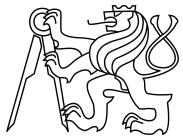
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Obfuskační nástroj pro Python
Student:	Bc. Martin Holoubek
Vedoucí:	Ing. Tomáš Zahradnický, Ph.D.
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

Seznamte se s taxonomií obfuskačních transformací [1]. Navrhněte nástroj, načítající kód v jazyce Python do vnitřní reprezentace (Internal Representation, IR). Na základě IR navrhněte a implementujte obfuskační transformace jako moduly se společným rozhraním. Jako minimální sadu transformací implementujte šifrování řetězců a slučování funkcí, případně jiné obfuskační transformace po dohodě s vedoucím. Po provedení transformací nástroj zapíše zpět výstup jako kód v jazyce Python, který musí zůstat spustitelný. Diskutujte potenci a odolnost transformací, dále proveďte srovnání s nástroji s obdobnými schopnostmi pro Python.

Seznam odborné literatury

1. Collberg C. et al. A Taxonomy of Obfuscation Transformations. Technical Report #148. University of Auckland. New Zealand. https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 8. září 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Obfuskační nástroj pro Python

Bc. Martin Holoubek

Katedra počítačových systémů
Vedoucí práce: Ing. Tomáš Zahradnický, Ph.D.

3. ledna 2018

Poděkování

Chtěl bych poděkovat vedoucímu své práce za čas a také ochotu kdykoliv poradit. Díky také všem kamarádům a spolužákům, bez nichž by ty roky byly o ničem. :) Poděkování patří i celé společnosti za to, že mi dovolila studovat a naivně čekala na výsledek. Hlavní dík ale patří rodině za to, že mě podporovala, snášela mé stížnosti a zachovala si zdravý odstup. Díky **mami**, **tati**, **brácho**!

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 3. ledna 2018 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Holoubek, Martin. *Obfuskační nástroj pro Python.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce se zabývá problematikou obfuskace pro programovací jazyk Python 3. Postupně jsou analyzovány existující přístupy a řešení. Dále je práce zaměřena na návrh nového modulárního nástroje spolu s obfuskačními moduly. Zvolené řešení je popsáno, implementováno a jsou shrnuty vlastnosti vytvořeného nástroje.

Klíčová slova python, obfuskátor, nástroj, převod, reverzní inženýrství, statická analýza

Abstract

This thesis deals with the issue of obfuscation for the programming language Python 3. Existing solutions and approaches are further analyzed and discussed. The thesis focuses on the design of a new modular tool with separate obfuscation modules. In following parts the implementation itself is described and overall properties are summarized.

Keywords python, obfuscator, tool, translation, reverse engineering, static analysis

Obsah

Seznam obrázků

Úvod

Skriptovací jazyk Python je jedním z nejpoužívanějších programovacích jazyků. Je využíván v různých oblastech — od bezpečnosti, přes umělou inteligenci, až po automatizaci a testování. Otevřená licence ho předurčuje k širokému uplatnění jak v korporátním prostředí, tak pro osobní použití.

Mezi jeho přednosti patří snadná přenositelnost a čitelnost zdrojových kódů. Tyto vlastnosti, i když často žádoucí, mohou být v některých aplikacích přítěží. Způsob distribuce a vlastnosti jazyka mohou případnému reverznímu inženýrovi usnadnit zisk informací o funkcionalitě programu, který byl v Pythonu vytvořen. To je nepřijatelné zejména ve firemním prostředí, kde jsou algoritmy a principy často proprietární a utajené.

Tvůrci mohou k ochraně svých programů využít speciální techniky určené k zakrývání informací a ztížení analýzy. Obecně se tyto techniky nazývají obfuskace a nástroje, které je provádějí obfuskátory. V této práci se budeme věnovat tématu obfuskace pro programovací jazyk Python 3.

V textu se nejprve zaměříme na analýzu stávajících nástrojů a řešení, které umožňují obfuskaci. Zhodnotíme možné přístupy a navrhneme modulární nástroj. Dále vytvoříme obfuskační techniky a algoritmy, které umožní modifikovat a zakrýt informace obsažené ve zdrojových kódech. Tyto obfuskace zaměříme specificky na jazyk Python 3 a jeho unikátní vlastnosti.

V implementační části popíšeme tvorbu samotného nástroje včetně modulů implementujících námi navržené techniky obfuskace. Kvalitu těchto technik se pokusíme zhodnotit pomocí veličin, které v textu zavedeme. Řešení nakonec srovnáme s existujícími nástroji.

Kapitola 1

Analýza

1.1 Programovací jazyk Python

Python je dynamicky typovaný programovací jazyk vynikající svou jednoduchou syntaxí. Jeho tvar se odklonil od syntaxe jazyků rodiny C, kdy byl při návrhu odstraněn hierarchický systém složených závorek. Jedná se však pouze o syntaktický rozdíl a Python přímo podporuje interoperabilitu [1] s ostatními jazyky a prostředími jako je C, C++, java, či .NET/Mono.

Kromě typové kontroly podporuje Python různá programovací paradigmata od objektově orientovaného, imperativního, procedurálního, či funkcionálního. Python je od počátku vyvíjen jako svobodný open-source projekt, který je zdarma distribuován pod permisivní licencí PSF [2]. V této otevřené licenci se autoři jazyka vzdávají nároků na jakoukoliv odměnu, naopak očekávají, že s výsledným produktem nebudou spojováni. Python je tedy výhodný jak pro korporátní, tak i pro osobní uživatele.

Obrázek 1.1: Logo programovacího jazyka Python [1]

Mezi další výhody je možné považovat snadnou dostupnost instalačních balíčků, které existují pro téměř všechny platformy od Windows, MacOS až po Linux a BSD. Postupem času se okolo Pythonu vytvořila početná komunita programátorů a autorů vytvářející zdarma dostupné moduly. Tyto balíčky

[1] `http://ironpython.net/` — implementace pro platformu .NET
[2] `https://docs.python.org/3/license.html` — Python licence

jsou instalovány pomocí nástroje *pip*, který je standardní součástí instalace [3]. Práce programátora je tak z veliké části zjednodušena. Tento přístup umožňuje tvůrcům soustředit se pouze na podstatné části řešeného problému.

Výše zmíněné vlastnosti jazyk předurčují k širokému uplatnění od krátkých skriptů, rozsáhlejších programů, až po webové aplikace a použití v umělé inteligenci. Během posledních let se Python pravidelně umisťuje na vyšších příčkách statistik oblíbenosti programovacích jazyků [2].

1.1.1 Historie Pythonu

Python v roce 1991 vytvořil Guido van Rossum. Během let byl jazyk postupně doplněn o další syntaktické a funkční prvky [3]. V poslední době například přibyla podpora pro asynchronní programování. V roce 2008 zásahy do jazyka vedly k rozdělení vývoje do dvou větví. Vznikl tak nový Python 3.0, který je zpětně nekompatibilní s původní verzí 2. Přes tyto rozdíly existují knihovny, které umožňují psát a distribuovat programy pro obě varianty jazyka.

Protože je jazyk distribuován pod otevřenou licencí, existují mimo oficiální implementaci *CPython* i neoficiální pro různé platformy: *PyPy*, *IronPython*, *Jython* a *Boost.Python*.

1.1.2 Vlastnosti Pythonu

Možnosti distribuce a způsoby využití skriptů skriptů jsou velmi široké. Jednou z nich je přímý přenos skriptů do cílového systému a spuštění v interpreteru. Je také možné předkompilovat skripty do bytekódu B.1 a následně je spustit přímo v interpreteru. Existují navíc různé nástroje pro interoperabilitu, je tak možné přímo vestavět zdrojové kódy Pythonu do programů vytvořených v jiném jazyce. Například nástroj IronPython umožňuje provázání s platformou .NET. Python také podporuje interaktivní režim — je možné jej spustit v konzoli, kde jsou jednotlivé příkazy, či bloky kódu ihned vyhodnocovány a uživatel dostává okamžitou odezvu.

Python je silně dynamický jazyk. Dynamičnost je tak vysoká, že umožňuje přímou sebe-modifikaci programů. Skript může za běhu zkompilovat externí kód a ten spustit. Výsledný program tak může být silně polymorfní.

Až do roku 2015 byla značnou nevýhodou Pythonu nízká podpora paralelismu [4]. Interpreter je inherentně sekvenční automat, pro paralelizaci problémů tak bylo nutné využívat multiprocessing, případně externí knihovny. Verze 3.5 tento problém obchází přidáním podpory pro výrazy umožňující asynchronní běh. Části programu mohou být nezávisle na sobě spouštěny jako úlohy a automaticky synchronizovány tak, aby se běh nenarušil. Klíčová slova, která toho umožňují jsou `async` a `await`.

Výše zmíněné schopnosti a přednosti předurčují tento jazyk k intenzivnímu použití. Python je oblíbeným jazykem používaným také v oblasti bezpečnosti.

[3] `https://pypi.python.org/pypi` — dostupné balíčky pro Python

Tyto nástroje je nutné distribuovat k cílovým uživatelům. Obvykle je nutné minimalizovat riziko úniku interních informací, postupů i algoritmů ze zdrojových kódů.

Kromě přímé distribuce skriptů je tak možné předkompilovat do binární formy, tzv. bytekódu, který je interpretovatelný stejně jako původní program. Kompilátor však provádí překlad způsobem, který je reverzibilní. Původní kód je tak možné téměř dokonale obnovit včetně komentářů. Programátorům a společnostem tak zbývá spolehnout se na nástroje, které provádějí změny zdrojových kódů a činí je nečitelnými. Takový nástroj se nazývá obfuskátor.

1.2 Obfuskace

Obfuskace je postup používaný při vývoji softwaru, který spočívá v zakrývání funkcionality a vnitřní struktury programu [5, s. 1]. Pod tímto termínem se skrývá celá řada principů a technik. Výsledkem je program, který si zachovává původní funkcionalitu, ale případnému útočníkovi ztíží pochopení vztahů a myšlenek použitých ve zdrojovém kódu. Útočníkem může být například reverzní inženýr konkurenční firmy, či jiné instituce, která má zájem na získání proprietárních informací.

Zakrývat lze kromě zdrojových kódů také zkompilované binární soubory, či jiné pomocné materiály. Modifikace mohou být prováděny ručně, kdy tvůrce programu použije vlastní metody a postupy, nebo automaticky za použití komerčních nástrojů.

Příkladem obfuskační techniky může být *odstranění komentářů*, *přejmenování identifikátorů*, nebo *systematické přetěžování funkcí* [5, s. 2]. Techniky se různí podle vývojářů daného nástroje a také podle cílového jazyka, či frameworku. Různé programovací jazyky poskytují vývojářům unikátní možnosti, které lze k obfuskaci použít.

1.2.1 Vývoj

Vývoj obfuskátoru typicky začíná určením cílového programovacího jazyka, případně platformy, na kterou bude nástroj cílit. Podle vybraného jazyka se dále odvíjí, které obfuskace bude možné navrhnout a implementovat. Základním otázkou je, zda daný jazyk je statický, či dynamický. Tedy zda poskytuje podporu pro sebemodifikaci. Takový program může měnit svou strukturu za běhu. Pak je na návrhu a implementaci, jaké modifikace vytvořit, aby se nezměnilo chování cílového programu a přitom byla obfuskace účinná.

Ve spojitosti s dynamičností jazyka se často uvádí pojem typově dynamický, či typově statický jazyk. Typově dynamické programovací jazyky nekontrolují typy proměnných během procesu kompilace, ale až za běhu programu. Pro autory obfuskačního nástroje to znamená, že typicky nelze snadno určit typy proměnných a výrazů, což může komplikovat některé modifikace.

Částečně lze tento problém obejít typovou dedukcí, která se podobá interpretaci kódu s tím rozdílem, že výsledná hodnota výrazu je nedůležitá, ale zajímá nás datový typ. Obecně však takový nástroj nelze vytvořit, protože části programu jsou typicky podmíněné uživatelským vstupem, na němž závisí další běh programu a tedy i typy proměnných. Snadno si lze domyslet, že u dynamických jazyků, které umožňují změnu programovacího kódu je tato technika použitelná pouze omezeně.

1.2.2 Taxonomie obfuskace

Techniky zakrývání informace ve zdrojových kódech lze posuzovat z několika úhlů pohledu pohledu [5, s. 2].
Jsou jimi:

- Intelektuální ochrana (Intellectual obfuscation)
  - Legální ochrana
  - Technická ochrana — obfuskace, šifrování, spuštění na serveru, trusted code
- Kvalita ochrany
  - Potence (Potency)
  - Resilience (Resiliency)
  - Cena (Cost)
- Cíl transformace (Transformation target)
  - Rozložení zdrojových kódů (Layout obfuscation)
  - Obfuskace dat (Data obfuscation)
  - Kontrolní obfuskace (Control obfuscation)
  - Preventivní obfuskace (Preventive obfuscation)

1.2.3 Vlastnosti obfuskace

Požadavky na obfuskátor se liší podle cílového jazyka i požadavků uživatelů. Společným cílem autorů úprav je jejich jednosměrnost, která označuje obtížnost zpětného obnovení kódu. Typickým příkladem jednosměrné (one-way) obfuskace je *odstranění komentářů* a *přejmenování identifikátorů*. Naopak vložené konstantní podmínky, jejichž logická splnitelnost se při běhu nemění, lze při pečlivé analýze odstranit a program restaurovat do původního stavu.

Otázkou je, jakým způsobem lze srovnávat kvalitu a další parametry obfuskačních technik. V praxi se prosadily tyto tři metriky [5, s. 7]:

- resilience — Resilience vyjadřuje odolnost provedených úprav vůči automatickému převodu do původní formy. Platí, že jednosměrné metody mají velmi vysokou resilienci, protože úplně chybí informace potřebná k obrácení efektu.

- potence — Potence je abstraktní míra zmatení pomyslného nepřítele. Tedy vyjadřuje námahu, kterou musí vynaložit čtenář upraveného kódu pro pochopení principů a myšlenek.

- cena — Cena popisuje rychlost běhu výsledného programu ve srovnání s původní verzí.

V dalším textu budeme hodnotit kvalitu obfuskací pomocí těchto tří metrik. Protože jde o částečně subjektivní srovnávání, vyjádříme jej ve třech úrovních — nízká, střední a vysoká.

Veličiny uvedeme na příkladu:

Mezi obfuskační techniky lze považovat i kompilaci zdrojových kódů jazyka C do binárního souboru. Obecně se tak děje z technologické nutnosti.

Spustitelný program není možné převést zpět do původního zdrojového kódu, protože velká část informace o proměnných, struktuře a vlastnostech programu je ztracena. Operace je tedy téměř jednosměrná a resilience tudíž vysoká. Kdokoliv, kdo zkoušel číst, či programovat přímo v binárním kódu, bude souhlasit, že potence úpravy je vysoká. Cenu operace nelze přesně určit, protože programový kód v jazyce C není spustitelný, chybí tedy reference. Obecně se však kompilace používá i z důvodu zrychlení programu, cena by v takovém případě byla nulová, či záporná.

Zkušený reverzní inženýr dokáže při dostatku času získat požadované znalosti z jakkoliv upraveného programu. Cílem útočníka navíc často bývá zjištění potřebných informací co nejrychlejším způsobem, ať už jde o zakompilovaná hesla, algoritmy, či myšlenky. Pokusíme se tedy navrhnout metody, které budou mít vysokou potenci i resilienci a nízkou cenu.

1.2.4 Lokalita obfuskace

Faktorem ovlivňujícím výslednou kvalitu obfuskace je také kontext ve kterém je algoritmus spuštěn. Operace můžeme aplikovat na několika úrovních od jednotlivých řádků kódu, funkcí, tříd, či celých modulů. Spuštěním obfuskací na vyšší úrovni lze dosáhnout vyšší potence i resilience, protože se celá úprava stává komplexnější. Od úrovně, na kterou se obfuskace zaměřuje, se také odvíjí složitost návrhu. Například přejmenování argumentů lokálních funkcí je poměrně snadnější, než stejný úkol provedený na úrovni modulu. Schopnosti a úroveň jednotlivých modifikací musíme proto pečlivě zvážit a zhodnotit jejich smysl a kvalitu.

1.2.5 Technologie obfuskace

Na vstupu obfuskačního nástroje je zdrojový kód, případně struktura, reprezentující spustitelný kód. Výstupem je program v téže formě umožňující jeho spuštění. Pod touto formou si můžeme představit například upravený zdrojový kód, bytekód, spustitelný soubor. Některé nástroje umí převést program do proprietárního formátu podporujícího šifrování, pro jehož spuštění je pak typicky nutné používat online ověřovací metodu [4].

Technicky je každý obfuskátor překladačem, který převádí program zapsaný vstupním jazykem do jazyka výstupního. Pokud navržený nástroj nevyžaduje brát v úvahu strukturu, lze jej realizovat jako textové úpravy aplikované konečným automatem. V mnoha jazycích lze tak například odstranit komentáře, aniž abychom znali přesnou strukturu a syntaxi jazyka. Pro složitější úpravy je výhodné použít převod zdrojového kódu do interní formy, tzv. abstraktního syntaktického stromu (AST), která zachovává hierarchii a další vlastnosti [6].

Tento tvar je využíván v kompilátorech při převodu jazyka do výstupního formátu. Například kompilací zdrojového programu v jazyce C vzniká binární spustitelný soubor. Kompilátory se tedy v mnohém podobají obfuskátorům, protože využívají podobné principy a postupy. Přesto je základním rozdílem mezi těmito nástroji motivace jejich tvůrců. Obfuskátory slouží primárně k zakrývání funkcionality, kdežto kompilátory k tvorbě spustitelných souborů, případně zrychlení jejich běhu.

Abychom mohli vytvářet složitější metody pro náš nástroj, musíme převést vstupní program do tvaru abstraktního syntaktického stromu. Tento převod se nazývá syntaktická analýza a provádí ji nástroj zvaný *parser*. Vstupní řetězec zapsaný pomocí příslušné gramatiky je nejprve ve fázi tokenizace zbaven netisknutelných znaků a je rozdělen na klíčová slova — *tokeny*. Ty jsou postupně předkládány parseru, který kontroluje zda jejich struktura a posloupnost odpovídá pravidlům gramatiky a vytváří potřebný syntaktický strom [7].

Výstupem syntaktické analýzy je strom, který obsahuje veškeré informace nutné k převodu. Obfuskace tedy převede vstupní zdrojové kódy do příslušných syntaktických stromů, aplikuje požadované úpravy a výstup vypíše zpět v původní formě. Aby bylo možné aplikovat jednotlivé úpravy, je strom procházen definovaným způsobem. Tři základní typy průchodu jsou preorder, inorder a postorder. Tyto se liší v pořadí prováděných operací na uzlech stromu.

