VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

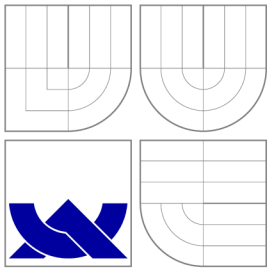
EMULÁTOR BYTE KÓDU JAZYKA JAVA VHODNÝ PRO DETEKCI A ANALÝZU MALWARE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

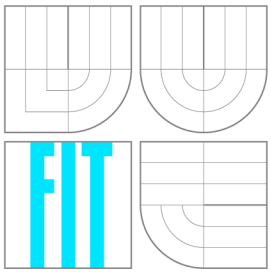
AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ KUBERNÁT

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EMULÁTOR BYTE KÓDU JAZYKA JAVA VHODNÝ PRO DETEKCI A ANALÝZU MALWARE

JAVA BYTE CODE EMULATOR SUITABLE FOR MALWARE DETECTION AND ANALYSIS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ KUBERNÁT
AUTHOR

VEDOUCÍ PRÁCE Doc. Ing.,Dipl.-Ing. MARTIN DRAHANSKÝ, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Cílem této práce bylo vytvořit virtuální stroj, emulující spouštění programů napsaných v programovacím jazyce Java, který by byl vhodný pro analýzu a detekci malware. Emulátor je schopen zjistit argumenty zneužitelných metod standardních tříd jazyka Java, pořadí volání těchto zneužitelných metod a také vlastní provedení testované aplikace. Celková funkcionalita byla otestována na vhodných příkladech, na kterých proběhlo i vlastní měření. V závěru práce je popsáno testování celkového řešení, kde jsou také uvedeny tabulky a grafy pro lepší znázornění dosažených výsledků.

Abstract

The goal of this thesis is to create a virtual machine that emulates a running programs written in Java programing language, which would be suitable for malware analysis and detection. The emulator is able to detect arguments of exploitable methods from Java standard classes, the order of calling these exploitable methods and also execution the test application. Overall functionality was tested on appropriate examples in which held its own measurements. At the end of the paper we describe testing of the emulator, which also contains tables and graphs for better results visualization.

Klíčová slova

virtuální stroj, Java Virtual Machine, virtuální stroj jazyka Java, byte kód, emulátor, emulace, škodlivý software, malware, soubory typu .class, detekce malware, analýza malware

Keywords

virtual machine, Java Virtual Machine, byte code, emulator, emulation, malicious software, malware, .class files, malware detection, malware analysis

Citace

Tomáš Kubernát: Emulátor byte kódu jazyka Java vhodný pro detekci a analýzu malware, diplomová práce, Brno, FIT VUT v Brně, 2013

Emulátor byte kódu jazyka Java vhodný pro detekci a analýzu malware

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. Ing., Dipl.-Ing. Martina Drahanského, Ph.D. Další informace mi poskytli pánové Ing. Jaroslav Nix a Ing. Jan Borůvka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Kubernát
21. května 2013

Poděkování

Děkuji vedoucímu své diplomové práce panu Doc. Ing., Dipl.-Ing. Martinu Drahanskému, Ph.D. za cenné rady, připomínky a celkové vedení diplomové práce. Dále bych chtěl mnohokrát poděkovat panu Ing. Jaroslavu Nixovi za poskytnutí možnosti věnovat se tomuto zajímavému tématu a panu Ing. Janu Borůvkovi za cenné rady během celého vývoje. V neposlední řadě patří obrovské díky celé mojí rodině za mentální i materiální podporu během celého dlouhého studia.

Obsah

Kapitola 1

Úvod

Bez internetu a síťové komunikace si dnešní život dokáže představit už jen málokdo. Slouží ke sdělování a vyhledávání informací, komunikaci, můžeme díky němu platit či seznamovat se. Je prakticky spojen se vším, čím se dnes člověk zabývá. Slouží každý den člověku jako prostředek komunikace, práce a také jako zdroj zábavy.

Globální komunikace by bez internetu byla ochuzena o významný komunikační kanál, díky kterému se prakticky zkrátily vzdálenosti napříč celou zeměkoulí. Útočníci se snaží napadnout koncové stanice uživatelů, různé aplikační servery nebo aktivní síťové prvky a zařízení zajišťující provoz sítě. Dalšími zařízeními, které se útočníci snaží infikovat jsou chytré telefony a tablety. Tato výkonná zařízení spolu s vysokorychlostními mobilními datovými připojeními mají obrovskou výpočetní sílu, kterou je také důležité chránit. Všechny tyto útoky jsou většinou vedeny prostřednictvím internetu či jiné datové síťě, ke které jsou napadané systémy připojeny.

Přes rozličná zařízení, platformy a procesorové architektury by bylo pro útočníka nevýhodné vytvářet několik verzí svého škodlivého programu. Proto útočníci často sahají k variantě programovacího jazyka, která by jim umožnila vytvořit jeden program a spustit ho na více systémech se stejným výsledkem.

Právě z tohoto důvodu je velice dobrou volbou programovací jazyk Java. Virtuální stroje provádějící programy napsané v tomto programovacím jazyku jsou portovány prakticky přes celé spektrum procesorových architektur i platforem operačních systémů. Tato práce se proto zaměřuje na detekci a analýzu chování spustitelných programů napsaných v programovacím jazyku Java. Zde se nezabývám zpracováním *.java* souborů, ale zpracovávám byte kód přeložený z těchto zdrojových souborů, který je uložen v souborech typu *.class*.

První kapitola čtenáři velice lehce osvětlí co je to vlastně škodlivý software a jaké je jeho základní dělení.

Dále je popsán samotný programovací jazyk *Java*, jeho architektura, struktura spustitelného balíčku a vytváření jednotlivých *.class* souborů.

Náplní mojí práce je prakticky tvorba *Java Virtual Machine*, proto další kapitola popisuje jeho architekturu a datové oblasti sloužící pro správný běh spouštěných programů. Stranou nezůstal ani popis některých již implementovaných virtuálních strojů.

V další kapitole je podrobně popsána struktura *.class* souboru. Jsou do detailů vysvětleny veškeré datové struktury *.class* souboru a je uveden i jejich účel.

Kapitola *Návrh a popis implementace* 4 shrnuje celkový návrh jednotlivých částí emulátoru, kterými jsou načítání dat do emulátoru, zpracování *.class* souborů, návrh realizace *heap*-u, samotného provádění programů či algoritmu pro automatickou správu paměti. Není

zapomenuto ani na návrh vlastního způsobu detekce a analýzy spouštěných programů.

V implementační části 5 jsou podrobně vysvětleny detaily implementace všech částí popsané v předchozí kapitole. Uvedeny jsou důležité datové položky, jejich význam a metody, které s nimi pracují.

Detailní popis jednotlivých fází testování je následně uveden v kapitole 6. Pro každý typ testování je uveden důvod, proč bylo testování provedeno právě tímto způsobem, jsou uvedeny parametry spuštění i výsledky, které byly jednotlivými testováními dosaženy.

Závěrečná kapitola nakonec shrnuje veškerou práci, která stála za vytvořením emulátoru. Jsou popsány všechna úskalí, problémy a úspěchy doprovázející moji činnost během celého vývoje. V neposlední řadě je vyzdvihnut celkový přínos mnou implementovaného emulátoru a jeho plusy či nedostatky směrem k detekci a analýze škodlivého softwaru.

Kapitola 2

Škodlivý software

Tato kapitola se lehce zabývá, co to vlastně malware je a jaké jsou jeho druhy. Jelikož se tento druh softwaru vyvíjí překotným způsobem, jsou vysvětleny pouze základní druhy škodlivého softwaru a virů obecně.

2.1 Co je to malware?

Malware[16] je zkratka pro malicious software, tedy škodlivý software. Jak už je z názvu patrné, tento druh programů provádí nějakou škodlivou činnost většinou bez vědomí uživatele napadeného systému.

Různé druhy malwaru jsou naprogramovány proto, aby dělaly odlišnou škodlivou činnost. Některé druhy jsou vytvořeny k omezení či ukončení běhu nějakého programu. Jiné jsou napsány za účelem převzetí kontroly útočníkem nad napadeným systémem nebo provedením určitých funkcí nad infikovaným systémem a to vše za účelem nějakého zisku pro útočníka. Další skupinou mohou být takové programy, které zjišťují o uživateli napadeného systému důvěrné informace, jako jsou např. uživatelské jména a hesla k účtům na internetu, čísla kreditních karet, atd.

Napadení systému nějakým škodlivým softwarem může mít za následek i finanční újmu pro provozovatele napadeného systému, který může vynaložit nemalé finanční prostředky na odstranění těchto programů nebo nápravu problémů způsobených útočníkem.

Základními typy malware jsou[16] : *rootkit*, *červ*, *trojský kůň*, *logická bomba* a *virus*.

V této práci se budu soustředit výhradně na malware naprogramovaný v programovacím jazyce *Java*, který je popsán v následující kapitole 3.

2.2 Typy malware

V zásadě rozlišujeme několik základních typů malware. Jsou to rootkity, internetoví červi, trojští koně, viry a logické bomby[16]. Ve většině případů malwaru se dnes nejedná o zástupce jenom jedné kategorie, ale o zástupce více kategorií zároveň.

2.2.1 Rootkit

Rootkit se nějakým vhodným způsobem snaží zakrýt svoji činnost a existenci před systémovým administrátorem i systémem jako takovým. Vhodně upravuje např. volání jádra tak, aby skryl sebe sama z výpisu běžících procesů či aby po sobě uklidil stopy v podobě otevřených popisovačů souborů nebo obsazených socketů.

2.2.2 Virus

Pokud je spuštěn tento typ malwaru, tak zkopíruje sama sebe do dalšího spustitelného souboru, čímž se dále rozšiřuje. Když virus uspěje, napadený program je prohlášen jako infikovaný. Jestliže je tento infikovaný program spuštěn, spustí se tím pádem i virus v něm obsažený a svojí činností infikuje další, ještě neinfikovaný, spustitelný program.

Tímto je zajištěno rozšiřování viru.

2.2.3 Červ

Červ je velice podobný viru. Je rovněž sebereplikující se, ale ne infikováním nějakého spustitelného programu. Tento typ vystupuje jako samostatná aplikace a jeho replikace nezávisí na použití nějakého dalšího spustitelného souboru jak je tomu u viru. Červ se totiž šíří sám od sebe, nejčastěji použitím počítačové sítě.

2.2.4 Trojský kůň

Tento typ malwaru na první pohled vypadá jako obyčejný neškodný program, který provádí to, co po něm uživatel chce. Na druhý pohled ale skrytě vykonává nějakou škodlivou činnost.

Standardní činností trojského koně je získávání hesla od uživatelů, které následně nějakým vhodným způsobem posílá svému tvůrci.

2.2.5 Logická bomba

Logická bomba je druh malware mající za úkol způsobit výrazné škody, např. zaměstnavateli, který propustil zaměstnance. Zaměstnanec se takto chce pomstí svému zaměstnavateli za propuštění.

Tento typ malwaru je spuštěn nějakou předem danou podmínkou. Pro předchozí příklad by to mohla být podmínka zmizení zaměstnance z výplatní listiny a akce může způsobit vymazání dat z firemních file serverů, restart serverů nebo nějakou další škodlivou činnost.

Logická bomba může vystupovat buď jako samostatný plnohodnotný program, nebo může být součástí nějakého většího celku, kde se ovšem velice obtížně hledá, pokud ji chceme odstranit. Většinou se jedná o malý a nenápadný program, kterého je těžké si všimnout.

Kapitola 3

Programovací jazyk Java a jeho virtuální stroj

3.1 Historie programovacího jazyku Java

V roce 1991 James Gosling se svým týmem nazvaným „*Green Team*“ začal pracovat na projektu, který měl v té době pracovní název „*Oak*“. Cílem bylo vytvořit virtuální stroj a programovací jazyk se syntaxí podobnou jazyku *C*, který by byl více jednotný a snazší než programovací jazyk *C++*.

První implementace programovacího jazyka *Java* byla zveřejněna v roce 1995 a kladla si za cíl heslo „*Write Once, Run Anywhere*“, tedy „*Napiš jednou, spusť kdekoliv*“. Dalším z cílů bylo spustit programy napsané v jazyce *Java* bez problémů všem uživatelům na jakékoliv platformě.

Při tvorbě jazyka *Java* se tvůrci snažili dosáhnout následujících pěti cílů:

- měla by být poskytnuta plná objektová orientace
- měla by být zajištěna proveditelnost programu na všech platformách
- měla by být poskytnuta vestavěná podpora pro užívání počítačových sítí
- měla by být navržena pro zabezpečené spouštění kódu ze vzdálených zdrojů
- měla by být snadno k použití vybráním dobrých částí z ostatních objektově orientovaných jazyků.

3.2 Jak Java ve skutečnosti funguje

Programy napsané v programovacím jazyce Java lze editovat prakticky v jakémkoliv textovém editoru. Jedná se o obyčejné textové soubory, které jsou překladačem zkompilovány do byte kódu spustitelného ve virtuálním stroji.

Do byte kódu, kterému rozumí Java Virtual Machine, nemusí být kompilovány pouze zdrojové soubory programovacího jazyku Java, ale mohou to být i zdrojové soubory jiných programovacích jazyků. O to, aby byly tyto soubory správně převedeny do byte kódu, se musí postarat dané kompilátory. Z jazyků, které lze takto zkompilovat do byte kódu spustitelného v Java Virtual Machine, můžeme zmínit např. *Clojure*[19], *Scala*[25][24], *Groovy*[17],

JRuby[18], *Jython*[22] a dokonce lze do byte kódu Javy přeložit i zdrojové soubory programovacího jazyka *C*.

Po zkompilování *Java* aplikace do byte kódu(sada *.class* souborů) máme již program, který je spustitelný na všech platformách, na kterých je implementovaný patřičný virtuální stroj řídící se podle specifikace *Java Virtual Machine*[12] společnosti *Oracle*[13].

Další fází je už vlastní spuštění aplikace. Virtuálním strojem jsou načteny všechny potřebné soubory a *Java Virtual Machine* už se postará o to, aby byly instrukce obsažené v *.class* souborech korektně interpretovány. Vše je lépe vidět na obrázku 3.4.

Kompilátorem je vždy vygenerován byte kód, který vždy dělá to stejné. A jelikož je vlastní aplikace jen souborem několika *.class* souborů, stačí tuto aplikaci interpretovat ve virtuálním stroji na platformě jaká se nám zrovna hodí, aniž bychom museli znova program překládat.

Tímto je dosažena portabilita mezi platformami a i heslo Jamese Goslinga „*Napiš jednou, spusť všude*“.

3.2.1 Kompilované programovací jazyky

Kompilované programovací jazyky mají oproti jazykům jako je *Java* jednu nespornou výhodu, kterou je rychlost jakou jsou výsledné programy prováděny procesorem. Toho je dosaženo tím, že zdrojové kódy kompilovaných jazyků jsou překládány přímo do strojového kódu konkrétního procesoru. Strojový kód je následně zaveden do paměti a přímo prováděn procesorem. Díky této úzké vazbě je dosažena výše zmíněná rychlost kompilovaných jazyků.

Tak jako vše, mají i kompilované programovací jazyky řadu nevýhod. Jednou z nich je portabilita. Programy přeložené z těchto zdrojových souborů nejsou vůbec přenositelné mezi operačními systémy či procesorovými architekturami. Tento fakt je dán tím, že jsou vždy kompilovány do specifické podoby co nejvíce vyhovující cílovému operačnímu systému nebo procesoru, na kterém program poběží. Proto nelze program jako takový(v binární podobě) spustit na odlišné platformě. Musí se vzít jeho zdrojové kódy, znovu je přeložit pro další konkrétní platformu a tím získat spustitelný binární soubor.

Mezi kompilované jazyky patří například *C*, *C++* nebo *Pascal*.

Způsob vytvoření spustitelného programu ze zdrojového kódu je znázorněn na obrázku 3.1.

3.2.2 Interpretované jazyky

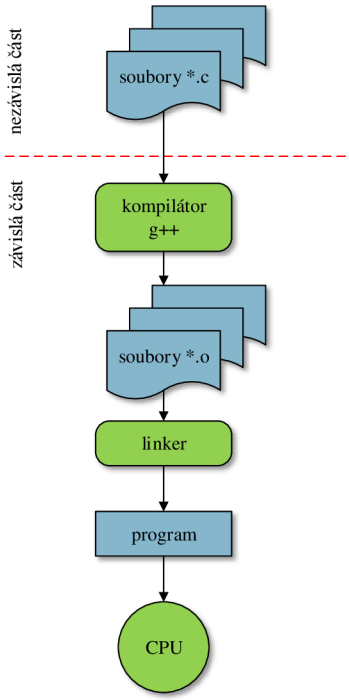
Interpretované programovací jazyky se liší od kompilovaných tím, že jejich kód není prováděn přímo procesorem, ale je interpretován. Proto musí být vždy přítomen interpret, který interpretaci provede. Zdrojové kódy těchto jazyků jsou lépe přenositelné napříč procesorovými architekturami i operačními systémy. Tato portabilita je ale vykoupena nižší rychlostí [7].

Pro jednoduché úkony jakými je například zpracování textů jsou ale vhodnou volbou, protože se programy řešící tyto problémy dají jednodušeji implementovat.

3.2.3 Kompilované i interpretované jazyky

Nejvíce nás ale bude zajímat tato skupina programovacích jazyků, která leží mezi oběma předešlými kategoriemi, protože do této skupiny patří programovací jazyk *Java*.

Zde se ze zdrojových souborů kompilací vytvoří byte kód, který je následně interpretován patřičným interpretem.



Obrázek 3.1: Způsob kompilace a spuštění programů napsaných v programovacím jazyku *C++*.

V případě jazyka *Java* se o interpretaci stará tzv. *Java Virtual Machine*, jehož jednotlivé části budou popsány v následujících kapitolách.

Zdrojový kód jazyka *Java* může být, pokud si to programátor přeje, překládán přímo do strojového kódu daného procesoru. O tuto kompilaci se stará tzv. *just-in-time* kompilátor. Takto přeložen může být buď celý zdrojový soubor nebo jeho části na začátku, nebo jeho části v průběhu interpretace.

Nutno podotknout, že se málokdy překládají *just-in-time* kompilátorem přímo zdrojové kódy v jazyce *Java*. Vetšinou se do strojového kódu dané platformy převádí již jednou přeložený *.class* soubor.

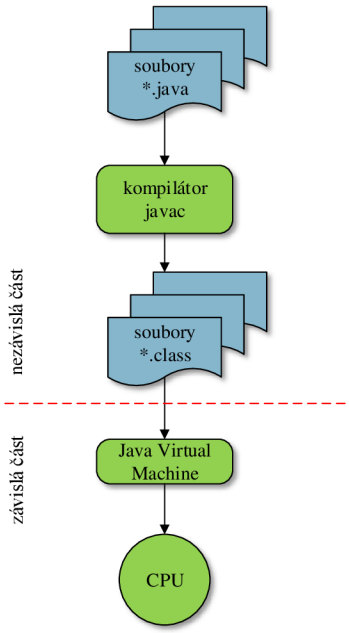
3.2.4 Práce s objekty

Jedním z hlavních rysů programovacího jazyka *Java* je jeho objektová orientace. Prakticky vše vystupuje v tomto jazyce jako objekt. Dokonce i jednoduché datové typy, jako jsou např. celočíselné konstanty, lze obalit tzv. *wrapper* třídou, čímž z nich vytvoříme objekty.

Instance třídy, jak se objektům taktéž říká, jsou uloženy vždy na *heap*-u a ve vlastním zdrojovém kódu se s nimi nepracuje přímo, ale přes reference na objekt.

Každá aplikace v programovacím jazyce *Java* vytváří velké množství objektů, které mezi sebou interagují vyvoláváním statických nebo nestatických metod. S využitím těchto interakcí program může vykonat rozmanité úkoly např. spustit animaci, vytvořit grafické uživatelské rozhraní či komunikovat po síti.

Pokud nějaký objekt dokončí svoji práci a už není potřeba nebo na něj nevede v programu žádná reference, tak je z paměti odstraněn a místo, které na *heap*-u zabíral je uvol-



Obrázek 3.2: Způsob kompilace a spuštění programů napsaných v programovacím jazyku *Java*.

něno. Toto místo může být následně použito k alokaci nově vznikajícího objektu.

Datovou oblast *heap* vždy spravuje nějaký automatický správce paměti tzv. *garbage collector*, který uvolní již nepotřebnou paměť a prohlásí ji za volnou. Programátor v jazyce *Java* nikdy explicitně neodstraňuje již nepotřebné paměti a ani k tomu nemá prostředky. Může maximálně vyvolat správce paměti, který celý *heap* uklidí.

Vybrané algoritmy a implementace budou popsány v následujících kapitolách. Pro lepší pochopení datových oblastí virtuálního stroje, včetně *heap*-u, je k přečtení kapitola 3.6.

3.3 Struktura spustitelného archivu *.jar*

Aplikace napsané v programovacím jazyku *Java* jsou ve skutečnosti sadou souborů byte kódu tohoto jazyka. Pro lepší šíření těchto aplikací je použit program *zip*, který tuto sadu zabalí do jednoho samostatného balíčku. Výsledný balíček nemá ovšem koncovku *.zip*, jak by se mohlo zdát podle programu, který provádí balení, ale má koncovku *.jar* podle *java archive*.

Mimo *.class* souborů obsahuje *.jar* balíček i prostředky aplikace, jakými mohou být textové soubory, obrázky nebo zvuky.

Další veledůležitou součástí je soubor *MANIFEST.MF*, který je uložen v archivu ve složce *META-INF*. Tento soubor implicitně obsahuje informace o verzi souboru *MANIFEST.MF*, verzi použitého *JDK* a také nesmíme zapomenout na vstupní bod celé aplikace[15]. Ten je zde uveden ve formátu *Main-Class: MyPackage.MyClass*. Tento zápis nám říká, že se bude volat metoda *main* třídy *MyClass* z balíčku *MyPackage*.

Pokud *.jar* soubor plní funkci knihovny, tak nemusí obsahovat žádný vstupní bod - metodu *main* - v žádné ze svých tříd.

Do virtuálního stroje jsou ovšem načteny pouze soubory, které mají koncovku **.class*. Jestli se jedná o byte kód třídy se kontroluje podle koncovky souboru a podle prvních 4 bajtů, které obsahují tzv. *magic number*. Tato konstanta musí vždy obsahovat hodnotu *0xCAFE BABE*. Pokud ji neobsahuje, tak je při načítání souboru virtuálním strojem vyhozena výjimka *java.lang.ClassFormatError*.

Hlavička každého *.class* souboru je složena z již zmíněné magické konstanty a ze dvou čísel. Prvním z nich je *minor version* a druhým je *major version*. Obě dvě společně rozlišují verzi patřičného *.class* souboru.

Díky těmto dvěma číslům lze řadit verze *.class* souborů lexikograficky. Číslo verze je složeno následovně. Pokud M bude *major version* a m *minor version*, potom je číslo verze formátu M.m. Soubory typu *.class* tedy seřadíme například jako $1.5 < 2.0 < 2.1$

Podrobněji jsou jednotlivé části *.class* souboru popsány v kapitole 3.8.2.

3.4 Vytváření *.class* souborů

Obecně je známo, že z každého jednoho zdrojového souboru napsaného v jazyce *Java* se vygeneruje jeden jediný soubor typu *.class* příslušící překládanému zdrojovému souboru, ale není tomu docela tak. Soubor typu *.class* je generován pro každou třídu, která je uvedena v celém programu, a protože jeden zdrojový soubor jazyka *Java* může obsahovat více tříd, může být počet vygenerovaných *.class* souborů větší[11].

Tento fakt je velice důležitý, protože načítání a parsování *.class* souborů je první fází, kterou mnout vytvářený emulátor provádí. Je tedy nutné přesně vědět jak a co načíst, a jak to později v emulátoru použít.

Máme sedm způsobů, kterými lze vytvořit *.class* soubor, a které jsou popsány v následujících částech. Pro shrnutí ještě uvedu jejich seznam[10]:

- běžná veřejná třída (top-level class)
- běžná neveřejná třída
- vnitřní třída (inner class)
- lokální třída (local class)
- anonymní třída
- výčet (enumeration)
- rozhraní (interface)

3.4.1 Běžná veřejná třída

Běžná veřejná třída (top-level class) je taková třída, která má modifikátor *public*, tudíž je přístupná z vnějšku třídy a je ve svém vlastním *.java* souboru. Tento soubor je pojmenován stejně jako třída, kterou obsahuje, ale ovšem s koncovkou *.java*.

