POLITECHNIKA BIAŁOSTOCKA

WYDZIAŁ INFORMATYKI

Katedra Oprogramowania

PRACA DYPLOMOWA MAGISTERSKA

Porównanie metod inżynierii wstecznej w językach kompilowanych do kodu bajtowego na przykładzie Pythona

WYKONAWCA: **Piotr Tynecki**
Imię i nazwisko

PODPIS: ....

PROMOTOR: **dr inż. Cezary Bołdak**
Imię i nazwisko

PODPIS: ....

BIAŁYSTOK 2014

Karta dyplomowa

POLITECHNIKA BIAŁOSTOCKA Wydział .... Katedra/Zakład	Studia .... stacjonarne/niestacjonarne .... studia I stopnia/ studia II stopnia	Nr albumu studenta .... Rok akademicki .... Kierunek studiów .... Specjalność ....

....
Imię i nazwisko studenta

TEMAT PRACY DYPLOMOWEJ:
....

Zakres pracy:
1. ....
2. ....

Słowa kluczowe (max 5):
…………………………………………………………………………………….

....
Imię i nazwisko, stopień/ tytuł promotora - podpis
Imię i nazwisko kierownika katedry - podpis

....	....	....
Data wydania tematu pracy dyplomowej - podpis promotora	*Regulaminowy termin złożenia pracy dyplomowej*	*Data złożenia pracy dyplomowej - potwierdzenie dziekanatu*

....	....
Ocena promotora	*Podpis promotora*

....	....	....
Imię i nazwisko, stopień/ tytuł recenzenta	*Ocena recenzenta*	*Podpis recenzenta*

Thesis topic: A comparison of reverse engineering methods for Python compiled binaries

SUMMARY

Python programming language is quickly gaining as a tool for creating commercial, business and closed-source software. This is because of possibility of building Python compiled binaries, also known as "frozen binaries". *Frozen binaries* are an executable archive which includes Python bytecode, interpreter and external libraries. The binary file is built by public exe-packers such as py2exe, py2app, cx_Freeze, bbfreeze or PyInstaller. It is portable and protected from regular users.

However, Software Reverse Engineering proves high susceptibility of Python *frozen binaries* for restoring original directory structure and source code. Despite the fact that Python *frozen binaries* are secured, performing decompression and decompilation, it is still possible to use the methods described in this paper.

Spis treści

1. Wstęp 6

  1.1 Wprowadzenie 6

  1.2 Cel i zakres pracy 7

2. Wykonywanie kodu 9

  2.1 Kompilacja i program wykonywalny 9

  2.2 Interpretacja 10

  2.3 Kod bajtowy i maszyna wirtualna 11

3. Inżynieria wsteczna oprogramowania 12

  3.1 Przykłady zastosowań inżynierii wstecznej oprogramowania 13

  3.2 Dezasemblacja, dekompilacja i debugowanie 14

  3.3 Inżynieria wsteczna oprogramowania w świetle prawa autorskiego 16

4. Język programowania Python 18

  4.1 Historia Pythona 18

  4.2 Wady i zalety Pythona 20

  4.3 Zastosowanie Pythona 22

  4.4 Python 2 a Python 3 24

  4.5 Alternatywne implementacje Pythona 24

    4.5.1 CPython 25

    4.5.2 Jython 25

    4.5.3 IronPython 26

    4.5.4 PyPy 26

  4.6 Wykonywanie kodu w Pythonie 28

    4.6.1 Kod bajtowy Pythona 28

    4.6.2 Zoptymalizowany kod bajtowy Pythona 29

    4.6.3 Python Virtual Machine 30

4.6.4 Struktura pliku .pyc 31

5. Narzędzia „zamrażające” Pythona 35

5.1 Czym są „frozen binaries”? 35

5.2 py2exe 36

5.3 py2app 38

5.4 cx_Freeze 38

5.5 bbfreeze 39

5.6 PyInstaller 40

6. Autorskie metody inżynierii wstecznej 45

6.1 Inżynieria wsteczna pythonowych binariów - py2exe 45

6.2 Inżynieria wsteczna pythonowych binariów - cx_Freeze 46

6.3 Inżynieria wsteczna pythonowych binariów - bbfreeze 47

6.4 Inżynieria wsteczna pythonowych binariów - PyInstaller 48

6.5 Kompresja i dekompresja plików wykonywalnych 49

7. Weryfikacja opracowanych metod inżynierii wstecznej na przykładach 51

7.1 Porównanie zrealizowanych metod inżynierii wstecznej 75

8. Podsumowanie 79

Literatura 81

1. Wstęp

1.1 Wprowadzenie

Python jako język programowania ogólnego przeznaczenia, do działania pisanych w nim programów wymaga środowiska, w skład którego wchodzi interpreter oraz cała struktura katalogów i plików biblioteki standardowej. Oznacza to, że bez odpowiedniego zaplecza uruchomieniowego, pythonowe programy stają się bezużyteczne.

Takie środowiska domyślnie dostępne są w większości systemów Unix oraz GNU/Linux. Zupełnie odwrotnie przedstawia się sytuacja w przypadku produktów firmy Microsoft. Dlatego też społeczność Pythona wypracowała na przestrzeni lat dwa rozwiązania, które pozwalają na pełną przenośność pythonowych programów, bez konieczności posiadania zainstalowanego w systemie interpretera oraz innych zależności.

Rozwiązanie pierwsze polega na użyciu oprogramowania tłumaczącego kod języka Python do postaci kodu C lub C++ i zrealizowaniu jego kompilacji, co powoduje otrzymanie na wyjściu natywnej wersji programu.

Podejście to zapewnia szybkość działania programu na poziomie kodu maszynowego oraz bardzo mały rozmiar pliku wynikowego. Jednakże, obecnie dostępne kompilatory (Cython, Nuitka, Shed Skin) Pythona mają liczne ograniczenia. Jednym z nich jest obsługa wyłącznie standardowej biblioteki Pythona. Oznacza to, że kompilacji nie zostanie poddany skrypt, który korzysta z popularnych, zewnętrznych bibliotek Pythona, tj. NumPy, SciPy, matplotlib czy Reportlab, rozszerzających możliwości języka.

Rozwiązanie drugie, sprowadza się do wykorzystania archiwizatorów, których działanie polega na połączeniu w jeden pakiet bajtowej reprezentacji pythonowego kodu źródłowego wraz z interpreterem i wszelkimi plikami pomocniczymi jakich program potrzebuje do działania (np. z multimediami). W rezultacie użytkownik otrzymuje wykonywalny, przenośny plik binarny o dość pokaźnym rozmiarze.

Archiwizowanie programu do postaci wykonalnej rozwiązuje jednak problem jaki stwarzają kompilatory Pythona, gdyż pakowanie nie narzuca ograniczeń względem

wsparcia dla bibliotek zewnętrznych języka. Dlatego też zdecydowanie częściej sięga się po tę metodę w przypadku rozbudowanych programów lub gdy brana pod uwagę jest komercyjna natura dystrybucji oprogramowania.

Program, który został przetłumaczony i skompilowany do postaci natywnego pliku wykonywalnego zdecydowanie trudniej jest poddać procesowi inżynierii wstecznej, niżeli program powstały w wyniku przeprowadzonej archiwizacji. Wynika to z prostej przyczyny. Archiwizatory tworzą tak naprawdę plik wykonalny będący niestandardowym archiwum ZIP lub zlib, który przy użyciu zaprezentowanych w niniejszej pracy algorytmów i dekompilatorów można sprowadzić ponownie do formy kodu źródłowego. W przypadku programów skompilowanych do postaci kodu maszynowego, taka praktyka jest niemal niemożliwa.

Oznacza to, że dystrybucja pythonowych programów powstałych w wyniku archiwizacji, nie zapewnia wystarczającego bezpieczeństwa i naraża interes oraz prawo własności intelektualnej ich autorów. Nie jest to jednak jednoznaczne z tym, że język Python nie powinien być wykorzystywany do tworzenia komercyjnego, zamkniętego oprogramowania. Sęk tkwi we właściwej metodzie jego dystrybuowania oraz umiejętności zabezpieczenia programów wykonywalnych.

1.2 Cel pracy

Niniejsza praca dyplomowa ma na celu zaprezentowanie autorskich metod inżynierii wstecznej, za pomocą których można dokonać dekompresji i dekompilacji wszystkich typów plików wykonywalnych zawierających skrypty Pythona, powstałych w wyniku użycia archiwizatorów. Każda z opisanych metod dąży do uzyskania zbliżonego do oryginalnego kodu źródłowego programów oraz wyłuskania z nich plików konfiguracyjnych, bazodanowych czy multimediów.

W związku z tym, że problematyka zagadnienia jest obszerna, przeprowadzone badania, zostały zawężone do plików wykonywalnych powstałych w wyniku użycia pakietów: py2exe, cx_Freeze, bbfreeze oraz PyInstaller. Prezentowane procedury przygotowane są pod kątem binariów tworzonych na systemy z rodziny Linux (Debian) i Windows (Windows XP, Windows 7, Windows 8). Pliki binarne powstały przy wykorzystaniu wersji Pythona 2.4, 2.5, 2.6, 2.7 i Pythona 3.3.

Pragnę również zaznaczyć, że nie popieram ani nie zachęcam do łamania zabezpieczeń oprogramowania objętego licencją. Czynności te są nieetyczne a przede wszystkim - nielegalne. Nie ponoszę odpowiedzialności za sposób wykorzystania przytoczonej tu wiedzy.

