BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2016
http://excel.fit.vutbr.cz

Prostredie pre vývoj statických analyzátorov nad LLVM

Veronika Šoková*

Abstrakt
Tento článok sa zaoberá návrhom verifikačného frameworku pre vývoj statických analyzátorov vo forme Clang/LLVM plug-inu pracujúceho nad abstraktnými doménami. Framework zahrňuje súbor transformačných priechodov, ktoré zjednodušujú analyzovaný súbor. Tým sa zmenší množina konštrukcií v programe, ktorú je nutné v analyzátore podporovať. Jadrom frameworku je jednoduchý verifikačný cyklus, ktorý prehľadáva priestor možných behov programu a budú sa z neho volať obálky inštrukcií, ktoré si tvorca analyzátora definuje vzhľadom na ním vytvorenú abstraktnú doménu. Aplikácia frameworku bude demonštrovaná na reimplementácii verifikačného nástroja Predator založeného na abstraktnej doméne symbolických grafov pamäte.

Kľúčové slová: framework — LLVM — Predator — symbolický graf pamäte — statická analýza — analýza tvaru — abstraktná interpretácia — dynamické dátové štruktúry

Priložené materiály: *N/A*

*xsokov00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Úvod

Vo svete vzniká čoraz viac kódu, pomocou ktorého sú riadené stále dôležitejšie systémy a zariadenia. S ohľadom na to, sú vysoko potrebné prístupy, ktoré zaručia, že programy budú pracovať korektne vzhľadom k danej špecifikácií. Jedným z prístupov ako toho docieliť, je *formálna analýza a verifikácia* [1]. Jej zásadnou technikou je *statická analýza*, ktorá informuje o správaní systému na základe zdrojového popisu bez nutnosti jeho vykonávania (alebo vykonávania s odľahčenou sémantikou). Z toho dôvodu je možné aspoň niektoré z uvedených analýz spúšťať aj nad nedokončeným alebo čiastočným kódom, ktorý nie je možné skompilovať a následne spustiť. Narozdiel od dynamického testovania, často nachádza chybu, kde vzniká a nie kde sa prejavuje.

Statické analyzátory sú teda prostriedkom na kontrolu správnosti zdrojového kódu a s ich pomocou sa odhaľujú chyby programátora. So stále rastúcim dopytom po takýchto analyzátoroch je aj žiaduce mať vhodné vývojové prostredie pre vytváranie takýchto analyzátorov, nad ktorým možno vytvárať nové statické analýzy. U vyvíjaných analyzátorov sa pritom kladie dôraz nielen na ich funkčnosť, ale tiež na jednoduchosť použitia, prehľadnosť a dobrý konceptuálny návrh, aby bolo možné čo najrýchlejšie zapojenie nového člena tímu do vývoja. K tomu by práve malo prispieť prostredie pre vývoj analyzátoru. Toto samotné prostredie by pritom malo byť jednoduché a prehľadné. Malo by ponúknuť rôzne nástroje k zjedušeniu kódu vstupného programu, umožniť použitie rôznych abstraktných domén pre reprezentáciu množín konfigurácií analyzovaných programov (a ich kombináciu) a nemalo by pritom obmedzovať tvorcu analyzátora pri jeho používaní a vyžadovať podporu širokej škály inštrukcií. K vytvoreniu nového prostredia nás motivovali limity existujúcich prostredí a súčasne bohatá skúsenosť s vývojom rady analýz v rámci našej výskumnej skupiny.

Prostredie pre tvorbu analyzátoru, ktorého návrh je hlavným prínosom tejto práce a ktoré by malo spĺňovať vyššie uvedené požiadavky, sme sa rozhodli postaviť nad prekladovým systémom LLVM [2]. LLVM je, podobne ako GCC, šírený pod slobodnou licenciou, ale narozdiel od GCC sa jedná o mladší projekt, čo môže viesť na lepší návrh a prehľadnosť kódov. To uľahčilo

návrh nami vyvýjaného prostredia a viedlo na jeho prehľadnejší kód, čo je významné s ohľadom na ďalší rozvoj a udržovanie prostredia. LLVM naviac ponúka aj prostriedky pre úpravu prekladačom vytvoreného medzikódu. Tieto transformácie možno využiť pre jeho zjednodušenie a tým zmenšiť množinu konštrukcií, ktoré je nutné podporovať. Jedná sa o systém, ktorý je stále vyvíjaný a upravovaný podľa nových požiadaviek a naviac ponúka front-endy pre väčšie množstvo programovacích jazykov.