Pokud tedy budeme mít soubor *BeznaTrida.java*, ve kterém bude následující kód,

```
public class BeznaTrida
{

}
```

bude po překladu vygenerován pouze soubor *BeznaTrida.class* obsahující byte kód výše uvedeného programu. Tělo třídy mezi složenými závorkami může být i prázdné. Na tvorbu *.class* souborů to nemá žádný vliv. Na jeho obsah samozřejmě ano.

3.4.2 Běžná neveřejná třída

Je to normální třída v jazyce Java, ale oproti předešlému případu s tím rozdílem, že nemá modifikátor public a není ve svém vlastním souboru. Následující zdrojový kód bude opět uložen v souboru *BeznaTrida.java*.

```
public class BeznaTrida
{
  ...
  ...
  ...
}

class BeznaNeverejnaTrida
{
  ...
  ...
  ...
}
```

V tomto případě bude po přeložení souboru *BeznaTrida.java* adresář obsahovat dva *.class* soubory. Soubory *BeznaTrida.class* a *BeznaNeverejnaTrida.class*.

3.4.3 Vnitřní třída

Vnitřní třída (inner class) je třída, která je uvedena uvnitř jiné třídy. Může být obsažena i uvnitř jiné vnitřní třídy. Pokud má být třída nazvána jako vnitřní (inner), nesmí mít modifikátor *static*. Vnitřní statické třídy jsou nazývány *nested classes*. Na rozdíl od nestatických vnitřních tříd, nemají statické vnitřní třídy přístup ke členům třídy, která vnitřní třídu obaluje.

Opět mějme soubor *BeznaTrida.java*, ale tentokrát již s trochu složitějším zdrojovým textem.

```
public class BeznaTrida{
  // vnitřní třída
  class VnitrniTrida01{
  }

  // druhá vnitřní třída
  class VnitrniTrida02{
    // vnitřní třída uvnitř vnitřní třídy
    class VnitrniTrida03{
    }
  }
}
```

Zde je již situace o něco složitější než-li v předchozím případě. Jméno *.class* souboru je složeno ze jména hlavní třídy, která vnitřní třídu obsahuje, znaku $ a jména vnitřní třídy.

Proto bude výpis souborů daného adresáře obsahovat tyto čtyři *.class* soubory.

```
BeznaTrida.class
BeznaTrida$VnitrniTrida01.class
BeznaTrida$VnitrniTrida02.class
BeznaTrida$VnitrniTrida02$VnitrniTrida03.class
```

3.4.4 Lokální třída

Lokální třídy jsou třídy deklarované uvnitř nějakého funkčního bloku, nejčastěji metody. Pro lepší představu je zase uveden jednoduchý zdrojový soubor s názvem *BeznaTrida.java*.

```
public class BeznaTrida
{
  void metoda1()
  {
    class LokalniTrida1{}
    class LokalniTrida2{}
  }

  void metoda2()
  {
    class LokalniTrida3{}
    class LokalniTrida4{}
  }
}
```

Pravidla pro pojmenování *.class* souborů lokálních tříd jsou podobné jako u vnitřních tříd, ale s jedním malým rozdílem. Po jménu třídy, která obsahuje metody s lokálními třídami následuje opět znak $, za kterým je generovaný index a až za tímto indexem je uvedeno jméno lokální třídy.

Proto je po překladu programem *javac* pro kód z této části vygenerováno dohromady 5 souborů - *BeznaTrida$1LokalniTrida2.class*, *BeznaTrida$1LokalniTrida1.class*, *BeznaTrida.class*, *BeznaTrida$1LokalniTrida3.class* a *BeznaTrida$1LokalniTrida4.class*.

3.4.5 Anonymní třída

Tento druh třídy je pojmenován způsobem něco mezi pojmenováním vnitřní třídy a lokální třídy. Název *.class* souboru je totiž složen ze jména třídy, která danou anonymní třídu obsahuje, za tento název je umístěn obligátní znak $ a až po tomto znaku následuje generovaný index, který je současně i poslední částí názvu *.class* souboru.

Opět mějme soubor *BeznaTrida.java*, který obsahuje následující zdrojový kód.

Po překladu tohoto zdrojového kódu vzniknou tři soubory typu *.class* - *BeznaTrida.class*, *Trida.class* a pro anonymní třídu *BeznaTrida$1.class*.

Často se tento druh třídy používá při volání funkce, u které je vytvářený objekt jako parametr. Vše je opět vidět v ukázkovém zdrojovém kódu.

```
class Trida
{
}

public class BeznaTrida
{
  Trida anonTrida = new Trida(){};
}
```

3.4.6 Výčtový typ (enumeration)

Generování *.class* souborů pro výčtový typ probíhá naprosto stejně jako pro obyčejnou třídy. Každý výčet má svůj vlastní *.class* soubor pojmenovaný podle názvu výčtového datového typu.

3.4.7 Rozhraní (interface)

Stejné pojmenování jako pro výčtový datový typ platí i pro pojmenování *.class* souborů pro rozhraní. Tedy i zde má každé rozhraní svůj vlastní *.class* soubor pojmenovaný podle sebe.

3.5 Virtuální stroj - Java Virtual Machine

Java Virtual Machine je variantou jednoduchého zásobníkového procesoru. Stejně jako reálný procesor i *Java Virtual Machine* má několik datových oblastí a instrukce, které pracují nad těmito oblastmi a daty, která jsou v těchto oblastech obsažena. Virtuální stroj programovacího jazyka *Java* neví nic o vlastních pravidlech, syntaxi ani sémantice tohoto jazyka, ale ví přesně, jak zacházet s byte kódem přeloženým ze zdrojových souborů jazyka *Java*.

Architektura *Java Virtual Machine* obsahuje několik datových oblastí, které jsou potřebné pro správné provádění programů napsaných v *Javě*. Těmito oblastmi jsou *registr PC*, *zásobníky JVM*, *heap*, *oblast metod (method area)* a *runtime constant pool*.

Některé z výše uvedených datových oblastní jsou vytvořeny již při startu vlastního virtuálního stroje 3.6.1 a jsou z paměti odstraněny až při ukončení virtuálního stroje, zbylé datové oblasti jsou vytvářeny vždy pro každé vlákno 3.6.2 při jeho startu a ničeny při jeho ukončení.

V této kapitole jsem vycházel z oficiální reference *Java Virtual Machine*[2] společnosti *Oracle*[13].

3.6 Architektura Java Virtual Machine

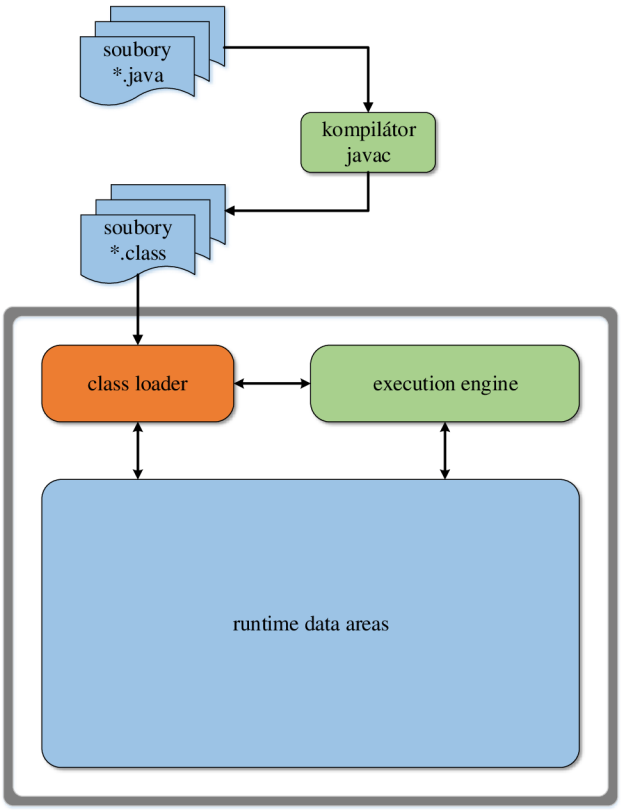
Práce *Java Virtual Machine* je nejlépe vidět na obrázku 3.4. Po přeložení zdrojových souborů programovacího jazyka *Java* vznikne kolekce *.class* souborů reprezentující vlastní spouštěný program. Jednotlivé *.class* soubory jsou postupně načítány pomocí *class-loader*-u, který se také stará o načítání dalších potřebných a odstraňování již nepotřebných *.class* souborů z virtuálního stroje. Vše se děje za běhu programu, kde dopředu nevíme, které balíčky ze standardního balíku jazyka *Java* budou potřeba.

Po načtení alespoň základního počtu *.class* souborů a naplnění datových struktur v oblastech *runtime data areas* 3.4 začíná vlastní provádění programu.

O toto provádění se stará *execution engine* 3.4, který interpretuje byte kód uložený v oblasti *method area* 3.6.1. Instrukce po instrukci je načtena z této oblasti a *execution engine* se stará o to, aby byly správně upraveny ostatní datové oblasti *JVM* obsahující data spuštěného programu. Zejména pak *Java Virtual Machine Stack* 3.6.2 skládající se z položek *Stack Frame*.

Všechny tři jednotky, které jsou zobrazeny na obrázku 3.4 vzájemně spolupracují. Kdyby tomu tak nebylo, tak by virtuální stroj nefungoval. *Execution engine* při vykonávání těl metod neví, jaké další třídy bude potřebovat. Proto musí komunikovat s *class-loader*-em, který jednotce *execution engine* poskytne *.class* soubory tříd, které jsou v danou chvíli potřeba.

Class-loader nově potřebné třídy načte a aktualizuje(doplní chybějící data) oblasti v *runtime data areas* odkud si je už *execution engine* načte a zpracuje.



Obrázek 3.3: Obecná architektura *Java Virtual Machine*[2].

3.6.1 Datové oblasti pro celý proces

Tyto datové oblasti jsou součástí *runtime data area* z obrázku 3.4 a jsou sdílené všemi vlákny spuštenými v rámci procesu.

Jedná se o *heap* 3.6.1, *method area* a *runtime constant pool*.

Heap

Heap je datová struktura, která slouží k ukládání vytvořených instancí jednotlivých tříd.

Objekty uložené na *heap*-u nejsou nikdy explicitně z paměti odstraňovány programátorem. Tento fakt je uveden přímo ve specifikaci programovacího jazyka *Java* i ve specifikaci *Java Virtual Machine*.

O veškeré uvolňování již nepotřebných objektů se stará automatický správce paměti tzv. *Garbage Collector*. To, jaký algoritmus pro *garbage collection* použít není ve specifikaci *JVM* explicitně uvedeno a je čistě na tvůrci *Java Virtual Machine*, kterou implementaci „uklízecího“ algoritmu použije. Toto závisí na účelu, na který bude výsledná implementace *JVM* použita. Stejné je to i s velikostí a flexibilitou *heap*-u. Tvůrce *JVM* si může zvolit jestli mít dynamickou nebo pevně danou velikost této datové struktury i její vlastní velikost. Velikost *heap*-u lze obvykle zadat při spouštění virtuálního stroje.

Co přesně a jakým způsobem *garbage collector* vykonává i jaké lze použít algoritmy pro správu paměti bude podrobněji vysvětleno v některé z následujících kapitol.

Přesná implementace *heap*-u je silně závislá na použitém algoritmu pro *garbage collection*, a proto se jí zde zatím zabývat nebudu. Vše bude napraveno v dalších částech textu.

Heap má ze všech částí *Java Virtual Machine* nejpodstatnější vliv na výkon *JVM*. Správnou volbou velikosti *heap*-u, algoritmu pro *garbage collecting* a jeho vhodnou implementací můžeme docílit výrazného zvýšení výkonu samotného virtuálního stroje. Pokud tedy chceme zvýšit výkon *JVM*, je *heap* ideální místo pro nějaké úpravy.

Method area

V této oblasti jsou uloženy struktury pro každou třídu načtenou pomocí *class-loaderu*-u, stejně tak je zde umístěn i *constant pool*, data metod a datových položek každé třídy. Jsou zde uloženy i speciální metody pro inicializaci instancí jednotlivých tříd a rozhraní. Ačkoli je *method area* logickou částí oblasti *heap*, není nijak spravována správcem paměti a nemusí být ani nutně spojitá.

Runtime constant pool

Je to reprezentace tabulky *constant pool*, která je uložena v *.class* souboru třídy nebo rozhraní. Obsahuje několik druhů konstant, které jsou známe již při překladu, i reference na metody a datové položky třídy, které jsou známé až při běhu programu.

Runtime constant pool je vytvořen, jakmile je načten patřičný *.class* soubor do virtuálního stroje.

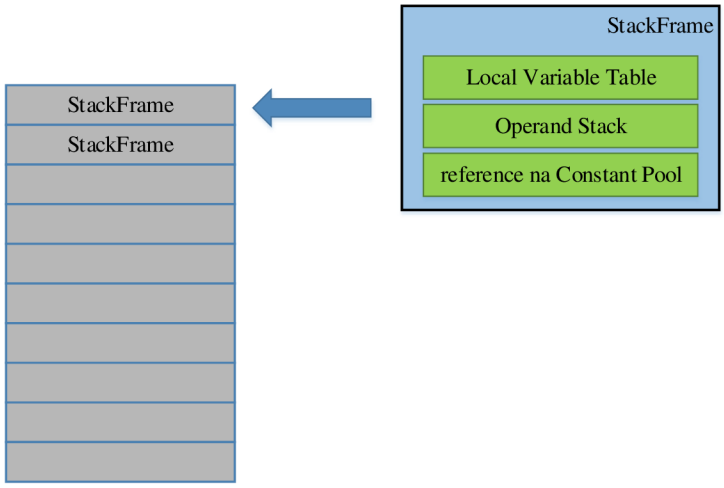
3.6.2 Datové oblasti vytvořené pro každé vlákno

Zásobník (*Java Virtual Machine Stack*)

Každé vlákno spuštěné v rámci jednoho běhu programu uvnitř *JVM* obsahuje vždy jeden zásobník, který je vytvořen při startu vlákna a odstraněn při jeho dokončení.

Skládá se z tzv. zásobníkových rámců (*Stack Frames*), které v sobě obsahují datové oblasti používané pro výpočet výsledků při běhu nějaké metody. Pro každou vyvolanou metodu se na tomto zásobníku vytvoří jeden *Stack Frame*, který reprezentuje vlastní běh této metody.

Při provádění těla dané metody virtuálním strojem jsou upravovány vždy datové oblasti patřičného zásobníkového rámce. Konkrétně se jedná o vrchol zásobníku daného vlákna, který se nazývá *TOS - Top Of Stack*.



Obrázek 3.4: Grafická reprezentace oblastí *JVM Stack*, *Stack Frame*, *Local Variable Table* a *Operand Stack*.

Pokud z nějaké metody vyvolávám metodu jinou, vytvoří se nový zásobníkový rámec, uloží se na vrchol zásobníku a začne se provádět nově vyvolaná metoda reprezentovaná nově vytvořeným rámcem.

Local Variable Table a *Operand Stack* jsou po vyvolání nové metody vždy prázdné a začínají se plnit až samým prováděním metody.

V tabulce *Local Variable Table* je nejprve uložen implicitní parametr *this*, za ním následují parametry volané metody a za nimi jsou uloženy lokální proměnné používané metodou. Velikost této tabulky je známá již při překladu, tudíž lze tuto tabulku napevno dynamicky alokovat.

Operand Stack daného zásobníkového rámce slouží pro uložení mezivýsledků při výpočtech metody a navíc, což je neméně důležité, slouží pro předání parametrů do nově vyvolávané metody nebo při vracení dat zpět do volající metody. Velikost tohoto zásobníku je opět známá při překladu, proto je jeho velikost pevně nastavena dopředu.

Registr PC

Java Virtual Machine podporuje běh více vláken najednou a pro každé vlákno musí existovat *program counter registr*, který obsahuje adresu právě prováděné instrukce. Pokud je ovšem prováděna nativní metoda, je obsah tohoto registru nedefinován.

Velikost tohoto registru musí být taková, aby zvládla pojmout návratovou adresu v rámci virtuálního stroje nebo ukazatel na platformě, na které je spouštěna nativní metoda.

3.7 Některé implementace virtuálních strojů

3.7.1 HotSpot

Tento virtuální stroj je vyvíjen přímo společností *Oracle*[13]. Na tomto virtuálním stroji pracovalo velké množství lidí už přes 10 let, počet zdrojových a hlavičkových *C/C++* souborů se blíží k 1.500 a dohromady dávají cca 250.000 řádků zdrojového kódu.

Obsahuje *class-loader*, interpret byte kódu, dva runtime kompilátory z byte kódu do nativních instrukcí, sadu vysokovýkonnostních knihoven pro synchronizaci či více jak 3 automatické správce paměti[6].

Java Standard Edition Platform obsahuje dvě implementace virtuálního stroje *HotSpot - Java HotSpot Client VM* a *Java HotSpot Server VM*[9]. Obě verze jsou si velice podobné, přesto se ale liší v detailech, které vyplývají ze způsobů jejich použití.

Java HotSpot Client VM

Tento virtuální stroj je vyladěn pro co možná nejlepší výkonnost při spouštění aplikací v klientském prostředí, kde se klade důraz na rychlé spouštění aplikací a malé zanechávání stop v paměti po skončení běhu programu.

Kompilátor pro tento virtuální stroj se nesnaží dělat výraznější optimalizace zdrojového kódu, proto je programy v tomto případě spouští rychleji. Díky tomuto faktu je tento typ virtuálního stroje vhodnější pro spouštění programů s grafickým uživatelským rozhraním.

Java HotSpot Server VM

Java HotSpot Server VM je maximálně uzpůsoben pro běh na serverech, kde je důležité aby program rychle běžel a měl výbornou odezvu. Dalším faktorem je i to, že programy spouštěné na serveru nebývají často restartovány, ale běží v kuse dlouhou dobu. Zde není potřeba, aby se program rychle spouštěl jako u verze přizpůsobené pro klienty, ale aby byl běh stabilní a rychlý.

Automatická správa paměti v HotSpot VM

Obvykle je automatická správa paměti považována za velmi náročnou snižující výkon celého virtuálního stroje. U virtuálního stroje *HotSpot* tomu tak není. Použitím mnoha optimalizací bylo dosaženo stejného nebo lepšího výkonu než mají jazyky, u kterých se o alokaci a dealokaci použité paměti musí starat explicitně sám programátor.

Virtuální stroj *HotSpot* obsahuje vícero automatických správců paměti a jsou používány podle toho, kde je *HotSpot* spuštěn. Jsou zde implementovány správci paměti pro rychlé spouštění programů i správci, které udržují uklizený *heap* i při velmi dlouhém běhu, kde jsou úniky paměti a fragmentace těžko předvídatelné.

Kolektory používané v *HotSpot*-u jsou naprosto precizní. Nehrozí zde žádné úniky paměti, veškerá paměť prohlášená za volnou je dostupná, veškeré objekty jsou relokovány, což má za následek prakticky žádnou fragmentaci a zvýšení lokality dat na *heap*-u.

Ve virtuálním stroji *HotSpot* jsou použity následující formy automatických správců paměti, které budou podrobně vysvětleny v některé z následujících kapitol.

- *Generational Copying Collection*
- *Parallel Young Generation Collector*

- *Mark-Compact Old Object Collector*
- *Mostly Concurrent Mark-Sweep Collector*
- *Parallel Old Generation Collector*

Jedinou větší nevýhodou tohoto virtuálního stroje je jeho použití na menším počtu procesorových architektur a to konkrétně na *x86*, *x86_64(AMD64)*, *Itanium*, *Sparc* a *UltraSparc*.

3.7.2 JamVM

Virtuální stroj *JamVM*[5][8][14] je jednou z mnoha open source variant *JVM*. Za cíle si klade rychlost, paměťovou nenáročnost a malou velikost. Z těchto důvodů se velice dobře hodí pro různá vestavěná zařízení nebo pro mobilní platformy. Neztratí se ovšem ani při použití na pracovních stanicích.

Narozdíl od virtuálního stroje *HotSpot* společnosti *Oracle*[13] nedisponuje tolika sofistikovanými funkcemi jako *HotSpot*, o což se ani nikdy tvůrci *JamVM* nikdy nesnažili. Díky tomuto faktu se podařilo dosáhnout tak vynikající velikosti.

JamVM je napsán převážně v jazyku *C*, ale časově kritické části jsou implementovány přímo ve strojovém kódu procesoru. Díky tomuto faktu je velice dobře portovatelný na nejrůznější procesorové architektury, samozřejmě je k tomu potřeba určité práce na zdrojových kódech.

Důležitou částí podporovaných procesorových architektur tvoří procesory *ARM* a *MIPS*, což je dáno nynejším masovým rozšiřováním tabletů a chytrých telefonů, které jsou většinou postaveny na procesorech *ARM*. Dalšími architekturami jsou např. *x86*, *x86_64* či *PowerPC* nebo *Sparc*. Z podporovaných operačních systému potom *Linux*, *BSD*, *MacOS* nebo *Solaris*.

JamVM používá *Mark & Sweep* automatického správce paměti, který je lehce modifikován. Může běžet ve vlastním vlákně a buď synchronně, nebo asynchronně.

Pro přístup k objektům uložených na *heap*-u neslouží tabulka nepřímých referencí, ale objekty jsou referencovány přímo odkazy na *heap*. Všechny výše popsané přístupy mají vliv na celkový výkon a minimalistické nároky.

V této části jsem vycházel z [5][8] a [14].

3.8 Soubor typu *.class* a jeho struktura

3.8.1 Datové typy *constant pool*-u

Tak jako každá datová struktura, má i *constant pool* několik datových typů, které využívá k ukládání dat. Nejprve si tedy vysvětlíme datové typy a poté i podrobně vlastní strukturu *.class* souboru.

Pro uložení čísel slouží v této struktuře datové typy *u1*, *u2* a *u4*. Typ *u1* má delku 8 bitů, typ *u2* je dlouhý 16 bitů a typ *u4* zabere v *.class* souboru 32 bitů.

Datové typy *cp_info*, *field_info*, *method_info* a *attribute_info* si můžeme opět představit jako struktury zapsané v jazyce *C*. Tyto složitější typy budou podrobněji popsány v následujících částech.

Pro lepší představu budou v tomto textu všechny složitější datové struktury zobrazeny ve stylu, který je známý z programovacího jazyka *C*.

V této kapitole jsem čerpal z oficiální specifikace virtuálního stroje společnosti Oracle[13] a to konkrétně části, kde je detailně popsán formát *.class* souboru.[3].

Datový typ *cp_info*

Tento datový typ reprezentuje obecný datový formát všech položek obsažených v *constant pool*-u.

```
cp_info
{
    u1 tag;
    u1 info[];
}
```

Každý záznam v tabulce *constant pool* musí začínat 8-bitovou hodnotou *tag*, která udává druh záznamu. Obsah pole bytů *info* se liší v závislosti na hodnotě položky *tag*. Možné hodnoty jsou znázorněny v tabulce 3.4 a popis jednotlivých typů je uveden v kapitole 3.9.

Datový typ *method_info*

Každá metoda třídy, která je tím pádem obsažena i v *.class* souboru, je popsána následující datovou strukturou. Takto popsány jsou i metody pro inicializaci objektů i konstruktory.