Hlavním předpokladem ke tvorbě parseru je gramatika, která vyčerpávajícím způsobem popisuje a definuje daný jazyk, respektive všechny věty, které do něj patří. Pro programovací jazyk Python je gramatika volně k dispozici ve zdrojových souborech, v opačném případě by ji bylo možné vytvořit zpětně podle existujících programů.

[4] http://bits.citrusbyte.com/protecting-a-python-codebase/ — Kompilace zdrojových kódů v Pythonu

```
int E,L,O,R,G[42][m],h[2][42][m],g[3][8],c
[42][42][2],f[42]; char d[42]; void v( int
b,int a,int j){ printf("\33[%d;%df\33[4%d"
"m  ",a,b,j); } void u(){ int T,e; n(42)o(
e,m)if(h[0][T][e]-h[1][T][e]){ v(e+4+e,T+2
,h[0][T][e]+1?h[0][T][e]:0); h[1][T][e]=h[
0][T][e]; } fflush(stdout); } void q(int l
                ,int k,int p){
                int T,e,a; L=0
                ; O=1; while(O
                ){ n(4&&L){ e=
                k+c[l] [T][0];
                h[0][L-1+c[l][
                T][1]][p?20-e:
```

Listing 1.1: Zkrácená ukázka obfuskace aplikované na program v jazyce C [8]

Struktura abstraktních stromů pro Python je poměrně jednoduchá. Je to hierarchický strom, ve kterém mohou uzly obsahovat atributy a těla složená z jiných uzlů. Zdrojový kód je vždy reprezentován jako *Modul*, který obsahuje tělo složené z jednotlivých výrazů, které zhruba odpovídají jednotlivým řádkům kódu. Výrazem může být například definice funkce, třídy, nebo podmíněný výraz. Vhodným zdrojem je oficiální dokumentace, která popisuje přípustnou strukturu spolu s typy uzlů a jejich atributy [9].

1.3 Vektory pro obfuskaci

Některé obfuskační techniky jsou obecně použitelné pro různé programovací jazyky, protože ty mezi sebou sdílejí společné vlastnosti. Téměř všechny jazyky například používají identifikátory, můžeme se tak zaměřit na jejich přejmenování. Společnou vlastností je také přítomnost komentářů, jejichž odstranění může být účinnou obfuskací. Jazyky, využívající složené závorky k oddělení kontextů, zase můžeme modifikovat odstraněním netisknutelných znaků bez dopadu na funkčnost.

Aby byly obfuskace účinné a obtížněji odhalitelné, měly by se zaměřit na unikátní vlastnosti daného jazyka. Vybrané syntaktické a sémantické vlastnosti, které jsou charakteristické pro Python:

- docstringy — Dokumentační tvar klasických komentářů specifická pro Python.
- dekorátory — Syntaxe umožňující vložit mezivrstvu mezi volajícího a volanou funkci.
- generátory — Funkce, které podporují tzv. lazy-evaluation, tedy vrací více hodnot postupně až ve chvíli, kdy je to nutné.

- záporné tvary cyklů — V Pythonu mohou mít cykly druhou větev, která se vykoná ve chvíli, kdy přestane platit podmínka — klíčové slovo `break` se na ni nevztahuje. 1.2
- lambda výrazy — Krátké funkce používané často k filtrování.
- lokální funkce a třídy — Python umožňuje definovat objekty na lokální úrovni.
- celá čísla — Celá čísla mají v Pythonu neomezený rozsah.
- comprehension — Speciální syntaxe pro vytvoření seznam modifikací jiného.
- syntaxe — Python využívá odsazení k určení kontextů.
- kódování — Python přímo podporuje *Unicode*, tedy i národní identifikátory a obsah.
- *async* a *await* — Python podporuje asynchronní volání.

```
while False:
  pass
else:
  magic_happen_here()
```

Listing 1.2: Ukázka cyklu s oběma větvemi

Tyto a další vlastnosti specifické pro Python bude výhodné využít při návrhu cílených obfuskačních technik.

1.3.1 Typické obfuskační metody

V této části uvedeme některé existující obfuskační metody a odhadneme jejich použitelnost v prostředí jazyka Python [5, 10].

Obfuskační nástroje existují pro širokou škálu programovacích jazyků i platforem, některé metody jsou obecně použitelné, zatímco jiné se zaměřují výhradně na dané vlastnosti cílového jazyka. Uvedeme nyní vybrané typické metody obfuskace. Každou z nich popíšeme a uvedeme možnosti aplikace pro jazyk Python 3:

- Odstranění komentářů
  - Obfuskace užitečná zejména pro jazyky, v nichž psané programy jsou distribuovány ve formě zdrojového kódu.
  - Tato metoda obfuskace je pro Python obecně použitelná.
- Přejmenování proměnných

  - Názvy proměnných mohou přinášet dodatečné informace o funkcionalitě programu a proto je často žádoucí provést přejmenování.
  - Python proměnné používá a podléhá tedy této modifikaci.

- Změna logického toku
  - Modifikace podmínek, skoků, vkládání neužitečného kódů a další úpravy vedoucí k zakrytí logického průchodu.
  - Pro Python je metoda obecně použitelná s drobnými omezeními způsobenými vysokou dynamičností jazyka, kdy není vždy možné staticky analyzovat zdrojový kód.

- Paralelizace zdrojového kódu
  - Specializací obfuskace logického toku může být paralelizace kódu. Systém vláken, procesů a synchronizačních primitiv je v takovém případě využit ke ztížení analýzy.
  - Python nativně podporuje práci s procesy a omezeně i s vlákny. Obfuskace je tady použitelná.

- Změna pořadí
  - Metoda podobná změně logického toku. Typicky se může jednat o změnu pořadí definic, deklarací a dalších výrazů tak, aby se funkcionalita výsledného programu nezměnila.
  - Metoda je pro Python opět použitelná a platí omezená způsobená dynamickou povahou jazyka.

- Agregace
  - Metoda provádějící změny více výrazů současně. Například *inlining funkcí* apod.
  - Metoda je použitelná se stejnými omezeními.

- Změna přístupu k datům
  - Univerzální obfuskace založená na změně datových struktur a přístupu k nim.
  - Python jako příklad imperativního jazyka používá datové struktury, na které je tato metody aplikovatelná.

- Obfuskace řetězců
  - Specializací datové obfuskace je úprava řetězců. Na ty je možné aplikovat metody k slučování, rozdělování, šifrování apod.

  - Python používá řetězce jako konstanty ve zdrojovém kódu, které úpravám podléhají.

- Převod na stavový automat
  - Obfuskace založené na převodu programu, nebo jeho části na stavový automat. Ten vykovává výrazy ve správném pořadí. Jedná se kombinaci agregace, změny pořadí a logického toku programu.
  - Tyto metoda je do jisté míra aplikovatelná i pro Python. Situace je opět komplikována dynamičností jazyka, kdy není možné vždy staticky analyzovat kód a vytvořit příslušný automat.

- Převod do tabulkové formy
  - Obfuskace založená na přidání indirekce. Spočívá ve vytvoření tabulky funkcí, které jsou na původních místech volány pomocí příslušného indexu.
  - Tato metoda je pro Python opět částečně aplikovatelná s omezeními plynoucími z dynamičnosti.

- Přetížení indukcí
  - Metoda spočívající v přejmenování jmenných prostorů, tříd, funkcí i proměnných použitím co nejméně názvů pomocí indukce.
  - Podmínkou provedení indukce je předchozí statická analýza, která musí správně zachytit jména tak, aby mohlo dojít k přejmenování. Tento požadavek komplikuje použití v případě Pythonu.

1.4 Existující řešení a nástroje

V současné době je Python ve verzi 3.6.2. [5] Nástroj, který budeme vytvářet zaměříme zejména na podporu této verze. Přesto, že jednotlivé vydání Pythonu se od sebe liší, jsou syntaktické i sémantické rozdíly poměrně malé. Nové verze přidávají vlastnosti a zachovávají zpětnou kompatibilitu, je tedy velmi pravděpodobné, že námi navržený nástroj bude funkční i pro další verze. Případně bude rozšíření podpory o starší i novější varianty poměrně snadné.

Nyní srovnáme existující nástroje sloužící k obfuskaci zdrojových kódů v Pythonu:

- py_compile — Modul pro Python 2 i 3 určený pro kompilaci zdrojového kódu do bytekódu. Primárně je určen ke zrychlení běhu programu a usnadnění distribuce. Lze jej využít i jako obfuskační nástroj. Existují

[5] `https://docs.python.org/3/whatsnew/3.6.html` — Současná verze oficiální implementace Pythonu *CPython*

však volně dostupné dekompilační nástroje. Při kompilaci jsou navíc zachovány informace o zdrojového kódu, jako jsou názvy proměnných a dokumentační řetězce. Obnovený kód je velice podobný původnímu, potence i resilience je tedy velmi nízká.

- Simon's Python Obfuscator — Webový nástroj pro Python 2.x s volně dostupnými zdrojovými kódy. Obfuskátor zvládá převod pouze jednoho vstupního souboru. V principu je program pouze rozdělen na tokeny a uložen jejich seznam spolu s pořadím ve formátu base64. Při následném spuštění výsledku je zdrojový kód opět převeden do původního stavu, zkompilován a spuštěn. Nástroj zároveň zachovává i komentáře a neprovádí žádné jiné změny, spíše než obfuskátor se tedy jedná o komprimační nástroj. Resilience je tedy nízká, stejně jako potence. [6]

- Oxyry Python Obfuscator — Webový nástroj pro Python 3.x, poskytována je za poplatek i offline verze. Mezi jeho schopnosti patří přejmenování funkcí, tříd, lokálních proměnných a argumentů. Odstraňuje všechny typy komentářů. Pokud zdrojový kód obsahuje speciální funkce, jako jsou `exec()`, dir(), locals() nebo globals(), výsledný program nemusí být stoprocentně funkční. To vychází z vlastností jazyka, kdy obfuskátor přejmenuje část kódu na které závisí dynamická funkcionalita. Odstranění komentářů a přejmenování jsou jednosměrné metody obfuskace, tedy nezvratné. Resilience je vysoká, potence střední, zejména proto, že přejmenování je kvalitní a použité identifikátory jsou náhodné. Čas běhu se oproti původnímu téměř nezmění. [7]

- Opy — Volně dostupný obfuskátor pro Python 3. Založen je na vlastním tokenizeru pomocí regulárních výrazů. Zvládá pouze přejmenování různých druhů identifikátorů. Zajímavá je pokročilejší konfigurace pro vyloučení některých částí z úprav. Aplikované úpravy jsou jednosměrné, resilience je tedy vysoká. Potenci hodnotíme po otestování jako střední. [8]

- pyobfuscate — Volně dostupný nástroj pro Python 2. Převod probíhá pomocí AST. Zvládá přejmenování identifikátorů a odstranění komentářů. Podporuje pouze Python 2 a jedná se o neudržovaný projekt. Úpravy jsou opět jednosměrné a resilience je vysoká, potence je opět střední. [9]

- pyminifier — Nástroj schopný provést minifikaci, kompresi a obfuskaci určený pro Python 3. Umí rekurzivně přejmenovat identifikátory, odstranit komentáře a specializací je vkládání Unicode znaků, které spadají do

[6] `http://pyobf.herokuapp.com/` — Oficiální stránky Simon's Python Obfuscator
[7] `http://pyob.oxyry.com/` — Oficiální stránky Oxyry Python Obfuscator
[8] `https://github.com/QQuick/Opy/` — Oficiální stránky Opy
[9] `https://github.com/astrand/pyobfuscate` — Oficiální repozitář pyobfuscate

vyšších pozic v tabulce. Nástroj je distribuován ve formě konzolové aplikace s nápovědou, jeho použití je poměrně snadné. Stejně jako v předchozích případech je resilience vysoká a potence střední. [10]

Kromě těchto nástrojů existují další způsoby, jak zkomplikovat čtení zdrojových kódů Pythonu. Nabízí se například kompilace přímo do spustitelného `.exe` souboru na platformě Windows.

Z popisu existujících nástrojů je zřejmé, že se jedná o úzce zaměřené jednoúčelové nástroje, které se typicky spoléhají na přejmenování identifikátorů a odstranění komentářů. Nevyužívají plný potenciál, který poskytuje modifikace abstraktního syntaktického stromu. Dále se dostatečně nezaměřují na vlastnosti specifické pro Pythonu a ponechávají prostor pro zpětnou analýzu aplikovatelnou na jisté programovací jazyky.

Tyto nedostatky se proto v dalším textu pokusíme pokrýt a navrhneme další modifikace zdrojového kódu za účelem rozšíření a zlepšení obfuskace. Navrhneme víceúčelový program, který bude multiplatformní a snadno použitelný. Dalším požadavkem bude modularita programu, která by měla umožnit jeho snadné rozšíření o další obfuskace.

1.5 Zhodnocení analýzy

V analytické části jsme popsali obfuskaci jako metodu úpravy zdrojových kódů a také obfuskátor jako nástroj, který tyto úpravy provádí. Dále jsme nastínili principy, které se při obfuskaci používají. Zavedli jsme tři charakteristiky, které mohou sloužit jako relevantní měřítka kvality obfuskačních technik. Zaměřili jsme naši analýzu na programovací jazyk Python 3. Dále jsme popsali existující metody a nástroje pro obfuskaci a pokusili se zhodnotit jejich přednosti a nedostatky.

Popsali jsme také existující a obecně používané obfuskační techniky. U popsaných metody jsme navíc uvedli možnosti aplikace pro programovací jazyk Python.

V další části uvedeme různé tradiční techniky obfuskace použitelné pro Python, spolu s nimi navrhneme i několik inovativních řešení. Kvality jednotlivých technik se pokusíme odhadnout za použití popsaných veličin. Navrhneme také funkční principy celého nástroje. Od našeho řešení očekáváme vysokou modularitu, schopnost přizpůsobení a vyšší počet jednotlivých změn ve vstupních kódech. Abychom toho dosáhli, modifikace budeme provádět pomocí abstraktního syntaktického stromu za použití popsaných metod.

[10] `https://liftoff.github.io/pyminifier/` — Oficiální stránky pyminifier

Kapitola 2

Návrh

V předchozí kapitole jsme popsali existující nástroje a zejména technologie, které využijeme při návrhu a implementaci obfuskačního nástroje. V této části navrhneme obfuskátor pro Python 3 jako modulární nástroj. Mezi jeho hlavní přednosti by měla patřit snadná rozšiřitelnost, důraz na znovupoužití a modularita. Dále pak navrhneme několik obfuskačních technik a pokusíme se nastínit jejich přednosti a využití.

2.1 Návrh řadiče

Abychom zajistili modularitu, oddělíme řídící část od obfuskačních modulů, které umístíme ve zvláštních souborech. Hlavní část bude tvořit jednoduchý program komunikující v textovém režimu, grafické rozhraní by pro takový nástroj nejspíš nemělo velký přínos. Hlavním úkolem bude zpracování parametrů příkazové řádky a jejich přenos jednotlivým modulům. Abychom mohli vytvářet jednotlivé obfuskační metody, program pro ně musí poskytovat jednotné rozhraní. Jednotlivé soubory obfuskátorů budou uloženy ve společné složce, řídící část provede při spuštění jejich enumeraci, vybrané z nich připraví pro použití a aplikuje na vstupní soubory. Popsané fáze jsou zobrazeny v grafu 2.1.

Z požadavků vyplývá přibližná struktura řídícího programu. Jednotlivé fáze budou následující:

1. První fází bude samotné načtení a parsování parametrů příkazové řádky. Kromě vstupní a výstupní složky pro načítání, resp. ukládání souborů bude mít uživatel možnost určit pořadí jednotlivých obfuskátorů. Dostupný by měl být také parametr pro specifikaci úrovně obfuskací. Program bude poskytovat nápovědu jak k jednotlivým modulům, tak i k celkovému použití. Mělo by jít také vypsat všechny moduly i podrobnosti pro jeden z nich. Dále by měl být podporován *verbose* režim, který bude

poskytovat podrobnější výpis informací. Další parametry postupně určíme během implementační fáze.

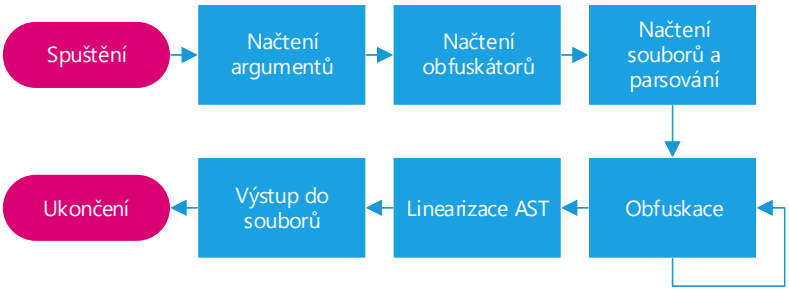
2. Načtení vstupních souborů ze vstupní složky a vytvoření příslušných syntaktických stromů AST.

3. Spuštění zvolených akcí nad jednotlivými syntaktickými stromy, které odpovídají příslušným vstupním souborům.

4. Zapsání modifikovaných souborů do výstupní složky tak, aby byla zachována funkcionalita původního programu.

V analytické části jsme uvedli, že obfuskace se mohou lišit úrovní svého působení. To musíme brát v úvahu i při návrhu řadiče všech operací. Abychom zvýšili kvalitu obfuskace, pokusíme se navrhnout nástroje tak, aby fungovaly na co nejvyšší úrovni. Budeme tedy předpokládat, že veškeré modifikace budou prováděny na celých modulech. V praxi to však znamená, že každý modul může obfuskace provádět na implementační zvolené úrovni bez rozdílu.

Důsledkem toho je nutnost provádět jednotlivé obfuskace sekvenčně za sebou tak, aby nemohla nastat případná nekonzistence mezi soubory. Zároveň musíme počítat s tím, že některé obfuskace mohou modifikovat názvy souborů, případně soubory slučovat a rozdělovat.



Obrázek 2.1: Navržené fáze obfuskátoru

Před spuštěním obfuskace tedy proběhne enumerace všech výstupních souborů předchozí úpravy. Akce bude poté postupně aplikována na samostatné soubory. Aby jednotlivé nástroje mohly provést předběžnou analýzu modulu, umožníme dvou-průchodovou aplikaci. V rámci hlavní smyčky budeme metodu pro spuštění obfuskace volat dvakrát s parametrem, který volanému kódu předá číslo průchodu. Dvou-průchodová aplikace by měla být volitelná a parametrizovatelná.

