Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Detekcia vzorov správania procesu v postupnosti adries

Bakalárska práca

2022
Matúš Hluch

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Detekcia vzorov správania procesu v postupnosti adries

Bakalárska práca

Študijný program: Informatika
Študijný odbor: Informatika
Školiace pracovisko: Katedra informatiky
Školiteľ: Ing. Dušan Bernát, PhD.

Bratislava, 2022
Matúš Hluch

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Matúš Hluch
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Detekcia vzorov správania procesu v postupnosti adries
Pattern recognition in address traces of a process

Anotácia: Postupnosti adries ku ktorým procesy počas svojho vykonávania pristupujú nie sú náhodné, ale je možné v nich nájsť isté vzory ktoré sú dôsledkom jednak princípov pamäťovej lokality, ale aj typického správania sa procesu. Podrobná znalosť týchto vzorov, respektíve správania sa procesu z hľadiska pamäťových prístupov, je dôležitá nie len pri návrhu pamäťového systému, ale aj pri analýze a odhalovaní škodlivých programov.

Analyzujte možné spôsoby zaznamenania postupnosti adries ku ktorým zvolený proces pristupuje počas svojho vykonávania (mechanizmus ptrace, Intel Pin tool a pod.). Pre zvolené prostredie (architektúra a operačný systém) navrhnite mechanizmus zaznamenávania adries na ktoré proces pristupuje (či už v inštrukčnej alebo dátovej pamäti), ako aj spôsob ich spracovania a interpretácie (napr. odhalenie neželaných aktivít procesu). Zvážte pritom možnosť transformácie postupnosti adries na graf. Riešenie implementujte a overte v testovacom prostredí.

Vedúci: Ing. Dušan Bernát, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 27.10.2021

Dátum schválenia: 04.11.2021 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....
študent vedúci práce

Poďakovanie: Ďakujem školiteľovi bakalárskej práce Ing. Dušanovi Bernátovi, PhD. za cenné rady, usmernenie, pomoc a pripomienky, ktoré mi významnou mierou pomohli pri tvorbe tejto práce.

Abstrakt

V našej práci sa venujeme prístupom procesov do operačnej pamäte. V postupnosti pristupovaných adries hľadáme vzory a skúmame ďalšie vlastnosti prístupov programov do pamäte. Špecifikom našej práce je vyjadrenie postupnosti pristupovaných adries grafovou reprezentáciou a skúmanie jej vlastností. V práci vytvárame viacero nástrojov a vizualizácií, ktoré umožňujú analyzovať niektoré vlastnosti či už spomínanej grafovej reprezentácie alebo postupnosti adries, a tak odhaliť aj niektoré vlastnosti analyzovaných programov.

Kľúčové slová: operačná pamäť, prístupy do pamäte, prístupové vzory, graf

Abstract

In this thesis we predominantly deal with memory accesses. We investigate memory access patterns in a sequence of accessed addresses and we study their properties and additional qualities of memory accesses. Our specific approach lies in representing the sequence of accessed addresses as a graph while at the same time we analyse its characteristics. In this thesis we develop several tools and visualisations, which enable analysis of the already mentioned graph representation or the analysis of the sequence of accessed addresses and thanks to that we are able to detect some features of analysed programs.

Keywords: computer memory, memory accesses, memory access patterns, graph

Obsah

Úvod **1**

1 Analýza problematiky **3**

1.1 Operačná pamäť . . . 3

1.1.1 Pamäťová hierarchia . . . 3

1.1.2 Architektúry pamäte . . . 4

1.2 Proces . . . 5

1.3 Pracovná množina . . . 5

1.4 Rozloženie pamäte . . . 6

1.5 ASLR . . . 6

1.6 Princíp lokality . . . 7

1.6.1 Sekvenčná lokalita . . . 7

1.6.2 Priestorová lokalita . . . 8

1.6.3 Časová lokalita . . . 8

1.7 Nástroje na záznam adries . . . 8

1.7.1 Pin Tool . . . 9

1.7.2 Dynamorio . . . 10

1.8 Existujúce práce . . . 11

2 Návrh riešenia **13**

2.1 Výber analyzovaných procesov . . . 13

2.2 Architektúra riešenia . . . 14

2.3 Reprezentácia postupnosti adries . . . 15

2.4 Nástroje pracujúce na zozname adries . . . 15

2.4.1 Princípy lokality . . . 15

2.4.2 Minimálny počet potrebných adries . . . 16

2.4.3 Rozdelenie adries podľa typu . . . 17

2.5 Nástroje pracujúce s grafom adries . . . 17

2.5.1 Silno súvislé komponenty . . . 18

2.5.2 Kompresia grafu . . . 18

3 Implementácia riešenia **19**

3.1 Programovací jazyk a knižnice 19

3.2 Implementácia nástrojov 20

3.2.1 Reprezentácia postupnosti adries 20

3.2.2 Princípy lokality 21

3.2.3 Minimálny počet potrebných adries 21

3.2.4 Silno súvislé komponenty 21

3.2.5 Kompresia grafu 22

4 Overenie riešenia **23**

4.1 Silno súvislé komponenty 23

4.2 Minimálny počet potrebných adries 25

4.3 Princípy lokality 28

4.4 Ďalšie namerané dáta 30

4.4.1 Kompresia grafu 30

4.4.2 Rozdelenie typov adries 30

Záver **33**

A Prílohy **1**

A.1 Požiadavky na spustenie 1

A.2 Popis súborov 1

Úvod

Operačné systémy patria k najvýznamnejším oblastiam informatiky. V dnešnej dobe sme stále viac obklopení technológiami, bez ktorých si väčšina ľudí nevie predstaviť svoju prácu, trávenie voľného času a takmer všetky činnosti, s ktorými sa v živote stretávame. Neoddeliteľnou softvérovou časťou počítačov, mobilov a iných zariadení je operačný systém. Z dôvodu ich všadeprítomnosti je mimoriadne dôležité, aby tieto operačné systémy a práca s nimi bola efektívna a bezpečná.

Jednou z dôležitých úloh operačných systémov je správa operačnej pamäte. Téme súvisiacej s operačnou pamäťou sme sa venovali aj v našej práci. Každý jeden program počas svojho vykonávania využíva operačnú pamäť. Program zapisuje do operačnej pamäte a číta inštrukcie, premenné, dátové štruktúry a dáta, ktoré sú potrebné na vykonanie. Aj jednoduchšie, kratšie, príkazy počas ich vykonávania spravia tisíce až státisíce takýchto prístupov do pamäte.

Preto je zaujímavé skúmať vlastnosti týchto prístupov do pamäte, či už z hľadiska efektivity alebo správania procesu. V praxi sa analyzujú prístupy programov do pamäte z rôznych dôvodov. Jedným z nich je zrýchlenie vykonávania programov zefektívnením týchto prístupov. V posledných pár rokoch sa výskum v tejto oblasti zameral aj na bezpečnosť a odhaľovanie nevhodného správania programov - napríklad vírusov.

Peter J. Denning a kolektív v 70. rokoch minulého storočia [15] zistili, že v postupnosti prístupov do pamäte sa nachádzajú určité vzory. My sme sa v našej práci taktiež venovali vlastnostiam a vzorom v prístupoch do pamäte. Vzory a vlastnosti sme analyzovali v postupnosti pristupovaných adries ale aj v nami skonštruovanej grafovej reprezentácii postupnosti adries. Cieľom našej práce je vytvoriť viacero nástrojov na analýzu prístupov do pamäte pracujúcich so spomenutými reprezentáciami.

Na začiatok - v prvej kapitole popisujeme základné pojmy týkajúce sa našej práce, ktoré sú potrebné k pochopeniu ďalších kapitol. Taktiež v nej spomenieme ďalšie publikácie venujúce sa podobnej problematike, uvedieme aké nástroje, knižnice a jazyk sme pri našej práci použili. V druhej kapitole navrhujeme riešenie, popisujeme nástroje, ktoré sme navrhli a v nasledujúcej spomenieme spôsob ich implementácie. V závere, v poslednej kapitole, zhodnotíme naše nástroje na vybraných programoch a uvedieme dosiahnuté výsledky.

Kapitola 1

Analýza problematiky

Prístupy procesov na pamäťové miesta a ich skúmanie, predstavujú veľmi dôležitú súčasť viacerých oblastí informatiky. Samozrejmou kategóriou sú operačné systémy, pričom táto téma je zaujímavá pre rôzne oblasti, pre ktoré je dôležitá rýchlosť práce s dátami, ako sú napríklad databázové systémy alebo oblasť informatiky venujúca sa algoritmom a dátovým štruktúram. Prvé práce zaoberajúce sa touto témou, boli preto písané už pár rokov po konštrukcii prvých počítačov s operačnou pamäťou. Vďaka tomu existuje mnoho pojmov a poznatkov venujúcim sa práve tejto téme. V tejto kapitole vysvetlíme základné znalosti potrebné k pochopeniu práce a spomenieme iné diela venujúce sa podobnej problematike. Taktiež popíšeme aké nástroje sa používajú na záznam postupnosti adries, na ktoré proces pristupoval.

1.1 Operačná pamäť

Operačná pamäť je jednou z hlavných častí počítača. Táto pamäť je v porovnaní so sekundárnou pamäťou výrazne rýchlejšia, avšak pomalšia ako vyrovnávacia pamäť na procesore.

Je využívaná vo viacerých oblastiach. Slúži najmä na ukladanie dát, ktoré budú v blízkej dobe používané. Ide o niektoré časti operačného systému, inštrukcie programov a dáta, ktoré tieto programy spracovávajú. Ďalším využitím môže byť napríklad RAM-Disk, kde sa vďaka rýchlosti operačnej pamäte, určitá jej časť využíva ako sekundárna pamäť. Z hardvérového hľadiska je operačná pamäť volatilná, teda po vypnutí energie sa dáta stratia. Je adresovaná priamo, v súčasnej dobe, na väčšine bežných počítačov 64 bitmi [7].

1.1.1 Pamäťová hierarchia

Operačná pamäť je súčasťou pamäťovej hierarchie. Bez pamäťovej hierarchie by boli počítače výrazne pomalšie ako skutočne sú. Je to spôsobené tým, že operačná pamäť,

a iné ďalšie pomalšie pamäte, sú niekoľkonásobne pomalšie oproti rýchlosti procesora.

Preto sa medzi registre procesora a operačnú pamäť pridala cache alebo vyrovnávacia pamäť. Vyrovnávacia pamäť je rýchlejšia, ako operačná pamäť ale na druhej stane je výrazne menšia, pretože je veľmi drahá. To je v podstate hlavnou myšlienkou pamäťovej hierarchie: ide o postupnosť pamätí, od najvzdialenejšej od procesora k najbližšej, poprepájané jedna za druhou. Tieto pamäte, ak postupujeme od najvzdialenejšej, majú postupne menšiu kapacitu ale zato väčšiu rýchlosť. Zároveň čím bližšie k procesoru pamäť je, tým je drahšia na jednotku priestoru. V dnešnom bežnom počítači, môžeme postupne ísť od najväčších pamätí, ktoré sú pripojené cez sieť, potom externým diskom a médiám, pevným diskom pripojeným priamo v počítači, následne k operačnej pamäti a niekoľkým vrstvám postupne sa zmenšujúcich a zrýchľujúcich sa vyrovnávacích pamätí až po registre na procesore.

Pamäťová hierarchia by však neurýchľovala výpočet, ak by neexistovali v prístupoch do pamätí vzory. Ak by boli prístupy do pamäte úplne náhodné, dáta z pomalších pamätí by sa museli neustále presúvať do rýchlejších a riešenie s pamäťovou hierarchiou by mohlo byť dokonca pomalšie ako riešenie bez neho. Vďaka vlastnosti bežných procesov pristupovať do pamäte s nejakými vzormi, sa môžu dáta presúvať tak, aby vyhovovali prístupovým vzorom, napríklad sa môžu presúvať väčšie bloky okolo adresy, na ktorú bolo pristúpené [16].