Praca została podzielona na dwie części, na część teoretyczną oraz projektową. W pierwszej części niniejszej pracy - rozdziały od 2 do 5, przedstawiam zarys teoretyczny dostępnych sposobów na wykonywanie kodu języków programowania, wprowadzam pojęcie inżynierii oprogramowania oraz jego aspekt prawny. Omawiam w niej także język Python, prezentuję od wewnątrz jego model wykonawczy kodu a także dostępne i popularne pakiety, które tworzą z jego źródeł i środowiska niezależne pliki binarne.

Druga cześć jest częścią projektową - rozdziały od 6 do 7, a jej przedmiotem jest prezentacja opracowanych, autorskich metod inżynierii wstecznej oraz zbadanie ich skuteczności na przygotowanej bazie pythonowych plików binarnych.

2. Wykonywanie kodu

Kod źródłowy jest podstawowym wynikiem pracy każdego programisty. Stanowi on tekst programu powstały przy użyciu składni wybranego języka programowania wysokiego bądź niskiego poziomu. W większości przypadków jego przetworzona forma do postaci kodu wynikowego (maszynowego) tworzy program, zawierający rozkazy (instrukcje, polecenia) gotowe do zrealizowania przez procesor, który można już bezpośrednio uruchomić na komputerze. W informatyce wyróżniamy dwa główne procesy tego typu translacji: kompilacja lub interpretacja.

2.1 Kompilacja i program wykonywalny

Kompilator to program tłumaczący kod napisany w języku wysokiego poziomu, takim jak C czy Pascal, i przekładający go na kod maszynowy. Każda instrukcja w języku wysokiego poziomu jest zamieniana na wiele instrukcji maszynowych. To, co wchodzi do kompilatora, nazywa się programem źródłowym, a to, co pojawia się na wyjściu, programem wynikowym. Proces zamiany nazywa się kompilacją, a program poddany takiej zamianie - programem kompilowanym. Wchodzący program musi być kompilowany, zanim można go będzie wykonać.

Kompilator nie tylko tłumaczy instrukcje programu, ale zawiera odwołania do procedur i funkcji z biblioteki systemowej i przydziela obszary w pamięci głównej. Następnie program wynikowy może być ładowany i wykonywany przez komputer. Kompilator może zwracać informacje o błędach (głównie składniowych) znalezionych w programie źródłowym w trakcie kompilacji. Dodatkowo, wykonuje operację optymalizacji kodu wynikowego.

Program wykonywalny to plik, który uruchamia się bezpośrednio w środowisku systemu operacyjnego. Zawiera binarną reprezentację instrukcji konkretnego typu procesora. Dodatkowo, zazwyczaj posiada wywołania systemowe,

dlatego pliki te są specyficzne nie tylko dla procesora co dla danego systemu operacyjnego.

Nazwy plików wykonywalnych różnią się sygnaturą pliku i rozszerzeniami, np. w DOS i Windows przyjęto `.scr`, `.com` i `.exe`. W systemach uniksowych pliki mają ustawiony atrybut wykonywalności (oznaczany literą x). W przypadku iOS, Mac OS X czy Symbian OS jest to rozszerzenie `.app`.

Programy wykonywalne są wynikiem poprawnej kompilacji. Zapewniają najwyższą wydajność, jednakże są ściśle powiązane z platformą sprzętową.

2.2 Interpretacja

Interpretator to program zmieniający skrypt napisany w języku wysokiego poziomu np. PHP i Perl na postać, która może być akceptowana i wykonywana przez komputer. Analizuje on każdą linię kodu języka programowania wysokiego poziomu, a następnie przystępuje do określonej czynności, co nazywa się interpretacją.

Jako przeciwieństwo kompilatora, interpretator nie dokonuje przetłumaczenia całego programu, zanim zacznie go wykonywać. Kod źródłowy wykonywany jest na bieżąco, w trakcie tłumaczenia. Takie postępowanie prowadzi do tego, iż program działa wolniej aniżeli w sytuacji, kiedy używany jest kompilator. Spowodowane to jest m.in. tym, że interpretator musi odczytać każdą użytą instrukcję, odnieść się do pamięci w celu ustalenia jej działania, a dopiero potem przechodzi do jej wykorzystania. Niemniej upraszcza on proces wprowadzania, wykonywania oraz zmieniania programu źródłowego.

Dodatkowo, języki interpretowane zapewniają przenośność programów, które często są niezależne od platformy i systemu operacyjnego.

2.3 Kod bajtowy, maszyna wirtualna

Alternatywnym rozwiązaniem dla kompilacji i interpretacji jest kompilacja programów do postaci pośredniej, tzw. kodu bajtowego (ang. byte code), a następnie przesyłanie go do maszyny wirtualnej, która interpretuje i wykonuje jego rozkazy. Języki programowania, które wprowadziły taki tryb pracy to m.in. Python, Java i C#.

Kod bajtowy jest niskopoziomową, niezależną od platformy reprezentacją kodu źródłowego. Natomiast pod pojęciem maszyny wirtualnej należy rozumieć interpretator kodu bajtowego.

Metodę tę stosuje się z myślą o szybkości wykonania - kod bajtowy zasadniczo działa o wiele efektywniej od oryginalnych instrukcji z kodu źródłowego zawartego w pliku tekstowym. Co więcej, jest on generowany tylko raz, o ile oczywiście nie zmieniony został kod źródłowy od czasu ostatniego zapisu pliku, dzięki czemu pomijany jest etap ponownej kompilacji.

Rezultat jest taki, że język programowania Python działa z szybkością znajdującą się pomiędzy wydajnością tradycyjnych języków kompilowanych a języków interpretowanych, zapewniając przy tym pełną przenośność programów.

3. Inżynieria wsteczna oprogramowania

Inżynieria wsteczna (ang. Reverse Engineering, RE) to proces odwrócenia cyklu produkcji oprogramowania, który pozwala opisać działanie oprogramowania na wyższej warstwie abstrakcji, na przykład dzięki diagramowi ERD[1] lub diagramowi klas języka UML[2].

Z kolei, gdy mowa o inżynierii wstecznej oprogramowania (ang. Software Reverse Engineering, SRE), należy myśleć o czynnościach, które z kodu maszynowego programu wyłuskują kod języka programowania zrozumiały dla ludzi, czyli cofają efekty pracy kompilatora.

Inżynierię wsteczną oprogramowania można podzielić na następujące techniki:

- obserwacja wymienianej informacji:

  Sposób wykorzystywany do łamania protokołów sieciowych, na podstawie analizy podsłuchiwanego ruchu w sieci LAN. Służy również do wstecznej inżynierii sterowników do różnych urządzeń. Ogólna koncepcja algorytmu polega na odtworzeniu danego protokołu lub sterownika i w analogicznym sposobie wymienia informacji z systemami zewnętrznymi;

- dezasemblacja za pomocą debuggerów (Deasemblerów):

  Metoda ciesząca się największą popularnością, dzięki której otrzymuje się kod programu w postaci mnemoników[3] danego języka programowania, na podstawie analizowanego kodu maszynowego;

---

1 **Digram ERD (ang. Entity-Relationship Diagram)** - rodzaj graficznego przedstawienia związków pomiędzy encjami używany w projektowaniu systemów informacyjnych do przedstawienia konceptualnych modeli danych używanych w systemie. Źródło: http://pl.wikipedia.org/wiki/Diagram_zwi%C4%85zk%C3%B3w_encji

2 **Diagram UML (ang. Unified Modeling Language)** - język formalny wykorzystywany do modelowania różnego rodzaju systemów oraz modeli systemów informatycznych. Stanowi głównie reprezentację graficzną - jego elementom przypisane są symbole, które wiązane są ze sobą na diagramach. Źródło: http://pl.wikipedia.org/wiki/Unified_Modeling_Language

3 **Mnemonik** - w językach assemblera jest to składający się z kilku liter kod, który oznacza konkretną czynność procesora. Przykładem mogą być: "add" (z ang. dodaj) czy "sub" (z ang. odejmij). Źródło: http://pl.wikipedia.org/wiki/Mnemonik_(informatyka)

- dekompilacja za pomocą dekompilatora:

  Niektóre dekompilatory w lepszym lub gorszym stopniu mogą cofnąć kod do postaci języka wysokiego poziomu. Technika szczególnie skuteczna w przypadku języków programowania opartych na kodzie bajtowym.

3.1 Przykłady zastosowań inżynierii wstecznej oprogramowania

Inżynieria wsteczna oprogramowania najczęściej kojarzona jest z projektami wojskowymi, kopiowaniem technologii opracowanych przez inne państwa lub firmy czy z łamaniem zabezpieczeń i modyfikowaniem programów oraz gier komputerowych.

Nie ulega wątpliwości, że powyższe skojarzenia są prawidłowe, gdyż techniki inżynierii wstecznej mają szerokie zastosowanie w dziedzinie oprogramowania, ze szczególnym zastosowaniem w bezpieczeństwie IT, przy analizie złośliwego oprogramowania.

Dzień za dniem badacze złośliwego oprogramowania (ang. malware)[4], analitycy sądowi czy administratorzy muszą stawiać czoła zagrożeniu bezpieczeństwa systemów informatycznych. Ich celem może być wyjaśnienie nieautoryzowanych ingerencji, ochrona użytkowników przed wirusem lub unikanie wystawiania sytemu na niebezpieczeństwo. Aby osiągnąć te cele, konieczna jest maksymalnie szczegółowa analiza działania złośliwego oprogramowania, z którym mamy do czynienia. Tu właśnie do gry wkracza inżynieria wsteczna oprogramowania.

