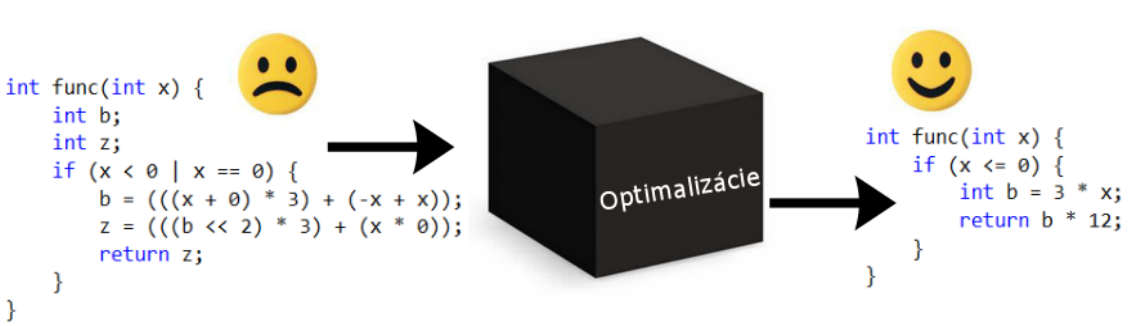
Excel@FIT 2015
http://excel.fit.vutbr.cz

Optimalizácia LLVM IR pre účely spätného prekladu

Jaroslav Kollár*



Abstrakt
Pojem bezpečnosť v rámci informačných technológií v dnešnej dobe nadobúda stále väčší význam. V súčasnosti sú najviac zraniteľné mobilné telefóny, tablety a iné podobné zariadenia. Na analýzu škodlivého softvéru môže byť využitá jedna z techník softvérového reverzného inžinierstva a to spätný preklad. Ten je možné realizovať pomocou spätného prekladača prevádzajúceho platformovo závislú binárnu aplikáciu do vysokoúrovňového kódu. Tento kód je potom možné analyzovať a získané informácie použiť pri tvorbe antivírusových programov. Z dôvodu zjednodušenia analýzy tohto kódu je vhodné ho upraviť do funkčne ekvivalentnej ľudsky čitateľnejšej podoby. O túto úpravu sa starajú optimalizácie v spätnom prekladači. Článok popisuje základné informácie o spätnom prekladači vyvíjaný firmou AVG Technologies. Konkrétne je popísaná štruktúra tohto spätného prekladača a jeho plánované využitie. Dôraz je kladený na optimalizácie, ktoré sú predmetom tohto článku. Pri nich sú uvedené problémy, ktoré riešia a zároveň sú prezentované spôsoby riešenia týchto problémov. Optimalizácie sú vykonávané nad jazykom LLVM IR predstavujúcim vnútornú reprezentáciu v uvedenom spätnom prekladači.

Kľúčové slová: Spätný prekladač, optimalizácie, LLVM IR

Priložené materiály: Stiahnuteľné ukážky

*xkolla03@stud.fit.vutbr.cz, *Fakulta informačných technológií, Vysoké učenie technické v Brne.*

1. Úvod

V dnešnej dobe je v rámci informačných technológií často kladený dôraz na bezpečnosť. Medzi jedno z najväčších rizík je možné zaradiť rôzne druhy škodlivého softvéru. Typickými príkladmi sú počítačové vírusy, trójske kone, internetové červy a iné [1]. Medzi populárnu techniku obrany proti uvedeným typom škodlivého softvéru patrí používanie antivírusových programov. Kvalita týchto programov však určuje poskytovanú úroveň bezpečnosti. Preto je dôležité dbať na vývoj techník umožňujúcich vylepšiť kvalitu programov bojujúcich so škodlivým softvérom. Jednou z techník, ktorú môžu použiť tvorcovia antivírusových programov, je softvérové reverzné inžinierstvo. Napríklad ide o využitie spätného prekladu nad získaným binárnym súborom. O spätný preklad sa postará spätný prekladač, pričom jeho úlohou je z tohto súboru vygenerovať program vo vysokoúrovňovom jazyku. Potom je možné

získaný kód preskúmať a vyhodnotiť jeho činnosť [2]. Na základe uvedeného je zrejmé, že spätný prekladač v rukách tvorcov antivírusových programov môže znamenať veľký prínos. Za týmto účelom je vyvíjaný aj spätný prekladač spoločnosťou AVG Technologies, ktorého popis sa nachádza v sekcii 2. Prostredníctvom verejne prístupnej služby si je možné vyskúšať jeho funkčnosť [3].