2.2 Návrh modulů

V této kapitole navrhneme jednotlivé obfuskace pro Python 3 jako moduly do výše popsaného nástroje. Každý z nich popíšeme jako algoritmus aplikovatelný na abstraktní strom. Všechny tyto nástroje budou později splňovat navržené rozhraní. Jednotlivé metody budeme vytvářet tak, abychom dosáhli kvalitní obfuskace s minimální cenou. Očekáváme také, že jednotlivé popsané metody bude možné modifikovat pro předchozí i následující verze Pythonu, případně pro jiné příbuzné programovací jazyky.

2.2.1 Odstranění komentářů

Oblíbenou obfuskační technikou, aplikovatelnou na většinu programovacích jazyků, je odstranění komentářů. V případě Pythonu existují kromě těch klasických uvozených znakem `#` ještě dokumentační, tzv. docstringy v trojitých uvozovkách sloužící k popisu tříd a metod. Jejich odstranění je typicky prvním krokem ke zpomalení analýzy. Výhodou může být také zmenšení velikosti souborů.

```
'''Docstring'''
def function():
  'Volny retezec'
  #Komentar
  return 3.14
```

```
def function():
  return 3.14
```

Obrázek 2.2: Program před a po odstranění komentářů.

Při parsování jsou komentáře uvozené znakem `#` automaticky odstraněny. O ty se tedy nemusíme starat. Dokumentační komentáře jsou reprezentovány stejně jako volně stojící řetězce. Zaměříme se tedy na vyhledání a odstranění uzlů, které reprezentují volně stojící řetězce. Tedy ty, které jsou přímo součástí posloupnosti výrazů.

```
Module(body=[
  Expr(value=Str(s='Docstring')),
  FunctionDef(
    name='function',
    body=[
      Expr(value=Str(s='Volny retezec')),
      Return(value=Num(n=3.14))]])
```

Listing 2.1: Předchozí zdrojový kód ve tvaru AST

Z výpisu 2.1 je zřejmé, že jediný případ, který musíme brát v úvahu je uzel typu `Str`, který se nachází jako výraz `Expr` přímo v těle modulu, či funkce.

Vytvořením této úpravy jako modifikace syntaktického stromu navíc obejdeme problém, kterým je kontext řetězce. Komentář v Pythonu je uvozen znakem `#`, přesto se najde mnoho případů, kdy se tento znak nachází a nenásleduje po něm komentář. Ošetřit všechny tyto případy pomocí regulárních výrazů může být obtížné, až nemožné. Obfuskace pomocí AST tedy proběhne pouhým odstraněním uzlů z příslušné pozice ve stromu.

Pro účely parametrizace obfuskace můžeme vytvořit dvoustavový příznak. V prvním případě modul odstraní pouze klasické komentáře a v druhém navíc ještě dokumentační. Tyto dva případy lze rozlišit podle kontextu, kde se daný řetězec nachází. Pokud stojí před definicí funkce, třídy, či metody, pak se jistě jedná o dokumentační komentář. V opačném případě jde o volně stojící řetězec.

Odstranění komentářů je jednoduchá a přesto poměrně účinná obfuskace, kterou disponují i konkurenční obfuskátory.

2.2.2 Relokace řetězců

Zdrojové kódy často obsahují řetězce ve formě konstantních výrazů. Protože jsou řetězce v Pythonu imutabilní, tedy jakákoliv operace s nimi vytvoří jejich hlubokou kopii, je možné tyto konstanty libovolně přesouvat. Zaměříme se na přesun do proměnných, kdy konstantní řetězce umístíme na počátek souboru, a jejich původních výskyty nahradíme novými proměnnými. Výpis 2.3 zobrazuje přesun řetězců v rámci jednoho souboru. Jako vedlejší projev obfuskace proběhla deduplikace řetězců, které se v původním kódu vyskytovali vícekrát.

Popíšeme postup pro obecnou variantu algoritmu, tedy dvou-průchodovou verzi, která dokáže přemístit řetězce z celého modulu do nově vytvořeného souboru. Ten bude nutné v každém obfuskovaném souboru načíst a nahradit výskyty příslušných řetězců proměnnými.

```
x = "val: [" + v + "]"
y = "ret: [" + v + "]"
```

```
rand1 = "val: ["
rand2 = "ret: ["
rand3 = "]"

x = rand1 + v + rand3
y = rand2 + v + rand3
```

Obrázek 2.3: Program před a po relokaci řetězců.

V první fázi projdeme všechny soubory upravovaného modulu a uložíme si řetězce pro pozdější použití. Ke každému řetězci přiřadíme unikátní identifikátor, který ho bude zastupovat. Pokud nalezneme dva stejné řetězce, použijeme dřívější identifikátor. Jejich neměnnost zaručuje správnou funkčnost i pokud jedna proměnná bude použita na více místech. Enumerace všech řetězců bude

čistě pasivní. Při každé návštěvě uzlu typu `Str`, tedy řetězce si uložíme jeho obsah a vygenerujeme nový identifikátor.

Po průchodu všemi soubory bude spuštěna druhá fáze. Obfuskátor nejprve vytvoří nový soubor, do kterého vypíše všechny proměnné spolu s přiřazenými uloženými řetězci. Tento přidaný soubor bude nutné importovat do každého procházeného modulu tak, aby bylo možné použít globální proměnné nacházející se v jiném souboru. Zároveň budeme nahrazovat původní řetězce vygenerovanými proměnnými. Každý uzel typu `Str` nahradíme novým, typu `Name`, který slouží pro načítání hodnot z proměnných.

Kromě relokace na úrovni modulu, kdy je vytvořen pouze jeden soubor, můžeme algoritmus upravit a vytvořit souborů více, případně umístit řetězce na začátek příslušných souborů.

Pokud bychom chtěli řetězce přemísťovat na náhodná místa ve zdrojových kódech tak, aby byla zachována funkcionalita, museli bychom správně odhadnout kontext, do kterého je možné provést přemístění. Pro složitost takového řešení tuto variantu dále používat nebudeme.

2.2.3 Relokace numerických konstant

Python v základní verzi podporuje několik typů čísel. Kromě celočíselného typu *integer* s neomezenou přesností jsou implementována i desetinná čísla *float*. Jejich přímým rozšířením jsou navíc čísla komplexní. Tyto tři typy čísel lze zapsat přímo pomocí literálů ve zdrojovém kódu. Stejně jako řetězce jsou i konstanty immutable, chovají se tedy jako neměnné bloky paměti.

Princip obfuskace bude shodný s přístupem, který platí pro řetězce. Popíšeme jej tedy jako shrnutí předchozího algoritmu. Obecná varianta bude opět dvou-průchodová. V první části projdeme výchozí strom AST a najdeme všechny konstanty. K těmto číslům vygenerujeme unikátní identifikátory a dvojici uložíme. V druhé fázi navážeme na enumerační a nahradíme konstanty proměnnými. Na závěr přidáme do zvláštního souboru všechny proměnné spolu s čísly a na přidaný soubor se odkážeme ve všech ostatních.

```
x = 2 + 3i
y = 148.2
```

```
rand1 = 2
rand2 = 3i
rand3 = 148.2

x = rand1 + rand2
x = rand3
```

Obrázek 2.4: Program před a po relokaci konstant.

I tuto obfuskační techniku lze parametrizovat, můžeme postup upravit a například vytvořit souborů více, případně umístit proměnné na začátek příslušných

zdrojových kódů.

2.2.4 Rozložení numerických konstant

Kromě přemístění konstant se můžeme zaměřit na změnu jejich tvarů, tedy na zesložitění výrazů. Jednotlivé konstanty lze skládat pomocí vhodných matematických operací tak, aby byl zachován požadovaný výsledek.

Práce s čísly je však v programování náchylná na chyby a operace často mohou končit neočekávanými výsledky. Námi provedené modifikované výrazy by tak mohly vést například k výsledkům, které jsou nereprezentovatelné. Výsledná hodnota by pak byla samozřejmě špatně. Mohlo by například dojít k přetečení čísel, či změně přesnosti čísel desetinných. Z tohoto důvodu se musíme omezit pouze na celočíselné konstanty.

Komplexní čísla jsou v Pythonu implementována jako rozšíření čísel reálných. Sdílí spolu tedy část vlastností. Kritická je zejména přesnost, resp. rozdíl skutečného výstupu od očekávaného algebraického výsledku. Každá doplněná operace může vnést do výsledku chybu a to si nemůžeme dovolit. Naši pozornost proto zaměříme na celá čísla typu *integer*. Ta navíc Python implementuje s neomezeným rozsahem, nemusíme se tak starat o problémy s přetečením apod [11].

V úvahu přichází zejména tyto celočíselné operace:
sčítání, odčítání, posun vlevo, posun vpravo, bitový *or*, bitový *and*, bitový *xor*, zbytek po dělení *mod* [11].

Vlastnosti rozkladu závisí zejména na volbě příslušných operací a také rozsahů generovaných čísel. Důležitým parametrem je hloubka vytvořených stromů resp. rekurze. Protože používáme binární operace, roste počet generovaných uzlů exponenciálně, je proto nutné tento parametr nastavit rozumně.

2.2.5 Přejmenování souborů

Soubory v projektech jsou typicky pojmenovávány tak, aby název reflektoval obsah. Abychom tuto informaci zakryli, přejmenujeme soubory pomocí unikátních identifikátorů. Během přejmenování je nutné brát v úvahu fakt, že Python umožňuje vzájemné odkazování souborů a modulů 2.2.

```
import file2

def dump():
  print(dir(file2))
dump()
```

Listing 2.2: Načtení externího souboru pomocí jeho jména

Na počátku enumerujeme všechny složky a soubory zadaného projektu. Z relativních cest vůči počáteční složce odvodíme relativní jména i absolutní

[11] Pro zbytek po dělení je nutné ošetřit případ, kdy je dělitel nula.

cesty k modulům. Ke každému z nich si uložíme náhodný identifikátor, který později použijeme k přejmenování souborů.

```
Module(body=[
  Import(names=[alias(name='file2')]),
  FunctionDef(name='dump',
    body=[Expr(value=Call(
      func=Name(id='print'),
      args=[Call(
        func=Name(id='dir'),
        args=[Name(id='file2')])]))]),
  Expr(value=Call(
    func=Name(id='dump')))])
```

Listing 2.3: Zkrácená verze AST ke zdrojovému kódu 2.2

V další fázi začneme procházet syntaktické stromy reprezentující jednotlivé vstupní soubory. Ve chvíli, kdy narazíme na uzly typu *Import* a *ImportFrom*, provedeme nahrazení za vygenerované jméno modulu. Tak zachováme relativní cesty mezi soubory. Běhové prostředí by se také mělo automaticky postarat o přejmenování souboru.

2.2.6 Změna notace atributů

Python jako objektově orientovaný jazyk umožňuje přístup k atributům jednotlivých instancí objektů. Zápis, který je k tomuto účelu používán, se nazývá tečková notace. Běhové prostředí také umožňuje použít metody `getattr` a `setattr` pro načtení, resp. uložení hodnoty do atributu. Využijeme možnosti tohoto převodu v náš prospěch a převedeme tečkovou notaci na zápis využívající volání funkcí [12].

```
a.b.c = 10
print(x.y.z)
```

```
setattr(
  getattr(a, 'b'),
  'c', 10)

print(
  getattr(
    getattr(x, 'y'),
    'z'))
```

Obrázek 2.5: Program před a po převodu na dynamické atributy.

Funkce *getattr* přijímá dva parametry — zdrojový objekt a název parametru, vrací příslušnou hodnotu. Funkce *setattr* má téměř stejné rozhraní, přijímá navíc třetí parametr, kterým je hodnota pro uložení.

Celou modifikaci rozdělíme na dva případy. V prvním budeme nahrazovat načítání atributu funkcí *getattr*. Při průchodu stromu nahradíme výskyty uzlu *Attrib*, který slouží načítání atributu. To poznáme podle nastaveného kontextu, který udává, zda je atribut určen pro čtení, nebo zápis. V druhém případě uzly pro zápis přímo nahradíme voláním funkce *setattr*. Tato varianta je komplikovanější, protože Python umožňuje vícenásobné přiřazení.

```
a.b, c.d, e = (1, 2, 3)
```

Listing 2.4: Ukázka přiřazení, které nelze jednoduše převést

Na pravé straně takového výrazu, může být v obecném případě n-tice, jejíž rozměry a struktura jsou v čase kompilace neznámé. Přiřazení více proměnných paralelně se z pohledu programátora jeví jako atomická operace. Nahrazení vícero funkcemi `setattr` tedy musíme zavrhnout, protože bychom tak mohli změnit konzistenci a výsledek operace. Zaměříme se proto pouze na jednoduchá přiřazení.

Algoritmus pro ukládání do atributu tedy musí ověřit, že se jedná pouze o jednoduché přiřazení. Pouze v takovém případě provedeme nahrazení.

Protože přímé načítání a ukládání atributů převádíme na funkční volání, můžeme očekávat časovou penalizaci za převod. Ať už z důvodu nedostupného cachování, či přímo volání funkcí. Oficiální implementace takový případ rozebírá a dochází k závěru, že přibližný rozdíl činí v průměru 1% [13].

2.2.7 Vkládání predikátů

Předchozí modifikace pracovaly pouze s modifikacemi původních souborů. Abychom zvýšili složitost zdrojových kódů vložíme do nich nové výrazy. Tyto výrazy mají často tvar tzv. *opaque predicates*, tedy podmínek u nichž je dopředu známo, zda budou vyhodnoceny kladně, nebo záporně. Nejčastěji podmínky zároveň volíme tak, aby nebylo na první pohled zřejmé, kterým způsobem budou vyhodnoceny. Kvalita, počet podmínek a jejich tvary jsou pak závislé na kreativitě implementace.

Syntaxe Pythonu podporuje několik typů podmíněných výrazů. Od klasických podmínek *if—else*, pře cykly *while* a *for*, až po *ternární operátor*. Každý podmíněný výraz včetně cyklů má navíc svůj negativní tvar, který můžeme při obfuskace také využít.

```
while False:
  pass
else:
  code_goes_here()
```

Listing 2.5: Negativní větev u cyklů

Princip obfuskace založíme na tom, že některé uzly, jako jsou například definice tříd, funkcí, či moduly, obsahují těla složená z velkého počtu uzlů. Mezi těmito uzly, které tvoří posloupnost, vždy vybereme dva, které budou

tvořit pomyslnou zarážku. Vytvoříme nový podmíněný výraz, jehož splnitelnost předem určíme. Jeho tělo bude tvořit seznam uzlů, mezi dvěma náhodně vybranými. Posloupnost uzlů z původního těla odstraníme a nahradíme vytvořenou podmínkou.

Ternární operátor se tedy pro naše aplikace nehodí, protože jeho tělo může obsahovat pouze jeden výraz a my bychom chtěli aplikaci provést na náhodném počtu výrazů. Typ vložených podmínek můžeme určit náhodně stejně jako predikáty, které do nich budeme vkládat. Predikáty je navíc možné dynamicky generovat, počet a rozmanitost tedy závisí pouze na implementaci.

Kromě použitých konstrukcí se musíme zaměřit i na výrazy, které určují logickou splnitelnost. Podmínky mohou být vytvořeny na několika úrovních od globálních proměnných až po lokální konstanty. Pro maximální efektivitu je možno podmínky vytvářet na úrovni modulu.

Dále je možné zapojit složitější výrazy jako jsou funkční volání, lambda výrazy, generátory, pole, hašovací tabulky apod. Vhodný je jakýkoliv výraz, u kterého je dopředu jisté, jakým způsobem bude vyhodnocen. Dynamičnost Pythonu navíc umožňuje vytvoření dynamických predikátů, které se s každým průchodem budou měnit. Například v závislosti na konečném automatu, či předdefinované sekvenci.

Případně je možné tyto možnosti kombinovat a vhodně řetězit. Od zvolených kombinací se také odvíjí výsledná časová náročnost. Je nutné brát v úvahu, že při každém průchodu přidanou podmínkou jsou její argumenty znovu vyhodnoceny. V průběhu implementace by bylo možná vhodné omezit místa, do kterých budou podmínky injektovány a vyhnout se například cyklům apod.

Důležitá je také míra náhody, tedy zapojení randomizace do generování predikátů.

Abychom mohli omezit počet nových výrazů, zavedeme si míru udávající počet vložených uzlů na 100 výrazů. Statisticky tak můžeme ovlivnit počet vložených podmínek. Každý uzel, tedy bude mít stejnou pravděpodobnost nahrazení podmínkou a tuto hodnotu můžeme parametrizovat.

2.2.8 Zakrývání vestavěných funkcí

Python 3 má řadu vestavěných funkcí, které slouží k různým účelům, od vstupně (`input`, ...) — výstupních (`print`, ...), přes konstruktory (`object`, `dict`, `set`, ...), až po ladící funkce (`memoryview`, `dir`, `hasattr`, ...). Můžeme prohlásit, že téměř každý program používá tyto funkce, protože jejich funkcionalita je nepostradatelná. Rozhraní a účel těchto funkcí je zdokumentován, při analýze tedy mohou být využity jako záchytné body. V této obfuskaci se proto zaměříme na zakrytí informace o skutečně volané funkci.

```
a = input()
c = set(a)
```

```
print(c)
```

Listing 2.6: Ukázkový zdrojový kód

Zaměříme se tedy na zakrytí těchto funkcí tak, aby nebylo na první pohled zřejmé, která z nich je volána. Tyto funkce jsou za běhu importovány z modulu builtins. Toho využijeme a místo globálních importovaných jmen využijeme lokální pomocí funkce `getattr`, která tedy zůstane jedinou očividnou.

```
import builtins as _
a = getattr(_, 'input')()
c = getattr(_, 'set')(a)
getattr(_, 'print')(c)
```

Listing 2.7: První krok k obfuskaci kódu 2.6

Pro účely analýzy z dokumentace uložíme seznam vestavěných funkcí [12]. Modifikaci pak založíme opět na průchodu AST, budeme hledat volání funkcí, které obsahuje funkci nalezenou v seznamu. Pokud takovou najdeme, její uzel nahradíme novým, který bude odpovídat volání funkce `getattr` s příslušnými argumenty. Na začátek modulu musíme přidat výraz pro importování modulu `builtins`.