Obsahem *.class* souboru nemohou být dvě metody, které mají zároveň stejný název i signaturu metody.

```
method_info
{
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Položka *access_flags* slouží pro rozhodnutí o přístupových právech a vlastnostech dané metody. Může nabývat hodnot zobrazených v tabulce 3.1.

Tabulka 3.1: Přístupová práva používaná pro metody[3].

Název	Hodnota	Popis
ACC_PUBLIC	0x0001	metoda je přístupná z vnějšku balíčku
ACC_PRIVATE	0x0002	metoda přístupná pouze zevnitř definující třídy
ACC_PROTECTED	0x0004	metoda přístupná i z podtříd
ACC_STATIC	0x0008	jedná se o třídní metodu
ACC_FINAL	0x0010	metoda nemůže být přetížena
ACC_SYNCHRONIZED	0x0020	vyvolání této metody je hlídáno monitorem
ACC_BRIDGE	0x0040	metoda je generována kompilátorem
ACC_VARARGS	0x0080	metoda s variabilním počtem parametrů
ACC_NATIVE	0x0100	metoda implementována v jiném jazyce než *Java*
ACC_ABSTRACT	0x0400	není implementováno její tělo
ACC_STRICT	0x0800	mód pro počítání v plovoucí desetinné čárce je *FP-strict*
ACC_SYNTHETIC	0x1000	metoda se nenachází ve zdrojovém kódu

Záznamy *name_index* a *descriptor_index* musejí být validními indexy do *constant_pool*-u a musejí referencovat struktury typu *CONSTANT_Utf8_info*. Hodnoty těchto řetězcových konstant reprezentují jméno a signaturu metody.

Pole prvků *attribute_info attribute* obsahuje informace o atributech dané metody. Význam jednotlivých atributů bude vysvětlen v některé z dalších kapitol. Velikost tohoto pole je známá už při kompilaci a udává jí hodnota *attributes_count*.

Nejdůležitějším atributem metody je atribut s názvem *Code*. Obsahuje totiž vlastní tělo metody, které bude provedeno virtuálním strojem. Každá metoda může mít pouze jeden takto pojmenovaný atribut. Výjimku tvoří metody, které jsou obsaženy v nějakém rozhraní. Takovéto metody nesmí mít implementováno žádné tělo, a tedy nemají žádný atribut pojmenovaný *Code*. Pokud metoda v rozhraní tělo má, nebude zdrojový soubor přeložen do bajt kódu a překlad skončí chybou.

Atribut *Code* obsahuje instrukce bajt kódu, kterému rozumí každý virtuální stroj splňující specifikaci[2]. Správná práce s obsahem tohoto atributu je nutná pro vlastní průběh emulace a zabírá velkou část mojí dosavadní práce. Proto bude podrobně vysvětlena v dalších částech diplomové práce včetně popisu jednotlivých instrukcí.

3.8.2 Struktura *.class* souboru

Každý soubor typu *.class* obsahuje sekvenci bytů, která reprezentuje přeložený kód v jazyce Java. Tento byte kód má vždy následující formu, kterou si můžeme představit jako strukturu zapsanou v programovacím jazyce *C*.

```
ClassFile
{
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

u4 magic

Čtyř-bytová položka *magic* slouží pro identifikaci *.class* souboru. Musí vždy obsahovat hodnotu 0xCAFEBABE, což je kontrolováno už při načítání *.class* souboru virtuálním strojem. Pokud načítaný soubor neobsahuje na svém začátku tuto konstantu, není virtuálním strojem načten a je vyvolána výjimka *java.lang.ClassFormatError*.

u2 minor_version

Položka *minor_version* představuje minor číslo verze tohoto *.class* souboru. Spolu s číslem, které je uvedeno v *u2 major_version* udává přesnou verzi uvedeného *.class* soubor.

u2 major_version

Pro uložení hlavního čísla verze *.class* souboru slouží záznam *major_version*. Jak již bylo zmíněno výše, *major_version* spolu s číslem *minor_version* reprezentuje verzi každého *.class* souboru.

Rozlišit verzi každého souboru je důležité, protože během vývoje programovacího jazyka Java bylo potřeba rozlišovat, ve které verzi je napsán konkrétní program, protože některé změny v jedné verzi nebyly zpětně kompatibilní s verzemi staršími.

Proto se může chovat virtuální stroj jinak v závislosti na verzi načteného *.class* souboru.

Přesná interpretace čísla verze již byla popsána v kapitole 3.3.

u2 constant_pool_count

Záznam *constant_pool_count* obsahuje počet položek v tzv. *constant_pool*-u. Jak bude vysvětleno dále v textu, *constant pool* je prakticky nejdůležitější částí vlastního *.class* souboru.

Co může být trochu matoucí je, že hodnota v položce *constant_pool_count* znamená počet položek v *constant pool*-u + 1. Toto je dáno tím, že v *constant pool*-u neprobíhá indexování od 0, ale od 1.

Proto je index do *constant pool*-u validní pouze tehdy, pokud je v intervalu od 1 do *constant_pool_count* - 1.

Jelikož má tento záznam 16 bitů, bude maximální počet položek v *constant pool*-u 65.536.

cp_info constant_pool [constant_pool_count]

Tato část obsahu byte kódu *.class souboru* je prakticky jeho nejdůležitější částí. Jak už je z názvu patrné, obsahuje veškeré konstanty daného souboru. Tato část je tak důležitá, že jí je dále věnována samostatná podkapitola 3.9.

u2 access_flags

Položka *access_flags* určuje, jakým způsobem se k dané třídě bude přistupovat. Obsahuje totiž informace o právech přístupů k této třídě nebo rozhraní. Je používána jako bitová maska a její hodnoty jsou vidět v tabulce 3.2.

u2 this_class

Hodnota této proměnné musí být validním indexem do *constant_pool*-u a záznam na tomto indexu musí být typu *CONSTANT_Class_info*. Z této položky jsme schopni zjistit celé jméno třídy nebo rozhraní definované tímto *.class* souborem.

u2 super_class

Zde se opět jedná o index do tabulky *constant_pool*. Jeho hodnota musí být buď platným indexem do *constant pool*-u nebo 0.

Tabulka 3.2: Přístupová práva pro třídu a rozhraní[3].

Název	Hodnota	Popis
ACC_PUBLIC	0x0001	třída je public, tudíž přístupná z vnějšku třídy
ACC_FINAL	0x0010	takto označená třída nemůže být nijak dále děděna
ACC_SUPER	0x0020	bude speciálně zacházeno s metodami nadtřídy 3.8.2
ACC_INTERFACE	0x0200	jedná se o rozhraní, ne o třídu
ACC_ABSTRACT	0x0400	třída s tímto flagem nesmí být instanciována
ACC_SYNTHETIC	0x1000	třída byla generována kompilátorem a není přítomna ve zdrojovém souboru
ACC_ENUM	0x4000	*.class* soubor není třída ani rozhraní, ale výčet

V případě, že se hodnota této položky bude rovnat nule, patří načtený *.class* soubor interní třídě programovacího jazyka Java a to konkrétně třídě *java.lang.Object*. Třída *Object* stojí v hierarchii dědění tříd na samotném vrcholu a tudíž nemá žádného předka. Všechny ostatní třídy mají minimálně jednoho předka a to právě třídu *Object*. Proto v tomto případě může být hodnota *super_class* rovna nule.

Pokud je ovšem větší než nula, musí být tato hodnota platným indexem do *constan pool*-u a struktura na tomto indexu musí být typu *CONSTANT_Class_info*. Z této struktury poté získáme celé jméno nadřazené třídy.

u2 interfaces_count

Toto políčko v rámci obsahu *.class* souboru vyjadřuje počet přímo implementovaných rozhraní danou třídou nebo rozhraním. Maximální počet implementovaných rozhraní je 65.536, protože toto číslo je 16 bitové.

u2 interfaces [interfaces_count]

Zde se, oproti poli *constant pool*, indexuje od 0, a proto platný index do tohoto pole je v rozmezí 0 až *interfaces_count* - 1. Každá z položek tohoto pole musí obsahovat validní index do *constant pool*-u, kde se musí nacházet struktura typu *CONSTANT_Class_info*. Tato struktura nám následně dá informace o tom, jaká třída je přímým super rozhraním této třídy nebo rozhraní.

u2 fields_count

Hodnota položky *fields_count* udává počet datových složek třídy nebo rozhraní, které jsou v *.class* souboru reprezentovány jako struktury typu *field_info*. Počet datových složek je omezen rozsahem této položky opět na 65.536 a zahrnuje jak počet instančních (nestatických) datových polí, tak i třídních (statických) datových polí.

Taktéž je touto konstantou alokováno pole *fields[]*.

field_info fields [fields_count]

Každá položka tohoto pole musí být struktura typu *field_info*, která nám dává podrobný popis dané datové položky tohoto rozhraní nebo třídy. Toto pole obsahuje pouze datové položky, které patří této třídě. Zděděné datové položky ze super třídy či super rozhraní se v tomto poli nevyskytují.

u2 methods_count

Počet struktur, které jsou obsaženy v poli *methods[]* zjistíme právě z tohoto záznamu. Každá položka pole *methods[]* musí být typu *method_info*.

Jelikož je *methods_count* 16 bitové číslo, je maximální počet implementovaných metod pro danou třídu či rozhraní roven 65.536.

method_info methods [methods_count]

Pole *methods[]* slouží pro uložení všech implementovaných metod dané třídy. Obsahuje přístupová omezení, velikosti mezí kontextu metody či vlastní kód metody v instrukcích byte kódu. Vše je podrobněji popsáno v dalších částech této práce.

Pokud se nejedná o třídu, ale o rozhraní, tak žádná metoda neobsahuje tělo. Jsou uvedeny jenom prostorová omezení kontextu metody, její přístupová práva a její podrobný popis.

Toto pole obsahuje veškeré metody deklarované ve třídě včetně statických metod, nestatických metod, metod pro inicializaci instance a jakékoliv metody pro inicializaci třídy či rozhraní.

Podobně jako u pole *fields[]* i zde nejsou obsaženy záznamy o metodách, které daná třída dědí od svých předků.

u2 attributes_count

Díky položce *attributes_count* můžeme zjistit, kolik atributů zpracovávaná třída obsahuje. Vlastní atributy jsou uloženy v poli *attributes[]*, kterému konstanta *attributes_count* udává rozměry.

attribute_info attributes [attributes_count]

Pole *attributes[]* se skládá z položek typu *attribute_info*, které obsahují data každého atributu dané třídy.

Podle specifikace virtuálního stroje se mohou v byte kódu jazyka Java objevit jako atributy pro třídu tyto atributy : *InnerClasses*, *EnclosingMethod*, *Synthetic*, *Signature*, *SourceFile*, *SourceDebugExtension*, *Deprecated*, *RuntimeVisibleAnnotations*, *RuntimeInvisibleAnnotations* a *BootstrapMethods*.

Pokud virtuální stroj načte *.class* soubor verze 49.0 nebo vyšší, tak musí rozeznat a správně načíst následující sadu atributů : *Signature*, *RuntimeVisibleAnnotations* a *RuntimeInvisibleAnnotations*.

Jestliže se jedná o verzi *.class* souboru 51.0 nebo vyšší je povinně zpracovávaný parametr pouze parametr *BootstrapMethods*.

Dále může byte kód obsahovat i různé specifické typy atributů. Pokud ovšem tyto atributy virtuální stroj neumí zpracovat, má povoleno je tiše ignorovat. Takovéto atributy, které nejsou uvedeny ve specifikaci virtuálního stroje, nesmějí mít vliv na sémantiku *.class* souboru a poskytují pouze rozšiřující informace.

Datový typ *field_info*

Jednotlivé záznamy této datové struktury mají totožný význam jako položky struktury *method_info* 3.8.1, ale s tím rozdílem, že zde položka *descriptor_index* neudává signaturu metody, ale typ datové položky.

Navíc je zde odlišná interpretace *access_flags* a to podle tabulky 3.3.

```
field_info
{
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Hodnoty a význam přístupových flagů se mírně liší od těch, které jsou uvedeny v tabulce 3.2. Jejich použití je ale stejné.

Tabulka 3.3: Přístupová práva pro datové položky metody nebo třídy[3].

Název	Hodnota	Popis
ACC_PUBLIC	0x0001	datová položka může být přístupná z vnějšku balíčku
ACC_PRIVATE	0x0002	dat. položka je přístupná pouze ze své třídy
ACC_PROTECTED	0x0004	dat. položka přístupná z potomků deklarující třídy
ACC_STATIC	0x0008	jedná se o třídní datovou položku
ACC_FINAL	0x0010	jedná se o konstantu, která nemůže být měněna
ACC_VOLATILE	0x0020	dat. položka nesmí být cacheována
ACC_TRANSIENT	0x0040	není zapisován ani čten persistentním object managerem
ACC_SYNTHETIC	0x0080	dat. položka není přítomna ve zdrojovém kódu
ACC_ENUM	0x0100	dat. položka deklarována jako prvek nějakého výčtu

Datový typ *attribute_info*

Tento datový typ slouží pro uložení různých atributů metod a datových položek v *.class* souboru.

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

Položka *attribute_name_index* musí být validním indexem do tabulky *constant_pool*, na kterém musí být struktura *CONSTANT_Utf8_info*. Ta obsahuje řetězcovou konstantu reprezentující název daného atributu.

Pole bytů *info*, jehož velikost udává *attribute_length*, obsahuje byte kód vlastního atributu.

Vysvětlení jednotlivých typů atributů podle *JVM* bude uvedeno v některé z dalších kapitol.

3.9 Pole konstant constant pool

Hlavní částí každého *.class* souboru je tzv. *constant_pool*. Je to pole pevné velikosti obsahující prvky typu *cp_info*. Velikost tohoto pole je známá již v době kompilace javovského

zdrojového textu.

Jak už je z názvu patrné, toto pole obsahuje veškeré konstanty potřebné pro správné fungování této třídy. Obsahuje řetězcové konstanty, celočíselné konstanty i konstanty v plovoucí desetinné čárce.

Z řetězcových konstant stojí za zmínku deskriptory a názvy metod a datových položek, názvy tříd použitých objektů nebo třeba návratové typy metod. Celočíselné konstanty i konstanty pro práci s proměnnými v plovoucí desetinné čárce mohou být použity pro inicializaci datových polí, jako atributy při volání jednotlivých metod nebo i jako konstanty v kódu jazyka Java.

Jelikož jednotlivé položky tohoto pole nejsou stejné velikosti, nelze používat klasickou aritmetiku pro přístup do polí prvků stejné velikosti.

Každá položka v *constant pool*-u musí začínat 8 bitovým číslem *tag*, které udává, jaký je přesný formát této položky. Veškeré možné struktury, které se mohou v *constant pool*-u vyskytnout, jsou popsány v následujících kapitolách.

Obecný formát položky, která je prvkem tabulky *constant_pool* je následující :

```
cp_info
{
    u1 tag;
    u1 info[];
}
```

8-mi bitový prvek *tag* může nabývat jedné z 15 hodnot, které jsou uvedeny v tabulce 3.4. V popisech jednotlivých typů struktur chybí vysvětlení účelu položky *tag*, protože je u každého typu stejné. Položka *tag* obsahuje hodnotu, podle které rozeznáme, jaký formát a uplatnění mají následující data.

Tabulka 3.4: Možné hodnoty atributu *tag* struktury *cp_info*[3].

Typ konstanty	Hodnota
CONSTANT_Class_info	7
CONSTANT_Fieldref_info	9
CONSTANT_Methodref_info	10
CONSTANT_InterfaceMethodref_info	11
CONSTANT_String_info	8
CONSTANT_Integer_info	3
CONSTANT_Float_info	4
CONSTANT_Long_info	5
CONSTANT_Double_info	6
CONSTANT_NameAndType_info	12
CONSTANT_Utf8_info	1
CONSTANT_MethodHandle_info	15
CONSTANT_MethodType_info	16
CONSTANT_InvokeDynamic_info	18

Struktura *CONSTANT_Class_info*

První struktura, kterou si popíšeme, je struktura *CONSTANT_Class_info*. Z tohoto typu struktury zjistíme název třídy nebo rozhraní, ve kterém se hledaná položka vyskytuje. Tento

typ záznamu je obsažen ve strukturách, jenž jsou popsány dále.

```
CONSTANT_Class_info
{
    u1 tag;
    u2 name_index;
}
```

u2 name_index

Hodnota *name_index* musí být platným indexem do *constant pool*-u a položka na tomto indexu musí být struktura typu *CONSTANT_Utf8_info*, která reprezentuje jméno třídy nebo rozhraní.

Struktura *CONSTANT_Fieldref_info*

Potřebné informace o datových položkách třídy jsou nepřímo uloženy ve struktuře *CONSTANT_Fieldref_info*. Ze struktury tohoto typu zjistíme jak jméno a typ dané datové položky, tak i třídu, ve které se nachází.

```
CONSTANT_Fieldref_info
{
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

u2 class_index

Hodnota této konstanty musí být platný index do *constant pool*-u a struktura na tomto indexu musí být typu *CONSTANT_Class_info*. Tímto dostaneme typ třídy nebo rozhraní, jehož je daná datová položka členem.

u2 name_and_type_index

Tento index musí být validním indexem do *constant_pool*-u, kde se musí nacházet typ struktury *CONSTANT_NameAndType_info*, ze které dostaneme název a typ dané datové položky třídy či rozhraní.

Struktura *CONSTANT_Methodref_info*

Záznam v *constant pool*-u tohoto typu slouží pro zjištění, která metoda má být vyvolána jednou z instrukcí byte kódu, které jsou k tomu určené. Z této položky získáme třídu, ve které se daná metoda nachází, vlastní jméno vyvolávané metody, návratový typ a seznam parametrů společně s jejich datovými typy.

```
CONSTANT_Methodref_info
{
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

u2 class_index

Hodnota této konstanty musí být platný index do *constant pool*-u a struktura na tomto indexu musí být typu *CONSTANT_Class_info*. Tímto dostaneme typ třídy nebo rozhraní, jehož je daná metoda členem. Položka na tomto indexu musí být ve výsledku třída a ne rozhraní.

u2 name_and_type_index

Význam této položky je stejný jako v předchozím případě ovšem s tím rozdílem, že zde tato položka nepopisuje jméno a typ datové položky třídy či rozhraní, ale popisuje jméno a typ metody. Pokud je prvním znakem názvu metody znak ⟨, musí se metoda jmenovat ⟨*init*⟩. V tomto případě se jedná o metodu pro inicializaci instance. Návratový typ této metody musí být vždy *void*.

Struktura *CONSTANT_InterfaceMethodref_info*

Struktura *CONSTANT_InterfaceMethodref_info* má prakticky totožný význam jako struktura *CONSTANT_Methodref_info*, ale s tím rozdílem, že objekt na indexu *class_index* musí být rozhraní a nesmí být třída.

```
CONSTANT_InterfaceMethodref_info
{
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

u2 class_index

Hodnota této konstanty musí být platný index do *constant pool*-u a struktura na tomto indexu musí být typu *CONSTANT_Class_info*. Tímto dostaneme typ třídy nebo rozhraní, jehož je daná metoda členem. Položka na tomto indexu musí být ve výsledku rozhraní a nikoliv třída.

u2 name_and_type_index

Typ struktury na tomto indexu musí být *CONSTANT_NameAndType_info* a po resoluci dostaneme název a popis příslušné datové položky třídy či rozhraní.

Struktura *CONSTANT_String_info*

Struktura *CONSTANT_String_info* slouží jako úložiště řetězcových konstant dané třídy. Takto jsou uloženy názvy metod, datových položek, návratové typy metod, parametry metod opět i s jejich typy či názvy atributů.

```
CONSTANT_String_info
{
    u1 tag;
    u2 string_index;
}
```

u2 string_index

Zde opět hodnota musí být validním indexem do *constant pool*-u. Struktura na tomto indexu musí být typu *CONSTANT_Utf8_info* reprezentující posloupnost kódových znaků v kódování Unicode, na kterou je daný řetězec inicializován.

Struktura *CONSTANT_Integer_info*

V tomto typu je uloženo číslo velikosti 32 bitů reprezentující datový typ *signed integer*.

```
CONSTANT_Integer_info
{
    u1 tag;
    u4 bytes;
}
```

u4 bytes

Toto 32-bitové číslo reprezentuje hodnotu čísla datového typu *Integer*. Číslo je uloženo v big-endian pořadí, kdy vyšší byte je první.

Struktura *CONSTANT_Float_info*

Stejně jako struktura *CONSTANT_Integer_info* i tato obsahuje 32 bitů dlouhou konstantu, tentokrát se však jedná o konstantu v plovoucí desetinné čárce.

```
CONSTANT_Float_info
{
    u1 tag;
    u4 bytes;
}
```

u4 bytes

32 bitů tohoto čísla reprezentuje číslo v plovoucí desetinné čárce datového typu *Float* ve formátu *single-precision* podle normy *IEEE 754*[1][20].

Číslo je uloženo v pořadí big-endian stejně jako u struktury typu *CONSTANT_Integer_info*.

Struktura *CONSTANT_Long_info*

Zde, stejně jako u struktury typu *CONSTANT_Double_info*, se datový typ long skládá ze dvou 32 bitů dlouhých částí. Na této věci by nebylo nic divného, ale *CONSTANT_Double_info* i *CONSTANT_Long_info* zabírají dva záznamy v *constant pool*-u i v *Local Variable Table* patřičného zásobníkového rámce. Tento způsob uložení není příliš šťastný, což s ohledem zpět uznali i samotní vývojáři virtuálního stroje.

```
CONSTANT_Long_info
{
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

u4 low_bytes a u4 high_bytes

Obě dvě položky této struktury nám dohromady dají 64-bitovou konstantu typu *Long*. Jak *low_bytes*, tak i *high_bytes* jsou uloženy v pořadí big-endian.

Výsledná konstanta je vypočtena následovně :

```
((long) high_bytes << 32) + low_bytes
```

Struktura typu *CONSTANT_Long_info* má jednu velkou nevýhodu a to tu, že jedna konstanta typu *Long* zabírá dva záznamy v tabulce *constant pool*. Pokud je takováto konstanta uložena v *constant pool*-u na indexu *n*, tak další použitelná položka je až na indexu *n+2*. Položka na indexu *n+1* musí být platná, ale je označena jako nepoužitelná.

Zavedení takovéhoto ukládání konstant typu *Long* bylo velice nešťastným rozhodnutím.

Struktura *CONSTANT_Double_info*

Konstanta typu *CONSTANT_Double_info* obsahuje 64 bitů dlouhé číslo, které je ale rozděleno na dvě části - *high_bytes* a *low_bytes*. Toto rozdělení vzniklo špatným návrhem, což uznali i sami tvůrci specifikace.

```
CONSTANT_Double_info
{
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

u4 low_bytes a u4 high_bytes

Pro konstanty typu *Double* platí úplně stejná pravidla jako pro konstanty typu *Long*. Tento typ konstanty také zabírá dvě položky v *constant pool*-u, *high_bytes* i *low_bytes* mají data uložená v pořadí big-endian a výsledná hodnota *Double* je zase vypočtena stejně jako u konstanty typu *Long*.

Jediným rozdílem je vlastní interpretace čísla. Zde je číslo uloženo ve formátu *double-precision* podle normy *IEEE 754*[1][20].

Struktura *CONSTANT_NameAndType_info*

```
CONSTANT_NameAndType_info
{
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

u2 name_index

Záznam na tomto indexu musí být struktura typu *CONSTANT_Utf8_info*, která reprezentuje buď speciální jméno metody $\langle init \rangle$, nebo platné jméno metody či datové položky.

u2 descriptor_index

Položka na tomto indexu musí být opět typu *CONSTANT_Utf8_info*, ale tentokrát obsahuje popis metody nebo datové položky.

Struktura *CONSTANT_Utf8_info*

Jednou z klíčových typů konstant virtuálního stroje je konstanta typu *CONSTANT_Utf8_info*. V těchto záznamech jsou uloženy veškeré řetězcové konstanty, které jsou známé již v době kompilace do bajt kódu.

Při volání funkcí nebo přístupů k datovým položkám třídy se tyto záznamy používají pro rozhodnutí, která metoda nebo datová položka se má zavolat nebo naplnit. Vše se totiž děje přes jména jednotlivých tříd, metod a datových položek, která jsou uložena právě v těchto záznamech.

Vyhledávání porovnáváním řetězců je výpočetně drahá záležitost, a proto není příliš ideální tuto techniku. Optimalizace bude popsána v jedné z následujících kapitol.

```
CONSTANT_Utf8_info
{
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

u2 length

Hodnota tohoto atributu struktury *CONSTANT_Utf8_info* udává délku pole *bytes[]* v bytech, a ne délku daného řetězce. Řetězec, který je uložen v poli *bytes[]* není ukončen ukončovacím znakem. Ukončovat řetězec zde není nutné, protože je dopředu známa jeho délka.

Struktura *CONSTANT_MethodHandle_info*

Struktura pro popis handlu metody se nazývá *CONSTANT_MethodHandle_info*.

```
CONSTANT_MethodHandle_info
{
    u1 tag;
    u1 reference_kind;
    u2 reference_index;
}
```

u1 reference_kind

Tato hodnota musí být v rozmezí od 1 do 9 a reprezentuje druh handlu metody, tedy to, jak se bude byte kód chovat.

u2 reference_index

Konstanta *reference_index* musí být platným indexem do *constant pool*-u a typ struktury na tomto indexu je závislý na hodnotě položky *reference_kind*.

