VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GRAFICKÝ PROHLÍŽEČ A JEDNODUCHÝ EDITOR ELF SOUBORU

GRAPHICAL VIEWER AND SIMPLE EDITOR OF ELF FILE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN OMACHT**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Omacht Martin**
Program: Informační technologie
Název: **Grafický prohlížeč a jednoduchý editor ELF souboru**
Graphical Viewer and Simple Editor of ELF File
Kategorie: Překladače

Zadání:

1. Seznamte se s multiplatformním formátem pro binární soubory ELF - Executable and Linkable Format. Seznamte se s knihovnami pro čtení a manipulaci s ELF.
2. Dle pokynů vedoucího vyberte knihovnu pro práci s ELF a analyzujte její schopnosti ohledně práce s netradičními úpravami ELF souborů.
3. Podle pokynů vedoucího navrhněte způsob zobrazení zadaného ELF souboru z různých pohledů včetně interaktivní navigace odpovídajícími částmi těchto pohledů. Podporujte i editaci ELF souboru a schopnost se vypořádat s ručně upraveným ELF souborem.
4. Navrženou aplikaci implementujte jako multiplatformní (Linux a další vybraná platforma).
5. Vyhodnoťte vytvořenou aplikaci a navrhněte její další vylepšení.

Literatura:

- TOOL INTERFACE STANDARD (TIS). *Executable and Linking Format (ELF) Specification* [online]. Version 1.2. May 1995 [cit. 2014-04-08]. Dostupné na https://refspecs.linuxbase.org/elf/elf.pdf.
- M. Wilding, D. Behman: Self-Service Linux(R): Mastering the Art of Problem Determination. Prentice Hall, 2005.
- Dle pokynů vedoucího a konzultanta

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Konzultant: Hack Robin, RedHatCZ
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 28. října 2018

Abstrakt

Tato práce má za cíl vytvoření aplikace s grafickým uživatelským rozhraním, která poskytuje grafický náhled na vnitřní strukturu binárního souboru ve formátu ELF a také umožňuje jednoduchou editaci. Aplikace zvládá načítat poškozené soubory a podporuje platformy Windows, Linux i macOS. Implementována je v jazyce C++ a grafické rozhraní je vytvořeno pomocí knihovny Qt. Pro zobrazení vnitřní struktury souboru ELF byl navrhnut a implementován vlastní diagram.

Abstract

The goal of this thesis is to create an application with graphical user interface that provides graphical overview of internal structure for binary files in ELF format while also allowing simple editing. The application handles corrupted files and supports Windows and Linux operating systems. It is implemented in C++ with graphical interface created using Qt library. To picture the internal structure of the given ELF file, a custom diagram was designed and implemented.

Klíčová slova

ELF, Executable and Linking Format, Qt, GUI, grafický prohlížeč, grafický editor, poškozený soubor, binární soubor, multiplatformní aplikace

Keywords

ELF, Executable and Linking Format, Qt, GUI, graphical viewer, graphical editor, corrupted file, binary file, multiplatform application

Citace

OMACHT, Martin. *Grafický prohlížeč a jednoduchý editor ELF souboru.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Grafický prohlížeč a jednoduchý editor ELF souboru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Další informace mi poskytl pan Robin Hack. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Omacht
15. května 2019

Poděkování

Rád bych poděkoval mému vedoucímu práce panu Ing. Zbyňku Křivkovi, Ph.D. za nápady, věcné připomínky a poskytnutou pomoc při řešení této práce. Dále taky panu Robinovi Hackovi za návrhy a odbornou pomoc.

Obsah

Kapitola 1

Úvod

ELF je objektový formát binárních souborů používaný pro sestavování a spouštění programů. Je adaptován na mnoha operačních systémech založených na systému Unix. Tyto soubory jsou většinou generovány překladači zdrojových kódů. Jedná se například o sdílené knihovny s příponou `.so` nebo objektové soubory s příponou `.o`. Linker následně tyto soubory sestaví do spustitelného souboru.

Běžný programátor s těmito soubory sice přijde do styku, nicméně nemusí znát jejich obsah ani je nijak upravovat. Pokud však kupříkladu vyvíjí program, který tyto soubory generuje nebo s nimi jinak pracuje, potřebuje studovat jejich vnitřní strukturu, soubor manuálně upravovat nebo zjišťovat, zda není poškozený a případně, jak je poškozený.

Pro účely zobrazování obsahu ELF souboru existuje řada nástrojů jako třeba readelf, objdump nebo HT Editor. Některé podporují i editaci, ale žádný z nich neumí graficky zobrazit vnitřní strukturu souboru nebo analyzovat chyby.

Cílem této práce je navrhnout a implementovat právě takovou aplikaci, která by uživateli graficky zobrazila vnitřní strukturu souboru ve formátu ELF, poskytla jednoduchý editor pro upravování obsahu a taky informovala uživatele o nalezených chybách v souboru. To vše by mělo mít přehledné grafické uživatelské rozhraní.

V následujících kapitolách bude přiblížen vývoj této aplikace. Jako první je představen formát ELF v kapitole 2, kde se čtenář seznámí s jeho strukturou a vlastnostmi. V kapitole 3 jsou analyzovány požadavky na aplikaci, zhodnoceny existující řešení a nakonec je představen návrh grafického uživatelského rozhraní vytvořeného na základě analýzy požadavků. Kapitola 4 se věnuje výběru vhodných knihoven pro implementaci aplikace. Vybírá se knihovna pro čtení a manipulaci ELF souborů, vytváření grafických uživatelských rozhraní a pro vykreslování diagramů. V předposlední kapitole 5 je popsána implementace samotné aplikace a problémy řešené během vývoje. Nejdříve je čtenář seznámen s použitými technologiemi, poté je představena architektura aplikace. Dále je přiblížena implementace vlastní knihovny libelf pro načítání a ukládání ELF souborů. Následuje řešení grafického uživatelského rozhraní, v předposlední sekci je rozebráno vykreslování diagramu struktury ELF souboru a nakonec jsou popsány použité metody testování aplikace. Závěrečné shrnutí této práce se nachází v poslední kapitole 6.

Kapitola 2

Formát ELF

Formát ELF [7] (Executable and Linking Format) byl původně vyvinut a publikován společností UNIX System Laboratories (USL) jako část binárního aplikačního rozhraní ABI (Application Binary Interface). Je to objektový formát souboru, který se používá při linkování programu (sestavování) a při spouštění programu.

Jsou tři hlavní typy objektových souborů [7]:

- *Objektový soubor* (angl. relocatable file), který obsahuje kód a data pro linkování s dalšími objektovými soubory pro vytvoření spustitelného souboru nebo pro vytvoření sdíleného objektového souboru. Většinou má příponu `.o`.
- *Spustitelný soubor* (angl. executable file), obsahuje program vhodný pro provádění přímo na procesoru. Většinou nemá příponu.
- *Sdílený objektový soubor* (angl. shared object file), obsahuje kód a data vhodné pro linkování buď s jinými relokačními nebo sdílenými objektovými soubory a vytvoření dalšího objektového souboru a nebo linkování se spustitelným souborem a jinými objektovými soubory. Jedná se o sdílené knihovny s příponou `.so`.

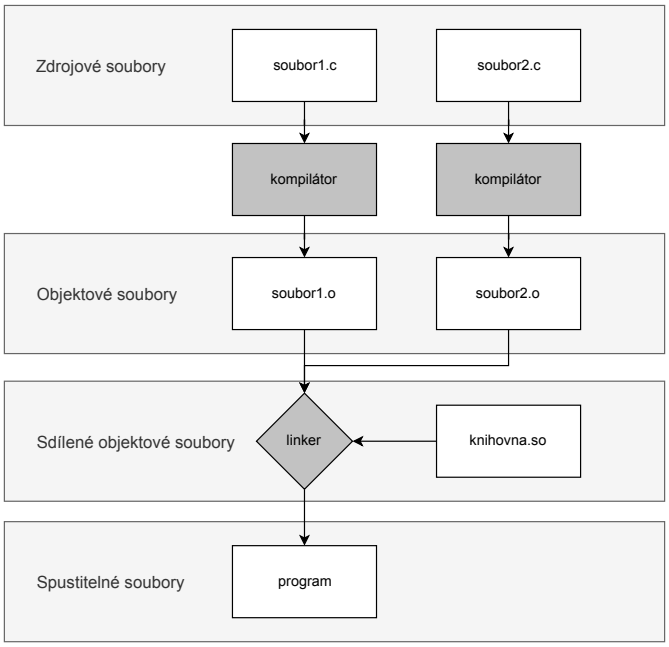
Vztah jednotlivých objektových souborů při kompilaci programu lze vidět na obrázku 2.1.

Formát poskytuje dva pohledy na obsah souboru v závislosti na aktivitě. Při linkování se obsah čte po sekcích a při spouštění programu po segmentech. V rámci jednoho segmentu se může vyskytovat více sekcí. Ukázka struktury ELF souboru z obou pohledů je na obrázku 2.2.

2.1 Hlavička ELF souboru

Soubor ELF začíná hlavičkou, která obsahuje informace o typu souboru a organizaci jeho obsahu.

Jak uvádí specifikace [7], hlavička začíná identifikační položkou `e_ident`, která obsahuje informace o způsobu interpretace obsahu souboru. Hodnoty této položky jsou čteny po bajtech, aby se zajistila stejná interpretace hodnot na všech architekturách (LSB/MSB, 32/64bit,...). První čtyři bajty obsahují tzv. magické číslo identifikující formát souboru jako ELF. Hodnota prvního bajtu musí být `0x7F` a hodnota zbylých tří musí být ASCII hodnota znaků `'E'`, `'L'` a `'F'`. Pátý bajt určuje třídu souboru, což znamená, zda se jedná o 32bitovou nebo 64bitovou architekturu (v budoucnu může být rozšířeno i na jiné architektury). Další bajt specifikuje kódování základních datových typů v souboru. Kódování může být buď



Obrázek 2.1: Vztah objektových souborů při kompilaci programu

little-endian nebo *big-endian*, obojí ve dvojkovém doplňku. Dále následuje už jenom verze hlavičky ELF souboru, která by měla být vždy 1, a několik rezervovaných bajtů.

Další položkou hlavičky je typ objektového souboru. Tím může být již zmiňovaný objektový soubor, spustitelný soubor nebo sdílený objektový soubor, ale také i tzv. *core dump* nebo typy závislé na procesoru.

Následuje položka specifikující potřebnou architekturu pro daný soubor. Hodnoty mohou označovat například architektury Intel (386), Motorola 68000, Intel 80860, ARM a další.

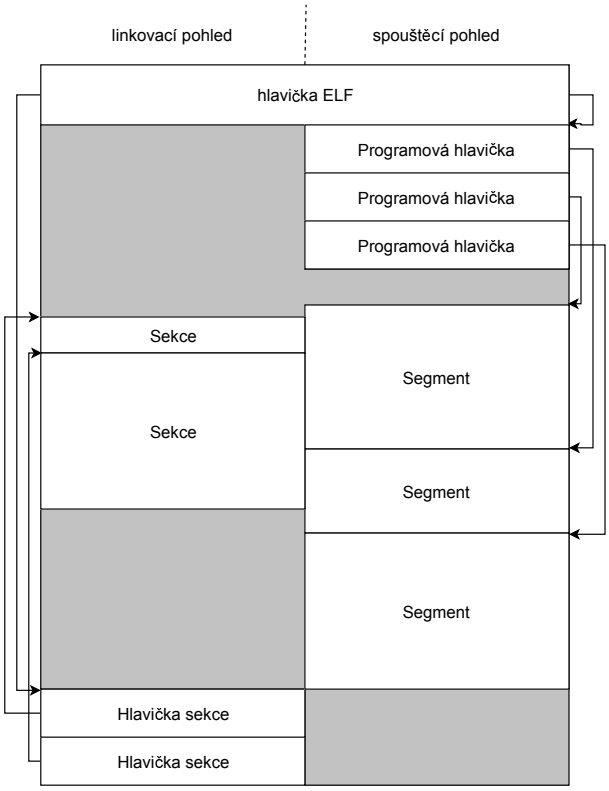
Čtvrtou položkou je verze objektového souboru. Hodnota 1 značí originální formát souboru. Různá rozšíření pak zavádí vlastní verze s vyššími hodnotami.

Vstupní bod programu určuje položka `e_entry`, jejíž hodnota je virtuální adresa, které systém při spuštění procesu předá kontrolu. Pokud soubor nemá žádný vstupní bod, hodnota je nulová.

Položka `e_phoff` značí relativní pozici tabulky programových hlaviček (více viz 2.3) vůči začátku souboru (v bajtech). Pokud je hodnota 0, soubor neobsahuje žádné programové hlavičky. Společně s položkami `e_phentsize`, která určuje velikost jednoho záznamu tabulky, a `e_phnum`, obsahující počet záznamů tabulky, definuje pozici, velikost a jednotlivé záznamy tabulky programových hlaviček. Pro lepší představu je význam těchto položek naznačen na obrázku 2.3.

Tabulka hlaviček sekcí (více viz 2.2) je definována stejným způsobem, akorát se používají položky `e_shoff`, `e_shentsize` a `e_shnum`.