Prvým ukážkovým analyzátorom, ktorý bude vyvinutý v novom prostredí, bude nová verzia analyzátoru Predator [3]. Predator analyzuje C programy pracujúce s dynamicky alokovanou pamäťou a hľadá chyby s tým spojené: neplatné dereferencie, nedefinované smerníky, viacnásobné uvoľnenie pamäte, pretečenie a iné. Predator pritom overuje prácu i s nízkoúrovňovými operáciami ako sú smerníková aritmetika (angl. *pointer arithmetic*), blokové operácie s pamäťou, reinterpretácia obsahu pamäte a zarovnanie adries.

Predator implementuje konkrétne analýzu, ktorú nazývame *analýza tvaru* (angl. *shape analysis*) [4]. Jej cieľom je charakterizovať *tvar* (podobu) všetkých dynamických dátových štruktúr viazaných smerníkmi, s ktorými sa v programe pracuje. Predator sa pritom obmedzuje na dátové štruktúry typu zoznam.

Aby sa bol Predator schopný vyrovnať s tým, že za behu programu s dynamickými dátovými štruktúrami môže vzniknúť neobmedzene veľa rôznych konfigurácií, používa sa konečná reprezentácia nekonečných množín konfigurácií pamäte, obsahujúcich podobný tvar dynamických dátových štruktúr tzv. *symbolické grafy pamäte* (SMG). SMG predstavujú abstrakciu stavov pamäte, kedy pochopiteľne dochádza k strate určitej informácie. To však môže viesť na falošné hlásenia, napr. falošné pozitíva — čiže v programe bez chýb nájde chybu. Vyhnúť sa tomuto dôsledku je samozrejme, s ohľadom na nekonečnosť stavového priestoru, problematické.

Nový nástroj by mal postupne zvládať štruktúry ako sú jednoduché a vnorené jedno- či obojsmerne viazané zoznamy, neskôr aj stromy a ďalšie dátové štruktúry, a to opäť v nízkoúrovňovej podobe tak ako sú používané v jadre Linuxu a ďalších programoch s dôrazom na efektivitu. Narozdiel od pôvodného nástroja Predator bude kladený veľký dôraz na budúcu rozšíriteľnosť použitej abstraktnej domény a jej kombináciou s ďalšími abstraktnými doménami, napr. pre popis obmedzení na hodnotách smerníkov, ktoré sú dôsledkom adresovej aritmetiky a ktorú Predator zvláda len v obmedzenej podobe (intervaly adries iba s konštantnými hranicami, nemožnosť kombinovať intervaly adries s usporiadaním nad adresami a pod.).

Zvyšok článku je členený nasledovne: v 2. sekcii je neformálne načrtnutá statická analýza použitá v Predatorovi, ktorý sa stane ukážkovým nástrojom postaveným nad naším frameworkom. V sekcii 3 je predstavený hlavný prínos tejto práce, teda návrh prostredia pre vývoj analyzátorov, a v poslednej 4. sekcii sa nachádza zhrnutie a výhľady do budúcna.

2. Nástroj Predator

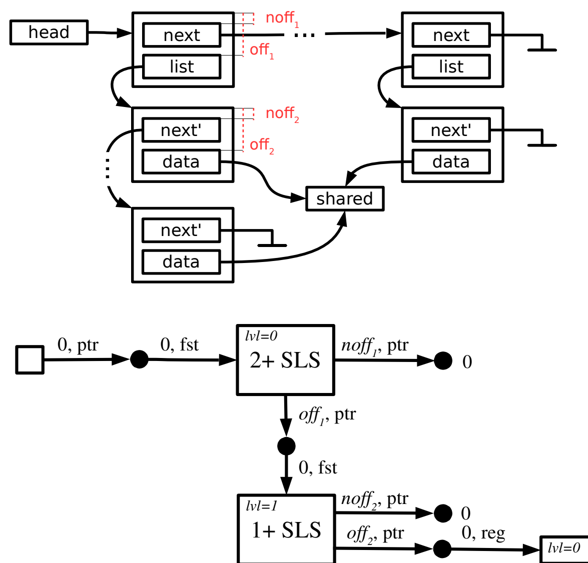
Táto sekcia popisuje prístup pre analýzu tvaru implementovanú v pôvodnom nástroji Predator.