1.1.2 Architektúry pamäte

Dáta a inštrukcie môžu byť v pamäti uložené rôznymi spôsobmi. Existujú dve známe architektúry pamätí. Jednou z prvých, navrhnutých už v roku 1947 je Harvardská architektúra. V tejto architektúre je pamäť rozdelená na dve časti. Jedna časť pamäte je určená na ukladanie inštrukcií, nazývaná aj inštrukčná pamäť a je určená iba na čítanie. Druhá časť pamäte slúži na ukladanie dát, nazýva sa tiež dátová pamäť a môže sa z nej čítať aj do nej zapisovať. Ďalšou známou architektúrou je Von Neumannova architektúra, častokrát nazývaná aj Princetonská architektúra. V tomto prístupe pamäť nie je rozdelená na dve časti ale aj dáta aj inštrukcie sa ukladajú do jednej pamäte určenej aj na zápis aj na čítanie.

Obidve architektúry majú svoje výhody aj nevýhody. Vďaka oddelenným pamätiam pri Harvardskej architektúre, môžu byť pamäte prispôsobené osobitne dátam a inštrukciám napríklad veľkosťou jedného slova. Na rozdiel od Von Neumannovej architektúry má procesor možnosť čítať z oboch pamätí naraz aj inštrukcie aj dáta, čo umožňuje rýchlejšie vykonávanie programov. Na druhej strane Von Neumannova architektúra je lacnejšia na konštrukciu a zároveň jednoduchšia na správu.

Na väčšine počítačov dnešnej doby sa využíva modifikovaná Harvardská architektúra. Táto architektúra je vlastne kombináciou predošlých dvoch. Modifikovaná Har-

vardská architektúra používa jednu pamäť aj na dáta aj na inštrukcie. Kvôli zrýchleniu vykonávania, sa medzi procesor a pamäť pridáva vyrovnávacia pamäť, ktorá už je rozdelená na vyrovnávaciu pamäť pre dáta a vyrovnávaciu pamäť pre inštrukcie. Preto má zmysel, aj v našej práci, skúmať prístupy do pamäte oddelene pre inštrukcie a dáta [11].

1.2 Proces

Na adresy do operačnej pamäte pristupujú procesy, ktoré sú jednou z hlavných súčastí operačných systémov. Proces je definovaný ako práve vykonávaný program. V operačných systémoch so zdieľaním času v jednej chvíli, môže existovať viacero procesov naraz, dokonca viac ako je počet dostupných fyzických procesorov.

Celý proces sa dá v každom momente opísať jeho časťami - registrami a jeho adresným priestorom. Na operačnom systéme, ktorý v práci budeme využívať sa používa virtuálna pamäť. Ak sa používa virtuálna pamäť, každý proces má súvislý virtuálny adresný priestor, čo z pohľadu procesu vyzerá, ako keby mal úplne vlastnú samostatnú pamäť. Každý prístup do tejto virtuálnej pamäte na virtuálnu adresu je následne prekladaný na fyzické adresy za pomoci časti procesora *Memory Managment Unit* [2].

1.3 Pracovná množina

Pracovná množina je pojem, ktorý má veľmi blízko pojmu priestorová lokalita. Obdobne ako s priestorovou lokalitou aj s pracovnou množinou prišiel Denning vo svojom výskume [5]. S príchodom multiprogramovania a virtuálnej pamäte došlo k výraznému spomaleniu niektorých programov. Vo virtuálnej pamäti môžu byť niektoré stránky uložené mimo pamäte RAM napríklad na disku. Dynamická alokácia pamäte pri multiprogramovaní nebola dokonalá a procesor musel neustále neefektívne načítavať stránky do pamäte z disku, na ktorý boli predtým presunuté.

Tento problém nazvali *thrashing*. Denning začal skúmať množinu stránok, ktoré programy počas vykonávania používali. V tom čase zadefinoval označenie pracovnej množiny pomocou $W(t, T)$, čo je množina stránok, na ktoré proces pristupoval počas doby pevnej dĺžky $(t-T, t)$, kde t a T určujú virtuálny čas určený počtom prístupov do pamäte. Takýto virtuálny čas použil preto, aby sa vyhol odchýlkam, ktoré sú spôsobené rôznymi prerušeniami. Pomocou experimentovania s pracovnou množinou Denning a kolektív zistili, že ak sa nenachádza celá pracovná množina procesu v pamäti, nastáva *thrashing*.

1.4 Rozloženie pamäte

Dáta sú v pamäti rozložené pre jednotlivé procesy do oblastí. Pri experimentálnom overovaní rôznych vlastností prístupov do pamäte budeme v tejto práci využívať operačný systém Linux a preto v tejto časti stručne a zjednodušene popíšeme oblasti virtuálneho adresového priestoru pre proces v Linuxe. Z hľadiska vzorov pri prístupoch do pamäte môže byť toto rozloženie zaujímavé, napríklad niektoré časti programov ako je rekurzívna funkcia, môžu častejšie pristupovať do niektorej oblasti ako do inej, v prípade rekurzívnej funkcie touto oblasťou môže byť zásobník.

Na začiatku virtuálneho adresového priestoru procesu sú v Linuxe namapované tieto oblasti: *text*, ktorá obsahuje inštrukcie, dátová oblasť obsahujúca inicializované dáta a oblasť BSS v ktorej sú neinicializované dáta. Za nimi nasleduje oblasť nazvaná *heap* alebo halda. V halde sa nachádza dynamicky alokovaná pamäť, ktorá je upravovaná systémovými volaniami, ako sú **brk()**, **sbrk()** a **mmap()**. Je teda alokovaná počas behu programu. V ďalšej oblasti sú mapovania pamäte na knižnice alebo mapovania na súbory. Predposlednou oblasťou virtuálneho adresového priestoru v Linuxe je zásobník. Do zásobníka sa dočasne ukladajú dáta pri volaní funkcie. V zásobníku sú uložené lokálne premenné, parametre funkcií a návratová adresa po ukončení vykonávania funkcie. Ak je nastavené znáhodnenie adresového priestoru, potom sú oblasti halda, zásobník a mapovania pamäte rozmiestnené tak, že sú medzi nimi náhodne veľké medzery. Na konci virtuálneho adresového priestoru je pridelená jeho časť jadru operačného systému. Do tejto oblasti používateľ nemôže zapisovať ani z nej čítať [3].

1.5 ASLR

Address Space Layout Randomisation alebo znáhodnenie adresového priestoru je technológia, ktorá je v dnešnej dobe využívaná vo väčšine operačných systémov. Funguje tak, že počiatočné adresy častí adresového priestoru programu, ako sú napríklad knižnice alebo zásobník, umiestňuje náhodne do pamäte. Cieľom tejto technológie je zamedziť alebo aspoň sťažiť rôzne útoky využívajúce zraniteľnosti pri prístupoch do pamäte. Na Linuxe je to nastaviteľná vlastnosť, ktorú je možné vypnúť.

Útoky využívajúce zraniteľnosti pamäte najčastejšie fungujú na základe nesprávnej implementácie programov v nízkoúrovňových jazykoch ako sú napríklad *C* alebo *C++*. Tieto programy umožňujú prácu s pamäťou na nízkej úrovni. V prípade nesprávnej implementácie je útočníkom neúmyselne umožnený prístup do pamäte. Následne vedia útočníci pomocou prístupu do pamäte ovplyvniť ďalšie vykonávanie programu.

Asi najznámejším a najjednoduchším typom takéhoto útoku je *buffer overflow*. Zraniteľnosť v tomto prípade vzniká pri zápise do buffera, ktorý prekračuje jeho veľkosť a v programe takáto situácia nie je ošetrená. Takto dokáže útočník zapisovať kdekoľ-

vek do pamäte procesu. Aby útočník ovplyvnil chod programu, môže prepísať časť pamäte s vykonávateľným kódom. Cieľom znáhodnenia adresového priestoru by malo byť znemožnenie takéhoto útoku. S použitím znáhodnenia adresového priestoru útočník nemôže vedieť, ktoré adresy v pamäti by mal prepísať. ASLR však nie je dokonalé riešenie tohto problému a dnes už existuje viacero riešení ako je ASLR možné obísť. Z dnes známych prekonaní ASLR existuje napríklad použitie hrubej sily, ktoré je aplikovateľné najmä pri 32 bitových systémoch alebo zistenie umiestnenia častí adresového priestoru cez rôzne postranné kanály [9].

1.6 Princíp lokality

V adresách, na ktoré procesy pristupujú, sa vyskytujú rôzne vzory. Procesy vo všeobecnosti pristupujú v určitom čase na niektoré adresy častejšie ako na ostatné. Pojmy princíp lokality alebo lokalita odkazov (locality of reference) označujú vlastnosť prístupov do pamäte, ktorou je zhromažďovanie prístupov procesov na niektoré časti adresového priestoru. So základným typom lokalít prišiel Denning, a to pri hľadaní riešenia urýchlenia práce s virtuálnou pamäťou a multiprogramovaním [6]. Vo svojich prácach objavil, už v roku 1967 a následne experimentálne potvrdil, viaceré typy princípov lokality.

V tejto časti práce popíšeme najbežnejšie typy lokalít a uvedieme zopár príkladov, kde sa môžu vyskytovať a čím sú spôsobené.

1.6.1 Sekvenčná lokalita

Sekvenčná lokalita je najtypickejším príkladom lokality. Pri sekvenčnej lokalite je pravdepodobnejšie, že proces bude pristupovať na postupne za sebou idúce adresy (jedna za druhou) teda lineárne. Presnejšie, ak proces pristúpi v čase t na adresu s číslom n, tak v čase $t + 1$ bude proces pristupovať k adrese s číslom $n + 1$.

Tento typ lokality môže vzniknúť ako následok viacerých vlastností programov. Jedným z príkladov môže byť jedna z najčastejšie používaných dátových štruktúr-jednorozmerné lineárne pole. Uvažujme lineárne pole, ktoré je implementované tak, že jednotlivé prvky poľa sú uložené na pamäťových miestach s adresami, ktoré nasledujú za sebou. Pri sekvenčnom prechádzaní takto implementovaného poľa cyklom, je možné sledovať v prístupoch do pamäte sekvenčnú lokalitu. Podobné je to aj s inštrukciami programov, ktoré sú uložené v pamäti jedna za druhou. Pri vykonávaní programu procesor načítava inštrukcie z pamäte a následne ich vykonáva. Okrem niektorých inštrukcií, ako napríklad inštrukcia skoku, inštrukcie volania alebo návratu z funkcie, procesor načítava z pamäte miesta s adresami za sebou, teda sekvenčne.

1.6.2 Priestorová lokalita

Sekvenčnú lokalitu môžeme zaradiť aj pod priestorovú lokalitu. Priestorová lokalita je zovšeobecnením prístupov do pamäte na adresy, ktoré sú blízko seba. Všeobecne to môžeme zapísať nasledovne, ak proces pristúpi v čase t na adresu s číslom n, tak v čase $t+1$ bude proces pristupovať na niektorú z adries v nejakom jej malom okolí k, na adresy v rozmedzí od $n-k$ až po adresu $n+k$.

Okrem príkladov, ktoré patrili pod sekvenčnú lokalitu, je možné spomenúť viacero príkladov všeobecnejšej priestorovej lokality. Podobne ako pri sekvenčnej lokalite sa môže priestorová lokalita vyskytovať pri využívaní rôznych dátových štruktúr. Zvyčajne sú dátové štruktúry implementované tak, že prvky, ktoré obsahujú, ukladajú do pamäte na miesta blízko pri sebe. Prechádzanie cez rôzne bežné dátové štruktúry, ako sú jednorozmerné alebo aj viacrozmerné polia, haldy, hašovacie tabuľky a mnohé ďalšie, vytvára priestorovú lokalitu. Priestorová lokalita je celkom prirodzená, pretože pri riešení problémov často problém rozdelíme na menšie podproblémy a informácie potrebné k riešeniu podproblému, ukladáme na miesta blízko seba.