Súčasťou uvedeného spätného prekladača sú optimalizácie. Ich hlavnou úlohou je zlepšiť čitateľnosť produkovaného vysokoúrovňového kódu. Zlepšenie je realizované prostredníctvom transformácie kódu na iný čitateľnejší kód. Zároveň musí platiť, že transformovaný kód je funkčne ekvivalentný s tým pôvodným. Čím je čitateľnosť kódu lepšia, tým jednoduchšia je pochopiteľne jeho analýza. Popis optimalizácií sa nachádza v sekcii 3.

Zhodnotenie prínosu prezentovaných optimalizácií sa nachádza v sekcii 4.

Záverečné zhrnutie obsahuje sekcia 5.

2. Spätný prekladač AVG Technologies

Spätný prekladač vyvíjaný spoločnosťou AVG Technologies je platformovo nezávislý na architektúre a súborových formátoch. Aktuálne sú podporované architektúry Intel x86, MIPS, ARM, inštrukčné rozšírenie Thumb, PIC32 a PowerPC. Medzi podporované súborové formáty patrí ELF a PE. Výstup spätného prekladu je možný v jazyku C alebo v modifikovanej verzii jazyka Python. Jeho využitie je plánované pri analýze malvéru [4].

Tento spätný prekladač sa skladá z troch častí. Z prednej časti, strednej časti a zadnej časti. Obrázok 1 demonštruje túto štruktúru. Ako je možné na ňom vidieť, tak niektoré časti spätného prekladača využívajú framework LLVM[1]. Nasledujúce podkapitoly sa venujú každej časti spätného prekladača osobitne.

2.1 Predná časť

Predná časť spätného prekladača je jediná platformovo závislá časť. Hlavnou úlohou prednej časti je prevod binárneho kódu platformovo závislej aplikácie do sekvencie inštrukcií podobných inštrukciám v jazyku LLVM IR[2]. Táto sekvencia inštrukcií je platformovo nezávislá a predstavuje vnútornú reprezentáciu prednej časti. V prednej časti dochádza k odstraňovaniu staticky linkovaného kódu z binárneho kódu, pretože tento kód zbytočne zvyšuje neprehľadnosť výsledného výstupného kódu. Týka sa to napríklad knižničných funkcií typu `printf`, kde je všeobecne známe, čo tieto funkcie vykonávajú. Takto upravený binárny kód je vstupom pre takzvaný inštrukčný dekodér. Jeho funkcionalita je podobná disassembleru, avšak výstupom nie je kód v jazyku assembler, ale sekvencia inštrukcií odpovedajúca reprezentácii LLVM IR. Poslednou inštanciou prednej časti je generátor kódu v jazyku LLVM IR. Tento generátor generuje LLVM IR kód z vnútornej reprezentácie tak, aby bol čo najvhodnejší pre ďalšie časti spätného prekladača [6].

2.2 Stredná časť

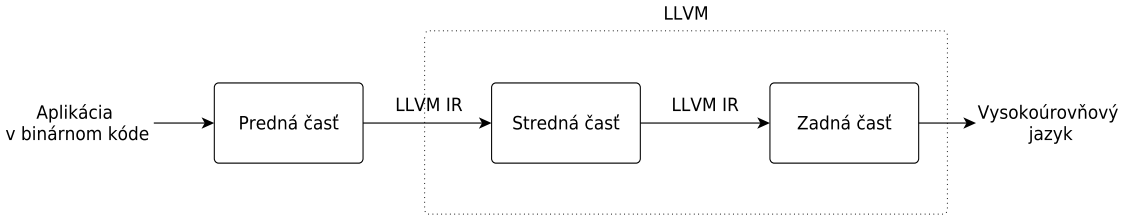
Vstupom strednej časti spätného prekladača je kód v jazyku LLVM IR. Rovnako je kód v tomto jazyku aj výstupom pre túto časť. Úlohou strednej časti je optimalizovať kód v jazyku LLVM IR tak, aby bol čo najvhodnejší pre zadnú časť a zvýšil tak schopnosť generovania čitateľnejšieho kódu zadnou častou spätného prekladača. Stredná časť spätného prekladača je prevažne postavená na frameworku LLVM. Tento framework obsahuje veľké množstvo vstavaných optimalizácií, ktoré sú využívané aby zlepšili kvalitu LLVM IR kódu. V tejto časti sa nachádza aj analýza idiómov, ktorá nie je súčasťou LLVM frameworku. Uvedená analýza je potrebná, pretože prekladač sa pri preklade snaží niektoré inštrukcie nahradiť za iné rýchlejšie vykonateľné inštrukcie. Často však vďaka tomu dôjde k použitiu viacerých inštrukcií miesto jednej. Vďaka tomu dochádza k zhoršovaniu čitateľnosti kódu, a preto sa táto analýza idiómov snaží vrátiť zmeny vykonané prekladačom [6]. V strednej časti sa okrem vstavaných optimalizácií a analýzy inštrukčných idiómov nachádzajú optimalizácie prezentované v tomto článku.

