Пројекат из Дизајна и анализе алгоритама

Статички анализатор за x86-64 асемблер

Милош Павловић

Математички факултет, Универзитет у Београду

e-mail: mr22008@alas.matf.bg.ac.rs

Јануар 2026.

Садржај

1. Поставка проблема

Циљ овог пројекта је развој **статичког анализатора за x86-64 асемблер** који омогућава проучавање контроле тока програма и идентификацију критичних структура у коду. Фокус је на изградњи графа контроле тока (CFG) и откривању јако повезаних компоненти (SCC).

Програм се представља као усмјерен граф чији чворови означавају основне јединице извршења, а ивице могуће прелазе током извршавања. За потребе овог пројекта, свака инструкција из редукованог скупа представљена је као појединачни чвор у коначном графу, што поједностављује изградњу CFG-а и визуелизацију SCC. Иако је рад ограничен на редуковани скуп инструкција, концепт се може уопштено применити на комплетан скуп x86-64 инструкција или друге архитектуре.

Анализа SCC омогућава идентификацију петљи, циклуса и критичних тачака контроле тока, што је корисно за оптимизацију, детекцију грешака и побољшање стабилности програма. Бонус је детекција недостижног кода.

За рјешење проблема SCC користи се **Тарџанов алгоритам**, о коме ће бити више ријечи у наставку. Осим њега, у наставку је наведен и донекле сродан, Габов алгоритам.

Пројекат практично конструише CFG из x86-64 програма и примјењује Тарџанов алгоритам за детекцију SCC. Редуковани скуп инструкција и појединачни чворови олакшавају визуелизацију и тестирање алата, док принципи примјене остају универзални и могу се применити и на сложеније програме.

Укупно, пројекат повезује теоријске основе алгоритама за SCC са практичном применом у статичкој анализи x86-64 кода, пружајући едукативан и функционалан алат за идентификацију структура контроле тока.

2. Граф контроле тока

У овој секцији укратко ћемо се позабавити графом контроле тока (CFG). Он представља формални модел програма у облику усмјереног графа. Чворови графа представљају инструкције или логичке цјелине програма, док ивице описују могуће прелазе током његовог извршавања. CFG је један од основних алата у статичкој анализи програма, јер експлицитно приказује све потенцијалне путеве извршавања.

У општем случају, чворови CFG-а представљају основне блокове — максималне секвенце инструкција без унутрашњих гранања. Међутим, ради једноставности имплементације и јасније визуелизације резултата, у овом пројекту је усвојен поједностављен модел: **свака инструкција из разматраног скупа представља један чвор графа**. Иако овакав приступ повећава број чворова, он не мијења суштину анализе и омогућава директну везу између асемблерског кода и структуре CFG-а.

Ивице у графу контроле тока одређују се на основу семантике инструкција. За секвенцијалне инструкције постоји ивица ка наредној инструкцији, док инструкције гранања (условни и безусловни скокови) уводе више излазних ивица. На тај начин CFG обухвата све могуће путеве извршавања програма, укључујући петље и условне структуре.

Једном конструисан, CFG представља основу за даљу анализу. Посебно, проблем проналажења јако повезаних компоненти над овим графом омогућава детекцију циклуса и петљи у програму. Овај корак је кључан за разумевање контроле тока, као и за примену напреднијих анализа и оптимизација.

У наставку ће бити размотрени алгоритми за проналажење јако повезаних компоненти и њихова примјена на граф контроле тока добијен из x86-64 асемблерског кода.

3. Јаке компоненте повезаности

Једно од основних питања при раду са графовима јесте: за задати чвор одредити до којих чворова се може стићи кренувши из њега, као и из којих све чворова је он достижан.

Формално, на скупу чворова уводимо релацију достижности $u \sim v$ ако постоји пут од чвора u до чвора v. Компонента повезаности је скуп чворова такав да између свака два чвора у њему постоји пут.

Ако је граф неоријентисан, ово је релација еквиваленције и проблем се једноставно рјешава. Довољно је из необрађеног чвора покренути обилазак графа и сви достигнути чворови одређују тачно његову компоненту (овдје до изражаја долази симетричност $\sim$).

Међутим, код оријентисаних графова немамо гаранцију симетричности релације и зато уводимо појам **јаке компоненте повезаности (SCC)**. Примјетимо да у овом случају прости алгоритам обиласка не даје рјешење. Прецизније, могли бисмо покренути обилазак графа из сваког чвора и онда одређивати који чворови су достижни из текућег и из којих чворова је он достижан, но то даје неефикасан алгоритам.