Mówiąc o inżynierii wstecznej oprogramowania, nie sposób ominąć pojęcia *crackingu* i *crackera*. Osoba zajmująca się łamaniem zabezpieczeń komputerowych, tj. zabezpieczenia zamkniętego oprogramowania czy serwerów WWW, określana jest

4 **Złośliwe oprogramowanie** - aplikacje i skrypty mające szkodliwe, najczęściej przestępcze lub złośliwe działanie w stosunku do użytkownika komputera czy instalacji przemysłowych.

mianem crackera. Co należy mocno podkreślić, cracker czyni to w sposób niezgodny z prawem.

Inżynieria wsteczna stosowana jest do łamania zabezpieczeń oprogramowania takich jak [1] [2]:

- ograniczenia czasowe, ilościowe lub funkcjonalne (wersje trial lub demo),
- ekrany przypominające o niepełnowartościowym oprogramowaniu i kupieniu pełnej jego wersji,
- numery seryjne lub hasło,
- sprawdzanie obecności klucza sprzętowego lub płyty CD / DVD.

Powyższa praktyka przyczyniła się do zjawiska określanego mianem „piractwa komputerowego".

Inżynieria wsteczna może także posłużyć także do [1] [2]:

- wprowadzenia zmiany wersji językowej aplikacji,
- usunięcia lub odblokowania dodatkowych funkcji w programie,
- wprowadzenia kodów / botów do gier komputerowych,
- poznania działania API usług zewnętrznych,
- osiągnięcia pewnej funkcjonalności, przy ominięciu konsekwencji wynikających z praw autorskich lub patentów.

3.2 Dezasemblacja, dekompilacja i debugowanie

Z inżynierią wsteczną oprogramowania bezpośrednio powiązane są trzy pojęcia: dezasemblacja, dekompilacja oraz debugowanie.

O ile asemblacja oznacza tłumaczenie instrukcji asemblera na kod maszynowy, to dezasemblacja polega na tłumaczeniu kodu maszynowego na instrukcje asemblera. Deasembler tłumaczy kod wynikowy programu do postaci języka asemblera.

W praktyce, ze względu na mniejszą dostępność dekompilatorów, częściej stosowana jest dezasemblacja, a następnie translacja kodu asemblera do postaci źródłowej w języku wysokiego poziomu. Deasembler nie ma zastosowania w przypadku innym niż tłumaczenie kodu maszynowego.

Deasemblery dzielą się na autonomiczne, czyli generujące pliki asemblera, które mogą być dalej analizowane, oraz interaktywne, umożliwiające wprowadzanie zmian w plikach binarnych z natychmiastowym podglądem skutku edycji.

Najczęściej wykorzystywanym deasemblerem jest IDA[5], dostępna zarówno w wersji otwartej jak i komercyjnej.

Z kolei dekompilator, to program przekształcający język maszynowy lub kod bajtowy do postaci języka wyższego poziomu. Proces tłumaczenia kodu nazywa się dekompilacją.

Dekompilacja nie odtwarza kodu źródłowego programu sprzed jego kompilacji, a jedynie postać źródłową w pewnym języku wyższego rzędu (zależnym od dekompilatora) i mającą identyczne działanie jak kod, który poddawany jest dekompilacji.

Do dekompilacji plików binarych najczęściej stosuje się Hex-Rays Decompiler[6], który stanowi rozszerzenie dla środowiska IDA Pro czy otwarto źródłowy dekompilator Boomerang[7].

Deasemblery różnią się od dekompilatorów przede wszystkim w jednym istotnym aspekcie. Chodź oba generują czytelny dla człowiek tekst, to dekompilatory robią to na wyższym poziomie, produkując kod o bardziej zwięzłej i prostszej w zrozumieniu formie.

---

5 http://en.wikipedia.org/wiki/Interactive_Disassembler

6 https://www.hex-rays.com/products/decompiler/

7 http://boomerang.sourceforge.net/

Debugowanie to proces nadzorowania wykonania programu. Odbywa się on przy użyciu debuggerów, czyli aplikacji służących do dynamicznej analizy innych programów, w celu odnalezienia i identyfikacji zawartych w nich błędów.

Podstawowym zadaniem debuggera jest sprawowanie kontroli nad wykonaniem kodu, co umożliwia zlokalizowanie instrukcji odpowiedzialnych za wadliwe działanie programu.

Obecne debuggery pozwalają na efektywne śledzenie wartości poszczególnych zmiennych, wykonywanie instrukcji krok po kroku czy wstrzymywanie działania programu w określonych miejscach.

Z moich obserwacji wynika, że stosowanie debuggerów nie cieszy się dziś dużą popularnością w gronie programistów, pomimo ich dostępności w większości współczesnych środowisk programistycznych (IDE). Z kolei, w społeczności osób zajmujących się inżynierią wsteczną, stanowią one podstawowy oręż pracy.

Godnymi polecenia debuggerami są OllyDbg[8] oraz gdb[9].

3.3 Inżynieria wsteczna oprogramowania w świetle prawa autorskiego

Ze względu na możliwe zastosowania, inżynieria wsteczna oprogramowania w wielu krajach jest techniką nielegalną. Przykładowo, w Stanach Zjednoczonych często jest zabroniona ze względu na licencje, które tego zakazują. Reguluje to paragraf 1201 DMCA[10] (ang. Digital Millennium Copyright Act), ustawy obowiązującej od 1998 w USA zabraniająca tworzenia i rozpowszechniania technologii, przy użyciu których mogą być naruszone cyfrowe mechanizmy ograniczeń kopiowania.
Z kolei w Unii Europejskiej istnieje Dyrektywa o Prawnej Ochronie Programów Komputerowych[11], która zezwala na stosowanie inżynierii wstecznej w celu badania interoperacyjności oprogramowania, jednakże zakazuje jej gdy dąży się do tworzenia

8 http://www.ollydbg.de/

9 http://www.gnu.org/software/gdb/

10 http://www.copyright.gov/1201/2003/index.html

11 **Dyrektywa Parlamentu Europejskiego I Rady 2009/24/WE z dnia 23 kwietnia 2009 r. w sprawie ochrony prawnej programów komputerowych,** http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2009:111:0016:0022:PL:PDF

konkurencyjnych produktów. Ponadto, rezultatów uzyskanych z zastosowania inżynierii wstecznej nie wolno upubliczniać.

Z kolei, analizując pojęcie inżynierii wstecznej oprogramowania w kontekście polskiego prawa, czyli Ustawy o prawie autorskim i prawach pokrewnych (art. 75 ust. 2 pkt 3 i ust. 3)[12], należy wysunąć następujące wnioski [14]:

- Inżynieria wsteczna jest dopuszczalna jedynie w celu uzyskania informacji koniecznych do osiągnięcia współdziałania niezależnie stworzonego programu komputerowego z innymi programami komputerowymi,
- Inżynieria wsteczna nie jest dozwolona, jeżeli informacje niezbędne do osiągnięcia współdziałania były łatwo dostępne dla legalnego użytkownika programu,
- Inżynieria wsteczna może dotyczyć tylko części programu niezbędnych do osiągnięcia współdziałania,
- Informacje uzyskane w wyniku inżynierii wstecznej nie mogą być zasadniczo przekazywane innym osobom, ani wykorzystywane do rozwijania, wytwarzania lub wprowadzania do obrotu programu komputerowego o istotnie podobnej formie wyrażenia lub do innych czynności naruszających prawa autorskie (np. stworzenia programu do obchodzenia zabezpieczeń technicznych).

12 **Ustawa z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych.**, http://isap.sejm.gov.pl/DetailsServlet?id=WDU19940240083

4. Język programowania Python

Python to język programowania ogólnego zastosowania, realizujący paradygmat obiektowy, strukturalny i (fragmentarycznie) funkcyjny. Najczęściej wykorzystuje się go do tworzenia skryptów systemowych, oprogramowania do obliczeń naukowych i inżynierskich oraz aplikacji internetowych. Jest również językiem, dla którego istnieje pokaźne wsparcie edukacyjne oraz duże zapotrzebowanie rynkowe.

Python posiada w pełni dynamiczny system typów i automatyczne zarządzanie pamięcią (Garbage Collection[13]). Dzięki swojej elastyczności i łatwości użycia, upraszcza wiele zadań w procesie wytwarzania oprogramowania. Python rozpowszechniany jest na otwartej licencji[14] umożliwiającej także komercyjne zastosowania. Pieczę nad rozwojem języka i jego społeczności sprawuje fundacja Python Software Foundation[15].

Python jest aktywnie rozwijany, posiada ogromną liczbę zewnętrznych bibliotek i szerokie grono użytkowników na całym świecie.

4.1 Historia Pythona

Pracę nad Pythonem rozpoczęto w 1989 roku. Jego głównym twórcą jest holenderski matematyk Guido Van Rossum, były pracownik Google, obecnie urzędujący w Dropboxie. Nazwy języka nie należy kojarzyć z gatunkiem węża, lecz z brytyjskim serialem telewizyjnym „Monty Python's Flying Circus", którego sympatykiem jest autor.

Python 1.2 był ostatnią wersją wydaną przez CWI[16], gdzie pracował Guido Van Rossum. Od 1995 roku, rozwój języka kontynuowano w CNRI[17], publikując wersje do 1.6 włącznie.