Jako poslední je v hlavičce položka `e_shstrndx` určující index záznamu v tabulce hlaviček sekcí, který obsahuje odkaz na sekci s tabulkou řetězců. Jedná se o posloupnost řetězců počínající i končící bajtem s hodnotou 0. Stejně tak i samotné řetězce jsou ukončeny binární nulou, jako například v jazyce C, a slouží jako názvy sekcí a symbolů v souboru. Odkazuje se na ně pomocí indexu bajtu v této tabulce (nemusí se odkazovat na začátek řetězce, lze



Obrázek 2.2: Ukázka možné struktury ELF souboru z obou pohledů.

tak odkazovat na podřetězec), výsledný název je pak posloupnost znaků od daného indexu do binární nuly.

2.2 Sekce

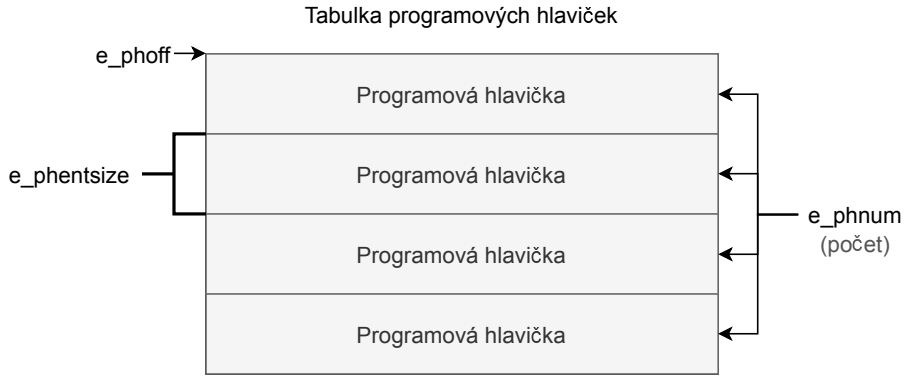
Při linkování se informace v souboru čtou po sekcích. Jednotlivé sekce obsahují buď data, která se mají nahrát do paměti při spuštění programu, nebo formátované metadata o ostatních sekcích. Tato metadata používá pouze linkovací program a patří mezi ně například tabulka symbolů, tabulka řetězců a podobně.

Jednotlivé sekce jsou identifikovány tabulkou hlaviček sekcí. Hlavička sekce obsahuje jméno sekce (index v tabulce řetězců), typ sekce, různé příznaky, adresu, na kterou se sekce načte v případě spuštění (pokud má být načtena, jinak 0), pozici sekce vůči začátku souboru (v bajtech), velikost sekce v bajtech a několik dalších položek závisejících na typu sekce [7].

Příznaky sekcí určují například, jestli sekce obsahuje zapisovatelná data, zda zabírá paměť při vykonávání programu, jestli obsahuje strojové instrukce a jiné vlastnosti závisející na procesoru.

Sekce musí splňovat několik podmínek [7]:

- Každá sekce má právě jednu hlavičku, ale každá hlavička nemusí mít sekci.
- Každá sekce okupuje jednu souvislou sekvenci bajtů v souboru (může být i nulový počet bajtů).
- Sekce se nesmí překrývat. Žádný bajt v souboru nesmí náležet do více sekcí.



Obrázek 2.3: Význam položek definující tabulku programových hlaviček.

- Soubor může mít neaktivní oblasti. Všechny hlavičky ani sekce nemusí dohromady zahrnovat všechny bajty v souboru. Data v neaktivní oblasti nejsou specifikována.

Speciální sekce

Podle specifikace [7] formát ELF definuje několik speciálních sekcí. Názvy těchto sekcí začínající prefixem `'.'` (tečka). Pokud tyto sekce aplikacím nevyhovují, mohou si definovat vlastní sekce, které by pak neměly mít prefix `'.'`, aby se zabránilo kolizím se speciálními sekcemi.

Stručný přehled speciálních sekcí a jejich obsahu je uveden v tabulce 2.1.

Název sekce	Obsah sekce
`.bss`	neinicializovaná data programu, při spuštění inicializována na samé nuly
`.data`, `.data1`	inicializovaná data programu.
`.debug`	ladící informace
`.dynamic`	informace pro dynamické linkování
`.hash`	hašovací tabulka symbolů
`.line`	informace o číslech řádků pro ladění, popisuje vztah mezi strojovým kódem a zdrojovým kódem
`.rodata`, `.rodata1`	data určená jenom pro čtení
`.shstrtab`	názvy sekcí
`.strtab`	řetězce, většinou reprezentující jména záznamů v tabulce symbolů
`.symtab`	tabulka symbolů
`.text`	instrukce programu

Tabulka 2.1: Přehled některých speciálních sekcí a jejich obsah [7].

2.3 Segmenty

Segmenty se používají při spouštění programu. Říkají operačnímu systému, na jakou adresu se má segment nahrát do virtuální paměti a jaké oprávnění tento segment má mít (zápis, čtení, spuštění). V rámci segmentu se může nacházet libovolný počet sekcí.

Tabulka programových hlaviček určuje pozici, velikost a další vlastnosti segmentů, jako například typ, virtuální adresu, na kterou se načte segment do paměti, velikost segmentu v paměti při spuštění a příznaky určující oprávnění segmentu.

Typ segmentu určuje, jak se má interpretovat jeho obsah. Kupříkladu typ `PT_LOAD` specifikuje, že se jedná o data, která mají být nahrána do paměti na adresu určenou v programové hlavičce segmentu. Typ `PT_DYNAMIC` zase určuje, že segment obsahuje informace k dynamickému linkování.

Velikost segmentu v paměti může být větší než jeho velikost v souboru, v takovém případě se přebytečné paměťové místo na konci segmentu naplní nulami.

Kapitola 3

Analýza požadavků a návrh aplikace

Kapitola se zabývá analýzou požadavků vyplývajících ze zadání, následným zhodnocením existujících řešení v oblasti zobrazování a editace ELF souborů a nakonec je představen návrh grafického uživatelského rozhraní výsledné aplikace včetně jeho realizace.

3.1 Požadavky na aplikaci

Cílem této bakalářské práce je implementovat aplikaci, která umožní grafické zobrazení ELF souboru s jednoduchým editorem. Aplikace musí podporovat minimálně Linux a jednu další vybranou platformu. Jelikož nevlastním zařízení s operačním systémem macOS, vybral jsem si jako druhou platformu Windows.

Dalším požadavkem je, že aplikace musí být schopna se vypořádat s netradičně upraveným (poškozeným) ELF souborem. Při načtení souboru se tedy musí zkontrolovat, jestli hodnoty v hlavičkách dávají smysl. Například, pokud odkaz na segment v programové hlavičce ukazuje na pozici, která je za koncem souboru, měla by aplikace uživatele upozornit na tuto skutečnost, tento segment přeskočit a pokračovat dál v načítání souboru.

Požadavky na zobrazení

Jak už ze zadání vyplývá, soubor ELF má být zobrazen graficky. To bude uživateli sloužit pro rychlý přehled o struktuře souboru a výskytu chyb v něm. Grafický náhled by měl zobrazovat soubor z pohledu linkování (sekce) i z pohledu spouštění (segmenty).

Nejlepší způsob jak takto zobrazit tento formát, je diagram jeho struktury, který znázorní propojení jednotlivých hlaviček/sekcí/segmentů a jejich pozici v souboru. Diagram by měl také obsahovat základní informace o souboru, jako například typ objektového souboru, cílová architektura, třída (32/64bit), kódování (little/big endian).

Požadavky na editaci

Aplikace by měla podporovat i editaci souboru ELF. Upravit by měly jít nejlépe všechny položky v hlavičkách. Bylo by dobré uživateli zobrazovat známé hodnoty u položek, kde to dává smysl. Uživatelské rozhraní by však nemělo omezovat v zadání neplatných hodnot, protože jednak je formát ELF rozšiřitelný a aplikace nemusí znát všechny platné hodnoty, ale také je uživatel může chtít zadat záměrně špatně. Některé hodnoty mohou nabýt smyslu až

při změně jiných hodnot, proto je vhodné uživatele pouze přiměřeně upozornit na nevalidní hodnoty a nebránit mu v další editaci souboru.

3.2 Existující řešení

Několik prohlížečů ELF souborů s grafickým uživatelským rozhraním již existuje, avšak neposkytují přehledné grafické zobrazení struktury souboru. Většinou také neumožňují editaci a poškozené soubory často neumí načíst nebo dokonce způsobí pád programu.

K dispozici jsou také nástroje příkazové řádky. Až na HT Editor tyto nástroje slouží jenom pro zobrazení obsahu souboru a nepodporují editaci.

T34 (ELF Viewer)

Jednoduchý prohlížeč ELF souborů T34[1] je dostupný pro operační systémy Linux a Windows. Nabízí pouze jednoduché uživatelské rozhraní s tabulkovým výpisem informací o souboru. Mimo výpis základních informací z hlavičky je možné zobrazit i hlavičky sekcí a programové hlavičky. Editace hodnot však není možná.

AnyELF

AnyELF[2] je plugin pro správce souboru Total Commander[3]. Pro čtení ELF souborů používá nástroj `dump` knihovny ELFIO, jehož výstup pak zobrazuje čistě textově. Zobrazení je tedy formátováno stejně jako v příkazové řádce a nepodporuje editaci hodnot. V textu umožňuje pouze hledat nebo z něj kopírovat.

ELF Parser

Další aplikací, která poskytuje grafické uživatelské rozhraní pro zobrazování ELF souboru, je ELF Parser[4]. Primární zaměření této aplikace je sice statická analýza souboru za účelem odhalení škodlivého softwaru, ale poskytuje také zobrazení obsahu hlaviček a některých sekcí a segmentů. Neumožňuje však editaci souboru a nepodporuje všechny typy ELF souborů. Dostupná je pro platformy Linux, Windows i macOS.

HT Editor

HT Editor[5] je propracovaný nástroj pro prohlížení, analýzu, ale i editování binárních souborů [8]. Podporuje nejenom formát ELF, nýbrž i další formáty, například COFF (Common Object File Format), LE (Linear Executables), PE (Portable Executable), CLASS (Java class file) a další [8]. Nabízí pokročilé vyhledávání, analýzu kódu a dat, zobrazení různých struktur souboru i instrukcí obsažených v sekcích a segmentech pomocí disassembleru. Při vyhledávání lze použít výrazy s vestavěnými funkcemi i matematickými operacemi, regulární výrazy nebo obyčejný text [8]. Výrazy lze použít i při blokových operacích, které slouží k hromadným modifikacím celých bloků souboru [8].

[1] https://sourceforge.net/projects/elfviewer/
[2] https://totalcmd.net/plugring/AnyELF.html
[3] https://www.ghisler.com/
[4] http://elfparser.com/
[5] http://hte.sourceforge.net

V případě porušení souboru například neplatným odkazem na tabulku programových hlaviček, program soubor sice načte, ale zobrazí uživateli obecnou chybu a nerozpozná že se jedná o formát ELF, tudíž nenabídne zobrazení jednotlivých struktur souboru. Neporadí si tedy moc dobře s poškozenými soubory.

Nástroj nemá grafické uživatelské rozhraní, ale pracuje pouze s příkazovou řádkou (rozhraní je však komplexní a propracované). Podporuje platformy Windows i Linux.

readelf

Program readelf je součástí GNU binutils[6] a slouží ke zobrazování informací o ELF souborech. Uživatelským rozhraním je pouze příkazová řádka. Podporuje 32bitové i 64bitové soubory a dokonce i archívy obsahující soubory ELF [2]. K dispozici je velké množství parametrů pro nastavení výpisů různých informací ze souboru, včetně možnosti dekomprese nebo relokace sekcí před výpisem.

Při načítání poškozeného souboru readelf vypíše chybu čtení souboru, ale neinformuje uživatele z jakého důvodu došlo k dané chybě. Neporušené části souboru však stále dokáže přečíst na rozdíl od jiných programů.

objdump

Nástroj objdump je podobný programu readelf. Je také součástí GNU binutils a také je jeho rozhraním příkazová řádka. Oproti readelf však umí načítat více různých formátů objektových souborů. Z tohoto důvodu však neumí vypsat všechny informace specifické pro formát ELF. Nástroj objdump poskytuje i disassembler pro velké množství architektur, kterým lze zobrazovat obsah souboru.

Poškozené soubory objdump příliš dobře nezvládá. Pokud je v souboru chyba, program pouze vypíše generickou chybu při jakémkoliv pokusu o načtení souboru a nezobrazí další informace ani o chybě ani o částech souboru, které nejsou poškozené a lze je přečíst.

3.3 Návrh grafického uživatelského rozhraní

Při návrhu grafického uživatelského rozhraní je potřeba brát v potaz požadavky na funkčnost aplikace. Jak již bylo zmíněno v sekci 3.1, je potřeba graficky zobrazit soubor ve formátu ELF a také umožnit jeho editaci. Aby byla zachována přehlednost grafického zobrazení, je aplikace rozdělena na dva pohledy: zobrazení a editace.

Základní rozložení aplikace je tvořeno klasickým hlavním menu v záhlaví, které obsahuje mimo obvyklé položky i tlačítka na pravé straně pro přepínání mezi pohledy na soubor zmíněné v předchozím odstavci.

Pod menu je umístěn panel se záložkami pro navigaci mezi právě otevřenými soubory. Pokud je obsah souboru v aplikaci změněn a úpravy nejsou uloženy, zobrazí se u jména souboru hvězdička. Pokud soubor obsahuje chyby, změní se barva jeho názvu na červenou.

Obsah panelu se záložkami je závislý na vybraném pohledu. V případě zobrazovacího pohledu zde bude diagram struktury souboru a při editačním pohledu podrobnější výpis všech položek s možností jejich editace. Jednotlivé pohledy jsou popsané níže.

