VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

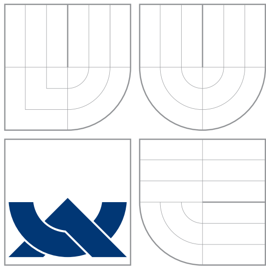

# ANALYZÁTOR KÓDU JAZYKA C

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

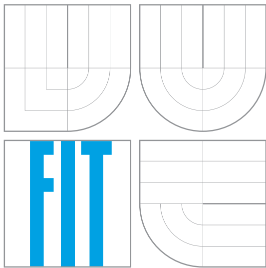

AUTOR PRÁCE DANIEL OVŠONKA
AUTHOR



BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY




FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# ANALYZÁTOR KÓDU JAZYKA C

CODE ANALYZER FOR C LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DANIEL OVŠONKA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. BORIS PROCHÁZKA

BRNO 2011

# Abstrakt

Tato bakalářská práce se zaobírá principy exploitovaní programů a detekci potencionálně zranitelných míst v programech. Tato detekce umožní vytváření bezpečnějších programových konstrukcí. V úvodu je čitatel obeznámen se základy programovacího jazyka C, jazyka Assembler a překládače GCC. Taktéž je uveden do problematiky exploitačných technik jako přetečení paměti, přetečení v segmentu haldy a BSS, přetečení čísel a formátovací řetězce. Dále je popsán samotný návrh, implementace a výsledky vytvořené aplikace.



# Abstract

This thesis deals with the principles of program exploitation and detection of potential vulnerabilities in the programs. This detection system offers to create safer program structures. At the beginning of the work the reader is familiarized with the basics of C programming language, assembly language and the GCC compiler. He is also introduced into the matter of exploitation techniques such as buffer overflow, heap overflow, BSS overflow, format string exploits and integer overflow. This work also describes the concept, implementation and results generated by application.



# Klíčová slova

Exploitace, Přetečení paměti, Jazyk Assembler, Jazyk C, Bezpečnost, Analyzátor kódu, Překladač GCC

# Keywords

Exploitation, Buffer overflow, Assembly language, C language, Security, Code analyzer, GCC compiler


# Citace


Daniel Ovšonka: Analyzátor kódu jazyka C, bakalářská práce, Brno, FIT VUT v Brně, 2011

# Analyzátor kódu jazyka C

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Borise Procházky

.......................
Daniel Ovšonka
17. května 2011

## Poděkování

Chtěl bych poděkovat panu Ing. Borisovi Procházkovi za pomoc a rady při řešení této bakalářské práce.

© Daniel Ovšonka, 2011.
*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Obsah

**Úvod** **3**

**1 Základné pojmy** **4**
1.1 Jazyk Assembler . . . . . 4
1.1.1 Základné registre procesora 80386 . . . . . 4
1.1.2 Práca so zásobníkom . . . . . 6
1.2 Jazyk C . . . . . 7
1.2.1 Segmentácia pamäti v jazyku C . . . . . 7
1.2.2 Práca s haldou . . . . . 8
1.2.3 Zásobník a zásobníkový rámec jazyka C . . . . . 9
1.2.4 Prekladač GCC . . . . . 10
1.2.5 Zhrnutie . . . . . 10

**2 Popis exploitačných techník** **11**
2.1 Pretečenie pamäti (Buffer Overflow) . . . . . 11
2.1.1 Pretečenie bufferu založenom na zásobníku (Stack Buffer Overflow) . 12
2.1.2 Pretečenie bufferu v segmente haldy (Heap Overflow) . . . . . 12
2.1.3 Pretečenie v segmente BSS . . . . . 12
2.2 Formátovacie reťazce . . . . . 13
2.3 Pretečenie čísel (Integer Overflow) . . . . . 13
2.4 Zhrnutie . . . . . 14

**3 Návrh a implementácia** **15**
3.1 Statická analýza . . . . . 15
3.2 Štruktúra programu . . . . . 15
3.3 Tabuľka funkcií . . . . . 16
3.3.1 Implementácia tabuľky funkcií . . . . . 16
3.4 Lexikálny analyzátor . . . . . 18
3.4.1 Flex . . . . . 19
3.4.2 Definičný súbor pre Flex . . . . . 20
3.4.3 Trieda lexicalToken . . . . . 20
3.5 Syntakticko-sémantický analyzátor . . . . . 21
3.5.1 Bison . . . . . 21
3.5.2 Definičný súbor pre Bison . . . . . 22
3.5.3 Činnosť syntakticko-sémantického analyzátora . . . . . 23
3.6 Bezpečnostný analyzátor . . . . . 24
3.6.1 Princíp priebehu odhaľovania problémov . . . . . 24
3.6.2 Práca s funkciami . . . . . 25

3.6.3 Práca s premennými . . . 27
3.6.4 Práca s neznámymi hodnotami . . . 27
3.6.5 Posun ukazatelov . . . 28
3.7 Modul výstupu . . . 28
3.8 Ukážka činnosti analyzátora . . . 28
3.9 Zhrnutie . . . 29

**4 Testovanie** **31**
4.1 Testovanie na vlastnej sade programov . . . 31
4.1.1 Pretečenie pamäti . . . 31
4.1.2 Formátovacie reťazce . . . 33
4.1.3 Pretečenie čísel . . . 33
4.1.4 Komplexný test . . . 33
4.1.5 Zhrnutie . . . 34
4.2 Testovanie na programoch tretích strán . . . 35
4.2.1 Obmedzenia zistené pri testovaní . . . 35
4.2.2 Možnosti ďalšieho vývoja . . . 36
4.2.3 Prostredie pri testovaní . . . 36

**Záver** **37**

**Seznam příloh** **39**

**A Návod na použitie** **40**

**B Obsah CD** **41**

**C Demonštračné programy** **42**

**D Metriky kódu** **54**

# Úvod

Úlohou nasledujúcej práce je ozrejmiť čitateľovi princípy fungovania programov napísaných v jazyku C a následné útoky, ktoré je na tieto programy možné použiť. Na výklad týchto princípov bolo nutné uviesť potrebné základy jazyka Assembler, uvedené v podkapitole 1.1 a taktiež spomenúť dôležité fakty o jazyku C, nachádzajúce sa v podkapitole 1.2. Vzhľadom na to, že programy bývajú prekladané pomocou kompilátora GCC, ktorý umožňuje odhaliť určité základné chyby v programoch, sa nachádza v kapitole 1.2.4 popis princípu fungovania tohto prekladača a taktiež ozrejmené využitie tvoreného analyzátora kódu pri procese vývoja programov.

Druhá kapitola obsahuje detailnejšie informácie o typoch útokov na programy. Spoločnou vlastnosťou pre zraniteľnosť programov proti všetkým spomenutým útokom je chyba, prípadne nepozornosť programátora. Proti týmto chybám sa bude snažiť bojovať analyzátor kódu, ich odhalením a uľahčením programátorovi ich opravu. Druhá kapitola je rozdelená do podkapitol, kde podkapitola 2.1 sa venuje pretečeniam v rôznych segmentoch pamäti. Podkapitola 2.2 opisuje útoky založené na formátovacích reťazcoch a podkapitola 2.3 sa zaoberá pretečením čísel. Hlavnou úlohou tejto kapitoly je ozrejmiť čitateľovi princípy spomenutých útokov, na ktoré sa analyzátor špecializuje, aby bol výklad návrhu samotnej detekcie zrozumiteľnejší.

Tretia kapitola je už zameraná na praktickejšie aspekty analyzátora kódu, konkrétne na samotný návrh a implementáciu jeho najpodstatnejších častí. Kapitola je ako v predošlých prípadoch členená do podkapitol na základe jednotlivých modulov programu. Čitateľ sa tu môže dozvedieť informácie o ukladaní dát programu 3.3.1, práci lexikálneho 3.4, syntakticko-sémantického 3.5 a nakoniec aj bezpečnostného analyzátora 3.6. Záver tejto kapitoly ponúka pohľad na celkovú štruktúru navrhnutého programu a náčrt interakcie medzi jeho modulmi.

Posledná štvrtá kapitola sa venuje testovaniu implementovanej aplikácie. Testovanie je dôležitou súčasťou pri vývoji programov. Samotné testovanie bolo rozdelené na dve fázy. V prvej časti vývoja bol analyzátor testovaný na sade jednoduchých modelových príkladov obsahujúcich chybové konštrukcie 4.1. Popis tejto sady programov tvorí čiastočný prierez množiny rozpoznávaných chybových konštrukcií v programoch. Druhú časť kapitoly tvorí fáza testov na programoch tretích strán 4.2, ktoré by mali overiť schopnosti analyzátora pri práci v reálnom prostredí. Na základe týchto testov sú následne odvodené určité obmedzenia a načrtnutý možný ďalší vývoj aplikácie. Po prečítaní tejto kapitoly by si mal čitateľ utvoriť obraz o vlastnostiach navrhnutého analyzátora a jeho využití.

Záver sa venuje zhrnutiu poznatkov publikovaných v tejto práci a zhodnoteniu dosiahnutých výsledkov.

Na vytvorenie tejto práce boli použité rôzne publikácie, ktoré sú uvedené v zodpovedajúcich kapitolách.

# Kapitola 1

# Základné pojmy

V tejto kapitole bude čitateľ uvedený do problematiky a budú zopakované základné pojmy, na ktoré sa bude v nasledujúcich kapitolách odvolávať a tieto pojmy budú ďalej rozvíjané. V práci sa zameriame na architektúru *Intel x86* a operačný systém *Linux*, na ktorej budú všetky problémy opisované. Prostredie je potrebné presne definovať, pretože pre pochopenie problematiky bude potrebné ozrejmiť základy jazyka *Assembler*, ktorý obsahuje odlišnosti pre rôzne architektúry, taktiež bude potrebné definovať základy jazyka *C* a prekladača *GCC* pre túto platformu.

Kapitola je rozdelená na tri časti. V prvej časti sa zameriame na jazyk Assembler, základnú prácu s registrami a zásobníkom. V druhej časti budú popísané základy jazyka C ako napríklad segmentácia pamäti programu a podobne. Niektoré časti sa budú prelínať a dopĺňať prvú časť. V poslednej časti kapitoly ozrejmíme funkciu prekladača GCC na našej architektúre.

## 1.1 Jazyk Assembler

Jazyk Assembler je programovací jazyk nižšej úrovne. Programovanie v tomto jazyku je značne špecifické, pretože ho nemôžeme oddeliť od fyzickej architektúry počítača a vo väčšine prípadov ani od operačného systému, v ktorom plánujeme pracovať. Ako už bolo spomenuté, zameriame sa na architektúru Intel x86, s ktorou sú v súčasnosti kompatibilné takmer všetky procesory na trhu.

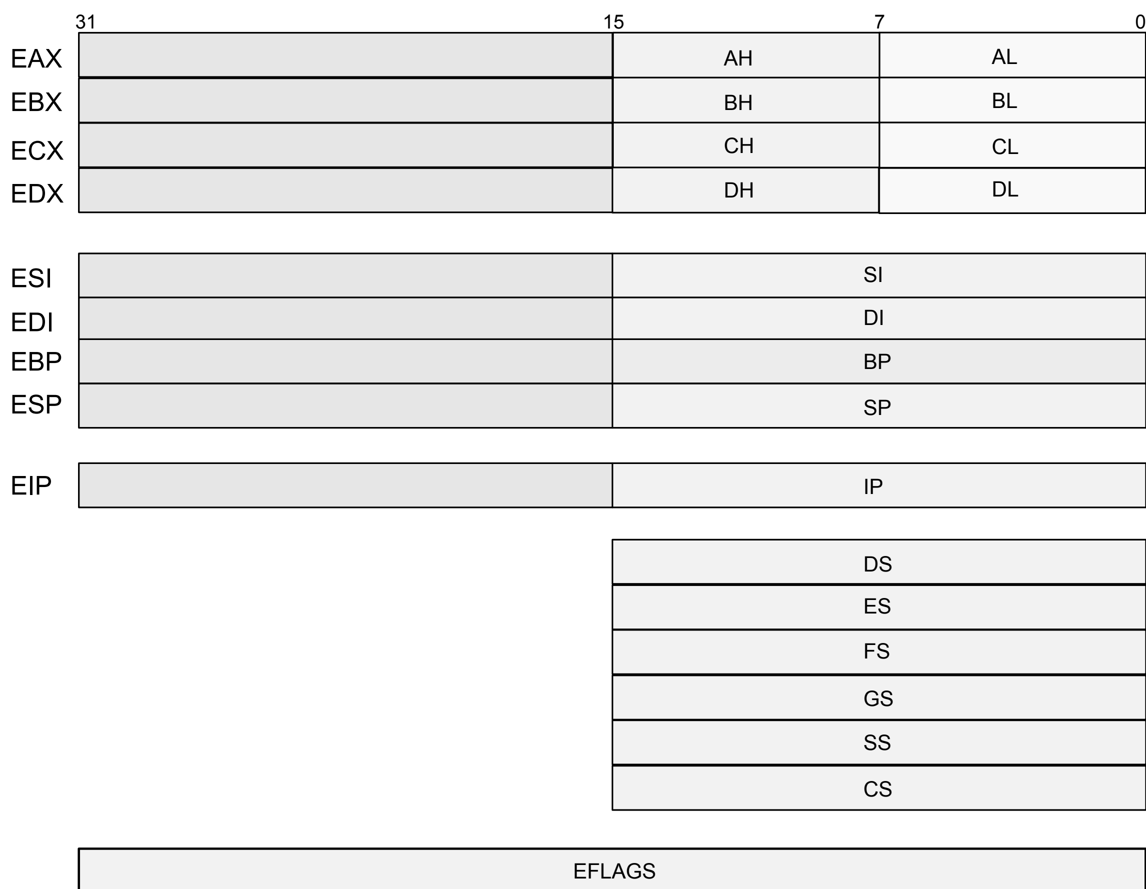
Aby sme mohli naplno pracovať s jazykom Assembler, musíme poznať registre procesora pre našu architektúru a samozrejme inštrukcie procesora. Keďže inštrukcií procesora je značné množstvo a nie sú pre našu problematiku úplne podstatné, nebudú rozoberané, avšak je možné ich dohľadať napríklad v [8], odkiaľ boli čerpané informácie aj pre ostatné informácie tejto podkapitoly, ktoré boli dopĺňané aj z [6].

### 1.1.1 Základné registre procesora 80386

Pri popise registrov budeme vychádzať z mikroprocesora *80386*, ktorý je 32 bitový a obsahuje tiež 32 bitovú zbernicu. To znamená, že môže pracovať s $2^{32}$ bytov pamäti. Z tejto architektúry vychádzali neskôr ďalšie procesory a väčšina moderných procesorov je s architektúrou x86 spätne kompatibilná.

Najdôležitejšie registre pre našu problematiku sú zobrazené na obrázku 1.1, ktorý si ďalej popíšeme.

Prvé štyri registre (`EAX`,`EBX`,`ECX` a `EDX`) sa nazývajú univerzálne registre. Sú využívané na rozličné účely. Napríklad ako dočasné premenné pre samotný procesor pri vykonávaní




Obrázek 1.1: Vybrané registre procesora 80386

strojového kódu. V poradí, v akom boli vymenované, sa nazývajú *Accumulator*, *Base*, *Counter* a *Data*, čo čiastočne predpokladá, na čo by mali byť používane. Každý z týchto registrov je rozdelený na dve 16 bitové časti, ktoré je možné adresovať samostatne. Tieto 16 bitové časti sú ešte rozdelené na dve 8 bitové časti, ktoré je možné taktiež adresovať samostatne. Tieto podmnožiny sú zavedené do týchto registrov kvôli spätnej kompatibilite so staršími 16 a 8 bitovými procesormi a taktiež sa dá pomocou týchto registrov šetriť pamäťové miesto.

Ďalšie štyri registre (`ESI`, `EDI`, `EBP`, `ESP`), ktoré je možné používať, sa nazývajú ukazatele, avšak ako v predchádzajúcom prípade, sa jedná o univerzálne registre. Ukazatele sa nazývajú preto, lebo sa často používajú na adresovanie pamäti ako napríklad prístup a indexovanie polí. Tieto registre je možné adresovať v 16 alebo 32 bitovom režime. Posledné dve písmená skratky reprezentujú názov registra. Prvé dva vymenované *Source Index* a *Destination Index* sa bežne používajú ako ukazatele na zdroj respektíve cieľ čítania prípadne zapisovania dát. Druhé dva spomenuté *Base Pointer* a *Stack Pointer* sú používané ako ukazatele do pamäte, kde adresujú objekt a sú dôležité pre správu pamäti programu a samotné vykonávanie strojového kódu. Podrobnejšie použitie registra `ESP` bude uvedené v nasledujúcej podkapitole.

Ako ďalší register spomenieme register `EIP`, ktorý sa nazýva *Instruction Pointer* a slúži ako ukazatel na inštrukciu v kódovom segmente pamäti, ktorá sa bude vykonávať v nasle-

dujúcom cykle procesora. Hodnotu tohto registra nie je možné meniť priamo, ale o jej zmenu sa stará sám mikroprocesor. Hodnota sa buď iba inkrementuje alebo sa mení podľa adries, na ktoré ukazujú skokové inštrukcie, prípadne inštrukcie volajúce podprogramy. Pomocou rôznych útokov, ktoré budú spomenuté v nasledujúcich kapitolách, je možné prepašovať do tohto registra adresu vlastného kódu útočníka, ktorý sa bude vykonávať a prevezme kontrolu nad vykonávaným programom.

Register príznakov `EFLAGS` je register obsahujúci jednobitové hodnoty príznakov, ktoré určujú najmä stav aritmeticko - logickej jednoty. Tento register nie je pre našu potrebu potrebné opisovať podrobne.

