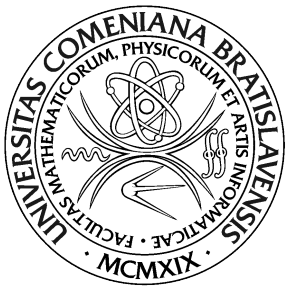
UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



FORMÁLNE METÓDY A BEZPEČNOSŤ

Diplomová práca

2020 Bc. Lukáš Blaha

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

FORMÁLNE METÓDY A BEZPEČNOSŤ

Diplomová práca

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

Bratislava, 2020 Bc. Lukáš Blaha

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Lukáš Blaha
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.9. aplikovaná informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský

Názov: Formálne metódy a bezpečnosť

Cieľ: Cieľom práce je výskum v oblasti využitia formálnych metód pre bezpečnosť počítačových systémov. Práca môže byť teoretická alebo teoreticko implementačná.

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

Dátum zadania: 26.10.2016

Dátum schválenia: 26.10.2016

doc. 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. bla bla

. . . . . .

Bratislava, 2020 Bc. Lukáš Blaha

Poďakovanie

..poďakovanie

Abstrakt

BLAHA, Lukáš: Formálne metódy a bezpečnosť [Diplomová práca]. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky. Katedra aplikovanej informatiky. Vedúci diplomovej práce: doc. RNDr. Damas Gruska, PhD. Bratislava, 2020, rozsah: strán,

Táto diplomová práca sa zameriava na problematiku dôveryhodnosti programov, keď konštrukcie v zdrojovom kóde programu môžu spôsobiť únik dôverných informácií. Riešením sú typovacie systémy s vhodne nastavenou politikou dôvernosti. Práca obsahuje príklady rôznych únikov dôverných informácií, popisuje vlastnosti programov zabezpeèujúcúce ich dôvernosť pre rôzne typy programov, zrozumiteľne oboznamuje čitateľa s teóriou typovacích systémov a ukazuje ako spĺňajú tieto vlastnosti. Pracá delej popisuje návrh a typovacieho systému pre jazyk Java. Rozoberá funkcionalitu existujúcich implementácií typovacích systémov. Venuje sa aj lexikálnej analýze potrebnej na spracovanie zdrojových kódov a následne opisuje implementáciu programu na verifikáciu infomačného toku a na príkladoch overuje jeho funkčnosť.

Kľúčové slová: informačný tok, typovací systém, noninterferencia, lexikálna analýza

Abstract

..abstract

Keywords:

Obsah

Kapitola 1

Úvod

Kapitola 2

Analýza informačného toku

Dnes žijeme, v rýchlej dobe, dobe internetu, dobe chytrých telefónov, dobe v ktorej sa okolo nás šíri nezastaviteľný prúd informácií. Dáta sú pre bežných ľudí dosiahnuteľnejšie ako nikdy predtým. Vďaka tejto ľahkej dostupnosti dát však nastáva čoraz viac problémov s únikmi informácií, ktoré by mali ostať utajené. Tento fenomén je možne vidieť takmer na dennom poriadku, pre ľudí sú ale najciteľnejšie úniky ich vlastných dát, informácií, ktoré poskytli webovým stránkam, ktoré ich ale nezvládli uchovať v bezpečí. Jedna z najznámejších a najpoužívanejších sociálnych sieti Facebook, čelí neustálym útokom zo strany hackerov, ktorý sa snažia neoprávnene získať dáta o jej používateľoch za účelom finančného profitu. Webovej aplikácií sa nie vždy úspešne darí brániť a získava za to veľmi negatívnu reputáciu v masmediálnom svete, ako aj medzi používateľmi. Ako by sa teda malo postupovať pri návrhu používateľských aplikácií, aby podobné riziká boli čo najviac minimalizované? Existujú rôzne pohľady na túto problematiku. Známy prístup je napríklad systém bezpečnostných práv, ten však nedokáže úplne zabezpečiť to že informácia, aj keď nepriamo dokáže preniknúť na miesta odkiaľ môže byť sponzorovaná útočníkom, ktorý na základe hierarchie práv, by ne-

mal k takejto informácií mať prístup. Ako jedným z možných riešení, tohto problému sa ponúka statická analýza informačného toku.