13 **Garbage collection** - jedna z metod automatycznego zarządzania dynamicznie przydzieloną pamięcią.

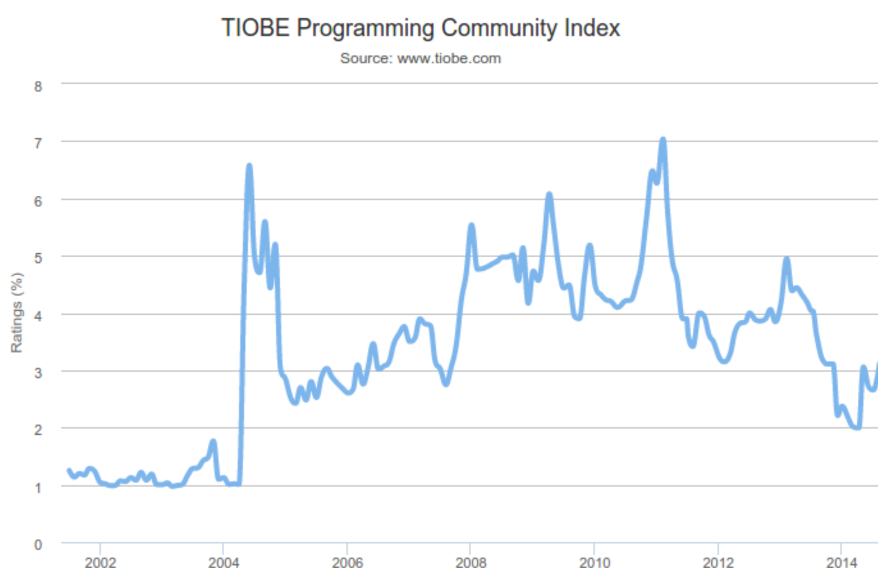
14 https://docs.python.org/2/license.html

15 **PSF** - (ang. Python Software Foundation) formalna organizacja non-profit, której misja polega na promowaniu, ochronie i rozwoju języka Python, a także wspieraniu międzynarodowej społeczności programistów piszących w tym języku i organizowaniu międzynarodowych konferencji.

16 **Centrum Wiskunde & Informatica (CWI)** - Centrum Matematyki i Informatyki w Amsterdamie.

W 2000 roku, Van Rossum i zespół pracujący nad jądrem Pythona przeniósł się do BeOpen.com w celu założenia zespołu BeOpen PythonLabs. Jedyną wersją wydaną przez tę formację był przełomowy Python 2.0. Po przejściu Guido Van Rossum do Digital Creations, całą własność intelektualną, począwszy od Pythona 2.1, przekazano w ręce fundacji Python Software Foundation. Od tamtego momentu rozwój serii 2.x nabrał rozpędu [13].

Kolejnym krokiem milowym w historii języka był Python 3.0 (zwany również *„Python 3000”* lub *„Py3k”*). Zaprojektowano go w celu wyeliminowania zgłoszonych błędów oraz duplikatów semantycznych w jego składni. Wprowadzenie tej wersji w grudniu 2008 roku zaburzyło kompatybilność języka z linią 2.x i wywołało zauważalny podział w społeczności. Obecnie jedyną linią rozwojową języka jest Python 3.x. Jej ostatnie wydanie miało miejsce 18 maja 2014 r. i zostało oznaczone numerem wersji Python 3.4.1. Gałąź 2.x nie jest już funkcjonalnie udoskonalana, ale na bieżąco aktualizowana pod kątem bezpieczeństwa.



Wyk 1. Historia popularności języka Python na przełomie ostatnich dwunastu lat.
Źródło: http://tiobe.com/

17 **Corporation for National Research Initiatives (CNRI)** - amerykańska organizacji non-profit, której głównym celem jest wspieranie rozwoju kluczowych technologii przetwarzania i udostępniania wiedzy z użyciem sieci komputerowych.

4.2 Wady i zalety Pythona

Python uznawany jest za czytelny, spójny, elastyczny i łatwy w użyciu język programowania. Dzięki regułom PEP8[18], wcięciom wyznaczającym bloki kodu, oraz pozbawionym klamrom ({, }), zapewnia estetykę kodu na najwyższym poziomie.

Programowanie w Pythonie kilkukrotnie zwiększa wydajność i produktywność programistów w porównaniu z językami kompilowanymi czy tymi ze statycznym typowaniem. Kod programu napisany w Pythonie jest znacznie mniejszy niżeli jego odpowiednik w C, C++, PHP czy Javie. Przekłada się to na mniejszą liczbę znaków do wpisania, którą to następnie trzeba sprawdzić i utrzymać w przyszłości.

Programy pisane w Pythonie działają natychmiast, bez konieczności długiej kompilacji i korzystania z narzędzi zewnętrznych co jeszcze bardziej zwiększa szybkość tworzenia kodu.

Standardowa implementacja Pythona (CPython) napisana jest w przenośnym ANSI C. W związku z tym, może być kompilowana praktycznie na każdej platformie będącej obecnie w użyciu. Python uruchamiany jest zarówno na mikrokontrolerach, komputerach stacjonarnych i przenośnych, urządzeniach mobilnych jak i na superkomputerach. Dostępny jest między innymi na systemy: Unix oraz GNU/Linux, Microsoft Windows i DOS, Mac OS (X i Classic), iOS, Symbian, Sailfish OS oraz Android.

Biblioteka standardowa Pythona stanowi ogromny zbiór wbudowanych i przenośnych modułów, bogatych w przydatne funkcje, obsługujące zadania programistyczne już na poziomie aplikacji. Liczba bibliotek zewnętrznych języka stale wzrasta, umożliwiając programiście tworzenia każdego typu oprogramowania.

Skrypty Pythona z łatwością komunikują się z innymi częściami aplikacji, wykorzystując do tego liczne wbudowane mechanizmy integracyjne. Zapewniają one możliwość wywoływania bibliotek języków C i C++, integrację z komponentami języków Java

18 **PEP8** - zdefiniowane reguły formatowania kodu i organizacji skryptów Pythona, http://legacy.python.org/dev/peps/pep-0008/.

i .NET, komunikację za pośrednictwem interfejsów SOAP, XML-RPC oraz REST. Python potrafi również sterować urządzeniami przez obsługę interfejsu RS-232.

Z pewnością silną cechą Pythona jest bardzo dobra dokumentacja języka[19] i dostępność bibliotek zewnętrznych z licznymi przykładami oraz wyjątkowo aktywna społeczność, zawsze skora do pomocy.

W związku z tym, że kod w Pythonie nie jest kompilowany do poziomu binarnego kodu maszynowego, a do kodu bajtowego, niektóre pisane w nim programy będą działały wolniej w porównaniu z aplikacjami napisanymi w języku w pełni kompilowanym. Z pewnością, trudno jest odczuć tę różnicę w przypadku przetwarzania plików czy konstruowania graficznego interfejsu użytkownika, ponieważ takie zadania są wewnątrz interpretera natychmiast wykonywane jako kod w C / C++. Jednakże, istnieją dziedziny dla których optymalizacja szybkości odgrywa kluczową rolę.

W programowaniu numerycznym bądź przy generowaniu animacji często wymagane jest, by najbardziej newralgiczne komponenty przetwarzały dane z prędkością programów w języku C (bądź lepszą). Przy takich zadaniach standardowa biblioteka Pythona nie zapewnia górnolotnej wydajności, dlatego należy posiłkować się bibliotekami wspierającymi obliczenia naukowe tj. NumPy / SciPy, bądź użyć alternatywnego interpretera PyPy[20]. Jeśli efekt nadal będzie niezadowalający, część aplikacji wymagającą optymalnej szybkości należy przedstawić jako kompilowane rozszerzenie np. języka Cython[21].

Za kolejną wadę standardowej implementacji Pythona można uznać brak możliwości współbieżnego wykonywania kodu przez wątki. CPython jest nieprzystosowany do pracy w trybie wielowątkowości, a to za sprawą użytej globalnej blokady interpretera (ang.

19 https://www.python.org/doc/

20 **PyPy** - ponad 6-krotnie szybszy od CPythona i mniej zasobożerny interpreter Pythona ze zintegrowanym kompilatorem JIT.

21 **Cython** - język programowania oparty na Pyrex, stanowi nadzbiór Pythona, wzbogacony o dodatkową składnię. Umożliwia deklarację typów zmiennych, atrybutów klas oraz wywoływanie funkcji języka C. Dzięki temu kompilator generuje wydajny kod C, co sprawia, że Cython staje się idealny do tworzenia zewnętrznych rozszerzeń Pythona.

Global Interpreter Lock, GIL)[22]. GIL zmniejsza konkurencję pomiędzy wątkami tego samego procesu interpretera. Powoduje to jednak brak wyraźnego zwiększenia wydajności programu, uruchomionego na komputerze wieloprocesorowym.

Jednakże, zwolennicy GIL argumentują, że jego obecność nie jest przypadkowa. Realizuje on koncepcję kooperatywnej wielozadaniowości. Dzięki temu interpreter najlepiej wie kiedy bezpiecznie przełączyć kontekst wykonania. Ponadto, dość często rozwiązanie z GIL jest bardziej efektywne od wielozadaniowości z wywłaszczeniem.