Na vlastnú analýzu možno pozerať ako na *abstraktnú interpretáciu* [1], kde ako abstraktná doména figurujú symbolické grafy pamäte (SMG) [5] a ako operátor *widening* [1] je použitá abstrakcia. Narozdiel od klasickej definície, neberie v úvahu dve postupne vygenerované abstraktné konfigurácie, ale len konfiguráciu aktuálnu. Konvergencia je pritom zaistená iba pre podmnožinu programov so zoznamami. Pre ostatné dátové štruktúry nie je v Predatorovi implementovaná abstrakcia, ktorá by zvládla konečne reprezentovať nekonečné množiny daných štruktúr.

Koncept SMG je inšpirovaný a do istej miery podobný *separačnej logike* [6] s induktívnymi predikátmi typu zoznam. Jedná sa však o čisto grafovo-orientovaný prístup, doplnený o podporu niektorých nízkoúrovňových operácií so smerníkmi.

2.1 Symbolický graf pamäte

Jedná sa o orientovaný bipartitný graf s dvomi typmi uzlov a dvomi typmi hrán. Uzly môžu byť buď *objekty* (hranaté uzly) alebo *hodnoty* (oválne uzly), viď



Obrázok 1. Hierarchicky vnorené jednosmerne viazané zoznamy so zdielaným objektom.

Obrázok 1. Objekty ďalej rozlišujeme na *regióny*, ktoré predstavujú konkrétny blok pamäte ležiaci na halde, zásobníku alebo v globálnom priestore, a *abstraktné objekty*, ktoré môžu byť jedno- alebo obojsmerne viazané segmenty zoznamov (SLS alebo DLS) a vznikli spojením neprerušovanej sekvencie konkrétnych regiónov s rovnakými offsetmi predstavujúcimi položku následníka *noff* a predchodcu *poff* (u DLS). Abstraktný objekt značíme n+SLS, kde $n \geq 0$ určuje minimálnou dĺžkou segmentov. Špeciálnym prípadom objektu je *nulový objekt*, ktorý predstavuje umiestnenie `NULL`. Pre všetky objekty si ešte pamätáme ich platnosť a veľkosť. Pre reprezentáciu hierarchických dátových štruktúr sa zavádza pre všetky uzly grafu atribút *level* (úroveň).

Hrana vedúca z objektu k hodnote je typu *„má hodnotu“* a určuje offset a typ položky a hrana vedúca z hodnoty k objektu je typu *„ukazuje na“* a určuje offset a druh cieľovej položky. Druhom môže byť región (*reg*), prvý/posledný prvok zoznamu (*fst*, *lst*) alebo všetky prvky nadzoznamu (*all*). Matematickým zápisom $r_1 \xrightarrow{off_1,ptr} a \xrightarrow{off_2,reg} r_2$ značíme, že v regióne r_1 na offsete off_1 je uložený smerník a ukazujúci na región r_2 na offsete off_2.

Symbolické vykonávanie inštrukcií nad SMG používa pre hodnoty *reinterpretáciu*. Definovať ju pre všetky možné dátové typy (a ich hodnoty) je náročné. Preto sa uvažuje minimum operácií, ktoré vedú na spoľahlivosť formálnej verifikácie. Čiže, ak verifikátor odpovie, že je program korektný — jeho správnosť bola matematicky dokázaná, tak aj je. S ohľadom na program, ktorý verifikujeme, sa jedná o pretypovanie a čítanie a zápis do pamäte. Špeciálne je použitá reinterpretácia nad vynulovanou pamäťou, čo sa využíva pri spracovaní štandardných C funkcií `calloc()` a `memset()`.

Ďalším operátorom nad SMG je *join* (spájanie), pri ktorom dochádza k strate informácií. Po spojení nemusia byť zachované pôvodné minimálne dĺžky zoznamových segmentov, môžu byť abstrahované offsety cieľov smerníkov a i. Bežne sa spúšťa pri spájaní viacerých konfigurácií vznikajúcich pri vetveniach a cykloch (podsekcia 2.2). Operácia *join* je implementovaná ako súbežné prehľadávanie do hĺbky (DFS) oboch grafov, ktoré chceme spájať, z určených počiatočných uzlov. Spájané objekty však musia byť kompatibilné: rovnaká úroveň zanorenia, veľkosť, offsety... Z každého objektu potom prebieha spájanie rekurzívne. Výsledok spájania určuje status $\mathbb{J} = \{\simeq, \sqsupset, \sqsubset, \bowtie\}$. Symbolom $\simeq$ značíme, že sú podgrafy izomorfné, $\sqsupset$, že ľavý je všeobecnejší než pravý, $\sqsubset$, že pravý je všeobecnejší než ľavý a $\bowtie$, ak sú neporovnateľné. Napr. pre abstraktné objekty platí:

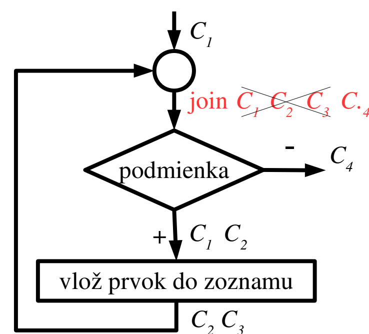
$$n\text{+SLS} \sqsubset m\text{+SLS, kde } n > m$$

Operátor *join* sa používa aj pri abstrakcii, kedy spájame rovnaký graf z dvoch rôznych uzlov, aby nám vznikol abstraktný objekt.

V Predatorovi sú *aritmetické operácie* vykonávané presne po určitú pevne zadanú hranicu. Nad touto hranicou je premenným priradená hodnota *unknown*, čiže o nej nevieme povedať nič. Naviac pri adresách sú použité intervaly s pevnými hranicami. V novom návrhu by oboje malo byť zovšeobecnené pomocou kombinácie SMG s vhodnými doménami pre popis množín číselných hodnôt: intervaly s premennými hranicami, oktagony, DBM [7] a pod.

2.2 Konfigurácie programu

Symbolická konfigurácia programu (SPC) značí dvojicu tvorenú obsahom pamäte v podobe SMG a mapovaním premenných zdrojového programu na regióny v SMG. SPC sa vytvára po symbolickom prevedení každej inštrukcie. Po vetvení a cykloch sa spájajú konfigurácie pomocou operátoru *join* (obr. 2), aby sa zredukoval počet SMG, ktoré sú priradené jednotlivým uzlom *grafu riadenia toku* (CFG) a je nutné ich preskúmať. Naviac sa môže po vykonaní cyklu urobiť abstrakcia, aby sa sekvencia regiónov tvoriacich zoznam konkrétnej dĺžky n nahradila segmentom s dĺžkou n+. Tým dochádza k akcelerácii výpočtu, ktorého cieľom je zaistiť konečnosť behu analýzy i pri práci so zoznamami s nekonečne veľa rôznymi dĺžkami.



Obrázok 2. Program vytvárajúci jednosmerne viazaný zoznam do premennej x. V C_1 ukazuje x na `NULL`. Do x sa vloží 1 prvok a vznikne C_2. Pri ďalšom cyklení sa do C_2 vloží ďalší prvok, *abstrakcia* to zovšeobecní na 2+SLS a vznikne C_3. *Join* spojí C_2 a C_3 do jednej konfigurácie C_4, ktorá bude 1+SLS. Ak by bola podmienka vyhodnocovaná nedeterministicky, bolo by nutné v `else` vetvi skúmať všetky konfigurácie spracované `then` vetvou (C_1, C_2, C_3, C_4). S *join* operátorom stačí skúmať iba C_4.

Spájanie prebieha nad každou premennou programu. Lebo bežne nie je SMG súvislý graf. Očakáva sa, že všetky objekty na halde budú dosiahnuteľné pomocou statických objektov alebo objektov na zásobníku (globálne a lokálne premenné programu). Na obrázku 2 je znázornené spájanie konfigurácií pri spracovaní cyklu. C_i nám určuje konfiguráciu programu po vykonaní predchádzajúceho uzlu grafu.

3. Návrh frameworku

Obsahom tejto sekcie je popis navrhovaného verifikačného prostredia pre tvorbu statických analyzátorov. Výsledný analyzátor bude spúšťaný ako Clang/LLVM plug-in. Tento prekladový systém bol zvolený pre jeho komplexnosť, prehľadnosť medzikódu a ľahkú napojiteľnosť ďalších analýz v dobe prekladu.

Cieľom je vytvoriť prostredie, ktoré by bolo čo najjednoduchšie a čo najmenej obmedzujúce pre autorov budúcich analýz z toho hľadiska, ako majú svoju analýzu písať. Preto sme rozhodli, že vytvorený rámec si nebude vynucovať, narozdiel napr. od prostredia CPAchecker [8], že analýza musí byť písaná formou reakcie na určité udalosti generované pri priechode CFG, že analýza musí podporovať tie či oné operácie a pod. Ďaľšou požiadakou potom je, aby bol medzikód, nad ktorým prebieha analýza, aj naďalej kompilovateľný (narozdiel od prostredia IKOS [9]). Samotné prostredie je preto navrhované úplne minimalisticky (obr. 3).