Pokud bude hodnota *reference_kind* 1, 2, 3 nebo 4, musí být struktura na indexu *reference_index* typu *CONSTANT_Fieldref_info*.

V případě, že *reference_kind* obsahuje hodnotu 5, 6, 7 nebo 8, bude na indexu *reference_index* struktura typu *CONSTANT_Methodref_info*.

Poslední možností, která může nastat, je, že hodnota v *reference_kind* bude 9. Potom je na indexu struktura typu *CONSTANT_InterfaceMethodref_info*.

Struktura *CONSTANT_MethodType_info*

Učel tohoto typu struktury se blíží účelu struktury typu *CONSTANT_Class_info*. I zde je pouze index do *constant pool*-u, ze kterého zjistím řetězec udávající návratový typ metody a její parametry.

```
CONSTANT_MethodType_info
{
    u1 tag;
    u2 descriptor_index;
}
```

Struktura tohoto typu se používá pouze pro zjištění jakého typu je daná metoda. Typ struktury, která by měla být na indexu daném položkou *descriptor_index*, musí být typu *CONSTANT_Utf8_info* obsahující řetězcovou konstantu s popisem metody.

Struktura *CONSTANT_InvokeDynamic_info*

Poslední popisovanou strukturou je struktura typu *CONSTANT_InvokeDynamic_info*, která je čistě používaná ve spojení s instrukcí byte kódu *invokedynamic* pro uvedení *bootstrap* metody, jména dynamického vyvolání metody, argumentů a návratového typu volání.

Volitelně lze z této struktury zjistit statické argumenty *bootstrap* metody.

```
CONSTANT_InvokeDynamic_info
{
    u1 tag;
    u2 bootstrap_method_attr_index;
    u2 name_and_type_index;
}
```

u2 bootstrap_method_attr_index

Hodnota tohoto prvku struktury obsahuje index do pole *bootstrap_methods*. Tato tabulka bude popsána v následujících kapitolách.

u2 name_and_type_index

Struktura typu *CONSTANT_NameAndType_info* musí být v tabulce *constant pool* na tomto indexu. Z této položky následně můžeme zjistit jak jméno metody, tak i její deskriptor - návratový typ a typy a počet parametrů.

Kapitola 4

Návrh a popis implementace

V této kapitole si popíšeme samotný návrh i výslednou implementaci celého emulátoru. Také přijde řeč na jeho jednotlivé části, bez kterých by emulátor nemohl vykonávat svou činnost. Těmito komponentami jsou načítání dat z *.jar* balíčku, *heap*, modul pro načítání dat z *.class* souborů, automatický správce paměti a část, která řeší použití metod a tříd, které jsou součástí standardní implementace programovacího jazyka *Java* a nejsou obsaženy v testovaném *.jar* balíčku.

Mimo jiné jsou zde popsány důležité třídy použité v mém emulátoru ve smyslu funkčnosti a vlivu na provádění emulace.

Při vývoji mého emulátoru jsem celou dobu vycházel z oficiální specifikace *Java Virtual Machine*[12] od společnosti *Oracle Corporation*[13], která je velice dobře a srozumitelně popsaná. Dalšími důležitými zdroji byly knihy *Inside The Java Virtual Machine* od Billa Vennerse[26] a *Java Virtual Machine* od Johna Meyera[23]. Díky těmto dvěma knihám se mi podařilo pochopit základní aspekty fungování *Java Virtual Machine*.

Celý vývoj jsem se řídil specifikací a těmito dvěma knihami až do doby, než jsem začal ohýbat emulátor směrem k detekci a analýze malware.

4.1 Popis provádění programu emulátorem

Virtuální stroj *Java Virtual Machine* je typem jednoduchého zásobníkového procesoru. Proto je jeho důležitou částí stack(zásobník), který zajišťuje veškeré provádění metod a statických funkcí přeloženého programu v programovacím jazyce *Java*.

Jelikož jsem měl zakázáno používat nástroje používající výjimky k ošetření chyb, musel jsem upustit i od použití standardní *C++ STL* knihovny, protože kontejnery z této knihovny používají právě výjimky k ošetření veškerých chybových stavů. Z tohoto důvodu jsem si vytvořil dvě šablony - *Vector* a *FixedSizeVector*. Obě dvě jsou prakticky totožné, pouze s tím rozdílem, že *Vector* nemá nijak omezenou kapacitu a může svoji velikost zvětšovat, dokud má k dispozici dostatek paměti. Naproti tomu šablona *FixedSizeVector* je inicializována na předem danou velikost. Obě dvě mají obvyklé metody, které dovolují s oběma kontejnery pracovat jako se zásobníkem.

Abych se vyhnul použití, a tudíž i implementaci kopírovacích konstruktorů u tříd, které budou ukládány do kontejnerů *Vector* a *FixedSizeVector*, ukládám do těchto dvou šablon pouze ukazatele. U kopírovacích konstruktorů bych jen velmi těžko zaregistroval chybu při alokaci bez použití výjimek.

FixedSizeVector je primárně určen k realizaci datové struktury *OperandStack*, který

slouží pro uchovávání mezivýsledků prováděných instrukcí virtuálního stroje, pro předávání parametrů do volaných metod a pro předávání návratových hodnot z těchto metod. Volání metod i jejich návrat je podrobněji popsán v implementační části v kapitolách 5.4 a 5.5.

Hlavní použití šablony *Vector* je implementace zásobníku, který realizuje průběh celého programu v programovacím jazyce *Java*. Obsahuje pointery na objekty třídy *StackFrame*, které reprezentují jednotlivé volané metody. Práce emulátoru probíhá tedy, dokud není jeho stack prázdný.

Při každém vyvolání jakékoliv metody, kromě nativní, se vytvoří nový objekt třídy *StackFrame* a umístí se na vrchol tohoto zásobníku. Vlastní implementace tohoto problému je detailněji popsána v kapitole 5.4.

4.2 Popis způsobu načítání dat

Jak již bylo zmíněno v kapitole 4.2, *.jar* balíček může obsahovat velké množství různých typů souborů. Jelikož vlastní aplikace v programovacím jazyce *Java* je zkompilována do byte kódu, což jsou soubory typu *.class*, můj emulátor načítá pouze soubory tohoto typu. Jedinou výjimku tvoří soubor *MANIFEST.MF* z adresáře *META-INF*.

Pro načtení všech potřebných souborů se postupně prochází kompletní adresářová struktura balíčku, při které se do paměti načítají veškeré *.class* soubory. Jména tříd, včetně kompletního názvu třídy skrz hierarchii balíčků, jsou načtena přímo z každého *.class* souboru.

Jakmile narazím na nějaký *.class* soubor, ihned provádím jeho zpracování. O to se stará třída *ClassFileProcessing*. Tato třída reprezentuje v paměti celý soubor typu *.class* načtený z *.jar* balíčku. Obsahuje metody a datové oblasti pro zpracování a následné uložení informací z *.class* souboru. Jedná se především o *constant pool*, který obsahuje veškeré číselné a řetězcové konstanty. Dále tato třída zpracovává datové položky dané třídy, tzv. *field*-y a nakonec zpracuje i metody, které jsou ve třídě implementovány.

Každá datová položka je uložena do objektu třídy *FieldInfoEntry* a obsahuje veškeré důležité informace o každé datové položce. Její typ a popis, přístupové flagy, offset v rámci objektu nebo v rámci třídy, pokud se jedná nebo nejedná o statický field.

Jedinou položkou, kterou nezpracovávám, jsou atributy třídy, metody nebo datové položky. Ve většině případů jde jen o volitelné informace, které nemusí být nutně u každé položky(třída, metoda, datová položka) uvedeny a které slouží pro potřeby debuggeru.

Výjimku tvoří atribut metody s názvem *Code*. Pokud se nejedná o nativní metodu, musí být vždy u metody přítomen. Obsahuje totiž instrukce pro vlastní provádění operací virtuálním strojem. Atributy jsou uloženy v instancích třídy *AttributeInfoEntry*.

4.3 Popis implementace emulátoru

Nejprve je nutné vysvětlit účel a použití dvou tříd, kterými jsou *Variable* a *StackFrame*.

Třída *Variable* slouží pro uložení operandů, argumentů metod a mezi výsledků v rámci provádění jedné konkrétní metody. Objekty této třídy mohou obsahovat veškeré datové typy, které byly zmíněny v tabulce 4.2. O to, který typ instance této třídy aktuálně obsahuje zjistíme z datové položky této třídy, která má název *m_dataType* a která je typu *DataType*. Hodnoty, které může nabývat tento typ jsou znázorněny v tabulce 4.1.

Datové položky jednotlivých typů nejsou uloženy každý zvlášť, ale v unii. Tento přístup výrazně zmenší velikost třídy, protože je paměť alokována pro největší prvek z dané unie. V tomto případě je to datový typ *double*, který má velikost 8 bajtů. Jaký datový typ tedy

Tabulka 4.1: Hodnoty, kterých může nabývat datový typ *DataType*.

Typ konstanty	Hodnota
Variable::BYTE	reprezentuje datový typ *byte*
Variable::BOOLEAN	reprezentuje datový typ *boolean*
Variable::SHORT	reprezentuje datový typ *short*
Variable::CHAR	reprezentuje datový typ *char*
Variable::INTEGER	reprezentuje datový typ *integer*
Variable::FLOAT	reprezentuje datový typ *float*
Variable::LONG	reprezentuje datový typ *long*
Variable::DOUBLE	reprezentuje datový typ *double*
Variable::REFERENCE	reprezentuje datový typ *reference*
Variable::INITIAL	počáteční hodnota tohoto typu. Pokud se v programu vyskytne tato hodnota, program končí chybou.

nakonec použít zjistíme z proměnné *m_dataType*, která nabývá jedné z konstant z tabulky 4.1.

Objekty této třídy slouží pro uchování informací ve strukturách každého zásobníkového rámce(*StackFrame*), jimiž jsou *Local Variable Table* a *Operand Stack*. *Operand Stack*, spolu s tabulkou *Local Variable Table*, slouží pro předávání parametrů do nově volaných metod. Pro vracení návratových hodnot zpátky do metod volajících slouží pouze zásobníky operandů *Operand Stack* jednotlivých metod.

4.4 Návrh *heap*-u pro alokaci objektů a polí

Jednou z hlavních součástí každého virtuálního stroje je *heap*. Je to datová struktura sloužící pro dynamickou alokaci objektů. Jelikož v programovacím jazyku *Java* pole vystupují jako objekty, slouží tato datová oblast i pro uložení polí a to jak jedno dimenzionálních, tak i multidimenzionálních. Multidimenzionální pole jsou jen pole jedno dimenzionálních nebo více dimenzionálních polí.

Heap je v mém emulátoru implementován jako jednorozměrné pole bajtů a při každé nové alokaci se z něj patřičná část vezme a přiřadí se k instanci nově alokované třídy. Podrobný popis implementace je popsán v kapitole 5.3.

Data z paměti nejsou procesorem načítána po jednotlivých bajtech, ale vždy se načítá takový kus paměti, který odpovídá šířce sběrnice procesoru. Z tohoto důvodu jsou veškerá data uložena na *heap*-u zarovnána na velikost dělitelnou 8. Takto bude mít procesor ulehčenou práci, protože adresa začátku paměti každého objektu je dělitelná 8. Například pokud je čistá velikost objektu 22 bajtů, je jeho velikost zaokrouhlena na 24 bajtů. Podrobný popis způsobu alokace objektů a polí je lépe znázorněn v kapitole 4.4.1.

Jako každá správná implementace virtuálního stroje potřebuje mít oddělená data pro statické(třídní) a nestatické(instanční) proměnné. V mojí implementaci tomu není jinak. Můj heap je rozdělen na dvě části, ve kterých je jedna část pro statická a druhá část pro nestatická data. V obou dvou případech je princip uložení dat prakticky stejný, přesto se od sebe nepatrně liší, což je popsáno v kapitolách 4.4.1 a 4.4.2.

4.4.1 Heap pro nestatická data

Na tomto *heap*-u se alokují data pro objekty, které jsou alokovány přes operátor *new* ve zdrojovém kódu programovacího jazyka *Java*.

Pro tento operátor se nachází instrukce přímo v bajt kódu metody, která požaduje alokaci objektu, a jmenuje se opět *new*. Při výskytu této instrukce zjistíme třídu, která má být instanciována, vytvoříme instanci této třídy a navrátíme referenci. Referencemi se nyní zabývat nebudeme, jejich podrobný popis bude uveden dále v textu v kapitole 4.4.1.

Operátor *new* i k němu přiřazená instrukce alokují vždy paměť na *heap*-u pro nestatická data, a to i když je tento operátor použit ve statickém kontextu. V tomto případě bude na statickém *heap*-u uložena pouze reference na objekt alokovaný na nestatickém *heap*-u. S takto vytvořeným objektem je dokonce jinak zacházeno při správě paměti(*garbage collecting*), kdy takový objekt není z nestatického *heap*-u nikdy odstraněn a nachází se v něm až do konce vykonávání programu.

Přiřazování referencí

Jelikož specifikace *Java Virtual Machine*[12] přikazuje použití *heap*-u, který je spravován automatickým správcem paměti, musíme zajistit bezproblémový přístup k instancím jednotlivých objektů. To nám zabezpečuje mechanizmus přidělování referencí a spolu s ním třída *ObjectHandle*.

Automatický správce paměti slouží k úklidu nepoužívaných objektů z *heap*-u, díky čemuž se programátor v programovacím jazyku *Java* nemusí starat o uvolňování již nepotřebné paměti sám(jako třeba v jazyce *C++* pomocí příkazů *new* a *delete*). Garbage collector odstraní z *heap*-u veškeré nedosažitelné objekty, což má a následek přesun dat v rámci *heap*-u. Jak a kdy se přesouvají jednotlivé instance je podrobněji popsáno v kapitole 4.4.3.

Kdybychom při operaci *new* vraceli přímo adresu paměti, došlo by k nárůstu výpočetní náročnosti, protože bychom museli všude, kde jsou živé přesouvané objekty používány, měnit jejich adresu v paměti. Díky použití referencí toto vyhledávání a aktualizace adres na více místech odpadá.

Při alokaci objektu nebo pole na *heap-u* pro nestatická data vždy vracíme referenci. Reference je index do tabulky objektů typu *ObjectHandle*, kde na každém indexu, pokud je validní, nalezneme adresu patřící konkrétní instanci třídy nebo pole. Poté při přesunu objektu na novou adresu se nemusí měnit adresy na všech místech, ale změní se pouze adresa přiřazená ke konkrétní referenci. Číslo reference všude v programu tím pádem zůstává nezměněno. Tímto přístupem se aktualizuje pouze jedna adresa a to ta, která je přiřazena k dané referenci.

Tabulka referencí je používaná i při alokaci nových objektů ve statickém kontextu. Na statickém *heap*-u se žádné objekty nealokují, obsahuje totiž pouze reference na objekty.

Při zavolání operátoru *new* v programovacím jazyku *Java* a následném volání instrukce *new* v bajt kódu se podíváme, jakou má objekt dané třídy velikost. Tuto velikost spolu s hlavičkou zabereme v rámci oblasti nestatického *heap*-u. Další operací, která následuje, je přiřazení volné reference. V proměnné třídy *ObjectHeap* si udržuji 32 bitovou hodnotu, která reprezentuje aktuálně první volnou referenci. Proměnná se jmenuje *m_referenceIndex* a je typu *DWord*. Máme tudíž k dispozici 2^{32} hodnot, jenž lze přiřadit jako referenci na objekt. Jelikož je reference s číslem 0 přiřazena hodnotě *null*, máme tedy možnost alokovat na *heap*-u $2^{32} - 1$.

Objekty typu *ObjectHandle* obsahují kromě adresy referencovaného objektu i flagy sloužící pro alokaci a automatickou správu paměti. Tato třída i použití jednotlivých flagů je

podrobně popsána v kapitole 5.3.1.

Vytváření objektů a jejich zarovnání

Jak již bylo napsáno v předchozí kapitole, velikost každého objektu je zaokrouhlena na takové číslo, které je dělitelné, 8 tzn. že adresa každého datového bloku začíná na adrese, která je dělitelná 8. Tento fakt urychlí načítání dat z paměti procesorem.

Jedním z faktorů, které zajišťují multiplatformnost, je, že primitivní datové typy mají vždy stejnou velikost na všech možných platformách využívající daný virtuální stroj. Velikost jednotlivých typů je vidět v následující tabulce.

Tabulka 4.2: Velikosti primitivních datových typů používaných v jazyce *Java*.

Datový typ	velikost v bajtech
byte	1
boolean	1
char	2
short	2
int	4
float	4
long	8
double	8
reference	4

Jediný datový typ, který může mít proměnlivou délku napříč platformami, je datový typ *reference*. Většinou se používá velikost 4B pro 32-bitové a 8B pro 64-bitové architektury. Dále v textu se budeme zabývat pouze případem, že velikost datového typu *reference* je 4B. Je nutné poznamenat, že *reference* ve skutečnosti není datový typ, ale pouze abstrakce datového typu. V tomto textu ale bude všude uvedeno, že se o datový typ virtuálního stroje jedná.

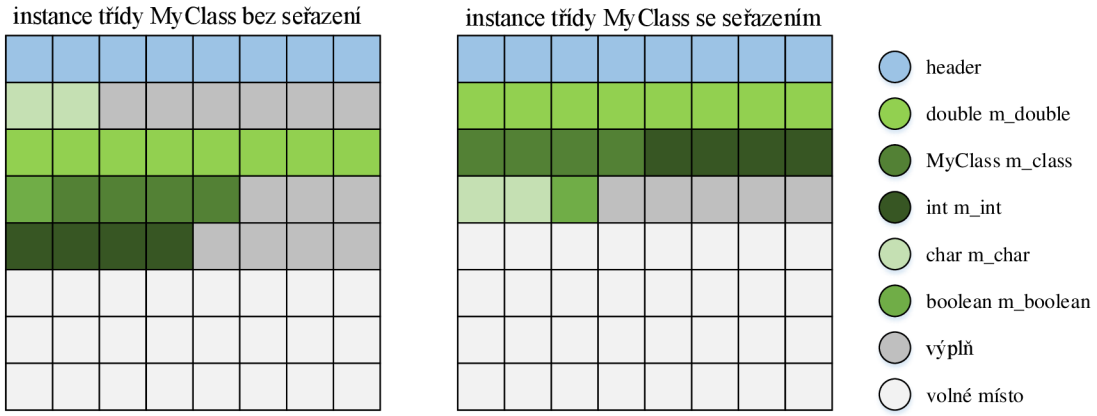
Dalším problémem, který zarovnání řeší, je plýtvání pamětí. Abychom se tohoto problému vyvarovali, musíme datové položky každé třídy řadit podle velikosti. Pokud by datové položky byly do paměti alokovány v pořadí, ve kterém jsou deklarovány ve zdrojovém kódu, vznikaly by výplňové díry, které by byly zabrány, ale neobsahovaly by žádná platná data.

Vše je nejlépe vidět na následujícím příkladu. Mějme třídu *MyClass* napsanou v programovacím jazyce *Java*.

```
public class MyClass
{
    char m_char;
    double m_double;
    boolean m_boolean;
    int m_int;
    MyClass m_class;
}
```

Na obrázku 4.1 je vidět, jak by vypadala alokace objektu třídy *MyClass* s neseřazenými a jak se seřazenými datovými položkami podle velikosti. Paměť si můžeme představit jako 2D tabulku o šířce 8 bajtů a jednotlivé položky jsou alokovány vedle sebe nebo pod sebe.

Ve skutečnosti je ale *heap* implementován jako jednorozměrné pole. Podrobnější informace o samotné implementaci haldy jsou uvedeny v kapitole 5.3.



Obrázek 4.1: Instance třídy *MyClass* s *a)* neseřazenými a *b)* seřazenými datovými položkami.

Jenom jednoduchým seřazením podle velikosti jednotlivých položek bylo ušetřeno 8 bajtů paměti. Při složitějších třídách může být úspora ještě daleko větší.

Řazení probíhá pouze podle velikosti. Datové položky třídy by šlo ještě seřadit podle názvu, ale není to nutné, a proto jsem od tohoto řešení ustoupil. Porovnávání řetězců je totiž nákladná operace a i když řazení datových položek neprobíhá příliš často, mohlo by pro větší počet složitějších tříd mít významný vliv na výkon celého emulátoru.

Alokace objektu musí být správná i v rámci hierarchie dědění. Pokud má třída nějakého rodiče, tak velikost objektu je součtem velikosti rodiče bez hlavičky a velikosti potomka také bez hlavičky. Takto vytvořená instance je samozřejmě opatřena hlavičkou. Řazení zde probíhá stejně jak v případu, kdy třída nemá žádného rodiče. Datové položky rodiče a potomka se mezi sebou neporovnávají a zůstávají seřazeny pouze v rámci své třídy. I v tomto případě se provádí zarovnání na 8 bajtů.

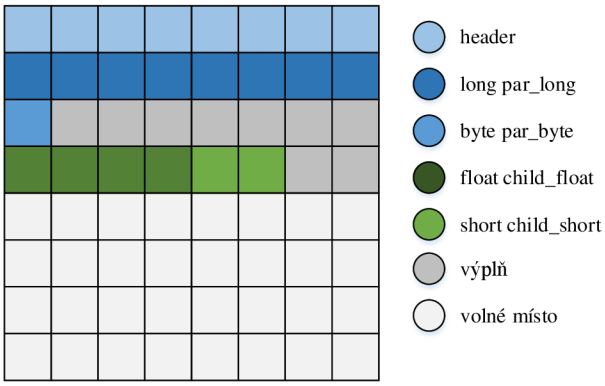
Vše je nejlépe vidět na dalším příkladu, kde je využita dědičnost. Třída *Child* dědí vlastnosti(datové položky) od třídy *Parent*.

```
class Parent                class Child extends Parent
{                           {
  long par_long;              float child_float;
  byte par_byte;              short child_short;
}                           }
```

Nejprve se alokuje hlavička pro objekt třídy *Child*, poté se alokují data třídy *Parent* a až nakonec jsou alokována data třídy *Child*. Na tomto příkladu je vidět jak se alokují data pro jednostupňové dědění, ale princip je stejný, i když by např. třída *Parent* dědila od nějaké další třídy. Opět by se nejdříve alokovala hlavička sloužící pro posledního potomka v hierarchii a až po ní by se postupně alokovala data od nejvyššího rodiče směrem dolů v rámci hierarchii dědění. Poslední částí objektu by byly data pro posledního potomka.

Nové vytvoření objektu probíhá příkazem *new* v programovacím jazyku *Java*, což má za následek vygenerování instrukce *new* v bajt kódu příslušného *.class* souboru. Za operačním

kódem instrukce *new* následuje 16-ti bitový index do příslušného *constant pool*-u kde musí být prvek reprezentující třídu nebo rozhraní. Při provádění této instrukce zavoláme příslušnou funkci *ObjectHeap::createObject*, která provede alokaci objektu na *heap*-u a následné navrácení přiřazené reference. Tato funkce je podrobněji popsána v kapitole 5.3.



Obrázek 4.2: Instance třídy *Child*, která dědí od třídy *Parent*.

Jednorozměrná a vícerozměrná pole

Alokace polí na *heap*-u je prakticky totožná s alokací objektů. Nejprve následuje 8 bajtová hlavička, dále 4 bajtová délka pole a za touto délkou následují buď samotná data nebo výplň a až po ní samotná data. Oba případy budou rozebrány v následujících částech.

Pro vytvoření jednorozměrného pole slouží instrukce *newarray* a *anewarray*[4]. Instrukcí *newarray* alokujeme prostor pro pole primitivního datového typu, jimiž jsou *byte*, *boolean*, *short*, *char*, *int*, *float*, *long* a *double*. Za touto instrukcí následuje 8-mi bitové číslo, které udává typ nově vytvářeného pole. Hodnoty jsou uvedeny v následující tabulce 4.3.

Tabulka 4.3: Hodnoty operandů pro rozlišení typu nově vznikajícího pole[4].