Ako posledné spomenieme segmentové registre, ktoré slúžia na výpočet skutočnej adresy, ktorá bude vystavená na dátovú zbernicu. Z týchto spomenieme register `SS` – *Stack Segmet*, ktorý spoločne s registrom `ESP` určuje adresu vrcholu zásobníka, ktorá je reprezentovaná touto dvojicou `SS:ESP`. A taktiež `CS` – *Code Segment*, ktorý spoločne s registrom `EIP` (`CS:EIP`) určuje celkovú adresu inštrukcie, ktorá sa bude vykonávať v nasledujúcom cykle mikroprocesora. Ostatné segmentové registre `DS` – *Data Segment*, `ES` – *Extra Segment* a `FS`, `GS` (*F* a *G* segment) sú používané pri adresovaní pamäti.

### 1.1.2 Práca so zásobníkom

Zásobník je dátová štruktúra typu *LIFO* (Last In - First Out), ktorá slúži na dočasné ukladanie hodnôt registrov, prípadne premenných do pamäte a po určitom čase je možné tieto hodnoty zo zásobníka vybrať. Zásobník taktiež slúži na predávanie parametrov medzi hlavným programom a podprogramami.

Zásobník v architektúre x86 nie je na procesore implementovaný ako hardwarový prvok, ale má vyhradené miesto v hlavnej pamäti – znázornené na obrázku číslo 1.2. Zásobník obyčajne začína na najvyšších adresách pamäti a s pridávaním ďalších údajov rastie smerom dole. To znamená, že dáta, ktoré boli uložené ako posledné, majú nižšiu adresu ako dáta na dne zásobníka.

Základnú prácu zo zásobníkom zabezpečujú dve inštrukcie jazyka Assembler:

- `PUSH op` – inštrukcia `PUSH` zabezpečuje vloženie operandu predaného ako jediný parameter inštrukcie. Vloženie na zásobník prebieha v dvoch krokoch. Najskôr sú dáta uložené na adresu pamäti určenú dvojicou `SP:ESP` a následne je hodnota ukazatela na zásobník dekrementovaná o veľkosť vložených dát v bajtoch, ktorá môže byť 2 (16 bitov) alebo 4 (32 bitov) bajty.

- `POP op` – inštrukcia `POP` slúži na vybratie dát zo zásobníka a ich uloženie do operandu, ktorý je predaný inštrukcii ako jediný parameter. Inštrukcia predstavuje opak k inštrukcii `PUSH`. Výber prebieha taktiež v dvoch krokoch, kedy sa najskôr nakopírujú dáta z adresy vrcholu zásobníka a následne dôjde k inkrementácii registra ESP o veľkosť vybraných dát v bajtoch, ktorá môže byť 2 (16 bitov) alebo 4 (32 bitov) bajty.

Jazyk Assembler obsahuje aj ďalšie inštrukcie na prácu zo zásobníkom, napríklad `PUSHA`, `POPA` a podobne, ktoré sú len odvodené od už spomenutých základných inštrukcií s tým, že majú definovaný operand už priamo v názve a neprijímajú žiaden parameter.

Obsah zásobníka sa mení aj pri volaní podprogramu inštrukciou `CALL`, kedy sa na zásobník uloží hodnota registrov `EIP`, prípadne aj `CS`, aby sa zachoval bod, odkiaľ sa do pod-

programu skákalo. Návrat z podprogramu zabezpečuje inštrukcia `RET`, ktorá obnoví obsah registra `EIP`, prípadne `RETF`, ktorá obnovuje zo zásobníka aj hodnotu registra `CS`.

Obdobná situácia nastáva aj v prípade obsluhy prerušenia, kedy sú na zásobník uložené hodnoty registrov `EIP`, `CS` a registera príznakov. Návrat je zabezpečený inštrukciou `IRET`, ktorá spomenuté registre obnoví zo zásobníka.

Táto podkapitola popísala základy jazyka Assembler pre architektúru x86. Tieto vedomosti budú ďalej rozvinuté v princípoch jazyka C nachádzajúcich sa v nasledujúcej kapitole.

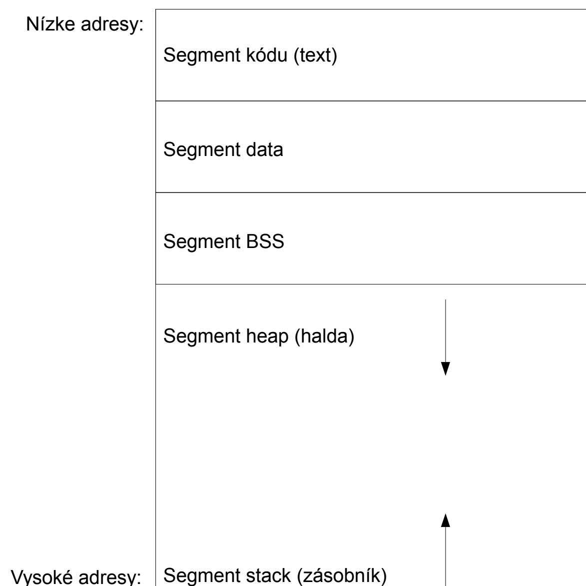
## 1.2 Jazyk C

Jazyk C je jazyk, ktorý sa ešte považuje za jazyk nižšej úrovne, avšak oproti jazyku Assembler je považovaný za jazyk vyššej úrovne. Je to obecne používaný jazyk, ktorý je možné používať aj pri systémovom programovaní. Jazyk C je otvorený jazyk rozšíriteľný pomocou knižníc. Na tento jazyk existuje množstvo rôznych prekladačov pre rôzne architektúry, a preto sa dá za určitých podmienok považovať kód v jazyku C za prenositeľný.

V tejto kapitole sa nezameriame na základné konštrukcie a princípy programovania v jazyku C. Tieto je možno dohľadať v [7], odkiaľ boli čerpané aj niektoré ďalšie informácie v tejto kapitole dopĺňané z [6] a [8]. Tieto základné programátorské techniky nie sú na ozrejmenie exploitačných techník až tak podstatné. Táto kapitola bude zameraná hlavne na prácu s pamäťou a jej segmentáciu.

### 1.2.1 Segmentácia pamäti v jazyku C

Pamäť skompilovaného kódu programu sa segmentuje do piatich segmentov. Sú to *text*, *dáta* (data), *bss*, *halda* (heap) a *zásobník* (stack). Jednotlivé segmenty reprezentujú určitú časť pamäti, ktorá ma v programe špecifický účel. Toto usporiadanie je zobrazené na obrázku 1.2.




Obrázek 1.2: Segmentácia pamäti v jazyku C

Jednotlivé časti si popíšeme trochu podrobnejšie:

- **Segment text** – často označovaný aj ako segment kódu (*code segment*). Do tejto časti pamäte je nahraný samotný preložený strojový kód programu. Do tohto segmentu ukazuje ukazatel `EIP`. Pri štarte programu je `EIP` nastavený na začiatok tohto segmentu. Vykonávanie, ako bolo spomenuté v predchádzajúcej podkapitole, nie je lineárne, pretože je ovplyvnené rôznymi skokovými inštrukciami a volaniami podprogramov a prerušení. Tento segment je špecifický aj tým, že do neho nie je povolený zápis a má pevnú veľkosť.
- **Segment data** – tento segment je určený na uloženie globálnych a statických premenných, ktoré sú inicializované v čase prekladu programu. Premenné v tomto segmente majú konštantnú veľkosť a v programe sú dostupné neustále bez ohľadu na kontext v programe. Oba dosiaľ spomenuté segmenty, sa nachádzajú hneď v binárnom súbore, ktorý vytvorí kompilátor.
- **Segment bss** – je obdoba dátového segmentu, avšak s tým rozdielom, že sú sem ukladané globálne a statické premenné, ktoré neboli v čase prekladu inicializované.
- **Segment heap** – je pamäťový segment premenlivej veľkosti, ktorý je riadený priamo programátorom. Veľkosť segmentu nie je pevná a mení sa pomocou alokačných a dealokačných funkcií jazyka. Pri zväčšovaní haldy narastajú adresy smerom nadol, to znamená, že adresy sa inkrementujú.
- **Segment stack** – taktiež dynamický segment pamäti, slúži na dočasné ukladanie lokálnych premenných funkcií a kontextu počas vykonávania programu. Nemá presnú statickú veľkosť. Na vrchol zásobníka ukazuje register `ESP`, ktorý sa pri vkladaní a vyberaní dát do zásobníka neustále mení. Začiatok zásobníka sa nachádza na vysokých adresách pamäti a rastie smerom nahor, to znamená, že s pribúdajúcimi dátami sa adresy zmenšujú. Spoločne so segmentom heap sa nenachádza tento segment v spustiteľnom binárnom súbore, ale do pamäti sú zavedené až pri štarte programu. Špecifickou prácou zo zásobníkom je uloženie kontextu pri volaní funkcií pomocou uloženia zásobníkového rámca (*stack frame*). Tento princíp bude opísaný v nasledujúcej podkapitole 1.2.3.

V tejto podkapitole boli opísané segmenty pamäti programu v jazyku C, tieto vedomosti ďalej ozrejmíme v nasledujúcich podkapitolách.

### 1.2.2 Práca s haldou

Pri ostatných spomenutých segmentoch, okrem haldy, je manažment pamäti pod správou prekladača a sám programátor sa nemusí o nič starať. Avšak pri halde je správa pamäti v réžii samotného programátora. Ten určuje, koľko miesta si bude program alokovať a aj presné množstvo pamäte potrebné v určitom okamihu vykonávania programu. Pamäť na halde je z programu najčastejšie dostupná pomocou ukazatelov. Táto správa sa uskutočnuje pomocou alokačných funkcii, kam patrí napríklad `malloc()` a dealokačných funkcií ako `free()`.

Funkcia `malloc()` prijíma ako svoj jediný parameter veľkosť, ktorú chceme na halde alokovať. Ako návratovú hodnotu vracia v prípade úspechu ukazatel na začiatok alokovaného miesta a v prípade neúspechu vracia hodnotu `NULL`.

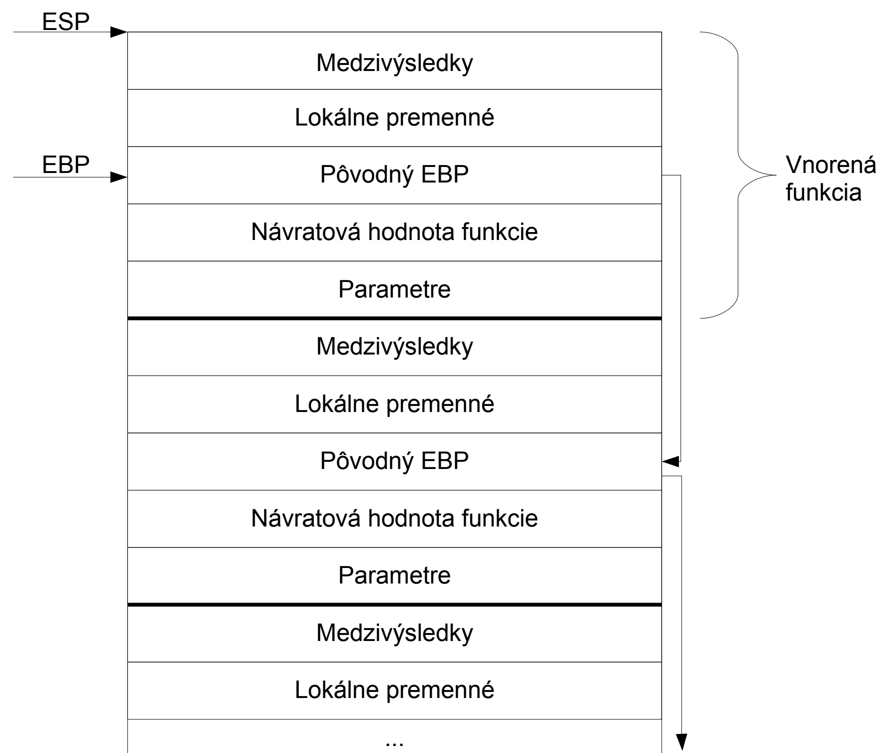
Na druhej strane funkcia `free()`, prijíma ako parameter ukazatel na pamäťové miesto, ktoré chceme uvoľniť a toto miesto bude možné neskôr opätovne použiť pri alokácii nových zdrojov.

### 1.2.3 Zásobník a zásobníkový rámec jazyka C

Prístup k lokálnym premenným funkcie pomocou ukazatela na vrchol zásobníku sa neosvedčil a tento prístup bol nahradený vhodnejším riešením. Ako jednoduchšie sa javilo uložiť si do registra hodnotu adresy, na ktorú ukazoval ukazatel na vrchol zásobníka `ESP` pred vstupom do volaného podprogramu. Toto umožňuje adresovať lokálne premenné funkcie a parametre relatívne k tejto adrese.

V praxi tento princíp funguje tak, že pri vstupe do podprogramu sa pôvodná adresa `EBP` uloží na zásobník a do `EBP` sa uloží aktuálna hodnota `ESP`. Po tomto úkone sa na zásobníku vyhradí priestor pre lokálne premenné a medzivýsledky, ktoré je samozrejme možné adresovať relatívne vzhľadom na `EBP`. Táto štruktúra sa nazýva *zásobníkové okno* alebo aj *stack frame*, demonštrované na obrázku 1.3.

Pri zanorení sa programu do svojich podprogramov vzniká na zásobníku takzvaná reťaz rámcov, pretože hodnota `EBP` uložená na zásobníku v aktuálnom rámci ukazuje na predošlý rámec. Týmto spôsobom je možné ľahko zistiť postupnosť volania podprogramov. Na druhej strane táto metóda môže byť zneužitá pri útoku na program. Tento útok bude vysvetlený v kapitole 2.1.1.




Obrázek 1.3: Zásobníkový rámec v jazyku C

V tejto podkapitole sme opísali základy práce s pamäťou a jej segmentáciu v jazyku C

pre architektúru Intel x86, ktoré by mali byť postačujúce na pochopenie priebehu útokov, ktoré budú neskôr opísané.

### 1.2.4 Prekladač GCC

Nasledujúca podkapitola sa venuje prekladaču *GNU C*. Informácie pre túto kapitolu boli čerpané najmä z [10] a [7]. Tento prekladač je pomerne rozšírený vzhľadom na to, že sa jedná o voľne šíriteľný software. Hlavnou výhodou tohto prekladača je použiteľnosť na rôznych architektúrach a operačných systémoch a taktiež schopnosť prekladať do binárnej podoby rôzne programovacie jazyky ako napríklad C/C++, Java , Fortran a Assembler. V tejto kapitole sa nebudeme zaoberať prekladačom GCC do hĺbky ale ozrejmíme o akú úlohu sa snaží vytvorený analyzátor kódu v procese tvorby spustiteľného programu.

Preklad programu prebieha zjednodušene v nasledujúcich krokoch:

1. Predspracovanie zdrojového kódu pomocou preprocesora. Úlohou preprocesora je uľahčiť prácu samotnému kompilátoru. Preprocesor napríklad odstraňuje komentáre, zabezpečuje vloženie hlavičkových súborov, rozvoj makier a nahrádza definované konštantné výrazy. Výsledkom je stále textový súbor.

2. Kompilátor prevezme kód od preprocesora a prevádza preklad do relatívneho objektového kódu počítača (`RTL`). Relatívny kód ešte neobsahuje adresy premenných a funkcií, ktoré v tejto časti prekladu nie sú známe. Výsledkom práce kompilátora je objektový kód(štandardne s príponou *.obj*).

3. Poslednú časť tvorby spustiteľného programu vykonáva zastavovací program (linker). Linker v relatívnom kóde nahradí adresy za absolútne a dereferuje dosiaľ neznáme odkazy, napríklad na funkcie z rôznych knižníc. Výsledkom práce linkera je samotný spustiteľný program.

Pre objasnenie práce vytvoreného analyzátora kódu sa bližšie pozrieme na prácu kompilátora a jeho detekciu chýb. Prekladač je so zapnutým prepínačom `-Wall` schopný odhaliť veľké množstvo chýb ako napríklad chyby vznikajúce pri konverzii dátových typov, priradzovanie nekompatibilných typov, chyba pri deklarovaní premennej, chýbajúce parametre funkcie alebo varovať užívateľa o nevyužitej premennej. Avšak tieto detekcie nie sú dostačujúce pre bezpečnosť programov. K útokom môže dochádzať aj v konštrukciách, ktoré sa pre prekladač javia ako transparentné. Rôzne druhy útokov budú spomenuté v kapitole číslo 2. Kvôli týmto prípadom je vhodné použitie analyzátora kódu, ktorý môže prípadné bezpečnostné chyby programu odhaliť a umožniť opravu programátorom.

### 1.2.5 Zhrnutie

Úlohou tejto kapitoly bolo objasniť teoretické základy jazyka Assembler a C a prekladača GCC, ktoré boli využité pri tvorbe návrhu vyvíjaného programu. Detailnejšie informácie vychádzajúce z tejto kapitoly budú uvedené v nasledujúcich kapitolách.

# Kapitola 2

# Popis exploitačných techník

V nasledujúcej kapitole bude čitateľ oboznámený so základnými exploitačnými technikami, na ktoré sa vytvorený analyzátor kódu jazyka C špecializuje. Jedná sa najmä o techniky založené na pretečení pamäti, medzi ktoré patrí pretečenie bufferu (*Buffer Overflow*) – pretečenie bufferu založeného na zásobníku (*Stack Buffer Overflow*), pretečenie v segmente haldy (*Heap Overflow*) a pretečenie v segmente BSS (*BSS Overflow*). Ďalej budú vysvetlené aj exploitačné techniky ako použitie formátovacích reťazcov (*Format String Exploit*) a pretečenie čísiel (*Integer Overflow*).