Prednú časť prostredia bude tvoriť clang [10], predná časť prekladového systému LLVM [2]. Tá nám zabezpečí lexikálnu, syntaktickú a sémantickú analýzu vstupného programu (primárne v jazyku C). Výstupom je vygenerovaný LLVM IR kód [11] v podobe modulu reprezentovaného CFG. Modul pozostáva z funkcií, ktoré sú rozdelené na základné bloky (BB). Pričom každý BB je zakončený ukončujúcou inštrukciou, ktorá nám určuje pokračujúce bloky.

Nasledujúcou súčasťou prostredia sú funkcie pre zjednodušenie medzikódu LLVM IR v podobe transformačných priechodov. Ďalej prostredie zahrňuje funkcie pre načítanie zjednodušeného medzikódu do pamäte a vzorové cykly pre rôzne typy priechodov týmto kódom. Jednotlivá konfigurácia programu sa spracováva v jednom priechode cyklu. Cyklíme, až kým nespracujeme všetky konfigurácie. Predpokladá sa, že autor konkrétnej analýzy si vyberie niektorý z týchto cyklov, prípadne si ho upraví na mieru svojej analýzy a doplní si do neho príslušné abstraktné spracovanie jednotlivých príkazov programu.

3.1 Úprava medzikódu

Účelom transformačných priechodov je zjednodušiť podobu LLVM IR, aby sa dosiahlo redukcie množiny konštrukcií jazyka, ktoré musia vytvárané analýzy podporovať.

Jedná sa o redukciu inštrukčnej sady a redukciu druhov operandov jednotlivých inštrukcií.

Štandardná práca s LLVM vyzerá tak, že najprv *predná časť* (clang) prevedie preklad do LLVM IR. Následne nad ním spustí *optimalizátor* (opt), ktorý vyprodukuje opäť LLVM IR. Nakoniec *zadná časť* vyprodukuje cieľový spustiteľný program. Pre potreby zjednodušenia kódu sme sa v našom prípade rozhodli napojiť hneď za výstup clang-u a tým mať väčšiu kontrolu, ktoré optimalizácie sa vykonajú. Pridané transformačné priechody sú vo forme špecifickej optimalizácie, i keď z pohľadu prekladu sa nejedná o reálne optimalizácie. Z oficiálnych [12] potom budú použité len tie, ktoré pre nás majú zmysel, menovite:

- PromoteMemoryToRegisterPass (`-mem2reg`),
- LowerSwitchPass(`-lowerswitch`).

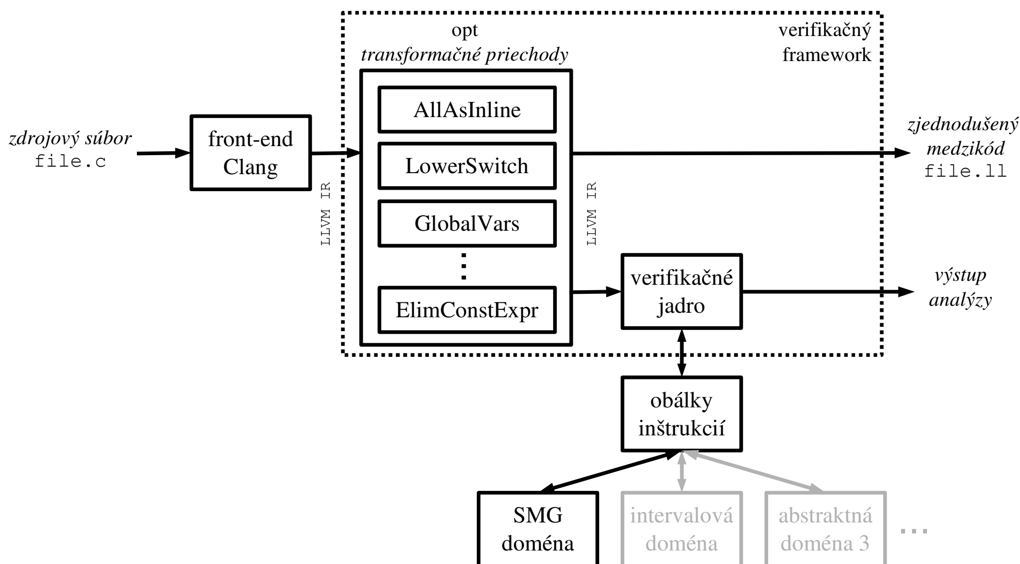
Pridané transformačné priechody zahŕňajú:

- AllAsInlinePass (`-all-inline`),
- GlobalVarsPass (`-global-vars`),
- LowerMemIntrinsicPass (`-lowermem`),
- LowerSelectPass (`-lowerselect`),
- EliminateConstExprPass (`-elim-constexpr`),
- EliminatePHIPass (`-elim-phi`).

3.1.1 Redukcia operandov

Operandom inštrukcie môže byť iná *inštrukcia* (predstavuje register obsahujúci výsledok danej inštrukcie), *konštanta* alebo *konštantný výraz*. V rámci analýz je vhodnejšie pracovať len s jednoduchými konštantami ako sú číslicové prípadne reťazcové literály. LLVM IR však môžu obsahovať aj *zložené literály* vznikajúce pri inicializácií poľa alebo štruktúry. Takéto zložené inicializátory globálnych premenných predstavujú špeciálnu konštrukciu, ktorú je potom potrebné v analyzátoroch zvlášť implementovať. Tieto konštrukcie možno odstrániť priechodom `-global-vars` (popísaný nižšie). V prípade lokálnej inicializácie, sa v LLVM umelo vytvára statická premenná inicializovaná príslušným zloženým literálom, ktorý je pomocou `llvm.memcpy` nahraný do lokálnej premennej. To odsraňuje `-lowermem` priechod. Daná inicializácia je rozložené na sekvenciu dvoch inštrukcií `getelementptr` rátajúcich adresu a `store`, ktoré už vkladajú jednoduchý literál na danú adresu.

Tiež nám vadí, ak bude operandom *konštantný výraz*, lebo musíme podporovať viac konštrukcií pri



Obrázok 3. Konceptuálny návrh frameworku pre verifikačné nástroje.

analýze. Naša transformácia vytvorí novú inštrukciu, ktorej výsledok predstavuje vyhodnotenie konštantného výrazu. Táto inštrukcia nahradí konštantný výraz na mieste operandu. Rieši to `-elim-constexpr`.

3.1.2 Redukcia inštrukčnej sady

Ďalším aspektom programov, ktorý nie je treba u jednoduchších analyzátorov uvažovať, je členenie programov na funkcie. Tohoto členenia sa dá zbaviť, pre nerekurzívne programy, vložením kódu funkcií na miesto ich volania. Toto samozrejme znižuje efektivitu výslednej analýzy, ale zjednodušuje jej tvorbu. Sploštený kód vytvoríme priechodom `-all-inline`.

Ďalšou úpravou, ktorú je možno aplikovať, je odstránenie inštrukcie `alloca` pomocou `-mem2reg`. Tým sa zamedzí alokácii na zásobníku. Dá sa aplikovať iba na programy, kde je každá premenná použitá len raz v `load` a `store` inštrukciách.

Analýzu môže tiež zjednodušiť, ak sa nainicializuje pamäť pre globálne a statické premenné na jednom mieste a nie až pri prvom použití (zamedzuje optimalizácií, kedy sa vytvárajú globálne objekty až pri ich prvom použití). Pomocou `-global-vars` sa vytvorí nová funkcia `__initGlobalVar()`, v ktorej budú priradenia pre všetky globálne premenné (inštrukcie `store`) a bude volaná ako prvá z `main()`. Tento priechod však nemožno použiť pri analýze otvoreného kódu. Bez tohoto priechodu nemožno použiť ani priechod `-elim-constexpr`, pretože by bola ignorovaná inicializácia globálnych symbolov.