4.3 Zastosowanie Pythona

Python jest językiem programowania ogólnego przeznaczenia. Sprawdza się doskonale w wykonywaniu prawdziwych zadań, z jakimi na co dzień mają styczność programiści. Jest używany w wielu różnych dziedzinach jako narzędzie do tworzenia skryptów dla innych komponentów, a także służące do implementowania samodzielnych programów.

Dziedziny w jakich Python jest wykorzystywany, dzielą się na kilka ogólnych kategorii:

- Programowanie systemowe,
- Aplikacje z graficznym interfejsem użytkownika (GUI), w tym te ukierunkowane na technologię multi-touch,
- Skrypty internetowe i programowanie aplikacji webowych,
- Integracja komponentów,
- Programowanie bazodanowe,
- Obliczenia numeryczne i inżynierskie,
- Grafika 2D/3D, porty szeregowe, USB, IrDA, Bluetooth,
- Przetwarzanie CSV, HTML, XML, JSON,
- Analiza języka naturalnego,

22 http://en.wikipedia.org/wiki/Global_Interpreter_Lock

- Robotyka i automatyka,
- Edukacja.

Powszechnie uważa się, że Python wykorzystywany jest przede wszystkim przez sympatyków ruchu Wolnego i Otwartego Oprogramowania oraz do tworzenia projektów i firm typu startup. Oczywiście tak twierdzą wyłącznie złośliwi, a na dowód tego przedstawię krótką listę najważniejszych korporacyjnych gigantów, które od lat z Pythonem mają wiele wspólnego:

- Google już w 2005 roku zaczął intensywnie wykorzystywać Pythona, tym samym pozyskując w swoje szeregi samego twórcę języka - Guido Van Rossum. Najbardziej popularny serwis służący do dzielenia się filmami video - YouTube, jest w większości napisany w tym języku. Popularna platforma programowania aplikacji webowych Google App Engine wprowadziła Pythona jako pierwszego w roli dostępnej technologii,
- Intel, Cisco, Hewlett-Packard, Seagate czy IBM wykorzystują Pythona do testowania swoich urządzeń,
- Facebook, Instagram, Pinterest, Dropbox czy RedHat, stosują go zarówno do procesów wewnętrznych wspomagając pracę innych usług jak i do realizacji kompleksowych serwisów internetowych czy programów klienckich,
- Industrial Light & Magic, Pixar i inne wykorzystują Pythona w tworzeniu filmów animowanych,
- Instytucie takie jak NASA, Los Alamos, AstraZeneca, NOAA, DARPA czy Departament Energii Stanów Zjednoczonych, wykorzystują Pythona do zadań programistycznych o charakterze naukowym i symulacyjnym,
- Agencja NSA wykorzystuje Pythona w kryptografii oraz analizach wywiadowczych,
- Nowojorska Giełda Papierów Wartościowych wykorzystuje Pythona jako fundament webowego systemu transakcji,

- Python stanowi nieodłączną część wielu dystrybucji Unix i GNU/Linux, czy oprogramowania obsługującego mikrokomputery Raspbbery Pi.

4.4 Python 2 a Python 3

Z początkiem grudnia 2008 roku wydany został Python 3.0, zwany również - ze względu na zakres zmian i unowocześnień - Pythonem 3000 (bądź „Py3k”). Sami autorzy języka podkreślają, że dawka nowości jest bezprecedensowa.

By uporządkować Pythona, a zarazem otworzyć nowe możliwości jego rozwoju, deweloperzy świadomie zerwali z kompatybilnością wsteczną, zarówno pod kątem składni języka, jak i jego biblioteki standardowej. Tak głębokie zmiany wywołały szereg dyskusji, przy okazji których doszło do podziału w społeczności na zwolenników i przeciwników rewolucji.

Obecnie Python 3.x stanowi jedyną linię rozwojową języka. Seria 2.x nie jest już technologicznie ulepszana, aczkolwiek w momencie pojawienia się nowych poprawek bezpieczeństwa, błyskawicznie wprowadzane są kolejne aktualizacje języka.

Python 3 jest stabilny i gotowy do produkcyjnych wdrożeń. Stanowi domyślną wersję języka w wielu dystrybucjach Unix i GNU/Linux. Liczba dostępnych bibliotek zewnętrznych[23] go obsługujących ciągle rośnie.

4.5 Alternatywne implementacje Pythona

Istnieją cztery najważniejsze implementacje języka programowania Python - CPython, Jython, IronPython oraz PyPy. CPython to standardowa, najczęściej używana implementacja. Pozostałe mają swoje ściśle określone cele i role, a także wady i zalety. Wszystkie wdrażają ten sam język, jednak w inny sposób wykonują programy.

23 https://python3wos.appspot.com/

4.5.1 CPython

Oryginalna i standardowa implementacja Pythona[24], napisana w przenośnym kodzie w języku ANSI C. Dokładnie tę wersję można pobrać z oficjalnej strony języka bądź znaleźć w większości komputerów z zainstalowanym systemem Unix, GNU/Linux czy Mac OS X. Ponieważ jest to referencyjna implementacja, najczęściej jest też najszybsza, najbardziej kompletna i ma największe możliwości w porównaniu z systemami alternatywnymi. Jak już zostało wcześniej wspominałem, CPython umożliwia integrację z językami C i C++, posiada także wiele bibliotek zewnętrznych.

4.5.2 Jython

Jython[25] to Python napisany całkowicie w Javie, skierowany na integrację z tym językiem. Implementacja ta składa się z javowych klas kompilujących kod źródłowy Pythona do kodu bajtowego Javy, który jest przekierowywany do maszyny wirtualnej Javy (Java Virtual Machine, JVM). Przełożony w ten sposób pythonowy kod wygląda i zachowuje się w czasie wykonywania zupełnie jak prawdziwy program napisany w Javie.

Celem Jythona jest przede wszystkim danie programistom możliwości tworzenia w Pythonie skryptów dla aplikacji napisanych w Javie. Skrypty Jythona mogą służyć jako aplety webowe, serwlety czy tworzyć GUI oparte na Javie (SWING, SWT, Qt Jambi). Co więcej, Jython umożliwia również importowanie i wykorzystywanie klas oraz bibliotek Javy w kodzie napisanym w Pythonie. Jest on jednak wolniejszy i skromniejszy wyglądem standardowej implementacji CPython.

Najnowsza wersja Jythona oznaczona została numerem 2.7.0b3. Jej finalne wydanie zgodne z CPythonem 2.7 planowane jest pod koniec 2014 roku. Frank Wierzbicki, osoba kierująca obecnie rozwojem Jythona, rozpoczął również pracę nad Jythonem 3000, zmierzając w kierunku obsługi Pythona 3.x.

---

24 http://python.org/

25 http://www.jython.org/

4.5.3 IronPython

IronPython[26] to powstała na początku 2003 roku implementacja języka Python w środowisku CLR platformy .NET i Mono firmy Microsoft. Stworzona w języku C#, zapewnia dostęp zarówno do olbrzymiej liczby standardowych i zewnętrznych bibliotek platformy .NET Framework (np. Presentation Foundation, Silverlight), jak i samego języka Python, dając programiście szeroki wachlarz możliwości tworzenia oprogramowania przeznaczonego głównie dla Windowsa.

Rozwój IronPythona w znacznym stopniu przyczynił się do powstania uniwersalnego środowiska uruchomieniowego DLR, odpowiedzialnego za wydajne wykonywanie dynamicznych języków programowania. Wskutek tego, szybkość działania tej implementacji jest porównywalna do standardowej implementacji Pythona (CPython).

IronPython dostępny jest na licencji Apache 2.0. IronPython posiada wsparcie we flagowym środowisku programistycznym firmy Microsoft, czyli Visual Studio. Udostępnia także własny kompilator `pyc.py`. Funkcjonalności te zapewnia narzędzie o nazwie Python Tools for Visual Studio[27], które w momencie pisania niniejszej pracy dostępne jest w wersji 2.1RC. Natomiast sam IronPython oznaczony jest numerem wersji 2.7.4, zachowując tym samym zgodność z CPythonem 2.7.

4.5.4 PyPy

W cieniu wielu zalet języka programowania Python, programiści coraz częściej dostrzegają fakt, iż prędkość działania standardowej, najpopularniejszej implementacji, CPython, nie zawsze dorównuje wydajności języków kompilowanych. W przypadku, gdy priorytetem staje się prędkość działania algorytmów zdrowy rozsądek podpowiada, aby wykorzystać wszelkie dostępne mechanizmy optymalizacji języka lub zmienić interpreter.