Spomenuté techniky sú založené na narušení pamäti. Tieto techniky sa snažia prevziať kontrolu nad vykonávaním toku programu. Toto je možné docieliť tým, že program je donútený vykonať úsek kódu, ktorý bol do pamäte uložený útočníkom. Tento druh útoku sa nazýva vykonanie svojvoľného kódu (*execution of arbitrary code*), pretože program vykoná časť kódu, ktorú vykonať nemal, ale vykoná sa časť určená útočníkom.

Zraniteľné časti v programoch, ktoré sa dajú zneužiť na prevzatie kontroly nad vykonávaním, sú obyčajne spôsobené nepozornosťou programátora. Za normálnych okolností pri výskyte udalosti neočakávanej programátorom program buď zhavaruje alebo sa správa inak ako sa od neho očakáva. Avšak v mnohých prípadoch bývajú chyby zneužívané útočníkmi na prevzatie kontroly nad vykonávaním programov. V súčasnej dobe sú tieto druhy útokov použité v rôznych internetových *malware* (malicious software – škodlivý kód).

## 2.1 Pretečenie pamäti (Buffer Overflow)

Pretečenie pamäti patrí medzi najstaršie exploitačné techniky, avšak objavuje a používa sa dodnes. Útok je založený na základnej vlastnosti, že keď je premennej pridelené určité pamäťové miesto, neexistuje žiadne bezpečnostné opatrenie, ktoré by zaistilo, že obsah premennej sa musí vojsť do tohto prideleného miesta. Táto úloha ostáva na pleciach programátora, ktorý však môže tento aspekt prehliadnuť a program sa stáva zraniteľným.

Pri neošetrení dĺžky vkladanej premennej do buffera môže prísť k situácii, že napríklad do buffera o veľkosti 20 bajtov sa snažíme vložiť 30 bajtov dát. Táto akcia bude povolená, pretože v programe nie je implementovaná bezpečná konštrukcia zisťujúca veľkosť vkladaných dát. Táto operácia prepíše aj dáta, ktoré nie sú pridelené bufferu. Pokiaľ sa jedná o kritické dáta, program zhavaruje. Táto situácia sa nazýva pretečenie bufferu (*buffer overflow* prípadne *buffer overrun*). Tento druh havárie programu je pomerne bežný, pretože chyba na kontrolu dĺžky premennej sa do programu dostane nepozornosťou veľmi jednoducho a o to ťažšie sa následne hľadá. Na odhalenie týchto aspektov programu by mal

slúžiť vytvorený analyzátor kódu, ktorý tieto chyby vyhľadá a programátor môže vytvoriť bezpečnejší a stabilnejší kód.

### 2.1.1 Pretečenie bufferu založenom na zásobníku (Stack Buffer Overflow)

Pretečenie bufferu založené na zásobníku vychádza z popisu základného pretečenia pamäti uvedeného v kapitole 2.1 s tým, že premenné sa nachádzajú v pamäti zásobníka. Keďže zásobník je dátová štruktúra typu *LIFO* (Last in – First out), pri pridávaní dát do zásobníka sa adresa uložená v ukazateli na aktuálny vrchol zásobníka (`SP`) postupne znižuje od adresy začiatku zásobníka. Z tohto princípu vyplýva, že v prípade pretečenia premennej uloženej na zásobníku môže dôjsť k prepísaniu napríklad zásobníkového rámca (*stack frame*) funkcie, v ktorého štruktúre sa premenná nachádza. Zásobníkový rámec obsahuje taktiež aj návratovú adresu funkcie (podrobnejší popis činnosti zásobníka sa nachádza v kapitole 1.2.3). Tento fakt značí, že v prípade pretečenia buffera na zásobníku môže byť prepísaná hodnota návratovej adresy funkcie. Ak dôjde k náhodnému prepísaniu tejto adresy program iba zhavaruje. Avšak v prípade, že útočník prepíše adresu cielene, môže ako návratovú adresu podvrhnúť adresu vlastného kódu, čím môže prevziať kontrolu nad vykonávaním programu.

Toto je možné docieliť napríklad použitím premenných prostredia (*enviroment variables*) v shelle systému *UNIX*, do premennej prostredia sa uloží kód, ktorý chce útočník vykonať. Výhodou použitia premenných prostredia je, že adresa premennej je ľahko zistiteľná a nemenná. Nevýhodou je funkčnosť tohto postupu len na jednej platforme. Po zistení adresy premennej prostredia môže útočník túto adresu podvrhnúť do návratovej adresy funkcie, potom pri návrate z funkcie sa do `EIP` (viď. kapitola 1.1.1) vloží podvrhnutá adresa obsahujúca útočníkov kód.

### 2.1.2 Pretečenie bufferu v segmente haldy (Heap Overflow)

Táto exploitačná technika vychádza taktiež z popisu uvedenom v kapitole 2.1 s tým rozdielom, že premenné sú alokované dynamicky na halde napríklad pomocou funkcie `malloc()`. Pri zápise rozsiahlejších dát ako je veľkosť alokovanej premennej môže dôjsť taktiež prepísaniu údajov, ktoré nepatria premennej, do ktorej sa zapisovalo. Na základe tohto javu môže útočník vykonať rôzne škodlivé akcie.

Na druhej strane tento princíp exploitovania sa v súčasnosti dostáva do úzadia, pretože v novších verziách používaných operačných systémov sú implementované rôzne metódy na udržanie konzistencie haldy. Tieto funkcie sú do operačných systémov pridané za účelom odhalenia prípadných útokov na haldu, napríklad rozpojenie haldy (*heap unlinking*), kedy v prípade zistenia nekonzistencie haldy ukončia program a vypíšu ladiace informácie.

### 2.1.3 Pretečenie v segmente BSS

Táto technika je založená na pretečení dát v premennej uloženej v segmente BSS. V segmente BSS sú uložené statické a globálne premenné, ktoré v čase prekladu programu neboli inicializované dátami. V prípade, že v čase prekladu je premenná inicializovaná, uloží sa do segmentu data. Podrobnejší popis segmentov pamäti v jazyku C je uvedený v kapitole 1.2.1.

Pri tomto druhu útoku platia princípy uvedené v predchádzajúcich kapitolách, kedy pri pretečení vkladania dát do premennej môže útočník zmeniť aj iné premenné a tým ovplyvniť vykonávanie programu.

## 2.2 Formátovacie reťazce

Použitie formátovacích reťazcov je ďalšia z účinných metód exploitovania, pomocou ktorej je možné prevziať kontrolu nad nesprávne zabezpečeným programom. Aj tento typ exploitov je založený na chybách, ktoré sú do programu zanesené programátorom a na prvý pohľad sa javia ako bezpečné konštrukcie. Na druhej strane tento druh chýb, ktoré by sa dali zneužiť na exploitáciu programov, je možné pomerne jednoducho odhaliť a odstrániť. Preto sa tento typ zraniteľnosti exploituje pomerne v menšom množstve.

Funkcia `printf()`, ktorá ako prvý parameter prijíma formátovací reťazec, pracuje so zásobníkom, a preto môže dôjsť pri nesprávnom použití k vzniku bezpečnostnej diery v programe. Funkcia postupne pri svojom zavolaní vloží na zásobník parametre v opačnom poradí, to znamená, že na vrchole zásobníka sa bude nachádzať samotný formátovací reťazec a na dne miesta zabraného dátami funkcie bude uložený posledný parameter. V prípade, že bude vo formátovacom reťazci požadovaných viac hodnôt parametrov ako je funkcii predaných, funkcia použije adresu dát zo zásobníka, ktoré by tomuto parametru odpovedali v prípade, keby bol parameter zadaný. To znamená, že funkcia by zapísala prípadne prečítala dáta z pamäti, ktorá jej nepatrí. Táto vlastnosť sa dá využiť na vykonanie škodlivých akcií. Avšak počet parametrov nie je možné bežne dynamicky meniť, tak bude nutné použiť iný prístup.

Práca funkcie `printf()`, prípadne `sprintf()` s formátovacím reťazcom spočíva v tom, že číta formátovací reťazec po znaku. Ak narazí na normálny znak, ktorý nie je formátovacím znakom, prepíše tento znak jednoducho na výstup. V prípade, že sa jedná o formátovací znak, obyčajne začínajúci znakom %, pomocou argumentu na zásobníku vykoná zodpovedajúcu akciu. Ako bolo spomenuté vyššie, tento argument sa nemusí na zásobníku nachádzať, čo spôsobuje potenciálne miesto na zraniteľnosť programu.

V prípade, že sa nachádza chyba napadnuteľná týmto spôsobom v programe, môže útočník pomocou formátovacích reťazcov prečítať dáta z ľubovoľného miesta v pamäti, prípadne vykonať zápis do ľubovoľného miesta v pamäti. Najzákladnejšou chybou, ktorú je možné zneužiť na útok pomocou formátovacieho reťazca je, ak funkcii `printf()` predáme ako parameter priamo samotný reťazec ako formátovací reťazec `printf(string)`, ktorý sa má tlačiť a nepredáme iba odkaz na tento reťazec `printf("%s",string)`. Z hľadiska základnej funkcie sa javia oba spôsoby ako ekvivalentné, avšak v prípade, že tlačený reťazec obsahuje formátovacie znaky, nevytlačia sa, ale zo zásobníka sa prečíta zodpovedajúca hodnota parametra, prípadne sa na adresu získanej zo zásobníka zapíšu dáta.

## 2.3 Pretečenie čísel (Integer Overflow)

Zraniteľnosť spôsobená pretečeným čísel typu integer je spôsobená tým, že čísla majú presne definovaný rozsah hodnôt. Minimálna a maximálna hodnota veľkosti dátového typu integer je závislá na architektúre, v ktorej daný program beží. Zraniteľnosť vyplývajúca z pretečenia číselného dátového typu je spôsobená nepozornosťou a nezohľadnením, že dátový typ ma presne definované minimum a maximum. Podrobnejšie informácie je možné čerpať z [2], odkiaľ vychádzala aj táto podkapitola. Zraniteľnosť spôsobenú číslami je možné rozdeliť do štyroch častí:

- **Pretečenie (overflow)** – môže nastať v prípade, že výsledok číselnej operácie, ktorý sa má uložiť do premennej, je väčší ako maximálna hodnota dátového typu premennej. To spôsobí, že hodnota pretečie a v premennej bude uložená nezmyselná

hodnota. Tento jav môže spôsobiť nepredvídateľné chovanie programu a taktiež predstavuje bezpečnostné riziko. V súčasnosti je táto chyba využívaná na exploitovanie v najväčšom množstve.

- **Podtečenie (underflow)** – opačný prípad ako pretečenie, nastáva, ak je výsledok číselnej operácie menší ako minimum dátového typu. Toto spôsobí, že výsledok pretečie a jeho hodnota nebude blízka minimu dátovému typu ale naopak, bude sa nachádzať v maximálnej časti rozsahu. Tento typ je zriedkavejší ako pretečenie, pretože sa vyskytuje len pri odčítavaní čísel.

- **Chyba konverzie znamienka (sign conversion error)** – chyba spôsobená tým, že číslo bez znamienka je reprezentované ako číslo so znamienkom, prípadne naopak. Toto spôsobí, že číslo ma úplne inú hodnotu ako je predpokladané. Pretože typicky sa na reprezentáciu čísel so znamienkom používa dvojkový doplnok, kde *MSB* (*Most significant bit*) predstavuje znamienko čísla. V prípade, že budeme toto číslo reprezentovať ako číslo bez znamienka, jeho hodnota môže byť neočakávaná.

- **Chyba zaokrúhľovania (truncation error)** – chyba spôsobená priradením čísla s dátovým typom s väčším rozsahom do premennej s menším rozsahom. V určitých prípadoch sa môže stať, že niektoré bity čísla budú zahodené a nebudú sa brať do úvahy. To spôsobí stratu časti dát a taktiež môže slúžiť ako bod zneužiteľný pri útoku na program.

## 2.4 Zhrnutie

Táto kapitola obsahuje popis základných exploitačných techník a z časti aj možnú obranu proti nim. Z týchto informácii sa vychádzalo pri návrhu aplikácie, umožňujúcej detekciu týchto slabých miest programov. Detaily návrhu budú popísané v nasledujúcich kapitolách.

# Kapitola 3

# Návrh a implementácia

Nasledujúca kapitola bude pojednávať o samotnom návrhu analyzátora kódu, postaveného na statickej analýze a jeho implementácii. Pozornosť bude zameraná najmä na štruktúru programu, komunikáciu medzi objektmi v programe a podrobnejší opis jednotlivých častí.

## 3.1 Statická analýza

*Statická analýza* nevykonáva samotný testovaný program v zmysle strojového kódu, ale sústredí sa len na analýzu zdrojového kódu a zber informácií o programe. Statická analýza sa používa hlavne na testovanie správnosti programov, ale je ju možné použiť aj napríklad pri optimalizácii.

Výhody statickej analýzy:

- Statická analýza umožňuje spracovávať často aj veľmi rozsiahle programy.
- Na jej vykonanie nie sú potrebné všetky knižnice programu, vstupy a výstupy prípadne všetky programové moduly.

Nevýhody:

- Pri analýze sa produkuje pomerne veľké množstvo falošných hlásení.
- Pridávaním nových typov chybových analýz vedie k ďalšiemu zvýšeniu množstva falošných hlásení.
- Statické analyzátory sú obyčajne špecializované na určitý typ problémov.

Tento krátky popis zobrazil základné informácie o statickej analýze, ktoré boli čerpané najmä z [11].

## 3.2 Štruktúra programu

Analyzátor sa skladá z troch základných jednotiek a to konkrétne *lexikálny analyzátor*, *syntakticko-sémantický analyzátor* a *bezpečnostný analyzátor*. Každý z týchto modulov analyzuje zdrojový kód vstupného programu na rôznych úrovniach. Spolu s týmito základnými jednotkami sa v programe nachádzajú aj pomocné moduly, ktoré abstrahujú rôzne vlastnosti

vstupných zdrojových programov, aby sa s nimi dalo pracovať v programe na vyššej úrovni. Hlavným pomocným modulom je takzvaná *tabuľka funkcií*, ktorá zabezpečuje nosnú dátovú štruktúru na uchovávanie informácii o programoch. Tabuľka funkcií je používaná všetkými analyzačnými jednotkami programu, ale každý modul k nej pristupuje iným spôsobom, preto bude tabuľka podrobnejšie popísaná ako prvá v nasledujúcej podkapitole. Ďalej bude nasledovať popis jednotlivých výkonných modulov v poradí, v akom nasleduje ich činnosť pri analýze programu.

Detailné podrobnosti o každom z modulov budú postupne uvedené v nasledujúcich podkapitolách.

## 3.3 Tabuľka funkcií

*Tabuľka funkcií*, ako už bolo spomenuté, zabezpečuje uloženie a správu informácií abstrahovaných zo vstupného súboru. Jedná sa o pomerne rozsiahlu dátovú štruktúru, pretože k analýze programu je potrebné vyabstrahovať veľké množstvo údajov, ktorých počet nie je dopredu známy. Z tohto dôvodu je potrebné tabuľku vytvárať dynamicky na halde. Týmto sa zabezpečí aj jej efektívne predávanie medzi jednotlivými modulmi, keďže bude pre každú inštanciu programu vytvorená iba jedenkrát a medzi modulmi bude predávaná iba vo forme odkazu.

Tabuľka funkcií v programe zabezpečuje taktiež udržiavanie *kontextu analýzy programu*, kde je možné zistiť a nastaviť aktuálnu pracovnú funkciu. Tento spôsob umožňuje oddelenie prístupu k jednotlivým funkciám a tak môžu mať funkcie oddelené množiny premenných a nebude dochádzať k prepisovaniu informácii o premenných s rovnakým názvom, ktoré sa vo vstupnom súbore nachádzajú v rozdielnych funkciách. V súvislosti s kontextom analýzy udržuje tabuľka funkcií aj informácie o aktuálne spracúvanom súbore.

### 3.3.1 Implementácia tabuľky funkcií

Tabuľka je v programe implementovaná pomocou viacerých tried. Hlavnou nadradenou triedou je trieda s názvom `programFunctions`. Táto trieda predstavuje najvyššiu abstrakciu informácií, pretože obsahuje rozhranie na jednotnú prácu s informáciami o funkciách. Na základe uloženého kontextu o pracovnom súbore a funkcii, umožňuje pridávanie symbolu k funkcii, nastavenie návratovej hodnoty a parametrov funkcie alebo pridanie príkazu k funkcii. Na základe tohto prístupu sa programátor v moduloch, ktoré inicializujú tabuľku funkcií, nemusí zaoberať, do ktorej funkcie má informáciu pridať, pretože o toto sa stará uložený kontext, ktorý je možné zmeniť z ľubovoľného modulu programu. Napríklad kontext o pracovnom súbore sa mení v lexikálnom analyzátore a kontext pracovnej funkcie je prestavovaný až v syntakticko-sémantickom analyzátore. Nosným prvkom tejto triedy je `STL` kontajner typu `map`, ktorý predstavuje určitý typ asociatívneho poľa, kde kľúčom je názov funkcie a hodnotou je ukazateľ na objekt triedy `functionStatementList`. To znamená, že trieda `programFunctions` obsahuje hlavne interné informácie určené na jednoduchšiu prácu s funkciami. Okrem toho obsahuje a spravuje aj *globálnu tabuľku symbolov* programu, kde sú uložené premenné dostupné zo všetkých funkcií.

Spomenutá trieda `functionStatementList` tvorí v triednej hierarchii tabuľky funkcií druhú abstraktnú úroveň. Hlavnou úlohou tejto triedy je uloženie informácií o jednej konkrétnej funkcii. Podobne ako nadradená trieda `programFunctions`, obsahuje táto trieda interné informácie o funkcii a nie konkrétne hodnoty. Cieľom tejto triedy je zjednodušiť prácu s podradenými čiastkovými tabuľkami, z ktorých sa tabuľka funkcií skladá. Preto táto

trieda obsahuje odkazy na objekty spravujúce štyri elementárne tabuľky, ktoré tvoria jeden abstraktný celok. A to konkrétne tabuľka symbolov, tabuľka parametrov funkcie, tabuľka príkazov funkcie a tabuľka udržiavajúca zoznam nájdených chýb vo funkcii. V nasledujúcich podkapitolách si tieto čiastkové tabuľky podrobnejšie popíšeme.

