УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Ирена Савић

СТАТИЧКА АНАЛИЗА КОДА СА ЦИЉЕМ ОПТИМИЗАЦИЈЕ ЗАУЗЕЋА МЕМОРИЈЕ

мастер рад

Београд, 2024.

Ментор:

др Мирко Спасић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Филип Марић, редовни професор
Универзитет у Београду, Математички факултет

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

Датум одбране: _____

Породици

Наслов мастер рада: Статичка анализа кода са циљем оптимизације заузећа меморије

Резиме: Ефикасност софтвера игра важну улогу у одрживости и перформансама система, што подстиче потребу за његовом оптимизацијом. Статичка анализа кода с циљем оптимизације заузећа меморије представља виталан корак у развоју софтвера који не само да побољшава перформансе апликација, већ доприноси и смањењу трошкова хардвера и енергетске потрошње. Ова тема истражује како прецизно и ефикасно анализирати кôд како би се идентификовала, минимизовала и елиминисала непотребна употреба меморије. Значај овог приступа је посебно евидентан у развоју уређаја са уграђеним рачунаром, где су ресурси ограничени и где оптимално управљање меморијом може имати пресудан утицај на перформансе и корисничко искуство. Циљ овог рада је развој методологије за статичку анализу ELF бинарних датотека, усмерен ка идентификацији потенцијалних места где је оптимизација употребе меморије могућа. При анализи се користе алати попут *readelf* и *radare2*, како би се прикупили подаци о коришћењу меморије, што омогућава детаљно разумевање њеног заузећа у свакој појединачној компоненти анализираног софтвера. Осим тога, биће представљена и практична имплементација система за анализу и визуализацију резултата, у циљу пружања јаснијег разумевања могућности за оптимизацију.

Кључне речи: статичка анализа кода, ELF бинарне датотеке, DWARF секција, алат *readelf*, алат *radare2*, оптимизација меморије, меморијско заузеће

Садржај

Глава 1

Увод

Развој софтвера подразумева сложене процесе чији је циљ не само функционалност и исправност програма, већ и његова ефикасност. Један од кључних аспеката ефикасности софтвера је управљање меморијом, што има директан утицај на перформансе система, трошкове хардвера и енергетску потрошњу. Овај изазов постаје посебно значајан у контексту система са уграђеним рачунаром (енг. *embedded devices*) и других уређаја са ограниченим ресурсима.

У овом раду биће истражене могућности статичке анализе кода као средства за стицање увида у заузеће меморијских ресурса у софтверским апликацијама. Статичка анализа омогућава да се, без покретања програма, прегледају различити аспекти кода који могу утицати на ефикасност употребе меморије. Анализа ће се фокусирати на ELF (енг. *Executable and Linkable Format*) формат бинарних датотека, при чему ће се користити алати као што су *readelf* и *radare2* који ће омогућити екстракцију података о меморијском заузећу.

Циљ овог рада је развој методологије која ће омогућити програмерима да идентификују потенцијална места у коду која могу довести до непотребне употребе меморије. Оваква анализа може бити кључна за препознавање меморијски интензивних делова кода, што је посебно корисно у фазама оптимизације софтвера.

Рад ће најпре пружити детаљан преглед формата ELF и DWARF (Глава 2), који ће служити као основа за даљу анализу. Након тога, биће описани постојећи алати који ће се користити за анализу, примери њихове употребе и добијени резултати (Глава 3).

У завршном делу, биће представљен алат *ELFAnalyzer*, развијен у оквиру овог мастер рада, чији је циљ анализа и визуализација података о меморијском

заузећу (Глава 4). Алат *ELFAnalyzer* врши статичку анализу меморијског заузећа на основу ELF датотека. Анализира величине глобалних променљивих и функција, што омогућава увид у заузеће меморије пре извршавања програма. У уграђеним системима, где се цео софтвер интегрише на чип са ограниченим ресурсима, ова информација је од велике важности. Меморијски ресурси у таквим системима су фиксно распоређени, па је неопходно да се статички ресурси оптимизују како би се програм ефикасно уклопио у расположиве меморијске сегменте. Алат пружа програмерима јаснији увид у коришћење меморијских ресурса и идентификују могућности за унапређење. Конкретно, биће показано како екстраховани подаци могу да укажу на критичне делове кода који заузимају највише меморије, што ће бити илустровано кроз визуализацију заузећа меморије појединачних софтверских компоненти кроз време. Континуирано праћење заузећа меморије омогућава програмерима да уоче значајне промене у меморијском оптерећењу појединачних компоненти. На пример, ако након ажурирања програма нека компонента нагло повећа заузеће меморије, то може сигнализирати постојање потенцијалних грешака, као што су погрешно повезивање непотребних модула или компилација без укључених меморијских оптимизација. Рано уочавање оваквих промена може помоћи у превенцији проблема у каснијим фазама развоја.

На крају, биће дат осврт на цео рад, укључујући разматрање практичне примене развијене методологије и могућности за даље проширење и унапређење алата *ELFAnalyzer* (Глава 5).

Глава 2

Бинарни формати ELF и DWARF

Ово поглавље пружа преглед основних елемената објектних датотека, укључујући анализу формата a.out и детаљан опис бинарних формата ELF и DWARF.

2.1 Објектне датотеке

Објектне датотеке представљају суштинску компоненту у процесу превођења изворног кода у извршиви програм. Компилатори и асемблери креирају објектне датотеке као међупроизводе током компилације, које се затим повезују коришћењем повезивача (енг. *linker*) и учитавају у меморију помоћу пуниоца (енг. *loader*). Објектне датотеке садрже машински кôд и друге метаподатке неопходне за правилно повезивање података и дебаговање. Оне представљају основу за креирање програма у извршивом облику (енг. *executable*).

Структура објектне датотеке

Објектна датотека садржи пет врста информација:

- *Информације у заглављу:* Опште информације о датотеци, као што су величина кода, назив изворне датотеке из које је преведена и датум креирања.
- *Објектни кôд:* Бинарне инструкције и подаци које генерише компилатор или асемблер.

- *Табела релокација:* Листа места у објектном коду која морају бити исправљена када повезивач промени адресе приликом повезивања у извршиву датотеку.

- *Симболи:* Глобални симболи дефинисани у овом модулу, симболи који се укључују из других модула или симболи које дефинише повезивач.

- *Информације за дебаговање:* Додатне информације о објектном коду које нису потребне за повезивање, али се користе за дебаговање. Ово укључује информације о изворној датотеци и броју линија у датотеци, локалним симболима, описима структура података које користи објектни кôд, као што су дефиниције структура у језику C[1].

Не садрже сви објектни формати све ове врсте информација, и могуће је имати веома корисне формате са мало или без икаквих додатних информација осим објектног кода [3].

Дизајн објектног формата

Дизајн објектног формата представља компромис вођен различитим начинима употребе објектне датотеке. Датотека може бити:

1. повезујућа (енг. *linkable*), користи се као улаз за едитор повезивача или пунилац за повезивање,

2. извршива (енг. *executable*), има могућност да се учита у меморију и покрене као програм,

3. датотека која се учитава (енг. *loadable*), може да се учита у меморију као библиотека заједно са програмом,

4. или било која комбинација ова три начина.

Неки формати подржавају само једну или две од ових употреба, док други подржавају све три.

Повезујућа датотека садржи опсежне информације о симболима и релокацији које су потребне повезивачу, заједно са објектним кодом. Релокација представља процес додељивања и прилагођавања меморијских адреса података и

[1]Неке објектне датотеке садрже, поред ових пет врста, и друге типове информација.

кода програма који зависе од позиције у меморији. Ово укључује промену адреса у коду и подацима како би одговарале конкретним меморијским адресама током извршавања програма. Овај процес омогућава програмима да се правилно учитају и извршавају на различитим меморијским локацијама, што је неопходно за правилно функционисање програма након повезивања објектних датотека у један извршиви програм [7]. Објектни кôд је често подељен на много малих логичких сегмената које ће повезивач различито обрадити. Извршива датотека садржи објектни кôд, обично поравнат на странице како би омогућио пресликавање датотеке у адресни простор, али не захтева симболе (осим ако неће радити динамичко повезивање у току извршавања) и захтева мало или нимало информација о релокацији. У зависности од детаља извршног окружења система (енг. *runtime environment*), датотека која се учитава може садржати само објектни кôд или може укључивати комплетне информације о симболима и релокацији како би омогућила симболичко повезивање током извршавања.

Постоји одређени сукоб између различитих намена објектног формата. Логички оријентисано груписање повезујућих сегмената не мора увек да се поклапа са хардверски оријентисаним груписањем извршивих сегмената. Повезивач учитава повезујуће датотеке део по део, док се извршиве датотеке учитавају по потреби у главну меморију током извршавања. Ова разлика је приметна између повезујућег формата *COFF*[2] и извршивог формата *PE*[3].

2.2 Формат a.out

a.out (енг. *assembler output*) је један од раних и широко коришћених формата у *Unix* оперативним системима. Његова једноставна структура омогућавала је брз развој и имплементацију, што је било од суштинског значаја за ране фазе развоја софтвера.

Кратка историја

Формат a.out развијен је почетком 1970-их као стандард за објектне и извршиве датотеке у *Unix* системима. Његов циљ био је да обезбеди једноставан

[2] *COFF* (*Common Object File Format*): Повезујући формат објектне датотеке коришћен у *Windows* систему.

[3] *PE* (*Portable Executable*): Извршиви формат датотеке коришћен у *Windows* систему за покретање програма.

и брз начин за складиштење програма у форми која је погодна за извршавање. Широко је прихваћен због своје једноставности и способности да подржи основне захтеве за компилацију и повезивање.

Иако је временом превазиђен напреднијим форматима као што је ELF, формат a.out је задржао своје место у историји као значајан корак у развоју *Unix* оперативних система. Име a.out се и даље користи као подразумевано име за излазне датотеке које креирају многи компилатори, иако модерни системи не користе његову оригиналну структуру.

2.3 Структура формата a.out

Формат a.out се састоји од неколико кључних компоненти:

- **Заглавље (енг. *header*)**:
  - Магични број: Идентификује датотеку као a.out датотеку.
  - Величине сегмената: Спецификује величине различитих сегмената унутар датотеке, укључујући сегмент текста програма (енг. *text segment*), сегмент података (енг. *data segment*) и сегмент неиницијализованих података (енг. *bss segment*).
  - Улазна тачка (енг. *entry point*): Адреса на којој програм почиње са извршавањем.
- **Сегменти (енг. *segments*)**:
  - Сегмент текста програма (енг. *text segment*): Садржи извршиви кôд програма.
  - Сегмент података (енг. *data segment*): Садржи иницијализоване променљиве.
  - Сегмент неиницијализованих података (енг. *bss, block started by symbol segment*): Садржи неиницијализоване статички алоциране променљиве које се могу иницијализовати на нулу при покретању програма.
- **Табела симбола (енг. *symbol table*)**: Садржи информације о симболима коришћеним у коду, као што су имена функција и променљивих. Симболи помажу повезивачу да правилно повеже различите делове програма.

- **Релокација (енг. *relocation*)**: Табела релокације садржи информације које повезивач користи да прилагоди адресе када се датотека учитава у меморију. Ове информације омогућавају исправно повезивање референци између различитих сегмената.