Tím, že převedeme název volané funkce na řetězec, můžeme následně s výhodou aplikovat další obfuskační akce pro jejich zakrývání. Pokud bychom chtěli princip ještě rozšířit, můžeme přidat další úroveň indirekce doplněním funkce, které bude obsahovat volání getattr s příslušnými parametry. Navíc bychom mohli jména vestavěných funkcí modifikovat a získávat je dynamicky pomocí složitějšího mechanismu.

Možným rizikem je program, ve kterém programátor omylem, nebo cíleně definuje funkci se jménem, které se překrývá s vestavěnou funkcí. Navržený algoritmus takový případ nerozezná a bude volat původní funkci z modulu `builtins`. Z důvodu dynamičnosti Pythonu navíc neexistuje obecné řešení tohoto problému, protože funkce mohou být kompilovány a přidávány za běhu. Druhým potenciálním problémem je změna seznamu vestavěných funkcí mezi verzemi obfuskátoru a Pythonu na cílové platformě.

2.2.9 Přejmenování argumentů

Jména proměnných a argumentů mohou být cenným zdrojem informace. Jednou ze zavedených obfuskačních technik je jejich přejmenování. Dynamičnost Pythonu celou situaci komplikuje. Funkce totiž mohou mít jmenné parametry, u kterých je jméno kritické při volání funkce. Během analýzy je téměř nemožné odhadnout jmenné argumenty příslušící dané funkci. Dokonce ani IDE pro Python neposkytují dostatečnou nápovědu pro toho chování.

Podobný problém hrozí u proměnných, které mohou být definovány dynamicky. Některé části kódu bychom tedy úspěšně přejmenovali a jiné by mohly

[12] `https://docs.python.org/3/library/functions.html` — Seznam vestavěných funkcí.

zůstat závislé na původních jménech. Zaměříme se tak pouze na přejmenování parametrů.

```
def func(a, b, c, *args,
    d=1, e=2, **kwargs):
  pass
```

```
def func(xxx, yyy, zzz,
    *aaa, d=1, e=2, **bbb
    ):
  pass
```

Obrázek 2.6: Program před a po přejmenování argumentů funkce.

Python podporuje několik typů argumentů. V ukázce 2.6 jsou použity všechny jejich typy:

- *a*, *b*, *c* — Normální parametry známé z ostatních jazyků, jako takové je lze jednoduše přejmenovat.
- **args* — Pole obsahující předané dynamické argumenty.
- *c*, *d* — Jmenné argumenty, které by při přejmenování proměnných mohly působit problémy. Jejich jména hrají důležitou roli při volání funkce, které není možné vždy detekovat.
- ***kwargs* — Seznam dynamických argumentů s klíčovými slovy, jedná se o tabulku jmen a hodnot.

Přejmenování argumentů není naprosto přímočaré, protože ve funkci se může vyskytnout definice jiné funkce se stejným názvem argumentů. Při změně názvů musíme tedy brát v úvahu jmenný prostor funkce — kontext.

Vytvoříme tedy objekt reprezentující kontext, který bude ukládat k dané funkci názvy jejích argumentů. Abychom byli schopni reflektovat hierarchii funkcí, bude si kontext udržovat odkaz na rodiče. Tento jednoduchý jmenný prostor umožní vyhledání jména v daném stromu funkčních definic.

Algoritmus přejmenování spustí průchod stromem AST, najde uzly reprezentující funkce a metody tříd. Pro každý průchod definicí vytvoříme nový kontext a přidáme do něj argumenty spolu s novými identifikátory. Pokud v těle narazíme na vnořenou definici funkce, rekurzivně aplikujeme stejný postup s tím rozdílem, že při vytvoření kontextu nastavíme správně jeho rodiče. Dalším krokem je samotné přejmenování použitých argumentů. Při něm vyhledáme podle původního názvu nový identifikátor v příslušném kontextu.

2.2.10 Převod definic na AST

Během spuštění programu vytvořeného v Pythonu probíhá na pozadí několik operací. Zdrojový kód není spouštěn přímo, ale je kompilován. Dochází tak

k převodu, během něhož je vstupní kód převeden na AST pomocí vestavěného parseru. Dále probíhá linearizace při níž se z AST vygeneruje bytekód. Ten je následně proveden interpreterem. Kromě tohoto přímého způsobu existuje druhý, při němž je kód předkompilován a uložen jako bytekód s příponou `.pyc`. Ten je později opět interpretován.

Tyto znalosti využijeme při tvorbě obfuskačního modulu. Python kromě kompilace umožňuje také dynamické spouštění výsledného kódu. K tomuto účelu slouží funkce `compile`, která přijímá program ve tvaru AST zapsaný pomocí konstruktorů příslušných objektů z modulu `ast`. Tato funkce vrátí zkompilované bytekód, který lze interpretovat pomocí funkce `exec`. Flexibilita Pythonu tak umožňuje vytvářet komplikované a dynamické programy a my ji využijeme k tvorbě obfuskační techniky.

Hlavní myšlenkou modifikace je převod některých částí zdrojového kódu na syntaktický strom, který bude uložen v příslušném souboru. Tento AST uložíme do globální proměnné, na vhodném místě provedeme jeho kompilaci a výsledný kód interpretujeme na původním místě. Původní kód tedy odstraníme a nahradíme ho voláním funkce `exec`, která do globálního jmenného prostoru injektuje původní kód.

```
def funkce():
  return 10
```

Listing 2.8: Vzorový program pro obfuskaci

Takto zkompilovat se dá jakýkoliv výraz, musíme brát však v úvahu jeho kontext. Problematické jsou lokálně definované funkce, které pomocí funkce `exec` nemohou být správně definovány. Důvodem je způsob implementace lokálních proměnných, které funkce `exec` nemůže správně modifikovat [14].

Naši pozornost tedy zaměříme na globální definice funkcí a tříd. Vytvoříme asociativní tabulku vygenerovaných identifikátorů a syntaktických stromů. Uzly, které odpovídají definicím, uložíme do této tabulky a nahradíme je voláním funkce `exec` s identifikátorem. Na konci analýzy projdeme uložené pole a na začátku modulu přiřadíme do jednotlivých proměnných příslušné syntaktické stromy. Tyto stromy před přiřazením musíme ještě zkompilovat, abychom se vyhnuli případné opakované kompilaci v cyklech.

Na závěr bude ještě nutné importovat modul `ast`, protože právě jeho funkci `compile` využijeme při kompilaci uložených stromů do bytekódu. Závislost na tomto modulu nás nijak neomezuje, protože je jedná o vestavěný modul Pythonu.

```
from ast import *
var = Module(body=[
  FunctionDef(name='funkce', args=arguments(),
    body=[Return(value=Num(n=10))])])
exec(compile(var, '', 'exec'))
```

Listing 2.9: Program 2.8 po aplikaci obfuskace — zkráceno

Teoreticky bychom mohli místo AST vložit do výsledného programu přímo zkompilovaný bytekód a rovnou ho interpretovat. Toto řešení by ale mohlo snížit přenositelnost, která je pro Python zásadní. Stačil by totiž minimální rozdíl mezi verzemi interpreteru a instrukční sady v různých prostředích a dopady by mohly být zásadní. Python samozřejmě mezi majoritními verzemi zachovává zpětnou kompatibilitu, přesto se jedná o riziko. Bezpečnější a přenositelnější je tedy uložení syntaktického stromu a kompilace až pomocí lokálního kompilátoru.

Výhodou této obfuskace je návaznost na ostatní metody. Metoda totiž některé části zdrojového kódu převede na řetězce, které opět podléhají jiným metodách obfuskace.

2.2.11 Přidání dekorátorů

Python poskytuje syntaxi pro vytváření tzv. funkčních dekorátorů [15, s. 28]. Dekorátor je funkce, nebo třída, která rozšiřuje nebo mění funkcionalitu jiné funkce, či třídy. Princip je jednoduchý, při zavolání dekorované funkce je nejprve zavolán dekorátor, kterému je původní funkce předána jako parametr. Na pozadí je tedy vytvořena nová funkce. V principu jde o vložení mezivrstvy modifikující vstup a výstup funkce.

Úkolem dekorátoru je typicky modifikovat původní funkci a vrátit ji jako návratovou hodnotu. Dekorátor často definuje novou vnitřní funkci, která poté volá původní s upravenými parametry, či modifikuje její návratovou hodnotu. Místo dekorované funkce je následně vrácena nově definovaná.

```
def funkce(a, b, c, *vargs, d=1, **kwargs):
  print(a, d)
  return b + c
```

Listing 2.10: Vzorový program pro obfuskaci

Techniku obfuskace založíme na modifikaci nalezených funkcí pomocí přidaného dekorátoru. Průchodem stromu AST najdeme definice metod a funkcí, následně je upravíme přidáním náhodných argumentů. To znamená, že původní volání těchto funkcí budou neplatná, protože se změní rozhraní funkce. Abychom zachovali chování původního programu, použijeme právě dekorátor. Ke každé modifikované funkci vytvoříme v AST uzel funkce, která bude reprezentovat nový dekorátor. K původní definici funkce přidáme nově definovaný dekorátor. Ten bude volat funkci s původními argumenty na správných místech, přidané parametry můžeme vyplnit náhodnými daty.

```
def wrapper(func):
  def inner(*vargs, **kwargs):
    return func(-9, *vargs, parB=-42, **kwargs)
return inner
```

```
@wrapper
def funkce(parA, a, b, c, *vargs, d=1, parB=0, **
    kwargs):
  print(a, d)
  return (b + c)
```

Listing 2.11: Program 2.10 po aplikaci obfuskace

Výpis 2.11 zobrazuje základní verzi obfuskace — dekorovanou funkci, které byly automaticky přidány parametry *parA* a *parB*. Zbylé argumenty jsou funkci předány pomocí dekorátoru.

Touto úpravou dosáhneme znepřehlednění programu. Místo volané funkce se nejprve zavolá dekorátor, který doplní nerelevantní parametry. Analýza se tím zkomplikuje, protože argumenty volání nebudou odpovídat těm na straně volané funkce.

Možným rozšířením je provázání s modulem, který injektuje predikáty. Tyto podmínky vložené do těla funkce by zesílily dopad obfuskace a zkomplikovaly analýzu. Predikáty by mohly záviset pouze na přidaných parametrech, jejichž hodnotu můžeme sami určit. Výsledný kód by v takovém případě vypadal, jako kdyby byl vytvořen záměrně a ne činností obfuskace.

2.2.12 Sloučení funkcí

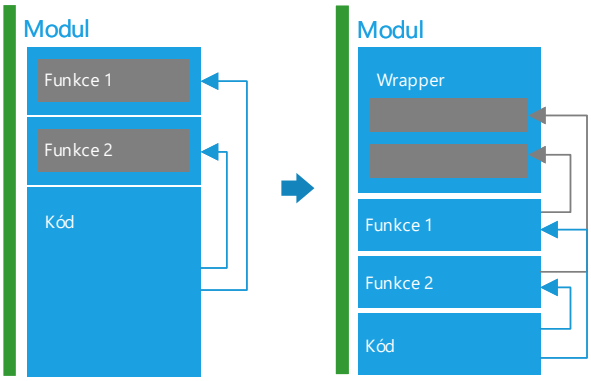
Vlastnosti Pythonu dovolují sloučení několika funkcí do jedné. Aby úprava dávala smysl, musí být vybrané funkce vždy definovány na stejné logické úrovni. Základní myšlenkou je rekurzivní aplikace obfuskace, která bude slučovat vhodné funkce na stejných úrovních vnoření.

Po sloučení několika funkcí vznikne tělo nové funkce. Aby byla zachována sémantika programu, umístíme tuto funkci před první ze slučovaných funkcí. Tělo nové funkce bude obsahovat podmínky určující, která z původních funkcí je volána. Tyto nové podmínky budou vytvořeny z těl původních funkcí. Tím nebude poškozena funkcionalita, protože zůstanou zachovány reference, globální proměnné, parametry a také návratové hodnoty.

Aby bylo možné tuto funkci volat, musíme vyřešit několik překážek. Každá ze slučovaných funkcí může mít jiné rozhraní, je tedy nutné sjednotit parametry. Jedním přístupem je využití dynamických argumentů `*vargs` a `**kwargs`, které umožňují předání libovolného počtu parametrů, resp. jmenných parametrů.

Druhým možným řešením je uložení původních argumentů jako atributy nějakého objektu. Python totiž umožňuje dynamické vytváření atributů pro některé datové typy, jako je například lambda výraz. Tento objekt můžeme přidat jako jeden argument, každá z původních funkcí se může poté odkazovat na původní argumenty pomocí atributů.

Druhou otázkou je způsob volání této vytvořené funkce. Z principu nelze detekovat všechny možné případy volání původních funkcí. Využijeme tedy



Obrázek 2.7: Schéma slučování

toho, že původní definice jsou vytvořeny na správných místech tak, aby program fungoval správně.

Odstraníme obsahy jejich těl a přidáme volání námi vytvořené funkce. Předáme definovaným způsobem původní argumenty a doplníme nový, ze kterého půjde odvodit funkce, jež měla být volána. Původní definice tedy budou sloužit pouze jako proxy objekt, který deleguje funkční volání.

Kromě argumentů musíme také zajistit vrácení příslušné návratové hodnoty.

```
def funkceA(a, b):
  return a - b

def funkceB(c, d):
  return c + d
```

Listing 2.12: Vzorový program pro obfuskaci

Poslední překážkou může být případ, kdy jedna z funkcí obsahuje klíčové slovo `yield`. `Yield` v Pythonu slouží k vrácení hodnoty z funkce podobně jako `return` s tím rozdílem, že funkce se stává generátorem [15, s. 3]. Tedy objektem, který využívá princip *lazy evaluation* a umožňuje vícenásobné vrácení hodnoty ve chvíli, kdy je to potřeba.

Problém může nastat v případě, kdy jedna z původních funkcí používá klíčové slovo `yield`. Tím se z celé sloučené funkce stane generátor a případné další volání končící slovem `return` vrátí výjimku.

```
def merged(fn, proxy, *vargs, **kwargs):
  if fn == 'funkceB':
    return (proxy.c + proxy.d)
  elif fn == 'funkceA':
    return (proxy.a - proxy.b)
```

```
def funkceA(a, b):
  proxy = (lambda : None)
  proxy.a = a
  proxy.b = b
  return merged('funkceA', proxy)

def funkceB(c, d):
  proxy = (lambda : None)
  proxy.c = c
  proxy.d = d
  return merged('funkceB', proxy)
```

Listing 2.13: Program 2.12 po aplikaci obfuskace

Tento problém obejdeme tak, že těla původních funkcí vložených do podmínek obalíme znovu definicemi funkcí, které ihned v příslušné větvi zavoláme. Námi upravené parametry funkce automaticky použijí z kontextu sloučené funkce, jiný rozdíl tedy nenastane. Ukázka obfuskace 2.13 zobrazuje provedené úpravy ve zkrácené verzi, která neošetřuje problém s generátory.

```
def funkce():
  return 10
#Predefinovani funkce
funkce = None
#Dynamicka verze tehoz problemu
globals()['funkce'] = None
```

Listing 2.14: Ukázka problematického chování

Možným rizikem této úpravy je špatně vygenerovaný kód plynoucí z dynamičnosti Pythonu. Problém může být například předefinování funkce na jiný typ objektu. Tento problém nastiňuje ukázka 2.14. Druhým případem může být funkce `exec` volaná mezi definicemi. Pokud jednu z těchto funkcí předefinuje, není možné zajistit bezchybnou funkčnost. Toto statická analýza nepostihne a vygenerovaný kód může být chybný. Abychom tomu předešli můžeme projít AST a detekovat úrovně na kterých jsou tyto funkce použity. Taková místa budou označena jako nebezpečná a obfuskace se neprovede.

2.2.13 Šifrování řetězců

Přesto, že modul pro přemístění řetězců poskytuje určitou míru obfuskace, stále je poměrně snadné ji odstranit. Abychom situaci znepřehlednili více, můžeme přistoupit k šifrování řetězců. Zde se nabízí hned několik možností a otázek, které je potřeba zodpovědět.

Za předpokladu, že zašifrujeme řetězce během samotné obfuskace a uložíme výsledek do souboru, bude nutné do programu vložit kód pro dešifrování. Dešifrováním chceme získat původní řetězce, dešifrovací klíče tedy musí přesně

odpovídat šifrovacím a musíme vytvořit způsob, jak je odvodit z výstupního souboru. Abychom navíc nezvyšovali délku běhu výsledného programu, musí být dešifrování provedeno co možná nejrychleji, ideálně pouze jednou a v místě, kde je to vhodné.

Musíme navíc vybrat vhodný šifrovací algoritmus tak, aby jeho použití nebylo závislé na externích modulech a zároveň bylo dostatečně rychlé. V úvahu připadají různé implementace blokových či proudových šifer. Vhodná by mohla být například proudová šifra *RC4* [16, s. 14]. Přesto, že již není standardy doporučována, je možné ji s výhodou použít. V tomto kontextu není tak důležitá samotná kvalita šifrování, jako spíše snadná implementace a nulová závislost na externím kódu [17].
Použití šifry *RC4* je následující:

V závislosti na zvoleném klíči jsou inicializovány vnitřní hodnoty algoritmu, který posléze generuje libovolně dlouhý proud bytů, jímž lze šifrovat. Šifrování i dešifrování probíhá téměř stejně. Data jsou v obou případech převedena na proud bytů a je na ně aplikována operace `xor` s generovaných klíčem.

Data je nutné za běhu rozšifrovat. Musíme tedy navrhnout funkci, u které nebude na první pohled zřejmý její účel, ale která bude tuto operace bezpečně provádět. Výhodou je, že na funkci můžeme aplikovat vybrané obfuskace tak, aby si zachovala rychlost a přesto byla nečitelná.

Abychom mohli data dešifrovat, je navíc nutné získat klíč. Kvalita obfuskace je přímo závislá na naší schopnosti skrýt jeho hodnotu, případně metodu, kterou je odvozen. Bezpečnějším postupem by bylo dynamické získání klíče z externího zdroje, např. zabezpečeného serveru. Abychom však neomezili přenositelnost a použitelnost výsledného kódu, pokusíme se klíč získat dynamicky ze zdrojového kódu.

```
print('Hodnota: ' + name + 'kg')
```

Listing 2.15: Vzorový program pro obfuskaci

Pokud bychom se mohli spolehnout na to, že obsah souboru zůstane konstantní, mohli bychom klíč odvodit přímo z obsahu a atributů skriptů. Dá se však očekávat, že po této modifikaci budou aplikovány další obfuskace, které by mohly strukturu narušit a proto tuto metodu zavrhneme.