2.3 Zadná časť

Vstupom zadnej časti je LLVM IR kód. Jej výstupom je kód vysokoúrovňového jazyka. Aktuálne podporované jazyky sú jazyk C a modifikovaná verzia jazyka Python. Dôležité je dodať, že je kladený dôraz na to, aby zadná časť generovala čo najčitateľnejší vysokoúrovňový kód. V zadnej časti sa rovnako ako v strednej časti nachádzajú optimalizácie vylepšujúce výstupný kód do čo najčitateľnejšej podoby [4].

[1] LLVM bol pôvodne navrhnutý ako inovatívny framework pre prekladače a pre tvorbu nástrojov na nich založených. Obsahuje množinu jazykovo nezávislých inštrukcií, veľké množstvo vstavaných optimalizačných algoritmov a analýz a prechodnú reprezentáciu [5].

[2] LLVM IR predstavuje prechodnú reprezentáciu, ktorá poskytuje typovú bezpečnosť, nízkoúrovňové operácie, flexibilitu a schopnosť reprezentácie takmer všetkých vysokoúrovňových jazykov [5].



Obrázok 1. Štruktúra spätného prekladača vyvíjaného spoločnosťou AVG Technologies.

3. Optimalizácie v strednej časti spätného prekladača

Táto sekcia sa venuje optimalizáciám nachádzajúcich sa v strednej časti spätného prekladača, ktoré nie sú súčasťou LLVM frameworku. Každá podsekcia obsahuje popis práve jednej optimalizácie. Pri popisoch jednotlivých optimalizácií budú pre lepšie pochopenie uvedené ukážky vo vysokoúrovňovom jazyku, prípadne v pseudokóde. Tieto optimalizácie sú však vykonávané nad jazykom LLVM IR.

Úlohou optimalizácií je primárne zlepšiť čitateľnosť výstupného kódu, prípadne urýchliť beh zadnej časti spätného prekladača a jeho optimalizácií. Urýchlenie je potrebné, pretože niektoré optimalizácie v zadnej časti sú problematické z hľadiska doby behu nad veľkými binárnymi súbormi. Konkrétne ide o optimalizáciu konvertujúcu globálne premenné na lokálne premenné a optimalizáciu odstraňujúcu mŕtve priradenia do globálnych premenných. Ako riešenie tejto pomalosti bol zvolený ich presun do strednej časti spätného prekladača. Pri návrhu tohto presunu sa vychádzalo z toho, že framework LLVM poskytne pre tieto optimalizácie lepšiu podporu, a tak dôjde k ich zrýchleniu. Ďalšie dve presunuté optimalizácie do strednej časti sú optimalizácia odstraňujúca nedosiahnuteľné funkcie a optimalizácia odstraňujúca kód za volaniami funkcií, pri ktorých nikdy nedôjde k návratu toku riadenia programu. Tieto dve uvedené optimalizácie netrpia problémami s rýchlosťou, ale ich presunom do strednej časti sa získa urýchlenie ďalších optimalizácií. Vďaka ním dôjde k odstráneniu kódu neovplyvňujúceho funkčnosť programu, a tak ostatné optimalizácie musia optimalizovať menšiu časť kódu.