Na spodní hraně okna se pak nachází stavový řádek, informující uživatele o aktuálním stavu aplikace. Obsahovat může například informace o stavu načítání souboru, o počtu nalzených chyb ve formátu, o jeho velikost atp.

[6] https://www.gnu.org/software/binutils/

Zobrazovací pohled

Hlavním prvkem tohoto pohledu je diagram struktury. Formát ELF v hlavičce obsahuje odkaz na tabulku hlaviček sekcí a tabulku programových hlaviček. Záznamy jednotlivých položek pak odkazují na sekce/segmenty. Jedná se tedy o stromovou hierarchii. Sekce a segmenty se mohou v souboru překrývat[7], jedná se totiž o dva různé pohledy na soubor (viz popis formátu ELF v kapitole 2). Tyto pohledy je také potřeba znázornit.

Původní myšlenkou bylo rozdělit pohled na dva další pohledy umístěné vedle sebe. Jeden by zobrazoval diagram z pohledu sekcí a druhý z pohledu segmentů. Pro tento účel byly navrhnuty dva typy diagramů. První z nich je vidět na obrázku 3.1b. Jeho výhodou je, že by zobrazoval hlavičky a sekce/segmenty tak, jak se nachází v souboru. Nevýhodou může být nepřehlednost při větším počtu různě zpřeházených sekcí/segmentů. Druhý typ diagramu na obrázku 3.1a touto nevýhodou netrpí, protože nereflektuje umístění bloků v souboru. Avšak kvůli této vlastnosti není na pohled jasné, jak spolu sekce a segmenty souvisí, tzn. které sekce se nachází ve kterých segmentech. U prvního typu je toto alespoň přibližně možné.

Lepší je však zobrazit oba pohledy v jednom diagramu, aby uživatel viděl, jak spolu souvisí sekce a segmenty. Za tímto účelem vznikl ještě jeden návrh 3.1c založený na blokovém diagramu z obrázku 3.1b. Stal se nakonec finálním, protože nejlépe zobrazuje strukturu ELF souboru. Uživatel díky němu vidí, kde se v souboru nachází jednotlivé hlavičky, sekce a segmenty a jak jsou na sebe odkazovány. Jelikož však diagram znázorňuje strukturu tak, jak se nachází v souboru, vedou velké soubory k tomu, že je pak velmi vysoký, aby bylo možné všechno zobrazit.

Celý pohled 3.2 se tak sestává z bočního panelu, kde jsou vypsány základní informace o ELF souboru (jeho hlavička), a z diagramu, který zobrazuje strukturu. Protože se diagram většinou nevejde do pohledu celý, musí podporovat posouvání. Ze stejného důvodu je vhodné také implementovat odkazy, které po kliknutí přesunou uživatele na odkazovaný blok. To umožní rychlejší orientaci v diagramu. Odkazů mezi jeho bloky může být při větším počtu sekcí nebo segmentů hodně a mohlo by to vést k nepřehlednosti šipek znázorňující tyto odkazy. Toto lze vyřešit zobrazováním šipek vedoucím do bloku nebo z něj, pouze pokud je na něm umístěn kurzor.

Editační pohled

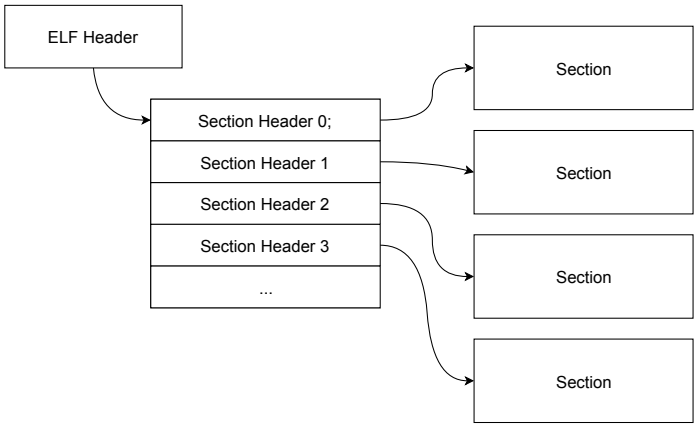
Návrh editačního pohledu je na obrázku 3.3. Zobrazuje hodnoty položek v jednotlivých hlavičkách souboru a umožňuje jejich editaci.

V prvním sloupci zleva je tabulka s položkami ELF hlavičky. V levém sloupci tabulky je název položky a v pravém je editovatelná hodnota spolu s významem této hodnoty (pokud je známý).

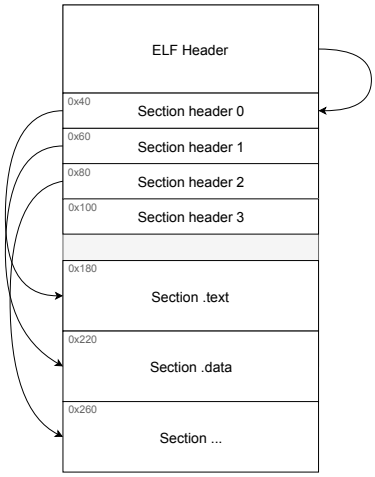
V druhém sloupci jsou hlavičky sekcí a segmentů. Aby v pohledu nebylo příliš mnoho sloupců, jsou tyto hlavičky seskupené do jednoho sloupce, v jehož záhlaví lze mezi nimi přepínat. Jelikož hlaviček může být velké množství, jsou tabulky pro jejich editaci pro větší přehlednost a rychlejší navigaci umístěné v rozklikávacích panelech.

V posledním sloupci se nachází sekce a segmenty přepínatelné stejně jako hlavičky v předchozím sloupci. Tento sloupec však ve výsledné aplikaci nebyl z časových důvodů implementován. Místo tabulek s editačními poli je jejich obsah zobrazen po jednotlivých bajtech, podobně jako je zobrazuje linuxový nástroj `hexdump` s přepínačem `-C`. Na začátku

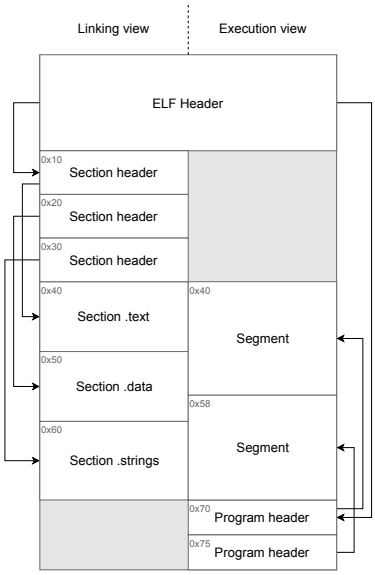
[7] Jednotlivé sekce se mezi sebou nemohou překrývat, pouze sekce se segmentem.



(a) Návrh uzlového diagramu.



(b) Návrh blokového diagramu.



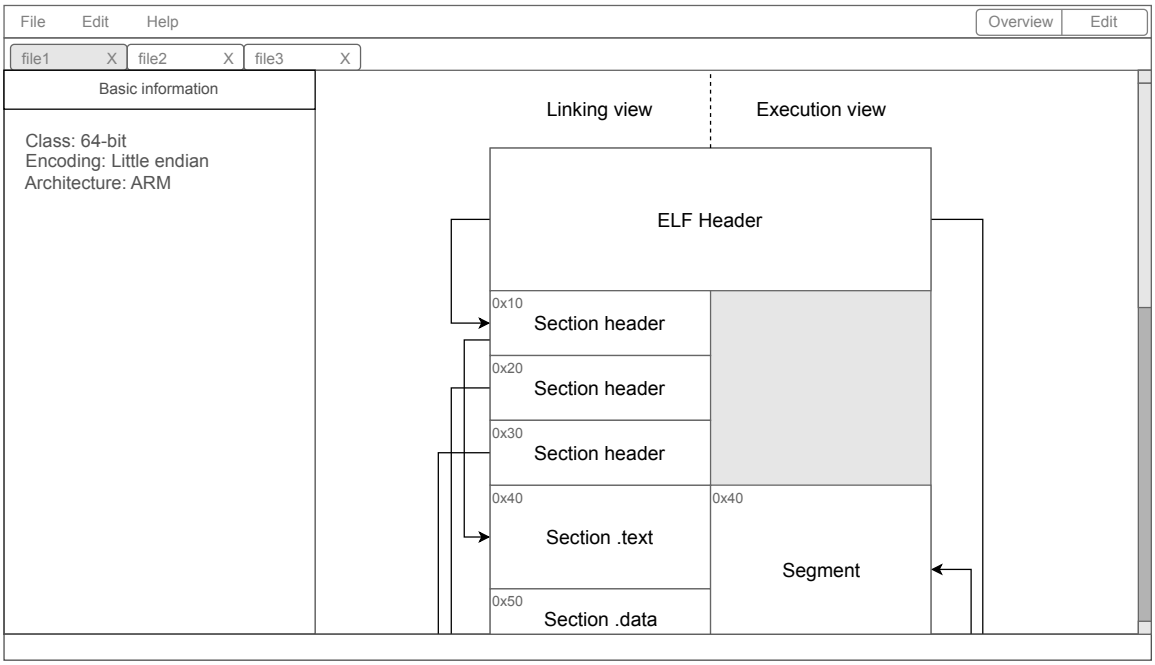
(c) Finální návrh diagramu.

Obrázek 3.1: Návrhy diagramů pro zobrazování struktury ELF souboru.

řádku je adresa prvního bajtu, následuje 16 bajtů v hexadecimální soustavě, které se dají editovat, a na konci řádku jsou tyto bajty převedené na znaky podle tabulky ASCII. Pokud znak není tisknutelný, je místo něj zobrazený znak tečka. Takto jsou zobrazeny všechny bajty v sekci/segmentu.

Výsledné grafické uživatelské rozhraní

Při implementaci se uživatelské rozhraní oproti návrhu mírně změnilo. Jak je vidno na obrázku 3.4, v diagramu jsou místo dvou sloupců čtyři, jelikož segmenty mohou překrývat hlavičky, což mělo za následek nepřehlednost diagramu, zvláště pokud chtěl uživatel kliknout na odkaz v hlavičce. Segmenty se sice stále mohou překrývat, ale to už nepůsobí takový problém jako překrytí hlavičky. Přepínat mezi překrytými segmenty je možné najetím kurzoru myši na programovou hlavičku, což přenese odkazovaný segment do popředí. Stejná funkčnost je implementována i pro sekce a jejich hlavičky. Ty se sice podle standardu



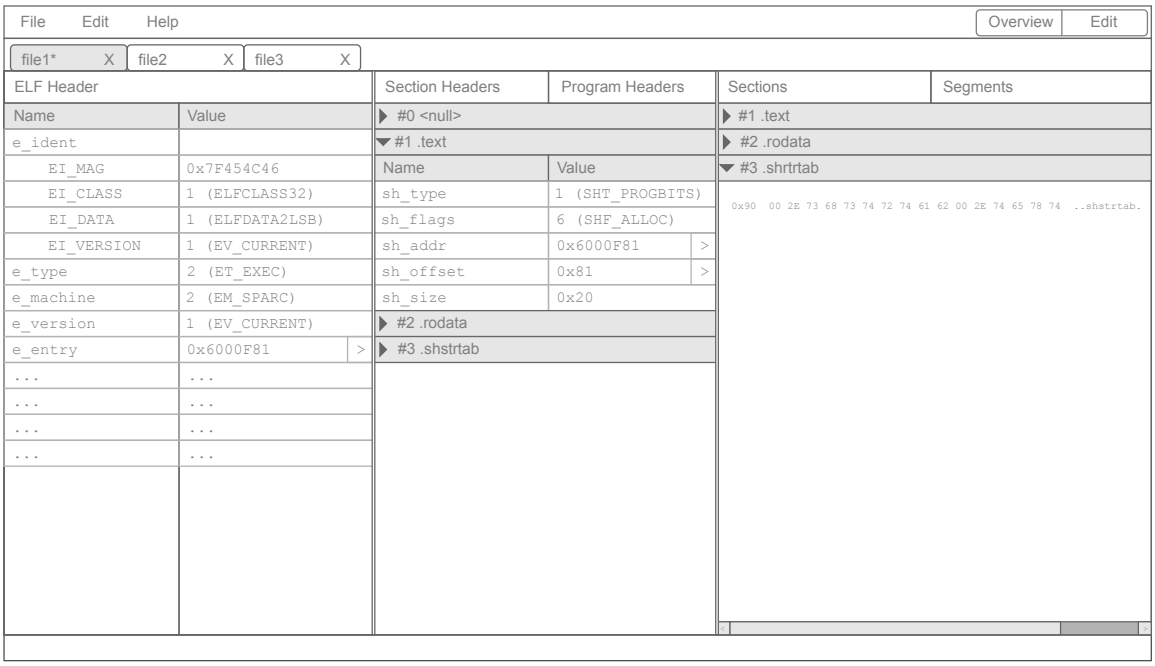
Obrázek 3.2: Návrh zobrazovacího pohledu uživatelského rozhraní.

překrývat nemohou, avšak aplikace musí zvládat i nestandardní soubory a v takovém případě může překrytí nastat i u sekcí. Odkazy jsou znázorněny černým kolečkem u něhož je název položky, která určuje pozici cíle odkazu. Kliknutím na kolečko se pohled přesune na odkazovaný blok.