Další možností je například vygenerování klíče určité délky předem, zašifrovat jím řetězce a uložit do souboru místo klíče návod k jeho sestavení. Můžeme tedy uchovat například pouze indexy ze zašifrovaného řetězce tak, že po aplikace vhodné matematické operace vytvoří data na příslušných indexech zvolený klíč.

Algoritmus nejprve projde AST a vyhledá uzly reprezentující řetězce. Ke každému z nich vytvoří proměnnou, která ho bude reprezentovat. Jednotlivé řetězce budou spojeny do jednoho. Abychom později mohli získat části řetězce, je nutné uložit pole prefixových součtů jejich délek. Původní uzly nahradíme zároveň načítáním z proměnných.

```
ciphered = b'...'
prefixes = [...]
def decrypt(inx):
  ...

varA = decrypt(0)
varB = decrypt(1)

print(varA + name + varB)
```

Listing 2.16: Program 2.12 po aplikaci obfuskace

Řetězec dále zašifrujeme zvolenou metodou a vložíme na počátek v AST. Zároveň doplníme uzly reprezentující příslušnou dešifrovací funkci a další, jejichž smyslem bude získání klíče. Vytvoříme volání této dešifrovací funkce tak, že využijeme spočítaný prefixový součet a klíč. Výstup z této funkce uložíme do příslušné proměnné. Toto ukládání do proměnných zajistí to, že jednotlivé řetězce budou dešifrovány pouze jednou i při vícenásobném výskytu.

Tato metoda obfuskace je silně variabilní a závisí na výběru jednotlivých komponent. Tuto možnost bychom mohli využít při parametrizaci. Jedním z parametrů by mohl být například použitý algoritmus, či struktura dešifrovací funkce.

2.3 Zhodnocení návrhu

V této kapitole jsme navrhli obecný nástroj, obfuskátor, pro programy napsané v jazyce Python 3. Odlišností od existujících řešení je zejména důraz na modularitu přístupu a možnosti rozšíření. Místo jedné komplexní metody jsme navrhli použití několik jednodušších, jejichž efekty se budou kumulovat. Dále jsme navrhli využití abstraktního syntaktického stromu jako nástroje umožňujícího efektivní modifikace vstupního kódu.

Kromě hlavního programu jsme vytvořili i řadu obfuskačních technik. Některé jsou obecně aplikovatelné i na jiné programovací jazyky, jiné cílené na vlastnosti jazyka Python. U každé z nich jsme popsali základní principy a přístup, shrnuli jsme očekávané vlastnosti a pokusili se určit jeho výhody a nevýhody. Námi navržený seznam obfuskací není zcela vyčerpávající a jistě je možné se zaměřit na další vlastnosti specifické pro Python, například *properties*, asynchronní programování, či generátory by mohly být vhodnými prostředky obfuskace.

V další kapitole se přesuneme k samotné implementaci. V textu popíšeme vytvoření řídícího programu spolu s jednotlivými moduly. Zvolíme konkrétní parametry a vlastnosti obfuskačních modulů. Dále každý z nich zhodnotíme z pohledu implementace a zvolených veličin: určíme resilienci, potenci a cenu.

Kapitola 3

Implementace

V této kapitole se zaměříme na samotnou implementaci námi navrženého nástroje. Výsledkem by měl být spustitelný program a několik obfuskačních modulů. Program tyto moduly enumeruje a umožní uživateli jejich spuštění přímo z příkazové řádky.

3.1 Technologie implementace

Abychom mohli využít přednosti úprav pomocí AST, musíme se zabývat otázkou tvorby parseru a technologií obfuskátoru. Jednou z možností je tvorba vlastního parseru, ručně, či pomocí dostupných nástrojů přímo z gramatiky jazyka. Naprogramování a odladění takového nástroje však bývá velice komplikované a zdlouhavé.

Další možností je využití existujícího interpreteru. Oficiální implementace Pythonu *CPython* používá vlastní modul k interpretaci, Jde se o implementaci vytvořenou v jazyce C. Ta samozřejmě obsahuje parser za účelem spouštění skriptů. Nabízí se tak použití volně dostupné implementace, která je navíc odzkoušená a vždy se bude maximálně shodovat se současnou distribuční verzí Pythonu [13].

Při implementaci obfuskačních modulů musíme brát v úvahu také technická omezení, která pro parsery platí. Typicky bývají implementovány pomocí rekurzivního sestupu, využívajícího systémový zásobník, jehož hloubka bývá omezena. V případě Pythonu je maximální počet úrovní 1500. Pro případnou změnu je nutné překompilovat Python s vhodně nastavenou hodnotou [14].

[13] Kromě oficiální implementaci existují ještě další parsery a refaktorovací nástroje pro Python. Například: `https://github.com/PyCQA/baron` a `https://github.com/python-rope/rope`

[14] `https://github.com/python/cpython/blob/master/Parser/parser.h` — hlavičkový soubor parseru omezující maximální hloubku

Python je silně interoperabilní, což znamená, že je poměrně snadné jej propojit s jinými programovacími jazyky. Existuje oficiálně podporovaný modul `ast`, který umožňuje přímý přístup k výše popsanému parseru. Skripty v Pythonu tak mohou načítat kód, upravit jej a převést do syntaktického stromu.

Jeho výhodami jsou kromě oficiální implementace také dostupnost, stabilita a automatická aktualizace. Vždy se změnou gramatiky Pythonu tak nebude nutné programovat parser a měnit interní strukturu programu. Bude stačit implementovat námi navržené obfuskační metody pro nový typ uzlu v AST. Výhodou je také možnost psát obfuskátor v Pythonu, který je často subjektivně hodnocen jako velice rychlý prototypovací nástroj. Jako platformu pro tvorbu nástroje tedy zvolíme právě Python v současné verzi 3.6.

Nyní jsme díky modulu `ast` schopni vstupní program převést do abstraktního tvaru a provést úpravy, které jsme navrhli. Poslední částí obfuskace je převod AST zpět do zdrojového kódu v jazyce Python. Modul `ast` je určen pouze k tvorbě a modifikaci syntaktických stromů spolu s jejich kompilací a spuštěním přímo ve skriptech.

Neexistuje však oficiální způsob převodu zpět do zdrojového kódu. I zde existuje několik možností. Kromě vytvoření vlastního modulu pro převod můžeme volit mezi volně dostupnými implementacemi. Existujícími knihovnami jsou například `astdump`, `codegen`, `rope`, `redbaron` a `astunparse`.

Velkou část z nich tvoří víceúčelové nástroje pro modifikaci zdrojových kódů v Pythonu, které prochází pomalým vývojem a neposkytují podporu pro poslední verze. Jednoúčelovou knihovnou, která poskytuje požadované rozhraní a hodí se k našim účelům, je `astunparse`. Tato knihovna je navíc šířena pod stejnou licencí *PSF* jako Python [15], takže s jejím použitím by neměly být legální problémy.

3.2 Implementace řadiče

Nyní navážeme na analytickou a návrhovou část práce. Při výběru technologií jsme zvolili Python 3 jako implementační programovací jazyk, který nabízí rychlé prototypování a vhodné knihovny. Pro parsování zdrojových kódů použijeme vestavěný modul `ast`. Abychom zaručili kvalitu výstupu, k linearizaci upraveného stromu AST využijeme volně dostupný modul `astunparse`, který je distribuován pod permisivní licencí.

V analytické části jsme popsali princip, na kterém bude nástroj jako celek fungovat. Abychom dosáhli oddělení jednotlivých částí programu použijeme objektově orientované programování, které Python standardně podporuje. K zajištění modularity jsme oddělili řídící část od obfuskačních modulů. Hlavní část programu se tedy nachází v souboru *driver.py*, který zároveň slouží jako spouštěcí skript.

[15] `https://docs.python.org/3/license.html` — Python licence

Struktura programu odpovídá návrhu z návrhové části. Jednotlivé fáze obfuskačního algoritmu jsou následující:

1. Program ihned po spuštění načte argumenty příkazové řádky. Python v defaultní instalaci nabízí celou řadu užitečných modulů. Pro parsování argumentů se hodí vestavěný modul *optparse* implementující třídu `OptionParser`. Pomocí tohoto parseru jsme vytvořili rozhraní pro komunikaci s uživatelem.

   Postupně během implementace jsme doplnili tyto argumenty:

   `--input-folder` — cesta ke vstupní složce

   `--input-file` — cesta ke vstupnímu souboru

   `--output-folder` — cesta k výstupní složce [16]

   `--list-actions` — seznam implementovaných obfuskátorů

   `--action` — přidá zvolený obfuskátor, může se opakovat

   `--extended-info` — vypíše rozšířené informace o zvoleném obfuskátoru

   `--level` — úroveň obfuskací s rozsahem 1 až 5

   `--striping-length` — délka prokládání generovaných identifikátorů

   `--verbose` — nastaví detailní úroveň výpisu informací

   `--help` — program vypíše nápovědu a ukončí se

   Kromě vstupní a výstupní složky pro načítání, resp. ukládání zdrojových kódů, může uživatel určit pořadí aplikovaných obfuskačních modulů. Důležitým parametrem je `level`, tedy kvalita jednotlivých obfuskací. Jejich resilience, potence i cena je ovlivněna právě tímto parametrem a je dále popsána u každého modulu zvlášť. Lze také vypsat seznam všech obfuskátorů spolu s jejich detaily.

2. Program dále načte jednotlivé vstupní soubory a jejich obsah pomocí modulu `ast` rozparsuje. Výstupem jsou příslušné syntaktické stromy *AST*, které jsou uloženy pro pozdější zpracování.

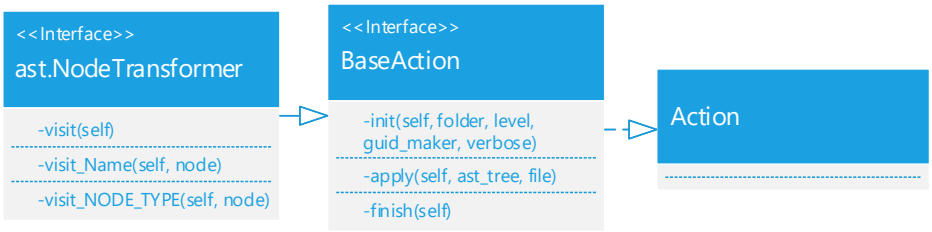
3. V uživatelem definovaném pořadí jsou spuštěny zvolené obfuskace. Každá z nich je sekvenčně spuštěna nad AST všech vstupních souborů. Tímto způsobem implementace zajišťuje konzistenci úprav.

4. Poslední fází běhu je linearizace modifikovaných stromů AST pomocí zvoleného modulu `astunparse`.

[16] Výstupní složka nemusí při spuštění existovat.

3.2.1 Interface pro obfuskaci

Abychom zajistili dostatečnou rozšiřitelnost programu, využíváme objektově orientované programování, které Python podporuje. Obfuskační moduly jsme tak navrhli jako třídy se společným rozhraním. Rozhraní je vynuceno využitím třídy `BaseAction`, od které ho všechny moduly podědí. Tato bázová třída vytváří rozhraní pro všechny implementované obfuskační moduly. Musí být tedy dostatečně široké a musí zahrnovat všechny společné požadavky 3.1 — viz. poznámka [17].



Obrázek 3.1: Rozhraní poskytované pro obfuskátory.

Aby byla umožněna modifikace zdrojových kódů, zvolili jsme bázovou třídu `BaseAction` třídu `NodeTransformer` z modulu `ast`. Díky tomu jsou obfuskační moduly potomky této třídy a mohou využít i jejího rozhraní. To umožňuje přetěžováním zvolených metod modifikaci vstupní strom AST. Třída `NodeTransformer` deklaruje metodu `visit`. Tato metoda spouští průchod předaným syntaktickým stromem způsobem preorder. Přetížené metody pro příslušné typy uzlů jsou pak automaticky volány a umožňují provedení libovolných úprav.

Rozhraní, které musí každý z obfuskačních modulů povinně implementovat obsahuje několik statických atributů, které jsou programem čteny a využívány k identifikaci modulu:

`name` — jméno modulu, pomocí něhož je modul zvolen na příkazové řádce

`author` — autor daného modulu

`date` — datum vytvoření modulu

`mode` — příznak, zda modul vyžaduje dvou-průchodovou aplikaci

`short_info` — popis shrnující v funkcionalitu a vlastnosti obfuskace

`detailed_info` — detailní a technický popis, včetně možných komplikací a varování. Zahrnuje také popis vlastností v závislosti na zvoleném parametru `level`

[17] Visit_NODE_TYPE je řada metod, které lze pro typy uzlů v AST přetížit

Nezbytná konstantní data platná během celé životnosti obfuskátoru jsou předávána přímo v konstruktoru. Ten má následující rozhraní:

```
__init__(self, folder, level, guid_maker, verbose)
```

`folder` — výstupní složka, do které budou uloženy obfuskované soubory

`level` — číselný parametr popisující úroveň zvolené obfuskace

`guid_maker` — instance pomocné třídy určené ke generování identifikátorů

`verbose` — logická hodnota rozlišující podrobnost výpisu informací

Kromě konstruktoru obsahuje požadované rozhraní ještě dvě metody — `apply` a `finish`. Metoda `apply` je volána pro každý vstupní soubor a má následující rozhraní:

```
apply(self, ast_tree, file)
```

`self` — instance obfuskátoru

`ast_tree` — syntaktický strom pro vstupní soubor, který je právě zpracováván

`file` — absolutní cesta k souboru, který je zpracováván

Metoda `finish` je volána pouze jednou pro každý objekt a to až po zpracování všech souborů. Jejím smyslem je umožnit dokončení některých modifikací. Má následující rozhraní:

```
finish(self)
```

`self` — instance obfuskátoru

Téměř všechny obfuskační moduly pro svou činnost potřebují generovat náhodné identifikátory. Abychom usnadnili jejich použití vytvořili jsme v souboru `utils.py` třídu `GuidMaker`, jejímž úkolem je právě generování identifikátorů. Vygenerované řetězce jsou složeny z dvojic znaků, které jsou si graficky podobné:

`OO`, `ll`, `Il`, `I1`

Je nutné generovat identifikátory náhodně tak, aby nedocházelo ke kolizím. Řešením narozeninového problému jsme spočítali minimální délku generovaného řetězce tak, aby pravděpodobnost kolize byla menší než 0.1% za předpokladu, že generujeme 3000 identifikátorů. Toto číslo jsme odhadli s dostatečnou rezervou pro typické použití. Výsledná délka je nastavena v konstruktoru třídy a lze ji případně změnit přímo v souboru `utils.py`.

Abychom mohli generovat identifikátory, bylo nutné zvolit zdroj náhodných dat. Po úvaze jsme vybrali modul `secrets`, který je v Pythonu vestavěný. Podle dokumentace se jedná o spolehlivý způsob získávání entropie, protože přímo využívá systémové zdroje [18].

3.2.2 Závěr

V této části jsme popsali implementaci hlavní část navrženého obfuskačního nástroje. Tento řídící modul poskytuje požadované rozhraní příkazové řádky pro uživatele. Dále nabízí rozhraní pro obfuskační moduly, jejichž implementaci popíšeme v dalším textu.

3.3 Implementace modulů

V této kapitole popíšeme námi implementované obfuskační moduly. Dále se pokusíme zhodnotit možná rizika plynoucí z jejich použití a odhadneme kvalitu pomocí definovaných veličin.

3.3.1 Odstranění komentářů

Pro odstraňování komentářů jsme vytvořili modul `CommentsRemover`, který je implementován v souboru `comments_remover.py`. Implementace odpovídá popsanému algoritmu z návrhové části.

Komentáře se mohou vyskytovat jako uzly v tělech některých objektů. Jde o `Module`, `AsyncFunctionDef`, `FunctionDef` a `ClassDef`. Přetížili jsme tedy příslušné metody s prefixem `visit_` z bázové třídy `ast.NodeTransformer`.

V těchto metodách je spuštěn rekurzivní průchod pro výrazy jejich těla. Postupně filtrujeme výrazy 3.1, které odpovídají řetězcům stojícím samostatně a dokumentačním komentářům:

- dokumentační komentář — řetězec vnořený ve výrazu — `ast.Expr(value = ast.Str())`
- samostatně stojící řetězec — `ast.Str()`

Klasické komentáře nejsou parsovány, proto se do AST nedostanou a vůbec se jimi nemusíme zabývat. Jako filtrovací mechanismus jsme využili lamdba výraz.

```
sieve = lambda c: isinstance(c, ast.Expr) and
   isinstance(c.value, ast.Str)
if level >= 1:
  sieve = lambda c: (isinstance(c, ast.Expr) and
     isinstance(c.value, ast.Str)) or isinstance(c,
     ast.Str)
```

Listing 3.1: Ukázka kódu nastavující filtr.

Modul zároveň poskytuje dvě úrovně obfuskace:

- v první odstraní pouze dokumentační komentáře

- v té druhé navíc smaže i nepoužité volně se vyskytující řetězce

Zhodnocení kvality:

- resilience — protože se jedná o jednosměrnou modifikaci je resilience velmi vysoká, provedené úpravy již nelze žádným způsobem vrátit zpět.
- potence — úroveň potence hodnotíme jako střední v závislosti na kvantitě a kvalitě původní dokumentace zdrojového kódu.
- cena — tato obfuskace je specifická v tom, že rychlost nepatrně vzroste. To je způsobeno tím, že parser nemusí zpracovávat nadbytečné řetězce. Cena je tedy nulová.

3.3.2 Relokace řetězců

Tento modul jsme implementovali jako třídu `StringsRelocator` v souboru `strings_relocator.py`. Jeho úkolem je přemístění řetězců z výrazů na počátek nového souboru a jejich nahrazení vytvořenými proměnnými. Implementace proběhla dle návrhu.

Vytvořili jsme pomocnou třídu `StringsGetter` 3.2. Jejím úkolem je nashromáždit řetězce ze všech souborů a zároveň k nim vygenerovat unikátní identifikátory. Po spuštění metody `apply` je vytvořena instance této třídy, který vyhledá v daném AST všechny řetězce a vrátí je jako slovník řetězců a vygenerovaných identifikátorů.

Nalezené řetězce jsou uloženy do třídní proměnné. Následně je spuštěna obfuskace na kořeni celého stromu. Obfuskace probíhá na úrovni modulu a rozpadá se dle nastavené úrovně na dva případy.

Buď jsou řetězce relokovány do nového souboru v rámci volání metody `finish`, pak je daný soubor načten na začátku modulu, nebo je na počátek jednotlivých modulů přidán celý seznam proměnných. Do těchto proměnných jsou zároveň uloženy příslušné řetězce.