Tabuľka symbolov a parametrov funkcie je implementovaná triedou `symbolTable`. Táto trieda spravuje záznamy o jednotlivých premenných, respektíve parametroch funkcie. Samotná implementácia je postavená podobne ako v triede `programFunctions` na `STL` kontajneri `map`, kde je asociatívnym kľúčom názov premennej alebo parametra a hodnotou je ukazateľ na objekt triedy `semanticDetails`.

Trieda `semanticDetails` predstavuje jednu z najnižších tried v hierarchii tabuľky funkcií, pretože obsahuje konkrétne údaje o danom symbole. Medzi konkrétne údaje napríklad patrí dátový typ symbolu, jeho veľkosť, aktuálna hodnota využitia tejto veľkosti alebo príznak pretečenia. Všetky tieto elementy je možné nastavovať, zmeniť a získať z nadradenej triedy `symbolTable`.

Tabuľka príkazov funkcie vychádza z triedy `statementTable`. Oproti tabuľke symbolov je implementácia tejto tabuľky podstatne zložitejšia, pretože príkazy majú v jazyku C rôznu syntax a sémantiku a problém taktiež spôsobuje, že na mieste parametrov vo funkcii môže byť opäť volanie funkcie. Trieda statementTable preto obsahuje iba `STL` kontajner typu `vector`, ktorý uchováva odkazy na objekty triedy `statementDetails`, ktoré predstavujú samotné príkazy.

Trieda `statementDetails` už obsahuje konkrétnejšie informácie o samotnom príkaze. Uchováva hodnotu určujúcu riadok, na ktorom sa daný príkaz nachádza vo vstupnom súbore, taktiež uchováva názov funkcie, prípadne interný názov príkazu a implementuje rozhranie k správe zoznamu parametrov príkazu. Zoznam parametrov príkazu je implementovaný pomocou `STL` kontajnera typu `vector`, ktorý uchováva ukazatele na objekty triedy `statementParams`.

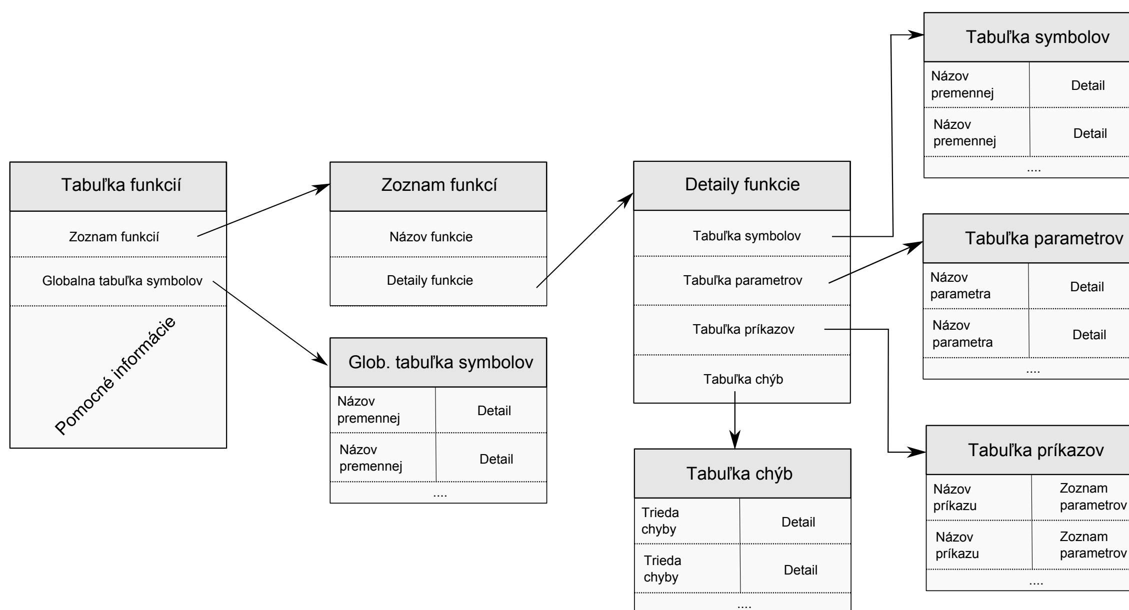
Trieda `statementParams` už spravuje konkrétne údaje o parametri príkazu funkcie. Rozlišuje typ parametra, jeho hodnotu, veľkosť, v prípade, že je parametrom volanie funkcie, obsahuje ukazateľ na objekt reprezentujúci volanú funkciu, ktorý je inštanciou triedy `statementDetails`. Z toho vyplýva, že je možné v programe vytvoriť ľubovoľné množstvo zanorení volania funkcií. Taktiež je možné vyčíslovať výrazy v parametroch príkazu aj keď obsahujú volanie funkcie. Podrobnosti o problematike volania funkcií v parametri funkcie budú podrobnejšie ozrejmené v kapitole 3.6.2.

Poslednou tabuľkou dotvárajúcou implementáciu triedy `functionStatementList` je elementárna tabuľka nájdených chýb a problémov vo funkcii. Jedná sa o jednoduchšiu tabuľku, ktorá uchováva zoznam nájdených chýb vo funkcii implementovaný pomocou `STL` kontajnera `vector`. V tomto zozname sú uložené detaily o jednotlivých chybách spravované triedou `errorInfo`, ktorá uchováva hodnoty ako číslo riadka vo vstupnom programe, kde sa chyba odohrala, triedu chyby a detail príkazu, kde sa chyba vyskytla. Hlavným dôvodom ukladania si informácii o chybách a nevypisovanie chyby priamo pri jej detekcii je zotriedenie detailov o chybách do prívetivejšej podoby pre užívateľa a ich výpis až po skončení celej analýzy.

Výsledkom opisu nachádzajúcom sa v tejto podkapitole je štruktúra tabuľky funkcií zobrazenej na obrázku číslo 3.1. Táto hierarchická štruktúra umožňuje načítať potrebné údaje zo súboru na jeden prechod súborom, čo bolo hlavným dôvodom tohto návrhu ukladania dát. Návrh taktiež poskytuje možnosť pracovať z externými hlavičkovými súbormi, odkiaľ budú načítané ďalšie funkcie testovaného programu. Opačným prípadom k tomuto prístupu by bolo vyhľadávanie chýb priamo za behu lexikálnej a syntakticko-sémantickej

analýzy, avšak v tom prípade by musel byť celý súbor prečítaný niekoľkonásobne. Z tohto dôvodu bol zvolený pomerne zložitejší prístup, ktorý je na druhej strane efektívnejší.

V nasledujúcej kapitole bola opísaná základná dátová štruktúra analyzačného programu, ďalšie podrobnosti o práci s touto štruktúrou budú vysvetlené v príslušných kapitolách jednotlivých modulov.




Obrázek 3.1: Štruktúra tabuľky funkcií

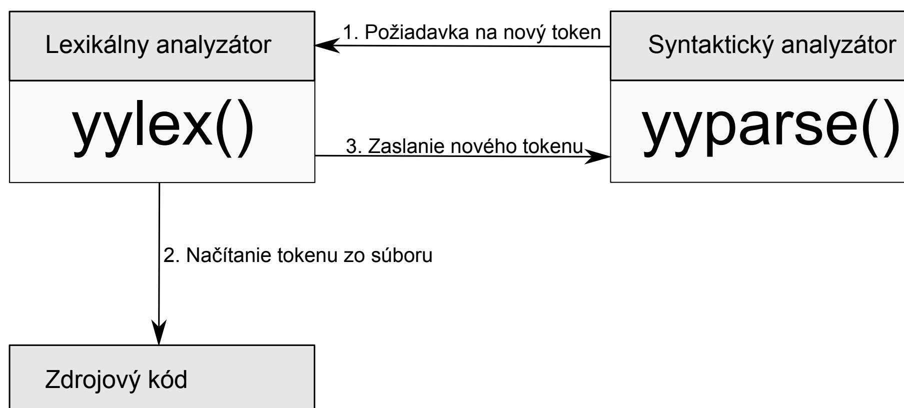
## 3.4 Lexikálny analyzátor

*Lexikálny analyzátor* je jediný modul, ktorý prichádza do kontaktu so zdrojovým kódom testovaného programu. Jeho základnou úlohou je čítanie vstupného súboru po znakoch, pričom je potrebné prečítať toľko znakov, aby bolo možné rozpoznať ďalší *lexikálny symbol*, často nazývaný aj *lexém* alebo *token*. Lexikálny analyzátor je implementovaný vo funkcii `yylex`, ktorá úzko spolupracuje so syntakticko-sémantickým analyzátorom. Princíp ich komunikácie je vyobrazený na obrázku číslo 3.2. Lexikálny analyzátor by mohol byť implementovaný aj priamo v syntakticko-sémantickom analyzátore, avšak tento prístup by spôsobil, že implementácia analyzátora by bola niekoľkonásobne náročnejšia a menej efektívna. Z tohto dôvodu bola zvolená možnosť, kedy je lexikálny analyzátor implementovaný externe a nie je súčasťou syntakticko-sémantického analyzátora.

Keďže lexikálny analyzátor je jediný modul programu pracujúci s textom vstupného programu, vykonáva aj ďalšie doplnkové funkcie okrem rozpoznávania lexémov. Medzi doplnkové funkcie napríklad patrí odstraňovanie medzier, oddeľovačov a komentárov z načítaného súboru. Lexikálny analyzátor taktiež spracúva makrá preprocesora. Medzi makrá preprocesora podporované analyzátorom patrí direktíva `#include`, ktorá umožňuje do súboru vložiť *externý hlavičkový súbor*. Názov vloženého súboru sa uloží počas analýzy súboru a po jej skončení začne analýza súboru vloženého direktívou `#include`. K ďalšej doplnkovej funkcii lexikálneho analyzátora patrí zisťovanie čísla riadka, na ktorom sa lexém nachádza.

Lexikálny analyzátor má samozrejme prístup aj k tabuľke funkcií, kde je jeho hlavnou úlohou meniť kontext pracovného súboru na základe názvu aktuálne spracovávaného vstupu.

Základné informácie o lexikálnom analyzátore boli čerpané z [1]. Konkrétna implementácia lexikálneho analyzátora do analyzačného modulu programu bola tvorená pomocou nástroja *Flex*, ktorého princíp bude popísaný v nasledujúcej podkapitole. Taktiež bude spomenutá aj trieda `lexicalToken`, ktorá tvorí spojovaciu časť s syntakticko-sémantickým analyzátorom.




Obrázek 3.2: Schéma komunikácie medzi lexikálnym a syntakticko - sémantickým analyzátorom

### 3.4.1 Flex

*Flex* je *open-source* implementáciou unixového nástroja *lex*. To znamená, že je to nástroj slúžiaci na generovanie lexikálneho analyzátora (*scanner*) na základe formálneho popisu jazyka. Scanner je program, ktorý rozpoznáva lexikálne vzory v texte a zoskupuje ich do prúdu tokenov.

Proces pri tvorbe analyzátora pomocou nástroja Flex spočíva v definícii formálneho jazyka pomocou regulárnych výrazov. Na základe vstupného definičného súboru vytvorí Flex kód v jazyku C prípadne C++. V samotnom kóde programu je špecifikovaný formálny jazyk prijímaný konečným automatom.

Vstupný súbor pre flex pozostáva z 3 častí. Má nasledujúcu štruktúru:

```
Definície
%%
Pravidlá
%%
Užívateľský kód
```

Kde:

- **Definície** – časť obsahujúca užívateľské definície premenných používaných v definičnom súbore. Užívateľ má možnosť na tomto mieste definičného súboru vložiť aj

makrá preprocesora, ktoré budú pregenerované do výstupného kódu. Táto časť taktiež môže obsahovať deklaráciu premenných, ktoré budú používané vo vygenerovanom kóde.

- **Pravidlá** – nosná časť vstupného definičného súboru obsahujúca definície jednotlivých lexémov pomocou regulárnych výrazov a k nim odpovedajúce akcie. Na základe týchto akcií posiela lexikálny analyzátor tokeny syntakticko-sémantickému analyzátoru.
- **Užívateľský kód** – v tejto sekcii má užívateľ možnosť definovať vlastný kód, napríklad pomocné funkcie používané v akciách pravidiel.

### 3.4.2 Definičný súbor pre Flex

Definičný súbor použitý na vygenerovanie lexikálneho analyzátora bol vytvorený na základe voľne dostupného definičného kódu [3] pre jazyk *ANSI C*, ktorý bol prepracovaný podľa potreby syntakticko-sémantického analyzátora. Oproti vzorovému kódu pre ANSI C boli vytvorené pre všetky lexémy nové akcie a taktiež bolo nutné vytvoriť sekciu užívateľských funkcií.

Príklad definície pravidla pre lexém:

```
{Letter}({Letter}|{Digit})*
{
  yylval->tok =
    (new lexicalToken(line,yy::token::IDENTIFIER,string(yytext)));
  return yy::token::IDENTIFIER;
}
```

Táto konštrukcia definuje akciu v prípade nájdenia tokenu vyhodnoteného ako *identifikátor*, kedy je vytvorený nový objekt triedy `lexicalToken` s nastavenými parametrami určujúcimi riadok, typ a názov identifikátora. Trieda `lexicalToken` je nosná trieda na prácu s tokenmi. Podrobnejšie bude popísaná v ďalšej kapitole.

Z popisu uvedeného v tejto kapitole vyplýva, že celá činnosť lexikálneho analyzátora spočíva v rozpoznaní tokenu pomocou konečného automatu, zapuzdrenie potrebných informácii do objektu triedy `lexicalToken` a zaslanie ukazatela na tento objekt na ďalšie spracovanie.

Výsledný definičný kód pre Flex je možné, pre jeho rozsiahlosť, nájsť v prílohe na CD. Ďalšie podrobnosti o práci nástroja Flex je možné nájsť v manuálových stránkach [9].

### 3.4.3 Trieda lexicalToken

Trieda `lexicalToken` zabezpečuje prenos potrebných informácií z lexikálneho analyzátora do syntakticko-sémantického analyzátora. Každý lexikálny token posielaný z lexikálneho analyzátora je zapuzdrený do objektu tejto triedy. Tento prístup bol zvolený hlavne kvôli jednotnosti práce s tokenmi a k zvýšeniu prehľadnosti syntakticko-sémantického analyzátora, kde sa s tokenmi ďalej pracuje. V prípade použitia viacerých dátových typov pre token by bolo pri ďalšom spracovaní nutné vykonávať nadbytočné kontroly, čo by znižovalo efektivitu výsledného analyzátora.

Táto trieda umožňuje uloženie údajov ako typ tokenu, veľkosť, riadok v zdrojovom programe kde sa token nachádza, prípadne názov tokenu, ak sa jedná napríklad o konštantu

alebo identifikátor. Podrobnosti o ďalšom spracovaní tokenov budú uvedené v nasledujúcej kapitole 3.5 venovanej syntaktickej a sémantickej analýze.

V tejto kapitole boli zhrnuté podrobnosti o činnosti lexikálneho analyzátora v celkovom pohľade na tvorený bezpečnostný analyzátor. Podrobnejší popis interakcií lexikálneho analyzátora z pohľadu syntakticko-sémantického modulu programu budú uvedené v ďalšom texte.

## 3.5 Syntakticko-sémantický analyzátor

*Syntakticko-sémantický analyzátor* tvorí nosnú a zároveň najrozsiahlejšiu časť celého analyzátora. Primárnou úlohou tohto modulu je určiť kontext programu a na základe toho ukladať informácie do tabuľky funkcií. Taktiež tento analyzátor určuje, či vstupný kód spĺňa syntaktické pravidlá analyzovaného jazyka. Pri tvorbe tohto modulu bol použitý nástroj *Bison*. Nástroj Bison si podrobnejšie popíšeme v nasledujúcej podkapitole.

Syntakticko-sémantický analyzátor je úzko prepojený s lexikálnym analyzátorom, ako je možné vidieť na obrázku 3.2. Zjednodušene povedané, funkčnosť modulu spočíva v zhlukovaní prúdu tokenov generovaných lexikálnym modulom do viet, odpovedajúcich zdrojovému jazyku. Na základe tohto rozdelenia do viet, je možné zistiť syntaktickú správnosť programu. V praxi je ale od syntaktického analyzátora požadované viac. V našom prípade je výstupom syntaktického modulu špeciálna reprezentácia kódu vyextrahovaná zo vstupného testovaného programu. U nás túto štruktúru predstavuje naplnená tabuľka funkcií, ktorá obsahuje iba dáta skutočne potrebné na analýzu zameranú na vyhľadávanie zraniteľných miest voči exploitácii.

Opačným prístupom pri implementácii tohto modulu teoreticky mohlo byť spracovanie zdrojového súboru ako text. To znamená, že prúd tokenov by sa spracovával sekvenčne a na základe napríklad konečného automatu by sa udržiaval kontext programu. Na základe kontextu by sa potom analyzátor rozhodoval pri spracovaní a ukladaní získaných dát do tabuľky funkcií. Tento prístup by bol oproti implementácii komplexného syntakticko-sémantického analyzátora jednoduchší, avšak na druhej strane by bola takáto analýza pomerne nepresná, hlavne pri nie úplne bežných konštrukciách jazyka. Tento prístup by taktiež neumožňoval zistiť syntaktickú správnosť jazyka. Najmä z týchto pádnych dôvodov bola v programe zvolená cesta implementácie kompletného syntakticko-sémantického analyzátora na základe *LR* gramatiky jazyka ANSI C pomocou nástroja Bison. Pomocou tejto implementácie je možné predspracovať určité časti kódu už počas vlastnej analýzy. Spracovanie už počas analýzy bolo zvolené napríklad pri výrazoch v jazyku C, kedy analyzátor vyčísli čo najväčšiu možnú časť výrazu a tým uľahčí a zrýchli vykonávanie samotnej bezpečnostnej analýzy.