Na úvod je potrebné zadefinovať si akúsi politiku dôvernosti, na základe ktorej budeme jednoznačne vedieť povedať, že či prišlo k možnému úniku dôveryhodných informácií, alebo nie. Teda touto politikou dôvernosti určíme kto ma práva narábať s akými informáciami a aké operácie je možné, respektíve nie je možne vykonať nad danou informáciou aby nedošlo k porušeniu tejto politiky. Analýza informačného toku bude nástroj na overenie či boli dodržané konkrétna politika dôvernosti a na základe jej výstupu, môžeme povedať či program spĺňa všetky jej podmienky a teda je bezpečný voči únikom, alebo nie je. Zvážme nasledujúci príklad vyplnenia žiadosti vratky daní, pomocou internetovej aplikácie. Používateľ si stiahne do svojho počítača program od predajcu, pomocou ktorého vyplní formulár pre vrátenie dane, pričom zadáva súkromné finančné informácie, ktoré by mali zostať utajené. Tento program môže odosielať údaje priamo inštitúciám zodpovedným za túto problematiku elektronicky, pričom pre zabezpečenie diskrétnosti sa použije šifrovanie. Program však ďalej môže odosielať naspäť predajcovi fakturačné údaje za použitie programu. Ako si môže používateľ byť istý, že tieto fakturačné informácie nebudú skryto obsahovať nejaké súkromné finančné informácie používateľa.

Statická analýza by v takomto prípade dokázala označiť či program spĺňa, alebo nespĺňa danú politiku dôvernosti a teda je možné že došlo prípadne aj k skrytému (nepriamemu) úniku informácií. Systém je teda bezpečný z hľadiska toku informácií, ak vonkajší útočník nemôže nadobudnúť znalosť o utajených informáciách, len vďaka interakcií s daným systémom. V princípe programovacích jazykov sa teda pozeráme na problém, tak že premenne (dáta) rozdelíme do množín tajných, ku ktorým by používateľ nemal mať prístup a

prístupných premenných s ktorými môže ľubovoľne manipulovať tak ako mu to program dovoľuje. Cieľom je zabezpečiť aby po ľubovoľnej manipulácií s prístupnými premennými nebolo možné zistiť nič o tých utajených (v ďalšej kapitole je možné nájsť konkrétne príklady uniknutia utajených informácií len vďaka manipulácií s dostupnými premennými) Politika dôvernosti ktorej dodržanie chceme skontrolovať je sformovaná ako politika informačného toku a mechanizmus na jej zaručenie. Tieto mechanizmy sú vo forme ovládacích prvkov.

Táto myšlienka pôvodne siaha pôvodne až do rokov sedemdesiatych minulého storočia, kedy boli publikované prvé práce. Za priekupníkov tejto myšlienky považujeme D. a P. Denningovích a ich prácu Certification of programs fot secure information flow. Následne už od týchto skorých časov bola daná problematika dosť skúmaná, čomu nasvedčuje aj jeden s najznámejších článkov zhrňujúcich tento výskum ktorý napísali A. Sabelfeld a A. C. Myers – Language-based information-flow security, ktorý cituje okolo stopäťdesiat ďalších prác.

Pre upresnenie ešte uvádzam rozdiel medzi dátovým dokom a informačným tokom. Informačný tok sa používa na verifikáciu prenosov informácií medzi programami alebo premennými, tak aby neboli porušené bezpečnostné podmienky. Analyzuje sa tok medzi premennými rozloženými do viac bezpečnostných levelov (L – low, H -high) a zabranuje implicitmým a explicitným tokom z H do L. Pri dátovom toku v oblasti počítačov záleží od od kontextu, najčastejšie sa ale spomína pri práci s audio – videom, prípadne konektivitou, kde udáva zvyčajne rýchlosť alebo kapacitu zapisovania dát, najčastejšie vo Mbit/s.

2.1 Základný princíp

Základný princíp analýzy informačného toku sa opiera o rozdelenie programových premenných do rozdielnych bezpečnostných levelov. Na najnižšej úrovni hovoríme o rozdelení do dvoch tried, ktoré klasifikujeme ako L - nízky bezpečnostný level (z anglického Low) a H - vysoký bezpečnostný level (z anglického High). Ďalej sa snažíme aby nedošlo k zverejneniu alebo pretečeniu dát označených ako H do L. Abstraktnejšie teda môžeme hovoriť o tzv. "mriežke" bezpečnostných levelov, v ktorých je možné dôjsť k informačnému toku len na hor tejto mriežky. Napríklad ak platí, že $L \leq H$, tak dovolený len tok z L do L, z H do H a z L do H. Toku z levelu H do L sa snažíme zabrániť.