Poslední fází je rekurzivní záměna původních řetězců za proměnné. Při průchodu uzlem reprezentujícím řetězec, je nahrazen načtením hodnoty z příslušné proměnné.

```
class StringsGetter(ast.NodeVisitor):
  def __init__(self):
    self.strings = {}

  def find_strings(self, ast_tree):
    self.visit(ast_tree)
    return self.strings

  def visit_Str(self, node):
```

```
self.strings[node.s] = get_guid()
```

Listing 3.2: Třída shromažďující všechny řetězce — zkráceno.

Zhodnocení kvality:

- resilience — provedené úpravy jsou reverzibilní, bylo by možné obrátit postup a řetězce dosazovat na původní místa. Resilience je tedy nízká.
- potence — úroveň potence je střední až vysoká, řetězce mohou typicky sloužit jako záchytné body při analýze kódu a jejich význam pro pochopení je značný.
- cena — cena je nízká. Přesto se může stát, že neoptimalizující prostředí bude zpomaleno úrovní indirekce, kterou náš mechanismus přidá. Při praktickém měření se však neprojevil pokles.

Relokace řetězců je jednoduchou obfuskací, která má potenciál znesnadnit analýzu a pochopení kódu. Přesto, že její resilience je nízká, její použití doporučujeme zejména s ohledem na nízkou cenu. Tento typ modifikace je vhodný ke kombinaci s ostatními moduly. Výhodné je například dřívější odstranění komentářů, které by byly zbytečně přiřazeny do proměnných přesto, že by nebyly použity.

3.3.3 Relokace numerických konstant

Na stejném principu, jaký využíval modul pro přemístění řetězců, je založen i tento obfuskátor. Implementace proběhla v souboru `constants_relocator.py` do třídy `ConstantsRelocator`. Víme, že přesun literálů nemá vliv na funkcionalitu a případnou chybu matematických operací. Obfuskace spočívá v přemístění vhodných literálů do nového souboru a jejich přiřazením do proměnných.

Algoritmus je jednoprůchodový a začíná spuštěním metody `apply` pro každý vstupní soubor. Je vytvořena instance třídy `ConstantsGetter`, která rekurzivně v modulu vyhledá všechny číselné konstanty. Při vyhledávání jsme přetížili metodu `visit_Num` a příslušná čísla ukládáme spolu s vygenerovanými identifikátory.

Tyto uložené identifikátory použijeme při zavolání metody `finish`. Ta vytvoří nový soubor a vloží do něj veškeré uložené proměnné a přiřadí do nich příslušná čísla. Zároveň jsme přetížili metodu `visit_num`, v níž jsou nalezené konstanty nahrazeny načtením z příslušných proměnných. Na počátek modulu je přidán odkaz na vytvořený soubor, abychom se mohli odkazovat na doplněné proměnné.

```
def apply(self, ast_tree, file):
  if level == 0:
    constants = {}
  for constant in find_constants(ast_tree):
```

```
    if const not in constants:
      constants[const] = new_constants[const]
  visit(ast_tree)
  return file
```

Listing 3.3: Metoda apply — zkráceno.

Dále jsme implementovali zjednodušenou variantu, která operuje na úrovni souborů místo celého projektu. Uložené konstanty jsou rovnou vkládány na počátek modulu a není tedy nutné vytvářet nový soubor.
Zhodnocení kvality:

- resilience — provedené úpravy jsou reverzibilní, obrácením postupu lze čísla dosazovat na původní místa. Resilience je tedy nízká.
- potence — úroveň potence hodnotíme jako střední, nahrazení čísel identifikátory znepřehlední kód a vyžaduje dodatečnou práci při analýze.
- cena — cena je nízká, přesto se může stát, že běh bude ovlivněn přidanou úrovní indirekce. Při měření byl však rozdíl časů zanedbatelný.

Přemístění číselných konstant je jednoduchá obfuskační technika, kterou s výhodou můžeme použít zejména díky nízké ceně. Vlastnosti jsou díky totožnému algoritmu shodné s předchozím modulem pro přemístění řetězců.

3.3.4 Rozložení numerických konstant

Algebraické vlastnosti čísel nám umožňují rozkládat číselné výrazy na podvýrazy tak, že výsledná hodnota je zachována. Výrazy jsou pak komplikovanější a jejich hodnota není na první pohled odhadnutelná. Tyto modifikace implementuje modul ve třídě `ConstantsScrambler` souboru `constants_scrambler.py`.

Jak jsme popsali v návrhové části, z této modifikace je nutné vyjmout desetinná a komplexní čísla. V opačném případě hrozí ztráta přesnosti. Algoritmus byl implementován podle návrhu. Metoda `apply` spustí prohledávání stromu. V případě nalezení celočíselné konstanty je vygenerován výraz, jehož hodnota je shodná s původní.

Generování výrazu probíhá rekurzivně a využívá binárních operací, které Python nativně nabízí. V každé úrovni rekurze je náhodně vybrána binární operace a vygenerován jeden z operandů. Druhým operandem je strom vrácený z rekurzivního volání. Abychom omezili velikost výrazu, je hloubka rekurze parametrizována.

Při dosažení nejnižší úrovně, je automaticky vrácen strom reprezentující náhodné číslo. Rozdíl mezi hodnotou výsledného stromu a požadované konstanty je korigován na nejvyšší úrovni. Abychom znali hodnotu, kterou strom reprezentuje, musí rekurze vracet navíc ještě číslo.

Další podporovanou operací je rozštěpení stromu. Část z něj je uložena do proměnné, která je vložena na začátek modulu. Odstraněná část je nahrazena právě touto proměnnou. Výsledkem je další zkomplikování výrazu.

```
def generate_int(num, depth):
  #List stromu
  if depth == max_depth:
    n = random.randint(val_min, val_max)
    return ast.Num(n=n), n

  tree, val = generate_int(num, depth+1)
  #Rozdeleni stromu
  if random.randint(0, 4) == 0:
    variables.append((get_guid(), tree))
    tree = ast.Name(id=name, ctx=ast.Load())

  #Oprava hodnoty
  if depth == 0:
    ...
  n = randint(val_min, val_max)
  op = randint(1, max_ops)
  #Vygenerovani binarni operace
  if op == 1:
    return ast.BinOp(left=tree, op=ast.Add(), right=
       ast.Num(n=n)),val+n
  ...
```

Listing 3.4: Metoda pro generování stromu — zkráceno.

Podporovanými binárními operacemi jsou:

`Add` — sčítání

`Sub` — odčítání

`LShift` — bitový posun vlevo

`RShift` — bitový posun vpravo

`BitOr` — bitový `OR`

`BitAnd` — bitový `AND`

`BitXor` — bitový `XOR`

`Mod` — zbytek po dělení

V případě zbytku po dělení `Mod` je nutné ošetřit případ, kdy se dělí nulou.

Kromě základního algoritmu jsme implementovali i parametrizaci. Pomocí nastavené úrovně je tak ovlivněna maximální hloubka rekurze, rozsahy generovaných čísel i binární operace, které jsou aplikovány. Podrobnosti jsou uvedeny v nápovědě k modulu, čí přímo v komentářích zdrojového kódu.
Zhodnocení kvality:

- resilience — střední až vysoká, provedené úpravy jsou reverzibilní, ale jejich odstranění vyžaduje značnou snahu a dynamické vyhodnocování výrazů.
- potence — úroveň potence je střední až vysoká, na pohled je výsledné číslo neodhadnutelné a je nutné výrazy ručně či automaticky vyhodnocovat.
- cena — cena závisí na rychlosti zvolených operací a zejména na hloubce vygenerovaných stromů. Platí, že s rostoucí hloubkou roste počet operací a tedy i cena.

Rozklad konstant je technika vhodná pro zakrývání hodnot čísel. Za vyšší potenci je ale nutné zaplatit vyšší cenou, kterou však lze parametrizovat. Pokud bychom chtěli cenu snížit, je možné na počátku relokovat konstanty na začátek souboru a přiřadit do proměnných. Strom pro každou z nich pak bude při spuštění vyhodnocen pouze jednou a v původních výrazech budou použity proměnné.

3.3.5 Změna notace atributů

Python podporuje objektový model programování a spolu s ním přístup k potřebných atributům. Existují dvě možnosti přístupu k atributu. Buď pomocí tečky oddělující zdrojový objekt od jména atributu (tečkové notace), nebo pomocí funkcí `getattr` a `setattr` [12].

Tento modul cíleně nahrazuje tečkovou notaci pomocí zmíněných funkcí. Výhodou je zesložitění výsledných výrazů a změna jména atributu z výrazu na řetězec. Na něj je poté možné aplikovat další obfuskace.

Modifikaci jsme implementovali v souboru `attributes_notation.py` jako třídu `AttributesNotation`. Samotný algoritmus odpovídá popisu z návrhové části. Metoda `apply` spustí průchod stromem AST. Při nalezení uzlů typu `attribut` je provedena jejich záměna za volání funkce `getattr`. Druhý případ je přiřazení — `assignment`, kdy naopak nahrazujeme funkcí `setattr`.

```
a.x, a.y = 1, 2
```

Listing 3.5: Ukázka nepřeveditelného výrazu.

Při nahrazování funkcí `setattr` je nutné zkontrolovat, zda přiřazujeme pouze do jednoho uzlu. Python totiž umožňuje vícenásobné přiřazování. Toto

chování však není možné převést na volání funkce `setattr` a proto se mu musíme vyhnout 3.5.

```
def visit_Assign(self, node):
node.value = self.visit(node.value)

#Pouze pri vyssi urovni obfuskace a jednoducha
   prirazeni
if self.level >= 1 and len(node.targets) == 1:
target = node.targets[0]
...
return node
```

Listing 3.6: Implementace obfuskace přiřazení do proměnné.

Do modul jsme v základní verzi implementovali nahrazení čtení atributu. Až při vyšším nastavení kvality obfuskace je provedeno i nahrazení za funkci `setattr`.
Zhodnocení kvality:

- resilience — modifikace je obousměrná, stačilo by obrátit běh algoritmu, resilience je z toho důvodů nízká
- potence — potence je nízká, protože informace zůstává na stejném místě, pouze změní tvar
- cena — cena je nízká, přesný čas záleží na implementaci a úrovni optimalizace interpreteru, dokumentace uvádí zpomalení cca 1% [13]

Změna notace pro přístup k atributům je výhodná zejména v kombinaci o ostatními modifikace. Informaci o tom, ke kterému atributu je přistupováno převádíme na řetězec. Na ten se vztahují ostatní obfuskační techniky, čehož můžeme využít. Za nízkou cenu, tak získáme možnost provést kvalitní obfuskaci.

3.3.6 Vkládání predikátů

Obfuskace vkládáním konstantních predikátů má za cíl celkové znepřehlednění kódů. Modul byl implementován jako třída `OpaquePredicatesInjector` v souboru `opaque_predicates_injector.py`.

Algoritmus začíná voláním metody `apply`, ta spustí průchod stromem. Pro uzly modulů, funkcí a tříd, podmíněných výrazů a další typů jsme implementovali příslušné metody pro průchod. Pro každý takový nalezený uzel zavoláme námi vytvořenou funkci `insert_predicates` 3.7. Ta operuje na celém těle, tedy seznamu výrazů.

Abychom mohli ovlivnit počet injektovaných predikátů, zavedli jsme míru vložení jako počet vložených podmínek na sto výrazů. Tato míra je volitelná

přímo jako atribut modulu, abychom zvýšili náhodu je navíc hodnota randomizovaná v rozsahu (80%, 120%).

Před samotným vkládáním podmínek určíme pro každé nalezené tělo počet predikátů. Poté náhodně rozdělíme seznam výrazů na tři části a prostřední z nich vložíme do nové podmínky. Musíme brát v úvahu krajní případy rozdělení. Důležitější než samotný algoritmus jsou predikáty a jejich vkládané tvary. Python podporuje několik typů podmínek a ty lze navíc používat v pozitivním i negativním tvaru.

```
def insert_predicates(self, body):
  rnd = uniform(0.8, 1.2)
  count = int(rnd*preds_per_100_lines*len(body))//100
  for x in range(count):
    start = randint(0, len(body)-1)
    end = randint(start, len(body)-1)
    if start == end:
      continue
    #Nahodny vyber predikatu
    pred = ....
    body = body[:start] + [pred] + body[end:]
  return body
```

Listing 3.7: Implementace obfuskace přiřazení do proměnné.

Kromě použitých syntaktických konstrukcí jsou důležité samotné konstantní podmínky. V základní verzi jsme implementovali několik randomizovaných možností. Kromě srovnávání náhodných čísel a řetězců jsme vytvořili tvary specifické pro Python.

Například je možné testovat přítomnosti náhodných atributů v rozličných objektech. Další tvary se zaměřují na používání vestavěné funkce `__eq__` určené pro porovnávání různých vestavěných typů. Mezi všemi těmito kombinacemi je vybíráno pro každý predikát náhodně pomocí uniformního rozdělení a systémového generátoru náhodných čísel. Třída je navržena tak, aby bylo snadné doplnit další predikáty a případně rozšířit možnosti.
Zhodnocení kvality:

- resilience — resilience je střední až vysoká, silně závisí na implementaci a kvalitě výrazů, které se v podmínkách generují, teoreticky je lze detekovat a odstranit
- potence — potence je střední, reverzní inženýr musí věnovat dodatečné úsilí, aby mohl sledovat logický tok programu
- cena — cena se odvíjí od počtu vložených výrazů, který můžeme nastavit

Vkládání konstantních podmínek je modifikace sloužící k rozptýlení pozornosti reverzního inženýra. Výhodou je, že cenu i potenci můžeme snadno parametricky ovlivnit.

3.3.7 Přejmenování argumentů

Jednou z možností jak znepřehlednit zdrojový kód je přejmenování identifikátorů. To je však v dynamickém jazyce komplikované a může vést k problémům při použití funkcí `eval` a `exec`. Abychom minimalizovali rizika, přejmenujeme pouze funkční argumenty. Tato technika je implementována třídou `ArgumentsRenamer` v souboru `arguments_renamer.py`.

Princip přejmenování je shodný s popisem v návrhové části. Najdeme definici funkce a přejmenujeme její parametry. Dále projedeme její tělo a přejmenujeme parametry vyskytující se ve výrazech. Výhodou je, že Python neumožňuje deklaraci lokální proměnné, jejíž jméno se shoduje s nějakým parametrem. Toho chování tedy nemusíme ošetřovat.

Python podporuje definici funkcí uvnitř jiných funkcí do hloubky omezené parserem. Z vnitřních definic se lze přímo odkazovat na proměnné vnějších kontextů, totéž platí i pro parametry. Při implementaci je tak nutné udržovat kontext, ve kterém byl argument vytvořen.

Třída reprezentující kontext, je implementována ve stejném souboru a umožňuje uložení jména proměnné spolu s novým identifikátorem. Každá instance může být součástí hierarchie a mít nastaveného rodiče. Při vyhledávání nového jména proměnné je prohledána celá hierarchie. Tímto přístupem jsme zajistili správné přejmenování parametrů vnořených funkcí, jejichž jména kolidují.

Samotné přejmenování začíná opět průchodem AST. Při nalezení funkční definice je vytvořen nový kontext, jehož rodičem je kontext původní. Následuje přejmenování argumentů a přidání jejich jmen do kontextu. Dále je na tělo funkce aplikována rekurzivní funkce. Při průchodu uzlem typu `Name`, který reprezentuje proměnnou, je dané jména nalezeno v kontextu a přejmenováno.

```
class Context(object):
  def __init__(self, parent=None):
    self.parent = parent
    self.vars = {}

  def set_variable(self, name, val=None):
    if val is None:
      val = self.guid_maker.get_guid()
    self.vars[name] = val

  def find_variable(self, name):
    if name in self.vars:
      return self
    elif self.parent is not None:
      return self.parent.find_variable(name)
    return None
```

```
def get_variable(self, name):
  ctx = self.find_variable(name)
  if ctx is not None:
    return ctx.vars[name]
  return None
```

Listing 3.8: Třída pro uchování kontextu — zkráceno.

Zhodnocení kvality:

- resilience — přejmenování je jednosměrné, resilience tudíž velmi vysoká
- potence — potence je střední až vysoká, záleží na struktuře programu, počtu argumentů a jejich používání
- cena — cena je nulová, doba běhu výsledného programu se téměř nezmění

Přejmenování argumentů je výkonnostně nenáročná obfuskace, která pro funkce a metody s velkým počtem parametrů může mít velký dopad. Protože navíc nemá žádný vliv na rychlost, lze ji pro použití doporučit.

3.3.8 Převod definic na AST

Python umožňuje pomocí funkcí `exec` a `eval` dynamické vykonávání kódu včetně definování funkcí i celých tříd. Existují dva postupy, jak tuto modifikaci provést. První možností je uložení dané části zdrojového kódu jako řetězce a následná kompilace voláním funkce `compile`. Tím je vytvořen bytekód, který je dále předán funkci `exec` a přímo vykonán.

Druhou možností je vytvoření přímo stromu AST pomocí příslušných objektů z modulu `ast` a následná kompilace funkcí `compile`. Při implementaci jsme zvolili druhou z možností, protože poskytuje vyšší úroveň potence i resilience. Navíc na ni lze snadno aplikovat další obfuskační techniky.

Algoritmus jsme implementovali jako třídu `DefinitionsToExec` v souboru `definitions_to_exec.py`. Po spuštění jsou nalezeny globální definice funkcí a tříd. Celé tyto podstromy jsou odstraněny z hlavního stromu. Na začátku modulu jsou pak přiřazeny do proměnných a navíc je jim předřazena funkce `fix_missing_locations` z modulu `ast`, která je nezbytná pro kompilaci, protože za běhu doplní čísla řádků 3.9.

```
def visit_Module(self, node):
  for i, child in enumerate(node.body):
    node.body[i] = self.visit(child)

  for name, tree in self.sources:
    parsed = ast.parse(name + ' =
       fix_missing_locations(' + tree + ')')
```

```
    node.body.insert(0, parsed.body[0])
  node.body.insert(0, ast.ImportFrom(module='ast',
     names=[ast.alias(name='*')]))
  return node
```

Listing 3.9: Metoda pro obfuskaci modulu — zkráceno.