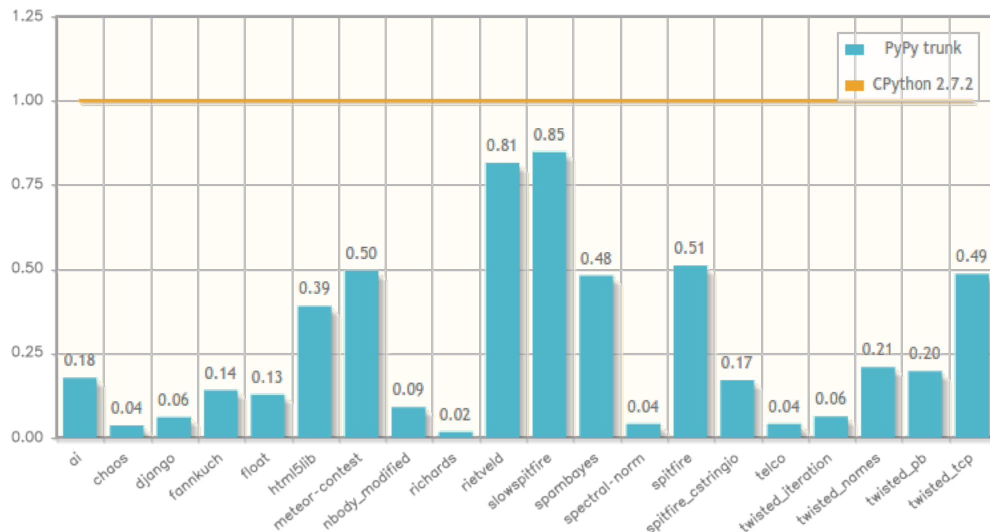
Alternatywnym sposobem na analizowanie i wykonywanie kodu Pythona jest projekt PyPy dostępny na zasadach licencji MIT. Osiąga on lepsze wyniki niż wspomniany CPython, bez konieczności korzystania z C, czy niskopoziomowych języków. Za wydajność

26 http://ironpython.net/

27 http://pytools.codeplex.com/

i stabilność odpowiada zintegrowany kompilator JIT[28]. W efekcie końcowym, daje to średnio 6.5x przyspieszenie w stosunku do aktualnej wersji standardowej implementacji Pythona.

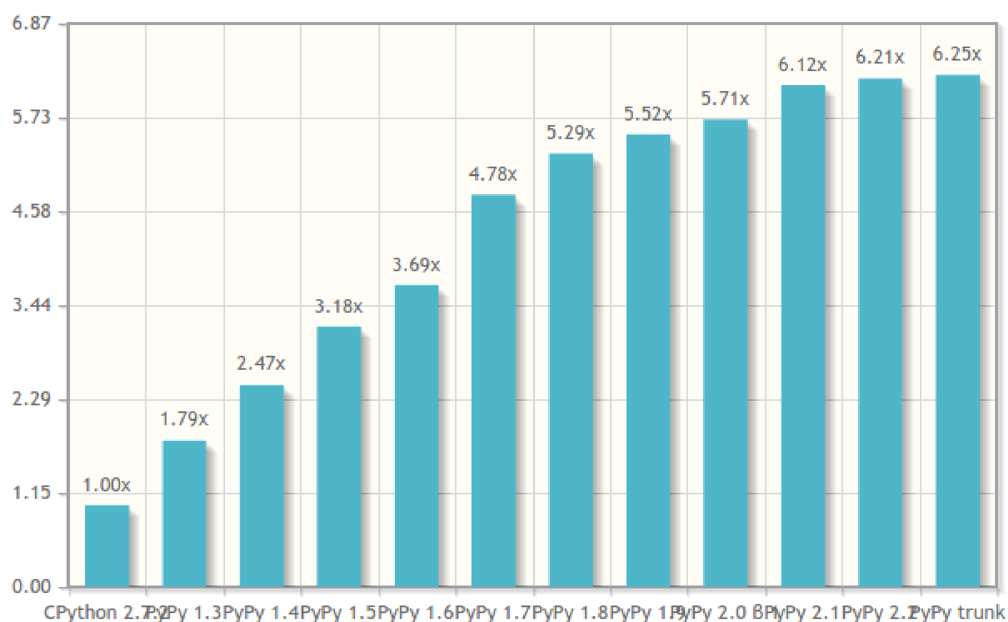
W celu dokładniejszego zobrazowania tego zjawiska, umieszczam poniżej dwa wykresy przedstawiające znormalizowane pomiary wydajnościowe i czasowe obu tych implementacji języka.



Wyk 2. Znormalizowane testy szybkościowe względem rozwojowej wersji PyPy a CPython 2.7.2 na przykładzie wybranych bibliotek. Źródło: http://speed.pypy.org/

Warto również wspomnieć, że deweloperzy PyPy pracują obecnie nad kompatybilnością interpretera z Pythonem 3 i możliwością korzystania w tym środowisku z biblioteki NumPy, wspierającej obliczenia naukowe w języku programowania Python. W tym celu, twórcy PyPy otrzymali we wrześniu 2014 roku kolejną dotację od Python Software Foundation, której wysokość może sięgnąć nawet $10,000.

28 **Just-In-Time (JIT)** - metoda wykonywania programów polegająca na kompilacji do kodu maszynowego przed wykonaniem danego fragmentu kodu.



Wyk 3. Przedstawienie wyników pomiarów uzyskanego przyspieszenia dla kolejnych wersji wydawniczych interpretera PyPy względem Pythona 2.7.2. Źródło: http://speed.pypy.org/

4.6 Wykonywanie kodu w Pythonie

W momencie, gdy wykonywane są w Pythonie (CPythonie) skrypty, dochodzi do kompilacji z kodu źródłowego do postaci kodu bajtowego, który zostaje podzielony na grupy instrukcji pojedynczych kroków. Jeśli proces Pythona ma uprawienia do zapisu na komputerze, kod bajtowy programów zostanie zapisany w plikach o rozszerzeniach `.pyc`, które oznaczają skompilowane źródło `.py`.

4.6.1 Kod bajtowy Pythona

Python zapisuje kod bajtowy w celu optymalizacji szybkości wykonania. Następnym razem, kiedy będzie wykonywać program, zostaną załadowane pliki `.pyc` i pominięty etap kompilacji - o ile oczywiście nie został zmieniony kod źródłowy programu. Python automatycznie sprawdza czas zapisu plików i na tej podstawie decyduje o ponownej kompilacji. W przypadku braku możliwości zapisu kodu bajtowego na naszym

komputerze, program nadal będzie działał - kod bajtowy będzie tworzony w pamięci i po prostu usuwany po zakończeniu działania aplikacji. Ponieważ jednak pliki `.pyc` przyspieszają rozpoczęcie wykonywania programu, w przypadku większych projektów lepiej jest upewnić się, że są one zapisywane na dysku twardym.

Pliki kodu bajtowego to także jeden ze sposobów na publikowanie programów napisanych w Pythonie. Interpreter z powodzeniem uruchomi program składający się z samych plików `.pyc`, nawet kiedy oryginalne pliki źródłowe są niedostępne.

Poniżej prezentuję pythonowy kod (Python 3.4.1) z przykładową funkcją `test` i jej reprezentację w kodzie bajtowym:

```python
def test(a=1, *args, b=3):
    print(args)

    c = None

    return a + b, c
```

Listing 1. Kod źródłowy pliku test.py.

```
  2           0 LOAD_GLOBAL              0 (print)
              3 LOAD_FAST                2 (args)
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  4          10 LOAD_CONST               0 (None)
             13 STORE_FAST               3 (c)

  6          16 LOAD_FAST                0 (a)
             19 LOAD_FAST                1 (b)
             22 BINARY_ADD
             23 LOAD_FAST                3 (c)
             26 BUILD_TUPLE              2
             29 RETURN_VALUE
```

Listing 2. Reprezentacja kodu bajtowego funkcji test dla pliku test.py.

4.6.2 Zoptymalizowany kod bajtowy Pythona

Uruchomienie interpretera Pythona z flagą `-O`, wygeneruje zoptymalizowany kod bajtowy skryptu, w postaci pliku `.pyo`.

Różnica w wydajności jest praktycznie niezauważalna, gdyż dodanie tego przełącznika usuwa z kodu wyłącznie wyrażenie asercji (assert). Podwójna flaga `-OO`, w niektórych przypadkach może spowodować nieprawidłowe działanie programów. Obecnie, usuwa ona z kodu bajtowego łańcuchy `__doc__` (tzw. doc-stringi), w wyniku czego staje się on bardziej kompaktowy.

W obu tych przypadkach programy przez nas uruchamiane nie zyskają na szybkości w działaniu, a wyłącznie skrócą czas ładowania plików `.pyc` lub `.pyo` i zyskają kilka KB na wielkości.

W związku z tym, że Python generuje kod bajtowy wyłącznie dla plików importowanych, pomijając te najwyższego poziomu w programie, twórcy języka udostępnili moduł o nazwie `compileall`, który jest w stanie wygenerować plik `.pyc` lub `.pyo` dla dowolnego katalogu ze skryptami bądź pojedynczego pliku zawierającego kod źródłowy naszej aplikacji. Użycie `compileall` można również wymusić z poziomu wiersza poleceń:

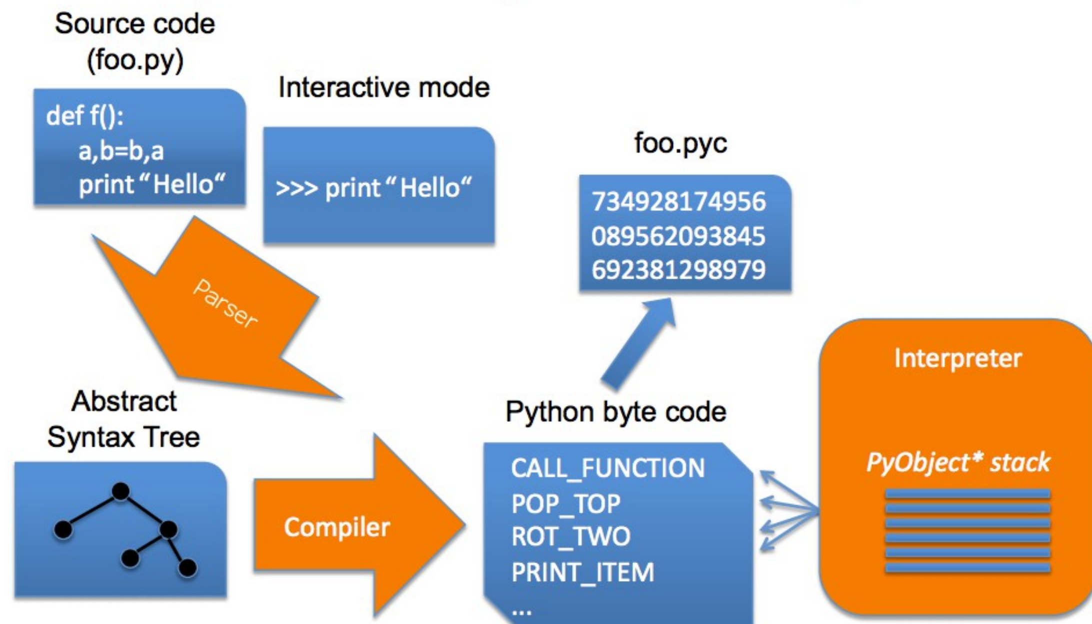
```
python -OO -m compileall nazwa_skryptu.py
```