Ďalší z hlavných bodov o ktorý sa opiera analýza informačného toku je typovací systém. Je to lepšie riešenie ako použite manuálnej analýzy, ktorá by pri dnešných zložitých programových riešeniach v podstate nebola možná. Výhody typovacieho systému spočívajú v tom že jeho formálna špecifikácia môže byť súčasťou jazyka, vďaka čomu môže byť verifikácia kódu automatická. Najznámejším riešením tohto druhu je implementácia jazyka JIF. Je to programovací jazyk, ktorý rozširuje Javu o analýzu a kontrolu toku, pomocou už vyššie zmieneného rozdelenia premenných na bezpečnostné levely. K tomuto jazyku sa ale dostaneme skôr. Typovací systém je teda nástroj na kontrolu a politiky dôvernosti, pomocou analýzy informačného toku.

Základné rozdelenie zakázaných tokov je na:

- Explicitný
- Implicitný

Príklad explicitného zakázaného toku:

```
int {public} l;
int {secret} h;
...
l = h;
```

Príklad implicitného zakázaného toku (keď dáta pretečú nepriamo):

```
int {public} x;
boolean {secret} b;
...
if(b){
        x = 1;
}
```

Ako je možné vidieť z tohto príkladu, vyčítať hodnotu premennej b, ktorá je označená ako secret, dokážeme vyčítať aj bez toho aby sme do nej robili nejaké priradenia. Kvôli nevhodne zvolenej podmienke, dokážeme jej hodnotu odhadnúť len z výstupu premennej x, ku ktorej máme prístup.

Ďalšie príklady na nebezpečný imlplicitný tok:

1. Pri takejto konštrukcií vieme dokonca vyčísliť presnú hodnotu premennej h.

```
int {public} l;
int {secret} h;
...
while (l < h){
        l = l + 1;
}
```

2. V tomto príklade môžeme vidieť, že nie je ani nutné pracovať s premennými ktoré majú bezpečnostný level L a aj napriek tomu môže dôjsť k pretečeniu H informácie, ktoré vieme odhadnúť na základe ukončenia programu. Program skončí iba ak je hodnota menšia ako nula.

```
int {secret} h;
...
while(h >= 0){
        skip;
}
```

3. V tomto príklade môžeme vidieť, že nie je ani nutné pracovať s premennými ktoré majú bezpečnostný level L a aj napriek tomu môže dôjsť k pretečeniu H informácie, ktoré vieme odhadnúť na základe ukončenia programu. Program skončí iba ak je hodnota menšia ako nula.

```
int {secret} h;
...
while(h >= 0){
        h = (h - 1);
}
```

4. Odhadnutie tajnej hodnoty, je možne aj na základe dĺžky behu programu. V tomto príklade, nám dĺžka závisí od počiatočnej hodnoty premennej h.

```
int {secret} h;
...
while(h >= 0){
        h = (h - 1);
```

```
}
```

5. V nasledujúcom príklade, už vieme odhadnúť len paritu tajnej premennej, teda či je jej hodnota kladná, alebo záporná. Stále sa však jedná o nebezpečný informačný tok.

```
int {public} l;
int {secret} h;
...
if(h >= 0){
        l = 1;
}else{
        l = 3;
}
```

6. Polia tiež môžu viesť k menšiemu pretečeniu. Ak pole A bolo na začiatku celé vynulované, potom táto konštrukcia vedie k prezradeniu hodnoty v premennej h, ktorá je bezpečnostný level secret.

```
int [] {public} a;
int [] {public} l;
int {secret} h;
...
a[h] = 1;
for(int i = 0; i < a.length; i++){
        if(a[i] == 1){
                l = i
        }
}
```

2.2 Noniterferencia

V tejto kapitole sa bližšie pozrieme na noninterferenciu. Program, ktorý dodrží noninterferenciu, je bezpečný vzhľadom na informačný tok.

Noninterfernica je sformalizovaný model striktnej viacúrovňovej politiky, popísaný Goguenom a Meseguerom v roku 1982 a zosilnená v roku 1984, ktorý zabezpečí, že program neprepustí informáciu z vyššieho bezpečnostného levelu do nižšieho.

Majme program C, ktorý má svoje vstupy a výstupy klasifkované do bezpečnostných levelov L a H. Program C má vlastnosť noninterferencie ak pri každom z n behov tohto programu s rovnakými L vstupmi vracia rovnaké L výstupy, nech je hodnota vstupných H premenných akákoľvek. Inak povedané, pre útočníka, ktorý ma prístup len ku premenným typu L začiatočného a koncového stavu, sa daný program C správa plne deterministicky.