Do míst, kde se původně tyto definice nacházely, program doplní volání funkcí `exec` a `compile`. Tím se dosáhne stejného efektu, jako kdyby byly v původních místech přímo definice. Aby bylo možné využívat metod z modulu `ast` je ho nutné přidat do všech změněných souborů.
Zhodnocení kvality:

- resilience — převod by bylo možné obrátit pomocí zpětné kompilace a dosazením na správná místa, resilienci hodnotíme jako střední až vysokou
- potence — potence je střední až vysoká, celkové znepřehlednění situace je značné
- cena — cena je nízká, přidaná režie souvisí pouze s voláním funkce `exec` a obsluhou kompilace, která se ovšem provádí pouze jednou. Možné zpomalení programu v důsledku nefunkčního cachování kompilovaných částí se nepodařilo potvrdit.

Převod části kódu na AST je účinná obfuskace, která projeví své kvality zejména v kombinaci s dalšími moduly. Například přesunutí nebo šifrování řetězců z uloženého stromu odstraní důležitá data a znepřehlední situaci.

3.3.9 Sloučení funkcí

Sloučení několika funkcí je metoda spočívající ve vytvoření nové funkce spojením několika původních. V závislosti na zvoleném parametru musí být možné volat libovolnou z původních funkcí. Při implementaci jsme museli vyřešit několik komplikací, které nyní popíšeme a uvedeme případná rizika plynoucí z použití.

Implementace modulu se nachází v souboru `functions_merger.py` jako třída `FunctionsMerger`. V metodě `apply` je spuštěn průchod stromem. V případě nalezení několika funkcí na totožné úrovni je zkontrolováno, zda nedochází k předefinování již existující funkce. Aplikace této metody by v takovém případě mohla vrátit špatné výsledky. Dále je ošetřen případ, kdy těla obsahují volání funkcí `exec` a `eval`, o toto se stará třída `IsSafeChecker`. Obsah vykonávané části nelze staticky rozpoznat. Možným opatřením by bylo injektovat předefinovanou funkci `exec` a `eval` na počátek souboru tak, aby byl uživatel varován.

Pokud na dané úrovni není nalezen problém, následuje samotné slučování funkcí. Jedním z problémů, které bylo nutné vyřešit, je předávání argumentů

nové funkci, protože jejich počty se nutně budou lišit. Parametry mohou být několika typů:

- klasické
- variabilní
- klasické s defaultní hodnotou
- variabilní s klíčovým slovem

Klasické argumenty jsou určeny svým jménem — místo toho, abychom předávali všechny postupně, vytvoříme zástupný objekt. Tento objekt bude mít atributy s hodnotami, které odpovídají příslušným argumentům. Jako vhodný objekt jsme zvolili lokálně definovanou lambda funkci, která může mít libovolný počet dynamických atributů.

Jednotlivá těla funkcí je také nutné projít a nahradit reference původních parametrů na nové odkazované pomocí zástupného objektu. V obfuskátoru je toho přejmenování reprezentováno třídou `ArgumentsReplacer`. Ta navíc automaticky přejmenovává odkazované parametry pomocí náhodných identifikátorů.

Variabilní argumenty jsou reprezentovány v jednom případě pomocí pole a v druhém pomocí slovníku. Tyto dvě proměnné tedy stačí předat do nové funkce a změnit názvy. Zároveň je nutné sjednotit jejich jména v jednotlivých tělech. Toto přejmenování implementuje třída `ArgumentsRenamer`.

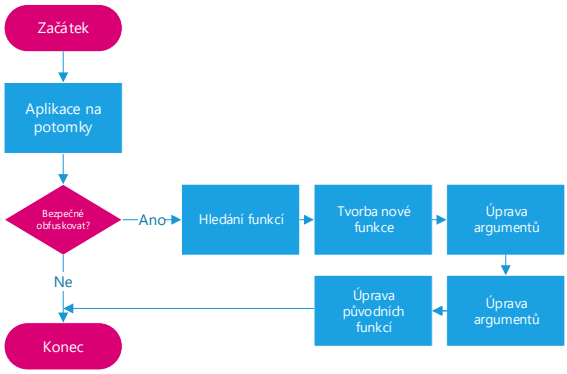
Výsledná funkce tedy má celkem čtyři parametry. První určuje zvolenou funkci, druhý reprezentuje klasické atributy a zbylé dva dynamické. Její tělo je složeno z podmínek, které testují vybranou funkci a příslušná těla funkcí.

Aby bylo možné správně volat nově vytvořenou funkci za každé situace, musíme upravit původní definice funkcí. Jejich těla algoritmus odstraní a nahradí je rutinou pro vytvoření proxy objektu s argumenty a voláním nově vytvořené funkce. Pokud by našim cílem bylo volat novou funkci přímo, museli bychom upravit každé funkční volání. To ale není obecně možné — např. při použití funkce `exec` a `eval`. Nutné bylo také ošetřit vrácení příslušné návratové hodnoty. Implementace tohoto modulu je velmi obsáhlá, pro přehlednost proto uvádíme pouze samotné schéma průběhu 3.2.

Při implementaci jsme narazili na problém, kdy volání některých převedených funkcí vracelo výjimky. Příčinou se ukázala být vlastnost Pythonu, kdy se každá funkce obsahující klíčové slovo `yield` stává generátorem.

Generátor se při volání chová odlišně než funkce. Pokud tedy sloučená funkce obsahovala tuto syntaxi, nebylo možné korektně volat funkce, které naopak toho slovo neobsahovaly. Abychom tento problém obešli, jsou jednotlivá těla funkcí automaticky obalena do lokálních funkčních definic a následně ihned zavolána. Tyto lokální funkce se tedy mohou stát generátory, ale vlastnost *být generátorem* se nepřenáší na námi vygenerovanou hlavní funkci [18].

[18] `https://docs.python.org/3/whatsnew/3.3.html` — Novinky v Pythonu verze 3.3



Obrázek 3.2: Schéma průběhu slučování funkcí

Zhodnocení kvality:

- resilience — resilience je vysoká, obfuskace přidá do programu několik úrovní indirekce a volání nových funkcí a zejména to, že tato modifikace je aplikována rekurzivně na všech úrovních, z ní dělá velmi odolnou vůči zpětné opravě.
- potence — kvalita vygenerovaného kódu je vysoká, původní informace o volání funkcí je silně změněna
- cena — cena je střední, pro každé volání funkce přidává dvě další volání spolu s vyhodnocením podmínky

Sloučení funkcí je silná obfuskace, která může být velmi účinná. Její použití s sebou však nese riziko. Hrozí, že vygenerovaný kód bude za určitých podmínek fungovat špatně nebo vůbec. Proto je nutné upozornit uživatele obfuskátoru na možné problémy a zároveň na výhody, které poskytuje.

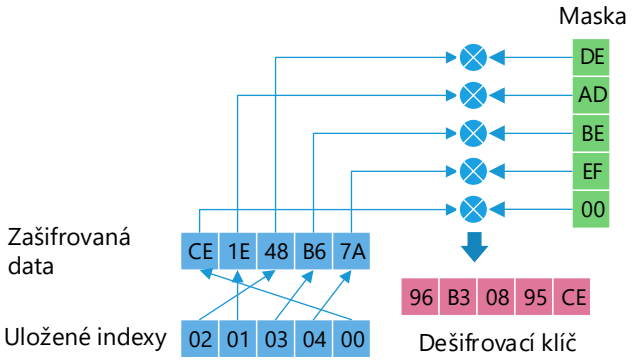
3.3.10 Šifrování řetězců

Modul pro šifrování souborů se jmenuje `StringsEncryptor` a je implementován v souboru `strings_encryptor.py`. Jeho hlavním cílem je sesbírání všech řetězců v modulu a jejich následné zašifrování odvozeným klíčem. Dále zajišťuje vložení dešifrovacího kódu do příslušných míst.

Abychom minimalizovali závislost na externím kódu, zvolili jsme univerzální proudovou šifru RC4. Ta umožňuje generování libovolně dlouhého klíče, kterým jsou data šifrována pomocí operace `xor`. Operace šifrování i dešifrování probíhá tedy shodně.

Po zavolání metody `apply` je postupně procházen strom AST a jsou nalezeny všechny řetězce. Každý takto nalezený řetězec je nahrazen funkcí, která po zavolání zajistí dešifrování správné jeho části. Řetězce uložené v poli jsou totiž sloučeny do jediného bytového pole a je tedy nutné znát počátek a konec.

Podle délky dat je vygenerován náhodný vstupní klíč. Ten je uložen a zároveň vstupuje do algoritmu RC4, který vytvoří šifrovací klíč. Data jsou pomocí šifrovacího klíče zašifrována.



Obrázek 3.3: Schéma odvození dešifrovacího klíče

Abychom byli schopni po spuštění programu dešifrovat uložené řetězce, je nutné odvodit nějakým způsobem klíč. Možnosti závisející na struktuře souboru a jeho vlastnostech jsme zavrhli, protože lze očekávat, že budou aplikovány další modifikující obfuskace. Přímé uložení klíče také postrádá smysl. Nakonec jsme zvolili nepřímé odvození klíče 3.3.

Do cílového modulu musíme vložit zašifrovaná data, použijeme je i k zakrytí klíče. Požadovaný klíč odvodíme pomocí zašifrovaných dat a pomocných hodnot. Pro každý byte klíče vygenerujeme náhodný index do pole dat. Na hodnotu uloženou v poli na tomto indexu aplikujeme operaci `xor` spolu s příslušným bytem dešifrovacího klíče. Dostaneme hodnotu masky. Tu spolu s indexy uložíme do dvou polí ve výsledném souboru.

Ze znalosti zašifrovaných dat, masky a příslušných indexů můžeme odvodit klíč a pomocí algoritmu RC4 dešifrovat data. Aby tento postup byl možný, je nutné injektovat do každého souboru se zašifrovanými řetězci dešifrovací funkce. Jejich jména i struktura je částečně randomizovaná, abychom zkomplikovali zpětnou analýzu. Tyto funkce implementují algoritmus RC4 spolu s odvozením klíčů. Přidali jsme navíc mechanizmus, který umožňuje cachování řetězce, aby při každém volání nedocházelo ke kompletnímu dešifrování. Zhodnocení kvality:

- resilience — resilience je střední až vysoká, námi zvolený způsob uložení klíče a randomizovaná struktura komplikuje automatickou analýzu
- potence — vysoká, zašifrování řetězců je silnou modifikací, které z čitelných řetězců vytvoří proud bytů, který na první pohled nepřináší nové informace

- cena — cena je v čase proměnlivá, pro každý řetězec je nutné zavolat dešifrovací funkci, v případě cachování je ale dopad na rychlost nižší až zanedbatelný

Šifrování řetězců se může vhodně doplňovat s relokací. Přiřazením hodnoty do proměnné je totiž výsledek uložen do proměnné a pro jeho použití není nutné volat žádnou další funkci.

3.4 Zhodnocení kvality

V této části otestujeme námi implementované metody na ukázkových zdrojových kódech v závislosti na zvolené kvalitě obfuskace. Dále se pokusíme stanovit kvalitu provedených modifikací. Pro každou metodu vytvoříme zdrojový program tak, aby byla obfuskace použitelná. Výsledné hodnocení je tak částečně zaujaté ve srovnání s použitím na reálných programech.

Zde uvedené zhodnocení kvality je subjektivní zejména výběrem zdrojového kódu. Námi vybrané kódy bylo tak vždy možné zvolenou obfuskační technikou modifikovat. To v reálném programu nemusí být vždy možné. V materiálech je proto přiložen volně šiřitelný projekt `python-poloniex`[19], na který byly postupně aplikovány různé obfuskace. Čtenář textu může vycházet ze srovnání mezi výchozím stavem a výstupy z nástroje. Sám tak může zhodnotit přínos jednotlivých modulů obfuskátoru pro svůj konkrétní projekt.

Jak jsme popsali již v průběhu implementace, velký potenciál je zejména v kombinaci různých technik a jejich několikanásobném použití. Zhodnocení víceúrovňové obfuskace by bylo časově velmi náročné, zaměříme se tedy na aplikaci samostatných modulů.

3.4.1 Odstranění komentářů

```
'''Definice funkce'''
def function(a, b):
  'Nepouzity retezec'
  # returns sum of args
  return a + b
```

Listing 3.10: Zdrojový kód před aplikací obfuskace.

```
def function(a, b):
  'Nepouzity retezec'
  return (a + b)
```

Listing 3.11: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0*.

[19] `https://github.com/s4w3d0ff/python-poloniex` — Repozitář testovaného projektu.

```
def function(a, b):
  return (a + b)
```

Listing 3.12: Zdrojový kód po aplikaci obfuskace s parametrem *level: 1* a více.

Zhodnocení:

- Resilience je vysoká v obou případech, protože se jedná o jednosměrné úpravy.

- Potenci hodnotíme jako střední. Ve variantě s vyšší úrovní 3.12 potence mírně stoupne, protože jsou odstraněny další informace ze zdrojových kódů.

3.4.2 Relokace řetězců

```
'''Definice funkce'''
  def function(a, b = 'Hodnota'):
  return a + b
```

Listing 3.13: Zdrojový kód před aplikací obfuskace.

```
lllllIlllIIIIIlI = 'Hodnota'
lIIlIIIlIIIIIIl1 = 'Definice funkce'
lIIlIIIlIIIIIIl

def function(a, b=lllllIlllIIIIIlI):
  return (a + b)
```

Listing 3.14: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0*.

```
from OOOOOOOO import *
OOOOOOOO

def function(a, b=OOOOOOOO):
  return (a + b)
```

Listing 3.15: Zdrojový kód po aplikaci obfuskace s parametrem *level: 1* a více.

Zhodnocení:

- Resilience je v prvním případě 3.14 nízká, induktivně by bylo možné dosadit proměnné na původní místa. V druhém případě 3.15 je situace zkomplikována mezimodulární závislostí.

- Potenci hodnotíme jako střední, ve variantě s vyšší úrovní 3.15 potence stoupne, protože jsou řetězce přesunuty do jiného modulu.

3.4.3 Relokace numerických konstant

```
def function(a=10):
  return a + 5
```

Listing 3.16: Zdrojový kód před aplikací obfuskace.

```
l1111111111 = 5
l1111111111 = 10

def function(a=l1111111111):
  return (a + l1111111111)
```

Listing 3.17: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0*.

```
from OOOOOOOOOOOOOOO import *

def function(a=OOOOOOOOOOOOOOO):
  return (a + OOOOOOOOOOOOOOO)
```

Listing 3.18: Zdrojový kód po aplikaci obfuskace s parametrem *level: 1* a více.

Zhodnocení:

- Resilience je v prvním případě 3.17 nízká, induktivně by bylo možné dosadit proměnné na původní místa. V druhém případě 3.18 je situace zkomplikována mezimodulární závislostí.
- Potenci hodnotíme jako střední, ve variantě s vyšší úrovní 3.18 potence stoupne, protože jsou konstanty přesunuty do jiného modulu.

3.4.4 Rozložení numerických konstant

```
def function(a=10):
  return a + 5
```

Listing 3.19: Zdrojový kód před aplikací obfuskace.

```
def function(a=(101 + (43 + -134))):
  return (a + ((131 + 128) - 254))
```

Listing 3.20: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0*.

```
llllllll = 9
lllllll = ((llllllll >> 1) >> 1)

def function(a=(((188 >> 1) >> 1) - 37)):
  return (a + (3 + lllllll))
```

Listing 3.21: Zdrojový kód po aplikaci obfuskace s parametrem *level: 1*.

```
OOOOOOOOOOOOOOO = (-371 - -437)
OOOOOOOOOOOOOOO = ((OOOOOOOOOOOOOOO & 226) - 209)

def function(a=(153 + OOOOOOOOOOOOOOO)):
  return (a + (7 + (((-141 - -135) >> 1) >> 1)))
```

Listing 3.22: Zdrojový kód po aplikaci obfuskace s parametrem *level: 2*.

```
lllIlIIIIlIIIIIlI = 324
lIIIIIIIIIIIIIII = (((lllIlIIIIlIIIIIlI << 1) ^ 697)
    << 1)
lllIIIIlIIIIIlIlI = (lIIIIIIIIIIIIIII - 683)

def function(a=(595 + lllIIIIlIIIIlIlII)):
  return (a + (2021 + ((((-513 >> 1) << 1) & -496) <<
      1)))
```

Listing 3.23: Zdrojový kód po aplikaci obfuskace s parametrem *level: 3* a více.

Zhodnocení:

- Resilience postupně 3.20, 3.21, 3.22, 3.23 roste od nízké až po vysokou. Odstranění je možné, ale vyžaduje značnou snahu a automatizaci.
- Potenci hodnotíme jako střední, ve variantách s vyššími úrovněmi 3.23 stoupá.

3.4.5 Změna notace atributů

```
def function(a, b):
  a.valA = 10
  return b.valB
```

Listing 3.24: Zdrojový kód před aplikací obfuskace.

```
def function(a, b):
  a.valA = 10
  return getattr(b, 'valB')
```

Listing 3.25: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0*.

```
def function(a, b):
  setattr(a,'valA',10)
  return getattr(b,'valB')
```

Listing 3.26: Zdrojový kód po aplikaci obfuskace s parametrem *level: 1* a více.

Zhodnocení:

- Resilience je v obou případech nízká, obrácení algoritmu je možné automatizovat.

- Varianta s nižší úrovní 3.25 nahrazuje pouze `getter` má tedy i nižší potenci. Druhá metoda provádí kvalitnější obfuskaci.

3.4.6 Vkládání predikátů

```
def function(a, b):
  c = a
  c = c + b
  return c - b - a
```

Listing 3.27: Zdrojový kód před aplikací obfuskace.

```
def function(a, b):
  c = a
  if hasattr(isinstance, 'minute'):
    pass
  else:
    c = (c + b)
  return ((c - b) - a)
```

Listing 3.28: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0*.

```
def function(a, b):
  if (object.__eq__(77, -58) is NotImplemented):
    while hasattr(bytes, 'cls'):
      pass
    else:
      c = a
  if hasattr(hex, 'text'):
    pass
  else:
    c = (c + b)
  return ((c - b) - a)
```

Listing 3.29: Opětovně spuštěná obfuskace.

Zhodnocení:

- Resilience je střední až vysoká. V současné verzi není implementována parametrizace, ale výstup je velmi randomizován.
- Potenci hodnotíme jako střední až vysokou, opět v závislosti na vygenerovaných podmínkách.

3.4.7 Přejmenování argumentů

```
def function(a, b, c=10, *d, **e):
  return e[a] + e[b] + e[d[c]]
```

Listing 3.30: Zdrojový kód před aplikací obfuskace.

```
def function(I11I, III1, I1I1=10, *I1II, **I111):
  return ((I111[I11I] + I111[III1]) + I111[I1II[I1I1
     ]])
```

Listing 3.31: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0* a více.