Испоставља се да јаке компоненте повезаности имају велику примјену. Само неке од примјена су:

- детекција циклуса у софтверима,
- спрјечавање deadlock-а у оперативним системима,
- веб технологије (кластеризација и оптимизација сајтова),
- дизајн логичких кола,
- анализа друштвених мрежа.

Данас постоји низ алгоритама који решавају овај проблем ефикасно. У наставку су приказани Тарџанов и Габов алгоритам који достижу оптималну (линеарну) ефикасност и уводе нове, важне и врло примјењиве концепте.

4. Тарџанов алгоритам

Иницијална рјешења за проналажење јако повезаних компоненти користила су наиван приступ, идејно описан на претходној страни, чија је временска сложеност износила $\mathcal{O}(V \cdot (V + E))$. Хронолошки прво линеарно рјешење, сложености $\mathcal{O}(V + E)$, дао је Тарџан 1972. године у свом раду *"Depth-first search and linear graph algorithms"*.

Јасно је да је при раду са графовима фундаментална операција њихов обилазак; заправо, обилазак представља интегрални дио сваког алгоритма над графовима. Међутим, није пожељно да се граф изнова обилази велики број пута, јер се тиме неминовно добија сложеност од барем $\mathcal{O}(V^2)$. Стога ефикасни алгоритми, по правилу, морају извући што је могуће више информација у току једног обиласка.

Испоставља се да је обилазак у дубину (DFS) у већини примјена знатно кориснији у односу на обилазак у ширину (BFS). Разлог за то лежи у чињеници да, полазећи од неког чвора, можемо конструисати одговарајуће DFS дрво датог графа. Ово дрво нам омогућава увођење важних статистика, попут долазне (и одлазне) нумерације чворова, помоћу којих даље можемо карактерисати чворове и гране графа.

Тарџан је у свом раду увео изузетно важан нови концепт, познат као *lowlink* нумерација. Основна идеја је да се приликом обиласка графа, за сваки чвор забиљежи чвор са најмањом долазном нумерацијом до којег се можемо вратити полазећи од текућег чвора, крећући се гранама DFS стабла, након чега је дозвољено коришћење највише једне повратне гране (видјети илустрацију).

Друго важно запажање јесте да, ако граф посматрамо кроз његово DFS стабло, свака јако повезана компонента има јединствени коријен. Другим ријечима, за текући чвор важи да се он или налази у некој компоненти чији смо коријен већ посјетили и коју тек треба да обрадимо, или је он сам коријен неке компоненте (тј. цијела компонента се налази у његовом поддрвету DFS стабла). Одатле слиједи да је довољно идентификовати управо коријене компоненти.

Сада можемо дати једноставну, али важну карактеризацију:

Теорема 4.1. *Нека је $G = (V, E)$ усмјерен граф и нека је над њим извршена DFS претрага. За чвор $v \in V$ важи да је он коријен неке јако повезане компоненте ако и само ако из чвора v није могуће достићи ниједан чвор са мањим долазним бројем. Еквивалентно, чвор v је*

корijен јако повезане компоненте ако и само ако важи:

$$\mathit{lowlink}(v) = \mathit{preorder}(v).$$

На крају, треба уочити да у тренутку излазне обраде чвора v познајемо и његов `preorder` и `lowlink`. Све јако повезане компоненте које се налазе у његовом поддрвету ће у том тренутку или већ бити обрађене, или ће припадати истој компоненти као и сам чвор v.

За потребе имплементације довољно је увести још један стек, на коме ће се налазити обрађени чворови којима још није додијељена компонента. Псеудокод Тарџановог алгоритма дат је у наставку:

```
procedura Tarjan(v):
    preorder[v] ← dolazni_broj
    dolazni_broj ← dolazni_broj + 1
    lowlink[v] ← preorder[v]

    gurni v na stek S

    za svakog susjeda u od v:
        ako susjed nije posjecen onda
            Tarjan(susjed)
            lowlink[v] ← min(lowlink[v], lowlink[susjed])
        inace ako je susjed na steku S onda
            lowlink[v] ← min(lowlink[v], preorder[susjed])

    ako lowlink[v] = preorder[v] onda
        C ← prazna komponenta
        ponavljaj:
            w ← skini vrh steka S
            dodaj w u C
        dok w   v
        prijavi komponentu C
```

5. Габов алгоритам

Још један ефикасан алгоритам за рјешење проблема проналаска SCC компоненти јесте Габов алгоритам. Представио га је Харолд Н. Габов у раду из 1978. године: *"Path-based depth-first search for strong and biconnected components"*. Може се посматрати као алтернатива Тарџановом алгоритму, јер се суштински они заснивају на истом концепту.