Táto bezpečnostná vlastnosť kladie dôraz na stav daného programu pred jeho spustením a po spustení, abstrahuje teda detailov spustenia programu, jazyka a vývojovej platformy a architektúre prostredia v ktorom je program spúšťaný. Formálna definícia noninterferencie znie nasledovne:

Definícia 1 (Noninterferencia) *Dva stavy programu $P : \sigma, \sigma'$ sú nerozlíšiteľné vzhľadom na low level premenné, zapisujeme $\sigma \sim \sigma'$, ak $\sigma(x) = \sigma'(x)$ pre všetky $x \in L$.*

Program C je bezpečný vzhľadom na informačný tok, ak $\sigma \sim \sigma'$ a $\sigma \to \tau$ a $\sigma' \to \tau'$ tak aj $\tau \to \tau'$. 0 ! 0 tak aj 0.

Majme $\sigma = \{l \to 2, h \to 5\}$ a $\sigma' = \{l \to 2, h \to 7\}$. Potom platí $\sigma \sim \sigma'$ avšak po vykonaní príkazu

```
l = h;
```

je hodnota l vo výsledných stavoch τ a τ' rôzna. Preto platí $\tau \nsim \tau'$. V podstate ľubovoľné priradenie

```
l = e ;
```

kde e je výraz nie je bezpečné, pokiaľ hodnota e závisí od h, ako napríklad $l = h + 5$ alebo $l = l * h$, avšak nie $l = h - h$. Povšimnutia hodné je, že noninterfernica je splnená iba pre deterministické programy.

Vzhľadom k tomu, že vonkajšie pozorovanie behu programu robí kontrolu informačného toku veľmi zložitou, väčšina práce na zabezpečení informačného toku smeruje na zabránenie toku z H do L pomocou určitého typu noninterferencie

2.3 Typovací systém

Ešte pred zadefinovaním typovacieho jazyku, je vhodné zadefinovať si jednoduchý imperatívny jazyk. Kvôli zjednodušeniu budeme uvažovať iba s dvoma bezpečnostnými úrovňami H a L.

Imperatívny jazyk pozostáva z:

- Prázdny príkaz
- Priradenie do premennej
- Sekvenčná kompozícia
- While cyklus
- Vetvenie

2.4 Noninterferencia

Abstraktná syntax imperatívneho jazyka:

(frázy) $p ::= e|c$

(výrazy) $e ::= x|n|e_1 ope_2$

(príkazy) $p ::= x := e|$

skip |

if e **then** c_1 **else** c_2 |

while e **do** c |

$c_1; c_2$

Metapremenná x sa používa pre identifkátory a n pre literály z množiny celých čísel. V typovaciom jazyku je dovolené používať len celočíslené hodnoty typu integer. Na reprezentáciu boolienovksých hodnôt *true* a $false$ sa používa 0 respektíve nenulové celé číslo. Program c je vykonaný pod pamäťou μ, ktorá mapuje identifikátory na hodnoty. Predpokladajme, že výrazy sú konečné a vykonávajú sa automaticky. Zápis $\mu(e)$ označuje hodnotu výrazu e v pamäti.

Vykonávanie príkazov je definované v štrukturálnej operačnej sémantike, ktorá je definovaná nižšie. Pravidlá v nej definujú tranzitnú reláciu $\rightarrow$ na konfiguráciách. Konfigurácia je chápaná buď ako pár $(c; \mu)$ alebo iba pamäť μ. V prvom prípade $(c; \mu)$ označuje, že sa aktuálne ide vykonať príkaz c. V druhom prípade sa už príkaz vykonal a vyprodukoval pamäť. Zápis $\rightarrow *$ označuje reflexívny tranzitívny uzáver (konečný prechod). Prechody a zmeny v konfiguráciách sú definované nasledovne:

(UPDATE) $\frac{x \in dom(\mu)}{(x{:=}e,\mu) \to \mu[x{:=}\mu(e)]}$

(NO-OPT) $(\ \textbf{skip}\ ,\mu) \to \mu$

(BRANCH) $\frac{\mu(e) \neq 0}{(\ \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2,\mu) \to (c_1,\mu)}$

$\frac{\mu(e) = 0}{(\ \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2,\mu) \to (c_2,\mu)}$