Datový typ pole	hodnota typu v bajt kódu
T_BOOLEAN	4
T_CHAR	5
T_FLOAT	6
T_DOUBLE	7
T_BYTE	8
T_SHORT	9
T_INT	10
T_LONG	11

Při provedení této instrukce v nějaké metodě se pole dané velikosti, která je uložena jako první prvek na zásobníku operandů, alokuje na *heap*-u. Přiřazená reference je vrácena do právě prováděné metody a uložena na vrchol zásobníku operandů[4]. Při vytvoření pole jsou všechny jeho prvky nastaveny na implicitní hodnoty.

Trochu odlišně vypadá instrukce *anewarray* sloužící pro vytvoření jednorozměrného pole referencí na *heap*-u. V tomto poli budou uloženy reference buď na třídu, nebo rozhraní a

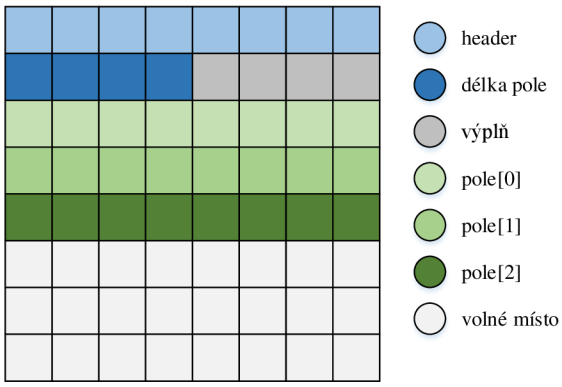
nebo další jednorozměrné nebo vícerozměrné pole. Tato instrukce za svým operačním kódem očekává 16-ti bitový index do *constant pool*-u, na kterém najdeme položku reprezentující třídu, rozhraní nebo pole. Záznam na tomto indexu musí být typu *CONSTANT_Class_info*, což už bylo popsáno v kapitole 3.8.1. Délka nově vytvářeného pole je opět uložena na vrcholu zásobníku operandů příslušné metody.

Po vytvoření pole je opět navrácena reference na toto pole a uložena na vrchol zásobníku operandů. I prvky tohoto pole jsou inicializovány na svoji implicitní hodnotu, která je v případě referencí hodnota *null*. Tímto způsobem se vytvářejí jednorozměrná pole referencí nebo části multidimenzionálního pole, což je popsáno v kapitole 4.4.1.

Alokace polí typu *double* a *long*

Při alokaci pole typu *long* nebo *double* se za hlavičkou a délkou pole objeví vždy 4B výplň a až po této výplni jsou za sebou uloženy 8-mi bajtové položky buď typu *long* nebo *double*. Za vlastními daty již žádné vyplňování není potřeba, protože data jsou vždy správně zarovnána.

Vše je nejlépe vidět na následujícím obrázku 4.3.



Obrázek 4.3: Pole *double pole = new double[3]* a jeho alokace na *heap*-u.

Alokace polí ostatních primitivních typů

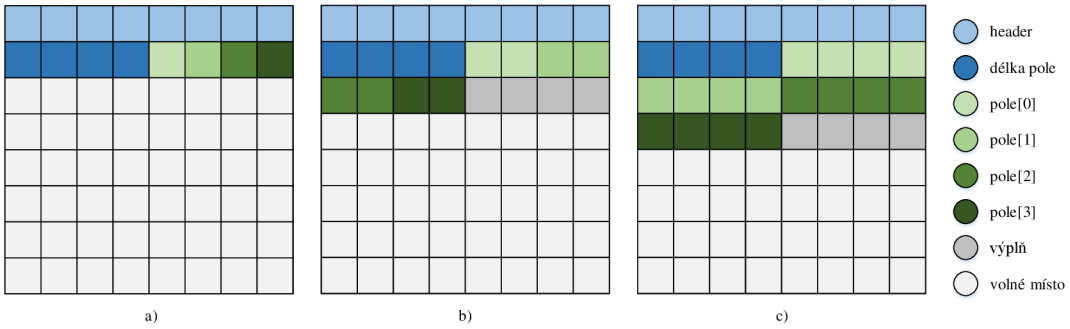
Tento způsob alokace se provádí pro pole typu *byte*, *boolean*, *short*, *char*, *int*, *float* a *reference*, protože se tyto datové typy vlezou do 4B, které následují za 4. bajty uchovávající délku samotného pole. Velikosti jednotlivých datových typů, kterých se to týká lze vidět v tabulce 4.2.

Pro tento případ jsou výplňové bajty umístěny na konci, což je vidět na obrázku 4.4.

Alokace vícerozměrných polí

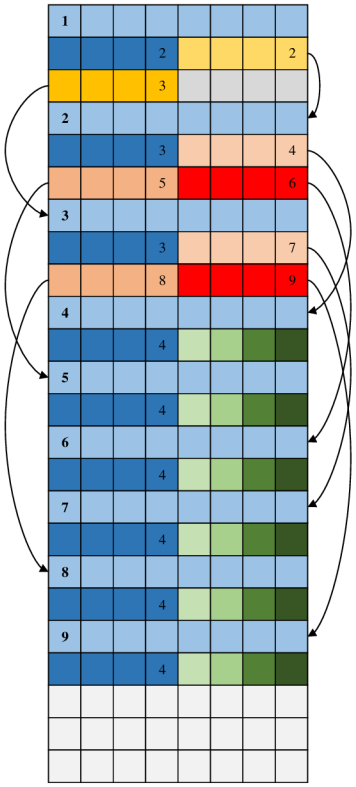
Vícerozměrná pole nejsou nic jiného, než-li pole referencí, na nějaké další pole ať už primitivního datového typu nebo referencí. Například pole *int [][]pole = new int[2][3]* je pole referencí velikosti 2, z čehož každá položka obsahuje referenci na jednorozměrné pole typu *int* obsahující 3 prvky.

Takto zanořovat se lze prakticky do nekonečna. Maximální počet dimenzí je však stanoven specifikací virtuálního stroje na 256[12].



Obrázek 4.4: Ukázky alokace pole velikosti 3 *a)* typu *byte a boolean*, *b)* typu *short a char* a *c)* typu *int, float a reference*.

Nejlépe je tato problematika k pochopení na následujícím obrázku 4.5, na kterém je zobrazena alokace trojrozměrného pole s velikosti dimenzí 2, 3 a 4. Máme tedy uloženo $2 * 3 * 4$ prvků typu *byte*. Záměrně jsem zvolil pole typu *byte*, protože je to datový typ s nejmenší možnou délkou a to 8 bitů. Deklarace tohoto pole v programovacím jazyku *Java* je *byte pole = new byte[2][3][4]*.



Obrázek 4.5: Pole *byte pole = new byte[2][3][4]* a jeho alokace na *heap*-u.

Zde je dobře k vidění neefektivita uložení vícerozměrných polí, která je dána velikostí datového typu *reference* a hlavně vlastním způsobem uložení vícerozměrných polí ve virtuál-

ním stroji. V tomto případě, kdy ukládáme 24 položek typu *byte*, tedy 24 bajtů, potřebujeme k uložení celého trojrozměrného pole celých 168 bajtů.

Nejprve je alokováno pole referencí velikosti 2, které zastřešuje celé trojrozměrné pole a jehož reference se bude vracet při provádění instrukce *multianewarray*. V našem případě má číslo 1 a je vybarveno dvěma odstíny žluté. Dále následuje vytvoření dvou polí typu *reference* velikosti 3, které bude obsahovat hodnoty referencí ukazující na poslední pole primitivního datového typu *byte*. Tyto dvě pole mají čísla referencí 2 a 3, což je uloženo v prvním poli a jsou vybarveny odstíny červené.

Posledními vytvářenými poli je 6 polí velikosti 4, které jsou již typu *byte* a uchovávají již samotné hodnoty. Na obrázku jsou vyobrazeny odstíny zelené. Výplň je po vzoru předchozích příkladů zobrazena šedou barvou.

Alokaci vícerozměrného pole způsobí v bajt kódu instrukce *multianewarray*[4]. Tato instrukce za svým operačním kódem předpokládá nejprve 16-ti bitový index do *constant pool*-u, na kterém leží položka typu *CONSTANT_Class_info* 3.8.1 nesoucí informace o tom, jaké pole se bude vytvářet. Za tímto indexem následuje 8-mi bitové číslo reprezentující počet dimenzí nově vznikajícího vícerozměrného pole. Díky tomu, že jde o 8-mi bitové číslo, může mít pole maximálně 256 dimenzí.

Další informací, kterou potřebujeme pro vytvoření multidimenzionálního pole, jsou velikosti jednotlivých dimenzí. Tyto údaje jsou za sebou uloženy v operandech na zásobníku operandů. Po provedení této instrukce je odstraněn přesně takový počet prvků z *Operand Stack*-u jaký je počet dimenzí[4].

4.4.2 Heap pro statická data

Této datové oblasti by se nejspíše nemělo říkat *heap*, ale podle specifikace virtuálního stroje[12] spíše *method area*. Jelikož je ale tento kus paměti alokován v rámci objektu *ObjectHeap*, říkám jí proto *heap* pro statická data nebo také statický *heap*.

Tato datová oblast slouží pro uchování statických dat každé použité třídy. Jsou to prakticky globální konstanty, které jsou přístupné buď přes název třídy nebo přímo přes instanci třídy. Prvně uvedená varianta je ovšem korektnějším a čistějším řešením. Pokud se ale programátor rozhodne měnit tato data přes nějakou instanci třídy, změna se projeví ve šech ostatních instancích.

Způsob alokace dat je hodně podobný alokaci, která je použitá ve variantě *heap*-u pro nestatická data. Odpadá zde ovšem jakákoliv hlavička před daty jedné konkrétní třídy. Každá statická datová položka má unikátní offset v rámci tohoto *heap*-u. Proto zde hlavička nebo použití objektů typu *ObjectHandle* není potřeba.

Data každé třídy jsou zde opět seřazena podle velikosti a taktéž zarovnána na 8 bajtů tak, aby měl procesor lepší a rychlejší přístup k těmto datům.

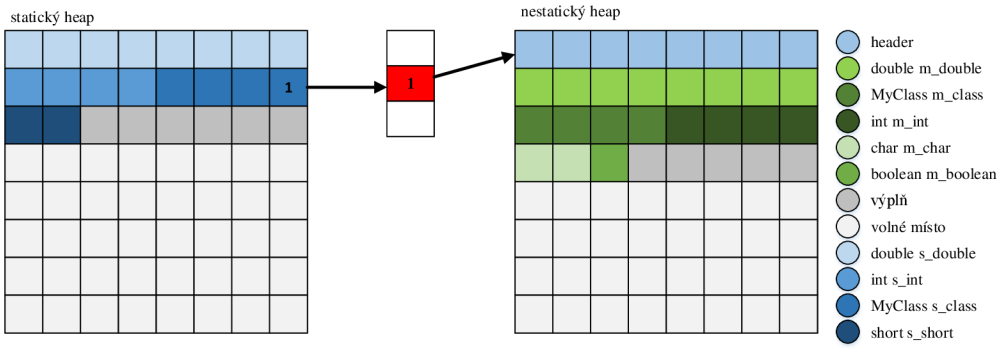
Na tomto *heap*-u jsou uloženy datové položky všech typů. Jediný rozdíl je v použití typu *reference*. V případě alokace objektu na nestatickém *heap*-u jsou jeho datové položky alokovány na stejném místě. Ovšem v rámci alokace statického objektu je na statickém *heap*-u uložena pouze reference na tento objekt a samotná data tohoto objektu jsou uložena na nestatickém *heap*-u tak, jak to bylo popsáno v kapitole 4.4.1. Takto vytvořený objekt nebude nikdy vymazán automatickým správcem paměti při provádění *garbage collecting*-u.

Mějme opět třídu *MyClass* použitou v obrázku 4.1 a k ní třídu *StaticClass*, která třídu *MyClass* používá jako svůj statický prvek.

Na obrázku 4.6 je zřetelně vidět, jak probíhá alokace instance třídy *StaticClass* obsahující statické datové položky, mezi kterými je i jedna typu *reference*.

```
class StaticClass
{
  static MyClass s_class;
  static int s_int;
  static double s_double;
  static short s_short;
}
```

```
public class MyClass
{
    char m_char;
    double m_double;
    boolean m_boolean;
    int m_int;
    MyClass m_class;
}
```



Obrázek 4.6: Alokace objektu třídy *StaticClass* zobrazená na obou typech *heap*-u.

Nejdříve dojde k seřazení statických datových položek stejně, jak již bylo popsáno v kapitole 4.4.1, a výpočtu offsetů jednotlivých statických datových položek. Dalším krokem je volání statického inicializátoru, jehož účel je popsán v následujícím oddstavci. V tomto inicializátoru se inicializují statické datové položky včetně vytváření statických objektů, tak jak je to v tomto případě. Alokace statického objektu se děje naprosto stejně jak v opačném případě, tudíž přes příkaz *new* a následně instrukci v bajt kódu *new*.

Při načtení třídy *StaticClass* emulátorem je první věcí, která se provede, inicializace statických datových položek dané třídy pomocí statického inicializátoru. Je to obdoba konstruktoru třídy a v bajt kódu každé třídy nesmí být žádný nebo maximálně jeden. Statický inicializátor třídy má pokaždé jméno <*clinit*>. Statický inicializátor sdružuje všechny statické inicializační bloky ve zdrojovém kódu jazyka *Java* do jednoho, což je zajištěno překladačem *javac*[10]. V případě, že tento inicializátor statických datových položek chybí, inicializují se tyto položky implicitními hodnotami.

4.4.3 *Mark & Compact* garbage collector

Při hledání toho pravého algoritmu pro automatickou správu paměti a procházení různých publikací byl prvním algoritmem, na který jsem narazil, algoritmus *Mark & Sweep*. V této části jsem vycházel z knihy *Garbage Collection*, kterou napsal Richard Jones[21].

Algoritmus *Mark & Sweep* je jedním z nejjednodušších algoritmů pro automatickou správu paměti. Tento algoritmus nejprve najde veškeré *root* objekty. To jsou takové objekty, na které vede v daném okamžiku, kdy dochází k úklidu paměti, v programu nějaká reference. Podrobnější vysvětlení tohoto problému čtenář nalezne v kapitole 4.4.3. A dále prochází veškeré další reference, které dané živé objekty obsahují. Tímto způsobem se projdou všechny objekty, které jsou aktuálně alokovány na nestatickém *heap*-u a pokud na ně

vede nějaká reference, jsou označeny jako živé. Ostatní objekty, na které nevede žádná reference, jsou označeny jako neživé, respektive nejsou označeny jako živé. Tato část *garbage collecting*-u se nazývá fáze *Mark*.

Po fázi *Mark* následuje fáze *Sweep*, které prochází postupně všechny objekty a maže veškeré objekty, které nejsou označeny jako živé. Místo v paměti, které bylo alokováno pro neživé objekty, je poté k dispozici pro nově alokované objekty.

Zdálo by se, že tento algoritmus je jednoduchý a ideální na použití, ale opak je pravdou. Díky tomu, že se nepoužívané objekty pouze smažou a jejich paměť je poté k dispozici, vznikají v paměti díry. Tomuto nechtěnému jevu se říká fragmentace. Celá paměť je po provedení správy paměti plná dír, které vznikly smazáním nepoužívaných objektů.

Fragmentace má za následek dvě neblahé věci. První z nich je, že musí být někde uchováno kde přesně se volná místa nacházejí. I vyhledání volného místa nemusí být triviální operací. Volná místa můžou být seřazena podle velikosti a následně sekvenčně procházena pro nalezení nejvhodnějšího volného místa. Druhým nekalým následkem je, že i když v součtu máme dostatek volné paměti, díky dírám stejně nemůžeme alokovat potřebný souvislý kus paměti pro alokaci objektu nového. To bude mít za následek buď nové volání správce paměti s nějakými přísnějšími pravidly pro určení živých objektů, nebo neprovedení potřebné alokace, následné vyhození výjimky a pravděpodobně konec programu.

Při přemýšlení jak se vyhnout fragmentaci *heap*-u při alokaci paměti pro instance tříd mi padla pozornost na kopírovací algoritmy pro automatickou správu paměti. Tyto algoritmy beze zbytku řeší fragmentaci haldy použitím dvou stejně velkých datových oblastí určených jako *heap*. V daný okamžik je ale aktivní pouze jedna z nich a druhá čeká na to, až bude použita pro *garbage collecting*.

Princip kopírovacích algoritmů spočívá v kopírování živých objektů do druhé(neaktivní) datové oblasti tak, že všechny objekty po skončení algoritmu tvoří jednolitý celek. Algoritmus začíná opět fází *Mark*, která označí všechny objekty na *heap*-u buď jako živé, či jako neživé. Po fázi *Mark* následuje kopírovací fáze. Ta prochází opět všechny objekty a pokud je aktuálně procházený objekt živý zkopíruje jej do nyní neaktivní datové oblasti. Kopírování živých objektů do neaktivní datové oblasti probíhá tak, aby nedocházelo k fragmentaci. Na konci této fáze máme v neaktivní oblasti pouze data objektů, které jsou živé a navíc tvoří jednolitý celek. Nakonec se aktivní a neaktivní oblasti prohodí, což má za následek, že se v paměti nacházejí pouze živé objekty bez fragmentace.

Zde by se zdálo, že už je algoritmus natolik sofistikovaný, že není třeba hledat žádný jiný další. Opak je ovšem pravdou. Jednou velkou nevýhodou je zdvojení paměti pro uložení dat pro objekty. Další nevýhodou tohoto přístupu pro automatickou správu paměti je zbytečné kopírování těch dat, která nejsou vůbec přesouvána v rámci aktuální haldy. Tedy těch dat, která jsou živá a která již na svém místě jsou. I toto kopírování bude mít ve výsledku dopad na výkonnost celého emulátoru.

Proto jsem hledal dál a rozhodl jsem se použít algoritmus s názvem *Mark & Compact*. Tento algoritmus pro *garbage collecting* spojuje výhody algoritmu *Mark & Compact* spolu s kopírovacími algoritmy. Z algoritmu *Mark & Sweep* přebírá tento algoritmus to, že dokáže správně označit objekty i s referenčními smyčkami, a také to, že udržuje lokalitu dat v paměti. Jinak řečeno v *heap*-u nevzniká fragmentace a to vše zvládá bez toho, aby potřeboval dvojnásobnou velikost paměti jak je tomu o kopírovacích algoritmů. Což je veliká výhoda oproti klasickým kopírovacím algoritmům.

Root objekty a jejich hledání

Základním kamenem pro vykonání určitého algoritmu, v tomto případě *Mark & Compact*, je nalezení tzv. *root* objektů. Jsou to objekty, na které je v danou chvíli provádění programu aktivní nějaká reference. Počítají se sem všechny objekty, na které je reference na statickém *heap*-u.

Dalším druhem *root* objektů jsou ty, jejichž reference jsou obsaženy v datových strukturách *Local Variable Table* všech objektů typu *StackFrame* v rámci celého emulátoru. Z těchto objektů se následně procházejí další objekty, které jsou z *root* objektů referencovány. Po projití všech *root* objetků i objektů, na které je z těchto objektů reference, označování končí a lze odstranit nepoužívaná data a přesunout živé objekty na správná místa. Toto vše dohromady zachová v paměti pouze živé objekty a vše pěkně na jednom jednolitém místě bez fragmentace.

Reference obsažené ve všech strukturách *OperandStack* nejsou procházeny, protože zde jsou uloženy pouze mezivýsledky operací.

Spouštění automatického správce paměti

Přístupů, kdy spouštět *garbage collector* je mnoho, ale ne všechny jsou vhodné. Jelikož *garbage collector* prochází celou obsazenou část *heap*-u, jde o velice drahou operaci co se týká výpočetního výkonu. Musíme totiž projít jak všechny *root* objekty, tak i všechny ostatní objekty, které jsou z *root* prvků dosažitelné.

Po tomto označení následuje přesun živých objektů v rámci *heap*-u na místa, která byla uvolněna správcem paměti. Posledním, co se v rámci úklidu provádí je aktualizace adresy objektu v rámci *heap*-u v objektu *ObjectHandle*, což se provádí již při přesouvání objektu na novou pozici.

Jedním z možných přístupů je volat *garbage collector* ihned po skončení provádění nějaké metody, tedy při výskytu jedné z instrukcí pro návrat - instrukce *return*, *ireturn*, *dreturn*, *freturn*, *areturn* a *lreturn*. Pokud by se ale volal *garbage collector* vždy při výskytu těchto instrukcí, znamenalo by to obrovský nárůst požadavků na prováděnou emulaci. *Garbage collecting* je totiž velice drahá operace.

Proto jsem se rozhodl volat *garbage collector* pouze pokud je to potřeba. Vyvolání funkce vykonávající *garbage collector* bude probíhat pouze tehdy, když nebude dostatek volné paměti pro vytvoření objektu.

4.5 Popis použití falešných metod a tříd

Programy v programovacím jazyce *Java* implementující řešení nějakého problému obsahují zdrojové kódy pouze vztahující se na tento problém. Ostatní věci, potřebné pro vykonání takovýchto programů, si *Java Virtual Machine* načítá ze balíčku standardních funkcí obsaženého v běhovém prostředí *Java Runtime Environment*. Tento balíček se jmenuje *rt.jar* a je obsažen v instalační složce běhového prostředí.

Balíček *rt.jar* nevystupuje jako spustitelný soubor, ale jako knihovna. Proto soubor *MANIFEST.MF* tohoto balíčku neobsahuje položku *Main-Class*. Není to ostatně potřeba, protože soubor není spustitelným programem v *Javě*. Programy si tedy data, která chtějí používat, musí od někud získat.

V balíčku *rt.jar* jsou obsaženy třídy, které implementují různé seznamy, hashovací tabulky a mapy, zápis do souborů, přístup k síťovým prostředkům, standardní vstup a výstup

a mnoho dalších funkcí.

Načítání těchto tříd má na starosti modul virtuálního stroje jménem *classloader*. Je to modul, který má za úkol načítat do paměti potřebné třídy a odstraňovat již nepotřebné. Tento problém by šlo vyřešit obdobně, ale já jsem se rozhodl jít cestou falešných tříd tvz. *fake* tříd. Díky tomuto přístupu nebudu muset načítat požadované třídy ze souborového systému, ale budu podmnožinu z těchto tříd uloženou přímo v paměti emulátoru. Tento přístup pro volání standardních metod a tříd bude mít výrazný vliv na rychlost provádění takto načtených metod. Vše je podrobněji popsáno v kapitole 5.10 a výsledky experimentů a měření zase v 6.

4.6 Návrh způsobu analýzy a detekce

Můj emulátor je hlavně platný v tom, že nepustí spouštěný program ven z tohoto emulátoru. Jedná se tedy o jednoduchý sandbox. V obecně používaných virtuálních strojích, popsaných v kapitole 3.7, programy zasahují mimo virtuální stroj například zápisem nebo čtením do souborů na disku, přístupem ke zvukovým zařízením či posíláním nebo příjmem skrz síťové rozhraní.

Přístup mimo virtuální stroj je v plnohodnotných virtuálních strojích zajišťováno pomocí volání nativních metod. Jsou to metody napsané v jiném programovacím jazyce než je *Java* a jsou spuštěny virtuálním strojem přímo na cílovém počítači. Virtuální stroj nemá v tuto chvíli žádné ponětí o tom, co nativní metoda vykonává. Obdrží až výsledek poté, co nativní metoda skončí. Toto je právě bod, na který je potřeba se zaměřit, protože právě pomocí volání nativních metod jsou zajištěny již zmíněné přístupy do operačního systému a jeho zařízení, např. pro zápis na disk či posílání dat na síťové rozhraní.

Pro spuštění maligního kódu je nutnou podmínkou přístup k těmto funkcím, protože je potřeba nejprve nějaký škodlivý kód zapsat na disk, následně jej spustit a v neposlední řadě musí být zajištěna komunikace po síti mezi napadeným systémem a útočníkem. Síťová komunikace ale nebude středem mého zájmu, protože tu zajišťuje již spuštěný maligní program. Mnou analyzovaný program se stará jenom o umístění škodlivého binárního souboru na cílovou napadenou platformu.

Mechanizmus volání nativních metod je ve standardních implementacích 3.7 virtuálních strojů zajišťován přes rozhraní *JNI - Java Native Interface*. Ve zdrojovém kódu jazyka *Java* deklarujeme metodu se všemi parametry i návratovým typem a tuto metodu označíme klíčovým slovem *native*. Toto má za následek to, že virtuální stroj nebude spouštět metody v rámci sebe sama, ale spustí ji přímo na cílové architektuře.