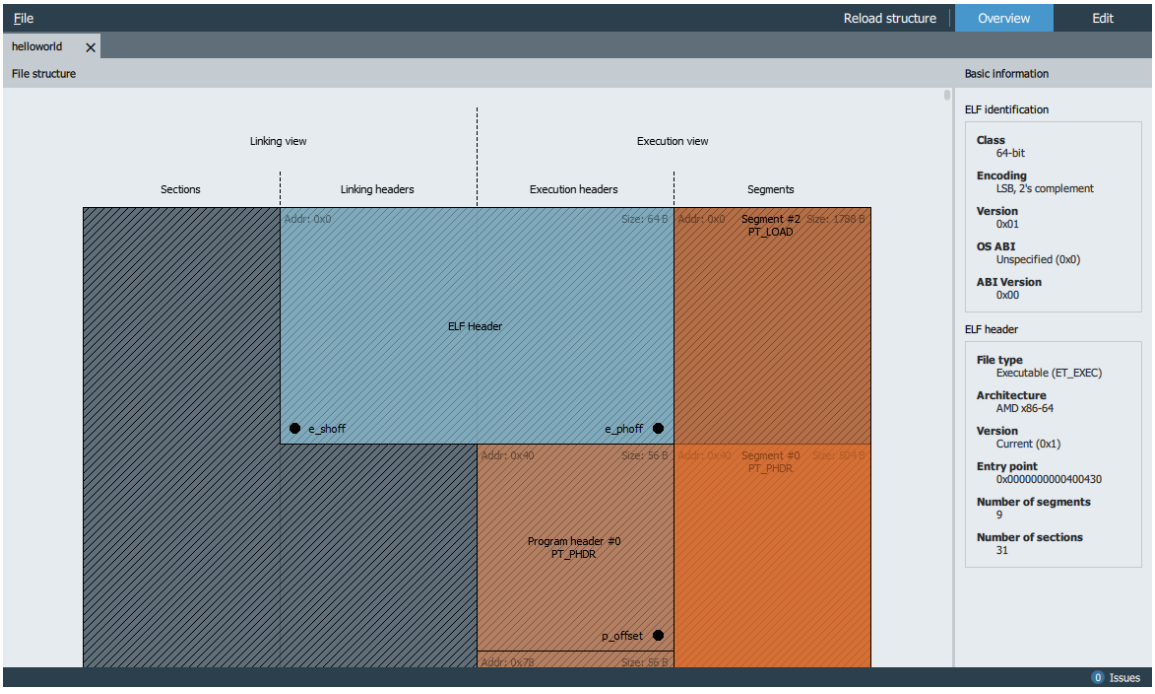
Editační pohled se také trochu změnil. Z obrázku 3.5 je zřejmé, že ve výsledné aplikaci není třetí sloupec, který, jak již bylo zmíněno výše, nebyl implementován. V tabulkách pro editaci položek se význam hodnoty z důvodu větší přehlednosti přesunul do samostatného sloupce nazvaného *Display value*. Hodnoty lze upravovat pouze v hexadecimálním tvaru, což uživatele neomezuje v zadání jakékoliv hodnoty v rozsahu datového typu položky a jednoduše se vstup validuje. Pokud hlavička sekce nebo segmentu není validní (tzn. že zasahuje mimo soubor), objeví se u jejího jména nápis *INVALID*, červený puntík a hodnoty v hlavičce jsou nulové a nelze je editovat. Jestliže položka `sh_name` neobsahuje validní index do sekce řetězců, je místo jména sekce vypsáno *«INVALID_NAME»*.

Dále přibylo tlačítko *Reload structure* na pravé straně hlavního menu, které slouží ke znovu načtení struktury ELF souboru. Ta se může změnit, pokud uživatel upraví některou z položek, co ovlivňuje umístění nebo velikost některé z hlaviček, sekcí či segmentů. Jelikož toto přepočítání struktury může být časově náročná operace, neprovádí se hned po upravení položky, ale její spuštění je ponecháno na uživateli.

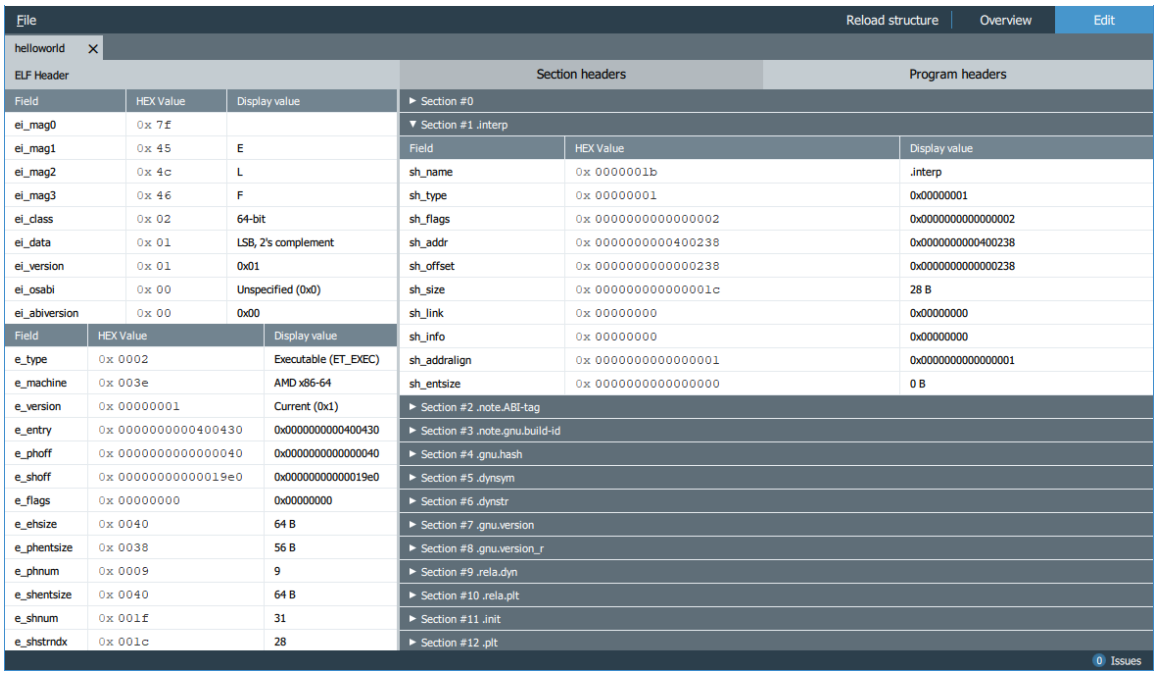
V pravém dolním rohu bylo přidáno tlačítko *Issues* ukazující aktuální počet problémů nalezených v souboru. Při kliknutí se zobrazí seznam těchto problémů s bližšími informacemi. Jaké problémy mohou v souboru nastat je blíže popsáno v sekci 5.3.



Obrázek 3.3: Návrh editační části uživatelského rozhraní.



Obrázek 3.4: Zobrazovací pohled výsledné aplikace.



Obrázek 3.5: Editační pohled výsledné aplikace.

Kapitola 4

Výběr knihoven

V této kapitole je popsán výběr vhodných knihoven pro použití při implementaci výsledné aplikace. V každé sekci je představeno několik zástupců, ze kterých je následně vybrán ten nejvhodnější, nebo je zvolena vlastní implementace v případě nenalezení vhodné knihovny.

4.1 Knihovna pro čtení a manipulaci ELF souborů

Jelikož aplikace má pracovat s binárními soubory ELF, je vhodné vybrat knihovnu pro čtení a zápis tohoto formátu. Aplikace však musí umět načítat i poškozené soubory, proto je potřeba se na toto kritérium zaměřit při výběru. Knihovna také musí být multiplatformní a měla by podporovat minimálně 64bitové ELF soubory, nejlépe však i 32bitové.

ELFIO

C++ knihovna, která podporuje 32bitové i 64bitové ELF soubory a také je přeložitelná na široké škále architektur a kompilátorů. I po sedmi letech od založení repositáře je knihovna stále udržována a rozšiřována původním vývojářem a díky otevřeným zdrojovým kódům i jeho uživateli. Poskytuje jednoduché objektové rozhraní pro čtení i zápis. Knihovna obsahuje pouze hlavičkové soubory, tudíž její použití v projektu je velmi snadné.

Načítání poškozených souborů tato knihovna moc neřeší (většinou chyby ignoruje), poskytuje ale metodu pro validaci souboru po načtení. Ta zatím detekuje pouze překrývající se sekce, ale bylo by možné ji rozšířit o další detekce. Vrací však pouze řetězec s popisem chyby, z čehož nelze v aplikaci zjistit, kde chyba přesně nastala. Další problém nastává v případě, že se knihovně kvůli chybě nepodaří soubor vůbec načíst, pak validační metoda nejde použít vůbec. Pokročilejší detekce poškozeného souboru by tedy vyžadovala větší zásah do kódu knihovny.

Elf.h

Hlavičkový soubor, který obsahuje pouze definice struktur a konstant pro načítání 32bitových i 64bitových ELF souborů. Tento soubor je součástí Linuxu[1] pod licencí GPL-2.0, pro operační systém Windows lze použít verzi, která je součástí aplikačního rámce Rekall[2] taktéž pod licencí GPL-2.0. Jelikož knihovna neobsahuje žádné funkce pro čtení nebo ma-

[1] https://github.com/torvalds/linux/blob/master/include/uapi/linux/elf.h
[2] https://github.com/google/rekall/blob/master/tools/windows/winpmem/executable/elf.h

nipulaci ELF souborů, je programátor odkázán na vlastní implementaci těchto operací. To však umožňuje plnou kontrolu nad načítáním a tedy i nad detekcí poškozených souborů.

elf-parser

Jednoduchá knihovna podporující pouze načítání 64bitových ELF souborů. Její výhodou je použití funkce `mmap`, která umožňuje namapovat soubor do virtuálního adresního prostoru aniž by bylo potřeba přečíst celý soubor z disku (čtení probíhá až při přístupu k paměti), což urychluje práci s velkými soubory. Funkce `mmap` je však systémová funkce operačního systému Linux a není k dispozici na platformě Windows. Knihovna také neumožňuje manipulaci s ELF soubory. Ve výsledku je tedy nevhodná pro použití v tomto projektu.

Závěr

Jelikož není dostupná knihovna, která by přímo splňovala požadavky, je potřeba implementovat potřebnou funkčnost vlastnoručně. Knihovna ELFIO je multiplatformní, dobře odladěná s podporou 32bitových i 64bitových souborů a naprogramována v jazyce C++, což z ní dělá vhodného kandidáta pro inspiraci při vytváření vlastní knihovny s přizpůsobeným aplikačním rozhraním vzhledem k potřebám aplikace.

4.2 Knihovna pro grafické uživatelské rozhraní

Cílem je vybrat vhodnou knihovnu pro programování grafických uživatelských rozhraní v jazyce C++. Knihovna musí podporovat alespoň platformy Linux, Windows a nejlépe i macOS. Jako kandidáti byly vybrány knihovny Qt, wxWidgets, Nana, GTKmm/GTK+, které tyto požadavky splňují.

Qt

Qt není jenom grafická knihovna, ale spíše celý aplikační rámec pro tvorbu uživatelských rozhraní nejenom v prostředí stolních počítačů, ale i vestavěných systémů. Je vyvíjen již od roku 1991 a řadí se mezi nejvyspělejší. K dispozici je verze zdarma (pod licencí LGPLv3) a placená komerční verze. Pro Qt je dostupné množství různých vazeb na jiné jazyky, například Python (PyQt), C# (QtSharp) nebo Haskell (HsQML).

Aplikační rámec Qt poskytuje také vlastní vývojové prostředí QtCreator, vlastní jazyk QML pro deklaraci vzhledu uživatelského rozhraní a nástroj MOC (Meta-Object Compiler), který rozšiřuje C++ o sloty a signály, překlad řetězců a další. Knihovna má také dobrou dokumentaci, řadu modulů a díky její rozšířenosti je k dispozici velké množství návodů. K nevýhodám se řadí například velikost knihovny nebo nemožnost použít nativní prvky GUI dané platformy (Qt pouze imituje jejich vzhled).

wxWidgets

Další vyspělá knihovna pro vytváření uživatelských rozhraní, která se zaměřuje na využívání nativních prvků GUI. Knihovna podporuje několik platforem, například wxGTK (Linux), wxMSW (Windows), wxOSX/Cocoa (macOS) nebo třeba i wxQt (port knihovny wxWidgets pomocí knihoven Qt). Poskytuje řadu vazeb i na jiné jazyky (např. Python, Perl, Ruby). Umožňuje deklarovat GUI pomocí XML, které pak lze načítat za běhu, zkompilovat

do binárních XRS souborů nebo zkompilovat do C++ kódu. Oproti Qt však neposkytuje tak obsáhlou a přehlednou dokumentaci a takové množství návodů.

Nana

Tato knihovna je velmi mladá a díky tomu využívá moderních standardů C++11/14/17 a není zatížena udržováním zpětné kompatibility. Je vyvíjena menším množstvím vývojářů a není zatím tak vyspělá a rozšířená, tudíž neposkytuje tolik možností, co ostatní uvedené knihovny. Dokumentace je také velmi strohá a množství návodů je relativně malé, stejně tak množství dostupných grafických prvků.

GTKmm/GTK+

GTKmm poskytuje C++ rozhraní pro knihovnu GTK+, která původně vznikla pro grafický editor GIMP a následně byla použita v prostředí GNOME. Tato knihovna také poskytuje mnoho vazeb na jiné jazyky, jako například Python, Perl, Ruby, Ada, Lua a další. Jedná se o jednu z nejpopulárnějších knihoven pro vytváření GUI na platformě Linux, pro kterou je primárně určen. Podporuje však i platformy Windows a macOS.