За разлику од Тарџановог алгоритма у којем експлицитно одређујемо *lowlink* бројеве, Габов уводи идеју са два стека. На први стек (стек S) додајемо чворове за које још увијек нисмо одредили компоненту. На другом стеку (стек V) се налазе потенцијални кандидати за коријене компоненти. Важно је напоменути да чворове на оба стека додајемо у редослиједу DFS обиласка графа.

Јасно је да ће улогу *lowlink* бројева сада преузети други стек, јер да бисмо знали коријенски чвор компоненте морамо знати можемо ли од њега стићи до неког који је посјећен раније. Ту информацију пратимо на првом стеку (слично као код Тарџановог алгоритма). Ако се чвор u нашао на стеку прије чвора v, то значи да постоји пут од u до v. Ако затим, у тренутку обраде чвора v, видимо да он има грану ка чвору u који се већ налази на стеку, тада смо сигурни да су u и v у истој компоненти. Међутим, како се u нашао прије v, сигурни смо да v не може бити коријен те компоненте, те можемо скинути са другог стека (стек потенцијалних коријена) чвор v, али и све чворове са већим долазним бројем од u.

Са друге стране, у тренутку излазне обраде чвора u, ако се он налази на врху стека потенцијалних коријена, онда он заиста јесте коријен текуће компоненте. Тада са првог стека скидамо све чворове закључно са чвором u (сви скинути чворови чине једну компоненту). Са другог стека скидамо чвор u.

Псеудокод Габовог алгоритма може се записати на следећи начин:

```
procedura Gabow(v):
    preorder[v] ← dolazni_broj
    dolazni_broj ← dolazni_broj + 1

    gurni v na stek S
    gurni v na stek P

    za svakog susjeda u od v:
        ako susjed nije posjecen onda
            Gabow(susjed)
        inace ako je susjed na steku S onda
            dok preorder[v] > preorder[susjed]:
                ukloni vrh steka P

    ako je vrh steka P = v onda
        C ← prazna komponenta
        ponavljaj:
            w ← skini vrh steka S
            dodaj w u C
        dok w   v
        ukloni vrh steka P
        prijavi komponentu C
```

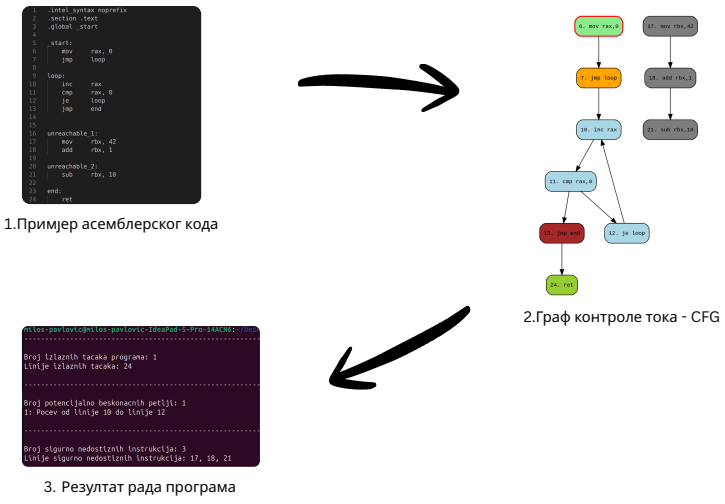
6. Резултат

У последњој секцији приказан је рад програма на конкретном примеру (слика испод). Изабран је мали пример, али такав да се јасно идентификују основне особине пројекта.

У првом кораку, на основу датог кода креирамо граф контроле тока. Почетна инструкција је уоквирена црвеном бојом, свака компонента повезаности обојена је различитом бојом, док су недостижне инструкције означене сивом.

Важно је истаћи да због једноставности обраде су узете неке додатне претпоставке: у коду се не појављују двије лабеле непосредно једна испод друге, нема функције call већ једном програму одговара тачно једна функција. Осим тога, идентификоване компоненте повезаности одговарају "спољашњим" циклусима (тј. петљама), али се могу јавити и угњеждене петље у оквиру једне јаке компоненте повезаности. Тај проблем се може ријешити детаљнијом анализом појединачних компоненти, нпр. анализом повратних грана, али је за потребе овог пројекта таква анализа изостављена.

На крају, последња слика приказује коначан резултат рада програма, односно да се програм понаша у складу са очекивањима.



Слика 1: Пример рада програма