### 3.5.1 Bison

*Bison* je nástroj šírený pod licenciou *GNU* (*General Public License*), ktorý primárne slúži na tvorbu syntaktického analyzátora (*parser*) na základe definovanej gramatiky. Bison vychádza z Unixového nástroja *Yacc*, s ktorým sa snaží udržať čo najvyššiu kompatibilitu.

Vstupom pre Bison je zvyčajne *bezkontextová gramatika*, najčastejšie zapísaná formou *BNF* (*Backus-Naur Form*). Výsledkom je deterministický parser založený na zásobníkovom automate.

Štruktúra vstupného súboru:

```
%{
 Prológ
%}
  Deklarácie určené pre Bison
%%
  Gramatické pravidlá
%%
  Epilóg
```

Kde:

- **Prológ** – poskytuje užívateľovi priestor deklarovať typy a premenné, ktoré budú používané v akciách. Taktiež je možné na tomto mieste deklarovať makrá preprocesora. V tejto sekcii sa odporúča deklarovať aj globálne používané funkcie `yylex` (lexikálny analyzátor) a `yyerror` (funkcia na výpis chybového hlásenia).

- **Deklarácie určené pre Bison** – naskytajú možnosť deklarovať mená terminálov a nonterminálov používaných v bezkontextovej gramatike. Pri tejto deklarácii je možnosť špecifikovať napríklad aj prioritu operátorov a dátové typy sémantických hodnôt rôznych symbolov.

- **Gramatické pravidlá** – predstavujú najdôležitejšiu časť vstupného súboru. V tejto sekcii je definovaná bezkontextová gramatika, ktorá určuje ako zostaviť nonterminál z jednotlivých terminálov. Ako bolo spomenuté, používa sa BNF forma.

- **Epilóg** – záverečná sekcia, kam je možné umiestniť akékoľvek ďalšie potrebné zdrojové kódy. Najčastejšie sa v tejto časti definujú funkcie, ktoré boli deklarované v prológu. V prípade, že sa k parseru používa iba jednoduchý obslužný program, môže sa sem umiestniť celý kód programu.

Ďalšie podrobnosti o nástroji Bison je možné dohľadať v manuále [5], odkiaľ boli čerpané detaily v tejto kapitole.

### 3.5.2 Definičný súbor pre Bison

Vstupný súbor pre nástroj Bison bol vytvorený na základe voľne dostupnej gramatiky [4] pre jazyk ANSI C. Táto gramatika však nevyhovovala potrebám analyzátora, preto bola do značnej miery upravená. Taktiež bolo nutné vytvoriť všetky sémantické akcie. Sémantické pravidlá boli vytvorené hlavne pre konštrukcie jazyka C, ktoré sú potenciálnym zdrojom nebezpečenstva.
Príklad pravidla vo BNF forme:

```
idList
: newId
| idList COMMA newId
;
```

Toto pravidlo špecifikuje všetky možnosti redukcií, z ktorých môžeme získať nonterminál `idList`. Znakom `|` sa oddeľujú jednotlivé varianty. Uvedené pravidlo patrí medzi takzvané rekurzívne pravidlá, keďže výsledný nonterminál sa vyskytuje aj na pravej strane

pravidla. Väčšina pravidiel pre Bison je zvyčajne rekurzívna, pretože to je jediný spôsob ako vyjadriť sekvenciu prvkov, ktorých je dopredu neznámy počet. Konkrétne uvedené pravidlo špecifikuje zoznam identifikátorov oddelených čiarkou, ktorý môže mať jeden až $N$ prvkov.

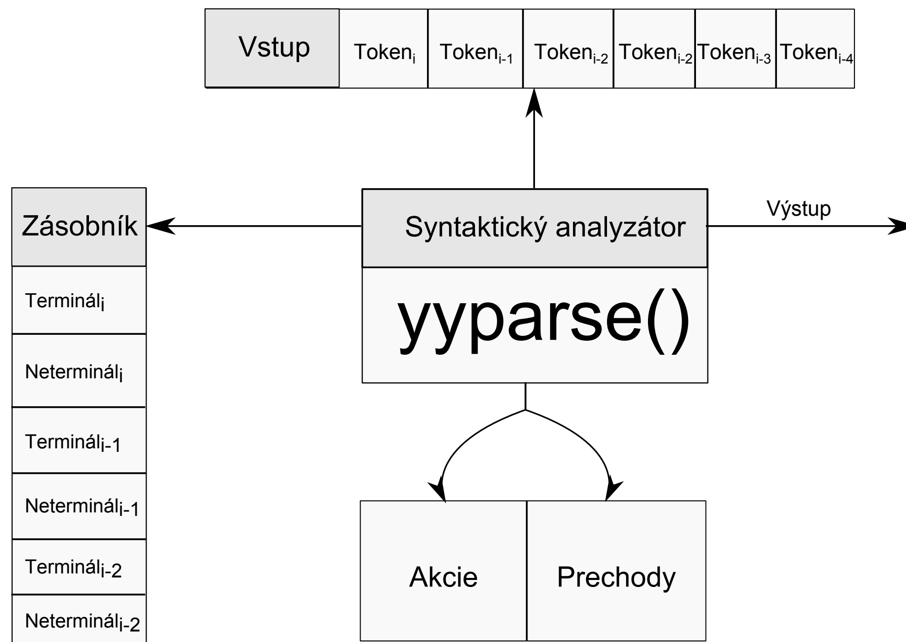
Výslednú bezkontextovú gramatiku použitú pri tvorbe analyzátora je možné nájsť v prílohe na CD, pretože sa jedná o pomerne rozsiahly súbor.

### 3.5.3 Činnosť syntakticko-sémantického analyzátora

Činnosť syntakticko-sémantického analyzátora pozostáva z čítania tokenov a ukladaní ich spolu s ich sémantickými hodnotami na zásobník. Ak dôjde k situácii, že posledných $N$ tokenov je v zhode s pravou stranou niektorého z pravidiel, je týchto $N$ tokenov nahradených za ľavú stranu daného pravidla. Táto operácia sa nazýva redukcia. Z tohto popisu vyplýva, že sa jedná o syntaktickú analýzu zdola nahor.

Syntaktická analýza zdola nahor sa snaží budovať derivačný strom zo vstupného reťazca smerom od listov po koreň. To znamená, že redukuje vstupný prúd tokenov na štartovací symbol definovanej gramatiky. Pri každom jednom z krokov je nahradený určitý počet symbolov za jeden symbol, až pokiaľ nie je dosiahnutý štartovací symbol, čo značí, že vstup bol syntakticky správny alebo nie je možné vykonať žiadnu redukciu, čo je spôsobené tým, že na vstupe nebol syntakticky správny prúd tokenov.

Syntaktický analyzátor pracujúci metódou zdola hore je v našom prípade založený na zásobníkovom automate. Celý schematický tvar je znázornený na obrázku 3.3. Skladá sa z vstupu, výstupu, zásobníka, prechodovej tabuľky a riadiaceho programu.




Obrázek 3.3: Schéma zásobníkového automatu

Riadiaci program komunikuje s lexikálnym analyzátorom a získava tokeny. Tieto tokeny zapuzdrené do objektu triedy lexicalToken následne ukladá na zásobník. Riadiaci program taktiež kontroluje a porovnáva hodnoty na zásobníku a na vstupe a na základe prechodovej tabuľky sa rozhoduje, či v nasledujúcom kroku vykoná redukciu alebo prečíta nasledujúci

symbol zo vstupu. V prípade redukcie sa podľa tabuľky vykoná odpovedajúca akcia.

Akcia reprezentuje kód, ktorý bol vložený vo vstupnom súbore pri generovaní pomocou nástroja Bison spolu s pravidlom.
Niektoré dôležité akcie si popíšeme:

- **Funkcia** – v prípade redukcie tokenov tvoriacich definíciu funkcie sa zo zásobníka získajú údaje o názve funkcie, type jej návratovej hodnoty a taktiež podrobnosti o názvoch a typoch parametrov. Tieto informácie sú následne uložené do tabuľky funkcií a táto funkcia je nastavená ako pracovná funkcia tabuľky funkcií.
- **Premenná** – redukcia premennej je pomerne jednoduchšia, pretože stačí získať zo zásobníka jej typ a uložiť informácie o nej do tabuľky symbolov aktuálnej pracovnej funkcie, prípadne, ak nie je aktívna žiadna funkcia, informácie sa uložia do globálnej tabuľky symbolov.
- **Volanie funkcie** – operácia vykonávaná pri volaní funkcie je obdobná ako pri deklarácii funkcie, to znamená, že sa zistí názov funkcie a určia sa hodnoty parametrov. Pri práci s parametrami je tu avšak zásadný rozdiel, pretože parametrom volania funkcie môže byť aj ďalšie volanie funkcie. Z toho dôvodu sú v tomto prípade parametre k funkcii priradené rekurzívnym spôsobom. Volanie funkcie taktiež môže obsahovať aritmetický výraz. Ako už bolo spomenuté, výraz sa čiastočne vyhodnocuje priamo pri vyhodnocovaní parametrov ak sa jedná o číselné hodnoty. Ak výraz obsahuje volanie funkcie, vyhodnotí sa iba čiastočne a uloží sa taktiež do tabuľky symbolov.
- **Operácia priradenia** – operácia kedy je premennej priradená hodnota. Najjednoduchší variant nastáva v prípade, ak je premennej priradená hodnota alebo konštanta, kedy sa tieto symboly uložia zapuzdrené do tabuľky príkazov. Komplikácie nastavajú v prípade, že hodnota na pravej strane výrazu je volanie funkcie alebo výraz. Vtedy sa postupuje tak, ako bolo uvedené v predchádzajúcom bode.

Po prijatí všetkých tokenov a vykonaní redukcií, ktoré vedú do stavu, kedy sa na vrchole zásobníka nachádza štartovací symbol, je syntakticko-sémantická analýza úspešne ukončená a naplnená tabuľka funkcií sa predáva ďalej na spracovanie. Formálne informácie o práci zásobníkového automatu boli čerpané z [1].

## 3.6 Bezpečnostný analyzátor

Činnosť *bezpečnostného analyzátora* prichádza na rad, až keď je zdrojový program kompletne sémanticky vyhodnotený a tabuľka funkcií je naplnená odpovedajúcimi hodnotami. Bezpečnostný analyzátor pracuje priamo s naplnenou tabuľkou. Na základe hodnôt tabuľky sa snaží odhaliť potenciálne nebezpečné miesta v programe, konkrétne sa špecializuje na miesta zraniteľné voči pretečeniu pamäti, formátovacím reťazcom a pretečeniu čísel.

### 3.6.1 Princíp priebehu odhaľovania problémov

Základný princíp analýzy je postavený na snahe čiastočne simulovať beh testovaného programu. Nejedná sa len o jednoduché porovnávanie veľkostí bufferov, s ktorými sa pracuje vo funkciách, ale na vyhľadávanie problémov sa využíva aktuálna obsadenosť bufferu, ktorá je odvodená z predpokladaného behu testovaného programu. Simulácia behu je postavená na základe volania funkcií. V prvej fáze vykonávania behu sa analyzátor v tabuľke funkcií

pokúsi vyhľadať hlavnú funkciu `main()` a začne vykonávať jej kód. Kódom rozumieme čiastkovú tabuľku príkazov, ktorú obsahuje tabuľka funkcií ku každej uloženej položke. Jej vykonávanie prebieha sekvenčne tak, ako boli príkazy postupne do nej vkladané. Počas vykonávania príkazov sa v prípade nájdenia problémov ukladajú údaje o nájdených problematických miestach do tabuľky chýb spracovávanej funkcie. V prípade úspešného dobehnutia funkcie `main()`, je činnosť bezpečnostného analyzátora ukončená a modifikovaná tabuľka funkcií je pripravená na ďalšie spracovanie modulom určeným na spracovanie chýb.

V nasledujúcich podkapitolách budú stručne opísané významné vlastnosti bezpečnostného analyzátora a problémy spojené s jeho implementáciou.

### 3.6.2 Práca s funkciami

Práca s funkciami tvorí nosnú časť bezpečnostného analyzátora. Najproblematickejšou sekciou je predávanie návratovej hodnoty z funkcie a predávanie parametrov, či už hodnotou alebo odkazom. Problémom je taktiež rekurzívne volanie jednej funkcie, pretože podmienku konca rekurzie nie je možné v statickej analýze komplexne zistiť.

Predávanie parametrov a návratovej hodnoty bolo vyriešené použitím lokálneho *zásobníka*, do ktorého sú uložené odkazy na zapuzdrené premenné a následne po prepnutí kontextu do inej funkcie, sú tieto parametre nakopírované zo zásobníka vo volanej funkcii. Tento princíp umožňuje predávať medzi funkciami ľubovoľný počet parametrov rôznych typov. Obdobný postup je použitý aj pri návrate z funkcie s tým rozdielom, že vo funkcii, do ktorej sa vracia kontext, sa u parametrov určuje, či sa jedná o volanie hodnotou alebo odkazom a na základe tejto vlastnosti sa buď ponechajú hodnoty lokálnych premenných funkcie, alebo je zmenená hodnota premennej na základe hodnoty parametra.

Problém rekurzívneho volania funkcie v testovanej funkcii, ktoré by v analyzátore spôsobilo vykonávanie nekonečného cyklu volania tej istej funkcie, bolo tiež vyriešené využitím lokálneho zásobníka, keď sa do neho ukladá postupnosť volaní funkcie. V prípade zistenia spomenutého rekurzívneho volania funkcie na zásobníku je ďalšie volanie zrušené a pokračuje sa ďalej vo vykonávaní programu.

Špeciálnym typom funkcií sú známe funkcie nedefinované priamo v programe programátorom, ale funkcie zo štandardnej knižnice jazyka C, u ktorých môže prísť k pretečeniu v pamäti, s ktorou pracujú. Na práci s týmito funkciami je postavená analýza pretečenia pamäti.
Spomenuté kritické funkcie a spôsob ich kontroly si popíšeme:

- `scanf()` – funkcia čítajúca dáta zo štandardného vstupu, ukladajúca ich na základe zadaných parametrov do odpovedajúcich premenných. Pre nás je najzásadnejší parameter `%s` vo formátovacej časti funkcie, ktorý sa snaží do odpovedajúcej premennej, zadaného ako ďalší parameter funkcie, načítať reťazec zakončený bielym znakom. Pri spracovaní tejto funkcie v analyzátore je kladený dôraz najmä na správne nastavenie vlastností premenných zadaných ako parametre. V prípade parametra `%s` je hlásené pretečenie, pretože môže byť na vstup zadané ľubovoľné množstvo znakov. Na druhej strane, ak je zadaný parameter `%Xs` kde $X$ je číselná hodnota udávajúca maximálny počet čítaných znakov, porovnáva sa tento počet s veľkosťou pridelenej pamäti odpovedajúcej premennej, do ktorej sa majú dáta uložiť.

- `gets(), fgets()` – funkcie načítavajúce reťazce, ukončené koncom riadka alebo znakom konca súboru, zo štandardného vstupu, prípadne v prípade `fgets()` zo súboru.

Ako jeden z parametrov prijímajú maximálnu dĺžku načítavaného reťazca. Táto hodnota sa porovnáva s veľkosťou pamäti, ktorá je priradená premennej zadanej ako ďalší parameter. Týmto spôsobom môžeme zistiť, či pri volaní týchto funkcií môže dôjsť k pretečeniu.

- `printf(),fprintf()` – funkcie slúžiace na zápis hodnôt na výstup programu, či už štandardný výstup v prípade `printf()`, alebo výstup do súboru v prípade `fprintf()`. Ako bolo spomenuté v kapitole 2.2, môže v týchto funkciách prísť k útoku pomocou formátovacích reťazcov. Analyzátor pri kontrole týchto funkcií kontroluje, či je zadaný formátovací reťazec správne zadaný ako *konštanta*. Ak je zadaný formátovací reťazec vo forme *premennej*, je hlásené potenciálne nebezpečné miesto v programe.

- `strcpy(), strncpy()` – funkcie slúžiace na kopírovanie reťazcov medzi dvoma blokmi pamäti. Prvý variant sa snaží kopírovať celý zdrojový reťazec do pamäte určenej ukazateľom zadaným ako druhý parameter vrátane koncového znaku `/0`. Druhý variant kopíruje iba prvých $N$ znakov zdrojového reťazca, kde $N$ je určené ako tretí parameter. Pri tomto kopírovaní môže taktiež dôjsť k pretečeniu pamäti, ak je objem kopírovaných dát väčší ako veľkosť cieľového bloku pamäti. Zisťovanie problematických miest v tomto prípade prebieha porovnávaním veľkostí cieľovej pamäti s aktuálnou obsadenosťou zdrojovej pamäti. Tiež sa berie do úvahy aj príznak pretečenia v zdrojovej premennej. V tejto funkcii sa pracuje aj s posunmi ukazateľov, keď nie je pamäť kopírovaná od svojho začiatku. Podrobnosti o posunoch ukazateľov budú uvedené v kapitole 3.6.5.