3.1 Usporiadanie PHI uzlov

S touto optimalizáciou úzko súvisia takzvané uzly PHI. PHI uzly sa využívajú pri SSA[3] pre spôsob umožnenia výberu hodnoty [7]. Vo vysokoúrovňovom jazyku, ako je napríklad jazyk C, je možné použiť pre výber hodnoty napríklad príkaz `if`. Na obrázku 2 je možné vidieť, že do premennej `b` je priradená hodnota jedna alebo dva. Na základe vyhodnotenia podmienky príkazu `if` sa rozhodne, ktorá hodnota to bude. Táto uvedená ukážka však nespĺňa podmienky SSA, pretože dochádza k dvojitému priradeniu hodnoty do premennej `a`.

```
a = 1;
if (v < 10)
    a = 2;
b = a;
```

Obrázok 2. Pseudokód nesplňujúci podmienky SSA.

Uvedený obrázok 2 je možné upraviť do varianty kódu spĺňajúceho podmienky SSA. Táto úprava zahŕňa vloženie uzla PHI. Výsledok tejto úpravy je možné vidieť na obrázku 3, na ktorom je kód pre jednoduchosť uvedený v pseudokóde. Na obrázku je vidieť, že do premennej `b` sa priradí hodnota na základe výsledku predchádzajúceho porovnania podmienky `if`. Výber hodnoty je teda určený blokom kódu, z ktorého dôjde k volaniu uzla PHI.

```
a1 = 1;
if (v < 10)
    a2 = 2;
b = PHI(a1, a2);
```

Obrázok 3. Pseudokód, ktorý spĺňa podmienky SSA.

V LLVM sa používa pre implementáciu PHI uzlov inštrukcia `phi`. S touto inštrukciou však súvisí jeden významný problém a to taký, že inštrukcie `phi` sú vyhodnocované paralelne [8]. Výstupom spätného prekladača je napríklad jazyk C, ktorý je vyhodnocovaný sekvenčne. Preto je potrebné upraviť kód v jazyku LLVM IR do funkčne ekvivalentného sekvenčného kódu. Úlohou tejto optimalizácie je teda vykonať zmienenú úpravu.

K pochopeniu riešenia tejto problematiky je potrebné uviesť pár informácií o LLVM IR kóde. Funkcia je zložená z takzvaných základných blokov. Tieto bloky obsahujú inštrukcie, pričom každý blok musí byť ukončený ukončovacou inštrukciou. Ukončovacie inštrukcie sú také, ktoré spôsobia pri vyhodnocovaní bloku prechod do iného bloku [5].

[3] *Static single assignment* je vlastnosť prechodnej reprezentácie určujúca, že do každej premennej môže dôjsť iba práve k jednému priradeniu. Súčasne musí byť splnená podmienka, že každá premenná je definovaná pred jej použitím [7].

Majme pre jednoduchosť pseudokód na obrázku 4, ktorý sa podobá kódu LLVM IR. Na tomto obrázku je možné vidieť dve `phi` inštrukcie, ktoré sa nachádzajú v základnom bloku s názvom `blok`. Prvá inštrukcia `phi` znamená, že do premennej `A` sa priradí hodnota 1. Výber tejto hodnoty je určený predchádzajúcim blokom, z ktorého prešlo riadenie toku do tohto bloku. V uvedenom príklade je pre jednoduchosť zobrazený len jeden predchodca, v skutočnosti musí byť definovaná pre každého predchodcu bloku nejaká hodnota. Na obrázku je ale možné demonštrovať aj paralelné vyhodnotenie týchto inštrukcií `phi`. Majme pred začatím vyhodnocovania týchto inštrukcií obsah premenných nasledovne: `A = 3, B = 4`. Pri paralelnom vyhodnotení dôjde k priradeniu hodnoty 1 do premennej `A` a súčasne k priradeniu hodnoty 3 do premennej `B`. Obsah premenných bude teda nasledovný: `A = 1, B = 3`. V prípade sekvenčného spracovania by došlo k uloženiu hodnoty `1` do premennej `B`. Táto hodnota by sa tam dostala z predchádzajúceho uloženia tejto hodnoty do premennej `A`.

```
blok:
    A = phi [ 1, %pred_blok ]
    B = phi [ A, %pred_blok ]
```

Obrázok 4. Príklad paralelného priradenia hodnôt pomocou inštrukcií `phi`.

Uvedenú rozdielnosť pri vyhodnocovaní paralelného a sekvenčného riešenia je možné vyriešiť na príklade uvedenom na obrázku 4 zámenou poradia inštrukcií `phi`.