Mezi vlastnosti knihovny GTK+ patří nativní vzhled grafických prvků uživatelského rozhraní, podpora motivů, podpora lokalizace aplikace nebo podpora UTF-8. Uživatelské rozhraní lze definovat pomocí XML souboru. Tyto soubory lze vytvořit ručně nebo pomocí programu GLADE, který umožňuje vytvářet GUI pro GTK+ v grafickém prostředí.

Závěr

Jelikož v projektu bude potřeba vykreslovat grafy pro náhled struktury souboru, byla vybrána knihovna Qt z důvodu její flexibility při vytváření složitějších grafických prvků. Dále také kvůli její rozšířenosti, kvalitě dokumentace a také jednoduchosti sestavování projektu v jejich vlastním vývojovém prostředí QtCreator.

4.3 Knihovna pro zobrazování diagramu

Aplikace má poskytovat přehledný náhled struktury ELF souboru pomocí diagramu. Ten by měl blokově zobrazovat strukturu sekcí a segmentů, jak je ukázáno na obrázku 3.1c. Je tedy vhodné prozkoumat již existující knihovny pro jejich vykreslování. Knihovna by měla využívat knihovny Qt a být dostupná pro platformy Linux, Windows a nejlépe i macOS. Je také potřeba, aby bylo možné programově definovat vlastní vzhled a umístění uzlu diagramu.

Ukázalo se, že takovéto knihovny nejsou k nalezení, proto jsou zde uvedeny knihovny, které se požadavkům alespoň trochu přibližují nebo by mohly být v ohledu zobrazování diagramu užitečné.

NodeEditor

NodeEditor[5] je knihovna založená na Qt, která se zaměřuje na grafové zpracování dat. Graf se skládá z uzlů, které provádí operace nad daty ze vstupů uzlu a jejich výsledek následně propaguje na svoje výstupy. Vstupy a výstupy uzlů jsou pak spojeny propoji, které přenáší data mezi jednotlivými uzly.

Knihovna umožňuje takovéto grafy vytvářet a editovat i uživatelem, umožňuje i do jisté míry upravit vzhled těchto grafů bez zásahu do knihovny. Je dostupná pro platformy Windows, Linux i macOS.

Jelikož je však knihovna silně zaměřena na zpracování dat a editaci grafu, což v tomto projektu není potřeba, nejeví se jako vhodná volba pro řešení tohoto problému.

Graphviz

Graphviz je kolekce softwaru pro zobrazování a manipulování abstraktních grafů [1]. Obsahuje jak programy s grafickým uživatelským rozhraním nebo rozhraním příkazové řádky, tak i knihovnu v jazyce C, která poskytuje funkce pro rozmístění uzlů grafu a vykreslování grafu [3]. Tato knihovna pracuje s grafy vytvořenými knihovnou Cgraph.

Cgraph poskytuje datové typy a operace pro grafy tvořenými uzly s atributy, hranami a podgrafy [4]. Zaměřuje se hlavně na reprezentaci grafu a je navrhnut s úsilím o co největší časovou a paměťovou efektivitu [4]. Atributy uzlů jsou ukládány jako dvojice řetězců název-hodnota [4], což umožňuje uzlům přiřadit vlastní atributy (avšak pouze řetězcové hodnoty).

Knihovna Graphviz umí vykreslovat grafy do různých obrázkových formátů (např. PNG, PostScript), ale poskytuje i aplikační rozhraní pro propojení s vlastním vykreslováním pomocí jakékoliv jiné knihovny [3]. Trochu jiným způsobem propojuje Graphviz s Qt i knihovna QGV[3]. Nejedná se o příliš vyspělou knihovnu, ale může sloužit jako základ pro vlastní implementaci.

Všechny zmíněné knihovny podporují minimálně Windows a Linux. Graphviz sice neposkytuje oficiální balíček pro Linux, ale existují balíčky třetích stran nebo je k dispozici zdrojový kód knihovny pro vlastní kompilaci.

Závěr

Jak již bylo zmíněno, knihovna NodeEditor není vhodný kandidát pro vykreslování navrženého diagramu struktury ELF souboru. Je možné ji pro tento účel použít, avšak primární zaměření této knihovny je zpracování dat, tudíž knihovnu by bylo potřeba značně upravit pro splnění požadavků.

Knihovna Graphviz už je pro tento účel použitelnější. Vykreslování diagramů pomocí Qt však tato knihovna neřeší a je potřeba toto implementovat stejně vlastnoručně. Další hlavní vlastností této knihovny je automatické rozmístění uzlu diagramu, což ale v tomto případě není žádoucí, jelikož je potřeba uzly umístit v závislosti na jejich pozici v souboru. Užitečná by tedy byla pouze knihovna Cgraph pro reprezentaci grafu. Nicméně ta podporuje pouze textové atributy uzlů, a to může být omezující. Proto si myslím, že je lepší naprogramovat vlastní řešení, které lze plně přizpůsobit potřebám.

[3] https://github.com/nbergont/qgv

Kapitola 5

Implementace

V kapitole je popsán způsob implementace klíčových částí aplikace.

5.1 Použité technologie

Aplikace je napsaná převážně v jazyce C++ se standardem C++14. Vyvíjena byla pro operační systémy Windows a Linux, kde na prvním zmíněném systému byla kompilována nástrojem MinGW, na druhém nástroji g++ a Clang. Sestavování aplikace probíhalo pomocí nástroje CMake (minimální požadovaná verze je 3.1). Pomocí tohoto nástroje se také stahuje a sestavuje knihovna Google Test využívaná pro testování menších částí aplikace. Verzování zdrojového kódu bylo zajištěno nástrojem Git a je k dispozici na webové stránce GitHub[1]. S touto platformou byly propojeny služby pro průběžnou integraci, které automaticky sestaví aplikaci a spustí testy při nahrání nových změn do repositáře. Pro testování na platformě Linux se používá Travis CI a pro Windows služba AppVeyor.

Pro vytváření grafického uživatelského rozhraní byla použita knihovna Qt ve verzi 5.11.2 s modulem QtQuick 2.11, který umožňuje programovat uživatelské rozhraní deklarativně v jazyce QML. Pro ladění vzhledu aplikace byl také používán nástroj GammaRay, jenž poskytuje inspekci i manipulaci Qt aplikace za jejího běhu a pomohl vyřešit a pochopit řadu problémů.

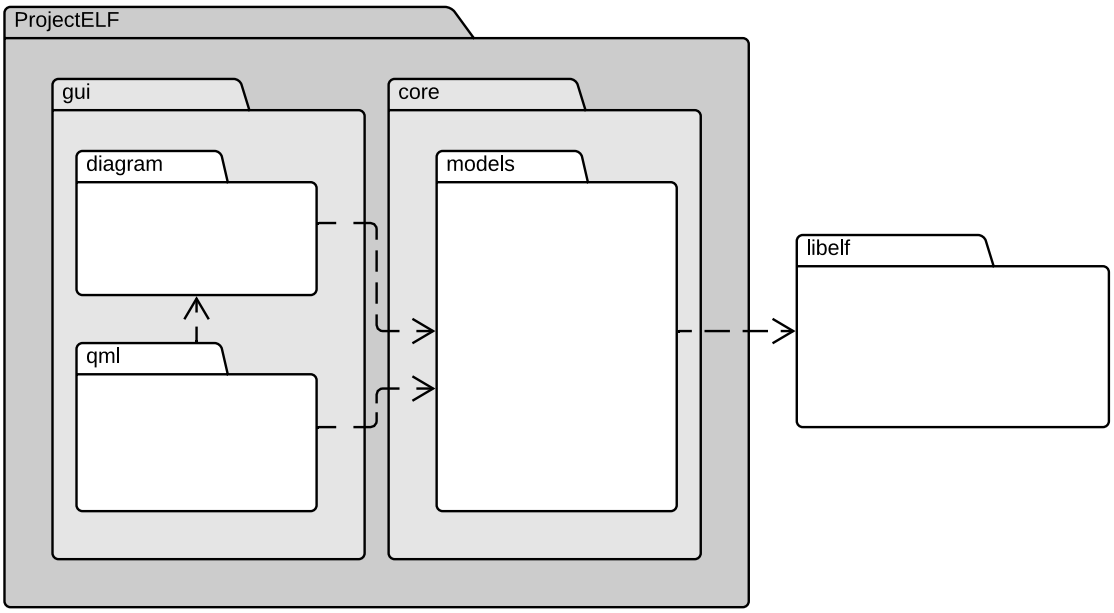
Vývoj v jazyce C++ probíhal v integrovaném vývojovém prostředí CLion od firmy Jetbrains, jenž poskytuje velmi pokročilé možnosti refaktorizace, inspekce a generování kódu. Tento program však nepodporuje jazyk QML, pouze v podobě zásuvného modulu zvýrazňující syntaxi. Proto byl pro vývoj v tomto jazyce využíván převážně nástroj QtCreator, který poskytuje pro tento jazyk i našeptávač a různé možnosti refaktorizace.

5.2 Architektura aplikace

Projekt je rozdělen na knihovnu libelf starající se o načítání, manipulaci a ukládání souborů ELF a na ní závislou aplikaci, která jí poskytuje grafické uživatelské rozhraní. Knihovna je k aplikaci linkována staticky a nemá žádné závislosti. Více o implementaci knihovny je popsáno v sekci 5.3.

[1] https://github.com/EnkeyMC/ProjectELF

Na diagramu 5.1 je vidět rozdělení struktury do balíčků[2] a závislosti mezi nimi. Na nejvyšší úrovni je to již zmíněná knihovna libelf a GUI aplikace nazvaná ProjectELF. Ta se dále dělí na `core` a `gui`.



Obrázek 5.1: UML diagram balíčků a jejich závislostí.

Balíček `core` se mimo pár pomocných tříd a rozhraní sestává z modelů (vnořený balíček `models`), které zaštiťují data z knihovny libelf pomocí vlastností (*properties*) z knihovny Qt, jenž umožňují vázat (*bind*) tato data na grafické uživatelské rozhraní v QML. Také je převádějí do vhodné podoby pro zobrazení (a zpět v případě vstupu).

Modely jsou pak využívány balíčkem `gui`, který se stará o zobrazování jejich dat v GUI. Je rozdělen na dvě části. Jednou je `diagram` implementující vykreslování diagramu struktury ELF souboru ve zobrazovacím pohledu včetně interakce (viz sekce 5.5). Druhá, `qml`, obsahuje veškerý ostatní zdrojový kód grafického uživatelského rozhraní v jazyce QML a využívá právě vlastností definovaných v modelech pro navázání GUI na data. Je závislá také na balíčku `diagram`, protože definuje umístění diagramu a předává mu správný model, který má zobrazovat.

Pokud potřebují modely uživatelskému rozhraní například oznámit chybu, používá se systém signálů a slotů knihovny Qt. Model definuje signál, který vyvolá v případě chyby, a v QML se definuje reakce na tento signál (tzv. slot). To umožňuje objekty volně vázat, tudíž nevzniká závislost modelů na GUI.

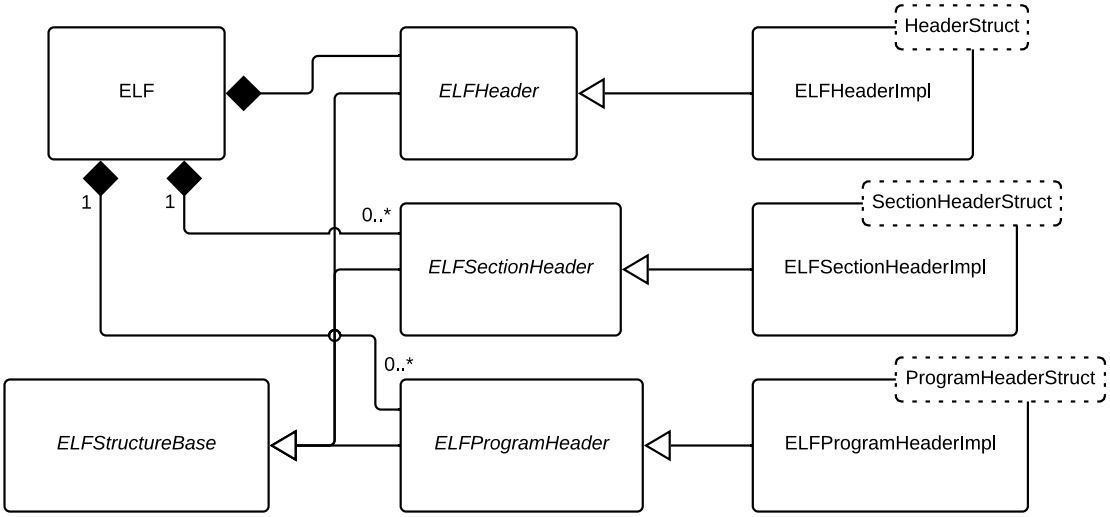
5.3 Knihovna libelf

Libelf implementuje načítání, ukládání a modifikaci ELF souborů. Je založena na knihovně ELFIO, ze které přebírá zdrojový soubor s definicemi datových typů a konstant formátu ELF a také zdrojový soubor s různými pomocnými makry a třídou pro konvertování endianity dat. Tyto soubory mají v záhlaví komentář s původní licencí. Původně byl částečně

[2]Balíčkem je zde myšlena logická část. Reálně se jedná o adresáře ve zdrojovém kódu.

převzat i algoritmus načítání souboru, ten byl však později přepsán z důvodů uvedených níže v této sekci.

