité spomenúť, že existuje Agatova transformácia, ktorá nám pomáha upraviť nevalidný program. Majme premenné x : H, y : L a nasledovný program:

```
if x = 0 then
        x := x * x; x := x * x;
else
        x := x + 1;
y := 0;
```

Tento program bude vyhlásený ako nebezpečný, pretože if vetva je typu H

cmd 2 a else vetva je typu H cmd 1, čo nám bráni aplikovať prvý variant pravidla IF a teda musíme vetvy pretypovať na H cmd L a aplikovať druhý variant IF, čo nám dáva H cmd H. Nasledujúci príkaz y := 0; typu L cmd 1 je potom podľa pravidla COMPOSE nevalidný, keďže H 6je podmnožina L. Aby sme program spravili validným, doplníme do else vetvy príkaz skip, čo nám dáva typ else vetvy H cmd 2 a umožňuje nám použiť prvý variant IF pravidla, čo nám dáva typ IF príkazu H cmd 3 a po použití pravidla COMPOSE dostávame typ pre program L cmd 4.

Kľúčovou myšlienkou dôkazu splnenia pravdepodobnej noninterferencie je, že ak máme dobre otypovaný thread c a pustíme ho pod dvoma Lekvivalentnými pamäťami, potom oba tieto behy urobia presne tie isté priradenia do L premenných v rovnakých časoch. Zaručením, že máme takúto vlastnosť, potom vieme ukázať, že takéto multithreadové programy spĺňajú pravdepodobnostnú noninterferenciu. Dôkaz si ešte vyžaduje zavedenie slabej pravdepodobnostnej bisimulácie a pre aktuálne účely je príliš rozsiahly.

Kapitola 3

Analýza existujúcich riešení

3.1 JIF

3.2 SPARK Examiner

Kapitola 4

Lexikálna analýza

4.1 ANTLR

Kapitola 5

Implementácia riešenia

V tejto kapitole sa budeme venovať implementácii typovacieho systému. Konkrétne rozšírime jazyk o typovacie prvky a implementujeme verifikátor toku, ktorý bude kontrolovať splnenie pravidiel typovacieho systému pre programy v takomto jazyku. Verifikátor bude podporovať 2 kontroly informačného toku. Kontrolu toku pre obyčajný, deterministický program (zhora-nadol) a kontrolu konkurentného programu (zdola-nahor), kde využijeme prvky zo Smithovho multithreadového systému z kapitoly 1.2, ktorý sme si ukazovali na jednoduchom abstraktnom jazyku a otestujeme si v praxi ako dokážu fungovať na reálnom jazyku. Ďalej bude verifikátor vedieť dodefinovať rozmedzie možných anotácií, v prípade chýbajúcich anotácií. Typovací systém sme sa rozhodli implementovať pre jazyk Java, keďže pre tento jazyk ešte nie je implementovaná kontrola toku na viacvláknových systémoch. V ďalších častiach si popíšeme si postup tvorby nášho riešenia, znázorníme si UML diagram a na ukážkach si overíme funkcionalitu verifikátora toku.

5.1 Programovací jazyk

5.2 Postup

5.3 Testovanie

Kapitola 6

Záver

Literatúra

[Alp11] Rafael H. Alpizar. Secure information flow via stripping and fast simulation. *Florida International University, Miami, Florida, USA*, 2011.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[Bor07] Andrew Bortz. Exposing private information by timing web applications. *World Wide Web Conference Committee, Banff, Alberta, Canada*, 2007.

[FS00] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. *Secure Internet Programming Laboratory,Department of Computer Science,Princeton University,Princeton, NJ 08544 USA*, 2000.

[Goe15] Tom Van Goethem. The clock is still ticking: Timing attacks in the modern web. *Department of Computer Science, Stony Brook University, Minds-Distrinet, KU Leuven, 3001 Leuven, Belgium*, 2015.

[Mye99] Andrew C. Myers. Jflow: Practical mostly-static information flow control. *Proceedings of the 26th ACM Symposium on Principles of*

Programming Languages (POPL '99), San Antonio, Texas, USA, 1999.

Zoznam obrázkov