Týmto riešením však ešte nie sú vyriešené všetky problémy. Premenné v inštrukciách `phi` môžu tvoriť cyklickú závislosť. Majme príklad opäť v pseudokóde na obrázku 5. V takomto prípade nie je možné docieliť ekvivalentnosť sekvenčného a paralelného vyhodnotenia vhodným preskladaním inštrukcií.

```
blok:
    A = phi [ B, %pred_blok ]
    B = phi [ A, %pred_blok ]
```

Obrázok 5. Príklad cyklickej závislosti premenných v inštrukciách `PHI`.

K riešeniu tohto problému je však ešte potrebné dodať, že k paralelnému vyhodnoteniu inštrukcií dochádza v rámci jedného základného bloku. Preto vďaka rozdeleniu inštrukcií `phi` do viacerých vhodne prepojených blokov je možné zabezpečiť ekvivalentnosť sekvenčného a paralelného vyhodnotenia.

3.2 Odstránenie mŕtvych priradení do globálnych premenných

Pod pojmom mŕtve priradenie do globálne premennej rozumieme situáciu, pri ktorej dochádza k priradeniu hodnoty do globálnej premennej a následne dôjde k opätovnému priradeniu nejakej hodnoty do tejto globálnej premennej. Dôležité je ale splniť ešte jednu podmienku, aby išlo o mŕtve priradenie. Priradzovaná hodnota do globálnej premennej nesmie byť nikde v kóde prečítaná z tejto globálnej premennej. Takéto mŕtve priradenia sú v kóde zbytočné a nemenia jeho funkčnosť. Za mŕtve priradenie môžeme považovať i takú situáciu, kedy dôjde k priradeniu hodnoty do globálnej premennej a táto hodnota ostane neprepísaná až do ukončenia behu programu. Zároveň však musí opäť platiť, že priradená hodnota nie je nikde použitá. Úlohou optimalizácie je odstrániť popísané mŕtve priradenia, pretože neovplyvňujú beh programu. Obrázok 6 demonštruje vyššie zmienené princípy. Riadky 4 a 8 obsahujú mŕtve priradenia. Pri mŕtvom priradení na riadku 4 však musí platiť, že nikde v kóde po riadok 6 nedôjde k použitiu priradenej hodnoty.

```
1  int g = 0;
2  void func() {
3      int x;
4      g = 2;
5      ...
6      g = 3;
7      x = g;
8      g = 4;
9  }
```

Obrázok 6. Ukážka obsahujúca mŕtve priradenia do globálnej premennej `g`.

Najdôležitejšou časťou tejto optimalizácie je správne určiť, či sa jedná o mŕtve priradenie. K tomuto účelu je možné navrhnúť analýzu prechádzajúcu celý kód, ktorá zbiera informácie o použití priradzovaných hodnôt do globálnych premenných. Na základe toho je potom možné určiť, či hodnota priradená do globálnej premennej je niekde inde použitá alebo vždy dôjde k prepísaniu tejto hodnoty.

3.3 Konverzia globálnych premenných na lokálne

Úlohou optimalizácie je konvertovať globálne premenné na lokálne všade tam, kde je to možné. Lokálne premenné sú vhodnejšie pre ďalšie optimalizácie, pretože skúmanie ich platnosti v kóde je jednoduchšie než v prípade globálnych premenných. Optimalizácia je dôležitá, pretože každý register, ktorý sa v binárnom kóde použije, je spätným prekladačom transformovaný na globálnu premennú. Vďaka tomu môže vzniknúť

veľký počet globálnych premenných, pretože registrov je veľké množstvo a dochádza k ich častému použitiu.

Rozhodnutie o tom, či globálna premenná môže byť vo funkcii konvertovaná na lokálnu premennú nie je také jednoduché. Dôležité je sa na túto problematiku pozrieť z pohľadu, kde všade môže byť použitá priradená hodnota. Pokiaľ sa nachádzajú všetky použitia priradenej hodnoty v rámci jednej funkcie, tak globálna premenná v priradení môže byť nahradená lokálnou premennou a rovnako aj na všetkých miestach použitia tejto hodnoty dôjde ku nahradeniu globálnej premennej za novú lokálnu premennú. Obrázok 7 zobrazuje kód pred aplikovaním optimalizácie a obrázok 8 obsahuje kód po aplikovaní optimalizácie. Na obrázku 8 je možné ukázať práve prínos konverzie globálnej premennej na lokálnu premennú. Po tejto konverzii je možné pomocou inej optimalizácie vykonať úpravu kódu tak, že priradená hodnota 2 do definície lokálnej premennej `gLoc` sa použije rovno v definícii premennej `x`. Na záver je potrebné vykonať odstránenie lokálnej premennej `gLoc`, nakoľko už nie je využívaná.

```
int g = 0;
void func() {
    g = 2;      // Konvertovatelné.
    int x = g; // Konvertovatelné.
    g = 4;      // Nekonvertovatelné.
    anotherFunc()
}
void anotherFunc() {
    x = g;      // Nekonvertovatelné.
}
```