- `strcat(), strncat()` – funkcie slúžiace na zreťazenie dvoch reťazcov. Podobne ako u funkcií `strcpy()` a `strncpy()` prijímajú ako parametre zdrojový reťazec a ukazateľ na cieľový reťazec. Vo variante `strncpy()` je ďalším parametrom počet znakov zo zdrojového reťazca, ktoré sa budú kopírovať. V prípade nájdenia tejto funkcie sa porovnáva veľkosť cieľovej pamäte so súčtom aktuálneho využitia tejto pamäte a objemom kopírovaných dát. Výsledok môže ovplyvniť aj príznak pretečenia, či už v zdrojovej alebo cieľovej premennej. Pri práci s týmito funkciami taktiež môže dochádzať k posunu ukazateľov v parametroch, ktoré bude podrobnejšie popísané v kapitole 3.6.5.

V programoch sa objavujú aj funkcie, ktoré nepatria medzi kritické, avšak v analyzátore je potrebné získať ich návratové hodnoty, a preto sa k nim pristupuje špecificky.
Tieto funkcie si stručne popíšeme:

- `strlen()` – funkcia zisťujúca dĺžku reťazca. V analyzátore sa ako návratová hodnota tejto funkcie považuje hodnota obsadenosti premennej zadanej ako parameter. Hodnota tejto funkcie sa vyhodnocuje až dynamicky počas behu bezpečnostnej analýzy, pretože v čase syntakticko-sémantickej analýzy nie je možné určiť hodnotu obsadenosti premennej.

- `malloc()` – funkcia umožňujúca dynamickú alokáciu pamäte na halde (popis práce s haldou bol uvedený v kapitole 1.2.2). Návratová hodnota tejto funkcie predstavuje v analyzátore veľkosť, ktorá bola alokovaná. Táto veľkosť je následne priradená premennej pre ktorú alokácia prebehla.

- `calloc()` – funkcia obdobná s funkciou `malloc()`, preto sa pri jej spracovaní k nej pristupuje podobným spôsobom. Špecializuje sa na dynamickú alokáciu polí, s tým

rozdielom oproti funkcii `malloc()`, že po alokácii pamäte túto pamäť aj vynuluje. Po jej spracovaní je premennej nastavená veľkosť vypočítaná na základe parametrov tejto funkcie.

- `realloc()` – funkcia podobná s predchádzajúcimi dvoma spomenutými prípadmi. Slúži na zmenu veľkosti už alokovanej pamäte. Po jej spracovaní sa nastaví nová veľkosť premennej zadanej ako parameter, prípadne premennej, do ktorej sa priraďuje návratová hodnota tejto funkcie.

Nasledujúca kapitola v stručnosti popísala zavedené prvky dynamického vyhodnocovania volania funkcií do statickej analýzy programu, čo by malo zabezpečiť presnejšie vyhodnotenie chýb v testovanom programe a menej falošných detekcií.

### 3.6.3 Práca s premennými

Správne a čo najpresnejšie vyhodnocovanie hodnôt premenných je v analyzátore taktiež jedným z najdôležitejších prvkov. Ich hodnota pri ďalšom spracovaní behu simulovaného programu určuje, či môže alebo nemôže dôjsť k pretečeniu.

Ako už bolo podrobnejšie spomenuté v kapitole 3.3.1, údaje udržiavané o premennej uchovávajú jej veľkosť, hodnotu využitia premennej a príznak pretečenia. V prípade vykonávania operácie priradenia, prípadne vykonávanie funkcie, ktorá manipuluje s pamäťou, sa na základe parametrov tejto operácie nastavia údaje premennej na príslušné hodnoty. Tieto hodnoty môžu byť v ďalšom behu programu opäť dynamicky menené. Z toho vyplýva, že sa bude vždy používať na vyhodnocovanie zraniteľnosti programu predpokladaná aktuálna hodnota premennej.

Ďalším dôvodom kontroly operátora priradenia, je detekcia chýb typu pretečenie čísel, ktoré boli popísané v kapitole 2.3. V analyzátore sa konkrétne sústredíme na chyby spôsobené nesprávnou konverziou znamienka. To znamená, že je nutné vykonať určitý druh typovej kontroly pri priradzovaní čísiel. Podrobnosti o problematike detekcie pretečenia čísel budú uvedené v kapitole 4.1.3.

### 3.6.4 Práca s neznámymi hodnotami

Neznáme hodnoty premenných v statickej analýze, kedy nepoznáme vstup programu, tvoria ďalší problém pri implementácii. V tomto prípade problém nastáva napríklad pri čítaní reťazca do pamäti, pretože v čase vykonávania analýzy nie je známe, čo bude pri samotnom behu programu zadané.

Pri zistení tohto problému sa analyzátor snaží určiť aspoň maximálnu možnú hodnotu, ktorú je možné zadať a túto hodnotu nastaviť následne ako aktuálnu hodnotu obsadenosti pre danú premennú. Ak nastane situácia, že dôjde k možnosti neobmedzeného počtu zadaných znakov, je ohlásené pretečenie a premennej je nastavený odpovedajúci príznak. Tento príznak má potom vplyv pri ďalšom spracovaní premennej.

Komplikácie pri vývoji analyzátora taktiež tvorí viacnásobné použitie príkazov `return` doplnených podmienenými príkazmi v užívateľských funkciách, pretože presne určiť, ktorá hodnota sa bude vracať nie je možné, preto je použitý prístup, kedy zaznamenajú všetky návratové hodnoty funkcie a po jej skončení sa vyberie hodnota, ktorá predstavuje najhoršiu možnosť pre daný program.

### 3.6.5 Posun ukazatelov

Jazyk C ukladá reťazce ako polia znakov, ku ktorým sa pristupuje cez ukazatel na prvý prvok. Tento princíp umožňuje jednoduchý prístup k celému reťazcu pomocou pripočítania aritmetickej hodnoty k ukazatelu. Táto vlastnosť sa pri práci s reťazcami pomerne často využíva. Z tohto dôvodu bolo vyčíslovanie posunu ukazateľa implementované aj v analyzátore. Vďaka tomuto prístupu je možné znížiť počty falošných detekcií.
Princíp ukážeme na príklade:
Máme dva reťazce:

```
char buffer1[20], buffer2 [10]
```

A volanie funkcie `strcpy()`:

```
strcpy(buffer2,buffer1 + 11)
```

V prípade vyhodnotenia bez posunu by bola hlásená možnosť pretečenia, pretože do pamäti o veľkosti 10 bajtov sa kopíruje maximálne 20 bajtov. Avšak v skutočnosti sa kopíruje maximálne 9 bajtov a k pretečeniu nemôže nikdy dôjsť.

Vďaka rekurzívnemu návrhu štruktúry ukladania príkazov do tabuľky funkcií je možné v tomto prípade vyčíslovať aj zložitejšie výrazy obsahujúce volania funkcií a napríklad aj aritmetické operácie.

## 3.7 Modul výstupu

Modul výstupu je modul spracovávajúci tabuľku nájdených chýb, uloženú pre každú funkciu v tabuľke funkcií. Na základe parametrov uložených v tejto tabuľke vygeneruje výstup prijateľnejší pre užívateľa s popisom typu chyby, riadku a názvy súboru, kde sa chyba nachádza a triedu nájdeného problému.

Príklady výstupov pre rôzne vstupy je možné nájsť v prílohe C.

## 3.8 Ukážka činnosti analyzátora

Činnosť analyzátora si predvedieme na nasledujúcej krátkej ukážke kódu testovaného programu, na ktorom si popíšeme činnosti všetkých modulov a ich kooperáciu.
Príklad kódu:

```
char buffer1[20],buffer2[15];
fgets(buffer1,sizeof(buffer1),stdin);
strcpy(buffer2,buffer1);
```

Tento modelový kód vykonáva jednoduchú činnosť, kedy načíta zo vstupu dáta do premennej `buffer1` a následne ich prekopíruje do premennej `buffer2`. Takýto jednoduchý príklad bol zvolený, na jednoduchšie ozrejmenie základných princípov činnosti, ktorú si teraz popíšeme.

V prvom kroku analýzy je spustený syntakticko-sémantický analyzátor, ktorý začne od lexikálneho analyzátora postupne žiadať tokeny. Prichádzajúci prúd tokenov je na základe pravidiel prechodov a akcií redukovaný na zásobníku. Konkrétne prvý riadok analyzátor vyhodnotí ako definíciu dvoch *premenných*, ktoré následné vloží do tabuľky symbolov a nastaví im odpovedajúce veľkosti.

Druhý riadok modelového príkladu sa na zásobníku zredukuje ako volanie *funkcie*. Akcia odpovedajúca tejto redukcii vloží novú položku do tabuľky príkazov, ktorý bude obsahovať údaje o volaní funkcie `fgets()` a všetky jej parametre.

Tretí riadok sa vyhodnotí taktiež ako volanie funkcie, konkrétne `strcpy()` a analogicky ako v predchádzajúcom prípade sa uloží do tabuľky príkazov. Týmto je činost lexikálneho a syntakticko-sémantického analyzátora ukončená.

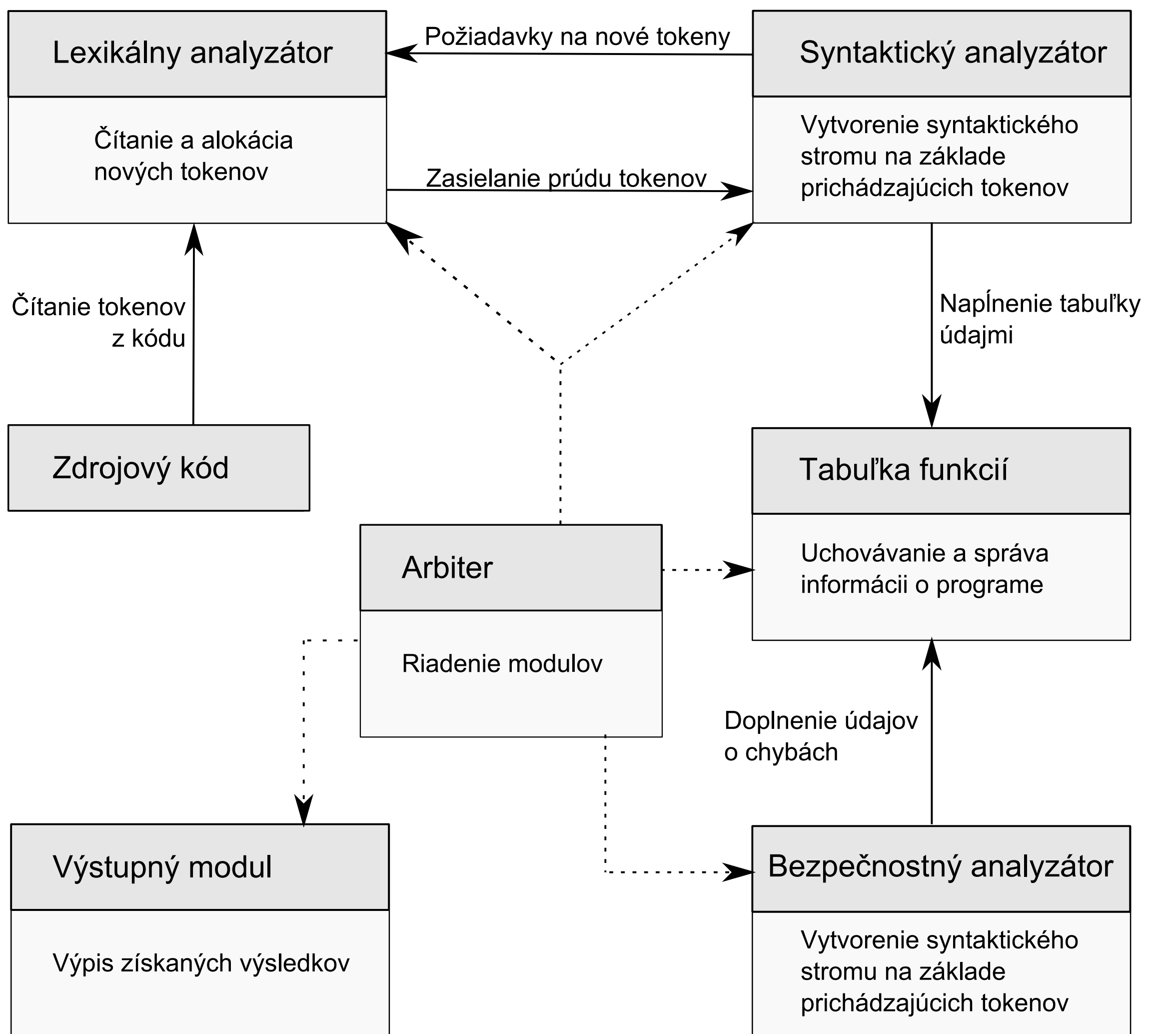
Na rad prichádza bezpečnostný analyzátor, ktorý začne sekvenčne prechádzať tabuľku príkazov funkcie, v našom prípade obsahujúcu dva príkazy. Prvý príkaz predstavuje volanie funkcie `fgets()`. Analyzátor preskúma a vyhodnotí, či sú parametre vyhovujúce a spĺňajú syntaktické pravidlá jazyka C. Následne začne parametre spracovávať, ako prvý krok získa z tabuľky symbolov ukazatel na záznam premennej `buffer1`, s ktorým funkcia pracuje. Druhý parameter, predstavujúci operátor `sizeof`, nemohol byť vyhodnotený počas sémantickej analýzy a vyhodnotí sa dynamicky až teraz pri spracovaní príkazu. Keďže analyzátor vykonáva statickú, analýzu, kde nepoznáme vstupy programu, nastaví sa veľkosť používaných dát premennej `buffer1` na maximálnu možnú veľkosť, ktorá je v tomto prípade rovná celkovej veľkosti pridelenej pamäti.

Posledný príkaz v tabuľke predstavuje volanie funkcie `strcpy()`. Analyzátor opäť získa z tabuľky symbolov záznamy o oboch premenných, zadaných ako parametre a na základe pravidiel dosadením veľkostí premenných vypočíta, či môže dôjsť k pretečeniu. V našom prípade to možné je, tak analyzátor vloží nový záznam do tabuľky chýb, s údajmi o umiestnení a triede chyby. Keďže tabuľka príkazov, je po vykonaní týchto operácií prázdna, končí tým činnosť aj bezpečnostného analyzátora.

Posledný krok celej analýzy je vypísanie detailov o nájdených chybách v prehľadnejšej forme na štandardný výstup programu. Následne dôjde už len k dealokácii použitých zdrojov a ukončeniu programu.

## 3.9 Zhrnutie

V tejto kapitole boli stručne popísané implementačné detaily a návrh analyzátora. Štruktúra návrhu celého programu je symbolicky znázornená na obrázku číslo 3.4, kde sú vyobrazené všetky moduly a znázornená komunikácia medzi nimi.




Obrázek 3.4: Kompletná schéma navrhnutého analyzátora

# Kapitola 4

# Testovanie

Nasledujúca kapitola sa venuje samotnému testovaniu programu. Testovanie by malo dokázať, že program bol navrhnutý a implementovaný správne a v našom prípade, či korektne vyhľadáva všetky triedy chýb. Za týmto účelom bolo potrebné vytvoriť testovaciu sadu krátkych programov pre každú skupinu chýb, ktoré by sme chceli v testovaných programoch vyhľadávať. Tento spôsob testovania musel byť zvolený, pretože okamžité testovanie na rozsiahlych programoch by bolo neefektívne a mnohé chyby nachádzajúce sa v nich by neboli odhaliteľné jednoduchým preskúmaním programu programátorom a preto by sa ťažšie ladili algoritmy na ich vyhľadávanie. Z tohto dôvodu bolo samotné testovanie rozčlenené na dve fázy. V prvej fáze prebiehalo testovanie na sade modelových príkladov a následne boli testované rozsiahlejšie programy tretích strán. Jednotlivé fázy si popíšeme podrobnejšie.

## 4.1 Testovanie na vlastnej sade programov

Vlastná zbierka programov vytvorená špeciálne na účely testovania analyzátora mala dopomôcť k počiatočnému testovaniu a k vyladeniu detekcií jednotlivých chybových konštrukcií. Samotné programy boli vytvárané postupne a začínajú jednoduchšími chybami, ktoré sú neskôr v ďalších programoch rozširované, na základe zvyšovania nárokov na analyzátor. Jednotlivé triedy chybových detekcií si na jednotlivých programoch popíšeme v nasledujúcich podkapitolách. Celé zdrojové kódy nebudú kvôli ich rozsiahlosti uvádzané a je ich možno dohľadať v prílohe C respektíve na priloženom CD. Výstupy analyzátora budú uvedené tiež iba v skrátenej forme, kompletné výstupy testovania sú taktiež priložené v prílohe C.

### 4.1.1 Pretečenie pamäti

Trieda chýb typu pretečenie pamäti je v analyzátore podporovaná v najväčšej miere, pretože sa v programoch objavujú v najväčšom množstve. Z tohto dôvodu sa tejto problematike venuje až päť vzorových príkladov. Teraz si ich postupne popíšeme.

Program `example1.c` je najjednoduchší z programov. Obsahuje potenciálne pretečenie pamäti, nachádzajúce sa vo volaní funkcie `scanf()`.
Samotná kritická konštrukcia z programu:

```
scanf("%s",buffer);
```

Táto operácia sa snaží načítať reťazec neobmedzenej dĺžky zo štandardného vstupu programu do premennej `buffer`. Keďže dĺžku reťazca, ktorý zadá užívateľ nepoznáme, predpo-

kladáme, že môže zadať nekonečné množstvo znakov a dôjde k pretečeniu. Túto konštrukciu rozpozná bez najmenších problémov a označí ju sa miesto potenciálneho nebezpečenstva.