(LOOP) $\frac{\mu(e) = 0}{(\ \textbf{while}\ e\ \textbf{do}\ c,\mu) \to \mu}$

$\frac{\mu(e) \neq 0}{(\ \textbf{while}\ e\ \textbf{do}\ c,\mu) \to (\ \textbf{while}\ e\ \textbf{do}\ )c,\ mu)}$

(SEQUENCE) $\frac{(c_1;\mu) \to \mu'}{(c_1;c_2,\mu) \to (c_2,\mu')}$

$\frac{(c_1;\mu) \to (c_1',\mu')}{(c_1;c_2,\mu) \to (c_1',c_2,\mu')}$

Kapitola 3

Analýza informačného toku z pohľadu OOP

3.1 Základné poroblémy

3.2 Typová inferencia

3.3 Abstraktná interpretácia

3.4 Abstraktná noninterferencia

3.5 Abstraktná noninterferencia orientovaná na triedy

Kapitola 4

Lexikálna analýza

4.1 Parser generátory

4.2 ANTLR

Kapitola 5

Existujúce riešenia riešenia

5.1 JIF

JIF je language-based implementácia analýzy informaèného toku. Je formovaný ako programovací jazyk, ktorý rozširuje Javu o kontrolu informačného toku. JIF kontroluje informačný tok aj počas kompilácie (staticky), aj počas runtimu (dynamicky).

Statická Kontrola Každý výraz má bezpečnostný typ zložený z dvoch častí: bežný dátový typ ako int a label/anotáciu, ktorá popisuje ako môže byť hodnota použitá. Kompilátor sleduje politiku prúdenia informácií zaznačenú anotáciami v danom programe a po akceptovaní daného programu (tzn. nenájde žiadne bezpečnostné chyby) preloží kompilátor program v JIFe na program v Jave, tzv. source-to-source kompilácia.

Dynamická Kontrola JIF má síce statické typy/anotácie, avšak využíva napríklad OpenFile metódu s dynamickým typom, ktorá v runtime vracia

handle s typom závisiacim od security typu aktuálne otvoreného sú-

boru. Preto napíklad JIFovská implementácia metódy append nemôže byť overená staticky kompilátorom.

JIF v podstate rozširuje Javu pridaním anotácií s bezpečnostnými typmi (označované ako "principal"), ktoré hovoria ako sa môžu dané informácie používať. V praxi sú principals používatelia, skupiny, atď.

```
int {Alice->Bob} x = 20;
```

Táto konštrukcia rozširuje základnú javovskú konštrukciu, ktorá hovorí, že x je typu integer a pridáva ďalšiu informáciu o tom, že informáciu v premennej x vlastní principal Alice a túto informáciu môže čítať principal Bob.

```
int {Alice->Bob, Chuck} y = 35;
x = y; // OK: policy on x is stronger
y = x; // BAD: policy on y is not as strong as x
```

V tejto časti kódu dochádza k bezpeènostnej chybe. Na začiatku deklarujeme, že informáciu v y vlastní principal Alice a čítať ju môže aj Bob, aj Chuck. Teda priradenie x = y; je v poriadku keďže y sa dostane z oblasti kde ju môže čítať Bob, aj Chuck do oblasti kde ju môže čítať len Chuck. Avšak druhé priradenie dovoľuje zverejniť informáciu, ktorú môže čítať len Bob aj Chuckovi. Tuto nastala bezpeènostná chyba a JIF kompilátor nám povie:

```
Main.jif:28: Label of right hand side not less
restrictive than the label for local variable y
y = x;
^
1 error.
```

JIF podporuje aj obmedzenia pre zápis. Táto anotácia hovorí, že w môže zapisovať do o.

```
o<-w
1 error.
```

JIF podporuje polymor

zmus a má konštrukcie aj pre deklasi

káciu:

```
int{Alice->;Alice<-*} b;
int{Alice->Bob;Alice<-*} y = 0;
if (b) {
        // pc is at level {Alice->;Alice<-*}
        declassify ({Alice->;Alice<-*} to {y}) {
                // at this point,
                // pc has been declassified
                // to the label of the local
                // variable y (that is,
                // {Alice->Bob;Alice<-*} )
                // permitting the assignment to y
                y = 1;
        }
}
```

V príklade vyššie vidno, že z premennej b nemôže čítať nikto. Premenná y je závislá od premennej b a i keď do nej priraďujeme konštantu 1 JIF kompilátor by nám vyhlásil chybu. Avšak priradenie je obalené do deklasi

kovaného úseku, v ktorom sa úroveň anotácie ktorej je aj b kastuje na anotáciu premennej y, čo povoµuje priradenie do y a zabezpečuje, že daný

program ostáva bezpečný.