1.6.3 Časová lokalita

Priestorová a aj sekvenčná lokalita do veľkej miery súviseli s prístupmi na adresy, ktoré boli blízko seba. Pri časovej lokalite nie je dôležité, na aké miesta bolo pristupované z hľadiska vzdialenosti jednotlivých adries. Časová lokalita poukazuje na vlastnosť programov pristupovať na adresy v nejakom čase viackrát. Môžeme ju definovať takto: ak proces pristúpi na adresu s číslom n v čase t, potom následne pristúpi k tejto istej adrese n za krátky čas s, teda v čase $t+s$.

Typickým príkladom na časovú lokalitu je iterovanie vo *for* cykle. Pri vykonávaní *for* cyklu sa postupne upravuje (najčastejšie inkrementuje alebo dekrementuje) iteračná premenná. Vždy po vykonaní tela cyklu sa pristupuje na adresu, na ktorej je táto iteračná premenná uložená. Takisto pred ďalším vykonávaním cyklu sa kontroluje podmienka s iteračnou premennou, ktorá rozhoduje o ďalšom vykonávaní tela cyklu. Časovú lokalitu je možné pozorovať aj pri vykonávaní rekurzívnych funkcií alebo volaní tej istej funkcie viackrát za sebou. Procesor po každom volaní bude pristupovať do pamäte na miesta, kde sú uložené inštrukcie danej funkcie.

1.7 Nástroje na záznam adries

Cieľom práce je analýza programov z pohľadu ich prístupov do pamäte. Programy a ani operačné systémy bežne neumožňujú sledovať na aké adresy procesy pristupujú. Z tohto dôvodu existuje viacero programov alebo nástrojov, ktoré takéto niečo dokážu zabezpečiť. V našej práci sme preto niektoré z nich odskúšali a preskúmali ich možnosti.

1.7.1 Pin Tool

Nástroj *Pin* je určený na dynamickú analýzu programov. Bol vytvorený spoločnosťou Intel venujúcej sa hlavne vývoju procesorov. Na nekomerčné účely sa môže využívať bez poplatkov. Základom tohto nástroja je pridávanie kódu do procesov, ktorý následne sleduje a zaznamenáva ich vykonávanie. *Pin* nemusí poznať zdrojový kód analyzovaného procesu, informácie o procese získava z jeho vykonávania. Funguje, ako just-in-time kompilátor, čo je typ kompilátora, ktorý postupne kompiluje program počas jeho vykonávania. V praxi je využívaný najmä na skúmanie vírusov a optimalizáciu programov [10].

Dôležitou vlastnosťou tohto nástroja je, že je transparentný voči vykonávaniu väčšiny procesov. To znamená, že pri ich vykonávaní sa celý proces správa rovnako, akoby sa správal mimo analyzovania s *Pin*. Existujú programy, ktoré dokážu rozpoznať dynamickú analýzu takýmito nástrojmi. Ide väčšinou o programy - vírusy, ktoré majú za cieľ vyhnúť sa takejto dynamickej analýze a následne zamaskovať škodlivú činnosť [13]. V našej práci by sme sa s týmto problémom nemali stretnúť, keďže zaznamenávame prístupy do pamäte bežných Unixových príkazov a niekoľko ukážkových programov, pri ktorých nie je očakávané, že by sa mali vykonávať inak, ako pri bežnom spustení bez použitia nástroja.

Zaujímavá je aj rýchlosť vykonávania týchto programov počas analyzovania, a to v porovnaní oproti bežnému vykonávaniu. V experimentoch v práci popisujúcej nástroj *Pin* [10], použili, ako príklad počítanie základných blokov (basic block). Základný blok je časť programu, v ktorej nedochádza k vetveniu napríklad pomocou podmienky *if* alebo skoku [8]. Pri použití viacerých optimalizácií, priemerné spomalenie otestované na viacerých programoch bolo približne dvojnásobné. V našom prípade je skúmanie prístupov do pamäte časovo náročnejšie na vykonanie, keďže pri každom jednom prístupe do pamäte, musí byť do súboru zapísaná adresa, na ktorú bolo pristupované a vstupno - výstupné operácie sú pomalé. Pri spustení programov bez záznamu adries a so záznamom pristupovaných adries bol rozdiel na niektorých nami testovaných jednoduchých programoch až tisícnásobný.

Rôzne nástroje na analýzu programov je možné vytvárať pomocou API z *Pin* nástroja. Pomocou tohto programu API, je programátorovi umožnené pridať vlastný kód do rôznych častí programu, napríklad za každú inštrukciu alebo vybrané inštrukcie podľa nejakej vlastnosti. Takisto umožňuje získať informácie z práve vykonávaného programu, akými sú napríklad obsahy registrov, alebo pre nás zaujímavé adresy, na ktoré bolo pristupované a použiť ich vo vlastnom kóde pridanom do programu.

Pri inštalovaní nástroja *Pin*, sú k nemu priložené základné ukážkové nástroje, vytvorené s *Pin* API určené na analýzu programov. Tieto nástroje sú celkom rôznorodé, napríklad je priložený nástroj, ktorým sa dá spočítať počet vykonaných inštrukcií alebo

nástroj na sledovanie volaní funkcií **malloc** a **free**. V príkladoch sa nachádzajú nástroje vhodné aj pre našu prácu: *pinatrace.cpp* a *itrace.cpp*. Nástroj *pinatrace* vypisuje do súboru všetky adresy, z ktorých proces čítal alebo do nich zapisoval. Vo výpise je uvedené, či bol urobený zápis alebo proces čítal a takisto pointer na inštrukciu, ktorá zápis alebo čítanie vykonala.

Ukážka výstupu 1.1: Ukážka výstupu nástroja *pinatrace*

```
...
0x7f66a9673e43: W 0x7ffe76053708
0x7f66a9673e58: W 0x7f66a96a5c00
0x7f66a9673e5f: R 0x7f66a96a6e70
...
```

Druhý nástroj - *itrace* zaznamenáva a vypisuje, taktiež do súboru, pointer na vykonávanú inštrukciu.

Na spustenie nástrojov je najprv potrebné nástroje skompilovať cez program Make s priloženým Makefile k nástroju *Pin*. Následne, v našom prípade, teda na Linuxe na x86, 64 bitovej architektúre sa obidva nástroje spúšťajú v priečinku, v ktorom sú súbory *pinatrace.cpp* a *itrace.cpp*, ako príkaz v shelli:

Príkaz 1.2: Príkaz na spustenie nástroja *pinatrace* na príkaze **ls**

```
$ ../../../pin -t obj-intel64/pinatrace.so -- ls
```

V tomto prípade je spustený nástroj *pinatrace* na programe **ls**.

1.7.2 Dynamorio

Nástroj *Dynamorio* je veľmi podobný nástroju *Pin*, taktiež je určený na dynamickú analýzu programov. Na rozdiel od nástroja *Pin*, sa tento nástroj vyvíja ako open source softvér. Podobne, ako pri nástroji *Pin*, nástroje na analýzu sa dajú vytvárať pomocou *Dynamoria* cez API. Pri porovnaní rýchlosti *Dynamoria* a nástroja *Pin* (v práci [10] popisujúcej nástroj *Pin* samotnými tvorcami Pinu) autori uznávajú, že nástroj *Dynamorio*, je v niektorých prípadoch o niečo rýchlejší, keďže jedným z hlavných cieľov *Dynamoria* je optimalizácia. Pri experimente s počítaním základných blokov však nástroj *Pin* prekonal v rýchlosti *Dynamorio*.

Dynamorio taktiež ponúka predpripravené nástroje na analýzu programov, ktoré sú pomenované ako "klient". Jedným z klientov je aj *memtrace_simple.c*, ktorý vytvára záznam pristupovaných adries. Pri porovnaní implementácie záznamov adries za pomoci *Pinu* a *Dynamoria*, je vidno výrazne komplikovanejší program s nástrojom *Dynamorio*. V prípade Linuxu na x86, 64 bitovej architektúre, na ktorom sme aj odskúšali nástroj *Pin*, pri spúšťaní niektorých klientov s nástrojom *Dynamorio* nastal problém.

Jedným z klientov bol aj klient určený na záznam pristupovaných adries. Aj napriek postupu v súlade s dokumentáciou pri spúšťaní klientov, sa pokus vždy skončil s chybou: *Floating point exception (core dumped)*. Rovnaký výsledok bol zaznamenaný pri viacerých starších verziách *Dynamoria*, a teda na našej verzii Linuxu na x86 64 bitovej architektúre, na ktorom sme pracovali, sa niektoré nástroje nepodarilo úspešne spustiť.

Jednou z výhod *Dynamoria* je funkčnosť tohto nástroja na viacerých architektúrach. Zatiaľ čo nástroj *Pin* funguje iba na x86 architektúrach, nástroj *Dynamorio* je implementovaný aj pre architektúru ARM. Keďže máme k dispozícii počítač Raspberry Pi 4B s 32 bitovou ARM architektúrou a bol na ňom spustený Linux, skúsili sme na ňom otestovať klienta na záznam pristupovaných adries. V tomto prípade fungoval klient bez chýb.

Ukážka výstupu 1.3: Ukážka výstupu klienta *memtrace_simple*

```
...
0xb6e01cf8:   4, r
0xb6e0134c:   4, stmdb
0xbe9b554c:  36, w
...
```

Klient *memtrace_simple* vytvára záznam inštrukcií, ktoré čítali alebo zapisovali do pamäte a na riadkoch za nimi sú uvedené adresy, na ktoré tieto inštrukcie zapisovali alebo z nich čítali. Formát výstupu je nasledovný: adresa, na ktorú bolo pristúpené, veľkosť dát na adrese a nakoniec názov inštrukcie alebo zápis/čítanie. Príkaz na vytvorenie záznamu adries je taktiež veľmi podobný, ako pri nástroja *Pin*.

Príkaz 1.4: Príkaz na spustenie nástroja *memtrace_simple.c* na príkaze **ls**

```
$ bin32/drrun -c samples/bin32/memtrace_simple.so -- ls
```

1.8 Existujúce práce

Jedným z prvých problémov, ktorý s použitím operačných pamätí prišiel, bol výkon multiprogramových systémov. Multiprogramové systémy využívajúce virtuálnu pamäť, ktorá je používaná dodnes, mali problém s výpadkom stránok. Výskumom tohto problému sa zaoberal v rôznych prácach Denning a kolektív [6, 15]. Začali sa zaoberať princípom lokality prístupu do pamäte, ktorý umožnil výrazné zrýchlenie virtuálnej pamäte. Väčšina výsledkov týchto prác je však viac ako päťdesiat rokov stará. Od tej doby sa výrazne pokročilo vo viacerých oblastiach. Veľké zmeny vidno napríklad v dnešnom hardvéri. Súčasné pamäte RAM a procesory sú mnohonásobne rýchlejšie ako tie z doby, keď boli písané Denningove práce. Programovacie jazyky, kompilátory

alebo interpretery prešli tiež rôznymi úpravami. Naša práca sa bude zaoberať aktuálnymi systémami a programami, aby sme overili, či výsledky získané v minulosti stále platia alebo ich doplnili dnešnými dátami.

Denning a kolektív v práci [15] skúmajú princípy lokality s reprezentáciou prístupov do pamäte, a to najmä ako postupnosť za sebou idúcich adries. V našej práci navrhujeme reprezentovať prístupy do pamäte ako orientovaný graf. S takýmto iným postupom môžeme odhaliť vzory v pamäťových prístupoch, ktoré by pri reprezentácii používanej Denningom a kolektívom bolo náročné nájsť.