Převzatý je také způsob podpory 32bitových i 64bitových souborů. Abstraktní třídy `ELFHeader`, `ELFSectionHeader` a `ELFProgramHeader`, které lze vidět na diagramu tříd 5.2, neobsahují konkrétní strukturu s daty, ale pouze deklarují čistě virtuální metody pro přístup k jednotlivým položkám hlavičky. Tyto metody mají návratový typ pro 64bitové soubory, takže je možné vrátit hodnotu z obou typů souborů bez ztráty dat (32bitové datové typy mají menší rozsahy hodnot než 64bitové). Konkrétní struktura se dosadí až jako parametr odvozené šablonové třídy (`ELFHeaderImpl`, `ELFSectionHeaderImpl`, `ELFProgramHeaderImpl`). V té jsou pak implementované metody, které již pracují se správnými datovými typy. Ve zbytku aplikace tedy není potřeba rozlišovat mezi typy souboru, protože jejich rozhraní je stejné.



Obrázek 5.2: UML diagram vybraných tříd knihovny libelf.

Načítání souboru

Ze začátku bylo načítání souboru implementováno stejným způsobem, jako v knihovně ELFIO. Obsah souboru se binárně načítal do struktur pomocí funkce `std::istream::read()`, což znamenalo, že každá struktura měla svoji kopii dat. To pak působilo problémy, pokud se některé struktury v souboru překrývaly (například segmenty běžně překrývají hlavičky), jelikož při modifikaci dat se upravila pouze jedna kopie a při ukládání zpět do souboru se mohlo stát, že druhá struktura data přepsala zpátky na původní. Zprvu se tento problém vyřešil obrácením pořadí ukládání struktur. Jako první se uložily segmenty, následně jejich hlavičky, poté sekce a jejich hlavičky, a jako poslední hlavička ELF. To však problém vyřešilo pouze pro validní soubory. Při ukládání nevalidního souboru stále mohla nastat situace, kdy se data přepsala.

Přešlo se tedy na jiný způsob načítání. Celý obsah souboru se zkopíruje do paměti aplikace a jednotlivým strukturám se pak nastaví pouze ukazatel na správné místo v této paměti a objekt se označí jako validní. Pokud struktura zasahuje mimo bajty souboru, její ukazatel nastaven není a objekt zůstane nevalidní, aby se zabránilo neoprávněnému přístupu do paměti. Tento způsob zajistí, že se data neduplikují, jelikož překrývající se

struktury přistupují do stejné paměti. Nevýhodou je kopírování celého obsahu souboru do paměti, i když nakonec nemusí být všechna data využita.

Ukládání souboru

Než se změnil způsob načítání, ukládal se soubor po jednotlivých strukturách, podobně jako se načítal. Začalo se od hlavičky souboru ELF, poté se uložily hlavičky sekcí a jejich sekce, nakonec programové hlavičky a jejich segmenty. Pozice struktur v souboru byly určeny aktuálními hodnotami odkazů v hlavičkách. Jak již bylo zmíněno, později bylo pořadí ukládání převráceno.

Poté co se změnilo načítání souboru, se ukládání výrazně zjednodušilo. Nyní se celá kopie souboru v paměti jednoduše uloží na disk.

Přístup k datům

Jelikož formát ELF podporuje způsob uložení bajtů ve formátu little-endian i big-endian a data jsou do struktur načítána binárně, nelze k datům přistupovat přímo. Je potřeba je při čtení či zápisu konvertovat. Proto je pro každou položku ve struktuře k dispozici sada metod. Jelikož jsou pro všechny položky z velké části stejné, jsou generovány makrem. To bylo převzato z knihovny ELFIO s mírnými úpravami, jako je třeba kontrola příznaku validity hlavičky před přístupem k položce, aby se zabránilo neplatnému přístupu do paměti. Makro generuje metody pro čtení a zápis hodnot s automatickým převáděním do správného formátu bajtů. Oproti ELFIO byla přidána navíc generace metody pro získání velikosti datového typu položky v bajtech, což je využito pro validaci vstupních hodnot v GUI.

Vypořádání se s chybami

Jak bylo zmíněno v sekci 3.1, aplikace musí zvládat i poškozené soubory. Knihovna toto řeší tak, že pokud jakákoliv struktura zasahuje mimo soubor, je nevalidní a není možné ji číst ani upravovat. Dá se tedy načíst například i soubor ve formátu big-endian, který ale v hlavičce bude mít nastavený formát bajtů na little-endian. To způsobí, že většina položek v hlavičkách bude obsahovat nesmyslné hodnoty. Převážné množství struktur tedy bude nevalidních, avšak minimálně hlavička ELF se načte, což umožní změnit hodnotu na big-endian a soubor tak opravit.

Jelikož položky mohou obsahovat jakékoliv hodnoty, může nastat situace, kdy v hlavičce bude definováno velké množství sekcí či segmentů. I když se v souboru nebudou reálně vyskytovat, knihovna pro ně stejně vytváří objekty, aby bylo možné zobrazit v GUI, že nejsou validní. To způsobovalo problémy jak časové, při načítání struktury, tak paměťové. Z tohoto důvodu se načte sekcí i segmentů pouze 100 a pro zbylé se již objekty nevytváří. Pro validní soubory by toto omezení nemělo být problém, protože počet sekcí v běžných spustitelných souborech bývá kolem 30 a segmentů kolem 10. Tento limit však lze ve výsledné aplikaci upravit argumentem příkazové řádky.

Když se soubor podaří načíst, je možné pomocí funkce `ELF::find_issues()` najít všechny chyby v souboru. Jejich seznam, které knihovna detekuje, je uvedený v tabulce 5.1. Chyby jsou rozděleny do tří závažností:

- Upozornění – malá chyba, nezasahuje do struktury souboru. Soubor lze načíst.
- Chyba – závažnější chyba, může ovlivnit strukturu souboru, ale soubor lze stále načíst.

- Kritická chyba – soubor nelze načíst.

Položka	Závažnost chyby	Kdy chyba nastane
`ei_mag`	Chyba	Pokud neobsahuje správné hodnoty (viz 2.1).
`ei_class`	Kritická chyba	Pokud soubor není 32 nebo 64bitový.
`ei_data`	Kritická chyba	Pokud soubor není ve formátu little-endian nebo big-endian.
`ei_version`	Upozornění	Pokud je hodnota 0.
`e_version`	Upozornění	Pokud je hodnota 0.
`e_phoff`, `e_phentsize`, `e_phnum`	Chyba	Pokud tabulka programových hlaviček zasahuje mimo soubor nebo překrývá hlavičku ELF.
`e_shoff`, `e_shentsize`, `e_shnum`	Chyba	Pokud tabulka hlaviček sekcí zasahuje mimo soubor nebo překrývá hlavičku ELF.
`e_ehsize`	Upozornění	Pokud hodnota neodpovídá reálné velikosti hlavičky.
`e_shstrndx`	Chyba	Pokud je hodnota větší nebo rovna počtu sekcí.
`sh_offset`, `sh_size`	Chyba	Pokud sekce není typu `SHT_NOBITS` a zároveň zasahuje mimo soubor.
`p_offset`, `p_filesz`	Chyba	Pokud segment zasahuje mimo soubor.
Sekce	Chyba	Pokud se sekce překrývají.
Tabulka řetězců	Chyba	Pokud tabulka nekončí binární nulou.
Tabulka řetězců	Upozornění	Pokud tabulka nezačíná binární nulou.

Tabulka 5.1: Seznam chyb, které aplikace detekuje.

Chyby formátu ELF jsou reprezentovány třídou `ELFIssue`, která obsahuje informace o závažnosti, zdroji a typu chyby. Pokud je zdroj chyby indexovatelný, tzn. že se v souboru může vyskytovat vícekrát (sekce, segment, hlavička sekce nebo programová hlavička), obsahuje třída i jeho index. Kromě indexu jsou všechny tyto informace uložené jako hodnoty výčtového typu (*enum*) a do textové podoby jsou převáděny až v aplikaci ProjectELF, což umožňuje případný překlad těchto řetězců do různých jazyků nebo programově dohledat, kde chyba nastala.

Pro jednodušší práci s chybami byl vytvořen i kontejner `ELFIssuesBySeverity`, který je třídí podle závažnosti. Objekty třídy `ELFIssue` lze do kontejneru jednoduše vkládat přetíženým operátorem `+=`. Stejný operátor umožňuje i vložit obsah jednoho kontejneru do druhého. Poskytuje mimo jiné i pomocnou metodu `get_issues()`, jenž vrátí všechny chyby v datovém typu `std::vector<ELFIssues>` pro jednoduchou iteraci.

Obsah sekcí a segmentů analyzován není, jelikož v aplikaci nebylo implementováno jeho zobrazování. Vyjma sekce `.shstrtab` (viz 2.1), která se využívá pro výpis názvu sekcí. U této sekce může nastat problém, že nebude řádně ukončená binární nulou a mohlo by při čtení řetězce snadno dojít k nepovolenému přístupu do paměti. Funkce `ELF::get_name()`, jenž navrátí řetězec z této tabulky podle zadaného indexu, tedy nemůže jednoduše vracet ukazatel na začátek řetězce. Místo toho předává zpět instanci třídy `std::string`, do které je nakopírován řetězec od indexu až do první binární nuly nebo konce sekce, podle toho, co nastane dřív. Tím se zajistí, že nedojde k nechtěnému čtení mimo alokovanou paměť.

5.4 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní je napsáno v jazyce QML, jenž je součástí Qt modulu QtQuick. Jedinou výjimkou je vykreslování diagramu, které je napsané v jazyce C++, z důvodu komplexnosti a výkonu (více viz v sekci 5.5). Výslednou podobu rozhraní lze vidět na obrázcích 3.4 a 3.5.

Použité QtQuick moduly

QtQuick obsahuje několik různých modulů. Pro ovládací prvky (tlačítka, vstupní pole, menu, atp.) byl použit modul QtQuick.Controls 2.4, který je nástupcem Qt Quick Controls 1. Je lépe optimalizovaný a jednodušší na použití. Postrádá však některé ovládací prvky, jako například `SplitView`, který bude přidán až ve verzi Qt 5.13. Ten umožňuje prvky rozložit do sloupců (případně řádků), jejichž velikost může uživatel manuálně měnit. To je použito jak ve zobrazovacím, tak editačním pohledu aplikace. Naštěstí lze prvky z jednotlivých modulů mezi sebou mixovat, takže pro tyto případy byly využity prvky ze staršího modulu QtQuick.Controls 1.4.

Pro rozložení ovládacích prvků byly použity položky z modulu QtQuick.Layouts 2.11. `StackLayout` zobrazuje vždy pouze jeden z přiřazených prvků, což bylo využito na všech místech, kde se přepíná mezi několika pohledy. Například přepínání zobrazovacího a editačního pohledu nebo přepínání mezi otevřenými soubory. `GridLayout` uspořádává prvky do mřížky o předem daném počtu sloupců či řádků (záleží na nastavení toku). To je využito pro tabulky v editačním pohledu. `ColumnLayout` a `RowLayout` jsou speciální případy `GridLayout`, které mají pouze jeden sloupec respektive řádek, a jsou použity pro rozložení prvků vertikálně či horizontálně. Díky všem těmto rozložením se uživatelské rozhraní dynamicky přizpůsobuje velikosti okna.