Nativní metoda ovšem k sobě musí mít odpovídající kód proveditelný na cílové architektuře. Hlavička tohoto kódu je vygenerována programem *javah*, podle které je poté napsán kód v cílovém jazyce. Tento kód je následně přeložen tak, aby z něj vznikla dynamicky linkovaná knihovna, která je následně načtena virtuálním strojem a použita při volání nativních metod. Vyvolání správné knihovny má na starosti metoda *System.loadLibrary* ze standardního balíku *Javy*.

Volání nativních metod v rámci standardních tříd *Java Runtime Environment* je řešeno většinou až v těch nejobecnějších třídách, tedy v těch, které jsou v rámci hierarchie dědičnosti na jejím začátku. Pro detekci, že se jedná o zneužitelné metody není nutné chodit tolik do hloubky. Bude stačit určitým způsobem označit ty metody, které nějakou cestou vedou k operacím pracujících přímo s operačním systémem. Následně stačí tato označení zaznamenat a poté podrobit analýze, jestli se jednalo nebo nejednalo o maligní sekvenci těchto operací.

Detekcí bude tedy porovnání vzniklé sekvence s předem připravenými vzory, které budou určovat to, jestli se jedná nebo nejedná o malware.

Analýza testovaného programu probíhá souběžně s vlastní detekcí. Jedná se prakticky jenom o výpis instrukcí, tak jak šly v programu za sebou. Po skončení provádění programu bude mít analytik k dispozici sled instrukcí a může se tedy pohodlně podívat, jakou činnost program udělal. Další možností analýzy je analyzovat jednotlivé třídy. To je využíváno pro psaní falešných tříd i pro vytvoření si představy, co vlastně jednotlivé třídy jsou schopny provádět.

Vlastní analýza a detekce jsou prakticky sjednoceny do jedné funkčnosti. Jelikož mám veškeré potřebné standardní funkce implementovány přímo v mém emulátoru, budu detekovat funkce přímo v rámci jejich volání.

Vždy když zavolám falešnou metodu, zaznamenám do logu jméno funkce, její deskriptor a hlavně její číselný identifikátor. Toto číslo je rozhodující v tom, jestli se bude jednat o škodlivou či neškodlivou sekvenci.

Kapitola 5

Implementace a její popis

V následujících částech této kapitoly bude popsána moje implemetace virtuálního stroje pro spouštění programů napsaných v programovacím jazyce *Java*. Jsou zde podrobněji popsány některé důležité třídy či jejich datové položky nebo význam jednotlivých flagů. Dále jsou vysvětleny i důvody, které mě vedly k použití konkrétních přístupů pro řešení problémů.

Nejprve je dobré si říct, jaké datové typy jsem v mém emulátoru použil. Jak již bylo zmíněno v kapitole 4.4.1, programovací jazyk *Java* má na všech platformách stejnou velikost primitivních datových typů. V mém emulétoru tomu není jinak. Tabulka 5.1 zobrazuje název mnou použitých datových typů i jejich velikosti.

Tabulka 5.1: Názvy a velikosti primitivních datových typů použitých v emulátoru.

Datový typ	velikost	typ v jazyce *C*
Byte	1B	unsigned char
Int8	1B	signed char
Word	2B	unsigned short
Int16	2B	signed short
DWord	4B	unsigned int
Int32	4B	signed int
QWord	8B	unsigned long long
Int64	8B	signed long long

Pro uložení proměnných v plovoucí desetinné čárce používám stejné datové typy, které se standardně používají v programovacím jazyce *C*. Tedy *float* a *double*.

5.1 Načítání dat *.jar* balíčku

Jako první fáze přichází na řadu načítání dat do aplikace. Veškeré potřebné informace jsou uloženy v *.jar* balíčku, jehož formát byl popsán v kapitole 3.3. Jak již bylo zmíněno, *.jar* balíček obsahuje mimo *.class* souborů i další různé soubory jako jsou např. obrázky, textové soubory, zvuky či videa. Tyto soubory nejsou mým emulátorem načítány, protože se zajímám pouze o vlastní provádění programu. Z tohoto důvodu načítám pouze všechny *.class* soubory. Výjimku tvoří pouze soubor *MANIFEST.MF* ze složky *META-INF*, o kterém bude řeč později.

Stejně jak je v kapitole 3.3 zmíněna struktura *.jar* balíčku, tak je tam i popsáno, že se jedná o archiv *.zip*, jenž má jenom změněnou koncovku. Do mého programu ale nevstupuje

celý archiv, ale je mu jen dodána cesta, kde je patřičný archiv rozbalen.

Jedinou výjimkou mezi načítanými soubory je soubor *MANIFEST.MF* ze složky *META-INF*. V něm jsou potřebné informace o balíčku. Nejdůležitějším údajem, který získávám z tohoto souboru je záznam *Main-Class*, podle kterého poznám, která metoda *main* se má začít provádět. Jelikož tento záznam obsahuje plné(*fully qualified*) jméno třídy, pro kterou se provádí statická metoda *main*, dostanu i jméno balíčku obsahující tuto třídu. Stejné jméno je použito i přímo v konkrétním *.class* souboru, a tudíž je ulehčeno vlastní porovnávání. Nemusím totiž nikterak upravovat jméno třídy. Můj program akceptuje za vstup pouze cestu s rozbaleným platným *.jar* balíčkem, tudíž obsahující *MANIFEST.MF* s validním záznamem *Main-Class*. V případě, že záznam *Main-Class* obsahuje jméno třídy, která není v balíčku, program skončí chybou a nic se neprovádí.

Načítání jednotlivých *.class* souborů má na starosti třída *Package*. Do tzv. *root* balíčku vstupuje řetězec obsahující již zmíněnou cestu. Tento balíček následně rekurzivně prochází celou adresářovou strukturu této složky a načítá jednotlivé *.class* soubory. Tyto soubory jsou postupně ukládány do vektoru *Vector< ClassFile * > Package::m_allFiles* a následně zpracovávány. Třída *ClassFile* slouží pro reprezentaci vlastního *.class* souboru a ve skutečnosti pouze uchovává název soubory v rámci souborového systému.

Instance třídy *ClassFile* jsou vstupem do další důležité třídy, kterou je třída *ClassFileStream* odvozená od třídy *BasSeekableStream*. Slouží pro otevírání a zavírání daného *.class* souboru a navíc jako rozhraní pro propojení mého emulátoru s výsledným softwarem. Nejdůležitější činností této třídy je ale načítání obsahu *.class* souborů do paměti a jeho následné zpracování ve třídě *ClassFileProcessing*, která bude podrobněji popsána v kapitole 5.1.1. Třída *BasSeekableStream* obsahuje pouze virtuální metody a slouží jako rozhraní pro napojení mého emulátoru do výsledného softwaru.

Po provedení těchto kroků je načítání dat hotovo a může se spustit vlastní emulace třídou *Emulator*.

5.1.1 Třída *ClassFileProcessing*

Ještě než bude popsána třída *Emulator*, musí být vysvětlen význam a činnost jedné z nejdůležitějších tříd mého emulátoru. Tou je třída *ClassFileProcessing*.

Tato třída totiž zajišťuje načtení každého souboru typu *.class* do paměti a jeho následné zpracování. Data jsou načítána z objektu třídy *ClassFileStream*, což bylo popsáno v předcházející části. Po načtení kompletního obsahu *.class* souboru následuje jeho zpracování.

Pro zpracování dat je implementováno několik metod, které jsou volány v pořadí podle toho, kterou část *.class* souboru zpracovávají. Pořadí jednotlivých částí bylo vysvětleno v kapitole 3.8.

Jelikož formát dat každého *.class* souboru obsahuje položky proměnné velikosti, nemůžeme skákat po položkách pomocí indexů, ale musíme jednotlivé položky načítat postupně jak jdou za sebou.

Kompletní zpracování *.class* souboru provede metoda *processStream*. Ta obsahuje ostatní metody sloužící k parsování jednotlivých datových částí *.class* souboru.

Metoda *processClassHead* zpracuje hlavičku souboru, která se skládá z magické konstanty *0x CAFE BABE* a z dvou verzí. Po hlavičce následuje zpracování *constant pool*-u metodou *processClassConstantPool*. Tato metoda pomocí metody *readConstantPoolEntry* rozparsuje jednu položku *constant pool*-u po druhé a všechny položky postupně uloží do pole položek *ConstantPoolEntry* se jménem *m_constantPool*. Od této chvíle lze již k jednotlivým položkám přistupovat pomocí indexu.

Po zpracování *constant pool*-u jsou na řadě přístupové flagy a informace o jménu aktuální třídy i jménu třídy nadřazené. Obě tyto položky jsou nepřímými referencemi do *constant pool*-u. O zpracování těchto informací se stará metoda *processClassAccessFlags*.

Metoda *processClassInterfaces* zpracovává další část příslušného *.class* souboru, kterou jsou informace o implementovaných rozhraních. V tomto případě se jedná pouze o pole položek typu *Word*. Tato 16-ti bitová čísla jsou indexy do *constant pool*-u, na kterých se nacházejí informace o implementovaných rozhraních. Opět se jedná o nepřímé reference.

Datové položky každé třídy zpracovává metoda *processClassFields*. Stejně jako u metody *processClassConstantPool* i tato metoda načítá jednotlivé datové položky do pole typu *FieldInfoEntry* se jménem *m_fields*. A opět, z důvodu proměnné velikosti záznamů o datových položkách, musíme jednotlivé položky načítat jak jdou po sobě. Po tomto zpracování budou již jednotlivé datové položky přístupné přes index.

Stejný přístup jsem zvolil i při načítání metod a atributů každé třídy. Metody ukládám do pole jménem *m_methods* typu *MethodInfoEntry*. Analogicky jsou uloženy i atributy třídy. Pro pole *m_attributes* je použit typ *AttributeInfoEntry*. O zpracování metod se starají metody *processClassMethods* a *readMethodInfoEntry*. Metody pro zpracování atributů budou tedy analogicky nazvány *processClassAttributes* a *readAttributeInfoEntry*.

Díky tomu, že jednotlivé velikostí polí jsou známy již v době překladu, mohu si alokovat dynamicky tato pole na pevnou velikost. Po tomto zpracování jsou již veškeré položky indexovatelné a je k nim tudíž snadnější a hlavně rychlejší přístup. Hlavně *constant pool* je potřeba mít takto uložený, protože veškeré nepřímé reference na datové položky, metody a atributy jsou realizovány přes indexy tohoto pole.

Jak je uvedeno v kapitole 3.8.2, *constant pool* není indexován od 0, ale od 1. Toto jsem vyřešil použitím makra *CP_INDEX*, které vždy daný index zmenší o 1. Tento krok je nutné použít při každém přístupu do *constant pool*-u, protože ten je v mém emulátoru uložen v poli s indexováním od 0.

Při načtení a zpracování všech *.class* souborů z balíčku přichází na řadu výpočet velikostí jednotlivých tříd spolu s počítáním offsetů pro jednotlivé datové položky obsažené v těchto třídách. Toto má na starosti metody *computeRealSizeAndOffsets* a *computeStaticFieldOffsets*.

5.2 Implementace vlastní emulace

Vlastní emulaci zajišťuje třída *Emulator*. Jsou v ní implementovány statické funkce, které reprezentují jednotlivé instrukce bajt kódu a manipulují s daty uloženými v tabulce *Local Variable Table* nebo v zásobníku operandů. Umožňují vlastní provádění jednotlivých metod obsažených v *.class* souborech.

Tyto funkce jsou uloženy v poli velikosti 256 a každá funkce je na tom místě, které se shoduje s operačním kódem dané instrukce. První implementací byl veliký switch zajišťující volání správné funkce. Mylně jsem se domníval, že použití pole bude rychlejší a méně náročné, ale opak byl pravdou. Toto řešení nepřineslo prakticky žádné zrychlení a pokud, tak naprosto zanedbatelné. Řešení přes velké pole funkcí bylo nicméně ponecháno pro lepší názornost a čitelnost zdrojového kódu.

Jelikož *Java Virtual Machine* je zásobníkovým procesorem, je další velice důležitou položkou třídy *Emulator* zásobník. Tato datová položka se nazývá *m_stack* a je typu *Vector* $<$ *StackFrame* * $>$. Zásobník obsahuje zásobníkové rámce reprezentující jednotlivé volané metody a v průběhu celého provádění programu se jeho velikost mění v závislosti, jak moc se zanořujeme v rámci volání metod.

Začátek programu probíhá uložením počáteční metody, jedná se o metodu *main*, a následném provádění jejího atributu *Code* 3.8.1. Vrchol zásobníku vždy obsahuje zásobníkový rámec aktuálně prováděné metody. Následně se načte první instrukce atributu *Code* dané metody a pokračuje se ve zpracování ostatních instrukcí, přičemž každá instrukce upravuje data podle specifikace[4].

Protože každá vyvolaná metoda je reprezentována konkrétním zásobníkovým rámcem, cyklus provádějící celou emulaci pracuje tak dlouho, dokud datová položka *m_stack* obsahuje alespoň jeden zásobníkový rámec. Jakmile jsou všechny zásobníkové rámce zpracovány, provádění končí.

Další nedílnou součástí každého virtuálního stroje je datová struktura *heap* sloužící pro alokaci objektů a polí. Specifikace *Java Virtual Machine* také udává, že *heap* je spravován automatickým správcem paměti a proto programátor nikdy explicitně neuvolňuje alokovanou paměť. Ostatně k tomu nemá ani žádné prostředky v programovacím jazyku *Java*. Implementace tohoto správce paměti je podrobněji vysvětlena v kapitole 5.8.

5.3 Heap a jeho implementace

Heap je v programu tvořen třídami *ObjectHeap* a *ObjectHandle*. Obě tyto třídy zajišťují správnou alokaci objektů a i jejich úklid v případě, že na tyto objekty již nevede žádná reference.

Třída *ObjectHeap* obsahuje dvě dynamicky alokovaná pole a obě jsou typu *Byte*. První pole *m_objectHeapBegin* slouží pro reprezentaci *heap*-u pro nestatická data. Statický *heap* je realizován polem *m_staticHeapBegin*. Obě tato pole jsou alokována na velikost 8MB, ale lze zvolit i jinou velikost těchto datových struktur. Také není nutné mít stejnou velikost pro obě oblasti, bez probélmů můžeme nezávisle na sobě nastavit odlišnou velikost pro každý *heap* zvlášť. Indentifikátory *m_objectHeapBegin* a *m_staticHeapBegin* nejsou přímo pole, ale pouze ukazatele na začátky pamětí.

Aktuálně volné místo, kam ukládat nová data na statický i nestatický *heap*, udávají ukazatele *m_objectHeapActual* a *m_staticHeapActual*. Konce těchto dvou polí se poté analogicky jmenují *m_objectHeapEnd* a *m_staticHeapEnd*. Jelikož jsem zvolil *garbage collector* zachovávající lokalitu paměti, a tudíž i nulovou fragmentaci, probíhá vždy alokace nového objektu od ukazatele *m_objectHeapActual*. Po úspěšné alokaci se tento ukazatel posune o takovou velikost, kterou zabral nově vytvořený objekt. Odpadá tedy hledání dostatečně velkého volného místa, které musím řešit, pokud je *heap* jakkoliv fragmentovaný.

Díky použití takovéhoto automatického správce paměti můžu vždy rovnou bez přemýšlení alokovat *heap* pro objekt vždy na adresu uloženou v proměnné *m_objectHeapActual*. Samozřejmě vždy kontroluji, jestli nová alokace nezasáhne mimo alokovaný prostor pro *heap*. V takovém případě se volá *garbage collector* a z paměti jsou odstraněny nepotřebné objekty. Používané objekty jsou postupně přesouvány těsně za sebe tak, aby nedocházelo k fragmentaci paměti.

Další významnou datovou položkou této třídy je dynamicky alokované pole typu *ObjectHandle* jménem *m_objectHandlesTable*. Jedná se o pole obsahující informace o tom, kam do paměti ukazuje adresa přiřazená k referenci a jaké flagy jsou nebo nejsou nastaveny pro danou referenci. Reference, používané v celém virtuálním stroji pro identifikaci polí a objektů, jsou indexy do této tabulky. Není proto nutné si nikterak explicitně uchovávat číslo reference pro daný objekt. Nemalou výhodou je i konstatní složitost při přístupu do této datové struktury.

Pro uchování první volné reference slouží atribut této třídy *DWord m_referenceIndex*. Počáteční hodnota tohoto počítadla je 1, protože proměnná uchovává aktuálně první volný index do tabulky *m_objectHandlesTable* a její maximální hodnota je $2^{32} - 1$. Při každé nové alokaci objektu nebo pole je tato proměnná inkrementována. Hodnota *m_referenceIndex* udává i místo, po které se prohledává tabulka *m_objectHandlesTable* při mazání a přesouvání objektů v rámci úklidu paměti. Tím, že reference s číslem 0 udává hodnotu *null* můžeme alokovat maximálně $2^{32} - 1$ objektů.

Velice důležitou komponentou třídy *ObjectHeap* je vektor referencí *m_staticReferences* uchovávající reference, jež jsou uloženy na statickém *heap*-u. Prvky tohoto vektoru jsou vždy všechny označeny jako *root* objekty a nebudou nikdy z *heap*-u odstraněny stejně jako objekty referencované z těchto objektů. Díky tomuto vektoru nemusím procházet jednotlivé použité třídy a hledat v nich reference. Jednoduše se projde tento vektor a tím se získá první část *root* objektů 4.4.3.

Třída *ObjectHeap* mimo datových položek obsahuje i několik metod, které pracují jak s pamětí alokovanou pro objekty, tak i s objekty samotnými. Metody *createObject*, *createArrayObject* a *createMultidimArrayObject* mají za úkol správně alokovat potřebný kus paměti a vrátit číslo reference tam, kde byla volána příslušná instrukce pro vytvoření objektu nebo jedno dimenzionálního nebo více dimenzionálního pole. Tyto instrukce již byly popsány v kapitole 4.4. Poslední metodou alokující data je metoda *createStringObject* vytvářející objekty třídy *java/lang/String*.

Metoda *ObjectHeap::createObject* má parametry typu *Variable*, přes který se vrací nově obsazená reference, a *JavaClass*, který reprezentuje třídu, pro kterou se má vytvořit nová instance. Tato metoda zjistí z objektu *JavaClass* velikost paměti, jenž je potřeba zabrat a tu se pokusí obsadit. Pokud tento proces neuspěje, volá se metoda implementující *garbage collecting*. Velikost získaná z objektu třídy *JavaClass* je navíc doplněna o hlavičku.

Další metodou v pořadí je metoda *ObjectHeap::createArrayObject* implementující vytváření jednorozměrného pole. Stejně jako předešlá metoda obsahují parametry parametr typu *Variable* pro návrat přiřazené reference. Rozdíl je v dalším parametru, který je buď typu *char* nebo *ArrayType*. Oba dva typy sdělují do funkce provádějící alokaci pole jakého typu pole je. Posledním parametrem je délka nově vytvářeného pole.

Nejsložitější funkcí pro alokaci dat na *heap*-u je metoda *createMultidimArrayObject* a to jak počtem parametrů, tak i vlastním tělem metody. Jako první parametr je opět parametr typu *Variable* pro návrat reference. Dále je uveden odkaz na zásobníkový rámec metody vytvářející vícerozměrné pole následovaný indexem do příslušného *constant pool*-u. Tyto dvě informace jsou nutné, protože jinak bychom nebyli schopní získat jméno a deskriptor nově alokovaného pole. Posledním parametrem je 8-mi bitový parametr obsahující počet dimenzí nově vznikajícího multidimenzionálního pole. Postup metody je již lépe znázorněn v kapitole 4.4.1 spolu s obrázkem.

Funkce nejdříve vytvoří pole referencí pro první dimenzi, což je reference, která se vrací zpátky do instrukce *multianewarray* a která se používá dále v programu. Následně funkce alokuje tolik polí, kolik má dimenzí a předchozí pole, pokud to nejsou pole pro poslední dimenzi, jsou správně vyplněna referencemi na nově vytvářená pole.

Jednu větší skupinu tvoří metody pro zápis nebo čtení prvků do nebo z pole. Popíšeme si například metody *writeIntToArray* a *getIntFromArray*. Metoda *writeIntToArray* má tři parametry, první je reference pole, do kterého se zapisuje, druhý je index v poli, jehož hodnota se mění a posledním parametrem je zapisovaná hodnota. Metoda *getIntFromArray* je prakticky totožná s předešlou. Jediný rozdíl je v tom, že hodnota na indexu se předává přes parametr funkce. Podobné metody mám pro veškeré datové typy uvedené v tabulce

4.2, které se od sebe liší jen v typech, které jsou zapisovány nebo čteny.

5.3.1 Třída *ObjectHandle*

Velice důležitou třídou je třída *ObjectHandle*. Tato třída dovoluje použití předávání referencí namísto přímých adres do paměti. Díky tomu, že každá reference obsahuje určitou adresu kde se objekt nachází, nemusíme při jeho přesunu v rámci *heap*-u měnit veškeré jeho odkazy, ale pouze změníme jednu adresu u konkrétní reference.

Jednou z částí této třídy jsou flagy, které můžeme vidět v tabulce 5.2.

Tabulka 5.2: Velikosti primitivních datových typů používaných v jazyce *Java*.

Flag reference	hodnota flagu
OHF_IS_PRIM_ARRAY	0x01
OHF_IS_OBJECT	0x02
OHF_IS_REF_ARRAY	0x03
OHF_GC_VISIT	0x04
OHF_IS_STATIC	0x10
OHF_CP_STRING	0x20
OHF_IS_LIVE	0x40
OHF_IS_VALID	0x08

Použití flagů mi dovoluje využít logických operací k realizaci sloučení více podmínek do jedné jediné komplexní. Tento fakt v důsledku nebude mít výrazný vliv na zlepšení výkonu, protože se *garbage collector* volá jen při nedostatku místa, ale zpřehledňuje samotný kód.

Dále si popíšeme význam jednotlivých flagů. Typ objektu se rozhoduje podle první dvou bitů a hodnoty jsou vidět v tabulce 5.2.

Flag *OHF_GC_VISIT* určuje, jestli byl objekt navštíven v rámci aktuálního běhu *garbage collector*-u. Jestliže ano, objekt se již v rámci úklidu paměti dál nezpracovává. Díky tomuto flagu jsou vyřešeny smyčky v rámci referencí mezi objekty.

V případě, že je nastaven flag *OHF_CP_STRING*, objekt je řetězec a není uložen na *heap*-u, ale v *constant pool*-u nějaké třídy. Takovýto řetězec i jeho délka je znám již při překladu zdrojových *.java* souborů a není nutné ho mít navíc uložen na *heap*-u.

Pokud je daný objekt živý, tedy dostupný z nějakého *root* objektu, jeho reference má nastaven flag *OHF_IS_LIVE* a při mazání objektů je takovýto objekt v paměti ponechán, případně přesunut na nové vhodnější místo.

Flag *OHF_IS_VALID* značí platnost nebo neplatnost aktuální reference. Při inicializaci tabulky *m_objectHandlesTable* nemá žádná reference nastaven tento flag. Ten je nastavován až při alokaci nového objektu nebo pole přes příslušné instrukce. Pokud reference nemá tento flag nastaven, je přeskočena *garbage collector*-em při úklidu.

5.4 Volání metod

Volání metod je v bajt kódu programovacího jazyka *Java* realizováno instrukcemi *invokespecial*, *invokevirtual*, *invokestatic* a *invokedynamic*. Zde si popíšeme pouze první tři zmíněné instrukce. Instrukce *invokedynamic* nebude popsána, protože slouží pro volání funkcí, u kterých nevíme v době překladu prakticky nic. Tyto funkce jsou obvykle používány v dy-

namicky typovaných jazycích, obvykle skriptovacích, které jsou překládány do bajt kódu spustitelného ve virtuálních strojích *Java Virtual Machine*.

Za každou ze tří zmíněných instrukcí, které si popíšeme, následuje 16-ti bitový index do příslušného *constant pool*-u. Na této pozici musí být položka *CONSTANT_Methodref_info*. Z této položky zjistíme třídu, jméno a deskriptor metody, což jsou veškeré údaje, které jsou potřeba k nalezení této metody. Jelikož se jedná o řetězcové konstanty, jejichž porovnávání je procesorově náročné, je po zjištění konkrétní metody uložen v proměnné *MethodInfo-Entry *m_classMethod* odkaz na tuto metodu. Pokud tedy bude metoda volána vícekrát, nebude nutné znovu hledat metodu pomocí porovnání řetězců, ale vezme se obsah zmíněné proměnné reprezentující přímo volanou metodu.

