VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

STATICKÁ ANALÝZA PARAMETRŮ MODULŮ JÁDRA LINUXU

STATIC ANALYSIS OF LINUX'S MODULE PARAMATERS

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE **NIKOLAS PATRIK**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
SUPERVISOR

BRNO 2019

Abstrakt

Táto technická správa sa zaoberá vytvorením LLVM priechodu pre výpis východzích hodnôt globálnych premenných. Práca popisuje LLVM framework a triedy ktoré sú potrebné na napísanie LLVM priechodu. Na implementáciu priechodu je použitý jazyk C++ a tento priechod je vytvorený za účelom získavania východzích hodnôt parametrov modulov jadra Linuxu.

Abstract

This technical report presents extracting default values of global variables from source code. It describes the LLVM framework and classes needed for writing an LLVM pass. Implementation of this LLVM pass is written in C++ and the pass is designated to be used for extracting default values of parameters of Linux kernel modules.

Kľúčové slová

statická analýza, LLVM, DiffKemp, pass, c++

Keywords

static analysis, LLVM, DiffKemp, pass, c++

Citácia

PATRIK, Nikolas. *Statická analýza parametrů modulů jádra Linuxu.* Brno, 2019. Semestrální projekt. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Vojnar, Ph.D.

Statická analýza parametrů modulů jádra Linuxu

Prehlásenie

Prohlašuji, že jsem tuto práci vypracoval samostatně pod vedením pana prof. Ing. Tomáša Vojnara, Ph.D. a Ing. Viktora Malíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Nikolas Patrik
24. januára 2019

Poďakovanie

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Obsah

1 Úvod **2**

2 LLVM framework **3**
2.1 Komponenty LVVM frameworku 3
2.1.1 Front-end 3
2.1.2 Priechody 3
2.1.3 Back-end 4
2.1.4 Linker 4
2.2 LLVM-IR 4
2.3 LLVM C++ API 4
2.3.1 LLVM Priechody 4

3 LLVM priechod pre získanie východzích hodnôt globálnych premenných **6**
3.1 CMake 6
3.1.1 Zostavenie priechodu mimo zdrojovej štruktúry LLVM 7
3.2 Trieda Constant 7
3.2.1 Podtrieda ConstantExpr 8

4 Experimenty **9**

5 Záver **11**

Literatúra **12**

Kapitola 1

Úvod

V poslednej dobe ide vývoj softvéru rapídne vpred, ale často je potrebné zachovávať časť funkcionality programov podľa dohodnutého kontraktu. Na overenie zmien v programe sa dá využiť napríklad statická analýza, ktorá skúma zdrojové súbory programov. Jedným z nástrojov, využívajúcich túto techniku je DiffKemp[1], ktorý skúma sémantiku parametrov modulov jadra Linux. Na analýzu využíva LLVM framework, konkrétne tzv. priechody, ktoré analyzujú zdrojové kódy modulov jadra, preložené do internej reprezentácie LLVM.

Cieľom tejto práce je popísať vývoj priechodu na získavanie východzích hodnôt globálnych premenných, pretože parametre modulu jadra Linuxu sú reprezentované ako tieto premenné. Tento vývoj zahrňuje oboznámenie sa s LLVM frameworkom a jeho dokumentáciou. Ďalej v tomto dokumente popíšeme konkrétne triedy potrebné na písanie daného priechodu a komplikácie, na ktoré je možné pri jeho vývoji natrafiť.

Nasledujúci text popisuje výsledky projektovej praxe a znalosti nadobudnuté počas celého semestra. V kapitole 2 sa bližšie zoznámime s LLVM frameworkom, povieme si niečo o tom ako funguje, z akých častí sa skladá a detailne si popíšeme niektoré z nich. V ďalšej kapitole 3 sa potom zoznámime s písaním konkrétneho priechodu ktorý bude schopný extrahovať východzie hodnoty globálnych premenných a detailne popíšeme jeho funkcionalitu a prostriedky ktoré využíva. Na záver, v kapitole 4, otestujeme vytvorený priechod na reálnych zdrojových kódoch z jadra Linuxu. Aby sme zaistili funkcionalitu nášho priechodu v prostredí, kde beží aj DiffKemp, testy budú vykonané v Docker kontajneri dodávaného spolu s nástrojom.