Okrem využívania postupnosti adries, na ktoré procesy pristupujú na zrýchlenie a zefektívnenie práce počítača, sa analýza týchto postupností používa aj v iných oblastiach. Zo záznamu pristupovaných pamäťových miest je možné napríklad odhadnúť správanie procesu. V práci od autorov Xu a kolektív [18] skúmali možnosti rozpoznávania malvéru. Na rozdiel od bežne používaných analýz malvéru cez kód programu alebo inštrukcií, skúmali programy podľa adries, na ktoré počas behu programu pristupovali. Výsledkom práce je automatický nástroj na odhaľovanie programov útočiacich na zraniteľnosti pri prístupoch do pamäte.

Problematickou oblasťou sa počas ich výskumu stala veľkosť záznamov adries, v ktorých sa pri bežnom programe zaznamenáva milióny až miliardy prístupov. Aby umožnili efektívne rozpoznávanie malvéru, záznamy prístupov procesov do pamäte redukovali na menšie záznamy - segmenty za sebou idúcich adries, ktoré obsahovali iba výber častí, ktoré boli väčšinou problematické v iných škodlivých programoch. V našej práci sa však nezameriavame priamo na zraniteľnosť pamäte, ale skúmame vzory vo všetkých prístupoch na adresy v pamäti.

Kapitola 2

Návrh riešenia

V tejto kapitole popíšeme výber analyzovaných procesov a spôsob ich analýzy.

2.1 Výber analyzovaných procesov

Na začiatku celej práce sme potrebovali vybrať programy, ktoré budeme analyzovať. Jedným z príkladov prác, ktoré sme už v skorších častiach práce spomínali, bol text venujúci sa nástroju *Pin* [10]. V tejto práci taktiež potrebovali vybrať programy, na ktorých bolo možné otestovať rýchlosť ich nástroja. Na testovanie použili najmä rôzne benchmarky, teda programy špeciálne určené na testovanie výkonnosti. My sme sa v našej práci nezameriavali na testovanie výkonu, ale na analýzu vlastností programov. Benchmarkové programy väčšinou majú mierne odlišné správanie ako bežné programy a preto by takéto programy mohli naše výsledky skresliť.

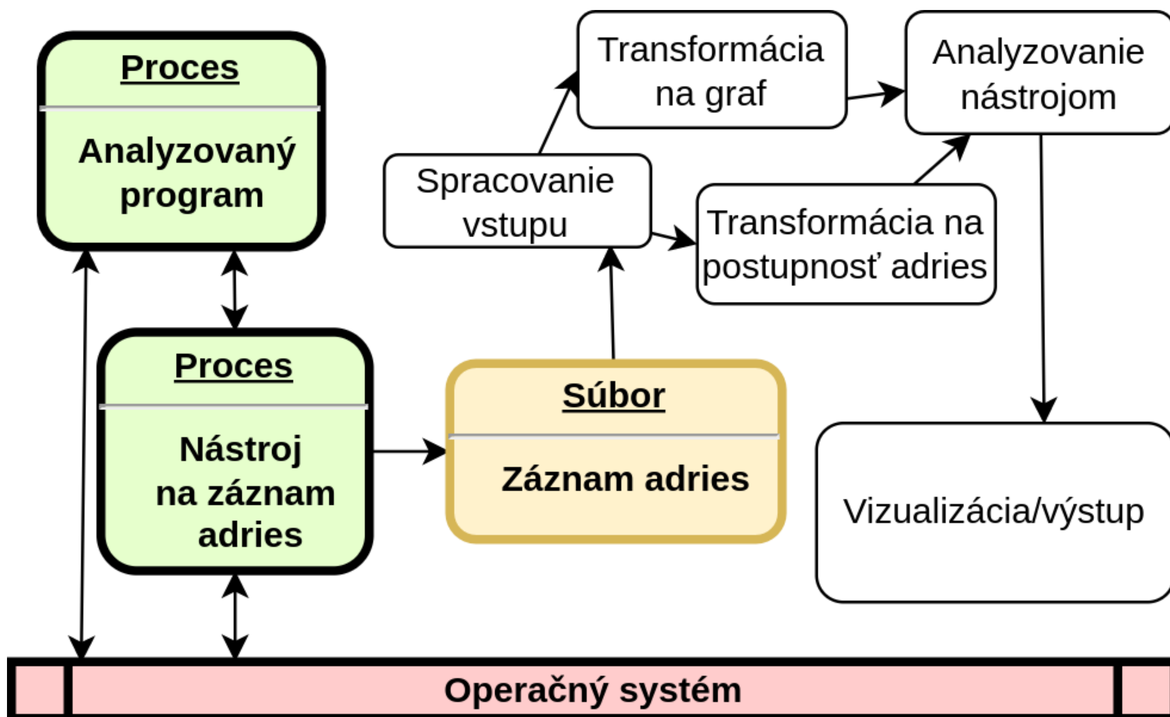
Tým, že sme robili záznam všetkých adries, na ktoré bolo pristupované, museli sme pri výbere procesov zobrať do úvahy problémy, ktoré to prináša. Prvým z problémov je, že takýto záznam trvá dlho, či už pomocou nástroja *Pin* alebo *Dynamorio*. Druhý s tým súvisiaci problém je veľkosť výstupných súborov vytvorených týmito nástrojmi. Príkladom je program v *Pythone*, ktorý iba vypíše "Hello world!". Záznam pristupovaných adries tohto programu má približne 300 megabajtov s približne 13 miliónmi zaznamenaných adries a vytváral sa pomocou nástroja *Pin* niekoľko minút. Aj napriek jednoduchosti a krátkosti programu je veľkosť súboru aj dĺžka trvania záznamu veľmi dlhá. Analýza by pri programoch s dlhším vykonávaním mohla byť problematická a spracovávanie viacerých dlhších programov by bolo náročné na veľkosť úložného priestoru a výpočtovú silu.

Berúc preto do úvahy vyššie uvedené, sme na analýzu vybrali a zaznamenali prístupy do pamäte najmä pre bežné Unixové príkazy. Ich správanie je typické bežným programom a tie, ktoré sme vybrali bolo možné zaznamenať a zanalyzovať vďaka ich krátkemu vykonávaniu. Sú tiež celkom reprezentatívnou vzorkou programov, keďže sú

tieto príkazy používané pomerne často pri obsluhovaní Linuxového operačného systému. Ďalšou výhodou týchto príkazov je, že sú open source, teda v prípade potreby je možné analyzovať až priamo zdrojový kód týchto príkazov. Keďže sú tieto príkazy používané obrovským množstvom používateľov, môžeme očakávať správnu a efektívnu implementáciu týchto príkazov.

Z vybraných príkazov spomenieme zopár príkladov. Použili sme príkaz **ls** s rôznymi prepínačmi. Tento príkaz sme následne spustili s dynamickou analýzou na viacerých priečinkoch bez a s použitím prepínača -l. Ďalej sme analyzovali ďalšie príkazy pracujúce s adresárovou štruktúrou, akými sú príkazy **rm**, **mkdir** alebo **touch**. Použili sme aj príkazy na rôzne výpisy akými sú príkazy **echo**, **less** a **cat**. Dobrou vlastnosťou týchto príkazov je možnosť spustiť ich na rôznych vstupoch veľmi jednoducho a tým vytvoriť väčší počet záznamov.

2.2 Architektúra riešenia



Obr. 2.1: Architektúra riešenia.

Obrázok 2.1 znázorňuje pohľad na architektúru nášho riešenia. Na začiatku celej analýzy spustí nástroj na záznam adries (napríklad *Pin Tool*) program, ktorý ideme analyzovať (napríklad príkaz **echo Hello**). Nástroj na záznam adries vykonáva na spustenom programe dynamickú analýzu a zaznamenáva pristupované adresy do súboru. Vstup zo súboru zaznamenaných adries je následne spracovaný a transformovaný na graf alebo

postupnosť pristupovaných adries. Postupnosť pristupovaných adries alebo graf sa ďalej analyzuje niektorými z postupov, ktoré popisujeme ďalej pri popise nástrojov. Nakoniec dáta získané analyzovaním za pomoci nástroja vizualizujeme. Nástroje je možné použiť s adresami z dátovej pamäte alebo aj z inštrukčnej pamäte.

2.3 Reprezentácia postupnosti adries

Dôležitou časťou našej práce je reprezentácia postupnosti pristupovaných adries. Najjednoduchšou možnosťou reprezentácie je reprezentovať postupnosť pristupovaných adries ako zoznam za sebou idúcich adries.

V našej práci budeme pracovať aj s inou reprezentáciou postupnosti pristupovaných adries. Našou novou myšlienkou v práci je reprezentovať pristupované adresy v programe ako orientovaný graf. Vrcholy takéhoto grafu G, označme ich množinou V, reprezentujú jednotlivé adresy, na ktoré bolo pristúpené. Orientované hrany H grafu G označujú prechod z jednej adresy na druhú. Presnejšie, orientovaná hrana (a_1, a_2), smerujúca z vrcholu a_1 do a_2 patrí do H práve vtedy, keď proces najskôr pristúpil na adresu a_1 a následne hneď za ňou na adresu a_2. Navyše, aby sa nestratili niektoré za sebou idúce prístupy do pamäte zopakované viackrát počas vykonávania programu, sú tieto hrany ohodnotené počtom takýchto prechodov, začínajúc váhou 1.

Táto grafová reprezentácia je vhodná najmä pri adresách pristupovaných do inštrukčnej pamäte. Mala by znázorňovať štruktúru vykonávaného programu. Napríklad, cyklus v programe by sa mohol preniesť do cyklu v grafe.

Vstupom pre postupnosť pristupovaných adries je vstupný súbor, napríklad vytvorený nástrojom *Pin*, výstupom je zoznam pristupovaných adries. Pre grafovú reprezentáciu je vstupom postupnosť adries, na ktoré proces pristupoval a výstupom je grafová štruktúra.

2.4 Nástroje pracujúce na zozname adries

V tejto časti popíšeme rôzne nami vytvorené nástroje, ktoré analyzujú prístupy procesov do pamäte.

2.4.1 Princípy lokality

Vytvorili sme nástroje na analyzovanie sekvenčného a priestorového vzoru v prístupoch procesov do pamäte. Nástroje prechádzajú postupnosťou pristupovaných adries v poradí v akom bolo na ne pristupované. Pri prechádzaní adresami sa kontrolujú podmienky prístupových vzorov, teda pri sekvenčnom, či je predošlá adresa o jedna menšia a pri priestorovom, či patrí do okolia adries predošlej adresy.

Špeciálne navrhnutým parametrom je parameter "preskoku". Určuje možnosť jedného porušenia princípu lokality v podpostupnosti. Napríklad, pri sekvenčnom prístupe, ak by sme mali nastavený parameter "preskoku" na True, tak aj postupnosť pristúpených adries: 5, 6, 7, 10, 8 by bola akceptovaná a pridaná do výstupu, aj keď očividne prístup na adresu 10 porušuje princíp sekvenčnej lokality.

Ďalší princíp lokality, ktorému sme sa venovali, je časová lokalita. Pri časovej lokalite je potrebné spočítať a zaznamenať adresy, na ktoré bolo pristúpené viackrát.

Vstupom pre nástroje hľadajúce sekvenčný a priestorový vzor je postupnosť pristupovaných adries a parametre na určenie vlastností princípu lokality, ktorý hľadáme. Pri sekvenčnom vzore ide o minimálnu dĺžku podpostupnosti, ktorú hľadáme a parameter preskoku. Pri priestorovom vzore je navyše pridaná veľkosť okolia, do ktorého môžu adresy patriť. Výstupom sú podpostupnosti, ktoré nástroj našiel. Pri časovej lokalite je situácia mierne odlišná, jediným vstupom je postupnosť pristupovaných adries a výstupom sú poradia prístupov na každú pristupovanú adresu.