Формат a.out је пружао основне функционалности потребне за повезивање и извршавање програма, али његова једноставност постала је препрека за напредније функционалности као што су динамичко повезивање и дебаговање. Како су захтеви за софтвером расли, постала су очигледна нека ограничења формата a.out:

- **Ограничена подршка за динамичко повезивање**: Формат a.out није био дизајниран за подршку дељеним библиотекама које се могу динамички учитати, што је постало неопходно са растом сложености апликација.
- **Недостатак флексибилности**: Фиксна структура формата a.out отежавала је проширења и прилагођавања за различите потребе и архитектуре.
- **Ограничена подршка за дебаговање**: Недостатак детаљних информација о симболима и релокацији отежавао је процес дебаговања и анализу програма.

Упркос овим ограничењима, формат a.out је одиграо важну улогу у развоју раних *Unix* система, омогућавајући основне функционалности за превођење и извршавање кода. Међутим, како су захтеви за софтвером расли, постало је јасно да су потребна напреднија решења. То је довело до развоја формата ELF, који је превазишао ограничења формата a.out и омогућио већу флексибилност, бољу подршку за динамичко повезивање и детаљније информације за дебаговање. a.out остаје подразумевано име излазне датотеке за извршиве датотеке које креирају одређени компилатори и повезивачи када није спецификовано име излазне датотеке, иако креиране датотеке заправо нису у формату a.out [9].

2.4 Формат ELF

ELF (извршиви и повезујући формат, енг. *Executable and Linkable Format*) је стандардни формат датотека који се користи за извршиве датотеке, објектни

кôд и дељене библиотеке (енг. *shared libraries*). ELF је развијен како би се обезбедио јединствени формат који може радити на различитим оперативним системима и хардверским платформама.

Формат ELF је важан јер стандардизује начин на који су бинарне датотеке структуриране, што олакшава развој и одржавање софтвера. Програмерима пружа дефинисане бинарне интерфејсе који функционишу кроз различита оперативна окружења, смањујући потребу за поновним кодирањем и превођењем кода када се прелази са једног система на други. Ово значајно поједностављује развој софтвера и доприноси бољој интероперабилности између различитих система [8].

Кратка историја

Компанија *UNIX System Laboratories (USL)* први пут је објавила ELF као део интерфејса за апликације у бинарном облику (енг. *Application Binary Interface (ABI)*) у оквиру оперативног система *Unix System V Release 4 (SVR4)* почетком 1990-их [8].

Настао је из потребе за стандардизацијом бинарних формата датотека како би се олакшала интероперабилност између различитих *Unix* система. Пре формата ELF, извршиве и објектне датотеке су биле структуриране на различите начине, што је отежавало дељење и поновну употребу кода међу различитим системима.

ELF је постао популаран због своје флексибилности и проширивости. Он подржава различите *endian*[4] формате и величине адреса, што омогућава његову употребу на различитим архитектурама процесора и оперативним системима. Ова флексибилност је омогућила широко прихватање формата ELF у многим *Unix* и *Unix*-сличним системима, као и у другим оперативним системима.

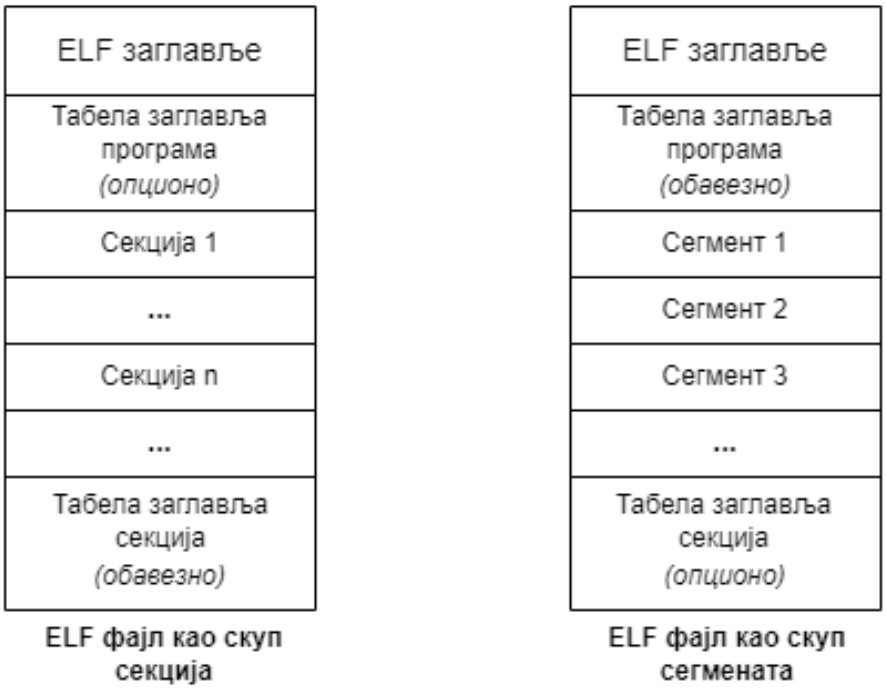
2.5 Структура ELF датотека

ELF датотеке долазе у три различите верзије: релокациона, извршива и дељени објекат (енг. *shared object*). Релокационе датотеке креирају компилатори и асемблери, а затим их повезивач обрађује пре извршавања. Извршиве да-

[4]endian је начин на који се бајтови редом постављају у меморији. Постоје два главна типа: big-endian и little-endian, где први значи да се бајт највише тежине смешта на најнижу адресу, а други да се бајт највише тежине смешта на највишу адресу.

тотеке имају завршену релокацију и све симболе решене, осим можда симбола из дељених библиотека који се решавају у току извршавања. Дељени објекти су дељене библиотеке које садрже информације о симболима за повезивач и директно извршиви кôд за време извршавања програма [3, 8].

ELF датотеке имају необичну дуалну природу, што се може видети на слици 2.1. Компилатори, асемблери и повезивачи третирају датотеку као скуп логичких секција описаних у табели заглавља секција, док системски пунилац третира датотеку као скуп сегмената описаних у табели заглавља програма. Један сегмент обично се састоји од неколико секција. На пример, сегмент „loadable read-only" може садржати секције за извршиви кôд, податке који могу само да се читају (енг. *read-only*) и симболе за динамичко повезивање. Релокационе датотеке имају табеле секција, извршиве датотеке имају табеле заглавља програма, а дељени објекти имају оба. Повезивач обрађује секције за даљу употребу, док су сегменти намењени за пресликавање у меморију.



Слика 2.1: Два тумачења ELF датотека[5].

[5]Иако слика приказује табелу заглавља програма одмах након ELF заглавља и табелу заглавља секција након секција, стварне датотеке могу се разликовати. Штавише, секције и сегменти немају одређени редослед. Само ELF заглавље има фиксну позицију у датотеци.

ELF заглавље

ELF заглавље (енг. *ELF header*) је први део сваке ELF датотеке и садржи основне информације о датотеци. Фиксно је позиционирано на почетку датотеке и описује формат и структуру саме датотеке. Према спецификацији формата ELF, структура ELF заглавља је дефинисана као у листингу 2.1.

```
1  typedef struct {
2      unsigned char e_ident[EI_NIDENT];
3      uint16_t e_type;
4      uint16_t e_machine;
5      uint32_t e_version;
6      uint64_t e_entry;
7      uint64_t e_phoff;
8      uint64_t e_shoff;
9      uint32_t e_flags;
10     uint16_t e_ehsize;
11     uint16_t e_phentsize;
12     uint16_t e_phnum;
13     uint16_t e_shentsize;
14     uint16_t e_shnum;
15     uint16_t e_shstrndx;
16 } Elf64_Ehdr;
```

Листинг 2.1: Структура ELF заглавља.

Поља ELF заглавља:

- `e_ident`: Идентификациони бајтови ELF датотеке.
- `e_type`: Тип објектне датотеке (релокациона, извршива, дељени објекат).
- `e_machine`: Тип архитектуре/процесора.
- `e_version`: Верзија ELF датотеке.
- `e_entry`: Адреса улазне тачке (*entry point*).
- `e_phoff`: Позиција табеле заглавља програма у датотеци.
- `e_shoff`: Позиција табеле заглавља секција у датотеци.
- `e_flags`: Процесорске заставице.
- `e_ehsize`: Величина ELF заглавља.

- `e_phentsize`: Величина уноса у табели заглавља програма.
- `e_phnum`: Број уноса у табели заглавља програма.
- `e_shentsize`: Величина уноса у табели заглавља секција.
- `e_shnum`: Број уноса у табели заглавља секција.
- `e_shstrndx`: Индекс табеле низова имена секција.

Табела заглавља програма

Табела заглавља програма (енг. *program header table*) описује сегменте програма који се учитавају у меморију. Сваки унос у табели описује један сегмент (листинг 2.2).

```
1  typedef struct {
2      uint32_t p_type;
3      uint32_t p_flags;
4      uint64_t p_offset;
5      uint64_t p_vaddr;
6      uint64_t p_paddr;
7      uint64_t p_filesz;
8      uint64_t p_memsz;
9      uint64_t p_align;
10 } Elf64_Phdr;
```

Листинг 2.2: Структура сегмента.

Поља програмског заглавља:

- `p_type`: Тип сегмента (LOAD, DYNAMIC, INTERP, NOTE, итд.).
- `p_flags`: Атрибути сегмента (READ, WRITE, EXECUTE).
- `p_offset`: Позиција сегмента у датотеци.
- `p_vaddr`: Виртуелна адреса у меморији где сегмент треба бити учитан.
- `p_paddr`: Физичка адреса (користи се у неким системима).
- `p_filesz`: Величина сегмента у датотеци.
- `p_memsz`: Величина сегмента у меморији.
- `p_align`: Поравнање сегмента.

Табела заглавља секција

Табела заглавља секција (енг. *section header table*) описује секције које су корисне за повезивање и дебаговање. Сваки унос у табели описује једну секцију (листинг 2.3).

```
typedef struct {
    uint32_t sh_name;
    uint32_t sh_type;
    uint64_t sh_flags;
    uint64_t sh_addr;
    uint64_t sh_offset;
    uint64_t sh_size;
    uint32_t sh_link;
    uint32_t sh_info;
    uint64_t sh_addralign;
    uint64_t sh_entsize;
} Elf64_Shdr;
```

Листинг 2.3: Структура секције.

Поља заглавља секција:

- `sh_name`: Име секције (индекс у табели ниски).
- `sh_type`: Тип секције (PROGBITS, SYMTAB, STRTAB, RELA, HASH, DYNAMIC, NOTE, итд.).
- `sh_flags`: Заставице секције (WRITE, ALLOC, EXECINSTR).
- `sh_addr`: Виртуелна адреса секције при извршавању.
- `sh_offset`: Позиција секције у датотеци.
- `sh_size`: Величина секције у бајтовима.
- `sh_link`: Веза ка другој секцији.
- `sh_info`: Додатне информације о секцији.
- `sh_addralign`: Поравнање секције.
- `sh_entsize`: Величина једног уноса ако секција садржи табелу.

Секције

Секције (енг. *section*) су основне јединице које садрже кôд, податке, симболе и релокационе информације.

- **.text**: Садржи извршиви кôд.
- **.data**: Садржи иницијализоване податке.
- **.bss**: Садржи неиницијализоване податке који се иницијализују на нулу.
- **.rodata**: Садржи податке само за читање.
- **.symtab**: Садржи табелу симбола.
- **.strtab**: Садржи табелу ниски.
- **.rel.text** или **.rela.text**: Садржи релокационе информације за .text секцију.