[1] DiffKemp[6]

Kapitola 2

LLVM framework

LLVM(Low level virtual machine) je infraštruktúra pre prekladač ktorá umožňuje optimalizácie kódu počas prekladu, linkovania alebo počas behu programu pre rôzne programovacie jazyky[2]. Výhodou LLVM je, že pre ľubovolný programovací jazyk stačí vytvoriť front-end na preklad programov daného jazyka do internej reprezentácie LLVM(jedná sa o pseudo-assembler ktorý poskytuje aj operácie z vyšších programovacích jazykov). Ďalej sa tento kód optimalizuje pomocou priechodov, konvertuje a linkuje sa do strojového kódu závislom na stroji na ktorom daný preklad beží. LLVM ale okrem optimalizácií môže byť taktiež skvelý aj na statickú analýzu kódu čo je cieľom tohto projektu. LLVM pracuje nad reprezentáciou programov ktorá pripomína asembler a ktorej porozumie aj človek. Iné prekladače túto výhodu nemajú pretože ich vnútorná reprezentácia programov sú zvyčajne dátové štruktúry v pamäti ktoré sú natoľko komplikované, že je problém sa v nich vyznať ako v celku.

2.1 Komponenty LVVM frameworku

LLVM je projekt ktorý zastrešuje rôzne fázy prekladu a podľa toho sa aj skladá z viacerých komponent. Tie sú popísané v nasledujúcich sekciách.

2.1.1 Front-end

Front-end slúži na preklad programu do internej reprezentácie LLVM a tým zjednodušuje ďalšie fázy prekladu, ktoré nemusia pracovať s vysokoúrovňovými vlastnosťami jazyka, ako je napríklad C++. V súčastnosti existujú front-endy na preklad z jazykov ako sú Ada, C, C++, D, Delphi, Fortran,Haskell, Julia, Objective-C, Rust a Swift[2].

2.1.2 Priechody

Priechody(v angličtine passes) zvyčajne transformujú programy v internej reprezentácií LLVM, pričom typicky zachovávajú funkcionalitu programu a zrýchľujú jeho beh [8]. Priechody ktoré menia kód sa nazývajú transformačné priechody a keď zachovajú funkcionalitu ide konkrétne o optimalizácie preloženého medzi-kódu(internej reprezentácie). Transformačné priechody ale nemusia nutne zachovávať funkcionalitu programu a môžu ľubovolne nahrádzať konkrétne inštrukcie, alebo všetko od základných blokov programu až po celé funkcie až moduly. V podstate celý vygenerovaný kód môžeme meniť tak ako momentálne potrebujeme. Okrem transformačných priechodov existujú ešte analyzačné priechody, ktoré sa vyžívajú napr. v statickej analýze, pričom umožňujú skúmať rôzne časti programov od

samotných inštrukcií, až po moduly ako celok. V nasledujúcej kapitole budeme využívať priechody, ktoré budú bližšie popísané v kapitole 2.3.1, na získanie východzích hodnôt globálnych premenných.

2.1.3 Back-end

Back-end sa stará o preloženie programov v internej reprezentácií do strojového kódu. LLVM v súčastnosti podporuje mnoho inštrukčných sád vrátane ARM, Qualcomm Hexagon, MIPS, Nvidia Parallel Thread Execution (PTX; nazývané NVPTX v LLVM dokumentacií), PowerPC, AMD TeraScale, AMD Graphics Core Next (GCN), SPARC, z/Architecture (nazývaný SystemZ v LLVM dokumentacií), x86, x86-64, and XCore [2].

2.1.4 Linker

Podprojekt `lld` je pokus o vytvorenie platformovo nezávislého linkeru. V prípadoch kde `lld` nestačí je možné použiť `GNU ld`. Použitie `lld` umožnuje optimalizácie v čase linkovania. Keď sú tieto optimalizácie zapnuté prekladač generuje LLVM bitkód namiesto natívneho binárneho kódu a potom generovanie natívneho kódu je spravené LLVM `lld` linkerom.