Druhý program examle2.c obsahuje taktiež pretečenie pamäti v čiastočne zložitejšej forme. Konkrétne sa jedná o pretečenie pri použití funkcie strcpy(). Uvedieme si problematické riadky tohto súboru potrebné na ďalší výklad:

```
char buffer1[20],buffer2[15];
strcpy(buffer2,buffer1);
```

Kritickou častou je vykonávanie funkcie strcpy(), kedy do pamäti s prideleným miestom o veľkosti 15 bajtov, môže byť kopírovaných až 20 bajtov. Táto konštrukcia je označená za chybnú iba v prípade, že sa na základe hodnôt parametrov zistí, že premenná buffer1 v čase kopírovania môže používať viac ako 15 bajtov, ktoré sa pri kopírovaní zmestia do pamäti premennej buffer1. To znamená, že výsledok tejto akcie závisí na kóde programu nachádzajúceho sa pred touto konštrukciou. V našom prípade programu example2.c, môže premenná buffer1 nadobudnúť maximálnu veľkosť, čo vedie k zahláseniu chyby analyzátorom.

Ďalší program venujúci sa pretečeniu pamäti example3.c, ktorý je opäť o niečo pokročilejší, pretože obsahuje kritické funkcie až vo volaní funkcie definovanej v programe. Samotný kód užívateľskej funkcie je kontrolovaný, až v momente jeho volania. To znamená, že jej kód bude vyhodnotený, až na základe aktuálnych hodnôt predaných funkcii ako parametre. V našom prípade predstavuje kritickú časť tento úsek:

```
strcpy(tmp_buffer,str1);
strcat(tmp_buffer,str2);
strcpy(str,tmp_buffer);
```

Tento modelový príklad vykonáva jednoduché spojenie dvoch reťazcov a výsledok kopíruje z dočasného pamäťového miesta do premennej určenej na uchovanie výsledku. Princíp detekcie je podobný ako v predošlom prípade, hlavným cieľom tohto príkladu je demonštrácia funkčnosti analyzátora pri detekcii problémov v užívateľských funkciách.

Modelový program číslo štyri predstavuje súbor example4.c dopĺňaný dvoma súbormi externého modulu example4lib.c a example4lib.h. Program vykonáva podobnú činnosť ako predchádzajúci príklad s tým rozdielom, že užívateľská funkcia sa nenachádza v jednom súbore s hlavnou funkciou programu. Ukázanie tejto vlastnosti analyzátora je aj hlavnou úlohou tohto príkladu. Analyzátor po zistení vloženia *externého hlavičkového súboru* zanalyzuje aj súbor odpovedajúci tomuto súboru a načíta z neho funkcie do tabuľky funkcií. Samotný výsledok analýzy je podobný s predošlým príkladom.

Posledným modelovým príkladom orientovaným na pretečenie pamäti je example5.c. Tento krátky kód je určený na demonštráciu funkčnosti predávania návratovej hodnoty funkcie. Program vykonáva volanie užívateľskej funkcie dva krát, kedy sa pri druhom volaní použije ako parameter volania návratová hodnota z predchádzajúceho volania. Pri prvom volaní k pretečeniu dôjsť nemôže, avšak keď je zavolaná opäť so zmenenými parametrami, je hlásené varovanie pred potenciálnym pretečením.

Táto kapitola sa venovala opisu testovania analyzátora pomocou modelových príkladov orientovaných na pretečenie pamäti. Ako už bolo spomenuté, kompletné zdrojové kódy týchto programov a im odpovedajúce výstupy analyzátora je možné dohľadať v prílohe.

### 4.1.2 Formátovacie reťazce

Formátovacím reťazcom sa venuje iba jeden testovací súbor, konkrétne `example6.c`, pretože jediný variant tejto triedy útokov, odhaliteľný pomocou statickej analýzy je volanie funkcie `printf()`, kde je ako formátovací reťazec predaný priamo reťazec, ktorý chceme vypísať. Druhou čisto teoretickou možnosťou detekcie problémov spojenými s formátovacími reťazcami, by bola kontrola odpovedajúceho počtu parametrov k formátovaciemu reťazcu. Avšak túto možnosť kontroluje už samotný prekladač a v prípade chyby program ani nepreloží. Keďže vstupom analyzátora je preložiteľný program, táto možnosť nemôže pri analýze nikdy nastať.

Kritickou sekciou, ako už bolo spomenuté, je konkrétne v tomto príklade riadok s volaním funkcie `printf(buffer)`. Tento problém je analyzátorom rozpoznaný a označený za nebezpečnú konštrukciu.

### 4.1.3 Pretečenie čísel

Pretečenie čísel patrí do poslednej triedy, ktorý analyzátor vyhľadáva v zdrojových kódoch testovaných programov. Ako už bolo spomenuté v predchádzajúcich kapitolách analyzátor vykonáva statickú analýzu, doplnenú o určité dynamické vlastnosti. Z tohto dôvodu je možné pri analýze pretečenia čísel rozpoznávať iba pretečenia založené na chybe *konverzie znamienka*, popísanej v kapitole 2.3. Na detekciu ostatných druhov pretečení by bola nutná kompletná dynamická analýza. Ako je uvedené v [2] nástroj *RICH*, slúžiaci na detekciu problémov spôsobených s pretečením čísel a ich automatickou opravou, pracuje priamo v kompilátore na základe dynamických dát. Konštrukcie rozpoznávané vytvoreným analyzátorom si popíšeme.

Pre potreby vytvoreného analyzátora boli vytvorené dva testovacie programy prvý `example7.c` je určený na demonštráciu funkčnosti vyhľadávania chýb, pri ktorých dochádza k priradzovaniu čísel a premenných s rozličnými znamienkami.

V programe sú definované dve premenné `unsigned int a` a `int b`. Následne dochádza k rôznym priradeniam. Konštrukcie ako napríklad `b = -10` a `a = b` sú následne označené ako zdroj možnej chyby konverzie znamienka.

Druhý modelový príklad `example8.c` je vytvorený na demonštráciu kontroly pri priradzovaní výsledku volania funkcie do premennej. V tomto prípade je taktiež vykonávaná typová kontrola, zisťujúca či môže dôjsť k pretečeniu spôsobeného konverziou znamienka.

Zdrojové kódy a odpovedajúce výstupy je možné taktiež vidieť v prílohách.

### 4.1.4 Komplexný test

Na záver testov s modelovými programami uvedieme testovací program `example9.c`, ktorý je jednoduchým príkladom na overenie pokročilých vlastností analyzátora. Program obsahuje všetky tri triedy chýb vyhľadávaných analyzátorom a jeho kompletný kód sa nachádza v prílohe C. Výsledky analýzy tohto programu si popíšeme postupne po krokoch.

V prvej časti behu prebieha definícia a inicializácia premenných, sú definované dva reťazce `buffer1` a `buffer2`, oba dĺžky 50 bajtov, ktoré sú funkciou `scanf()` naplnené na maximálnu dĺžku.

Následne je zavolaná funkcia `scat()`, ktorej návratová hodnota bude priradená do premennej `result`.

Vo funkcii `scat()` sa vytvoria dva pomocné reťazce `tmp_buffer` a `tmp_buffer1` alokovaním dynamicky na halde. Pri alokácii je použité volanie:

```
char *tmp_buffer =(char *)malloc(2*sizeof(char)*strlen(str1))
```
respektíve
```
char *tmp_buffer1 =(char *) malloc(4*sizeof(char)*strlen(str2))
```

Spracovanie tohto volania predstavuje pokročilú vlastnosť analyzátora, kedy sa veľkosť alokovaných dát počíta z časti už počas sémantickej analýzy, kedy je vyčíslená počiatočná časť výrazu a kompletné vyčíslenie zahrňujúce volanie funkcie `strlen()` je zavŕšené dynamicky počas samotnej bezpečnostnej analýzy. V našom prípade sa nastaví veľkosť premennej `tmp_buffer` na 100 bajtov a veľkosť premennej `tmp_buffer1` na 200 bajtov. Týchto 200 bajtov sa v nasledujúcich 4 krokoch aj zaplní.
Toto tvrdenie je dokázané nasledujúcim volaním:

```
strncpy(tmp_buffer,tmp_buffer1,99);
strncpy(tmp_buffer,tmp_buffer1,101);
```

ktoré testuje, správne vyčíslenie týchto veľkostí, v prvom prípade sa program pokúša nakopírovať do premennej `tmp_buffer` 99 bajtov dát, táto operácia prebehne bez problému, bez hlásenia chyby, v druhom prípade pri pokuse nakopírovania 101 bajtov dát k pretečeniu už dôjde. To znamená, že premenná ma skutočne pridelenú veľkosť 100 bajtov, vďaka správnemu vyčísleniu výrazu.
Další krok predstavuje operácia

```
if (strcmp(tmp_buffer,tmp_buffer1) != 0)
{
  return tmp_buffer1;
}
return tmp_buffer;
```

Táto konštrukcia taktiež pri statickej analýze predstavuje problém, pretože podmienený výraz nemôže byť vyhodnotený a nemôže byť určené, ktorá návratová hodnota sa bude z funkcie vracať. V tomto prípade sa použije princíp popísaný v kapitole 3.6.4, kedy je z funkcie vrátený najhorší variant. V našom prípade premenná `tmp_buffer1`, o veľkosti 200 bajtov. Tvrdenie je potvrdené nasledujúcim volaním funkcie `test()`, kedy sa do premennej o velkosti 150 bajtov pokúsi funkcia `strcpy()` nakopírovať návratovú hodnotu funkcie `scat()` – teda obsah premennej `tmp_buffer1`. Pri tejto operácii samozrejme dochádza k pretečeniu. Pri teste tohto programu sú taktiež odhalené aj 2 ďalšie chyby, ktoré program obsahuje. Kompletný výstup testu sa nachádza v prílohe.

### 4.1.5 Zhrnutie

Táto kapitola uviedla základný popis jednoduchých programov používaných na testovanie analyzátora pri vývoji detekcií všetkých druhov chýb. Pri testoch na tejto sade programov bol analyzátor pomerne úspešný, pretože sa jednalo o programy s priamočiaro vloženými chybami. Programy ale splnili svoj účel, pretože aspoň z časti ukázali možnosti samotného analyzátora pri vyhľadávaní chýb.

Analyzátor avšak nie je obmedzený iba na spomenuté jednoduché programy, ale dokáže pracovať aj s rozsiahlymi, bežne používanými programami, s malými obmedzeniami, ktoré budú uvedené v nasledujúcej kapitole.

## 4.2 Testovanie na programoch tretích strán

Testovanie na programoch nevytvorených priamo za účelom vývoja a testovania analyzátora, by malo dokázať, že analyzátor sa dokáže vyrovnať aj zo vstupnými konštrukciami, s ktorými sa pri návrhu a vývoji spočiatku nemuselo počítať. Analyzátor bol otestovaný na väčšom množstve programov, avšak väčšina z nich bola buď kvalitne napísaná, prípadne obsahovala chybové konštrukcie, ktoré nebolo možné rozpoznať pomocou zvoleného typu statickej analýzy.

Zo všetkých testovaných programov zdôrazníme tri konkrétne programy, ktoré sú taktiež umiestnené na CD prílohe a v tejto kapitole si ich popíšeme.

Prvým, najjednoduchším, programom je kód v súbore `snipplr.c`, ktorý predstavuje krátky modelový kód pretečenia pamäti. Ako potenciálne nebezpečná konštrukcia je analyzátorom označená `strcpy(buffer1, argv[1])`. Kde `buffer1` má pridelenú veľkosť 5 a keďže analyzátor považuje vstupný parameter programu za potenciálne nekonečný, je tento úsek považovaný za nebezpečný.

Ďalším testovaným programom bol program `binStromy.c`, demonštrujúci prácu s AVL binárnymi stromami. Pri analýze tohto programu bola tiež nájdená jedna chybná konštrukcia, konkrétne `scanf("%s",vstup)`, kedy sa zo vstupu môže načítať ľubovoľný počet znakov do premennej obmedzenej veľkosti.

Posledným spomenutým príkladom je `demo.c`, predstavujúci pomerne zložitý demonštračný príklad o pretečení pamäti. Tento príklad tvorí najkomplexnejší uvedený test analyzátora. V tomto programe analyzátor označil ako chybné 3 konštrukcie:

- `repeat = (int)rep` – kde `repeat` je číslo bez znamienka a `rep` je číslo typu `double`. V tomto prípade môže dôjsť k chybnej konverzii znamienka, prípadne k inému druhu číselného pretečenia.
- `strcpy(filename, argc[i])` – analyzátor v tomto prípade označil túto konštrukciu za nebezpečnú, pretože považuje vstupný parameter programu za neobmedzený. Avšak po nahliadnutí do kódu programu, je vidieť, že program má tento aspekt ošetrený a jedná sa o falošné hlásenie, typické pre statickú analýzu programov.
- `strcat(filename,".img")` – nasledujúca konštrukcia už avšak môže spôsobiť zápis mimo pridelenú pamäť, pretože kontrola dĺžky vstupného parametra z predchádzajúceho príkazu neberie do úvahy predĺženie reťazca o ďalšie štyri znaky, preto je toto hlásenie analyzátora oprávnené.

Táto kapitola pojednávala o testovaní analyzátor na programoch tretích strán. Na základe tohto testovania boli zistené niektoré jeho obmedzenia, ktoré budú popísané v ďalšom texte. Taktiež ukázala niektoré jeho slabé stránky, ako napríklad nedostatočné vyhodnocovanie podmienok v príkazoch a cykloch, čo môže viesť k falošným hláseniam.

### 4.2.1 Obmedzenia zistené pri testovaní

Pri testovaní bolo odhalených niekoľko obmedzení, ktorých odstránenie by bolo neúmerne náročné a samotné výsledky analýzy by sa zlepšili iba minimálne. Hlavným obmedzujúcim faktorom je syntakticko-sémantický analyzátor, pretože pre jeho činnosť je potrebné poznať kompletnú gramatiku analyzovaného jazyka, s ktorým pracuje. Navrhnutý analyzátor je stavaný na normu ANSI jazyka C, ktorú však niektoré programy nesplňujú.

Ďalším problémom syntaktického analyzátora je práca s užívateľsky definovanými typmi, ktoré sú preto v programoch preskakované a musí dochádzať k zotaveniu syntaktickej analýzy. Podobný problém nastáva aj pri užívateľských makrách určených pre preprocesor. Do tejto kategórie spadá aj napríklad používanie ukazateľov na funkcie a ich predávanie ako parametrov.

Veľkým problémom je aj vyhodnocovanie podmienených príkazov a nemožnosť odhaliť úseky kódu, ktoré sa pri behu skutočného programu nevykonajú. V našom prípade sú z analýzy vyradené, iba úseky nachádzajúce sa v užívateľských funkciách, ktoré nie sú v programe volané. Z týchto poznatkov je možné odvodiť, že najväčším limitujúcim faktorom je sémantická analýza.

### 4.2.2 Možnosti ďalšieho vývoja

Ďalšie možnosti rozvoja vytvoreného analyzátora sú pomerne široké. K rozšíreniu môže prísť vo všetkých moduloch, pretože možnosti jazyka C, ktorý je vstupom programu sú veľmi rozsiahle.

K najvýznamnejšiemu vylepšeniu aplikácie by došlo rozšírením sémantickej analýzy, pretože tá sa v súčasnej dobe špecializuje hlavne na kritické konštrukcie v programoch. Najlepšou voľbou by bolo vytvorenie kompletnej sémantickej analýzy, ktorá by následne umožňovala kompletnú interpretáciu zdrojového kódu, zvýšil by sa podiel nájdených chýb a znížil by sa počet falošných hlásení. Kompletná interpretácia by umožnila napríklad kompletné vyhodnocovanie podmienok a detekciu kódu ktorý sa nevykonáva.

Možnostiam na ďalší rozvoj je otvorený aj bezpečnostný analyzátor, do ktorého je možné jednoducho pridávať obsluhu spracovania funkcií, prípadne celých konštrukcií. Tu sa naskytá aj možnosť pridávania pravidiel na detekciu rozličných typov útokov, akým sa v súčasnosti analyzátor venuje.

K rozvoju by mohlo taktiež prísť aj v ostaných moduloch analyzátora, ktoré by avšak mali menší vplyv na počty správnych a falošných hlásení. Mohli by avšak zvýšiť efektivitu vykonávania programu a zoptimalizovať vyhľadávanie kritických sekcií programu.

### 4.2.3 Prostredie pri testovaní

Na testovanie bol potrebný počítač s nainštalovaným prekladačom jazyka *C++*, na korektné preloženie zdrojových kódov aplikácie. V našom prípade prebiehalo testovanie na školskom serveri `Merlin` s nainštalovaným operačným systémom `Linux` a prekladačom `g++` verzie *4.4.6*.

Keďže aplikácia bola implementovaná so snahou na prácu na rôznych platformách, funkčnosť bola otestovaná aj systémoch `Windows` a `Free BSD`, kde program taktiež pracoval korektne.

# Záver

Úlohou tejto práce bolo uviesť teoretické znalosti potrebné k pochopeniu nebezpečných častí programu z hľadiska exploitácie a uviesť čitateľa do tejto problematiky Na základe týchto teoretických znalostí zahrňujúcich okrem princípov exploitovania tiež základy priebehu prekladu programu a prácu s jazykom Assembler, bolo možné vytvoriť návrh a následnú implementáciu analyzátora kódu jazyka C. Analyzátor bol vytvorený v implementačnom jazyku C++, s použitím nástrojov Bison a Flex, ktoré čiastočne zefektívnili jeho tvorbu.

Ďalšie kapitoly práce sa už venujú praktickým aspektom zisteným pri tvorbe analyzátora. Dôraz sa kládol na popis jednotlivých modulov a ich vzájomnej interakcii a na popis najvýznamnejších vlastností jeho činnosti. Medzi tieto vlastnosti napríklad patrí analýza postavená iba na jednoprechodovom načítaní každého súboru, čím sa snaží dosiahnuť vyššiu efektivitu pri spracovaní. Dôležitým prvkom je aj snaha o čiastočnú simuláciu behu analyzovaného programu, založenú na volaní funkcií a predávaní parametrov medzi nimi. Snahou tohto kroku pri návrhu programu, bolo vniesť určité prvky dynamiky do statickej analýzy. Medzi kľúčové vlastnosti sa radí aj určovanie posuvov ukazatelov alebo vyhodnocovanie volaní funkcií zadaných ako parameter inej funkcie. Ich snahou bolo zlepšiť presnosť vyhľadávania a znížiť počet falošných hlásení.