Динамичка секција

Динамичка секција (енг. *dynamic section*) садржи информације потребне за динамичко повезивање.

```
typedef struct {
    int64_t d_tag;
    union {
        uint64_t d_val;
        uint64_t d_ptr;
    } d_un;
} Elf64_Dyn;
```

Листинг 2.4: Структура динамичке секције.

Поља динамичке секције:

- `d_tag`: Тип динамичког уноса (`DT_NEEDED`, `DT_PLTRELSZ`, `DT_PLTGOT`, `DT_HASH`, `DT_STRTAB`, `DT_SYMTAB`, `DT_RELA`, `DT_REL`, `DT_INIT`, `DT_FINI`, итд.).
- `d_val/d_ptr`: Целобројна вредност или адреса зависно од типа уноса.

DWARF секције

ELF датотека, поред поменутих основних компоненти, такође може садржати и специјализоване секције за информације за дебаговање. Једне од најважнијих таквих секција су секције које користе формат DWARF.

2.6 Формат DWARF

DWARF (енг. *Debugging With Arbitrary Record Formats*) је формат за складиштење информација за дебаговање који омогућава алатима за дебаговање да пруже детаљне информације о програму. Ове информације за дебаговање обухватају пресликавање изворног кода, симболичка имена, типове података и структуре података.

Кратка историја

DWARF је развио др Брајан Расел у *Беловим лабораторијама* (енг. *Bell Labs*) 1988. године за употребу са C компилатором и sdb дебагером у *Unix System V Release 4 (SVR4)*. Верзија 1 је стандардизована 1992. године и прихваћена у сектору уређаја са уграђеним рачунаром. *PLSIG* (*Programming Languages Special Interest Group*, део *Unix International* удружења) је наставио развој DWARF-а како би смањио величину података за дебаговање и додао подршку за нове језике као што је C++. Верзија 2 објављена је као нацрт стандарда 1993. године, али финални стандард никада није објављен.

DWARF 3 је развијен под окриљем *Free Standards Group* и објављен је у децембру 2005. године. Ова верзија је додала бољу подршку за архитектуру *HP/Intel IA-64* и побољшала документацију. DWARF 4 је објављен у јуну 2010. године и додата је подршка за архитектуре *VLIW*, побољшана је подршка за језике и компресија података за дебаговање. DWARF верзија 5 је објављена 2017. године. Донела је бројна побољшања и нове функције, укључујући побољшану подршку за паралелне архитектуре и додатне информације о дебаговању. Тренутно је у развоју верзија 6 DWARF стандарда [2, 6, 1].

Преглед формата DWARF

Већина модерних програмских језика је блоковски структурирана: сваки ентитет (нпр. дефиниција класе или функције) је садржан у другом ентитету. Свака изворна датотека у C програму може садржати више дефиниција података, више дефиниција променљивих и више функција. Унутар сваке C функције могу бити више дефиниција података праћених извршивим наредбама. Нека наредба може бити сложена наредба која садржи дефиниције података и извршиве наредбе, што ствара лексичке опсеге где су имена позната само унутар опсега у којима су дефинисана.

DWARF прати овај модел тако што је такође блоковски структуриран. Сваки описни ентитет у DWARF-у (осим највишег који описује изворну датотеку) је садржан у родитељском и може садржати подентитете. Ако чвор садржи више ентитета, они су сви рођаци, односно повезани. Опис програма у DWARF-у је дрволика структура, слична апстрактном синтаксном стаблу компилатора, где сваки чвор може имати децу или рођаке. Чворови могу представљати типове, променљиве или функције. Ово је компактан формат где се пружају само информације потребне за опис аспекта програма. Формат је проширив на униформан начин, тако да дебагер може препознати и игнорисати проширење, чак и ако не разуме његово значење.

Дизајниран је такође да може описати готово сваки процедурални програмски језик на било којој машинској архитектури, уместо да буде везан за један језик или једну верзију језика на ограниченом броју архитектура.

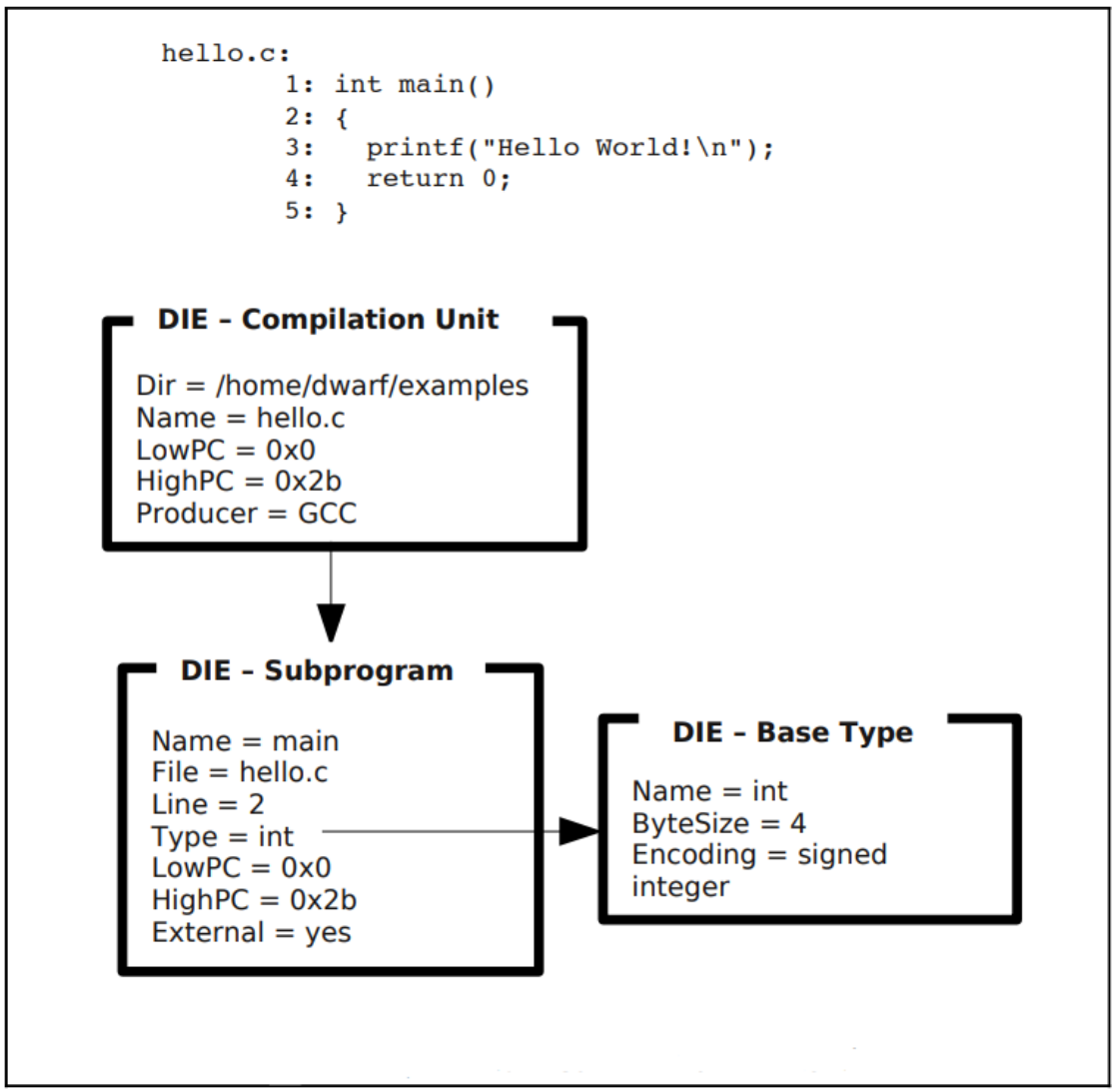
Иако је DWARF дефинисан на начин који омогућава његову употребу са било којим форматом објектне датотеке, најчешће се користи са форматом ELF. Сваки од различитих типова DWARF података се чува у својој секцији. Називи ових секција сви почињу са `.debug_`. Неке од њих су `.debug_info`, `.debug_abbrev`, `.debug_line`, `.debug_str` и др.

2.7 Структура DWARF секције

У овом поглављу описује се структура DWARF секције, укључујући уносе информација за дебаговање (DIE), њихове типове и композицију типова. Представљено је и како се неки сложенији типови и функције приказују у формату DWARF.

Унос информација за дебаговање (DIE)

Основни описни ентитет у DWARF-у је унос информација за дебаговање - DIE (енг. *Debugging Information Entry*). DIE има таг који спецификује шта DIE описује и листу атрибута који садрже детаље који описују ентитет. Сваки DIE (осим највишег) је садржан у или припада родитељском DIE и може имати рођаке или децу DIE. Атрибути могу садржати различите вредности: константе (као што је име функције), променљиве (као што је почетна адреса функције) или референце на други DIE (као што је тип вредности коју функција враћа).



Слика 2.2: Графичка репрезентација DWARF података [2].

На слици 2.2 је приказан једноставан C програм `hello.c` са поједностав-

ном графичком репрезентацијом његовог DWARF описа. Највиши DIE представља јединицу компилације (енг. *compilation unit*). Има два детета: први је DIE који описује функцију *main*, а други описује базни тип *int*, који је тип вредности коју *main* враћа. DIE потпрограма је дете DIE јединице компилације, док се базни тип DIE реферише атрибутом *Type* у DIE потпрограму.

Типови уноса информација за дебаговање (DIE)

DIE уноси се могу поделити у две опште категорије: они који описују податке, укључујући типове података, и они који описују функције и други извршиви кôд.

Већина програмских језика има софистициране описе података са различитим уграђеним типовима података, показивачима и структурама података, као и начине за креирање нових типова података. DWARF апстрахује основне типове података и представља их на начин такав да се могу користити за све подржане језике. Примарни типови, подржани директно на хардверу, су основни типови (енг. *base types*). Остали типови података се конструишу као колекције или композиције ових основних типова.

Сваки програмски језик дефинише неколико основних скаларних типова података. На пример, и C и Java дефинишу *int* и *double*. Java пружа потпуну дефиницију ових типова, док C само одређује неке опште карактеристике, дозвољавајући компилатору да изабере стварне спецификације које најбоље одговарају циљном процесору.

DWARF основни типови пружају најнижи ниво пресликавања између једноставних типова података и њихове имплементације на хардверу циљне машине. Ово чини дефиницију типа *int* експлицитном и за Java и за C и омогућава коришћење различитих дефиниција унутар истог програма.

```
1 DW_TAG_base_type
2         DW_AT_name = int
3         DW_AT_byte_size = 4
4         DW_AT_encoding = signed
```

Листинг 2.5: Основни тип *int* на 32-битном процесору.

```
1 DW_TAG_base_type
2         DW_AT_name = int
3         DW_AT_byte_size = 2
```

```
4           DW_AT_encoding = signed
```

Листинг 2.6: Основни тип *int* на 16-битном процесору.

Листинг 2.5 приказује DIE који описује тип *int* на типичном 32-битном процесору. Атрибути одређују име (*int*), енкодирање (означени целобројни тип) и величину у бајтовима (4). Листинг 2.6 показује сличну дефиницију *int* на 16-битном процесору.

```
1 DW_TAG_base_type
2           DW_AT_name = word
3           DW_AT_byte_size = 4
4           DW_AT_bit_size = 16
5           DW_AT_bit_offset = 0
6           DW_AT_encoding = signed
```

Листинг 2.7: 16-битни тип *word* смештен на виших 16-битова 32-битне речи.