2.2 LLVM-IR

Jadrom projektu LLVM je jeho interná reprezentácia preloženého programu(ďalej už iba IR). Jedná sa o medzi-kód, ktorý je dosť podobný assembleru. IR je silne typová jednoduchá (redukovaná) inštrukčná sada(alebo inak nazývaná aj RISC – Reduced Instruction Set Computing), ktorá abstrahuje detaily ohľadom strojovo závislých operácií. Napríklad volanie funkcie je zapuzdrené v inštrukciách `call` a `ret` pričom tie sú volané s explicitnými parametrami. Taktiež namiesto fixného počtu registrov. IR používa nekonečne mnoho dočasných v tvare `%0,%1,%2,...`. LLVM poskytuje tri izometrické (funkcionálne rovnaké) formy IR: **čitateľný pseudo-assembler formát, C++ formát** pre front-endy a strojovo čitateľný **bitcode**.

2.3 LLVM C++ API

Keďže LLVM je napísané v C++, je pochopiteľné, že poskytuje veľkú škálu API taktiež v C++. LLVM si silno zakladá na využívaní C++ STL (štandardná knižnica) a preto aj API, ktoré poskytuje využíva podobné princípy ako C++. Malé rozdiely sú v tom, že využíva vlastné šablóny na statické a dynamické pretypovania alebo nevyužíva triedu *std::string* z STL. Implementuje si vlastnú, funkčne rovnakú triedu *StringRef* avšak s vnútornou implementáciou viac podobnou reprezentácií vhodnej do IR. Najväčšia časť C++ API ktoré LLVM poskytuje sa skladá z frameworku na písanie priechodov.

2.3.1 LLVM Priechody

Čo sú priechody a načo sa používajú sme už uviedli v časti 2.1.2. V tejto časti sa bližšie zameriame na to aké druhy priechodov existujú a na aké účely sa používajú. Podľa toho, ako priechod funguje, volíme z ktorej z nasledujúcich tried bude náš priechod dediť: *ModulePass, CallGraphSCCPass, FunctionPass, LoopPass, RegionPass alebo BasicBlockPass*[5].

Samozrejeme existujú ešte iné triedy, z ktorých môže nový priechod dediť, avšak pre potreby toho projektu ich nie je potrebné zahrnúť do tohoto textu. Podľa názvu je už približne možné zistiť, ako častou kódu sa bude daný priechod zaoberať. Nové priechody typicky preťažujú metódy triedy *Pass*, ako sú `doInitialization()` a `doFinalization` a rovnako aj metódy príslušnej triedy, z ktorej dedia (napr. `runOnFunction()` v prípade dedenia z triedy *FunctionPass*)

Priechody môžu byť hierarchicky zoradené takto:

- **ModulePass** - priechod prechádza programom ako celkom.
- **CallGraphSCCPass** - priechod prechádza programom v poradí volania funkcií.
- **FunctionPass** - priechod prechádza každou funkciou.
- **LoopPass** - priechod prechádza jednotlivými cyklami.
- **RegionPass** - priechod prechádza zloženými príkazmi.
- **BasicBlockPass** - priechod prechádza základnými blokmi programov(tj. časťami kódu, kde sa nemení tok programu).

Priechody majú všestranné využitie od optimalizácií až po analýzu kódu. V nasledujúcej kapitole navrhneme priechod, ktorý bude schopný vypísať východziu hodnotu globálnej premennej za účelom zistenia hodnoty parametru modulu jadra Linuxu.

Kapitola 3

LLVM priechod pre získanie východzích hodnôt globálnych premenných

Cieľom nasledujúcej kapitoly je popísať navrhnutý LLVM priechod[1] ktorý je schopný extrahovať východziu hodnotu globálnej premennej a vypísať ju na výstup. Tento priechod dedí od triedy *ModulePass*, pretože trieda *Module* ako jediná poskytuje možnosť vyhľadať v module programu jeho globálne premenné. Ak je priechod volaný bez parametrov, vypíše všetky hodnoty globálnych premenných v náhodnom poradí(teda nie v poradí, v akom sa nachádzajú v zdrojovom súbore) a to tak, že iteruje cez triedu *GlobalVariableList* a na východziu hodnotu každej premennej volá metódu `writeConstant(Constant *C)`. Táto metóda bude ďalej popísaná v sekcií 3.2.