Priechod `-lowermem` je užitočný, ak analyzátor ešte nepodporuje spracovanie C štandardných funkcií pre prácu s pamäťou. Tieto funkcie sa nachádzajú v LLVM IR, aj keď ich nepoužije priamo programátor, a to pri inicializácii štruktúr a polí. Takto LLVM implicitne vytvorené `llvm.memcpy` a `llvm.memset` sa dajú nahradiť bez straty informácií. Pri programátorom použitých funkciách sa nekopíruje výplň (angl. *padding*) medzi položkami štruktúr. Nahradenie prebehne, len ak sú zdrojové a cieľové miesta typovo a veľkostne rovnaké. V nasledujúcom prípade sa nepodarí `memcpy()` odstrániť.

```
char *from = "abc";
char to[10];
memcpy(&to,from,4);
```

V prípade, že analýza nevyžaduje *SSA formu* [13], môžeme pomocou priechodu `-elim-phi` odstrániť Φ-inštrukcie (`phi`). Pozostávajú z postupnosti dvojíc (val_i, BB_i), kde val_i predstavuje hodnotu, ktorá sa priradí do premennej, ak bolo skočené z bloku BB_i.

Ak vykonávame nepodmienený skok z BB_x na BB_y, ktorý obsahuje priradenie $var \leftarrow \Phi(var_x, BB_x)$, vložíme pred skokovú inštrukciu v BB_x priradenie $var \leftarrow var_x$. Ak by sa jednalo o podmienený skok na BB_y, vytvoríme nový BB_{xy} s dvomi inštrukciami: priradenie $var \leftarrow var_x$ a nepodmienený skok na pôvodný cieľ BB_y. V podmienenom skoku nahradíme cieľ BB_y vsunutým BB_{xy}.

Ďalšou špecifickou konštrukcou je ternárny výraz v C, ktorý je reprezentovaný v LLVM IR ako inštrukcia `select`, ktorú možno odstrániť použitím priechodu `-lowrselect`. Ak základný blok *BB* obsahuje danú inštrukciu $var \leftarrow (cond)?var_{true} : var_{false}$, rozdelíme ho v mieste inštrukcie a jeho druhá časť bude tvoriť BB'. Inštrukciu `select` nahradíme podmieneným skokom na BB_{true} a BB_{false}. BB_{true} bude obsahovať priradenie $var \leftarrow var_{true}$ a nepodmienený skok na BB'. BB_{false} bude obsahovať priradenie $var \leftarrow var_{false}$ a ne-

podmienený skok na BB'.

Na záver možno bez straty informácií odstrániť aj LLVM inštrukciu `switch` reprezentujúcu konštrukciu switch v jazyku C. Jedná sa už o oficiálny priechod `-lowerswitch`.

3.2 Verifikačné jadro

V prípade klasickej analýzy je vstupným bodom prehľadávania prvá inštrukcia vo funkcii `main()`. Pri analýze otvoreného kódu sa bude prehľadávať z určitej funkcie, no nebude to `main()` a teda veľa vecí nebude inicializovaných.

Tvorca analyzátora dostane zjednodušené CFG a môže sa rozhodnúť, či bude prehľadávať stavový priestor pomocou DFS alebo BFS. V prvom prípade sa budú rozpracované inštrukcie ukladať do fronty a v druhom prípade na zásobník.

Programátor tiež musí doimplementovať jednotlivé obálky inštrukcií LLVM IR, ktoré budú spúšťané vo verifikačnom cykle. Taktiež musí doimplementovať operátory *join*, abstrakciu a *entailment checking*, ktorý otestuje, či pridávaný stav programu už nie je pokrytý existujúcim stavom.

3.3 Abstraktná doména

Pre potreby reimplementácie Predatora je čiastočne implementovaný formálny model predstavený v sekcií 2. Každej inštrukcii zodpovedá priradený zoznam konfigurácií, ktorý obsahuje aktuálnu kópiu SMG po symbolickom vykonaní danej inštrukcie a mapovanie všetkých používaných premenných programu.

V LLVM IR je premenná v globálnom priestore značená ako `@var` a v lokálnom priestore ako `%var`. O správne mapovanie premenných programu na regióny SMG sa starajú prevodníky typu `Value*` (LLVM IR) → `Region` (SMG doména). Jeden sa vytvára pre globálny priestor, v prípade, že program obsahuje globálne premenné, a niekoľko pre lokálne priestory. V prípade rekurzie bude rovnaká LLVM premenná `%var` v rôznych úrovniach zanorenia namapovaná na iný región. Prevodníky sa prehľadávajú hierarchicky. Počnúc od aktívneho lokálneho priestoru, práve spracovávaného rámca zásobníka (angl. *stack frame*), po globálny priestor.