Основни типови омогућавају компилатору да опише скоро свако пресликавање између скаларног типа програмског језика и начина на који је заправо имплементиран на процесору. Листинг 2.7 описује 16-битну целобројну вредност која је смештена на виших 16 битова речи од 4 бајта. У овом основном типу постоји атрибут величине у битовима (`DW_AT_bit_size`) који одређује да је вредност величине 16 битова и померај (`DW_AT_bit_offset`) од нултог бита на вишој позицији.

Композиција типова

Именовану променљиву описује DIE који има различите атрибуте, од којих је један референца на дефиницију типа. Листинг 2.8 описује целобројну променљиву x. DIE `DW_TAG_base_type` описује *int* као означени целобројни тип који заузима 4 бајта, док `DW_TAG_variable` DIE описује име и тип, који показује на основни тип.

```
1 <1>: DW_TAG_base_type
2                   DW_AT_name = int
3                   DW_AT_byte_size = 4
4                   DW_AT_encoding = signed
5
6 <2>: DW_TAG_variable
```

```
7                   DW_AT_name = x
8                   DW_AT_type = <1>
```

Листинг 2.8: DWARF опис променљиве „*int x*”.

DWARF користи основне типове за конструисање других типова података помоћу композиције. Нови тип се креира као модификација другог типа. На пример, листинг 2.9 приказује показивач на *int* на типичној 32-битној машини. Овај DIE дефинише тип показивача, спецификује да је његова величина 4 бајта, и затим показује на основни тип *int*. Пример сложенијег типа као што је „*const char **argv*” је описан у листингу 2.10.

```
1 <1>: DW_TAG_variable
2                   DW_AT_name = px
3                   DW_AT_type = <2>
4
5 <2>: DW_TAG_pointer_type
6                   DW_AT_byte_size = 4
7                   DW_AT_type = <3>
8
9 <3>: DW_TAG_base_type
10                  DW_AT_name = int
11                  DW_AT_byte_size = 4
12                  DW_AT_encoding = signed
```

Листинг 2.9: DWARF опис показивача „*int *px*”.

```
1 <1>: DW_TAG_variable
2                   DW_AT_name = argv
3                   DW_AT_type = <2>
4
5 <2>: DW_TAG_pointer_type
6                   DW_AT_byte_size = 4
7                   DW_AT_type = <3>
8
9 <3>: DW_TAG_pointer_type
10                  DW_AT_byte_size = 4
11                  DW_AT_type = <4>
12
```

```
13 <4>: DW_TAG_const_type
14                  DW_AT_type = <5>
15
16 <5>: DW_TAG_base_type
17                  DW_AT_name = char
18                  DW_AT_byte_size = 4
19                  DW_AT_encoding = unsigned
```

Листинг 2.10: DWARF опис „*const char **argv*".

Неки сложенији типови података

Већина програмских језика подржава груписање података у неке сложеније структуре. Иако сваки језик има своју терминологију, сви ови типови података могу да се опишу у DWARF-у. Па тако, постоје DIE за структуре, уније, класе и интерфејсе. DIE за класу је родитељски DIE за остале који описују чланове податке класе. Свака класа има име и опционо друге атрибуте. Описивање изгледа веома слично опису једноставне променљиве, иако могу постојати неки додатни атрибути[6]. Остали типови података имају сличну организацију.

Променљиве су генерално прилично једноставне. Имају име које представља део меморије (или регистра) који може садржати неку вредност. Типови вредности које променљива може садржати, као и ограничења на то како се може мењати (нпр. да ли је константна) су описани типом променљиве.

Оно по чему се разликују променљиве је где се вредност променљиве чува као и њен опсег. Опсег дефинише где је променљива позната унутар програма. DWARF чува информације где је променљива декларисана у изворној датотеци као уређену тројку (датотека, линија, колона).

Променљиве су у DWARF-у подељене у три категорије: константе, формални параметри и променљиве. Константе су именоване вредности које се не могу мењати, формални параметри су вредности прослеђене функцијама, а променљиве су опште меморијске локације које могу садржати различите вредности [1].

[6] C++ омогућава одређивање да ли је члан јавни (енг. *public*), приватни (енг. *private*) или заштићени (енг. *protected*).

Функције

За представљање функција, DWARF користи потпрограмски DIE (`DW_TAG_subprogram`). Он има име, локацију у изворном коду и атрибут који указује да ли је у питању екстерни потпрограм, тј. да ли је видљив изван тренутне јединице компилације. Потпрограмски DIE има атрибуте који означавају нижу и вишу меморијску адресу за опсег који та функција заузима, или листу меморијских опсега, ако функција не заузима континуирани простор. Нижа меморијска адреса се подразумева као улазна тачка потпрограма (енг. *entry point*) уколико није експлицитно наведена. Повратна вредност функције је описана типским атрибутом (енг. *type*), док функције које немају повратну вредност (као *void* фунцкије) једноставно немају овај атрибут. Такође, потпрограмски DIE садржи и DIE-ове који описују параметре функције и локалне променљиве, при чему се прати редослед у листи аргумената.

Јединица компилације

Већина програма се састоји од више датотека. Свака изворна датотека која чини програм се компилира независно и затим се повезује заједно са системским библиотекама да би се направио програм. DWARF назива сваку засебно компилирану изворну датотеку јединицом компилације (енг. *compilation unit*).

DWARF подаци за сваку јединицу компилације почињу са *Compilation Unit DIE*. Овај DIE садржи опште информације о компилацији, укључујући директоријум и име изворне датотеке, програмски језик који се користи, низ који идентификује произвођача DWARF података и помераје у DWARF секцијама података како би се олакшало проналажење информација о бројевима линија и макроима.

Ако је јединица компилације континуирана (тј. учитана је у меморију у целости), тада постоје вредности за доње и горње меморијске адресе за јединицу компилације. Ово олакшава дебагеру да идентификује која је јединица компилације креирала кôд на одређеној меморијској адреси. Ако јединица компилације није континуирана, тада компилатор и повезивач обезбеђују листу меморијских адреса које кôд заузима.

Compilation Unit DIE је родитељ свих DIE-ова који описују јединицу компилације. Генерално, први DIE-ови ће описивати типове података, затим глобалне податке, а онда функције које чине изворну датотеку. DIE-ови за променљиве

и функције су у истом редоследу као у изворно датотеци.

Глава 3

Алати за анализу ELF датотека

Постоје различити алати који омогућавају преглед и интерпретацију садржаја ELF датотека. Неки од најпознатијих таквих алата су:

readelf: Алат из пакета *GNU binutils*[1] који омогућава преглед информација о ELF датотекама, укључујући заглавља, секције, симболе и информације за дебаговање.

objdump: Користи се за дисасемблирање[2] и приказ различитих информација о ELF датотекама, такође део пакета *GNU binutils*.

elfutils: Колекција алата и библиотека за манипулацију ELF датотека, коју је развила компанија *Red Hat*.

Ghidra: Софтвер за реверзно инжењерство који је развила агенција *NSA* (америчка *Национална Безбедносна Агенција*, енг. *National Security Agency*) и који између осталог пружа и напредне могућности за дисасемблирање и декомпилирање ELF датотека.

radare2: Комплетан оквир за реверзно инжењерство и анализу бинарних формата (укључујући и ELF) који подржава широк спектар функционалности.

[1] *GNU binutils* је колекција програмских алата за обраду бинарних и објектних датотека, коју подржава *GNU* пројекат [5].

[2] Дисасемблирање је процес претварања машинског кода извршиве датотеке назад у асемблерски језик.

3.1 Алат *readelf*

Алат *readelf* је један од најзначајнијих алата у оквиру пакета *GNU binutils*. Он омогућава детаљну анализу и преглед различитих компоненти ELF датотека, пружајући увид у структуру и садржај ових датотека. *readelf* се широко користи за преглед заглавља секција, симбола, динамичких секција као и информација за дебаговање.

Основне функционалности

Основна синтакса за коришћење алата:

```
readelf [options] <filename>
```

Где `options` представља опције којима се одређују информације које ће *readelf* извући и приказати, а `filename` је ELF датотека која се анализира, тачније путања до датотеке.

Неке од најважнијих опција су [4]:

-a или --all: Приказује све доступне информације о ELF датотеци. Ова опција је корисна када је потребно добити комплетан преглед свих секција и заглавља унутар ELF датотеке.

```
readelf -a <filename>
```

-h или --file-header: Приказује ELF заглавље датотеке. ELF заглавље садржи основне информације о типу датотеке, архитектури, верзији и улазној тачки програма.

```
readelf -h <filename>
```

-S или --section-headers или --sections: Приказује заглавља секција. Свака секција у ELF датотеци има своје заглавље које описује њен тип, величину и позицију у датотеци.

```
readelf -S <filename>
```

-s или --symbols: Приказује табелу симбола. Табела симбола садржи информације о свим симболима који су дефинисани или референцирани у датотеци.

```
readelf -s <filename>
```

-l или --program-headers: Приказује заглавља програма. Заглавља програма описују сегменте програма који ће бити учитани у меморију током извршавања.

```
readelf -l <filename>
```

-d или --dynamic: Приказује динамичке секције. Динамичке секције садрже информације потребне за динамичко повезивање.

```
readelf -d <filename>
```

--debug-dump[=type]: Приказује различите типове информација за дебаговање, односно различите DWARF секције. Неки од могућих типова су:

- `info`: Основне информације о симболима из секције `.debug_info`
- `abbrev`: Скраћене информације о симболима из секције `.debug_abbrev`
- `rawline`: Линије изворног кода из секције `.debug_line`

```
readelf --debug-dump=info <filename>
```

--dwarf-depth=N: Приказује DIE-ове до дубине N (не укључујући N)[3]

```
readelf --debug-dump=info --dwarf-depth=2 <filename>
```

[3]С обзиром да су DWARF информације структуриране као стабло, овде се мисли на дубину стабла.

Примери

У овој секцији биће приказани неки примери коришћења алата *readelf* за анализу ELF датотека једноставног C програма, који исписује „Hello, World!”.

Садржај датотеке `hello.c` приказан је у листингу 3.1:

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello, World!\n");
5     return 0;
6 }
```

Листинг 3.1: Садржај `hello.c` датотеке.

Да би се изгенерисала ELF датотека, потребно је извршити следећу наредбу приликом превођења:

```
gcc -o hello hello.c
```

Након чега ће се добити извршива ELF датотека названа `hello`.

Приказ ELF заглавља

За приказ информација из заглавља, користи се следећа наредба:

```
readelf -h hello
```

Након чега се добија излаз:

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400540
```

```
Start of program headers:          64 (bytes into file)
Start of section headers:          5024 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           56 (bytes)
Number of program headers:         9
Size of section headers:           64 (bytes)
Number of section headers:         30
Section header string table index: 29
```

Тумачење излаза:

- `Magic`: Магични бројеви који идентификују датотеку као ELF датотеку.
- `Class`: Означава да је у питању ELF64 датотека (64-битна).
- `Data`: Означава формат endian.
- `Version`: Верзија формата ELF.
- `OS/ABI`: Оперативни систем и *ABI (Application Binary Interface)*.
- `ABI Version`: Верзија *ABI*-ја (обично 0).
- `Type`: Тип ELF датотеке.
- `Machine`: Архитектура за коју је датотека компилирана.
- `Version`: Верзија ELF заглавља.
- `Entry point address`: Адреса улазне тачке (где програм почиње са извршавањем).
- `Start of program headers`: Позиција у односу на програмска заглавља (у бајтовима од почетка датотеке).
- `Start of section headers`: Позиција у односу на заглавља секција (у бајтовима од почетка датотеке).
- `Flags`: Специфичне заставице за процесор (обично 0).
- `Size of this header`: Величина ELF заглавља у бајтовима.