V případě volání metod je důležité mít i deskriptor dané metody, protože teprve podle něj zjistíme, kterou metodu přesně vyvolat. V programu může být více metod se stejným názvem lišícím se pouze počtem a typy parametrů. Tento fakt je jedním ze základních rysu polymorfizmu programovacího jazyka *Java*.

První instrukcí, kterou si popíšeme, je instrukce *invokespecial*. Slouží pro volání metod, které jsou nějak speciální. Jde o konstruktory všech tříd, privátní metody a metody, které jsou zděděné z nějaké rodičovské třídy. Při volání neprivátních metod z nějakého rodiče je také použita tato instrukce.

Další instrukcí je instrukce *invokevirtual*. Tato instrukce řeší volání obyčejných metod v programu *Java*. Tedy těch metod, které jsou označeny modifikátorem *public* a nachází se v aktuální třídě.

Poslední popisovanou instrukcí bude instrukce *invokestatic*. Jak je již z názvu patrné, instrukce *invokestatic* volá pouze statické (třídní) metody, tedy metody označené klíčovým slovem *static* v kódu programovacího jazyka *Java*. Volání statické metody je prakticky stejné, jako volání metody nestatické, ale s tím rozdílem, že při volání statické metody se nekopíruje implicitní parametr *this*.

Metody implementující jednotlivé instrukce pro volání metod se v emulátoru jmenují velmi podobně jako instrukce, které reprezentují. Jsou implementovány jako statické metody třídy *Emulator* a nesou názvy *inst_invokespecial*, *inst_invokevirtual* a *inst_invokestatic*. Jako první věc při provádění těchto funkcí je načtení 16-ti bitového indexu, následuje vyhledání vlastní metody, pokud už není vyhledaná a poslední částí je samotné spuštění metody.

Parametry nově volané funkce musí být uloženy na zásobníku operandů volající funkce ve správném pořadí. Toto pořadí kontroluje metoda *StackFrame::checkStack* podle deskriptoru metody, který je předán do metody *checkStack* jako parametr. V případě, že metoda *checkStack* skončí s chybou, končí i celé provádění emulace. Jinak se pokračuje ve zpracování metody. Je vytvořen nový objekt typu *StackFrame*, jeho tabulka *Local Variable Table* je naplněna daty z *Operand Stack*-u volající metody a následně je nově vytvořený zásobníkový rámec umístěn na vrchol prováděcího zásobníku. A jelikož je vždy brána následující instrukce metody, jejíž zásobníkový rámec je na vrcholu zásobníku, začne se provádět nově vyvolaná metoda. Nutno poznamenat, že všechny proměnné ze zásobníku operandů reprezentující parametry nově volané metody, jsou z tohoto zásobníku odstraněny.

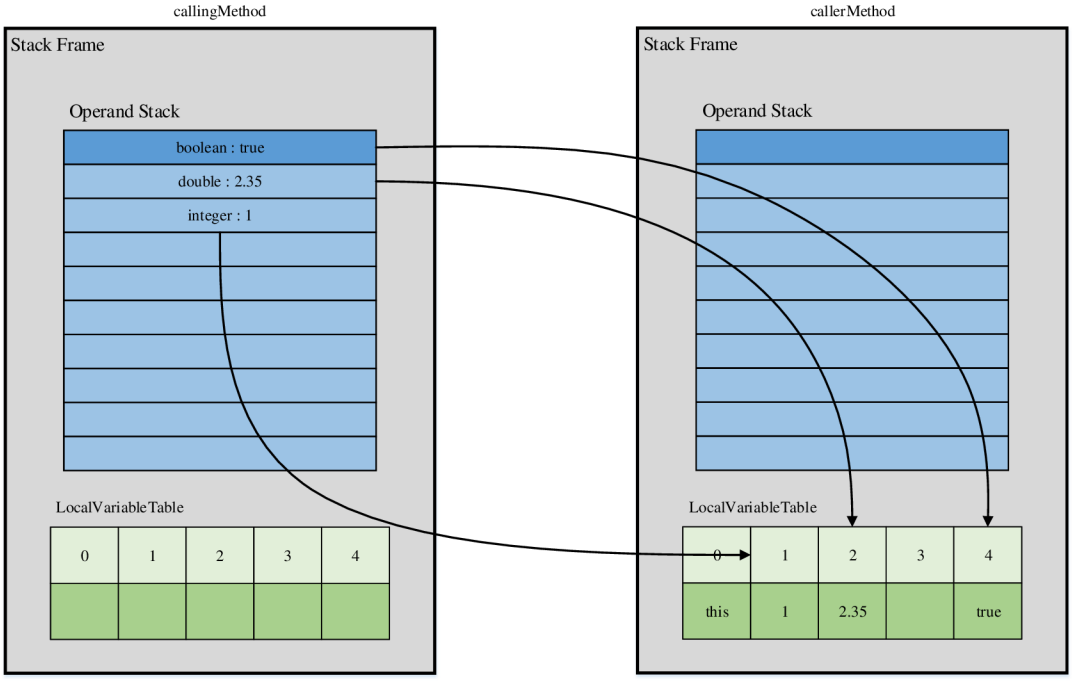
Pro lepší představu mějme následující dvě metody *callingMethod*, která volá metodu *callerMethod*.

Na obrázku 5.1 je vidět jak jsou přesunuta data z *Operand Stack*-u metody *callingMethod* do tabulky *Local Variable Table* metody *callerMethod*. Zde je opět k vidění analogie s uložením konstant typu *long* a *double* v *constant pool*-u, kde tyto typy zabírají dvě místa. Při kopírování dat z *Operand Stack*-u volající metody do tabulky *Local Variable Table* volané metody dochází také k obsazení dvou položek tabulky v případě typů *long* a

```
void callingMethod()
{
  ...
  callerFunction(1, 2.45, true);
  ...
}
```

```
void callerMethod
  ( int i, double d, boolean b )
{
  ...
}
```

double. I v tomto případě tvůrci specifikace *Java Virtual Machine* přiznali, že se nejednalo o nejšťastnější řešení.



Obrázek 5.1: Přesun dat při volání metody z jiné metody.

Obrázek 5.1 znázorňuje volání nestatické metody. V tomto případě je na první položku tabulky *Local Variable Table* na indexu 0 kopírován implicitní parametr *this*. Pokud by volaná metoda byla statická, tento implicitní parametr by se nepředával a všechny položky v tabulce *Local Variable Table* by byly o jedno posunuté doleva.

Žádná přístupová práva aktuálně nekontroluji, protože očekávám vždy takový program, který by toto zkorumpované neměl. Program, který je podroben analýze mým emulátorem musí být totiž schopný samostatně běžet v jakémkoliv *Java Virtual Machine*. Proto očekávám vždy korektní a bez problémů spustitelný *.jar* balíček.

Účel zásobníkových rámců a registru PC

Nejprve je nutné poznamenat, že *m_registerPC* typu *DWord* je datovou položkou třídy *StackFrame*. Objekty třídy *StackFrame* reprezentují kontext konkrétní metody. Jsou uloženy na hlavním zásobníku celého emulátoru, z něhož se vždy zpracovává pouze vrchol zásobníku, tzv. *Top of Stack*. Vrchol zásobníku udává aktuálně prováděnou metodu.

V mém emulátoru se funkce registru *PC* trochu liší. Nemám totiž jenom jeden v rámci emulátoru, ale používám vždy jeden v rámci metody. Každá metoda má tedy svůj vlastní *m_registerPC* uložený ve svém zásobníkovém rámci a při provádění metody se inkrementuje nebo dekrementuje tento registr podle potřeby. O kolik se má *m_registerPC* inkrementovat nebo dekrementovat rozhoduje počet operandů dané instrukce nebo, v případě skoků, instrukce pro skoky.

Inkrementace probíhá vždy po provedení nějaké instrukce metody a je způsobeno jejím sekvenčním prováděním. Zmenšování tohoto čísla je způsobeno instrukcemi pro řízení běhu programu, které jsou použity například v cyklech, v nichž je potřeba vracet se na dřívější bod v rámci provádění. Tyto instrukce jsou konkrétně popsány v kapitole 5.6.

Registr *m_registerPC* obsahuje číslo instrukce, která bude provedena jako další instrukce v pořadí. Instrukce v bajt kódu programovacího jazyka *Java* nemají všechny stejný počet operandů. Každá konkrétní instrukce tedy posune tento registr tak, jak je uvedeno ve specifikaci[4].

5.5 Návrat z metod

Konec metod, tedy jejich návrat do volajících metod, zajišťují instrukce *areturn*, *ireturn*, *freturn*, *dreturn* a *return*. V programovacím jazyku *Java* odpovídá těmto instrukcím příkaz *return* a podle deskriptoru metody překladač *javac* vygeneruje patřičnou instrukci.

Hodnota, která se vrací, je uložena na vrchol zásobníku operandů metody, která aktuálně končící metodu vyvolala. Jakmile dojde k tomuto uložení, je z hlavního zásobníku odstraněn vrchol, což přepne kontext na volající metodu.

5.6 Řízení běhu programu

Tyto instrukce přímo ovlivňují datovou položku aktuálního zásobníkového rámce metody *m_registerPC*. Operand, který následuje za tímto typem instrukcí určuje o kolik a jakým směrem se bude proměnná *m_registerPC* posouvat od pozice aktuální instrukce. Jedná se o 16-ti bitovou znaménkovou hodnotu udávající tento posun. Pokud je tato hodnota záporná, skáče se v programu na instrukci blíže začátku. Pokud je kladná, skok vede blíže ke konci metody.

V obou dvou případech dochází ke kontrole, jestli se neskáče mimo instrukční kód dané metody.

Těmito instrukcemi jsou řešeny podmínky a cykly v programovacím jazyce *Java*.

5.7 Zpracování výjimek

Jako mechanizmus ošetření chyb v programovacím jazyku *Java* slouží výjimky. Výhodou oproti ošetřování chybových stavů návratovými kódy a jejich kontroly přes podmínky je to, že do jednoho bloku pro kontrolu, jestli nenastala nějaká výjimka, můžeme zahrnout více operací končící nějakou chybou. Nemusíme tedy ošetřovat každou operaci zvlášť. Za tento komfort ale zaplatíme o něco více náročnějším zpracováním.

Oblast, kterou chceme hlídat, je uzavřena do bloku *try*. Tento blok je následován několika bloky *catch* odchytávajícími výjimky, které by mohly nastat. Tyto jednotlivé bloky *catch* se provádějí postupně, tudíž je velice důležité zvolit správné pořadí. Pořadí se promítne i do bajt kódu každé metody, což bude popsáno v následujících odstavcích. Poslední nepovinnou

částí je blok *finally*, který se provede vždy bez ohledu na to, jestli došlo k nějaké výjimce nebo nikoliv.

Atribut *Code* každé metody může, ale nemusí obsahovat pole s názvem *exception_table*, které má následující strukturu[12].

```
u2 exception_table_length;
{   u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
} exception_table[exception_table_length];
```

Jelikož je *exception_table_length* velikosti 16 bitů, je možné zachytit v jenom *try* bloku 2^{16} typů výjimek. Hodnoty proměnných *start_pc* a *end_pc* udávají platnost daného typu výjimky v rámci bajt kódu příslušné metody. Typ výjimky zjistíme z čísla *catch_type*, což je index do patřičného *constant pool*-u, na kterém musí být záznam *CONSTANT_Class_info*[12]. Poslední položkou je *handler_pc*. Jedná se o pozici v bajt kódu dané metody, na které se skočí, pokud nastane patřičná výjimka.

Třída *MethodInfoEntry* obsahuje pole položek *Word* jmenující se *m_exceptionTable*, které zastává funkci tabulky *exception_table* zmíněné v předchozích odstavcích. Výjimka je vyhozena vždy nějakou konkrétní instrukcí bajt kódu jazyka *Java*. Pro příklad uveďme instrukci *idiv* provádějící dělení dvou celočíselných operandů. Pokud je v tomto případě druhý operand roven 0, je vyhozena výjimka typu *ArithmeticException*.

V mém emulátoru zajišťuje zpracování metoda *throwException* ze třídy *StackFrame* a vždy je volána pro zásobníkový rámec metody, kde se výjimka vyskytla. Tato metoda se podívá, která výjimka nastala a jestli je pro ní handler. Pokud ano, tak se skočí na správnou instrukci, která je uložena v již zmíněné tabulce. Pokud handler chybí, výjimka je delegována do zásobníkového rámce nadřazené metody.

Ještě před skokem na správnou instrukci se musí vytvořit objekt obsahující data výjimky. To se děje standardně přes metodu *ObjectHeap::createObject*. Reference na tento objekt je uložena na vrchol zásobníku operandů, což má za následek, že se můžou data z neočekávaného průběhu šířit dále skrz zásobníkové rámce. Záleží poté jenom na programátorovi, jak s daty ze vzniklé výjimky naloží.

5.8 Implementace automatické správy paměti

Implementace algoritmu *Mark & Compact* sestává z několika částí, které si v následujícím textu podrobně popíšeme. Lepší představu z jakých částí se *garbage collector* může skládat je uveden následující pseudokód:

```
nalezeni_root_objektu();
faze_mark();
mazani_a_presun_zivych_objektu();
```

První fází je fáze *Mark*, tedy označování živých a neživých objektů. Nejprve je všem validním záznamům v tabulce *m_objectHandlesTable* nasteaven flag na *OHT_FLAG_VISIT* na 0, což nám všechny objekty označí jako nenavštívené *garbage collectorem*. Dále si vypočteme veškeré *root* objekty, což nám zajistí startovací bod použitého algoritmu. Takovéto objekty postupně procházíme s tím, že jsou následně procházeny i jejich datové položky.

Pokud jsou v nich nalezeny nějaké další reference, které mají *OHT_FLAG_VISIT* nastaven na 0, jsou zařazeny do zpracování stejně jako *root* prvky na začátku. Při zahájení zpracování nějakého obyčejného nebo *root* objektu je tomuto objektu nastaven *OHT_FLAG_VISIT* flag na 1. Toto zajišťuje správné označení všech objektů, i když mezi sebou objekty mají udělány smyčky mezi referencemi. Po skončení fáze označování začíná fáze, která uvolní paměť pro objekty, ke kterým už se z programu nedá dostat a přesune objekty těsně za sebe tak, aby tvořily jednolitý celek.

Toto provádím tak, že si stále uchovávám adresu prvního volného místa. Na tuto adresu se budou kopírovat data, která jsou reprezentací živého objektu. Pokud smažu nějaký objekt, tak první volné místo je adresa začátku paměti pro tento odstraňovaný objekt. Jakmile opět narazím na živý objekt, použiji uchovanou adresu a na tu to adresu přesunu data tohoto objektu. Adresa začátku volného místa je posunuta o to, kolik daný objekt zabíral místa. Jelikož je moje implementace *heap*-u realizována jako pole bajtů, stačí k adrese přičíst velikost přesouvaného objektu v bajtech. Při této operaci musím také patřičně upravit záznam v tabulce *m_objectHandlesTable*.

Když takto projdu veškeré validní reference z tabulky *m_objectHandlesTable*, je *heap* uklizen, plný živých objektů a bez jakékoliv fragmentace.

Všechny čtyři fáze nemám oddělené v rozdílných metodách, ale jsou všechny implementovány za sebou v jedné funkci s názvem *runGarbageCollector*. Toto řešení ušetří několik volání funkcí a tím i způsobí úsporu času.

5.9 Robustnost aplikace

Jednou z podmínek byla robustnost celého řešení. Jelikož celý program bude později zakomponován do složitějšího softwaru, nesmí nastat za žádnou cenu jakýkoliv pád aplikace. Musí být ošetřeny všechny možné i nemožné stavy, které mohou nastat. Vše je řešeno vracením návratových kódů a jejich správným ošetřením. Ovšem jenom tam, kde je to potřeba. Je zbytečné navracet z metod návratový kód tam, kde to nemá smysl. Například ve funkcích provádějících pouze přiřazování hodnot do proměnných primitivních datových typů.

Musí být ošetřeny veškeré dynamické alokace polí a objektů přes operátor *new*. V tomto případě je za operátorem *new* ještě uvedena konstanta *(std::nothrow)*, která zajistí, že při nezdařené alokaci nebude vyvolána výjimka *bad_alloc*, ale bude se vracet konstanta *NULL* do nově alokované proměnné. Příkaz *new* se bude chovat prakticky jako příkaz *malloc* z jazyka *C*. Tento stav už lze bez problémů ošetřit i bez použití výjimek. Jedinou nevýhodou tohoto řešení je, že při vícero alokacích musím ošetřovat každou zvlášť a nemůžeme je všechny uzavřít do bloku *try*.

V případě, že jakákoliv metoda vrací nějaký návratový typ jedná se o typ *ErrorCode*. Je to výčtový datový typ, který pouze zpřehledňuje zápis chybových kódů. Spolu s tímto typem mám implementována dvě pomocná makra s prakticky totožnou funkcionalitou. Makro *HANDLE_ERROR* má jako parametr text, který má způsobená chyba vypsat na standardní výstup, a hodnotu, podle které se rozhodne, jestli se jedná nebo nejedná o chybu. Druhým makrem je makro s názvem *HANDLE_ERROR_CLEAN*, které má stejné parametry jako předchozí uvedené a navíc je ve třetím parametru předávána funkce, která provede správné uvolnění alokované paměti. Obě dvě makra také vypíší jméno zdrojového souboru i řádek, na kterém chyba vznikla. Navíc na svém místě zavolají příkaz return s patřičným chybovým kódem a prováděnou funkci ukončí.

5.10 Implementace falešných metod a tříd

Jednou z hlavních částí mého emulátoru byla implementace tzv. falešných tříd, metod a datových položek. Jsou to, jinak řečeno, informace o třídách, které jsou v obvyklých virtuálních strojích dynamicky načítány z balíčku standardních knihoven umístěného v instalační složce konkrétního virtuálního stroje a běhového prostředí *Java Runtime Environment*. Ve standardní instalaci *Java Runtime Environment* od společnosti *Oracle*[13] se tento balíček nachází ve složce *C:\Program Files\Java\jre7\lib* a jmenuje se *rt.jar*.

Základní princip *fake* tříd spočívá v tom, že nic není načítáno dynamicky, ale vše je obsaženo v paměti již při spuštění mého emulátoru. Jenom toto má za následek zrychlení celkového provádění. Jak bylo popsáno v kapitole 5.1.1, offsety jednotlivých datových položek každé třídy načtené z *.jar* balíčku jsou počítány jakmile načteme veškeré *.class* soubory z testovaného balíčku. V případu falešných tříd se již nemusí nic počítat, protože je vše již vypočítáno předem.

Nejvýraznější zrychlení je ovšem dosaženo použitím falešných metod. Ty jsou totiž implementovány přímo v jazyce *C++* a jejich kód je upraven na míru mého emulátoru. Jelikož je kód spuštěn přímo, vyhneme se tím spouštění metod přes emulátor a jeho zásobníkové rámce jak je to popsáno v kapitole 5.4.

Ústřední třídou pro implementaci falešných metod, ale i metod načtených z *.class* souborů, je třída *Method*. Od této třídy totiž dědí jak třída *MethodInfoEntry*, jenž už byla popsána, tak i třída *FakeMethod*, kterou si popíšeme vzápětí. Třída *Method* obsahuje datové položky, které jsou společné pro oba dva typy metod a navíc obsahuje i virtuální metody, které musí být implementovány v obou dvou podtřídách. Tento fakt zajišťuje to, že všude mohu předávat typ *Method* a nemusím rozlišovat, jestli se jedná o metodu z balíčku či falešnou metodu.

Velice důležité jsou metody pro získání jména a deskriptoru dané metody. Ty jsou reprezentovány textovým řetězcem a jeho délkou. Porovnání dvou metod probíhá pomocí porovnání jejich řetězcových konstant jména a deskriptoru. Tento fakt se může zdát jako naprosto neefektivní, ale jelikož je vždy známa délka každého řetězce, porovnáme nejprve jejich délky a jen jestli jsou stejné, přejdeme k vlastnímu porovnání řetězců přes funkci *strncmp*. Takto jsou prováděny veškerá porovnání řetězců v mém emulátoru.

V tomto místě mého emulátoru jsem dosáhl největšího zrychlení celého provádění. Jelikož *fake* metody jsou napsány v jazyce *C*, jejich provádění bude mnohokrát rychlejší než u metod načtených z *.jar* balíčku, které jsou prováděny emulátorem. Implementovány jsou převážně metody pro práci se třídami *java/lang/String*, *java/lang/StringBuilder* a *java/lang/StringBuffer* a metody tříd zajišťující práci se soubory, adresáři a registry. Na tomto místě jsou zaznamenávány informace o potenciálním maligním chování testovaného programu. Vše je podrobněji popsáno v následujících kapitolách.

Stejné jako falešné metody vystupují i falešné třídy a datové položky. V případě tříd je rodičovskou třídou třída *JavaClass*. Opět obsahuje společné datové položky pro oba potomky, třídy *ClassFileProcessing* a *FakeClass*. Nejdůležitějším společným prvkem obou tříd je velikost třídy. Ta slouží pro alokaci správné velikosti dat na *heap*-u. I u implemetace tříd jsou společné abstraktní metody, které musí být implementovány v obou třídách. Ve vlastním emulátoru potom můžu používat přímo třídu *JavaClass* a nemusím složitě zjišťovat, zdali se jedná o metodu z balíčku nebo o falešnou metodu. Prostě zavolám abstraktní metodu a buď se zavolá metoda ze třídy *FakeClass* nebo *ClassFileProcessing*.

Objekty tříd *FakeClass* a *ClassFileProcessing* obsahují navíc pole metod a pole datových položek, ve kterých probíhá hledání patřičných metod nebo datových položek při provádění

jednotlivých instrukcí.

Stejné je to i v případě datových položek. Zde je hlavní třídou třída s názvem *Field* a odvozené potom *FakeField* a *FieldInfoEntry*. Jejich použití je poté stejné jako u tříd či metod.

5.11 Implementace analýzy a detekce

Analýza a detekce je implementovaná v metodách třídy *Emulator* a také ve třídě *JavaStdMethods*, kde jsou implementovány falešné metody. Pokud program volá nějakou metodu, která má vliv na to, jestli se započítá do testované sekvence, je toto volání metody zaznamenáno do logu. Tento log obsahující sekvenci potenciálně maligní činnosti je následně porovnáván oproti určitým předem připraveným sekvencím, o nichž víme, že škodlivé jsou. Tyto sekvence jsou uloženy v poli *Sequences::m_sequences*.

Do výsledného logu jsou zaznamenávány volané metody tak jak šly za sebou. Tímto zápisem vznikají různě dlouhé sekvence provádění. U funkcí, které se neúčastní škodlivého chování, není žádný záznam prováděn, tudíž se ani do vlastního porovnávání nedostanou.

Log, o kterém byla v minulých odstavcích řeč, není skutečným logem. Jedná se o vektor v paměti, který je plněn při provádění testovacích programů. Tento log není nijak fyzicky zapsán do souboru, jak tomu obvykle v těchto případech bývá.

Pokud by testování, zdali se jedná nebo nejedná o špatnou sekvenci, dělali po každém zápisu do nově vytvářené sekvence, byla by tato operace náročná a příliš by zatěžovala celkové provádění. Proto jsem se rozhodl provádět porovnání sekvencí po provedení většího počtu zápisu do nově vytvářené sekvence. Tím číslem, pro které jsem se rozhodl, je číslo 5.

Jakmile je délka nově vytvářené sekvence dělitelná 5, číslo po číslu ji porovnám a pokud se do nějaké sekvence trefím, končím emulaci s výsledkem, že se jedná o malware. V opačném případě se pokračuje dále v provádění. Zvolil jsem číslo 5, protože nejlépe dokáže prezentovat princip detekce na příkladech.

Vlastní porovnání aktuální sekvence se škodlivými sekvencemi zajišťuje statická metoda třídy *Sequence* jménem *isMaliciousSeq*. Tato funkce navrátí *true* v případě sekvence reprezentující maligní chování a *false* v opačném případě.