V záverečných častiach textu je čitateľ oboznámený s výsledkami testovania navrhnutej aplikácie. Na základe týchto zverejnených údajov si môže utvoriť obraz o schopnostiach analyzátora. Výsledky ukazujú, že analyzátor môže slúžiť pri vývojoch programov na vyhľadávanie základných programátorských chýb, ktoré môžu znamenať nestabilitu, prípadne až bezpečnostnú hrozbu programov.

V tejto časti je taktiež uvedený popis nedostatkov programu, ktoré či už viac alebo menej zasahujú do kvality analýzy. Za hlavný nedostatok sa považuje určité obmedzenie sémantickou analýzou programov, s ktorým je treba počítať pri jeho používaní.

# Literatura


[1] Beneš, M., Češka, M. a Hruška, T. *Překladače.* 1999. Skriptum VUT Brno.

[2] Brumley, D. et al. *RICH: Automatically Protecting Against Integer-Based Vulnerabilities* [online]. rev. 2007 [cit. 2011-05-02]. Dostupné na: <`http://www.cs.cmu.edu/~dbrumley/pubs/integer-ndss-07.pdf`>.

[3] Degener, J. *ANSI C grammar, Lex specification* [online]. rev. 1995 [cit. 2011-05-01]. Dostupné na: <`http://www.lysator.liu.se/c/ANSI-C-grammar-l.html`>.

[4] Degener, J. *ANSI C Yacc grammar* [online]. rev. 1995 [cit. 2011-05-02]. Dostupné na: <`http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`>.

[5] Donnelly, C. a Stallman, R. *Bison* [online]. rev. August 05, 2010 [cit. 2011-05-02]. Dostupné na: <`http://www.gnu.org/software/bison/manual/`>.

[6] Erickson, J. *Hacking - umění exploitace.* 2. vyd. Brno, CZ: ZONER software, s.r.o, 2009. 544 s. ISBN 978-80-7413-022-9.

[7] Herout, P. *Učebnice jazyka C - 1. díl.* 5. vyd. České Budějovice, CZ: KOPP, 2008. 271 s. ISBN 978-80-7232-351-7.

[8] Marek, R. *Učíme se programovat v jazyce Assembler pro PC.* 1. vyd. Brno, CZ: Computer press, a. s., 2007. 228 s. ISBN 80-7226-843-0.

[9] Paxson, V., Estes, W. a Millaway, J. *The flex Manual* [online]. rev. 10 September 2007 [cit. 2011-05-02]. Dostupné na: <`http://flex.sourceforge.net/manual/`>.

[10] Stallman, R. M. et al. *Using the GNU Compiler Collection* [online]. 2010, rev. 2011-05-09 [cit. 2011-05-10]. Dostupné na: <`http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc.pdf`>.

[11] Vojnar, T. *Formal Analysis and Verification* [online]. rev. 10. leden 2011 [cit. 2011-04-20]. Dostupné na: <`http://www.fit.vutbr.cz/study/courses/FAV/public`>.

# Seznam příloh

Dodatek A Návod na použitie
Dodatek B Obsah CD
Dodatek C Demonštračné programy
Dodatek D Metriky kódu

# Dodatek A

# Návod na použitie

Vytvorený program pracuje ako konzolová aplikácia, preto je jej ovládanie pomerne jednoduché. Pri prvom použití je aplikáciu potrebné preložiť, príkazom `make` spusteným v hlavnej zložke programu, ktorý zabezpečí preloženie zdrojových kódov do spustiteľnej podoby. Vytvorený spustiteľný súbor má názov `canalyzer`.

Program `canalyzer` prijíma štandardne parameter `-h` prípadne `--help`, slúžiaci na výpis nápovede priamo do konzoly.

V prípade nezadania parametra `-h`, sa za jediný parameter považuje relatívna cesta k súboru, ktorý chceme analyzovať. Analyzátor sa tento súbor pokúsi otvoriť a v prípade úspechu sa vykoná kompletná analýza predloženého programu.
Príklad volania programu:

```
./canalyzer examples/ThirdParty/demo.c
```

Ďalšie príklady použitia analyzátora je možné nájsť v prílohe C.

Ďalšie informácie je možné nájsť v súbore `Readme`, prípadne kompletná dokumentácia kódu vygenerovaná nástrojom *Doxygen* sa nachádza v súbore `doc/html/index.html`.

# Dodatek B

# Obsah CD

Zložky:

- **doc** – projektová dokumentácia, vygenerovaná z kódu použitím nástroja *Doxygen.*
- **examples** – zdrojové kódy demonštračných príkladov a programov tretích strán určených na testovanie.
- **grammarFiles** – definičné gramatiky pre nástroje *Bison* a *Flex.*
- **src** – zdrojové kódy analyzátora vytvorené v jazyku C++.
- **tex** – zdrojové kódy textovej časti práce vytvorené pre typografický systém LaTeX.

Súbory:

- **Makefile** – súbor definujúci preklad programu.
- **Readme** – súbor popisujúci základné použitie programu.

# Dodatek C

# Demonštračné programy

I. **example1.c**

```
/*
 example1.c - Číta znaky zo štand. vstupu a mení velké
    pismená na malé.
 Zraniteľnosť: Buffer owerflow
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char *argv)
{
 int i,tmp;
 char buffer[25];
 scanf("%s",buffer);
 for(i=0;i<strlen(buffer);i++)
 {
 putchar(tolower(buffer[i]));
 }
 printf("\n");
 return EXIT_SUCCESS;
}
```

II. **example2.c**

```
/*
 example2.c - Koprírovanie reťazca
 Zraniteľnosť: Buffer owerflow
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>


int main(int argc, char *argv)
{
 char buffer1[20],buffer2[15];
  fgets(buffer1,sizeof(buffer1),stdin);
  strcpy(buffer2,buffer1);
  printf("%s\n",buffer2);
 return (EXIT_SUCCESS);
}
```

III. **example3.c**

```
/*
 example3.c - Konkatenácia reťazcov
 Zraniteľnosť: Buffer owerflow
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void scat(char *str1,char *str2,char *str)
{
    char tmp_buffer[36];
    strcpy(tmp_buffer,str1);
    strcat(tmp_buffer,str2);
    strcpy(str,tmp_buffer);
    return;
}
int main ()
{
    char buffer1[20],buffer2[20],buffer3[35];
    fgets(buffer1,sizeof(buffer1),stdin);
    fgets(buffer2,sizeof(buffer2),stdin);
    scat(buffer1,buffer2,buffer3);
    printf("%s\n",buffer3);
    return 0;
}
```

IV. **example4.c**

```
/*
  example4.c - Konkatenácia reťazcov s použitím modulov
  Zraniteľnosť: Buffer owerflow
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "example4lib.h"

int main ()
{
    char buffer1[20],buffer2[20],buffer3[35];
    fgets(buffer1,sizeof(buffer1),stdin);
    fgets(buffer2,sizeof(buffer2),stdin);
    scat(buffer1,buffer2,buffer3);
    printf("%s\n",buffer3);
    return 0;
}
```

**example4lib.c**

```
#include "example4lib.h"

void scat(char *str1,char *str2,char *str)
{
    char tmp_buffer[36];
    strcpy(tmp_buffer,str1);
    strcpy(tmp_buffer+strlen(str1),str2);
    strcpy(str,tmp_buffer);
    return;
}
```

**example4lib.h**

```
#include <string.h>

void scat(char *str1,char *str2,char *str);
```

V. **example5.c**

```
/*
 example5.c - Číta dva reťazce zo štand. vstupu a skonkatenuje ich.
  Použitie dynamickej alokácie.
 Zraniteľnosť: Buffer owerflow
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *scat(char *str1,char *str2)
{
 char *tmp_buffer = malloc(45*sizeof(char));
 strcpy(tmp_buffer,str1);
 strcpy(tmp_buffer+strlen(str1),str2);
 return tmp_buffer;
}
int main ()
{
 char buffer1[20],buffer2[20];
 char *buffer3 = malloc(45*sizeof(char));
 char *buffer4 = calloc(45,sizeof(char));
 scanf("%19s",buffer1);
 scanf("%19s",buffer2);
 buffer3 = scat(buffer1,buffer2);
 buffer4 = scat(buffer1,buffer3);
 printf("%s\n",buffer4);
 return EXIT_SUCCESS;
}
```

VI. **example6.c**

```
/*
 example6.c - Číta znaky zo štand. vstupu a mení
  veľké písmená na malé.
 Zraniteľnosť: Format String
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char *argv[])
{
 int i;
 char buffer[25];
 scanf("%24s",buffer);
 for(i=0;i<strlen(buffer);i++)
 {
  buffer[i] = toupper(buffer[i]);
 }
 printf(buffer);
 return EXIT_SUCCESS;
}
```

VII. **example7.c**

```
/*
 example7.c - Práca s číslami
 Zraniteľnosť: Integer Overflow
*/

#include <stdio.h>
#include <stdlib.h>



int main ()
{
 unsigned int a;
 int b;
 b = -10;
 a = b;
 b = a;
 a = -10;
 a = 100;
 printf("%d\n",a*b);
 return 0;
}
```

VIII. **example8.c**

```
/*
 example8.c - Práca s číslami.
 Zraniteľnosť: Integer Overflow
*/

#include <stdio.h>
#include <stdlib.h>


int substract(int a, int b)
{
  return a - b;
}

int main ()
{
  int a,b;
  unsigned int c;
  printf("Zadajte cislo jedna: ");
 while (scanf("%d",&a)!=1)
 {
    printf("Zadajte cislo jedna: ");
    scanf("%*s");
  }
 printf("Zadajte cislo dva: ");
 while (scanf("%d",&b)!=1)
 {
    printf("Zadajte cislo dva: ");
    scanf("%*s");
  }
  c = substract(a,b);
 printf("%d\n",c);
 return 0;
}
```

IX. **example9.c**

```
/*
 example9.c - Komplexný príklad demonštrujúci
  schopnosti analyzátora.
 Zraniteľnosť: Buffer Overflow, Integer Overflow, Format String
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* scat(char *str1,char *str2)
{
 char *tmp_buffer =(char *) malloc(2*sizeof(char)*strlen(str1));
 char *tmp_buffer1 =(char *) malloc(4*sizeof(char)*strlen(str2));
 strcpy(tmp_buffer1,str2);
 strcat(tmp_buffer1,str2);
 strcat(tmp_buffer1,str2);
 strcat(tmp_buffer1,str2);
 strncpy(tmp_buffer,tmp_buffer1,99);
 strncpy(tmp_buffer,tmp_buffer1,101);
 if (strlen(tmp_buffer) > 100)
 {
 return tmp_buffer1;
 }
 return tmp_buffer;
}

int test (char *str1)
{
 char tmp [150];
 strcpy(tmp,str1);
 return -1;
}

int main ()
{
 char *result;
 char buffer1[50],buffer2[50];
 unsigned int j;
 scanf("%49s",buffer1);
 scanf("%49s",buffer2);
 result=scat(buffer1,buffer2);
 j = test(result);
 printf(result);
 printf("\n");
 return EXIT_SUCCESS;
}
```

# Výpis testov pre sadu modelových programov

I. **example1.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example1/example1.c
+--------------------------------------------------------------------+
+                              C Analyzer Result                     +
+--------------------------------------------------------------------+
+ Filename: examples/example1/example1.c                             +
+--------------------------------------------------------------------+
+ Error log - Function main():                                       +
+     Line                     Operation              Problem        +
+       14                         scanf      Buffer Overflow        +
+--------------------------------------------------------------------+
```

II. **example2.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example2/example2.c
+--------------------------------------------------------------------+
+                              C Analyzer Result                     +
+--------------------------------------------------------------------+
+ Filename: examples/example2/example2.c                             +
+--------------------------------------------------------------------+
+ Error log - Function main():                                       +
+     Line                     Operation           Problem           +
+       15                        strcpy   Buffer Overflow           +
+--------------------------------------------------------------------+
```

III. **example3.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example3/example3.c
+----------------------------------------------------------------------+
+                                C Analyzer Result                     +
+----------------------------------------------------------------------+
+ Filename: examples/example3/example3.c                               +
+----------------------------------------------------------------------+
+ Error log - Function main():                                         +
+ OK - No problems found.                                              +
+----------------------------------------------------------------------+
+ Error log - Function scat():                                         +
+     Line                     Operation            Problem            +
+       14                        strcat     Buffer Overflow           +
+       15                        strcpy     Buffer Overflow           +
+----------------------------------------------------------------------+
```

IV. **example4.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example4/example4.c
+----------------------------------------------------------------------+
+                                C Analyzer Result                     +
+----------------------------------------------------------------------+
+ Filename: examples/example4/example4.c                               +
+----------------------------------------------------------------------+
+ Error log - Function main():                                         +
+ OK - No problems found.                                              +
+----------------------------------------------------------------------+
+ Filename: examples/example4/example4lib.c                            +
+----------------------------------------------------------------------+
+ Error log - Function scat():                                         +
+     Line                     Operation            Problem            +
+        7                        strcpy     Buffer Overflow           +
+        8                        strcpy     Buffer Overflow           +
+----------------------------------------------------------------------+
```

V. **example5.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example5/example5.c
+----------------------------------------------------------------------+
+                                C Analyzer Result                     +
+----------------------------------------------------------------------+
+ Filename: examples/example5/example5.c                               +
+----------------------------------------------------------------------+
+ Error log - Function main():                                         +
+ OK - No problems found.                                              +
+----------------------------------------------------------------------+
+ Error log - Function scat():                                         +
+     Line                     Operation             Problem           +
+       14                        strcpy     Buffer Overflow           +
+----------------------------------------------------------------------+
```

VI. **example6.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example6/example6.c
+----------------------------------------------------------------------+
+                                C Analyzer Result                     +
+----------------------------------------------------------------------+
+ Filename: examples/example6/example6.c                               +
+----------------------------------------------------------------------+
+ Error log - Function main():                                         +
+     Line                     Operation             Problem           +
+       19                        printf       Format string           +
+----------------------------------------------------------------------+
```

VII. **example7.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example7/example7.c
+----------------------------------------------------------------------+
+                                C Analyzer Result                     +
+----------------------------------------------------------------------+
+ Filename: examples/example7/example7.c                               +
+----------------------------------------------------------------------+
+ Error log - Function main():                                         +
+     Line                     Operation             Problem           +
+       16          Assignment statement    Integer Overflow           +
+       17          Assignment statement    Integer Overflow           +
+       18          Assignment statement    Integer Overflow           +
+----------------------------------------------------------------------+
```

VIII. **example8.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example8/example8.c
+--------------------------------------------------------------------+
+                              C Analyzer Result                    +
+--------------------------------------------------------------------+
+ Filename: examples/example8/example8.c                             +
+--------------------------------------------------------------------+
+ Error log - Function main():                                       +
+     Line                    Operation              Problem         +
+       31          Assignment statement    Integer Overflow         +
+------------------------------------------------------ -------------+
+ Error log - Function substract():                                  +
+ OK - No problems found.                                            +
+--------------------------------------------------------------------+
```

IX. **example9.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/example9/example9.c
+--------------------------------------------------------------------+
+                              C Analyzer Result                    +
+--------------------------------------------------------------------+
+ Filename: examples/example9/example9.c                             +
+--------------------------------------------------------------------+
+ Error log - Function main():                                       +
+     Line                    Operation              Problem         +
+       42          Assignment statement    Integer Overflow         +
+       43                        printf       Format string         +
+--------------------------------------------------------------------+
+ Error log - Function scat():                                       +
+     Line                    Operation              Problem         +
+       19                       strncpy     Buffer Overflow         +
+--------------------------------------------------------------------+
+ Error log - Function test():                                       +
+     Line                    Operation              Problem         +
+       30                        strcpy     Buffer Overflow         +
+--------------------------------------------------------------------+
```

X. **snipplr.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/ThirdParty/snipplr.c
+----------------------------------------------------------------------+
+                               C Analyzer Result                      +
+----------------------------------------------------------------------+
+ Filename: examples/ThirdParty/snipplr.c                              +
+------------------------------------------------------ ---------------+
+ Error log - Function main():                                         +
+     Line                    Operation             Problem            +
+       12                       strcpy     Buffer Overflow            +
+----------------------------------------------------------------------+
```

XI. **binStromy.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/ThirdParty/binStromy.c
+----------------------------------------------------------------------+
+                               C Analyzer Result                      +
+----------------------------------------------------------------------+
+ Filename: examples/ThirdParty/binStromy.c                            +
+----------------------------------------------------------------------+
+ Error log - Function main():                                         +
+     Line                    Operation             Problem            +
+       80                        scanf     Buffer Overflow            +
+----------------------------------------------------------------------+
```

XII. **demo.c**

```
xovson00@merlin: ~/$ ./canalyzer examples/ThirdParty/demo.c
+----------------------------------------------------------------------+
+                               C Analyzer Result                      +
+----------------------------------------------------------------------+
+ Filename: examples/ThirdParty/demo.c                                 +
+----------------------------------------------------------------------+
+ Error log - Function main():                                         +
+     Line                    Operation             Problem            +
+       61          Assignment statement   Integer Overflow            +
+       71                       strcat     Buffer Overflow            +
+       72                       strcat     Buffer Overflow            +
+----------------------------------------------------------------------+
```

## Dodatek D

# Metriky kódu

| | |
|---|---|
| **Počet súborov:** | 39 |
| **Počet riadkov zdrojového kódu:** | 13 178 |
| **Veľkosť statických dát:** | 284 445B |
| **Veľkosť spustiteľného súboru:** | 387 360B (Linux, bez ladiacich info.) |