- `Size of program headers`: Величина сваког уноса у програмским заглављима у бајтовима.
- `Number of program headers`: Број уноса у програмским заглављима.
- `Size of section headers`: Величина сваког уноса у заглављима секција у бајтовима.
- `Number of section headers`: Број уноса у заглављима секција.
- `Section header string table index`: Индекс у табели стрингова заглавља секција.

Приказ DWARF информација до дубине 2

За приказ DWARF информација за дебаговање до дубине 2, користи се следећа наредба:

```
readelf --debug-dump=info --dwarf-depth=2 hello
```

Након чега се добија следећи излаз:

```
Contents of the .debug_info section:

  Compilation Unit @ offset 0x0:
   Length:        0x3a (32-bit)
   Version:       2
   Abbrev Offset: 0x0
   Pointer Size:  8
 <0><0x0>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <0x1>   DW_AT_producer    : (strp: 0x2e "GNU C11 11.2.0
                                -mtune=generic -march=x86-64 -g")
    <0x5>   DW_AT_language    : 12       (C11)
    <0x6>   DW_AT_name        : (strp: 0x43 "hello.c")
    <0xa>   DW_AT_comp_dir    : (strp: 0x4b "/home/user")
    <0xe>   DW_AT_low_pc      : 0x400540
    <0x16>   DW_AT_high_pc    : 0x400554
    <0x1e>   DW_AT_stmt_list  : 0x0
```

```
<1><0x3a>: Abbrev Number: 2 (DW_TAG_subprogram)
   <0x3b>   DW_AT_external   : 1
   <0x3c>   DW_AT_name       : (strp: 0x58 "main")
   <0x40>   DW_AT_decl_file  : 0x1
   <0x41>   DW_AT_decl_line  : 3
   <0x42>   DW_AT_prototyped : 1
   <0x43>   DW_AT_type       : (ref4) 0x5b
   <0x47>   DW_AT_low_pc     : 0x400540
   <0x4f>   DW_AT_high_pc    : 0x400554
   <0x57>   DW_AT_frame_base : (exprloc)
   <0x0>    DW_OP_call_frame_cfa
```

Тумачење излаза:

- `DW_TAG_compile_unit`: Овај таг означава јединицу компилације која представља датотеку `hello.c`.
- `DW_AT_producer`: Ниска која означава који компилатор је коришћен.
- `DW_AT_language`: Језик у којем је кôд написан.
- `DW_AT_name`: Име изворне датотеке.
- `DW_AT_comp_dir`: Директоријум у којем је датотека компилирана.
- `DW_AT_low_pc` и `DW_AT_high_pc`: Распон адреса у ком се налази кôд.
- `DW_AT_stmt_list`: Позиција у `.debug_line` секцији.
- `DW_TAG_subprogram`: Овај таг означава функцију `main`.
- `DW_TAG_external`: Индикација да ли је функција глобално доступна.
- `DW_AT_name`: Име функције (`main`).
- `DW_AT_decl_file` и `DW_AT_decl_line`: Локација у датотеци где је функција декларисана.
- `DW_AT_prototyped`: Индикација да ли функција има прототип.
- `DW_AT_type`: Тип повратне вредности функције (референца на други DIE који описује тип).

- `DW_AT_low_pc` и `DW_AT_high_pc`: Распон адреса у ком се налази кôд функције `main`.
- `DW_AT_frame_base`: Локација основне адресе стека.
- `DW_OP_call_frame_cfa`: DWARF операција која представља тренутни стек оквир.

3.2 Алат *radare2*

Пројекат *radare2* (познат и као `r2`) започео је Серђи Алварез (*Sergi Àlvarez*, познат под псеудонимом *pancake*) 2006. године са циљем да развије мали алат за опоравак избрисаних датотека са хард-диска. Првобитно замишљен као једноставан хексадецимални едитор, пројекат се касније развио у моћан алат и постао комплетан оквир за реверзно инжењерство и напредну анализу бинарних датотека. *radare2* се развија у оквиру отворене заједнице и континуирано се унапређује уз допринос великог броја корисника [10].

Алат *radare2* подржава велики број различитих архитектура и инструкцијских сетова. Ово омогућава корисницима да анализирају и модификују бинарне датотеке на различитим платформама, од стандардних архитектура попут *Intel x86* и *ARM*, до мање уобичајених попут *MIPS*, *PowerPC* и многих других. Корисници могу приступити свим функционалностима оквира *radare2* оквира путем интерактивног интерфејса који омогућава коришћење различитих инструкција или скрипти.

Кључне функције *radare2* укључују:

- Хексадецимално едитовање и измену бинарних података
- Напредну анализу кода и дисасемблирање
- Дебаговање у реалном времену
- Измену и поправку извршивих датотека
- Визуализацију података
- Подршку за аутоматизацију и коришћење скрипти

rabin2

Оквир *radare2* састоји се од већег броја алата који се могу користити за специфичне задатке. Један од најчешће коришћених алата у оквиру овог оквира је `rabin2`.

rabin2 је специјализован алат за извлачење различитих информација из бинарних датотека, као што су симболи, импортоване и експортоване функције, заглавља и секције. Подржава много различитих формата датотека (*Java CLASS, ELF, PE, Mach-O, ...*) [10].

Основне функционалности

Општа синтакса за коришћење изгледа овако:

```
rabin2 [-AcdeEghHiIjlLMqrRsSUvVxzZ] [-@ at] [-a arch] [-b bits] [-B addr]
       [-C F:C:D] [-f str] [-m addr] [-n str] [-N m:M] [-P[-P] pdb]
       [-o str] [-O str] [-k query] [-D lang mangledsymbol] <filename>
```

Објашњења неких опција:

- `-@ [addr]`: Секција, симбол или импортована функција на адреси `addr`.
- `-a [arch]`: Подешавање архитектуре (нпр. x86, arm, или <arch>_<bits>).
- `-b [bits]`: Подешавање битова (32, 64 ...).
- `-c`: Листа класа.
- `-d`: Приказ информација за дебаговање (*dwarf*).
- `-e`: Почетна тачка програма (*entry point*).
- `-E`: Глобални симболи који могу да се експортују.
- `-h`: Опис свих опција (*help*).
- `-H`: Поља заглавља.
- `-i`: Симболи импортовани из библиотека.
- `-I`: Информације о бинарној датотеци.
- `-j`: Излаз у формату *JSON*.

- `-l`: Повезујуће библиотеке.
- `-n [str]`: Секција, симбол или импортована функција по имену `str`.
- `-o [str]`: Излазна датотека/директоријум за операције писања.
- `-p`: Приказ физичких адреса.
- `-qq`: Приказ мање информација.
- `-r`: radare излаз.
- `-R`: Релокације.
- `-s`: Симболи.
- `-S`: Секције.
- `-SS`: Сегменти.
- `-SSS`: Секције мапиране на сегменте.
- `-t`: Хешеви датотека.
- `-T`: Потпис датотеке.
- `-V`: Информације о верзији бинарне датотеке.
- `-w`: Приказ `try/catch` блокова.
- `-z`: Ниске (из секције података).

Пример коришћења *rabin2* за извлачење симбола

За приказ симбола (-s) из ELF датотеке у формату JSON (-j) користи се следећа наредба:

```
rabin2 -s -j <filename>
```

Ова команда ће извршити извлачење симбола и приказати резултате у формату *JSON* на стандардном иизлазу. На пример, за ELF датотеку `hello`, која је добијена из `hello.c` програма из претходног поглавља, излаз ће имати велики број симбола који су неопходни за извршавање програма чак и за овако једноставне програме. На пример, секције као што су `.text`, `.data`, `.bss`, `.rodata`

су стандардне за већину ELF датотека. Ове симболе додају компилатор и повезивач па су самим тим и део излаза ове наредбе. Неки од релевантних симбола изгледају овако:


```
{
  "name": ".rodata",
  "flagname": "sym..rodata",
  "realname": ".rodata",
  "ordinal": 18,
  "bind": "LOCAL",
  "size": 0,
  "type": "SECT",
  "vaddr": 8192,
  "paddr": 8192,
  "is_imported": false
},
{
  "name": "hello.c",
  "flagname": "sym.hello.c",
  "realname": "hello.c",
  "ordinal": 36,
  "bind": "LOCAL",
  "size": 0,
  "type": "FILE",
  "vaddr": 0,
  "paddr": 0,
  "is_imported": false
},
{
  "name": "main",
  "flagname": "sym.main",
  "realname": "main",
  "ordinal": 61,
  "bind": "GLOBAL",
  "size": 32,
  "type": "FUNC",
```

```
  "vaddr": 4425,
  "paddr": 4425,
  "is_imported": false
},
{
  "name": "imp.printf",
  "flagname": "sym.imp.printf",
  "realname": "printf",
  "ordinal": 2,
  "bind": "GLOBAL",
  "size": 16,
  "type": "FUNC",
  "vaddr": 4176,
  "paddr": 4176,
  "is_imported": true
}
```


Тумачење излаза:

- `name`: Назив симбола који је дефинисан у извршивој датотеци, а који може да модификује *radare2* у неким случајевима.
- `flagname`: Ознака симбола унутар радног оквира *radare2*.
- `realname`: Прави назив симбола који је оригинално дефинисан у бинарној датотеци.
- `ordinal`: Редни број симбола у табели симбола.
- `bind`: Видљивост и доступност симбола (`LOCAL` за локалне симболе, `GLOBAL` за глобалне).
- `size`: Величина симбола у бајтовима.
- `type`: Тип симбола (нпр. `SECT` за секције, `FUNC` за функције, `FILE` за датотеке).
- `vaddr`: Виртуелна адреса на којој се налази симбол.
- `paddr`: Физичка адреса на којој се налази симбол.

- `is_imported`: Да ли је симбол импортован (`true` или `false`).

Глава 4

Практична реализација система за анализу

У овој глави описана је имплементација алата *ELFAnalyzer* за анализу ELF датотека, који је јавно доступан на сервису *GitHub*[1]. *ELFAnalyzer* користи алате *readelf* и *rabin2*, и из сваке ELF датотеке извлачи податке о симболима, функцијама и променљивим, а затим их групише по софтверским компонентама на основу дефинисаних конфигурација.

При једном покретању, алат *ELFAnalyzer* врши статичку анализу потребне меморије за дати извршиви програм. Ова анализа обухвата величине глобалних променљивих и функција, пружајући увид у то колико меморије програм заузима пре самог извршавања. Уградни системи, који често имају ограничене меморијске ресурсе, могу имати значајну корист од овакве анализе јер омогућава програмерима да ефикасно распореде меморијске ресурсе на основу статичких података доступних у ELF датотеци.