4.6.3 Python Virtual Machine

Po skompilowaniu programu do kodu bajtowego (lub załadowaniu z istniejących plików `.pyc` / `.pyo`) jest on przesyłany do wykonania do Python Virtual Machine (PVM). Maszyna wirtualna Pythona, to tak naprawdę „wielka pętla", która przechodzi przez instrukcje kodu bajtowego, jedna po drugiej, w celu wykonania działań i operacji z nimi związanych. Stanowi ona silnik wykonawczy (ang. runtime engine) tego języka. Jest zawsze obecna jako część systemu Pythona (komponent), który odpowiedzialny jest za samo wykonywanie, skryptów. Z technicznego punktu widzenia jest ostatnim etapem interpretera Pythona.

Rysunek 1. przestawia strukturę wykonawczą Pythona. Należy pamiętać, że cały ten stopień skomplikowania jest niewidoczny dla programistów. Tradycyjny model wykonywania kodu Pythona ogranicza się do dwóch zasadniczych etapów. Pisany przez programistę kod źródłowy Pythona (plik `.py`) przekładany jest na kod bajtowy

(plik `.pyc` / `.pyo`), który to następnie wykonywany jest przez maszynę wirtualną Pythona (PVM). Kod jest kompilowany automatycznie, a następnie interpretowany.



Rys 1. Tradycyjny model wykonawczy kodu Pythona 2.7. Źródło: http://huangx.in/

4.6.4 Struktura pliku .pyc

Binarny pliki `.pyc` składa się z fragmentów [9] [10]:

- pierwsze cztery bajty to tzw. „magic number”, czyli wartość określająca wersję Pythona, której użyto do kompilacji kodu; Tabela 1. zawiera zestawienie wartości reprezentujących poszczególne wersje interpreterów,
- kolejne cztery bajty to data modyfikacji pliku źródłowego zapisana w postaci Unix timestamp[29],
- następne cztery bajty to nic niewnosząca sekwencja 0 (występuje tylko w Pythonie 3),

29 http://en.wikipedia.org/wiki/Unix_time

- pozostała część pliku to serializowane obiekty Pythona przy użyciu wbudowanego w język modułu `marshal`[30].

<table>
<tr><th>Wersja interpretera Python</th><th>Magic number</th></tr>
<tr><td>Python 1.5</td><td>20121</td></tr>
<tr><td>Python 1.5.1</td><td>20121</td></tr>
<tr><td>Python 1.5.2</td><td>50428</td></tr>
<tr><td>Python 1.6</td><td>50428</td></tr>
<tr><td>Python 2.0</td><td>50823</td></tr>
<tr><td>Python 2.0.1</td><td>50823</td></tr>
<tr><td>Python 2.1</td><td>60202</td></tr>
<tr><td>Python 2.1.1</td><td>60202</td></tr>
<tr><td>Python 2.1.2</td><td>60202</td></tr>
<tr><td>Python 2.2</td><td>60717</td></tr>
<tr><td rowspan="2">Python 2.3a0</td><td>62011</td></tr>
<tr><td>62021</td></tr>
<tr><td>Python 2.4a0</td><td>62041</td></tr>
<tr><td>Python 2.4a3</td><td>62051</td></tr>
<tr><td>Python 2.4b1</td><td>62061</td></tr>
<tr><td rowspan="4">Python 2.5a0</td><td>62071</td></tr>
<tr><td>62081</td></tr>
<tr><td>62091</td></tr>
<tr><td>62092</td></tr>
<tr><td rowspan="2">Python 2.5b3</td><td>62101</td></tr>
<tr><td>62111</td></tr>
<tr><td>Python 2.5c1</td><td>62121</td></tr>
<tr><td>Python 2.5c2</td><td>62131</td></tr>
<tr><td>Python 2.6a0</td><td>62151</td></tr>
<tr><td>Python 2.6a1</td><td>62161</td></tr>
<tr><td rowspan="4">Python 2.7a0</td><td>62171</td></tr>
<tr><td>62181</td></tr>
<tr><td>62191</td></tr>
<tr><td>62201</td></tr>
</table>

30 https://docs.python.org/2/library/marshal.html

	62211
Python 3000	3000
	3010
	3020
	3030
	3040
	3050
	3060
	3061
	3071
	3081
	3091
	3101
	3103
Python 3.0a4	3111
Python 3.0a5	3131
Python 3.1a0	3141
	3151
Python 3.2a0	3160
Python 3.2a1	3170
Python 3.2a2	3180
Python 3.3a0	3190
	3200
	3210
Python 3.3a1	3220
Python 3.3a4	3230
Python 3.4a1	3250
	3260
	3270
	3280
Python 3.4a4	3290
	3300
Python 3.4rc2	3310
Python 3.5a0	3320

Tab 1. Zestawienie wartości „magic number” odpowiadających poszczególnym wersjom interpretera Pythona.

Wykorzystywany do serializacji moduł `marshal` działa podobnie do modułu `pickle`. Różnica polega na tym, że `pickle` zapisuje stan obiektów Pythona, włącznie z tymi zdefiniowanymi przez użytkownika, z kolei `marshal` ogranicza się do tych reprezentowanych przez wewnętrzne typy danych języka. Warto dodać, że `marshal` jest przy tym kilkanaście razy szybszy od `pickle`.

Ustalenie wersji Pythona użytej do kompilacji kodu źródłowego do postaci kodu bajtowego jest niezbędne, w celu przeprowadzenia dalszych procesów związanych z inżynierią wsteczną oprogramowania.

5. Narzędzia zamrażające Pythona

Społeczność Pythona stworzyła narzędzia umożliwiające pakowanie skryptów tego języka do postaci samodzielnych plików wykonywalnych, nazywanych potocznie „zamrożonymi plikami binarnymi” (ang. frozen binaries). Dystrybucja Pythona w takiej postaci, zwalnia użytkowników komputerów z konieczności posiadania zainstalowanego środowiska uruchomieniowego i bibliotek zewnętrznych języka w systemie. Co więcej, ponieważ sam kod programu osadzony jest w pliku binarnym, w rezultacie jest on ukryty przed osobami niepowołanymi. To rozwiązanie jest w szczególności atrakcyjne dla twórców oprogramowania komercyjnego.

Do tworzenia zamrożonych plików wykonywalnych wykorzystuje się darmowe exe-packery, tj. py2exe, py2app, cx_Freeze, bbfreeze czy PyInstaller. Narzędzia te trzeba pobrać niezależnie od Pythona [3].

5.1 Czym są „frozen binaries”?

Zamrożone pliki binarne obejmują kod bajtowy skryptów Pythona wraz z jego maszyną wirtualną (interpreterem) i wszelkimi plikami multimedialnymi, niezbędnymi do działania programu. Istnieje kilka odmian tej koncepcji, jednak rezultatem końcowym najczęściej jest pojedynczy binarny program wykonywalny (na przykład plik `.exe` w przypadku systemu Windows), który można w łatwy sposób udostępniać użytkownikom.

Zamrożone pliki binarne zasadniczo działają z szybkością zbliżoną do oryginalnych plików źródłowych. Ich rozmiar, z tytułu dołączonej maszyny wirtualnej Pythona i bibliotek dynamicznych, sięga od kilku do kilkudziesięciu MB.

By móc przeprowadzić inżynierię wsteczną oprogramowania zamrożonych plików binarnych Pythona, ustalę najpierw sposób działania exe-packerów - czyli procesu budowania programów wykonywalnych.

5.2 py2exe

Standardowa biblioteka Pythona zawiera pakiet `distutils`[31], który służy do kompilowania modułów i rozszerzeń języka, jak również instalacji zewnętrznych pakietów w środowisku. Stanowi ona ustandaryzowany system dystrybucji umożliwiający instalowanie, rozwijanie i udostępnianie pythonowego kodu.

py2exe[32] rozszerza możliwości pakietu `distutils`, umożliwiając tworzenie ze skryptów Pythona samodzielnych plików wykonywalnych (32-bitowych oraz 64-bitowych) dla systemów z rodziny Windows. py2exe rozwijane jest w ramach dwóch gałęzi:

- py2exe 0.6.x - przeznaczona do obsługi Pythona 2.4 - 2.7,
- py2exe 0.9.x - przeznaczona do obsługi Pythona 3.3 - 3.4.

Użycie py2exe 0.6.x sprowadza się do stworzenia pythonowego skryptu `build.py`, którego kod definiuje konfigurację dla procesu budowania i przekazuje ją do metody `setup` z pakietu `distutils`.