2.4.2 Minimálny počet potrebných adries

Ďalší nástroj je určený na odhad minimálneho počtu adries potrebných na korektné vykonanie programu. Tento nástroj je inšpirovaný postupom z teórie kompilátorov a hlavne témou riešenia problému alokácie registrov.

Alokácia registrov je optimalizačný problém, riešiaci priradenie registrov procesora pre premenné. Registrov na procesore je obmedzený počet, na dnešných architektúrach môžeme nájsť okolo desiatky registrov. Na druhej strane, premenných, ktorým sa registre prideľujú, môže byť vo väčšine programovacích jazykoch neobmedzený počet a ich počet závisí od programu. Cieľom alokácie registrov je najefektívnejšie vykonanie programu, teda minimalizovanie používania premenných z operačnej pamäte [4].

Jedným z najznámejších riešení problému alokácie registrov je alokácia registrov pomocou farbenia grafu. S implementáciou tejto myšlienkou prišli v práci z roku 1980 Chaitin a kolektív [4]. Alokáciu registrov pretransformovali na riešenie problému farbenia grafu. Vrcholy v grafe určujú práve používané premenné a hrana sa pridáva medzi vrcholy vtedy, ak spolu nemôžu zdieľať jeden register (mohli by sme napríklad stratiť dáta z premennej, ktoré ešte budeme potrebovať). Nakoniec sa hľadá ofarbenie s minimálnym počtom farieb. Každá farba reprezentuje 1 register, teda pridelenie farby znázorňuje pridelenie registra premennej. Ak počet farieb prevýši počet registrov, potom nemôžu byť všetky premenné uložené v týchto registroch.

V našej práci sme podobne, ako vo výskume od Chaitina a kolektívu [4] chceli transformovať problém minimálneho počtu potrebných adries na korektné vykonanie programu na farbenie grafu. V tomto prípade by vrcholy reprezentovali adresy a hrana medzi vrcholmi by sa nachádzala vtedy, ak adresy reprezentované danými vrcholmi

nevieme nahradiť jednou adresou. Problém farbenia grafu je však NP - úplný a počty adries, na ktoré proces pristúpi počas vykonávania, sú v miliónoch. Znamená to teda, že graf, ktorý by sa ofarboval, by mal milióny vrcholov, čo je takmer neriešiteľné v normálnom čase.

Inšpirovali sme sa teda iným algoritmom riešiacim problém alokácie registrov. Tento prístup navrhol Pollet a kolektív a nazýva sa Linear scan [12]. Príslušný algoritmus je rýchlejší a aj jednoduchší na implementáciu ako algoritmus navrhnutý Chaitinom a kolektívom, na druhej strane, výstupom je o niečo menej kvalitná alokácia registrov. Princíp algoritmu je nasledovný: premenným sa pridelia intervaly, na ktorých je premenná aktívna. Intervaly sa zoradia a algoritmus postupne prejde cez všetky intervaly a v každom bode kontroluje, či začal nový interval. Ak áno a k dispozícii je voľný register, tak sa premennej, ktorej zodpovedá daný interval, priradí tento register. Ak nejaký interval končí, potom sa môže register, ktorý bol priradený pri začatí daného intervalu, vrátiť medzi voľné.

V našom prípade pracujeme namiesto registrov s adresami. Adresu považujeme za aktívnu od prvého prístupu k nej až po posledný prístup. Ďalej postupujeme rovnako, ako v algoritme Linear scan.

Vstup pre tento nástroj je postupnosť pristupovaných adries a výstupom je číselný odhad počtu minimálne potrebných adries na korektné vykonanie.

2.4.3 Rozdelenie adries podľa typu

Tento nástroj rozdelí všetky pristúpené adresy na adresy, ktoré boli iba čítané, iba zapisované alebo čítané pred zápisom. Toto je možné spraviť prechodom cez všetky pristúpené adresy a pre každú adresu si počas prechodu značiť postupnosť operácií, ktoré boli vykonané (zápis, čítanie). Nakoniec je podľa týchto postupností možné rozdeliť adresy na spomínané skupiny.

Tento nástroj na vstupe prijme postupnosť pristúpených adries s operáciou, ktorá bola na danej adrese vykonaná a výstupom sú adresy rozdelené na typy (iba čítané, iba zapisované, čítané pred zápisom).

2.5 Nástroje pracujúce s grafom adries

V tejto časti popíšeme nástroje, ktoré sme vytvorili pomocou analýzy nami navrhnutého grafu adries.

2.5.1 Silno súvislé komponenty

Tento nástroj je určený na analýzu štruktúry grafu adries. Pomocou tohto nástroja vyhľadávame v grafe pristupovaných adries silno súvislé komponenty.

Silno súvislý komponent je podgraf orientovaného grafu, ktorý je maximálny taký, že medzi každou dvojicou vrcholov z tohto podgrafu existuje cesta. Rozdelenie grafu na všetky silno súvislé komponenty, tvorí rozklad množiny vrcholov grafu [17].

Vstup v tomto nástroji tvorí graf pristupovaných adries a výstup tvoria vrcholy z grafu pristupovaných adries rozdelené do zoznamov na základe ich príslušnosti do silno súvislých komponentov.

2.5.2 Kompresia grafu

Nástroj, ktorý komprimuje graf pristupovaných adries, vychádza z teórie kompilátorov. Cieľom tohto nástroja je zmenšiť celkovú veľkosť grafu, a to zmenšením počtu vrcholov a hrán v tomto grafe tak, aby sa zachovala základná štruktúra tohto grafu. Podobne zaujímavou vlastnosťou môže byť aj tá skutočnosť, o koľko sa podarilo graf pristupovaných adries zmenšiť.

Hlavná myšlienka celého algoritmu na zjednodušenie grafu je inšpirovaná základnými blokmi (basic block) a control-flow grafmi. Control-flow graf je graf, ktorý popisuje možné prechody programom. V control-flow grafe sú vrcholmi základné bloky a hrany označujú skoky z jedného základného bloku do druhého [1].

V našom nástroji na kompresiu grafu adries sme namiesto zdrojového kódu programu pracovali s pristupovanými adresami a z toho vytvoreného grafu. Za základný blok sme prenesene považovali maximálny súvislý podgraf, v zmysle počtu vrcholov, v ktorom každý vrchol má výstupný stupeň 1. Navyše sme pridali podmienku, aby váha hrán medzi týmito vrcholmi bola rovnaká.

Vstupom je graf, ktorý chceme skomprimovať. Výsledkom je skomprimovaný graf a taktiež je možné získať počet skomprimovaných vrcholov v grafe.

Kapitola 3

Implementácia riešenia

V tejto časti popíšeme implementáciu navrhnutého riešenia.

3.1 Programovací jazyk a knižnice

Hlavným programovacím jazykom v našej práci, pri implementácií experimentov so záznamom postupnosti pristupovaných adries, je *Python*. *Python* sme vybrali najmä vďaka dostupnosti mnohých užitočných knižníc a jednoduchej práci s textom a textovými súbormi, čo tvorilo značnú časť našej práce. Tento programovací jazyk je známy svojou jednoduchosťou používania, veľmi dobrou čitateľnosťou programu a keďže ide o interpretovaný jazyk, debugovanie je taktiež pomerne jednoduché.

Výber *Pythonu* má však aj niektoré svoje nevýhody spájajúce sa aj s tým, že ide o interpretovaný jazyk. Na niektoré jeho úskalia sme narazili aj pri našej práci. Problém bol s jeho rýchlosťou a spotrebou pamäte pri vykonávaní grafových algoritmov, používajúcich rekurziu. Tento nedostatok by bolo možné vyriešiť odstránením rekurzie z programu, ale mohlo by to zneprehľadniť program a zaviesť do neho chyby.

Z tohto dôvodu sme na riešenie daného problém využili *Pypy*, čo je alternatívna implementácia *Pythonu*. Je to just-in-time kompilátor, využívajúci rôzne optimalizácie, ktorý dokáže niekoľkonásobne rýchlejšie vykonávať väčšinu programov, ktoré majú dlhší priebeh, a to pri využití menšieho objemu pamäte ako klasický *Python*. V bežných programoch funguje bez potreby zmeny kódu a podporuje väčšinu známych knižníc [14].

Knižnica, ktorá v *PyPy* nie je podporovaná a ktorú sme v práci využívali na vizualizácie je Matplotlib. Niektoré výpočtové časti programu sme preto spúšťali osobitne od časti programu, ktorý používa Matplotlib - jednu cez *Pypy*, druhú cez *Python*. Pomocou knižnice Matplotlib sme v práci vytvárali grafy rôznych typov pre vizualizáciu nameraných dát.

Program sme odskúšali na Linuxe na x86 64 bitovej architektúre. Záznam pristu-

povaných adries sme získavali pomocou nástroja *Pin* na x86 64 bitovej architektúre a pomocou nástroja *Dynamorio* na ARM 32 bitovej architektúre.

3.2 Implementácia nástrojov

V tejto časti uvedieme, akým spôsobom sme jednotlivé nástroje implementovali.

3.2.1 Reprezentácia postupnosti adries

Na reprezentáciu zoznamu pristupovaných adries sme vytvorili viacero tried. Každá trieda zodpovedá jednému typu vstupného súboru, napríklad sme implementovali triedu zodpovednú za vstupný súbor z nástroja *pinatrace*. Všetky tieto triedy sú inicializované parametrom - cestou ku vstupnému súboru, ktorý je potom spracovaný a na základe neho sa nastavia atribúty danej inštancie. Základný atribút je pri všetkých triedach postupnosť pristúpených adries, ktorá je uložená v zozname. Pri niektorých triedach sú dodatočné atribúty, ako napríklad pri triede reprezentujúcej vstup z nástroja *pinatrace*, je ďalším atribútom zoznam všetkých vstupných riadkov s popisom adresy a operácie, ktorá s ňou bola vykonaná.

Pri navrhnutej grafovej reprezentácii sme adresy, teda vrcholy, namapovali na čísla od 1 po počet pristupovaných adries. Čísla sú adresám priraďované v poradí ich prvého prístupu na danú adresu (opakujúce sa adresy budú teda namapované na rovnaké číslo). Takéto namapovanie zjednodušilo implementáciu grafu a zároveň rieši problém s ASLR. Bez takéhoto mapovania, by kvôli ASLR mohli mať aj procesy, ktoré vykonali presne to isté, rôzne grafy. Po namapovaní vrcholov na čísla od 1 po počet vrcholov, môžu mať takéto procesy rovnaké grafy.

Graf inicializujeme ako triedu, ktorá má ako vstupný parameter niektorú z tried reprezentujúcich postupnosť pristupovaných adries. Na základe zadanej triedy potom vytvoríme inicializovaný graf. Graf implementujeme ako zoznam slovníkov s dĺžkou rovnou počtu adries, na ktoré bolo pristúpené. Znamená to, že index v tomto zozname určuje číslo namapované na niektorú adresu. Každý slovník na nejakom indexe v tomto zozname popisuje, ktoré vrcholy sú susedné s vrcholom s číslom rovným indexu, na ktorom sa tento slovník nachádza. Kľúče v týchto slovníkoch sú vrcholy, do ktorých smeruje hrana a hodnota určuje váhu tejto hrany. Zoznam slovníkov je časť triedy reprezentujúcej náš graf adries, v ktorej sú navyše ešte pridané 2 slovníky určené na mapovanie adresy na jej priradené číslo a naopak, ktoré potom následne využívame v našich nástrojoch na analýzu.