Naopak, ak chceme vypísať východziu hodnotu konkrétnej premennej, zavoláme priechod s parametrom `-var`. Danú premennú získame pomocou metódy `getGlobalVariable()` triedy *Module* ktorá nájde premennú podľa mena. Táto metóda vracia hodnotu typu *GlobalVariable* s ktorou budeme naďalej pracovať. Východziu hodnotu premennej získame pomocou metódy `getInitializer()` ktorá je metódou triedy *GlobalVariable*. V jazyku C sú všetky globálne premenné podľa štandardu inicializované, okrem premenných deklarovaných s modifikátorom `extern`. Tu sa však jedná o deklaráciu miesto definície a preto ich budeme ignorovať.

Následne potrebujeme získať hodnotu v človeku čitateľnej forme. To už nie je tak jednoduché zistiť, pretože hodnota globálnej premennej môže byť rôznych typov. LLVM využíva triedu *Constant*, ktorá v sebe ukladá hodnotu danej premennej, tak ako ju vnútorne reprezentuje v bitkóde. Túto hodnotu dekódujeme tak, že pretypujeme triedu *Constant* na jednu z jej príslušných podtried. O túto funkcionalitu sa stará funkcia `writeConstant()` ktorá má dva parametre, výstupný prúd typu *raw_ostream* a hodnotu typu *Constant*. Ďalej o tejto funkcií v sekcií 3.2.

3.1 CMake

CMake je nástroj na automatizáciu prekladu zdrojových súborov[1]. LLVM na zostavenie svojich nástrojov a knižníc používa CMake a preto keď píšeme kód, ktorý využíva niečo z knižnice LLVM, najjednoduchšiu voľbou je práve použitie CMake. Hlavnou prednosťou

[1] GlobalVariablePass[7]

CMake je jeho jednoduchosť a aj preto ho vývojári LLVM využili na zostavenie projektu. V nasledujúcej sekcií je zobrazené ako prebieha zostavenie priechodu mimo zdrojovej štruktúry LLVM.

3.1.1 Zostavenie priechodu mimo zdrojovej štruktúry LLVM

V tejto časti popíšem proces, akým sa linkujú zdrojové súbory priechodu so zdrojovými súbormi LLVM a jeho knižnicami[3]. Na to aby sme mohli linkovať knižnice LLVM do nášho priechodu musíme inicializovať potrebné premenné prostredia do CMake. O to sa stará direktíva `find_package` ktorá s parametrom `LLVM REQUIRED CONFIG` vyhľadá v adresári, kde je inštalované LLVM, súbor s menom LLVMConfig.cmake a podľa neho nastaví príslušné premenné prostredia. Aby sme potom mohli následne vložiť náš priechod do adresára zdrojových súborov LLVM pridáme nasledujúci kód do súboru **CMakeList.txt**:

```
list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
include(AddLLVM)
```

Tieto direktívy nám zároveň umožnia využívať dalšie direktívy v CMake ktoré definuje LLVM. Ďalej je potrebné zariadiť aby bol priechod preložený s parametrom `-fno-rtti`, pretože LLVM je zvyčajne preložené s týmto parametrom a bez neho by použitie nášho priechodu v nástroji `opt` zahlásilo chybu linkovania.