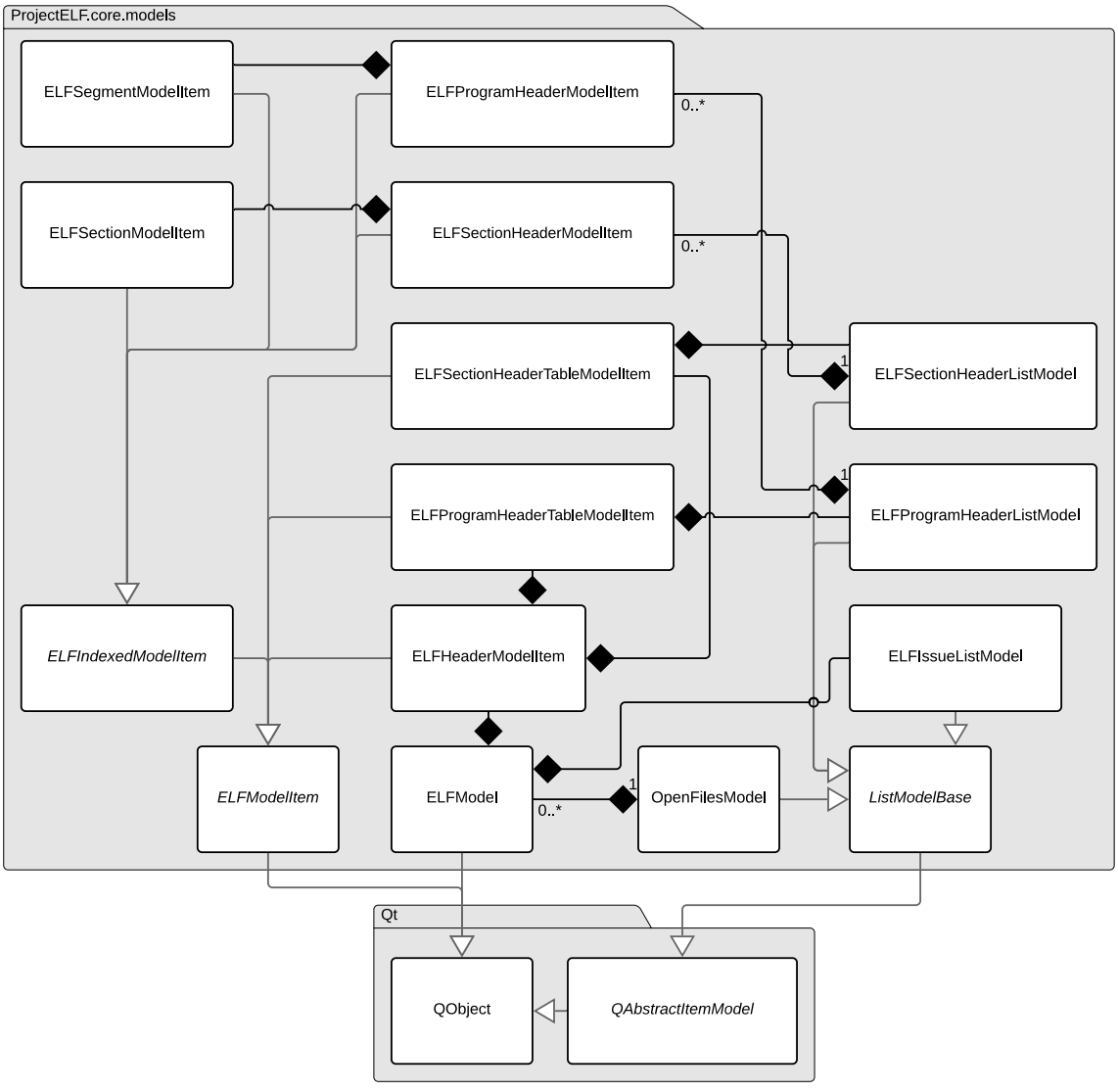
Posledním použitým modulem je QtQuick.Dialogs 1.3. Ten obsahuje řadu jednoduše použitelných dialogů. Například `MessageDialog`, který slouží pro zobrazení zprávy nebo položení otázky uživateli. Dialogu lze kromě textu nastavit i ikonu, ze sady standardních ikon, a tlačítka typu *Ok*, *Cancel*, *Save* a podobně. `FileDialog` zase souží k vybrání souboru ze souborového systému pro načítání nebo k vytvoření nového v případě ukládání. Všechny tyto dialogy používají pokud možno nativní systémové dialogy, jinak využívají implementaci knihovny Qt.

Propojení s knihovnou libelf

Grafické uživatelské rozhraní je propojeno s knihovnou libelf pomocí tříd z balíčku `models` (viz obrázek 5.1). Tyto třídy obalují objekty knihovny a zpřístupňují jejich data pomocí makra `Q_PROPERTY` z knihovny Qt, které vytváří tzv. vlastnosti (*properties*). Ty lze pak

v jazyce QML vázat na jiné vlastnosti. Tyto vazby pak díky signálům zajišťují, že když se změní hodnota v modelu, automaticky se změní i hodnota vázané vlastnosti v GUI a dojde k překreslení. Jelikož objekty tříd z knihovny libelf nelze takto vázat, je potřeba k jejich datům přistupovat pouze přes modely, jinak by se při změně neaktualizovalo uživatelské rozhraní.

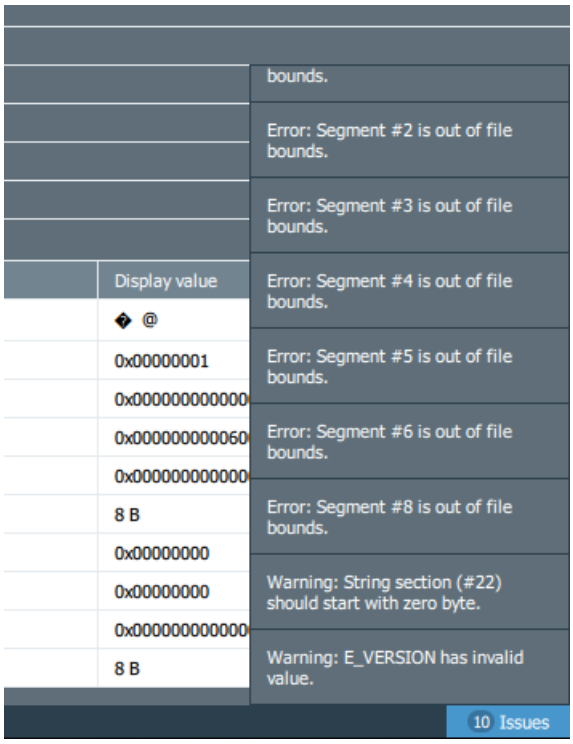
Hierarchie modelů je zachycena UML diagramem na obrázku 5.3. Všechny třídy vycházejí z typu `QObject`, což jim umožňuje používat systém signálů a slotů a lze je registrovat jako QML typ. Třídy, které dědí z `QAbstractItemModel`, lze přímo používat jako model pro QML komponenty zobrazující seznamy (`ListView`, `Repeater` a podobné). Hierarchie typů vycházející z `ELFModelItem` byla primárně vytvořena pro použití v diagramu, avšak později se ukázalo, že není úplně vhodná pro napojení na QML komponenty. Musely být tedy přidány třídy `ELFSectionHeaderListModel` a `ELFProgramHeaderListModel`, jenž dědí ze třídy `QAbstractItemModel`, tudíž mají již zmíněné výhody pro zobrazení v QML.



Obrázek 5.3: UML diagram tříd modelů.

Kořenovým modelem aplikace je `OpenFilesModel`. Ten uchovává seznam otevřených souborů. Pro každý soubor si uchovává jeho cestu (kde je uložen na disku) a objekt třídy `ELFModel` s jeho strukturou. V GUI je napojený na záložky se seznamem otevřených souborů a poskytuje metody pro načítání, ukládání a zavírání souborů, ale také i pro znovu načtení struktury souboru, což je napojeno na tlačítko *Reload structure*.

`ELFModel` obaluje třídu `ELF` z knihovny libelf. Mimo jiné obsahuje ukazatel na hlavičku `ELFHeaderModelItem`, informaci o tom jestli byla data modelu upravena, a také seznam chyb v souboru v podobě objektu třídy `ELFIssueListModel`. `ELFModel` se využívá kdekoliv se vypisuje identifikační část hlavičky ELF (`ei_class`, `ei_data`,...) nebo jeden z jeho vnořených modelů. `ELFIssueListModel` se používá pro zobrazování seznamu chyb, který lze vidět na obrázku 5.4.



Obrázek 5.4: Obrázek seznamu chyb v souboru ve výsledné aplikaci.

V tabulce 5.2 lze vidět, jak modely korespondují s třídami z knihovny libelf. Pro každou položku ELF souboru jsou v modelech vytvořeny tři vlastnosti. Jedna je pro čtení a zápis hodnoty položky v hexadecimálním tvaru. Druhá je pouze pro čtení a pokud možno převádí hodnotu položky na řetězec s jejím významem. Například hodnotu 1 položky `ei_class` převede na řetězec „32-bit“. Tento převod je definován ve třídě `ELFValueConverter`. Pokud pro hodnotu není nalezena konverze, vrátí se řetězec „Unknown (0x`<val>`)“, kde `<val>` je hodnota v hexadecimálním tvaru. Hodnoty označující počet nebo index se převádí do decimálního tvaru a velikosti se převádějí na bajty. V hexadecimální soustavě zůstávají adresy nebo položky jejichž konverze zatím nebyla implementována. Existují ještě speciální případy, kdy se hodnota převádí na znak (`ei_mag0` až `ei_mag3`) nebo pomocí tabulky řetězců (`sh_name`).

Třetí vlastností je velikost položky v bajtech. Ta se může měnit v závislosti na tom, zda je soubor 32 nebo 64bitový. Její hodnota je využívána pro validaci hodnot ve vstupním poli pro

editaci položky. Pro editaci je použitý ovládací prvek `TextField`, který umožňuje nastavit maximální délku vstupu a masku. Jelikož se hodnoty editují v hexadecimálním tvaru, stačí pro validaci nastavit maximální délku na dvojnásobek velikosti položky v bajtech a maskou povolit pouze hexadecimální znaky[3].

Změna hodnoty v modelu se provede, až pokud vstupní pole obsahuje validní hodnotu a byla stisknuta klávesa `enter` nebo pole ztratilo tzv. *focus*, což zabrání vyskakování chyb již během editace.

Model	Reprezentuje
`ELFModel`	`ELF`
`ELFHeaderModelItem`	`ELFHeader`
`ELFSectionHeaderTableModelItem`	Kolekce `ELFSectionHeader`
`ELFProgramHeaderTableModelItem`	Kolekce `ELFProgramHeader`
`ELFSectionHeaderModelItem`	`ELFSectionHeader`
`ELFProgramHeaderModelItem`	`ELFProgramHeader`
`ELFSectionModelItem`	Data sekce z třídy `ELFSectionHeader`
`ELFSegmentModelItem`	Data segmentu z třídy `ELFSegmentHeader`

Tabulka 5.2: Mapování modelů na třídy z knihovny libelf.

5.5 Vykreslování diagramu

Jelikož nebyla nalezena vhodná knihovna pro vykreslování diagramu, byla zvolena vlastní implementace. Toto vykreslování je poměrně komplexní a může být náročné na výkon počítače, proto je tato část grafického rozhraní napsána v jazyce C++ místo QML.

Pro vykreslování byla nejdříve zvažována třída `QGraphicsScene`, jenž umožňuje kromě různých tvarů vložit do scény i komplexnější objekty typu `QWidget`. Další výhodou je, že ve všech potomcích scény, lze jednoduše reagovat na uživatelský vstup. Avšak tato třída je již zastaralá a není podporována v jazyce QML.

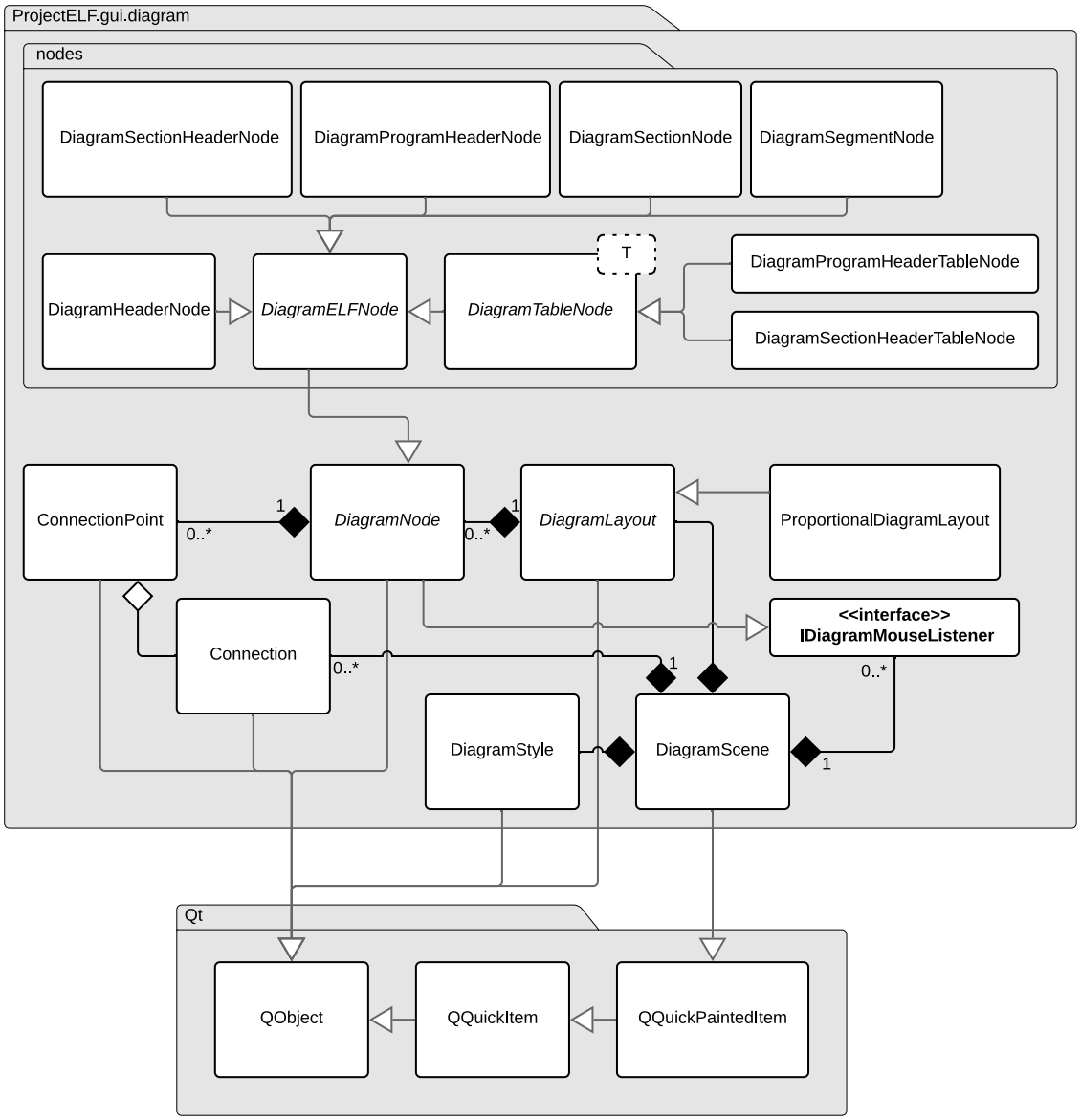
Pro jazyk QML existují dvě komponenty pro vykreslování vlastního obsahu: `Canvas` a `QQuickPaintedItem`. První z nich je určená pro vykreslování pomocí jazyka JavaScript přímo v rámci QML, což není příliš výkonné a sama dokumentace [6] doporučuje používat `QQuickPaintedItem` s implementací v C++ pro větší plátna, časté aktualizace nebo animace. Zvolena tedy byla komponenta `QQuickPaintedItem`.