5.2 SPARK Examiner

SPARK je formálne defnovaný programovací jazyk, ktorý je založený na programovacom jazyku Ada. Je určený na vývoj softvéru s vysokou integritou a v systémoch, kde je nutná predvídateľnosť a vysoká spoľahlivosť operácií. Využíva sa napríklad na vývoj softvéru pre jadrové elektrárne, lietadlá, vesmírne lode a lekárske systémy. Jazyk SPARK je zložený z obmedzenej, dobre definovanej podmnožiny jazyka Ada. Používa anotované metainformácie na popisovanie želaného správania komponentu a individuálnych požiadaviek počas behu progamu. Anotácie sa zapisujú do špeciálnych komentárov(#). Čiže aj keď sa nevykoná analýza, tak je program stále valídnym kódom v jazyku Ada.

SPARK Examiner je nástroj, ktorý je zabudovaný priamo v IDE a vykonáva dva druhy statickej analýzy:

kontrola správnej formulácie a konzistentnosti programu s návrhom zahrnutým v anotáciách.

voliteľná verifikácia. Vykoná sa mechanizmom dôkazu a ukáže, že program má určité špecifické vlastnosti. Najpriamočiarejším dôkazom je, že program je bez výnimiek. Može byť použitý aj na demonštráciu, že po dosiahnutí bezpčnostných vlastností sa ich program drží.

V programe napísanom v SPARKu sa ako prvé definujú bezpečnostn éúrovne:

```
package Classify is
--Security levels
```

```
        UNCLASSIFIED : constant := 0;
        RESTRICTED : constant := 1;
        CONFIDENTIAL : constant := 2;
        SECRET : constant := 3;
        TOPSECRET : constant := 4;
end Classify;
```

Potom označíme premenné bezpečnostnou úrovňou:

```
--# inherit Classify;
package KeyStore
--# own SymmetricKey(Integrity => Classify.SECRET);
--# RotorValue(Integrity => Classify.RESTRICTED);
is
procedure Rotate;
        --# global in RotorValue;
        --# in out SymmetricKey;
        --# derives SymmetricKey from
        --# SymmetricKey, RotorValue;
         ...
end KeyStore;
```

Informačný tok je deklarovaný explicitne pomocou anotácie derives:

```
procedure Rotate;
        --# derives SymmetricKey from
        --# SymmetricKey, RotorValue;
```

Hodnota SymetricKey je závislá od hodnoty SymetricKey a RotorValue. SPARK Examiner na rozdiel od iných riešení nevyžaduje nový programovací

jazyk. Jeho kód pozostáva len zo špeciálnych komentárov, vďaka ktorým je možný jeho vývoj nezávisle od jazyka. Hlavným zámerom je bezpečnosť.

5.3 Flow Caml

5.4 Flow

5.5 Information flow checker

Cieľovým jazykom pre veri

kovanie je jazyk C. Jazyk C je rozšírený o konštrukcie toku, čo umožnuje anotovať premenné v kóde a na základe nich kontrolovať politiku toku. Verifi

kátor poskytuje dva typy kontroly: klasickú kontrolu pre deterministický program a kontrolu pre konkurentný program, čo žiadny nástroj z vyššie spomínaných nedokáže. Na kontrolu konkurentných programov využíva teóriu zo Smithovho systému. Verifi

kátor potom načíta a zverfii

kuje kód a vyhlási či je program v poriadku alebo je chybný, s popisom chyby (keďže v konkuretných programoch máme viac obmedzení ako len priame a nepriame pretečenie informácie). Veri

kátor ďalej má podporu pre deklasifi

káciu a teda poskytne programátorovi možnosť na vlastnú zodpovednosť povoliť nebezpeèný tok s cieľom použiť potrebný príkaz. Okrem toho vie verifi

kátor vygenerovať anotácie pre program, ktorý nie je úplne oanotovaný.

5.6 JIFc

Kapitola 6

Implementácia

6.1 Postup

6.2 Obmedzenia

6.3 Case studies

Kapitola 7

Používateľská príručka

7.1 Nasadenie na aplikačný server

7.2 Administrácia

Kapitola 8

Záver

Zoznam obrázkov