Важно је напоменути да *ELFAnalyzer* не прати динамичку меморију која се алоцира током извршавања програма, нити може да процени колико ће меморије бити заузето на програмском стеку током извршавања. Ови аспекти меморијског заузећа зависе од динамичких услова током рада програма и не могу се анализирати статички пре самог покретања.

Континуираним коришћењем алата могуће је пратити како се меморија софтвера мењала током различитих фаза развоја. Омогућава детаљно праћење меморијског заузећа сваке компоненте кроз време, чувајући податке у бази. На основу ових података, визуализује промене у заузећу меморије, омогућавајући

[1] `https://github.com/sairenaa/master-thesis`

праћење еволуције софтверских компоненти и идентификацију потенцијалне области за оптимизацију. *ELFAnalyzer* такође омогућава преглед детаљне листе симбола за сваку компоненту, пружајући прецизан увид у допринос сваког симбола укупном меморијском заузећу.

4.1 Дизајн и развој система

Пројекат је развијен користећи концепте објектно-оријентисаног програмирања и веб технологија у програмском језику *TypeScript*. Графички кориснички интерфејс, тј. фронтенд је изграђен уз помоћ оквира *Angular*, док је бекенд реализован у окружењу *NodeJS*. За развој десктоп апликације коришћен је оквир *ElectronJS* који омогућава коришћење веб технологија[2]. За анализу ELF датотека коришћени су алати *readelf* и *rabin2*, док се за чување података користи *PostgreSQL* база.

Улазни подаци

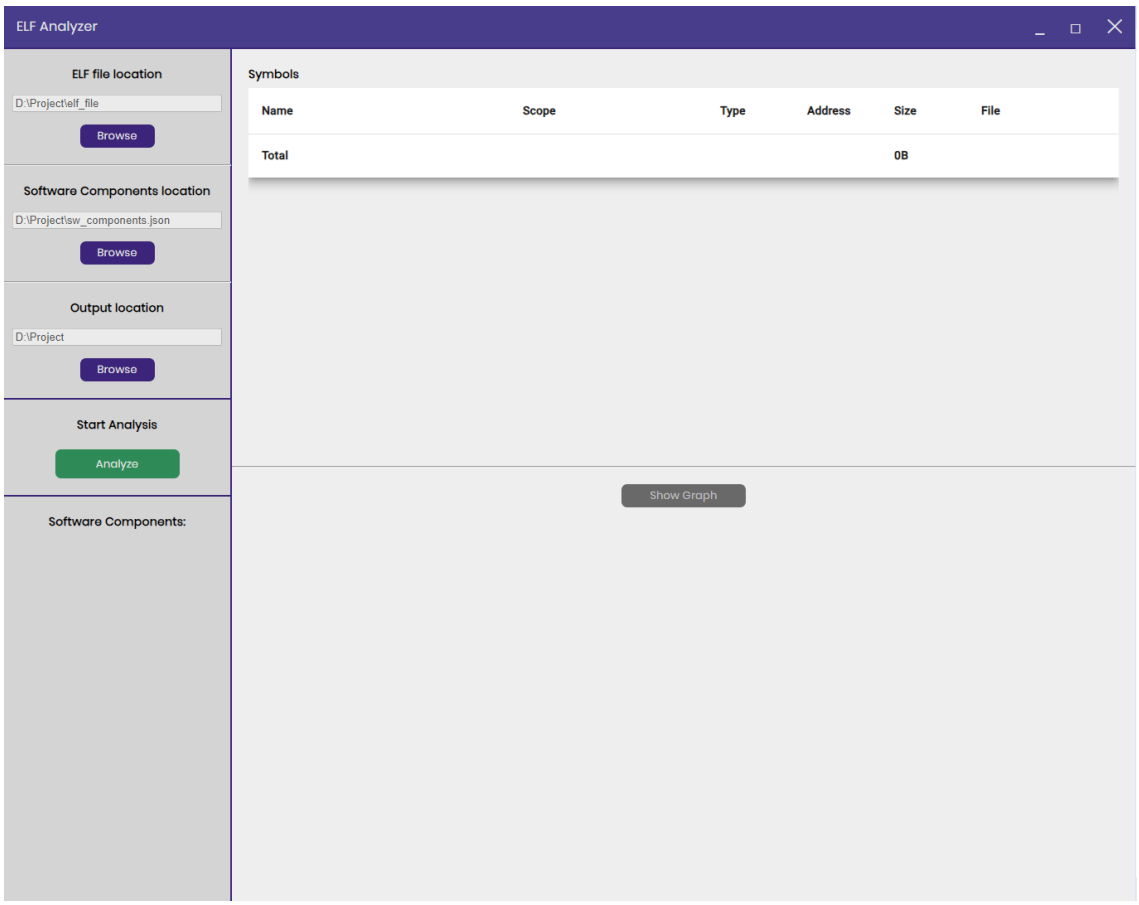
Дефинисање софтверских компоненти од којих се састоји пројекат који се анализира представља почетни корак у овој имплементацији, самим тим, фокус је на анализи сваке компоненте појединачно. Очекивани улаз описа је датотека у формату *JSON* приказана у листингу 4.1. На овај начин дефинише се садржај компоненти, тачније, од којих се датотека састоје.

```
{
"components": [
        {
                "name": "Component1",
                "files": ["file1.c", "file2.c"]
        },
        {
                "name": "ComponentN",
                "files": ["fileN1.c", "fileNm.c"]
        }
        ]
}
```

Листинг 4.1: Софтверске компоненте у формату JSON.

[2] *ElectronJS* има уграђену машину прегледача (енг. *browser engine*) *Chromium* и окружење *NodeJS* у свој бинарни кôд.

Поред описа пројекта у виду софтверских компоненти, очекивани улаз је свакако и ELF датотека. Тачније, очекују се путање до одговарајућих датотека (Слика 4.1).



Слика 4.1: Графички кориснички интерфејс алата *ELFAnalyzer*.

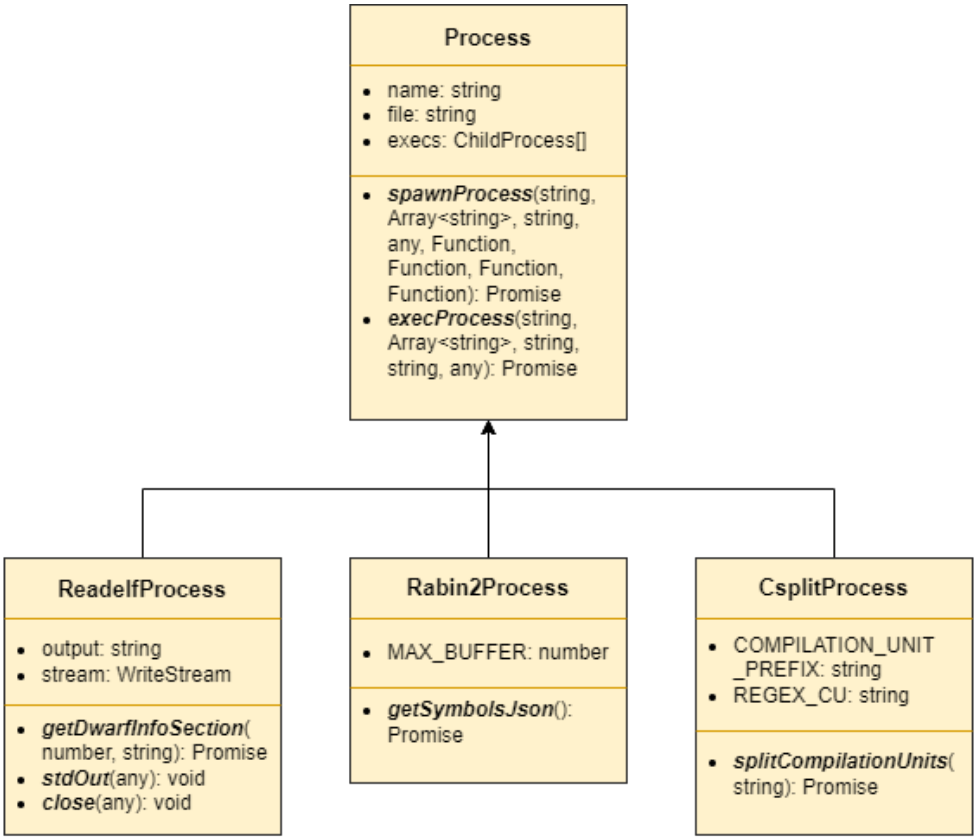
Анализа улазних података

За анализу ELF датотека коришћени су алати *readelf*, *rabin2* и *csplit*, који се извршавају као процеси у оперативном систему са одговарајућим аргументима, помоћу *NodeJS* пакета *child_process*. С обзиром да је пројекат имплементиран на оперативном систему *Windows*, *readelf* и *csplit* алати су преведени у извршиве (*.exe*) датотеке како би се могли покренути и користити за анализу. Алат *csplit* је помоћни алат који се користи да подели излазну датотеку добијену *readelf* наредбом за извлачење информација за дебаговање на јединице компилације[3].

[3]Код великих пројеката, па самим тим и великих ELF датотека, излаз који се добија приликом извлачења информација за дебаговање може бити величине неколико гигабајта,

Овакав приступ коришћења алата, помоћу *child_process* пакета, омогућава аутоматизовану анализу датотека директно из *NodeJS* окружења.

У ову сврху, креирана је базна класа *Process* која служи као основа за све процесе који се користе за анализу (класе које је наслеђују су *ReadelfProcess*, *Rabin2Process* и *CsplitProcess*). У базној класи су дефинисане омотач методе (енг. *wrapper function*) *spawnProcess()* и *execProcess()* које позивају редом одговарајуће методе из пакета *child_process*: *spawn()* и *execFile()* са одговарајућим аргументима (Слика 4.2).



Слика 4.2: УМЛ дијаграм класа.

Класа *ReadelfProcess* се користи за извлачење DWARF информација. Метода *getDwarfInfoSection()* у овој класи позива *spawnProcess()* методу са одговарајућим аргументима, и на тај начин постижемо извршавање следеће наредбе:

```
readelf.exe <elf_file> --debug-dump=info --dwarf-depth=2
```

самим тим парсирање такве датотеке није ефикасно. У ту сврху, уведена је подела на јединице компилације, при чему се парсира свака од њих посебно.

након чега се излаз чува у датотеци *dwarf_ info.txt*.

Класа *Rabin2Process* имплементира методу *getSymbolsJson()* која извршава следећу наредбу позивањем *execProcess()* методе:

```
rabin2.exe -s -j <elf_file>
```

чиме се добијају информације о свим симболима у формату JSON који се чувају у датотеци *symbols.json*.

CsplitProcess класа имплементира методу *splitCompilationUnits()* која позива извршавање *csplit* процеса за поделу на јединице компилације, чиме се добију одговарајуће датотеке за сваку јединицу компилације. Наредба којом се ово постиже изгледа овако:

```
csplit.exe -f <prefix> -k <dwarf_info.txt> /Compilation/ {*}
```

Овом наредбом се спецификује следеће:

- `-f <prefix>`: Префикс који ће бити додат сваком излазу
- `-k`: Чување датотека чак и ако су празне
- `<dwarf_info.txt>`: Путања до улазне датотеке која се разбија
- `/Compilation/`: Шаблон који одређује да ће се датотеке разбити сваки пут када се наиђе на линију која га садржи
- `{*}`: Шаблон који указује да се настави са дељењем све док се не дође до краја датотеке

Класа *Analyzer* је уведена као главна класа која управља целокупним процесом анализе ELF датотека. Ова класа интегрише сва три претходно поменута процеса кроз одговарајуће методе:

- *getDwarfInfo()*: која позива методе *getDwarfInfoSection()* из класе *ReadelfProcess* и *splitCompilationUnits()* из класе *CsplitProcess* тако што их извршава у ланцу обећања[4] (енг. *Promises*) асинхроно.