3.2.2 Princípy lokality

Nástroje určené na hľadanie sekvenčného vzoru a priestorového vzoru majú spoločný základ programu. Nástroje prechádzajú postupnosťou pristupovaných adries v poradí v akom bolo na ne pristupované. Pri prechádzaní adresami si ukladáme do premennej aj predošlú adresu, na ktorú bolo pristúpené. Potom nasleduje časť, v ktorej sa prístupy líšia pre sekvenčný a pre priestorový vzor. Kontrolujú sa podmienky, a to nasledovne: pre sekvenčný vzor sa skontroluje či je predošlá adresa o 1 menšia, pre priestorový vzor sa kontroluje či predošlá adresa patrí do požadovaného intervalu. V prípade porušenia skúmaného vzoru sa ešte skontroluje, či je parameter preskoku nastavený na True a či už bol použitý, podľa toho môže byť podpostupnosť predĺžená.

Pri priestorovom vzore je možné pomocou parametra nastaviť veľkosť tohto intervalu. Pre oba princípy lokality je možné nastaviť minimálnu dĺžku podpostupnosti spĺňajúcu princíp lokality, ktorá bude pridaná do výstupu. Parameter preskoku je nastaviteľný ako boolovská hodnota. Jedným z parametrov je pri všetkých implementovaných nástrojoch na hľadanie princípov lokality aj inštancia niektorej triedy zoznamu pristupovaných adries.

Nástroj pre časovú lokalitu sme implementovali pomerne jednoducho, a to prechodom cez všetky pristupované adresy a ukladaním poradí prístupov do slovníka s kľúčom rovným číslu adresy.

3.2.3 Minimálny počet potrebných adries

Aby sme odhadli minimálny počet potrebných adries na korektné vykonanie programu, každej adrese priradíme 1 interval, v ktorom je aktívna, ako dvojicu - začiatok intervalu a koniec intervalu. Následne sme postupovali ako v algoritme Linear scan. Potom prechádzame zoradené intervaly, ale na rozdiel od Linear scan algoritmu počas prechodu nepriraďujeme registre, ale iba počítame, koľko intervalov sa v každom momente prekrýva. Predpokladáme, že čím viac intervalov sa v jednom momente prekrýva, tým viac adries je potrebných na korektné vykonanie programu. Toto číslo, teda maximum prekrývajúcich sa intervalov vráti program ako výstup. Vstupom je pre funkciu jediný parameter, a to inštancia niektorej z tried reprezentujúcich zoznam pristupovaných adries.

3.2.4 Silno súvislé komponenty

Na vyhľadanie všetkých silno súvislých komponentov v grafe pristupovaných adries sme použili Tarjanov algoritmus [17]. Na riešenie problému sme ho v našom nástroji použili najmä vďaka jeho rýchlosti, keďže grafy na ktorých pracujeme sú pomerne veľké. Na rozdiel od niektorých iných algoritmov určených na vyhľadanie silno súvislých

komponentov, vyhľadá Tarjanov algoritmus silno súvislé komponenty počas jedného prechodu DFS s časovou zložitosťou $O(|V| + |E|)$, kde $|V|, |E|$ sú počty vrcholov a hrán zadaného grafu.

Z dôvodu veľkosti grafov, s ktorými sme pracovali, nastal aj s Tarjanovým algoritmom problém s využívaním veľkého množstva pamäte a pomalým výpočtom za použitia *Pythonu*. Predpokladáme, že najväčší problém bol spôsobený rekurziou využívanou počas výpočtu. Tento problém sme vyriešili použitím alternatívnej implementácie *Pythonu PyPy*.

Funkcia hľadajúca silno súvislé komponenty má na vstupe inštanciu niektorej z tried reprezentujúcich zoznam pristupovaných adries.

3.2.5 Kompresia grafu

Kompresiu grafu sme implementovali tak, že základné bloky v grafe pristupovaných adries sme nahradili jednou hranou s váhou, akú majú nahrádzané hrany. Základné bloky sme v grafe hľadali pomocou prechodu grafu algoritmom DFS. DFS spúšťame z prvej pristupovanej adresy. Počas prechodu grafom, algoritmus kontroluje výstupný stupeň vrchola, ktorým prechádza a kontroluje, či je váha hrany rovnaká ako váha zvyšných hrán, ktoré program dovtedy kontroloval. Ak sú podmienky splnené, potom ukladáme vrcholy do zoznamu, podľa ktorého sú na konci programu nahradené novou hranou.

Trieda komprimujúca graf má jediný inicializačný paramter - samotný graf pristupovaných adries a atribút s počtom vrcholov, ktoré boli skomprimované.

Kapitola 4

Overenie riešenia

V tejto kapitole uvedieme nami namerané dáta a objavené vzory správania procesu v postupnosti adries. Postupne popíšeme výsledky pre jednotlivé nástroje a taktiež spomenieme niektoré nami vytvorené vizualizácie výsledkov.

4.1 Silno súvislé komponenty

Nástroj určený na vyhľadávanie silno súvislých komponentov sme spustili na nami vybraných programoch popísaných v kapitole návrh riešenia. Nástroj sme spustili na adresách inštrukcií, ale aj na adresách, do ktorých proces zapisoval alebo z nich čítal.

Silno súvislé komponenty adries, z ktorých bolo čítané a do ktorých bolo zapisované, nemali veľmi významné výsledky na programoch, na ktorých sme tento nástroj vyskúšali. V každom grafe adries sa nachádzalo viacero silno súvislých komponentov s 1 vrcholom a následne 1 silno súvislý komponent obsahujúci všetky zvyšné vrcholy.

Tak ako sme predpokladali, zaujímavejšie výsledky sme dostali z adries inštrukcií. Náš predpoklad bol, že adresy inštrukcií majú určitú spojitosť so štruktúrou programu. V pristupovaných adresách procesov sme taktiež našli viacero komponentov súvislosti s 1 vrcholom a 1 komponent súvislosti s väčším počtom vrcholov. Okrem nich sa v každom nami otestovanom programe nachádzali silno súvislé komponenty s nasledovnými počtami vrcholov: 37, 31, 10.

Aby sme adresy v týchto silno súvislých komponentoch bližšie preskúmali, mierne sme upravili nástroj *itrace*. Do nástroja *itrace* sme na koniec záznamu pristupovaných adries dali skopírovať do osobitného súboru obsah súboru */proc/self/maps*. Súbor */proc/self/maps* je z pseudosúborového systému *proc*, ktorý popisuje rôzne vlastnosti procesov. Súbor *maps*, ktorý sme využili, popisuje jednotlivé regióny pamäte daného procesu. O každom regióne sa v súbore nachádzajú informácie ako sú napríklad počiatočná adresa a koncová adresa regiónu, prístupové práva potrebné na prístup k regiónu, v prípade, že je región mapovaný zo súboru, tak názov tohto súboru a ďalšie.

Zo súboru *maps* procesu, v ktorom sme hľadali silno súvislé komponenty, sme využili intervaly regiónov a súbory z ktorých sú mapované. Každú adresu zo silno súvislých komponentov s počtami vrcholov 37, 31, 10 sme pre každý nami skúmaný program porovnali s intervalmi v *maps* súbore. Všetky tieto adresy boli namapované zo súboru */usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2*. Podľa manuálových stránok ide o dynamický linker určený na načítavanie zdieľaných knižníc do operačnej pamäte. Tieto silno súvislé komponenty môžu vznikať vďaka štruktúre programu dynamického linkera. Napríklad silno súvislé komponenty môžu v programe zodpovedať cyklu alebo niektorej funkcii.

Aby sme potvrdili, že dané silno súvislé komponenty sa nachádzajú iba v dynamicky linkovaných programoch, skompilovali sme jednoduchý program pomocou *gcc* kompilátora s prepínačom *-static*. Tatko skompilovaný program je následne linkovaný staticky. Pre porovnanie sme tento jednoduchý program skompilovali aj bez prepínača *static*, vtedy je predvolené dynamické linkovanie. Po preskúmaní silno súvislých komponentov na oboch záznamoch inštrukčných adries na dynamicko linkovanom programe, aj na staticky linkovanom programe, sa potvrdilo naše očakávanie. V grafe záznamu pristupovaných adries programu, ktorý bol linkovaný dynamicky, sa nachádzali silno súvislé komponenty s veľkosťami 37, 31, 10, zatiaľ čo v prípade staticky linkovaného programu sa vo výstupe silno súvislé komponenty s týmito počtami vrcholov nenachádzali.

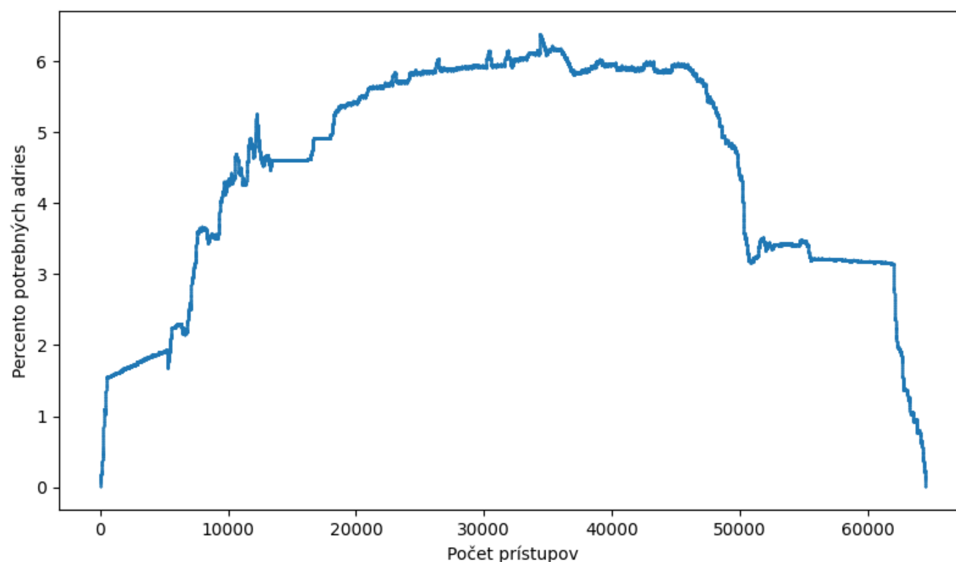
Náš objav sme následne overovali aj na počítači Raspberry Pi 4B s ARM architektúrou a taktiež s Linuxovým operačným systémom. V tomto prípade sa medzi silno súvislými komponentami nenachádzali komponenty s veľkosťami 37, 31, 10 ako to bolo na x86 Linuxovom systéme. Na rozdiel od toho, sa tu však v každom nami skúmanom programe nachádzali silno súvislé komponenty s počtami vrcholov 35, 28, 7. Podobne ako na x86 systéme sme skontrolovali z akých súborov boli dané adresy mapované. Všetky adresy z týchto silno súvislých komponentov patrili do regiónov namapovaných na súbor */usr/lib/arm-linux-gnueabihf/ld-2.28.so*, čo je taktiež dynamický linker. Pomocou týchto zistení, vieme iba na základe záznamu pristupovaných adries do inštrukčnej pamäte na oboch systémoch odhadnúť, či daný program bol linkovaný dynamicky alebo staticky.

Pre ďalšie používanie tohto nástroja navrhujeme nasledovný postup: najprv vytvoriť záznamy pristupovaných adries programov, ktoré nás zaujímajú a následne skontrolovať, či sa v každom zázname nenachádzajú rovnako veľké komponenty súvislosti. Nakoniec sa adresy v týchto komponentoch porovnajú s intervalmi v súbore */proc/self/maps* na rozpoznanie, čo dané komponenty súvislosti spôsobilo.