Struktura tříd a jejich funkce

Hierarchie tříd v balíčku `ProjectELF.gui.diagram` je znázorněna na obrázku 5.5. Hlavní třídou je `DiagramScene`, která vychází z již zmiňované komponenty `QQuickPaintedItem`. Třída je závislá na objektu typu `DiagramStyle`, jenž určuje barvy pro vykreslování diagramu, a typu `ELFModel` poskytující data pro zobrazení. Oba objekty jsou předávány přes vlastnosti v jazyce QML pro možnost případné dynamické změny. Třída dále obsahuje instanci abstraktní třídy `DiagramLayout`, která má zatím jedinou implementaci

[3] Například maska daná řetězcem „HHHH“ povolí pouze čtyři hexadecimální znaky

`ProportionalDiagramLayout`, kolekci hran[4] diagramu v podobě objektů třídy `Connection` a také kolekci objektů typu `IDiagramMouseListener`. Při změně struktury modelu se veškeré bloky a propoje v diagramu odstraní a vytvoří se znovu podle nové struktury. Zde je prostor pro optimalizaci, kdy by se nemusel celý diagram sestavovat znovu, ale pouze by se přizpůsobil nové struktuře.



Obrázek 5.5: Zjednodušený UML diagram tříd balíčku `ProjectELF.gui.diagram`.

`DiagramNode` je základní třídou pro všechny typy bloků v diagramu. Kromě třídy `QObject` dědí také z rozhraní `IDiagramMouseListener`, aby mohla přijímat události myši. Blok může definovat pojmenované body ve formě objektů typu `ConnectionPoint` pro připojení propojů (`Connection`). Například `DiagramHeaderNode` definuje bod „`e_shoff`“ pro odkaz na tabulku hlaviček sekcí a „`e_phoff`“ pro tabulku programových hlaviček. Každý blok také specifikuje svoji procentuální výšku a pozici v diagramu. Tyto hodnoty jsou dány

[4]Propojení mezi dvěma bloky diagramu.

desetinným číslem od nuly do jedné, kde nula znamená nulovou výšku respektive začátek diagramu a jednička, že blok je přes celý diagram nebo na jeho konci v případě pozice. Dále blok udává svoji minimální výšku v pixelech.

Abstraktní třída `DiagramELFNode` je bázová pro všechny bloky diagramu reprezentující strukturu ELF souboru. Pracuje s ukazatelem na obecný `ELFModelItem` z balíčku `ProjectELF.core.models`. Konkrétní třídy vycházející z `DiagramELFNode` pracují již s konkrétními modely, které reprezentují. Každý `ELFModelItem` dává k dispozici svoji adresu a velikost v souboru. `DiagramELFNode` tedy počítá svoji procentuální výšku jednoduchým vztahem 5.1 a svoji pozici vztahem 5.2.

$$procentualniVyska = \frac{velikostModelu}{celkovaVelikostSouboru} \tag{5.1}$$

$$procentualniPozice = \frac{adresaModelu}{celkovaVelikostSouboru} \tag{5.2}$$

`DiagramLayout` se stará o rozložení diagramových bloků (`DiagramNode`). Umístění bloků se přepočítává pouze při změně jejich struktury, což šetří výpočetní výkon. Aktuálně jediná implementace je `ProportionalDiagramLayout`. Ta rozmisťuje bloky přesně tak, jak se nacházejí v souboru. Nejdřív vypočítá výšku diagramu tak, aby žádný blok po umístění nebyl menší než jeho minimální výška. Následně se pro každý blok vypočítá obdélník (`QRect`) na základě jeho procentuální výšky a pozice vynásobené vypočtenou výškou diagramu. Horizontální souřadnice se určí podle sloupce, do kterého blok náleží. Obdélník je pak bloku přiřazen a definuje jeho umístění.

Jak již bylo zmíněno v sekci 3.3, pro velké soubory mohou vznikat vysoké diagramy. Proto byla snaha implementovat i `CondensedDiagramLayout`, jenž by zkrátil části diagramu, kde nedochází k žádným změnám (nekončí ani nezačíná žádný blok). Vytvoření takového algoritmu však nebylo jednoduché a kvůli řadě problémů byl jeho vývoj pozastaven a zanedlouho zrušen.

`Connection` vykresluje propojení mezi dvěma bloky diagramu v podobě šipky. V případě, že odkazovaný blok není validní (je mimo soubor), je propoj místo šipky ukončen červeným křížkem. Začátek propoje se připojuje na `ConnectionPoint`, který si jej uloží, aby mohl při kliknutí přesunout pohled na odkazované místo. Konec propoje se připojuje na vrchní hranu odkazovaného bloku (pouze v případě, že je validní), jelikož odkazy v ELF souboru vždycky ukazují na první bajt struktury. `Connection` je ve výchozím stavu neviditelný. Je však připojen na signály `hoverEntered` a `hoverLeaved` propojených bloků, při kterých se nastavuje viditelnost propoje tak, aby byl viditelný pouze při najetí myši na blok. To diagram zpřehledňuje, protože pokud by se zobrazovalo velké množství propojů najednou, lehce by se v nich ztrácelo.

Může však nastat situace, kdy se vertikální část dvou šipek překrývá a není pak jasné, kam která vede. Řešením je rozdělením šipek do „úrovní“. Ty, jenž vedou z bloku typu `DiagramHeaderNode`, mají úroveň 0 a propoje vedoucí z tabulky hlaviček do sekcí či segmentů mají úroveň 1. Čím vyšší je úroveň propoje, tím je vertikální část šipky dál od diagramu, což zabrání překryvu. Jediný problém nastává, pokud jsou dva nebo více bloků přes sebe. Při najetí na ně se pak šipky zobrazí ve stejné úrovni a překrývají se. Toto již v aplikaci není řešeno a je to ponecháno pro budoucí rozšíření.

Uživatelský vstup

Jelikož `QQuickPaintedItem` pro své potomky neřeší uživatelský vstup, bylo potřeba propagování událostí na položky diagramu implementovat vlastnoručně.

Všechny položky, které chtějí přijímat události myši, musí implementovat rozhraní třídy `IDiagramMouseListener`. Ta definuje metody jak pro příjem událostí při stisknutí a uvolnění tlačítek myši, vstupu myši na položku nebo při opuštění položky, tak pro použití ve `VerticalBinaryTree` z balíčku `ProjectELF.core`. Ten se využívá pro rychlejší vyhledávání položek, které mají událost dostat. Jedná se o upravenou verzi datové struktury QuadTree, jenž zanedbává horizontální dimenzi. Jelikož má diagram konstantní šířku, ale bývá velmi vysoký, není potřeba plochu rozdělovat na kvadranty, jak se to dělá v datové struktuře QuadTree, ale stačí ji rozdělovat vertikálně. Každý uzel `VertivalBinaryTree` tedy obsahuje místo čtyř pouze dva potomky (vrchní a spodní). Strom se musí při každé změně polohy bloků v diagramu znovu sestavit, to se však děje velmi zřídka (pouze při stisknutí tlačítka *Reload structure*). Při vyvolání události stisku či uvolnění tlačítka myši jsou v tomto stromu vyhledány všechny položky, které obsahují souřadnice myši v době vyvolání události, podobným způsobem, jako se vyhledává v binárním vyhledávacím stromě. Vyhledaným položkám se následně předá událost.

5.6 Testování

Aplikace byla testována převážně manuálně. Pouze na načítání ELF souborů a několik dalších tříd byly vytvořeny jednotkové testy. Manuální testování probíhalo na následujících platformách:

- Windows 10
  - Qt 5.11.2, MinGW 5.3 32-bit
  - Qt 5.12.3, MinGW 7.3 64-bit
- Ubuntu 16.04
  - Qt 5.11.3, GCC 64-bit
- macOS Mojave (10.14.5)
  - Qt 5.12.3, Clang 64-bit

Přestože aplikace nebyla primárně vyvíjena pro operační systém macOS, byla nakonec na této platformě otestována a ukázalo se, že i na ní je aplikace funkční.

Aplikace se testovala na sadě souborů, která je k dispozici na přiloženém CD ve složce `elf-test-files`. Sada obsahuje 9 validních souborů různých druhů převzatých z repositáře knihovny ELFIO a 22 nevalidních souborů, které testují různé poškození ELF souboru, jenž aplikace detekuje. Díky testování pomocí těchto souborů bylo nalezeno a následně opraveno mnoho chyb. Při testování detekce a zvládání chyb byly soubory upravovány i přímo v editoru aplikace.

Při testování s velkými soubory se ukázalo, že 32bitová verze aplikace má problém s načítáním souborů větších než přibližně 30 MB. Při pokusu alokovat paměť pro soubor je vyvolána výjimka typu `std::bad_alloc` i přes to, že se zdá být dostatek volné paměti RAM. Příčina tohoto chování nebyla zjištěna. 64bitová verze aplikace na tento problém nenaráží.

Kapitola 6

Závěr

V rámci této bakalářské práce vznikla aplikace pro zobrazování a editaci souborů ELF v grafickém uživatelském rozhraní. Ve zobrazovacím pohledu přehledně znázorňuje pomocí diagramu vnitřní strukturu souboru a lze pomocí něj zjistit, jak jsou v souboru umístěné jednotlivé hlavičky, sekce a segmenty. Diagram také zobrazuje odkazy mezi jednotlivými bloky a umožňuje mezi nimi základní navigaci. Editační pohled pak nabízí možnost úpravy všech hodnot v hlavičce ELF souboru, v hlavičkách sekcí i v programových hlavičkách. Aplikace také zvládá načítat poškozené soubory a uživatele informuje o nalezených chybách v souboru. Načítat lze 32bitové i 64bitové ELF soubory. Podporované platformy jsou Windows, Linux i macOS.

Před implementací aplikace bylo potřeba se seznámit s formátem ELF. Následně se na základě analýzy požadavků vytvořil návrh grafického uživatelského rozhraní, který byl průběžně konzultován s vedoucím práce i odborným konzultantem. Dále byly potřeba vybrat vhodné knihovny pro implementaci aplikace v jazyce C++. Knihovna Qt byla vybrána jako nejvhodnější pro vytvoření grafického uživatelské rozhraní. Pro čtení a upravování souborů ELF vhodná knihovna nalezena nebyla, proto byla zvolena vlastní implementace na základě knihovny ELFIO. Vhodnou knihovnu pro vykreslování diagramu se také nepodařilo nalézt a nezbylo tedy nic jiného než implementovat vlastní řešení. Poté již následovala samotná implementace, která je popsána v kapitole 5.

V aplikaci se podařilo implementovat téměř všechny navržené funkce a vlastnosti. Nestihlo se však implementovat zobrazování a editace obsahu sekcí a segmentů v editačním pohledu. Aplikace má také pár omezení. Ve výchozím stavu nelze zobrazit více jak 100 sekcí či segmentů, jelikož 32bitová aplikace má problémy s alokací paměti pro tolik položek. 64bitová více položek zvládne, ale aplikaci to velmi zpomalí. Tento limit je však možné pomocí argumentu příkazové řádky upravit. 32bitová verze má také problém s načítáním souborů nad 30 MB. Z těchto důvodů je doporučeno používat 64bitovou verzi.

V budoucnu by aplikace mohla být rozšířena například o lepší navigaci v diagramu, kompaktnější zobrazení diagramu větších souborů nebo lepší řešení překrývajících se bloků diagramu. Také by bylo dobré přidat zobrazování a možnost editace obsahu sekcí a segmentů, jak bylo původně navrženo. Vhodné by bylo také lépe aplikaci optimalizovat pro načítání větších souborů. Rozšířit by se mohly také možnosti úprav, jako například přemisťování sekcí a segmentů, jejich přidávání a tak podobně.

Literatura

[1] Ellson, J.; Gansner, E. R.; Koutsofios, E.; aj.: Graphviz and dynagraph — static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, Springer-Verlag, 2003, s. 127–148.

[2] Free Software Foundation: readelf manual page. 2018, [Online; navštíveno 22. 1. 2019].
URL http://man7.org/linux/man-pages/man1/readelf.1.html

[3] Gansner, E. R.: Using Graphviz as a Library (cgraph version). 2014, [Online; navštíveno 21. 1. 2019].
URL https://graphviz.gitlab.io/_pages/pdf/libguide.pdf

[4] North, S. C.; Gansner, E. R.: Cgraph Tutorial. 2014, [Online; navštíveno 21. 1. 2019].
URL https://graphviz.gitlab.io/_pages/pdf/cgraph.pdf

[5] Pinaev, D.: Qt5 Node Editor. 2017, [Online; navštíveno 21. 1. 2019].
URL https://github.com/paceholder/nodeeditor

[6] The Qt Company Ltd.: Qt Documentation – Canvas QML Type. 2019, [Online; navštíveno 15. 5. 2019].
URL https://doc.qt.io/qt-5/qml-qtquick-canvas.html

[7] TIS: *Executable and Linking Format (ELF) Specification*. [Online; navštíveno 17. 1. 2019].
URL https://refspecs.linuxbase.org/elf/elf.pdf

[8] Weyergraf, S.: HT Editor Documentation. [Online; navštíveno 22. 1. 2019].
URL http://hte.sourceforge.net/readme.html