[4]Обећања су објекти који представљају успешан завршетак или неуспех асинхроне операције и њен резултат.

- *getSymbols()*: која позива методу *getSymbolsJson()* из класе *Rabin2Process* након чега користи помоћне методе за писање и читање из датотеке (*writeFile()* и *readFile()* из помоћне класе *Util*), такође кроз ланац обећања.

Након што су подаци извучени из ELF датотеке помоћу метода *getDwarfInfo()* и *getSymbols()*, класа *Analyzer* наставља са процесом парсирања и мапирања добијених података. Метода *parseCompilationUnits()* је одговорна за парсирање јединица компилације које су креиране у претходним корацима. Ова метода користи *DwarfParser* класу за обраду и структурирање података из јединица компилације, при чему се екстрахују релевантне информације, конкретно, променљиве и функције.

Процес парсирања се одвија тако што се улазне датотеке учитавају и обрађују линију по линију, где се свака линија анализира и укључује у одговарајућу структуру података. На крају парсирања, подаци се чувају у формату *JSON* за даљу употребу и анализу.

Када је парсирање завршено, класа *Analyzer* користи методу *mapSymbolsToFiles()* да повеже симболе добијене из *rabin2* процеса са одговарајућим јединицама компилације на основу адресних опсега. Ово омогућава да се тачно идентификује који симболи припадају којим деловима кода унутар пројекта.

Поред тога, класа *Analyzer* омогућава и иницијализацију софтверских компоненти кроз методу *initSwComponents()*, која користи *SwComponentsParser* за учитавање и обраду *JSON* датотеке која садржи информације о компонентама софтвера. Ове компоненте се затим користе за креирање тренутног стања пројекта (што је представљено класом *ProjectImage*) која визуализује структуру софтвера и односе између различитих делова система.

На крају, метода *generateProjectImage()* интегрише све прикупљене и обрађене податке како би генерисала финалну слику пројекта (објекат класе *ProjectImage*), која се чува у формату *XML* (листинг 4.2):

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <ProjectImage date="2024-08-07T13:30:14.342Z">
3   <SWC name="Component1" size="632">
4     <SYMBOL name="main" filename="main.c"
5         size="632" type="FUNC" address="0x11e9" scope="GLOBAL"/>
6   </SWC>
7   <SWC name="Component2" size="776">
```

```
8       <SYMBOL name="symbol1" filename="complex.c"
9           size="88" type="FUNC" address="0x1461" scope="GLOBAL"/>
10      <SYMBOL name="symbol2" filename="complex.c"
11          size="64" type="OBJ" address="0x15c9" scope="GLOBAL"/>
12      <SYMBOL name="symbol3" filename="complex.c"
13          size="20" type="FUNC" address="0x15b5" scope="GLOBAL"/>
14      <SYMBOL name="symbol4" filename="complex.c"
15          size="46" type="FUNC" address="0x173b" scope="GLOBAL"/>
16      <SYMBOL name="symbol5" filename="complex.c"
17          size="20" type="FUNC" address="0x15a1" scope="GLOBAL"/>
18      <SYMBOL name="symbol6" filename="complex.c"
19          size="103" type="FUNC" address="0x16d4" scope="GLOBAL"/>
20      <SYMBOL name="symbol7" filename="complex.c"
21          size="75" type="FUNC" address="0x163e" scope="GLOBAL"/>
22      <SYMBOL name="symbol8" filename="complex.c"
23          size="75" type="FUNC" address="0x1689" scope="GLOBAL"/>
24      <SYMBOL name="symbol9" filename="complex.c"
25          size="4" type="OBJ" address="0x1609" scope="GLOBAL"/>
26      <SYMBOL name="symbol10" filename="complex.c"
27          size="232" type="FUNC" address="0x14b9" scope="GLOBAL"/>
28    </SWC>
29  </ProjectImage>
```

Листинг 4.2: Пример *XML* структуре објекта класе *ProjectImage*.

Оваква архитектура система омогућава да се алати *readelf*, *rabin2* и *csplit* могу лако заменити неким другим алтернативним алатима који имају исту или сличну сврху, што алату *ELFAnalyzer* даје додатну флексибилност.

4.2 Визуализација података и праћење заузећа меморије кроз време

Током развоја великих софтверских пројеката, посебно у индустријама са строжим ограничењима, попут аутомобилске или авио-индустрије, праћење потрошње меморије је од велике важности. Многи пројекти имају дефинисана меморијска ограничења за компоненте, које је потребно поштовати како би се осигурала стабилност и перформансе система. Континуирано праћење омогућава увид у то како се меморијски ресурси користе и обезбеђује рано откривање потребе за оптимизацијом, чиме се пројекат држи у оквирима задатих мемо-

ријских ограничења. Ово праћење такође помаже тимовима да прате трендове у развоју и да благовремено реагују на промене у потрошњи меморије.

Да би се омогућило дугорочно праћење и анализа заузећа меморије кроз време, генерисани *ProjectImage* објекат, који је сачуван у формату XML у виду датотеке, такође се уписује и у *PostgreSQL* базу података. Чување ових података у бази омогућава да сваки резултат анализе буде доступан за касније поређење и праћење еволуције софтверских компонети у оквиру пројекта.

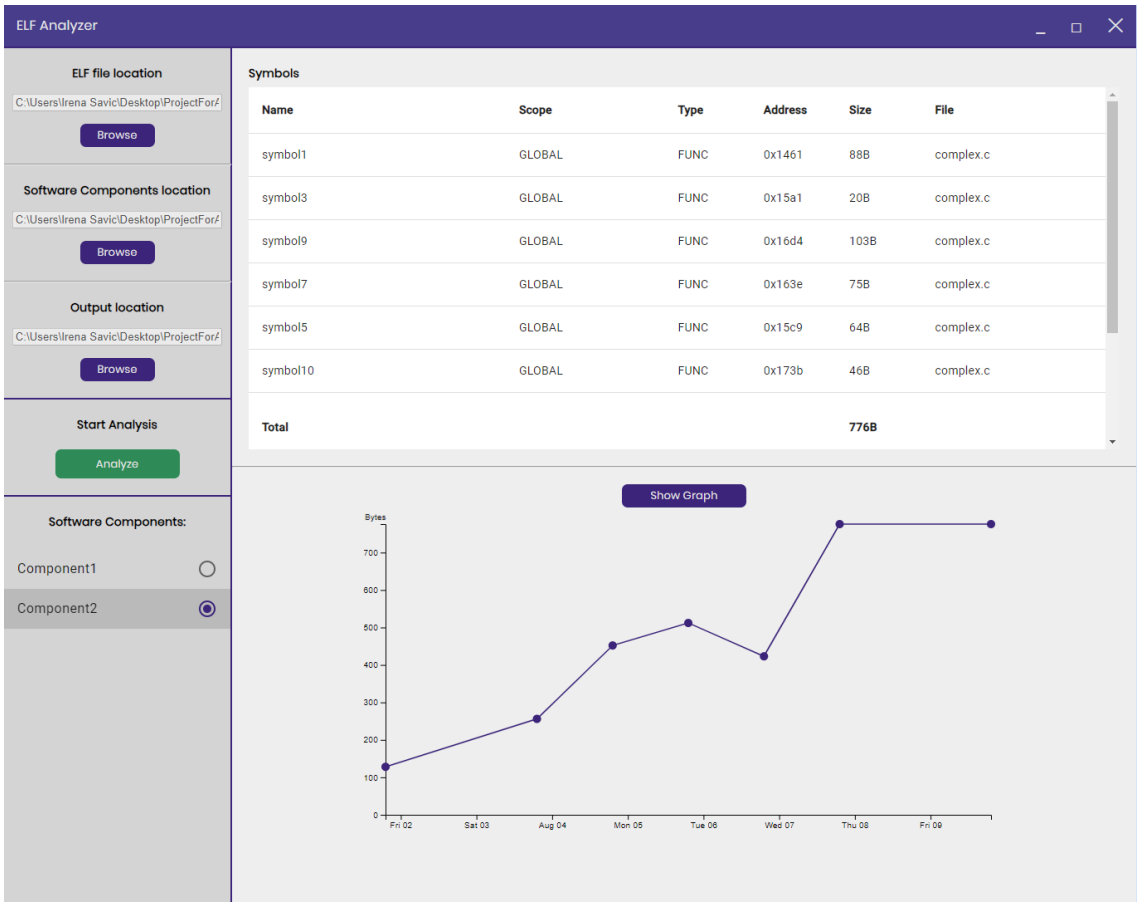
За потребе визуализације података у пројекту, шема базе је дефинисана са две табеле. Табела *elf* садржи основне информације о ELF датотеци. Табела *softwarecomponent* садржи податке о свакој софтверској компоненти — њено име, заузеће у бајтовима, датум када је анализа извршена, и којој ELF датотеци припада.

Унутар класе *Analyzer*, метода *saveProjectImage()* је одговорна за чување резултата анализе у базу података. Ова метода најпре проверава да ли у табели *elf* већ постоји запис за дату ELF датотеку. Ако запис постоји, користи њен идентификатор да повезује нове податке о софтверским компонентама у табелу *softwarecomponent*. Уколико запис не постоји, креира се нови запис за ELF датотеку, а затим се подаци о софтверским компонентама додају у базу. При чему се ово врши приликом сваке анализе пројекта.

Након што се резултати анализе сачувају у бази података, они се користе за визуализацију заузећа меморије током времена. Помоћу *d3.js* библиотеке развијени су графички прикази који омогућавају преглед промене величине одређене софтверске компоненте у бајтовима кроз различите фазе развоја пројекта. Свака анализа додаје нови запис у базу, а ове вредности се приказују на графику, где је оса времена повезана са заузећем меморије у бајтовима.

Поред графика који визуализује заузеће меморије, систем омогућава и преглед детаљне листе симбола изабране компоненте. Листа приказује који су симболи присутни у компоненти, њихове величине и адресе у меморији, што омогућава прецизну анализу доприноса сваког симбола укупном меморијском заузећу.

Следећа слика приказује пример графика за меморијско заузеће једне компоненте кроз време, као и пратећу листу симбола:



Слика 4.3: Табеларни и графички приказ заузећа меморије софтверске компоненте.

Примена алата *ELFAnalyzer* на пројекту *TinyCC*

За потребе демонстрације рада алата *ELFAnalyzer*, коришћен је пројекат *TinyCC*[5] (користи се још и назив *TCC*). *TinyCC* представља компилатор за програмски језик C, познат по својој једноставности и ефикасности.

Да би се превео пројекат *TinyCC* и генерисала ELF датотека, коришћена су упутства која су доступна у документацији самог пројекта. Процес компилације извршен је уз употребу *GCC* компилатора. Прво је покренута конфигурациона скрипта `configure`, која припрема пројекат за компилацију тако што подешава одговарајуће параметре и проверава зависности. Након успешне конфигурације, извршена је компилација пројекта коришћењем наредбе `make`, која преводи све изворне датотеке и генерише коначну ELF извршиву датотеку.

Због природе пројекта *TinyCC* и начина на који је конфигурисан процес компилације, неке изворне датотеке из самог пројекта нису укључене у финалну ELF датотеку која се генерише. Ово се дешава јер процес компилације обрађује само оне датотеке које су неопходне за конкретну платформу на којој се пројекат извршава. Као резултат, DWARF информације, које се користе за анализу, обухватају само датотеке које су заправо укључене у генерисану ELF датотеку. Датотеке које су обухваћене овим процесом се потом групишу у компоненте у складу са њиховом улогом и функционалношћу у оквиру *TinyCC* пројекта. Сходно томе, извршено је груписање у компоненте које представљају логичке целине у оквиру пројекта. На пример, компонента *TCC Core* обухвата кључне датотеке као што су `tcc.c` и `tccgen.c`, док је *Preprocessor* компонента резервисана за датотеку `tccpp.c`, која је одговорна за претпроцесирање кода. На сличан начин су дефинисане и остале компоненте: *Linker*, *Library Support*, *Target Arch Support* и *Debugging Support*.

У листингу 4.3 је дат преглед свих компоненти са припадајућим датотекама, тачније садржај датотеке `components.json` која се користи приликом анализе у оквиру алата *ELFAnalyzer*.