Obrázok 7. Ukážka kódu pred optimalizáciou konverzie globálnych premenných na lokálne.

```
int g = 0;
void func() {
    int gLoc = 2; // Optimalizované.
    int x = gLoc; // Optimalizované.
    g = 4;
    anotherFunc();
}
void anotherFunc() {
    int z = g;
}
```

Obrázok 8. Ukážka kódu po optimalizácii konverzie globálnych premenných na lokálne.

Najdôležitejšou časťou tejto optimalizácie je správne určiť, na ktorých miestach môže dôjsť k nahradeniu globálnej premennej za lokálnu. K tomuto účelu je možné navrhnúť analýzu, ktorá prechádza celý kód a zbiera si informácie o použití priradzovaných hodnôt do globálnych premenných. Na základe toho je potom možné určiť, či hodnota priradená do globálnej premennej je použitá i mimo funkcie, v ktorej bola priradzovaná.

3.4 Odstránenie nedosiahnuteľných funkcií

Hlavnou motiváciou tejto optimalizácie je odstrániť funkcie, ktoré nie sú nikdy volané. Na obrázku 9 je to funkcia s názvom `func2()`. Je vidieť, že takáto funkcia neovplyvní beh programu. Vďaka odstráneniu týchto funkcií dôjde ku zmenšeniu kódu, ktorý je potrebný analyzovať inými optimalizáciami.

```
void func1() {
    ...         // Nevolá func2().
}
void func2() { // Vhodná k odstráneniu.
    ...
}
int main() {
    func1();
}
```

Obrázok 9. Ukážka kódu znázorňujúca princíp optimalizácie odstraňujúcej nedosiahnuteľné funkcie.

Určenie nedosiahnuteľných funkcií je možné realizovať vďaka takzvanému grafu volania funkcií [9]. S touto optimalizáciou však existujú obmedzenia, kedy nie je možné odstrániť funkcie, ktoré nie sú priamo volané. Ide o tieto situácie:

- Pokiaľ nie je známy vstupný bod programu, nie je možné odstrániť žiadnu funkciu. Je to z toho dôvodu, že nevieme určiť, či práve táto funkcia nie je vstupným bodom programu.
- Funkcie môžu byť volané nepriamo (prostredníctvom ukazateľa). Preto je potrebné zohľadniť nepriame volanie podľa návratovej hodnoty volanej funkcie, počtu parametrov volanej funkcie a ich typu.

3.5 Odstránenie kódu za volaniami funkcií, ktoré nevrátia tok riadenia programu

Optimalizácia odstraňuje kód za funkciami, z ktorých nikdy nedôjde k návratu toku riadenia na miesto, ktoré spôsobilo toto volanie. V jazyku C ide napríklad o funkcie `abort()`, `exit()` a pod [10]. Obrázok 10 znázorňuje princíp tejto optimalizácie.

K určeniu, že ide práve o takéto funkcie je potrebné zohľadniť nasledujúce skutočnosti. Meno funkcie zodpovedá jednému z mien funkcií, pri ktorých nedôjde k návratu toku riadenia. Okrem toho je však potrebné dbať aj na návratový typ tejto funkcie, jej parametre ako aj typy týchto parametrov. Nemenej dôležitým faktorom je, že pre takúto funkciu sa v spätne preloženom kóde nachádza len jej deklarácia a definícia tejto funkcie chýba.

```
int main() {
    ...
    exit(1);   // Nedôjde k návratu.
    x = g;     // Možné odstránit.
    return 2; // Možné odstránit.
}
```

Obrázok 10. Ukážka kódu znázorňujúca princíp optimalizácie odstraňujúcej kód za volaniami funkcií, ktoré nevrátia tok riadenia programu.