Zhodnocení:

- Resilience je vysoká. Provedená modifikace je jednosměrná. Parametrizace není implementována.
- Potenci hodnotíme jako střední až vysokou, v závislosti na struktuře původního programu a pojmenování argumentů.

3.4.8 Převod definic na AST

```
def function(a, b):
  return a + b * 10
```

Listing 3.32: Zdrojový kód před aplikací obfuskace.

```
from ast import *
I11I1III11I11 = fix_missing_locations(Module(body=[
   FunctionDef(name='function', args=arguments(args=[
   arg(arg='a', annotation=None), arg(arg='b',
   annotation=None)], vararg=None, kwonlyargs=[],
   kw_defaults=[], kwarg=None, defaults=[]), body=[
   Return(value=BinOp(left=Name(id='a', ctx=Load()),
   op=Add(), right=BinOp(left=Name(id='b', ctx=Load()
   ), op=Mult(), right=Num(n=10)))))], decorator_list
   =[], returns=None)]))
exec(compile(I11I1III11I11, 'I111II111I11I', 'exec'))
```

Listing 3.33: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0*. Pro referenci nezkráceno.

Zhodnocení:

- Resilience je střední až vysoká, bylo by možné obrátit průběh obfuskace a kód dekompilovat.
- Potenci hodnotíme jako střední až vysokou, znepřehlednění situace je velmi dobré. Modul byl implementován jako deterministický s jednou úrovní parametru.

3.4.9 Sloučení funkcí

```
def functionA(a, b):
  return a + b

def functionB(x, y):
  return x - y
```

Listing 3.34: Zdrojový kód před aplikací obfuskace.

```
def l111111111(lll1111111, lll1111111, *lll1111111,
   **lll1111111):
  if (lll1111111 == 'll1111111l'):
    def l111111111():
      return (lll1111111.l111111111 - lll1111111.
         l111111111)
    return l111111111()
  elif (lll1111111 == 'lll1111111'):
    def l111111111():
      return (lll1111111.llll111111 + lll1111111.
         lll1111111)
    return l111111111()

def functionA(a, b):
  lll1111111 = (lambda : None)
  lll1111111.llll111111 = a
  lll1111111.llll111111 = b
  return l111111111('lll1111111', lll1111111)

def functionB(x, y):
  lll1111111 = (lambda : None)
  lll1111111.l111111111 = x
  lll1111111.l111111111 = y
  return l111111111('ll1111111l', lll1111111)
```

Listing 3.35: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0* a více.

Zhodnocení:

- Resilience je vysoká, jedná se o kombinaci několika úprav, které přidají další úrovně indirekce.
- Potenci hodnotíme jako vysokou, informace o parametrech a volání funkce je silně pozměněna.

3.4.10 Šifrování řetězců

```
def function(a, b='retezec'):
  return a + b
```

Listing 3.36: Zdrojový kód před aplikací obfuskace.

```
I1IIIII1II = bytearray(b'\xf8\x82\x8b\xc8\x13\x10O')
I11I1II1I1 = bytearray(b'\x06\x05\x06\x01\x05\x01\x06
   \x00\x01\x00\x04\x00\x00\x00\x03\x05\x06\x02\x02\
   x02\x04\x04\x05\x02\x04\x01\x01\x01\x02\x00\x00\
   x03\x01\x06\x05\x06\x02\x06\x04\x05\x05\x03\x00\
   x03\x03\x00\x01\x02\x05\x01\x03\x01\x03\x03\x02\
   x01\x01\x00\x00\x06\x03\x02\x00\x06\x06\x06\x02\
   x00\x04\x06\x01\x05\x02\x05\x06\x02\x01\x02\x04\
   x04\x00\x05\x06\x02\x02\x04\x06\x02\x05\x01\x01\
   x06\x04\x04\x06\x02\x01\x01\x02\x00\x05')

def II11111111(III1II111I, II1111III1):
  def IIIIII1II1(II1III11II, I111II1I11, I11II1II11,
     I11IIII1I1):
    ...

  def I1I1I1III1(II1III11II, I111II1I11, III1111II1,
     I11II1II11, d):
    ...

  def II1I11I111(I11IIII1I1):
    ...

def II1III111I(I1111111I1, I11III11I1):
  return II11111111(I1111111I1, I11III11I1).decode('
     utf-8')
II1I1I1I1I = None

def function(a, b=II1III111I(0, 7)):
  return (a + b)
```

Listing 3.37: Zdrojový kód po aplikaci obfuskace s parametrem *level: 0* a více. Zkráceno.

Zhodnocení:

- Resilience je střední až vysoká, implementovaná metoda je poměrně odolná vůči automatické extrakci řetězců.
- Potenci hodnotíme jako vysokou, informace o použitých řetězcích se silně modifikována.

V této sekci jsme zhodnotili implementované techniky obfuskace jako nezávislé moduly. Jejich hlavní kvalita ale spočívá v kombinaci, případně několikanásobném použití. Tímto způsobem se zvýší metriky kvality a zároveň značně stoupne složitost opačného procesu. Deobfuskační algoritmus by musel odhadnout pořadí aplikovaných transformací, případně iterativně zkoušet, což zvyšuje nároky na čas a výpočetní výkon.

3.5 Srovnání s ostatními nástroji

V analytické části jsme uvedli několik nástrojů, které se zabývají obfuskací pro jazyk Python 3. Většinu z nich jsme zhodnotili jako jednostranně zaměřené nástroje, kterým chybí možnosti pro dodatečné rozšíření. Navrhli a implementovali jsme proto modulární nástroj, které umožňuje opakované aplikování jednodušších modifikací.

Srovnání s programem `py_compile` je z důvodu odlišného zaměření nevhodné. `py_compile` si klade za cíl zejména kompilaci do bytekódu a zrychlení běhu programu. Kromě toho existují volně šiřitelné dekompilátory, které dokáží původní program obnovit téměř dokonale.

- `Simon's Python Obfuscator` je jednoúčelový nástroj, který zvládá obfuskaci pouze jednoho souboru zároveň. Samotná modifikace spočívá spíše v komprimaci zdrojového kódu. Při spuštění je původní kód obnoven a kompilován. Celý proces lze snadno obrátit a získat původní kód včetně komentářů. Tento nástroj by bylo možné případně implementovat jako jeden z modulů.

- `Oxyry Python Obfuscator` je webový nástroj, který zvládá odstraňování komentářů a také přejmenování proměnných, funkcí a argumentů. Odstranění komentářů a přejmenování argumentů podporuje námi implementovaný nástroj také. Protože tento obfuskátor přejmenovává i globální identifikátory, je možné, že výsledný program bude za určitých podmínek nefunkční. Zejména pokud využívá dynamické funkce jako `exec`, `eval` apod. Z tohoto důvodu jsme tento typ přejmenování neimplementovali, ačkoliv je možné ho doplnit.

- `Opy` má podobné schopnosti jako předešlé nástroje. Zajímavou vlastností je možnost vyloučit některý soubor, či jeho část z modifikace. Tuto vlastnost jako jedinou náš nástroj neimplementuje. Bylo by možné upravit hlavní řídící program a pomocí parametrizace tuto možnost doplnit. Doplnění by navíc vyžadovalo podstatný zásah do jednotlivých modulů a změnu společného rozhraní.

- `pyminifier` jako jediný nástroj využívá možnosti unicode, tedy vkládání speciálních snaků. Dokáže také přejmenovávat identifikátory a odstranit řetězce.

Z porovnání existujících nástrojů vyplývá, že pro velkou část námi implementovaných modulů neexistuje adekvátní odpověď a tudíž není přímé srovnání možné. Chybí zejména moduly pro práci s řetězci a konstantami, dále chybí modifikace zaměřené specificky na vlastnosti Pythonu. Velký potenciál vidíme v kombinaci jednodušších modifikací za účelem kvalitní obfuskace.

3.6 Zhodnocení implementace

V této části jsme popsali implementaci obfuskačního nástroje pro Python 3. Vytvořili jsme modulární program, který umožňuje spuštění obfuskačních modulů nezávisle na sobě a zároveň umožňuje jejich parametrizaci. Dále jsme popsali implementované moduly spolu s vlastnostmi a případnými problémy. Pokusili také jsme se stanovit kvalitu vytvořeného výstupu. Do samostatné sekce jsme oddělili zhodnocení kvality modulů aplikovaných na reálné ukázky zdrojových kódů.

Program byl navržen a implementován tak, aby poskytoval dostatečnou flexibilitu. Je tedy přímo možné implementovat další obfuskační moduly a rozšíření. Z pohledu obfuskací jsme implementovali jak typické modifikace funkční pro ostatní jazyky a prostředí, tak moduly specifické pro Python 3, které využívají jeho unikátních vlastností. Účinnými metodami obfuskace ukázalo být *šifrování řetězců*, *sloučení funkcí*, *přejmenování argumentů*, *odstranění komentářů*. Skutečný potenciál tohoto programu se však ukazuje až ve vhodné kombinaci několika obfuskačních technik. Pro některé moduly jsme tak uvedli vhodnou kombinaci s ostatními metodami a očekávané vlastnosti.

Závěr

Cílem této práce bylo analyzovat, navrhnout a implementovat obfuskační nástroj pro jazyk Python 3 spolu se zadanými technikami obfuskace. Požadavkem bylo vytvořit modulární program, který bude snadné rozšířit a doplnit o nové techniky.

V části práce věnované analýze jsme podrobně představili programovací jazyk Python verze 3 a uvedli čtenáře do problematiky obfuskace. Následoval popis možných přístupů a použitelných technik. Součástí analýzy je také představení existujících nástrojů určených k modifikaci zdrojových kódů.

Navrhli jsme modulární nástroj určený k obfuskaci. Kromě nástroje samotného jsme vytvořili algoritmy a techniky použitelné k obfuskaci zdrojových kódů v jazyce Python. V práci jsme dále popsali jejich principy, přednosti a případná omezení.

V implementační části jsme představili námi zvolené technologie. Popsali jsme tvorbu obfuskačního programu spolu s rozhraním poskytovaným pro obfuskační moduly. Na základě algoritmů z návrhové části jsme tyto moduly implementovali, popsali jejich vlastnosti a zhodnotili kvalitu.

Práci jsme zakončili srovnáním existujících řešením s námi vytvořenou sadou obfuskačních technik.

Literatura

[1] Python Software Foundation: Logo of the Python programming language. [cit. 2017-08-20]. Dostupné z: http://legacy.python.org/community/logos/

[2] TIOBE software BV: TIOBE index — statistics of programming languages usage. [cit. 2017-08-20]. Dostupné z: https://www.tiobe.com/tiobe-index/

[3] Guido van Rossum: *Python History*. [cit. 2017-08-20]. Dostupné z: https://svn.python.org/projects/python/trunk/Misc/HISTORY

[4] Python Software Foundation: *Python 3.5.1 Release Notes*. [cit. 2017-08-20]. Dostupné z: https://www.python.org/download/releases/3.5.1/

[5] Collberg, C.; Thomborson, C.; Low, D.: A Taxonomy of Obfuscating Transformations. 1997. Dostupné z: https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf

[6] Knuth, D. E.: On the translation of languages from left to right. *Information and Control*, ročník 8, č. 6, 1965: s. 607–639.

[7] Melichar, B.: Syntax directed translation with LR parsing. *Lecture Notes in Computer Science*, 1992: s. 30–36.

[8] IOCCC: The International Obfuscated C Code Contest. [cit. 2017-08-20]. Dostupné z: http://www.ioccc.org/years.html#2013

[9] Python Software Foundation: *Python Abstract Syntax Trees - Grammar*. [cit. 2017-08-20]. Dostupné z: https://docs.python.org/3/library/ast.html#abstract-grammar

[10] Popa, M.: Techniques of Program Code Obfuscation for Secure Software. 2011. Dostupné z: file:///C:/Users/holoubekm/Desktop/51-160-1-PB.pdf

[11] Python Software Foundation: *Python Floating Point Arithmetic*. [cit. 2017-08-20]. Dostupné z: https://docs.python.org/3/tutorial/floatingpoint.html

[12] Blake, E. H.; Cook, S.: On Including Part Hierarchies in Object-Oriented Languages with an Implementation in Smalltalk. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '87, London, UK, UK: Springer-Verlag, 1987, ISBN 3-540-18353-1, s. 41–50. Dostupné z: http://dl.acm.org/citation.cfm?id=646147.679045

[13] Python Software Foundation: *Syntax For Dynamic Attribute Access*. [cit. 2017-08-20]. Dostupné z: https://www.python.org/dev/peps/pep-0363/

[14] Python Software Foundation: *Documentation of the function exec*. [cit. 2017-08-20]. Dostupné z: https://docs.python.org/3/library/functions.html?highlight=exec#exec

[15] Jędrzejewski-Szmek, Z.: Advanced Python. [cit. 2017-08-26]. Dostupné z: https://python.g-node.org/python-summerschool-2011/_media/materials/advanced_python/advanced_python-handout.pdf

[16] Gupta, S. S.: Analysis and Implementation of RC4 Stream Cipher. 2013. Dostupné z: http://souravsengupta.com/pub/phd_thesis_2013.pdf

[17] Garman, C.; Paterson, K. G.; der Merwe, T. V.: Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C.: USENIX Association, 2015, ISBN 978-1-931971-232, s. 113–128. Dostupné z: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/garman

[18] Python Software Foundation: *Documentation of the secrets module*. [cit. 2017-08-20]. Dostupné z: https://docs.python.org/3/library/secrets.html

Příloha A

Přílohy

A.1 Návod k použití

Typické příklady spuštění programu:

- Zobrazení nápovědy:

```
python3 driver.py --help
```

- Výpis všech dostupných modulů:

```
python3 driver.py --list-actions
```

- Zobrazení podrobností pro vybraný modul:

```
python3 driver.py --extended-info FunctionsMerger
```

- Spuštění obfuskátoru na soubory vstupní složky za použití všech dostupných metod:

```
python3 driver.py -i input_folder \
-o output_folder \
-a ConstantsRelocator \
-a ConstantsScrambler \
-a DefinitionsToExec \
-a FunctionsDecorator \
-a FunctionsMerger \
-a ArgumentsRenamer \
-a CommentsRemover \
-a AttributesNotation \
-a OpaquePredicatesInjector \
-a BuiltinsScrambler \
-a StringsEncryptor \
```

```
-a StringsRelocator \
--striping-length=0 \
--verbose \
--level=0
```

- Kombinace několika stejných metod s nejvyšší úrovní obfuskace:

```
python3 driver.py -i input_folder \
-o output_folder \
-a FunctionsMerger \
-a FunctionsDecorator \
-a FunctionsMerger \
--verbose \
--level=5
```

- Kombinace vhodná k obfuskaci řetězců:

```
python3 driver.py -i input_folder \
-o output_folder \
-a AttributesNotation \
-a StringsRelocator \
-a StringsEncryptor \
--verbose \
--level=5
```

A.2 Tvorba nového modulu

Ve složce *actions* jsme za účelem rozšíření připravili prázdný soubor obfuskátoru. Tato třída má implementované rozhraní a nachází se v souboru *empty.py*. Její metody jsou okomentovány, aby bylo zřejmé, jak pokračovat.

A.3 Technické poznámky

Během implementace jsme narazili na několik technických otázek, které je vhodné poznamenat:

- Hloubka rekurze Parseru — Parser zpracovávající zdrojové kódy Pythonu má fyzicky omezenou hloubku zanoření na 1500. To podle dané struktury zdrojového programu omezuje například maximální zanoření podmínek, funkcí a dalších struktur v kódu včetně závorek. Jedná se o vlastnost implementace, pro změnu by bylo nutné překompilovat zdrojové kódy Pythonu [20]. Některé obfuskační techniky (`OpaquePredicatesInjector`,

[20] `https://github.com/python/cpython/blob/master/Parser/parser.h` — hlavičkový soubor parseru omezující maximální hloubku

`FunctionsMerger`, ...) mohou hloubku kódu zvyšovat, je proto dobré výsledný program spustit, aby byla jeho správná funkčnost ověřena.

- Hloubka rekurze — Interpreter, který spouští programy v Pythonu, má omezenou hloubku rekurze, aby nedošlo k vyčerpání systémového zásobníku.

  Kód pro získání a nastavení současného limitu:

  ```
  import sys
  #Soucasna maximalni hloubka
  print(sys.getrecursionlimit())
  sys.setrecursionlimit(2000)

  #Na platforme Windows 10 je hloubka defaultne
     omezena na 1000.
  ```

  Hodnotu je třeba nastavit s rozmyslem, protože interpreter využívá přímo systémový zásobník. Pokud by docházelo k častým výjimkám z důvodu překročení maximální hodnoty, může být nutné překompilovat Python s vhodně nastaveným parametrem `stack`.

  Některé námi vytvořené obfuskační moduly (`DefinitionsToExec`, `FunctionsMerger`, ...) přidávají k funkcím další vrstvy a zvyšují tím spotřebu zásobníku. Při používání obfuskátoru v paměťově náročných programech je třeba toto brát v úvahu.

- Obfuskace vnořených modulů — Námi vytvořený nástroj je určen k obfuskaci zdrojových souborů v dané složce. V současné době není možné obfuskovat hierarchii modulů a vnořených složek. V případě potřeby je možné spustit obfuskátor na jednotlivé složky zvlášť.

Příloha **B**

Používané pojmy

obfuskace	téma této práce ♡ — postup úpravy zdrojových kódů, která cíleně zakrývá informace a postupy v nich uložené
strom	pojem z teorie grafů, graf ve kterém existuje právě jedna cesta mezi každými dvěma uzly
AST	abstraktní syntaktický strom
parsování	fáze kompilace během níž je vytvořen AST
immutability	vlastnost objektu v Pythonu, kdy každá operace vytvoří jeho kopii
bytekód	program zapsaný v instrukční sadě dané platformy
property	atribut třídy, který má přidružené metody typu `getter` a `setter`
cachování	(kešování) je způsob optimalizace, kdy jsou data uložena v dočasné paměti

Příloha C

Obsah přiloženého CD

- `Holoubek_Martin_DP_2018`
  - `src` — zdrojové kódy nástroje
    - `actions` — složka s obfuskátory
    - `driver.py` — hlavní spustitelný soubor
  - `samples` — ukázka aplikace na reálném projektu
    - `input` — výchozí zdrojové kódy projektu `python-poloniex`
    - `outputs` — výstupy z jednotlivých obfuskací
  - `latex` — zdrojové kódy textu práce
  - `Holoubek_Martin_DP_2018.pdf` — text práce ve formátu `pdf`