```
1 {
2     "components": [
3         {
4             "name": "TCC Core",
5             "files": ["tcc.c", "tccgen.c", "tccasm.c", "tccrun.c
                ↪ "]
6         },
```

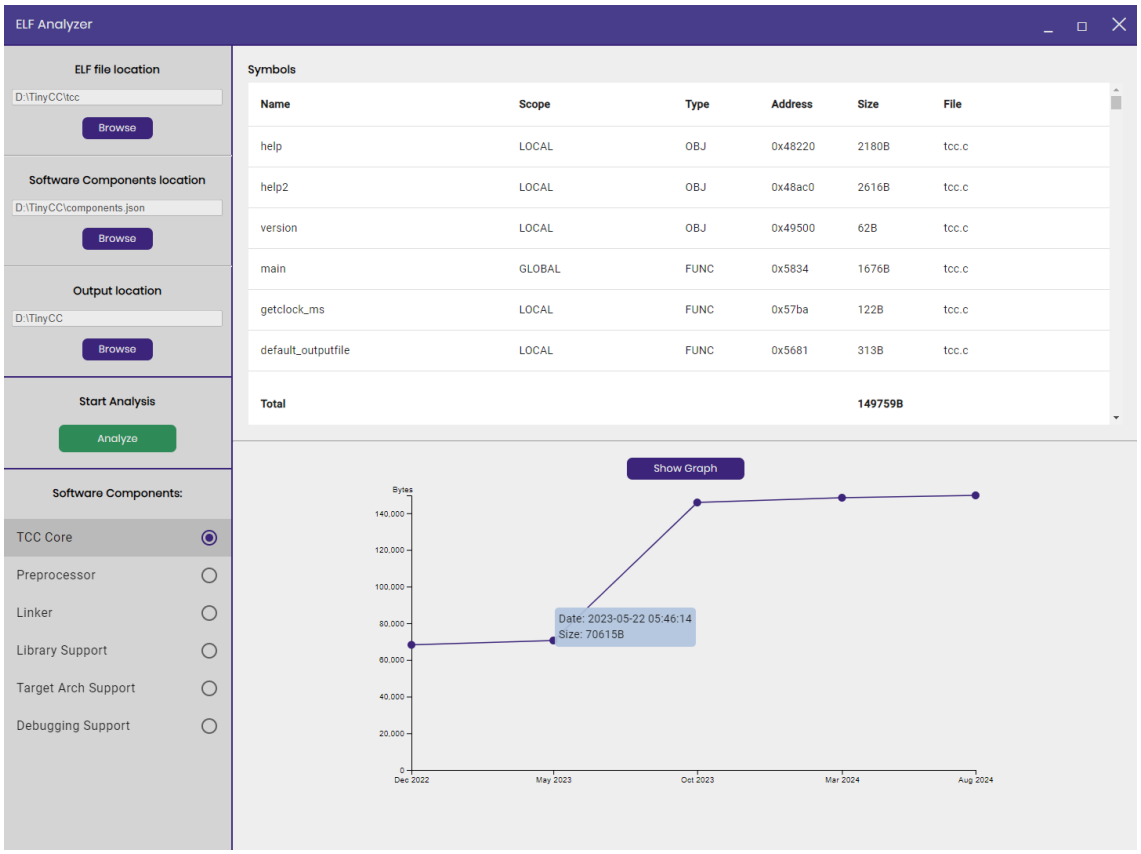
[5] https://github.com/TinyCC/tinycc

```
7           {
8               "name": "Preprocessor",
9               "files": ["tccpp.c"]
10          },
11          {
12              "name": "Linker",
13              "files": ["tccelf.c"]
14          },
15          {
16              "name": "Library Support",
17              "files": ["libtcc.c"]
18          },
19          {
20              "name": "Target Arch Support",
21              "files": ["x86_64-gen.c", "x86_64-link.c", "i386-asm
                  ↪ .c"]
22          },
23          {
24              "name": "Debugging Support",
25              "files": ["tccdbg.c"]
26          }
27      ]
28  }
```

Листинг 4.3: Софтверске компоненте у *TinyCC* пројекту.

Да би се илустровале промене у развоју пројекта *TinyCC*, анализа је спроведена више пута, при чему су превођене различите претходне верзије пројекта на сваких пет месеци. Овај приступ пружа увид у то како су различите модификације кода утицале на величину и структуру компоненти.

За ову сврху, фокус је стављен на компоненту *TCC Core*. На графику који приказује развој ове компоненте кроз време, може се уочити значајан раст у периоду између маја и октобра 2023. године. У мају 2023. године, величина компоненте је била 70615B, док је у октобру 2023. године порасла на 145872B. Овај скок у величини компоненте може се приписати додавању нових функционалности или увођењу значајних измена које су повећале комплексност постојећег кода (Слика 4.4).



Слика 4.4: Графички приказ заузећа меморије софтверске компоненте *TCC Core* (мај 2023).

Иако графици пружају широк преглед укупног раста компоненте, они не откривају конкретне детаље о томе шта је допринело овом расту. Међутим, овакав преглед омогућава да се тачно идентификује период у развоју пројекта који је вредан детаљније анализе, што може помоћи у бољем разумевању утицаја специфичних промена на целокупни пројекат. Детаљнија анализа може се спровести уз помоћ листе симбола коју алат *ELFAnalyzer* генерише за сваку компоненту. Ова листа омогућава увид у специфичне симболе и њихово меморијско заузеће, чиме се може прецизније утврдити које су измене у коду довеле до повећања меморије, било да је реч о изменама постојећих симбола или додавању нових.

Детаљнијом анализом уз помоћ алата *ELFAnalyzer*, утврђено је да је значајан скок у меморији наступио при ажурирању пројекта 6.9.2023. Опис ажурирања при коме је дошло до скока меморије, доступан уз одговарајућу команду `git commit`, указује на избацивање нове верзије компилатора, што сугерише да су додате нове функционалности које су значајно утицале на повећање мемо-

ријског заузећа (приближно дупло у односу на претходну верзију). Из описа промена у том периоду може се закључити да су у пројекту додате функционалности као што су подршка за основне информације за дебаговање и оптимизације у самом компилатору.

Глава 5

Закључак

У овом раду је истражено како се статичка анализа кода може користити за пружање увида у заузеће меморијских ресурса у софтверу. Истраживање је обављено на примеру формата ELF, који се природно намеће због своје широке употребе, као формат који је стандардизован, флексибилан и омогућава интероперабилност између различитих оперативних система и хардверских платформи. Може да подржи различите архитектуре и величине адреса, што га чини универзалним решењем за многе врсте софтвера. Већина *Linux* дистрибуција користи ELF као подразумевани формат за извршиве датотеке и библиотеке. Такође, многи алатни ланци (енг. *toolchains*) као што су *GCC* и *LLVM*, генеришу ELF датотеке.

Коришћени су алати *readelf* и *radare2*, који су се показали као погодни за анализу формата ELF због својих специфичних карактеристика. *readelf*, као део традиционалног *Unix* окружења, пружа поуздане резултате у анализи структура ELF датотека и широко је доступан на различитим платформама, што га чини корисним алатом за почетне кораке у анализи. С друге стране, *radare2* се истиче својом флексибилношћу и могућношћу проширења, што омогућава даљи развој и надоградњу система описаног у овом раду. Овај алат пружа напредне опције за анализу, које могу бити корисне за дубље испитивање кода, као и за потенцијално проширење система у будућности, на пример кроз декомпилирање или детаљнију визуализацију структуре програма.

Примена ове методологије може бити корисна у контексту великих пројеката, као што су они у аутомобилској индустрији, где се често јављају изазови у погледу заузећа меморије. Компаније се у таквим ситуацијама суочавају са избором између улагања у нови хардвер са више меморије или ангажовања ис-

кусних инжењера који би оптимизовали постојећи кôд. Идентификација потенцијалних проблема са меморијом у раним фазама развоја софтвера омогућава да се ови проблеми уоче на време, што може значајно допринети уштеди времена и ресурса. Редовно праћење заузећа меморије током читавог развојног процеса може помоћи у превенцији непланираних проблема са перформансама или трошковима, смањујући потребу за скупим хардверским изменама или каснијим оптимизацијама кода.

Осим постојећих могућности, овај систем нуди простор за даљи развој и проширење. На пример, интеграција са напреднијим алатима или додатним техникама анализе могла би побољшати прецизност и омогућити дубљи увид у заузеће меморије. Поред тога, будући развој система могао би обухватити и праћење динамички алоциране меморије, попут меморије на хипу или програмског стека, чиме би се проширили могућности алата за анализу целокупног меморијског заузећа програма током његовог извршавања. Овакав развој би унапредио тачност анализе и омогућио још бољу оптимизацију програма са сложеним меморијским захтевима. Такође, аутоматизација и интеграција у постојеће *CI/CD* процесе, који подразумевају континуирану интеграцију и континуирану испоруку софтвера, могла би осигурати континуирано праћење меморијских ресурса током читавог животног циклуса софтвера, што би допринело ефикаснијем развоју и одржавању софтвера.

Библиографија

[1] DWARF Standard. Official dwarf standard website. https://dwarfstd.org/.

[2] Michael J. Eager. Introduction to the dwarf debugging format. https://dwarfstd.org/doc/Debugging%20Information%20Format.pdf, April 2012.

[3] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann, 2000.

[4] Linux man page. readelf(1) - linux man page. https://man7.org/linux/man-pages/man1/readelf.1.html.

[5] GNU Project. Gnu binutils. https://www.gnu.org/software/binutils/, 2024.

[6] The DWARF committee. Dwarf 5 standard. https://dwarfstd.org/DWARF5.pdf, February 15 2017.

[7] Cornell University. Executable and object files. https://www.cs.cornell.edu/courses/cs414/2004su/homework/hw2.pdf, 2023.

[8] UNIX System Laboratories. *Executable and Linkable Format (ELF) Specification*, 1995.

[9] Rupert Wood. What to do with a.out. https://gcc.gnu.org/legacy-ml/gcc-help/2002-04/msg00075.html, April 2002. gcc-help (Mailing list).

[10] Sergi Àlvarez. The radare2 book. https://book.rada.re/, 2024.