Hlavní třídou, která obsahuje informace důležité pro detekci škodlivého chování, je třída *Sequences*. Je v ní uloženo dvourozměrné pole typu *DWord* jménem *m_sequences* obsahující číselné posloupnosti, které reprezentují jednotlivé vzorky špatného chování. Každý řádek této tabulky je jedna sekvence maligního chování.

Jádrem detekce je tedy funkce třídy *Sequences* jménem *isMaliciousSeq*. Ta provádí vlastní porovnání jednotlivých sekvencí takovým způsobem, aby co nejefektivněji prošla všechna data a porovnala je s aktuální sekvencí. Tato funkce je schopná se vypořádat s různě dlouhými sekvencemi a dokonce mapovat menší sekvence doprostřed těch větších.

Díky výstupu této funkce můžu v prováděcí smyčce emulátoru lehce rozpoznat, jestli se jedná o škodlivou či neškodnou sekvenci, a v případě škodlivé sekvence celou emulaci ukončit.

Kapitola 6

Testování a vyhodnocení výsledků

Rychlost provádění programů napsaných v programovacím jazyku *Java* v mém emulátoru se velice složitě testovala, ale i tak přináší docela zajímavé výsledky. Je pravdou, že jsem možná porovnával neporovnatelné. Virtuální stroj *HotSpot* společnosti *Oracle* již prošel mohutným vývojem a velké množství kódu je maximálně optimalizováno.

Co se týká testování robustnosti mého řešení bylo prováděno tak, že všude tam kde se vyskytovalo makro *HANDLE_ERROR* nebo *HANDLE_ERROR_CLEAN* byl napevno předán chybový kód jako kdyby došlo ke konkrétní chybě. Ve všech případech program skončil chybou a veškeré zdroje byly správně uvolněny. Nedocházelo ani k únikům paměti, na což jsem se pečlivě zaměřoval již od začátku návrhu. K testování, jestli nedochází ke špatným dealokacím paměti, jsem využíval knihovnu *CRT Debug*. Program musí být ovšem přeložen spolu s debugg informacemi.

Hlavním prvkem, který jsem prakticky testoval během celého vývoje bylo vlastní provádění programů v mém emulátoru. Musel jsem otestovat veškeré instrukce, které virtuální stroj normálně používá. Na přiloženém médiu jsou přiloženy příklady programů, na kterých je dobře patrná správnost provádění jednotlivých instrukcí. Jsou zde příklady pro testování aritmetických operací, řízení běhu programu, průběhu vykonávání cyklů či alokace dat na *heap*-u. Podrobný obsah přiloženého datového média je uveden v příloze C.

Pěkné jsou příklady testující správné alokace dat na *heap*-u a to jak na statickém, tak i nestatickém, nebo příklady testující automatickou správu paměti. Pro zobrazení obsahu obou dvou *heap*-ů je nejlepší spustit program přímo ve vývojovém prostředí *Visual Studio*, zastavit si program na správném místě a poté si zobrazit obsah jedné či druhé paměťové oblasti. Klávesová zkratka *Alt+3* zobrazí debuggovací okno *Watch*. Do jakéhokoliv řádku vepíšeme *Emulator::m_objectHeap*, přes který se dostaneme jak k adrese statického, tak i nestatického *heap*-u. Stačí si již vybrat adresu, zkopírovat ji do schránky a zobrazit přímo obsah paměti pomocí klávesové zkratky *Alt+6* s vložením adresy do adresního řádku. V tomto pohledu je také krásně vidět i průběh alokace či dealokace paměti pro objekty a pole.

6.1 Testování rychlosti

V prostředí Windows jsem pro spouštění programů v *Javě* používal virtuální stroj *HotSpot* společnosti *Oracle*[13] a to v 64 bitovém sestavení. V prostředí Linux byl testovací program spouštěn opět na 64 bitovém sestavení virtuálního stroje *IcedTea* ve verzi 1.12.5.

Můj emulátor je obecně multiplatformní, protože používám pouze funkce a knihovny z obecné specifikace jazyka *C++*. Jediná závislá část je procházení souborového systému

a načítání *.class* souborů do paměti emulátoru. Z tohoto důvodu nelze přeložit ani spustit můj emulátor v jiném prostředí než vytváří *Microsoft Visual Studio*, ve kterém probíhal a bude nadále probíhat celý vývoj.

Emulátor byl testován v *Release* sestavení, kde jsou vypnutá veškerá nastavení překladače pro podporu debuggingu a vypnutá makra *assert* pro kontrolu práce programátora.

V prostředí Windows bylo měření času prováděno utilitou *measure-command* v terminálu *PowerShell*, v Linuxu byl čas měřen programem *time*.

Testování probíhalo na počítači s procesorem *AMD Athlon X2 240* se 4GB operační paměti a to jak na systému *Windows*, tak i na platformě *Linux*. Systém *Windows* byl konkrétně ve verzi *Windows 7 Professional 64*. Pro testování v *Linuxu* jsem používal systém *Ubuntu x86_64* s nenáročným grafickým prostředím a jádrem 3.5.0-18-generic. Při veškerých testováních byly u obou systémů vypnuty veškeré nepotřebné služby i programy, které by mohly mít vliv na výsledky měření. Testování také probíhala bez přístupu do sítě, tudíž nemohla být ovlivněna ani případnými požadavky na aktualizaci systému.

Následující jednoduchý zdrojový kód byl použit pro vlastní testování rychlosti provádění. Pro lepší představu o nárocích jednotlivých variantách virtuálních strojů jsem testoval od 1 milionu opakování až po 20 milionů opakování. Je zde výborně vidět, že virtuální stroje *HotSpot* a *IcedTea* jsou optimalizovány co možná největším způsobem. Ani při jednom testu nepřesáhla doba provádění 0.2 sekundy a i časy jednotlivých testů se nemění, i když jsou v nich veliké rozdíly co se týká počtu opakovaní vnitřního cyklu.

Můj emulátor byl testován ve dvou variantách. V první variantě archiv obsahoval *.class* soubory pro obě dvě třídy. V druhé variantě byl odebrán z balíčku soubor *Vypocty.class* z důvodu použití statické metody *secti* jako metody falešné a tedy implementované přímo v emulátoru.

```
class Vypocty
{
    static public long secti( long a, long b )
    {
      return ( a + b );
    }
}

public class Cyklus
{
    public static void main( String[] args )
    {
        long vysledek = 0;
        for( long i = 0; i < 1000000; i++ )
        {
            vysledek = secti( vysledek, i );
        }

        System.out.println( vysledek );
    }
}
```

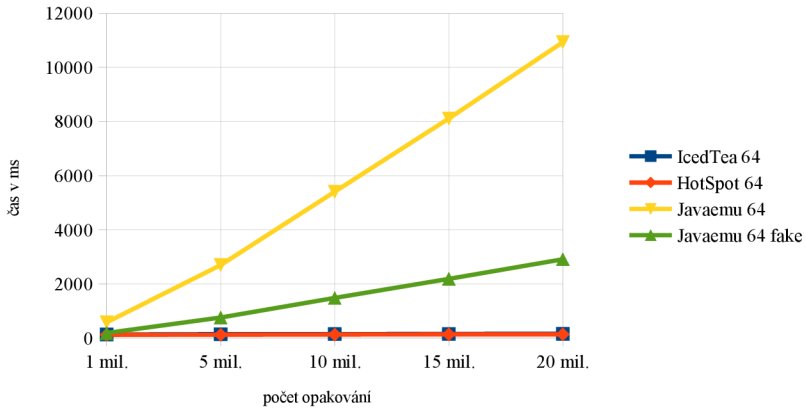
Testování probíhalo vždy 10 krát pro každý počet opakování cyklu a z naměřených hodnot byl následně vypočítán aritmetický průměr. Výsledky jsou uvedeny v tabulce 6.1 a

graficky znázorněny grafu 6.1.

Tabulka 6.1: Rychlost provádění jednoduchého programu v jazyce *Java*.

Varianta JVM	1 mil.	5 mil.	10 mil.	15 mil.	20 mil.
IcedTea 64b	0.134 s	0.145 s	0.149 s	0.154 s	0.158 s
HotSpot 64b	0.124 s	0.126 s	0.131 s	0.138 s	0.142 s
JavaEmu 64b	0.583 s	2.701 s	5.406 s	8.114 s	10.936 s
JavaEmu 64b fake funkce	0.185 s	0.759 s	1.485 s	2.189 s	2.913 s

V grafu na obrázku 6.1 je pěkně vidět, jaké jsou rozdíly ve vykonávání výše uvedeného programu.



Obrázek 6.1: Rozdíly provádění programu na různých virtuálních strojích.

6.2 Testování analýzy a detekce

Testování vždy probíhalo na několika programech napsaných v programovacím jazyku *Java*. Nejprve jsem si myslel, že budu moci aplikaci otestovat na ostrých maligních vzorcích, ale opak byl nakonec pravdou. Nepodařilo se mi získat žádný škodlivý program, který by byl kompletním balíčkem *.jar*. Vždy se jednalo o fragmenty spustitelných balíčků. Buď chyběl soubor *MANIFEST.MF*, nebo neobsahoval záznam *Main-Class* a nebo se jednalo o soubor několika *.class* souborů, které spolu prakticky nesouvisely.

Pro účely testování jsem si tedy vytvořil několik testovacích programů. Tyto programy ve skutečnosti žádnou škodlivou činnost nedělají, ale plně testují můj emulátor. Především jsem se zaměřil na to, které operace vedou k nějakému malignímu chování. Cílem škodlivého softwaru je nějakým způsobem poškodit napadený systém. Ať už z něj vytvořit bot a přidat jej do nějakého botnetu nebo převzít nad napadeným systémem kontrolu. Jsou v nich použity konkatenace řetězců, zápisy do souborů, vytváření adresářů, editaci registrů či komunikace po síti. Třídy, které se podílejí na škodlivém chování jsou znázorněny v příloze A. Veškeré mnou vytvořené programy v programovacím jazyku *Java* sloužící pro otestování vlastní detekce jsou uloženy na přiloženém paměťovém médiu ve složce *DETECTION_TESTS*.

V aktuální implementaci mého emulátoru je pro ukázku detekováno jen menší množství metod z malého počtu tříd. Ve skutečnosti se detekce bude účastnit mnohem větší množství tříd a výsledky detekce budou porovnávány s mnohem větším počtem možných sekvencí. Pro názornou ukázku funkčnosti detekce a analýzy jsou však tyto příklady naprosto dostačující.

Z toho, jak jsou tyto metody volány za sebou, vytvořím výslednou sekvenci, která je nakonec porovnávána s předem připravenými sekvencemi udávajícími to, jestli se jedná nebo nejedná o maligní chování.

Veškeré třídy a metody, které jsem používal pro tvorbu sekvencí, jsou uvedeny v příloze A.

Jakmile spustím detekci a program zdárně doběhne do konce, je zobrazeno hlášení o výsledku. Tedy informace o povaze testovaného programu a tedy to, co bylo účelem tohoto projektu. Tato informace se již dále nikterak nezpracovává. V plánu je ovšem zapojení mého emulátoru do většího softwaru, ve kterém by tato informace měla posloužit pro další zpracování.

Kapitola 7

Závěr

Úkolem této práce bylo přímo implementovat virtuální stroj pro provádění programů napsaných v programovacím jazyku *Java*, což se mi nakonec úspěšně povedlo. Mému emulátoru chybí do plnohodnotného virtuálního stroje implementovat *classloader* a modul pro vykonávání nativních metod. Tyto dva moduly ale ovšem nebyly cílem a pro účely detekce vše funguje jak má.

Classloader je nahrazen použitím falešných tříd a spušťení nativních metod dokonce není ani žádoucí. Proto volání nějaké nativní metody vždy skončí chybou a ukončením prováděné emulace. Pokud bych povolil a implementoval volání nativních metod, mohl by se testovaný program dostat přímo k prostředkům operačního systému, na kterém můj emulátor běží. Tímto by se prakticky plnohodnotně program spustil a provedl škodlivou činnost. Z tohoto důvodu je volání nativních metod zakázáno a je do jisté míry nahrazeno falešnými metodami, které v případě volání nativních metod nezasahují přímo do operačního systému, na kterém emulátor běží, ale pouze správným způsobem upraví kontext volající metody tak, aby bylo zajištěno správné pokračování provádění programu.

Emulátor je schopný jak spustit úplný program napsaný v programovacím jazyce *Java*, tak i provést analýzu konkrétního *.class* souboru z předem načteného balíku. Toto především slouží pro zobrazení informací o této analyzované třídě, které následně budou použity pro vytvoření její instance ve třídě *FakeClass*. Momentálně se tyto informace do zdrojového souboru zadávají ručně, což není příliš optimální řešení. Do budoucna se plánuje vytvoření nástrojů pro automatickou tvorbu falešných tříd, metod a datových položek, které bude využívat informace z analýzy tímto emulátorem.

Vlastní detekce naopak provede testovaný program tak, jako kdyby byl spouštěn na realném virtuálním stroji. Není tomu ovšem tak a spuštěný program se nikdy nedostane ven z emulátoru. Po provedení emulace je nakonec na standardní výstup vypsána informace o tom, jak testování dopadlo.

Během celého vývoje jsem prováděl neustálé testování na datech, která jsem si sám vytvořil a jejichž část je obsažena i na přiloženém médiu. Nejhorší částí bylo programování vlastního parsování *.class* souboru, které mi vůbec nešlo a na kterém jsem strávil velkou část celého vývoje. Na druhou stranu je nutné dodat, že implementace *heap*-u a automatického správce paměti spolu s alokací objektů na *heap*-u, byla ta část práce, kterou jsem si naopak nejvíce užil a do které jsem se nejvíce ponořil.

Celou dobu co jsem vyvíjel tento emulátor jsem se pevně držel specifikace virtuálního stroje[12] i přesného popisu veškerých instrukcí[4], ale nezabýval jsem se absolutně nějakými většími optimalizacemi. Například implementacemi některých částí emulátoru do assembleru. Tyto optimalizace jsou implementovány ve virtuálních strojích, které se obecně

používají na různorodých platformách a které jsou v tomto textu zmiňovány. Pro připomenutí to jsou virtuální stroje *HotSpot*, *JamVM* a *IcedTea*. Zdrojové kódy těchto virtuálních strojů budou sloužit pro inspiraci při provádění některých použitelných optimalizací.

Aktuálně je s emulátorem, který jsem vytvořil, počítáno jako s další částí složitějšího softwaru pro detekci a analýzu malware. Z tohoto důvodu vývoj emulátoru bude nadále výrazně pokračovat směrem k začlenění tohoto emulátoru do již zmíněného sofistikovanějšího programu. Je v plánu také promyslet a prozkoumat nová vylepšení, která se objevovala pravidelně během celého vývoje a která by se teoreticky mohla dále využít.

Literatura

[1] *IEEE Standard for Floating-Point Arithmetic*. 3 Park Avenue, New York, NY 10016-5997, USA: The Institute of Electrical and Electronics Engineers, Inc., Srpen 2008, ISBN 978-0-7381-5752-8, 1–58 s., doi:10.1109/ieeestd.2008.4610935.
URL http://dx.doi.org/10.1109/ieeestd.2008.4610935

[2] Chapter 2. The Structure of the Java Virtual Machine. [online], 2013.
URL http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html

[3] Chapter 4. The class File Format. [online], 2013.
URL http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html

[4] Chapter 6. The Java Virtual Machine Instruction Set. [online], 2013.
URL http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

[5] Domovská stránka JamVM. [online], 2013.
URL http://jamvm.sourceforge.net/

[6] The HotSpot Group. [online], 2013.
URL http://openjdk.java.net/groups/hotspot/

[7] Interpretované programovací jazyky - Wikipedia. [online], 2013.
URL http://cs.wikipedia.org/wiki/Interpretovaný_jazyk

[8] JamVM na wikipedii. [online], 2013.
URL http://en.wikipedia.org/wiki/JamVM

[9] The Java HotSpot Performance Engine Architecture. [online], 2013.
URL http://www.oracle.com/technetwork/java/whitepaper-135217.html#1

[10] javac - Java programming language compiler. [online], 2013.
URL
http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javac.html

[11] The Java™Tutorials - Classes. [online], 2013.
URL http://docs.oracle.com/javase/tutorial/java/javaOO/classes.html

[12] The Java™Virtual Machine Specification. [online], 2013.
URL http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[13] The Oracle Corporation website. [online], 2013.
URL http://www.oracle.com/index.html

[14] SourceForge stránky virtuálního stroje JamVM. [online], 2013.
URL http://sourceforge.net/projects/jamvm/

[15] Understanding the Default Manifest. [online], 2013.
URL http://docs.oracle.com/javase/tutorial/deployment/jar/defman.html

[16] Aycock, J.: *Computer viruses and malware*. Advances in information security, Springer, 2006, ISBN 978-0-387-30236-2.
URL http://books.google.cz/books?id=DqNQAAAAMAAJ

[17] Barclay, K. A.; Savage, W.: *Groovy programming: an introduction for Java developers*. New York: Morgan Kaufmann Publishers, c2007, ISBN 01-237-2507-0.

[18] Edelson, J.; Liu, H.; [foreword by Martin Odersky]: *JRuby cookbook*. O'Reilly Media Inc, první vydání, 2009, ISBN 978-0-596-51980-3.

[19] Halloway, S.: *Programming Clojure*. Pragmatic Bookshelf, druhé vydání, 2009, ISBN 19-343-5686-7.

[20] sponsor Microprocessor Standards Committee of the IEEE Computer Society: *IEEE standard for floating-point arithmetic*. Institute of Electrical and Electronics Engineers, 2008, ISBN 978-073-8157-535.

[21] Jones, R.; Lins, R.: *Garbage collection: algorithms for automatic dynamic memory management*. New York: Wiley, c1996, ISBN 04-719-4148-4.

[22] Juneau, J.; Baker, J.; Ng, V.; aj.: *The definitive guide to Jython: Python for the Java platform*. New York: Distributed by Springer-Verlag, první vydání, 2009, ISBN 14-302-2527-0.

[23] Meyer, J.; Downing, T.: *Java virtual machine: an introduction for Java developers*. O'Reilly, druhé vydání, 1997, ISBN 15-659-2194-1.

[24] Pollak, D.; [foreword by Martin Odersky]: *Beginning Scala*. Apress, new ed. vydání, 2009, ISBN 978-1-4302-1989-7.

[25] Subramaniam, V.: *Programming Scala*. Pragmatic Bookshelf, 2008, ISBN 978-1-9343-5631-9.

[26] Venners, B.: *Inside the Java virtual machine*. New York: McGraw-Hill, druhé vydání, c1999, ISBN 00-713-5093-4.

Příloha A

Seznam tříd a metod podílejících se na detekci

V tabulce A.1 jsou uvedeny třídy obsahující metody, které se podílejí na detekci škodlivého chování. Spolu s jejich názvy jsou uvedeny i hodnoty, kterými jsou jednotlivé metody reprezentovány v předem připravených sekvencích a kterými vytvářejí testované sekvence průběhu programu.

Sekvence, proti kterým je testováno vlastní chování, jsou uvedeny ve zdrojových souborech mého emulátoru, přesněji ve zdrojovém souboru *sequences.cpp* třídy *Sequences* a to v poli *m_sequences*. Detailní použití tohoto pole je uvedeno v kapitole 5.11.

Tabulka A.1: Třídy a jejich metody podílející se na tvoření maligních sekvencí.

Název třídy	název metody	kód metody
java/lang/StringBuilder	StringBuilder(String)	20
	append(StringBuffer)	21
	append(String)	22
	append(char [])	23
	append(long)	26
	append(char)	27
	append(float)	28
	append(double)	29
java/io/File	File(String)	50
	createTempFile(String, FileAttribute)	51
	createNewFile()	52
java/io/FileWriter	FileWriter(String)	70
java/io/File	File(String)	90
java/io/BufferedWriter	BufferedWriter(FileWriter)	110
	write(String)	130

Tento seznam metod, které mohou být potenciálně součástí maligního chování, je velice malým zlomkem z celkového počtu tříd a metod, které se ve výsledku budou logovat a následně porovnávat. Pro ukázku a demonstraci detekce zde ale naprosto dostačuje i tato malá podmnožina a čtenář si může udělat detailní obrázek o principu celé detekce.

Zbytek aktuálně detekovaných tříd a metod je pro znázornění uveden ve zdrojových kódech v souboru *function_values.txt*.

Příloha B

Manuál

V této části je popsáno, jakým způsobem spustit mnou vytvořený emulátor pro analýzu konkrétní třídy a jak pro spuštění detekce na nějaký konkrétní program napsaný v programovacím jazyce *Java*. Nejlepší jsou ukázky na příkladech, takže si na nich projdeme oba případy použití.

B.1 Spuštění analýzy konkrétní třídy

Prvním příkladem bude analýza třídy *java/lang/String*, která se nachází v balíčku standardních tříd *Javy*. Balíček *rt.jar*, který je uložen v kořenovém adresáři přiloženého média, si rozbalíme například do složky *C:\rt*. Po této operaci bude obsahovat kompletní strukturu balíčku ze zmíněného archivu.

Pro spuštění analýzy konkrétní třídy slouží obecný příkaz:

```
.\classparser.exe -info < jmeno tridy > < cesta k balicku >
```

Pro výše zmíněný příklad to bude tedy následovně :

```
.\classparser.exe -info java/lang/String C:\rt
```

Po provedení jsou na standardní výstup vypsány veškeré důležité informace o třídě, která byla analyzována.

B.2 Spuštění emulace validního programu

Mnohem zajímavější variantou je spouštění programu v tomto emulátoru. To se děje přes parametr *-run*, který je následován cestou s rozbaleným *.jar* archivem.

Mějme tedy rozbaleny balíček *program.jar* do složky *C:\program*.

```
.\classparser.exe -run C:\program
```

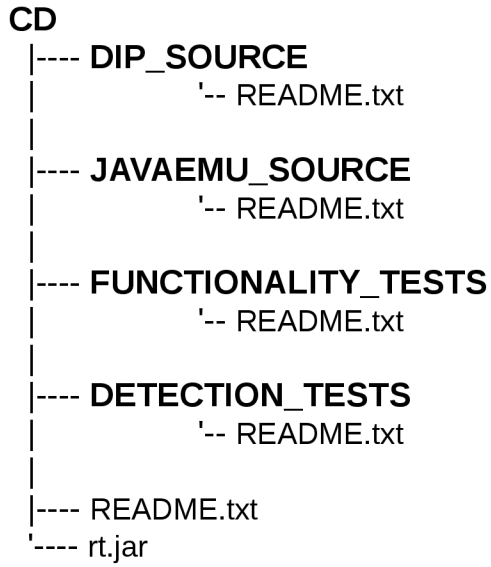
Pokud je v hlavičkovém souboru *macros.h* definováno makro *PRINT_INSTRUCTIONS*, uvidíme na standardním výstupu názvy všech instrukcí jak jdou v rámci provádění konkrétního programu za sebou. Jelikož je výpis do terminálu časově náročný, může se celkové provádění emulace značně časově protáhnout. Na vlastní funkčnost to ale nemá absolutně žádný vliv.

V případě, že makro *PRINT_INSTRUCTIONS* definováno nebude. Zobrazí se pouze hlášení o tom, zdali se jedná nebo nejedná o škodlivý program.

Příloha C

Obsah DVD

Na přiloženém DVD jsou v adresářové struktuře, uvedené níže, uloženy veškeré soubory potřebné pro účely této práce. V každé složce datového média je uložen soubor *README.txt*, který podrobně vysvětluje obsah a použití konkrétní složky a souborů v ní uložených.



Obrázek C.1: Obsah přiloženého datového média.

Adresář *DIP_SOURCE* obsahuje veškeré potřebné soubory pro překlad vlastního textu diplomové práce. Ve složce *JAVAEMU_SOURCE* je uložen kompletní projekt v programu *Microsoft Visual Studio* reprezentující výsledný emulátor. Zdrojové kódy i přeložené aplikace sloužící pro otestování veškeré potřebné funkcionality mého emulátoru jsou uloženy v adresáři *FUNCTIONALITY_TESTS*. Poslední složkou je složka *DETECTION_TESTS* obsahující zdrojové kódy aplikací sloužících pro otestování vlastní detekce škodlivého chování.

Soubor *rt.jar* v kořenovém adresáři datového média je balíček se standardními funkcemi programovacího jazyka *Java* zkopírované z instalační složky běhového prostředí *Java Runtime Environment*.