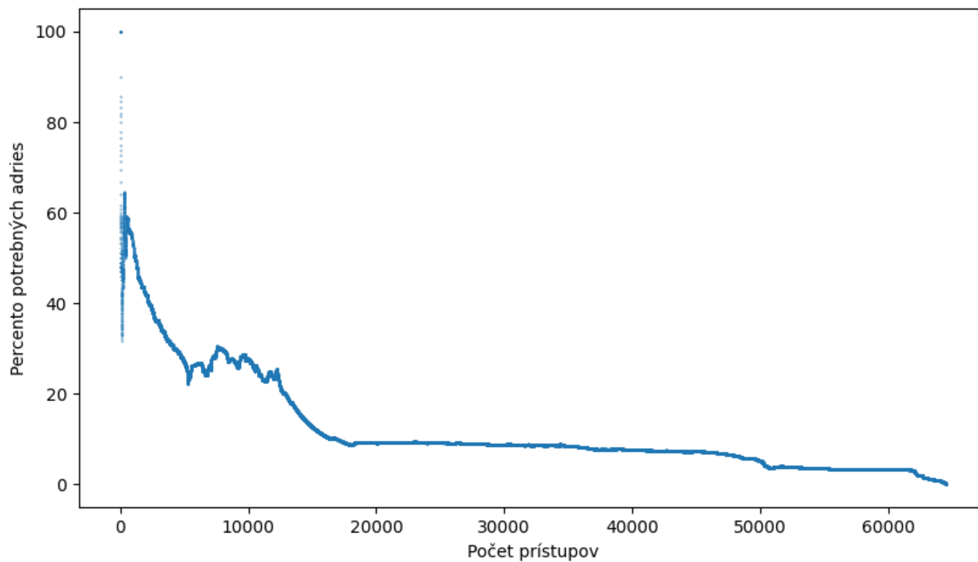
4.2 Minimálny počet potrebných adries

Odhad minimálneho počtu potrebných adries na korektné vykonanie programu sme na procesoch skúšali 2 spôsobmi. Prvý spôsob predstavoval výpočet potrebných adries z celkového počtu všetkých použitých adries. Na nami skúmaných procesoch to bolo podľa odhadu v priemere okolo 7%. Najviac adries 23%, bolo potrebných pri programe vypisujúcom "Hello world!" v *Pythone*, teda pri práci interprétera a najmenej 2% pri nami vytvorenom jednoduchom programe v *C++* na výpis prvočísel.

Druhý prístup, ktorý sme zvolili, bol priebežný výpočet minimálne potrebných adries na vykonanie programu prepočítaný po každom prístupe do pamäte. V takomto prístupe boli 2 možnosti ako tento priebežný výpočet počítať. Prvou možnosťou je počítať percentá v pomere k počtu všetkých pristupovaných adries 4.1, druhou možnosťou je počítať percentá v pomere k počtu dovtedy pristúpených adries 4.2.



Obr. 4.1: Minimálny počet adries v pomere k počtu všetkých pristúpených adries pre príkaz **echo Hello**. Počet prístupov by sme mohli v grafe nahradiť aj časom.



Obr. 4.2: Minimálny počet adries v pomere k počtu doteraz pristupovaných adries pre príkaz **echo Hello**.

Algoritmus 4.1: Ukážkový program v jazyku *C*

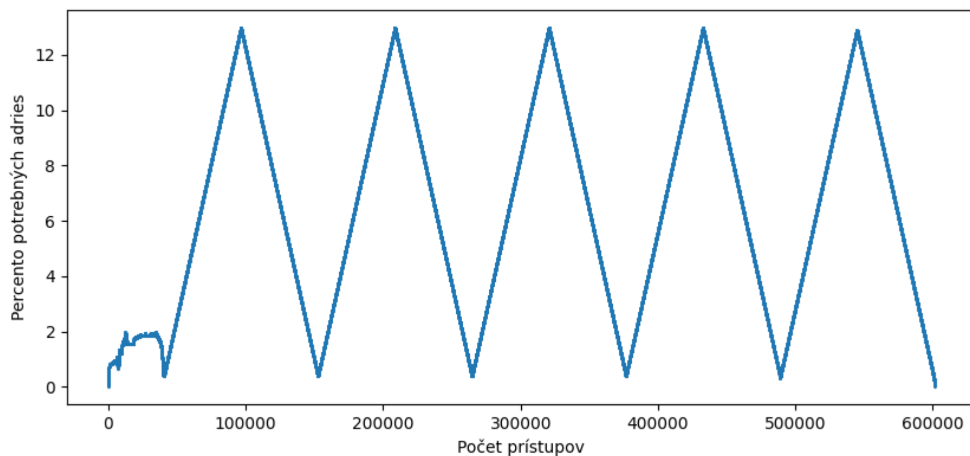
```
#include <stdlib.h>
int main() {
    for(int i = 0; i < 5; i++){
        int *a = malloc(4096 * sizeof(int));
        for(int k = 0; k < 2; k++){
            for(int j = 0; j < 4096; j++){
                a[j] = 12345;
                a[j]++;
                int b = a[j];
            }
        }
        free(a); \\ odstranene v druhej verzii programu
    }
    return 0;
}
```

K tomuto nástroju sme ešte vytvorili 1 ukážkový program 4.1 v jazyku *C* v 2 verziách. V oboch verziách program päťkrát naalokuje pole veľkosti 4096 integerov. Ďalej v tom istom cykle dvakrát s každou adresou vykoná jednoduché operácie (pričíta, priradí do inej premennej...) z toho dôvodu, aby bola adresa aspoň na nejakom intervale

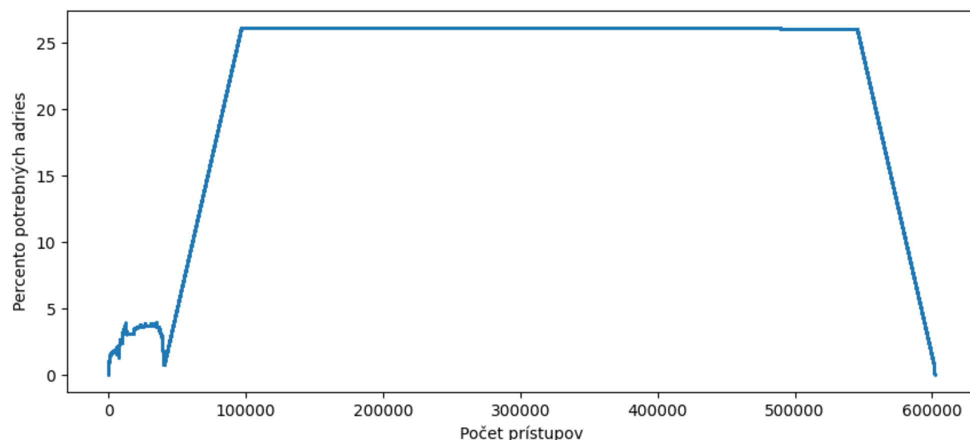
aktívna. Na konci sa tieto 2 verzie líšia, v jednej je každý cyklus ukončený volaním **free()** na začiatku cyklu alokovaného poľa, v druhej toto volanie chýba.

Z uvedených dvoch programov sme vytvorili graf minimálneho počtu potrebných adries. Vo verzii bez volania **free** je možné vidieť významné zmeny v grafe 4.3. Týchto zmien je tam presne päť a môžeme ich pripísať tomu, že bez volania **free** sú pri ďalšej alokácii pridelené iné, doteraz nepridelené adresy. V grafe pre program 4.4, v ktorom bolo volané volanie **free**, sa tieto zmeny v grafe nenachádzajú, teda alokáciou sú po dealokácii pridelené tie isté adresy.

Takýto graf nemusí značiť nekorektný program. Graf by mohol vyzerať podobne v prípade alokovania viacerých polí a ich dealokovanie všetkých naraz až na konci programu.



Obr. 4.3: Verzia ukážkového programu bez volania **free()**.

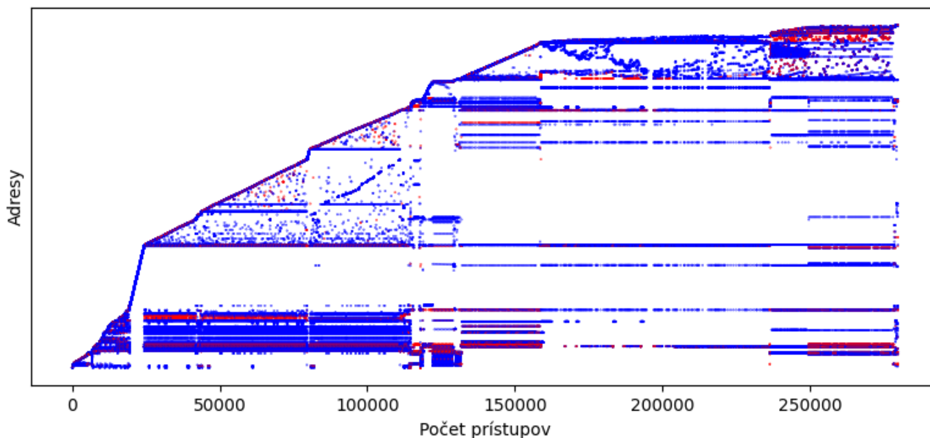


Obr. 4.4: Verzia ukážkového programu s volaním **free()**.

4.3 Princípy lokality

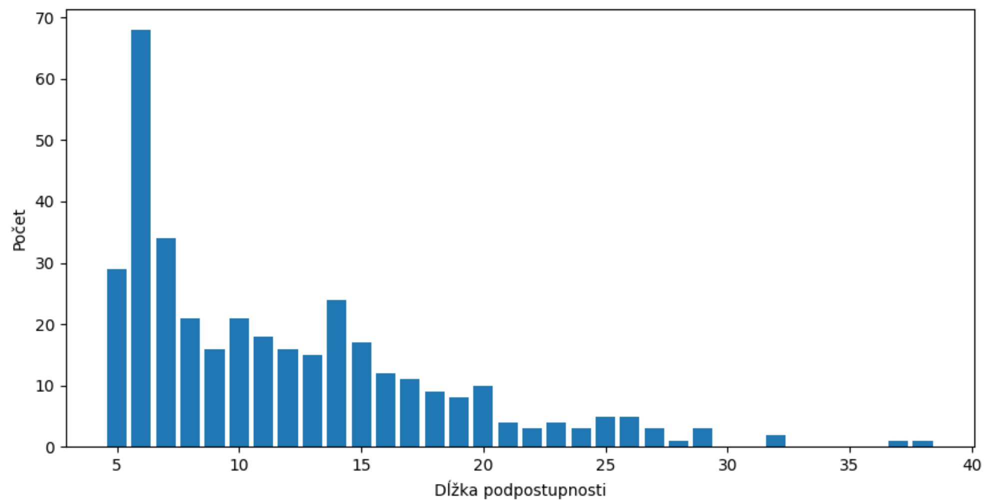
Pri nástrojoch analyzujúcich princípy lokality sme namerané dáta taktiež vizualizovali v grafe. Sekvenčný a priestorový vzor sme znázorňovali v klasickom 2D bodovom grafe. Na x-ovej osi sme značili, koľký prístup do pamäte zobrazujeme, na y-ovej osi, číslo danej adresy. Farbou bodu v grafe nakoniec určíme, či adresa patrí do nejakej podpostupnosti tvoriacej prístupový vzor. Adresy sa ale nedali na y-ovej osi rozmiestniť podľa ich číselnej hodnoty, keďže adries, na ktoré bolo pristúpené bol obrovský počet a vďaka ASLR boli medzi číselnými hodnotami adries veľmi veľké náhodné rozostupy.

Tento problém sme vyriešili tak, že adresy sa podľa ich čísel zoradia od najväčšej po najmenšiu a každej adrese sa týmto zoradením priradí jej poradové číslo. Hodnoty na y-ovej osi sú následne určené týmito poradovými číslami adries. Príklad vizualizácie priestorového vzoru v dátovej pamäti na príkaze **ls** je možné vidieť na obrázku 4.5.