3.2 Trieda Constant

Trieda *Constant* je podtriedou triedy *Value*, čo znamená, že je schopná uchovávať nejakú hodnotu[4]. Ako už z názvu vyplýva táto hodnota je konštantná, čiže nemenná. Inicializátory globálnych premenných sú inštanciami triedy *Constant* čo znamená, že hodnota globálnych premenných je uchovaná v nich. Avšak trieda *Constant* je obalový typ pre rôzne typy aké môžu globálne premenné v LLVM IR nadobúdať. Pre naše potreby však musíme túto hodnotu prekonvertovať do človeku čitateľnej formy a teda pretypovať ju na príslušnú na podtriedu podľa jej typu. LLVM definuje vlastné pretypovacie šablóny a tie je možné využiť pri určovaní správneho typu inicializátoru. Každá trieda v LLVM obsahuje statickú metódu `classof`, ktorá určuje na akú triedu sa smie daná trieda pretypovať. To znamená že keď sa nám podarí úspešne pretypovať triedu na jednu z jej podtried, zistili sme si vlastne jej typ. Danú triedu *Constant* je možné pretypovať na nasledujúce podtriedy:

- ConstantInt - reprezentuje všetky celočíselné hodnoty rôznych bitových šíriek, tj. aj znaky (bitovej šírky 8).
- ConstantFP - reprezentuje desatinné čísla.
- ConstantArray - reprezentuje konštantné pole.
- ConstantDataArray - reprezentuje dáta uložené v poradí za sebou ako pole. Metódou `getElementAsConstant()` je možné získať položku na indexe, ktorý je daný ako parameter metódy. Táto metóda vracia položky ktoré sú inštanciami triedy *Constant* a na ktoré aplikujeme postup popísaný vyššie. V prípade, že položky tejto triedy sú znaky môžeme vypísať danú triedu ako reťazec.
- ConstantStruct - reprezentuje štruktúry a k jej zložkám pristupujeme pomocou metódy `getOperand()`. Ďalej už iba rekurzívne aplikujeme uvedený postup .

- ConstantAggregateZero - reprezentuje prvky ktoré neboli inicializované a sú teda automaticky inicializované na nulu.
- ConstantPointerNull - reprezentuje ukazateľ ktorý ukazuje na null(teda neukazuje nikam).
- ConstantExpr - viz nasledujúca podsekcia 3.2.1

3.2.1 Podtrieda ConstantExpr

Podtrieda ConstantExpr je trochu komplikovanejšia, ako ostatné podtriedy a to preto, že okrem danej hodnoty je nositeľom výrazu. Najčastejším prípadom je, že daná trieda je nositeľom inštrukcie `getElemPtr` ktorá sa využíva na získanie ukazateľa do pamäte. V našom priechode sa s týmto stretávame, ak chceme vypísať hodnotu reťazca, ktorý nie je definovaný ako pole, ale ako ukazateľ na oblasť pamäte kde sa daný reťazec nachádza. V tom prípade pomocou metódy `getOperand()` dostávame hodnotu na ktorú daný ukazateľ ukazoval. Táto hodnota je typu *Value* a teda je potrebné ju pretypovať na nejakú podtriedu. My si zvolíme *GlobalVariable*, aby sme sa uistili že hodnota na ktorú odkazuje je naozaj globálne premenná. V prípade úspešného pretypovania s hodnotou typu *GlobalVariable* už ďalej pracujeme ako sme popísali v predchádzajúcich častiach.

Kapitola 4

Experimenty

V tejto kapitole sa budeme zaoberať overovaním funkčnosti priechodu napísaného v predchádzajúcej kapitole. Overovanie funkčnosti bude prebiehať v Docker kontajneri v ktorom beží nástroj DiffKemp. Daný kontajner si stiahneme a spustíme pomocou príkazov:

```
$ docker pull viktormalik/diffkemp-devel
$ docker run -ti viktormalik/diffkemp-devel bin/bash
```

V kontajneri si potom naklonujeme repozitár ktorý obsahuje náš priechod:

```
$ git clone https://github.com/Petku/GlobalVariablePass.git
```

V danom repozitári sa nachádzajú dva zdrojové kódy(`sys.ll` a `coredump.ll`), ktoré zodpovedajú súborom `kernel/sys.c` a `fs/coredump.c` z verzie jadra Linuxu 3.10.0-862, preložených do internej reprezentácie LLVM. Na týchto súboroch budeme testovať výstupy nášho priechodu. Zo zdrojových kódov vieme, že očakávané hodnoty testovacích premenných sú:

Súbor	**Názov premennej**	**Typ**	**Očakávaný výstup**
sys.ll	poweroff_cmd	string	`"/sbin/poweroff"`
	overflowuid	int	65534
	C_A_D	int	1
	cad_pid	struct pid *	`null`
coredump.ll	core_pattern	string	`"core"`
	core_pipe_limit	int	0
	core_uses_pid	int	0

Predtým než začneme získavať východzie hodnoty daných premenných musíme si náš priechod najskôr preložiť. Priechod preložíme nasledujúcimi príkazmi:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Keď už máme náš priechod preložený, zavoláme ho s parametrom `-var <meno_premennej>` pre dané testovacie premenné a výstup porovnáme s očakávanými výsledkami. Priechod spustíme nasledujúcim príkazom:

```
$ opt -load LLVMGlobVars.so -var meno_premennej -globvar < subor.ll > /dev/null
```

Pre zjednodušenie si vytvoríme nasledujúci skript(v repozitári pomenovaný **test.sh**):

```
for variable in poweroff_cmd overflowuid C_A_D cad_pid
do
    opt -load LLVMGlobVars.so -var variable -globvar < sys.ll > /dev/null
done
for variable in core_pattern core_pipe_limit core_uses_pid
do
    opt -load LLVMGlobVars.so -var variable -globvar < coredump.ll > /dev/null
done
```

Spustenie tohoto skriptu vyzerá následne:

```
$ chmod u+x test.sh
$ ./test.sh
/sbin/poweroff
65534
1
null
core
0
0
```

Výstup nášho priechodu sa zhoduje z očakávanými výstupmi, takže overovanie funkčnosti nášho priechodu dopadlo úspešne.

Kapitola 5

Záver

Cieľom projektovej praxe bolo vytvoriť priechod ktorý bude získavať a vypisovať východzie hodnoty globálnych premenných zodpovedajúce východzím hodnotám parametrov modulu jadra Linuxu. Daný priechod je napísaný v C++ a v budúcnosti je v pláne ho prepísať v skriptovacom jazyku Python, aby ho bolo možné zaintegrovať do projektu DiffKemp ktorý je taktiež napísaný v Pythone. Vytvorený priechod bol testovaný v prostredí Docker kontajneru, v ktorom je vyvíjaný aj nástroj DiffKemp aby daný priechod fungoval aj na konkrétnej verzii LLVM, ktorú nástroj používa.

V tejto práci je popísaný znalostný základ potrebný na napísanie LLVM priechodu a taktiež je tu podrobne opísaný vývoj priechodu ktorý extrahuje východzie hodnoty globálnych premenných zo zdrojových súborov modulov jadra Linuxu. Popisuje triedy potrebné na spracovanie hodnôt z internej reprezentácie LLVM a ukazuje programovacie techniky pri písaní priechodov ako sú dynamické pretypovania na podtriedy, ktoré reprezentujú napr. rôzne typy globálnych premenných.

Táto práca slúži ako znalostný základ na písanie priechodov a snaží sa priblížiť sa k vývoju zložitejších analyzačných priechodov.

Literatúra

[1] Cedilnik, A.; Hoffman, B.; King, B.; aj.: CMake.
URL https://cmake.org/

[2] LLVM Project: The LLVM Compiler Infrastructure.
URL http://llvm.org/

[3] LLVM Project: Developing LLVM passes out of source. 21.01.2019.
URL
https://llvm.org/docs/CMake.html#developing-llvm-passes-out-of-source

[4] LLVM Project: LLVM Programmers Manual: The Constant class and subclasses. 21.01.2019.
URL http:
//llvm.org/docs/ProgrammersManual.html#the-constant-class-and-subclasses

[5] LLVM Project: Writing LLVM pass. 21.01.2019.
URL http://llvm.org/docs/WritingAnLLVMPass.html

[6] Malík, V.: DiffKemp.
URL https://github.com/viktormalik/diffkemp

[7] Patrik, N.: GlobalVariablePass.
URL https://github.com/Petku/GlobalVariablePass

[8] Samspon, A.: LLVM for grad students. 03.08.2015.
URL https://www.cs.cornell.edu/~asampson/blog/llvm.html