4. Záver

Hlavnou náplňou článku bol návrh nového prostredia pre statické analyzátory. Toto prostredie je postavené nad LLVM, pre jeho komplexnosť a prehľadnosť medzikódu, ako už bolo popísané vyššie. Prostredie je zamýšľané ako minimalistické, aby čo najmenej obmedzovalo vývojárov analýz a súčasne im uľahčilo ich vývoj. S ohľadom na to sme rozhodli, že základom prostredia bude sada zjednodušujúcich transformácií nad LLVM IR. Ďalej prostredie zahŕňa niekoľko vzorových verifikačných cyklov nad zjednodušeným LLVM CFG načítaným v pamäti. Nad nimi si vývojári vytvoria svoje konkrétne analýzy. Praktická aplikácia frameworku je demonštrovaná reimplementáciou nástroja Predator, ktorého koncept bol stručne popísaný.

Aktuálne je implementovaná časť transformačných priechodov a navrhnutý verifikačný framework. Návrh SMG a SPC, pre potreby novej verzie nástroja Predator, je vo fáze dokončovania.

V najbližšej budúcnosti bude dokončená implementácia frameworku a analyzátora založeného na SMG, ktorá sa otestuje na nových zdrojových súboroch ako aj podmnožine stávajúcich testov dodávaných s Predatorom.

Vo vzdialenejšej budúcnosti je v pláne využiť lepšiu rozšíriteľnosť prostredia a v ňom implementovanej novej verzie Predatora a rozšíriť ju o rysy, ktoré pôvodný Predator nepodporoval. Medzi ne patrí napr. lepšia podpora aritmetiky, či už na celočíselných dátach programov, tak aj na adresách v rámci adresovej aritmetiky, podpora analýzy otvorených programov, podpora analýzy s bohatšou triedou dynamických dátových štruktúr a pod.

Poďakovanie

Chcela by som poďakovať môjmu vedúcemu prof. Tomášovi Vojnarovi a kolegovi Ing. Tomášovi Fiedorovi za cenné rady a pripomienky k textu práce. Taktiež ďakujem anonymným recenzentom za ich cenné rady a pripomienky.

Táto práca bola podporovaná projektom GA ČR 14-11384S.

Literatúra

[1] VOJNAR, T. a LENGÁL, O. *FAV Lectures* [online]. [cit. 2015-11-26]. Dostupné na: http://www.fit.vutbr.cz/study/courses/FAV/public.

[2] LATTNER, C. a AL et. *The LLVM Compiler Infrastructure* [online]. 2007 [cit. 2014-04-27]. Dostupné na: http://llvm.org/.

[3] DUDKA, K., PERINGER, P. a VOJNAR, T. *Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic*. 2011. 23 s. Tech. rep.

[4] BERDINE, J., CALCAGNO, C., COOK, B. et al. Shape Analysis for Composite Data Structures.

In *Proc. of CAV'07*. [b.m.]: Springer, 2007. S. 178–192. LNCS, sv. 4590.

[5] DUDKA, K., PERINGER, P. a VOJNAR, T. Byte-Precise Verification of Low-Level List Manipulation. In *Proc. of SAS'13*. [b.m.]: Springer, 2013. S. 215–237. LNCS, sv. 7935.

[6] REYNOLDS, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. [b.m.]: IEEE Computer Society, 2002. S. 55–74.

[7] JEANNET, B. a MINÉ, A. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proc. of CAV'09*. [b.m.]: Springer, 2009. S. 661–667. LNCS, sv. 5643.

[8] BEYER, D. *CPAchecker: The Configurable Software-Verification Platform* [online]. [cit. 2016-01-09]. Dostupné na: http://cpachecker.sosy-lab.org/.

[9] KOGA, D. *IKOS: Inference Kernel for Open Static Analyzers* [online]. [cit. 2016-01-09]. Dostupné na: http://ti.arc.nasa.gov/opensource/ikos/.

[10] *clang: a C language family frontend for LLVM* [online]. [cit. 2014-05-04]. Dostupné na: http://clang.llvm.org/.

[11] *LLVM Language Reference Manual* [online]. 2003, 2016-01-08 [cit. 2016-01-09]. Dostupné na: http://llvm.org/docs/LangRef.html.

[12] *LLVM's Analysis and Transform Passes* [online]. 2016-04-08 [cit. 2016-04-09]. Dostupné na: http://llvm.org/docs/Passes.html.

[13] ZHAO, J., NAGARAKATTE, S., MARTIN, M. M. K. et al. Formal Verification of SSA-based Optimizations for LLVM. *SIGPLAN Not.* June 2013, roč. 48, č. 6. S. 175–186. ISSN 0362-1340.