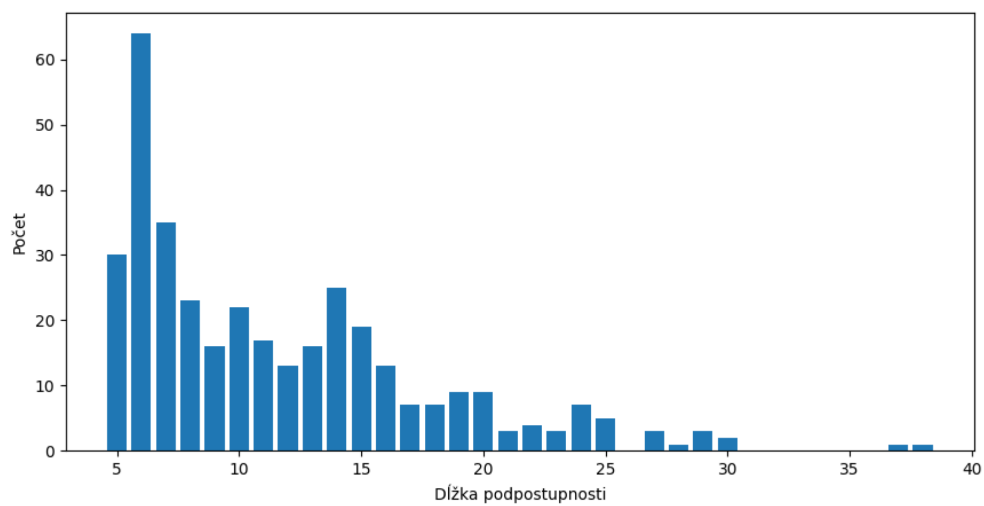


Obr. 4.5: Priestorový vzor s okolím 5 adries na príkaze **ls** s minimálnou dĺžkou podpostupnosti 5, červenou farbou sú vyznačené adresy patriace do niektorej podpostupnosti.

Zaujímavou vizualizáciou sú aj početnosti dĺžok podpostupností spĺňajúcich prístupový vzor. Špeciálne porovnanie grafov početností s použitím parametra preskoku a bez použitia parametra preskoku, môže určiť správnosť implementácie programu alebo kompilácie. Ak je medzi týmito početnosťami veľký rozdiel, t.j. po umožnení jednorázového porušenia prístupového vzoru vznikajú dlhšie podpostupnosti alebo ich je výrazne viac, potom môže byť niekde problém. Môže ísť napríklad o neefektívne prechádzanie štruktúrou, alebo program nie je kvalitne skompilovaný (nebola vhodne spravená alokácia registrov). Príklady stĺpcových grafov pre príkaz **mkdir** sa medzi sebou, s použitím parametra preskoku 4.6 a bez neho 4.7, veľmi nelíšia. V oboch prípadoch tvoria zobrazené dĺžky podpostupnosti približne 3% všetkých adries, na ktoré bolo pristúpené.



Obr. 4.6: Graf vizualizujúci počty podpostupností sekvenčného vzoru jednotlivých dĺžok spĺňajúce sekvenčný vzor na príkaze **mkdir** s možnosťou jednorazovo tento vzor porušiť.



Obr. 4.7: Graf vizualizujúci počty podpostupností sekvenčného vzoru jednotlivých dĺžok spĺňajúce sekvenčný vzor na príkaze **mkdir**.

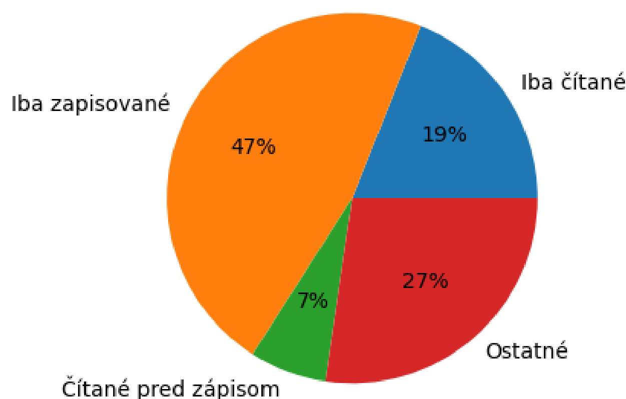
4.4 Ďalšie namerané dáta

4.4.1 Kompresia grafu

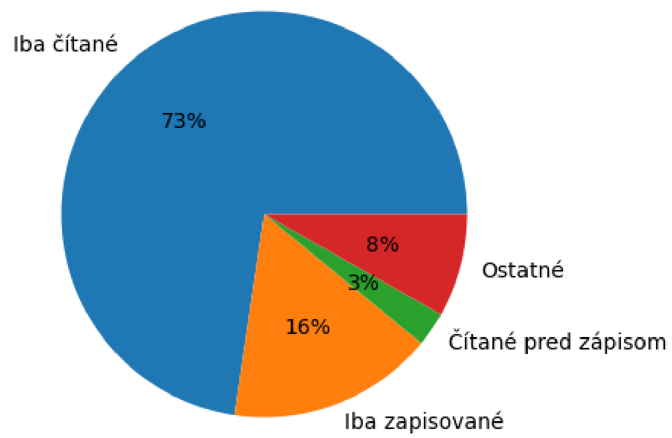
Pri nástroji na kompresiu grafu sme vyčíslili, o koľko percent je možné pomocou tohto nástroja graf zmenšiť. Zaujímavým výsledkom je, že pokiaľ sme nástroj spustili na pristupovaných adresách v dátovej pamäti, grafy sa na nami skúmaných procesoch dali skomprimovať v priemere o 25%, grafy vytvorené z pristupovaných adries do inštrukčnej pamäte sa dali v priemere skomprimovať až o 35%.

4.4.2 Rozdelenie typov adries

Adresy sme rozdelili podľa typu na adresy, na ktoré bolo iba zapisované, adresy, z ktorých bolo iba čítané a adresy, z ktorých bolo najskôr čítané a až potom do nich zapisované. Prekvapivo, vo všetkých nami skúmaných procesoch, takéto adresy tvorili spoločne väčšinu všetkých použitých adries. Uvedené správanie nemusí byť nekorektné, môže ísť napríklad o prácu so zdieľanou pamäťou alebo časti pamäte namapovanej zo súboru. Taktiež môžu niektoré systémové volania využívať časť adresného priestoru na vstup alebo výstup. Medzi rôznymi procesmi je možné vidieť veľké rozdiely v podieloch týchto typov adries ako je možné vidieť na príklade programu v *Pythone* 4.8 a príkazu **id** 4.9.



Obr. 4.8: Pristupované adresy z dátovej pamäte programu v *Pythone* vypisujúci "Hello world!" rozdelené podľa typu.



Obr. 4.9: Pristupované adresy z dátovej pamäte príkazu **id** rozdelené podľa typu.

Záver

Vyskúšali a porovnali sme nástroje určené na dynamickú analýzu programov. Na ich základe sme následne vytvorili viacero nástrojov určených na analyzovanie prístupov do pamäte a ich vplyvu na vlastnosti programov. Využili sme pri tom dva rôzne prístupy. Doteraz známy prístup využívajúci jednoduchú postupnosť pristupovaných adries a nami vytvorený prístup pracujúci s grafovou reprezentáciou pristupovaných adries.

Prvý a doteraz známy prístup pracujúci s postupnosťou pristupovaných adries sme použili na vytvorenie nástrojov určených na vyhľadanie a vizualizáciu prístupových vzorov zadefinovaných Denningom [6]. Navyše sme skonštruovali nový nástroj, ktorý odhaduje z postupnosti pristúpených adries minimálny počet potrebných adries na korektné vykonanie programu. Zaujímavé výsledky priniesol aj nástroj počítajúci pristupované adresy, adresy, ktoré boli iba čítané a adresy, ktoré boli iba zapisované a ďalšie.

Nami skonštruovaný nový prístup vyjadrujúci prístupy do pamäte pomocou grafovej reprezentácie sa osvedčil pri nástroji, ktorý analyzuje silno súvislé komponenty v tejto reprezentácii. Pomocou tohto nástroja sa nám podarilo zistiť, že niektoré silno súvislé komponenty sa nachádzajú v prístupoch do inštrukčnej pamäte v dynamicky linkovaných programoch. Toto zistenie sme potvrdili na dvoch systémoch s rôznou architektúrou. Ďalším nástrojom, ktorý sme vytvorili, je nástroj určený na kompresiu grafu pristupovaných adries. Inšpiráciou pri jeho zostrojovaní boli control-flow grafy a základné bloky.

Nástroje a výsledky dosiahnuté v práci je možné uplatniť vo viacerých oblastiach. Primárne sú nástroje cielené na analyzovanie vlastností a efektivity programov pri práci s pamäťou. Oblasťou, kde by mohli mať najmä nami vytvorené vizualizácie prístupov do pamäte prínos, je výučba operačných systémov. Umožňujú vizualizovať viaceré teoretické poznatky z témy operačných pamätí na skutočných dátach.

Našu prácu a vytvorené nástroje by bolo možné rozšíriť viacerými smermi. Zaujímavé by mohlo byť skombinovať niektoré naše nástroje, a tak vytvoriť špecializovaný nástroj určený na analyzovanie prístupov do pamäte. Mohlo by ísť napríklad o nástroj pracujúci so strojovým učením, ktoré by nástroje mohlo poprepájať, prípadne skombinovať. Ďalší výskum by sa mohol venovať aj grafovej reprezentácii postupnosti adries, ktorú sme v práci navrhli.

Literatúra

[1] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces.* CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2018.

[3] R. Bharadwaj. *Mastering Linux Kernel Development: A Kernel Developer's Reference Manual.* Packt Publishing, 2017.

[4] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[5] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[6] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, July 2005.

[7] Irv Englander. *The Architecture of Computer Hardware and System Software.* John Wiley and Sons, Inc., Hoboken, NJ, USA, 2007.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[9] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, 2013.

[10] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, jun 2005.

[11] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Embedded technology series. Elsevier Science, 2012.

[12] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, sep 1999.

[13] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. Measuring and defeating anti-instrumentation-equipped malware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 73–96, Cham, 2017. Springer International Publishing.

[14] Alexander Roghult. Benchmarking python interpreters: Measuring performance of cpython, cython, jython and pypy, 2016.

[15] Jeffrey R. Spirn and Peter J. Denning. Experiments with program locality. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, AFIPS '72 (Fall, part I), page 611–621, New York, NY, USA, 1972. Association for Computing Machinery.

[16] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall, 2010.

[17] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[18] Zhixing Xu, Aarti Gupta, and Sharad Malik. Trace-based analysis of memory corruption malware attacks. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing*, pages 67–82, Cham, 2017. Springer International Publishing.

Dodatok A

Prílohy

Program a ostatné nižšie spomínané súbory a priečinky sú zverejnené na stránke: `https://gitlab.com/matu9s/memory-address-traces-patterns` alebo aj na priloženom CD.

A.1 Požiadavky na spustenie

Program bol testovaný v *Pythone 3.9* a *PyPy 3.11* na Linuxe s x86 64 bitovou architektúrou. Potrebné knižnice na spustenie sú v súbore **requirements.txt**.

A.2 Popis súborov

Program je rozdelený do súborov **data_structures.py**, **analyses.py**, **results_visualisations.py** v priečinku **memory_address_traces**.

- V súbore **data_structures.py** sa nachádzajú dátové štruktúry používané v programe. Sú tu štruktúry reprezentujúce vstupné súbory zoznamov pristupovaných adries vytvorené nástrojmi *Pin* a *Dynamorio* a grafová štruktúra postupnosti pristupovaných adries.
- V súbore **analyses.py** sú implementované jednotlivé nástroje určené na analyzovanie pristupovaných adries.
- V súbore **results_visualisations.py** sú pomocné funkcie, ktoré vizualizujú alebo vypisujú výsledky získané pomocou nástrojov.
- Ukážkový program vysvetľujúci základné použitie sa nachádza v **example.py**
- Príklady niektorých výsledkov a vizualizácií je možné nájsť v priečinku **results_examples**