4. Zhodnotenie

Posúdenie zlepšenia čitateľnosti kódu v prípade niektorých optimalizácií môže byť veľmi subjektívne. Napríklad pre optimalizáciu konvertujúcu globálne premenné na lokálne dôjde k nárastu množstva kódu. Na druhú stranu je ale potrebné zohľadniť, či takýto kód nebude vhodnejší pre iné optimalizácie riešiace tento problém tak, že ho transformujú do výrazne čitateľnejšej podoby. Optimalizácia odstraňujúca nedosiahnuteľné funkcie, optimalizácia odstraňujúca kód za volaniami funkcií a optimalizácia odstraňujúca mŕtve priradenia má jednoznačný prínos na zvýšenie čitateľnosti kódu, nakoľko dôjde k odstráneniu kódu neovplyvňujúceho beh programu.

Okrem čitateľnosti kódu bol v článku prezentovaný problém pomalosti optimalizácií v zadnej časti spätného prekladača. Nasledujúca tabuľka 1 zobrazuje porovnanie rýchlostí dvoch optimalizácií. Ide o optimalizácie, ktoré boli presúvane zo zadnej časti do strednej časti z dôvodu ich pomalosti.

Tabuľka 1. Zhodnotenie zrýchlenia niektorých presunutých optimalizácií zo zadnej časti do strednej časti.

Meno optimalizácie	Zadná časť	Stredná časť
Odstránenie mŕtvych priradení	09h 55m 24s	00h 06m 23s
Konverzia globálnych premenných na lokálne	08h 21m 46s	00h 08m 15s

Uvedené časy sú prebrané z nočných testov spätného prekladača, pri ktorých je testované veľké množstvo súborov o rôznych veľkostiach. Na servery sú všetky testy spúšťané paralelne na 24 procesoroch. V tabuľke sú prezentované sekvenčné časy pre lepšiu názornosť dosiahnutých výsledkov. V prípade optimalizácie odstraňujúcej nedosiahnuteľné funkcie došlo k zrýchleniu behu spätného prekladača približne o 10 hodín.

5. Záver

V tomto článku bol prezentovaný spätný prekladač vyvíjaný spoločnosťou AVG Technologies. Hlavná časť článku sa venovala optimalizáciám v strednej časti spätného prekladača. Zároveň bol prezentovaný prínos presunu niektorých optimalizácií zo zadnej časti spätného prekladača do strednej časti.

Na základe sekcie venujúcej sa zhodnoteniu dosiahnutých výsledkov je vidieť, že presun optimalizácií zo zadnej časti do strednej časti výrazne vylepšil dobu behu spätného prekladača. K zlepšeniu časov došlo nielen vďaka tomuto presunu, ale aj navrhnutiu nových algoritmov pre tieto optimalizácie. Optimalizácia odstraňujúca globálne priradenia, optimalizácia konvertujúca globálne premenné na lokálne a optimalizácia odstraňujúca nedosiahnuteľné funkcie zlepšila celkovú dobu behu spätného prekladača na nočných testoch približne o 28 hodín, čo predstavuje urýchlenie o cca 7 %.

Literatúra

[1] TechTerms.com. Malware. [online], [cit. 2015-03-19]. http://www.techterms.com/definition/malware.

[2] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Inc., 2005. ISBN-10: 0-7645-7481-7.

[3] AVG Technologies. Retargetable Decompiler. [online], [cit. 2015-03-19]. https://retdec.com/.

[4] Petr Zemek. *Design of a Language for Unified Code Representation*. Interná technická správa projektu Lissom, 2012.

[5] LLVM Project. LLVM Language Reference Manual. [online], [cit. 2015-19-03]. http://llvm.org/docs/LangRef.html.

[6] Jakub Křoustek. *Retargetable Analysis of Machine Code*. PhD thesis, Brno university of technology, 2015. Odovzdaná.

[7] GCC. Static single assignment. [online], [cit. 2015-03-19]. https://gcc.gnu.org/onlinedocs/gccint/SSA.html.

[8] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. *Formalizing the LLVM Intermediate Representation for Verified Program Transformations*. 2012. ISBN-978-1-4503-1083-3.

[9] Wikipedia contributors. Call graph. [online], [cit. 2015-04-11]. http://en.wikipedia.org/wiki/Call_graph.

[10] Petr Zemek. Méně známé skutečnosti o C: Funkce, které se nevrací. [online], [cit. 2015-19-03]. http://cs-blog.petrzemek.net/node/123.