Kod skryptu `build.py` umożliwia zdefiniowanie m.in:

- głównego skryptu Pythona podlegającego kompilacji,
- typu kompilowanej aplikacji (konsolowa, okienkowa - GUI),
- zależności w kompilowanym programie (tj. zewnętrzne biblioteki Pythona lub pliki multimedialne),
- metadanych i ikony dla pliku wykonywalnego,
- poziomu optymalizacji rozmiaru pliku wynikowego.

Wszystkie możliwe opcje konfiguracji procesu kompilacji zostały wylistowane i opisane na stronie projektu w artykule „List of options”[33].

31 https://docs.python.org/2/distutils/

32 http://www.py2exe.org/

33 http://www.py2exe.org/index.cgi/ListOfOptions

Tak przygotowany skrypt należy uruchomić z poziomu wiersza poleceń (CMD):

```
python build.py py2exe
```

W wyniku działania powyższego polecenia, konsola wyświetla listing informacji z postępu realizowanej kompilacji. W międzyczasie powstają dwa katalogi, `build` i `dist`. Pierwszy jest używany w trakcie budowania binariów jako przestrzeń robocza, z kolei w drugim dostępny jest właściwy program wykonywalny `.exe`, gotowy do użycia i dystrybucji.

Katalog `dist`, poza plikiem uruchomieniowym, którego nazwa odpowiada kompilowanemu skryptowi, w zależności od konfiguracji pliku `build.py`, może również zawierać inne pliki, niezbędne do działa binariów. Uwzględnia je tabela 2 [5].

Nazwa pliku	Opis
python27.dll[34]	Biblioteka interpretera języka Python 2.7.
library.zip[35]	Archiwum ZIP zawierające wykorzystywane pythonowe moduły w postaci plików `.pyc` (kodu bajtowego). Są to zarówno moduły standardowej biblioteki Pythona jak i te stworzone przez użytkownika.
*.pyd	Pliki `.pyd`, czyli biblioteki skompilowanych modułów Pythona.
*.dll	Pliki .pyd posiadają zależności z bibliotekami `.dll`, stąd możliwa jest ich obecność.
w9xpopen.exe	Aplikacja wymagana do poprawnego uruchamiania programu w środowisku Win9x (Windows 95, Windows 98, Windows ME).

Tab 2. Zestawienie dodatkowych plików z katalogu `dist`, po przeprowadzonej kompilacji z użyciem py2exe.

34 Nazwa pliku zależna jest od użytej do kompilacji wersji Pythona. W tym wypadku byłby to Python 2.7.x.

35 Jest to domyślna nazwa pliku, jednakże użytkownik może określić własną nazwę. Co więcej, archiwum wcale nie musi być dostępne w postaci odrębnego pliku `.zip`. Użytkownik może zdecydować na umieszczenie go wewnątrz pliku uruchomieniowego `.exe`.

5.3 py2app

py2app[36] podobnie jak py2exe stanowi dostępne na licencji MIT rozszerzenie pakietu `distutils`, z tym że służy do budowania samodzielnych aplikacji ze skryptów Pythona dla systemów Mac OS X. py2app jest dystrybuowany z PyObjC[37], „mostem" pomiędzy Objective-C a Pythonem, oferującym doskonały sposób na tworzenie aplikacji na tę platformę z użyciem interfejsu Cocoa w języku Python. py2app pozwala również na budowanie pakietów instalacyjnych - plików .mpkg.

Najnowsza wersja py2app została oznaczona numerem wersji 0.9.

W tym przypadku, kompilacja skryptów Pythona do postaci wykonalnej jest analogiczna do py2exe. Odpowiednio skonstruowany i uruchomiony skrypt `build.py`, tworzy katalogi `build` i `dist` z plikiem binarnym `.app`.

Sposób wywołania `build.py` przedstawia się następująco:

```
python build.py py2app
```

5.4 cx_Freeze

cx_Freeze[38] to zbiór skryptów i modułów wykorzystywanych do tworzenia aplikacji wykonywalnych ze skryptów języka programowania Python. Działa w sposób zbliżony do opisywanych powyżej py2exe i py2app. Ten projekt wyróżnia to, że jest wieloplatformowy - obsługuje zarazem kompilację pod kątem systemów Windows, Linux jak i Mac OS X. cx_Freeze dystrybuowany jest na otwartej licencji PSF.

Najnowsza wersja cx_Freeze została oznaczona numerem wersji 4.3.3 i obsługuje Pythona 2.6 - 3.4. Z kolei, wcześniejsze wersje zapewniają wsparcie dla Pythona 2.3 - 2.5.

36 https://pythonhosted.org/py2app/

37 https://pythonhosted.org/pyobjc/

38 http://cx-freeze.readthedocs.org/en/latest/

cx_Freeze można użyć na trzy sposoby:

1. Pierwszy polega na skorzystaniu z dołączonego narzędzia `cxfreeze`, przeznaczonego do budowania prostych programów.
2. Jako drugą metodę, należy uznać opisywane już tworzenie pliku `build.py`, zawierającego konfigurację procesu budowy.

   Sposób wywołania `build.py` przedstawia się następująco:

   ```
   python build.py build
   ```

3. Ostatnia polega na bezpośredniej pracy z klasami i modułami używanymi wewnętrznie przez cx_Freeze.

Każda z metod tworzy domyślnie katalog `dist`, zawierający plik wykonywalny programu, archiwum `.zip` oraz wymagane biblioteki `.pyd`, `.dll` lub `.so`.

Ponadto, cx_Freeze posiada bardzo przydatną funkcjonalność, umożliwiającą pakowanie skompilowanej aplikacji od razu do postaci programu instalacyjnego `.msi` lub `.dmg`, co jest niezmiernie przydatne w przypadku oprogramowania ukierunkowanego na dystrybucję.

5.5 bbfreeze

bbfreeze[39] jest exe-packerem bazującym na cx_Freeze. Dodatkowo umożliwia dołączanie do binariów plików `.egg` (rzadko używany format pythonowych pakietów) oraz jednoczesną kompilację kilku skryptów Pythona w oparciu o jedno środowisko uruchomieniowe. Pakiet udostępniany jest na licencji MIT.

bbfreeze wykorzystywany jest wyłącznie do tworzenia binariów dla systemów Unix oraz Windows. Obsługuje wersje Pythona 2.4 - 2.7.

39 https://pypi.python.org/pypi/bbfreeze/

bbreeze można użyć na trzy sposoby:

1. Pierwszy z nich polega na wywołaniu (dostarczonego w ramach pakietu) binarnego programu `bbfreeze` na skrypcie Pythona, który mam „zamrozić”:

```
bbfreeze nazwa_skryptu.py
```

2. Drugi sposób użycia sprowadza się do wywołania wcześniej przygotowanego skryptu konfiguracyjnego `build.py`:

```
python build.py bdist_bbfreeze
```

3. Ostatni sprowadza się do wykorzystania udostępnianego API, za pomocą którego można przeprowadzać zaawansowane operacje, w wyniku których otrzymuję plik binarny,

Należy nadmienić, że pakiet bbfreeze nie jest już rozwijany, a jego ostatnia wersja została wydana 20 stycznia 2014 r i oznaczona numerem wersji 1.1.3.

5.6 PyInstaller

PyInstaller[40] to najbardziej popularne narzędzie „do zamrażania” skryptów Pythona do postaci samodzielnych plików binarnych. Dostępne jest na platformy Windows, Linux, Mac OS X, Solaris oraz AIX. Współpracuje z wersjami języka Python 2.2 - 2.7. Narzędzie objęte jest licencją GPLv2.

Na tle wyżej zaprezentowanych exe-packerów, PyInstaller wyróżniają następujące funkcjonalności:

- budowanie plików wykonalnych o znacznie mniejszym rozmiarze - dzięki zastosowaniu przezroczystej kompresji,
- wykorzystanie wsparcia systemu operacyjnego do ładowania bibliotek współdzielonych (DLL),

40 http://www.pyinstaller.org/

- tworzenie całkowicie samodzielnych, pojedynczych plików wykonalnych,
- automatyczne wsparcie dla binarnych bibliotek używanych przez `ctypes`,
- możliwość zastosowania binarnej kompresji przez UPX[41].

Ponadto, PyInstaller ceniony jest za wbudowany mechanizm automatyzujący proces tworzenia binariów i dołączania do nich zależności.

PyInstaller można użyć na dwa sposoby:

1. Pierwszy z nich polega na wywołaniu (dostarczonego w ramach pakietu) binarnego programu `pyinstaller` na skrypcie Pythona, który „zamrażam”:

   ```
   pyinstaller nazwa_skryptu.py
   ```

2. Z kolei, drugi sposób użycia sprowadza się do stworzenia pliku specyfikacji `.spec` - zgodnie z wytycznymi dokumentacji PyInstaller[42], oraz przekazania go do binarnego programu `pyinstaller`:

   ```
   pyinstaller plik_specyfikacji.spec
   ```

W obu wywołaniach, na ekranie konsoli pojawiają się logi z opisem poszczególnych kroków budowy programu binarnego. Kolejno tworzone są katalogi `build` i `dist`, i tak jak we wcześniejszych przypadkach, `build` zawiera dane tymczasowe - wykorzystywane podczas kompilacji, a `dist` katalog z programem wykonywalnym i niezbędnymi bibliotekami `.pyd`, `.dll` lub `.so`.

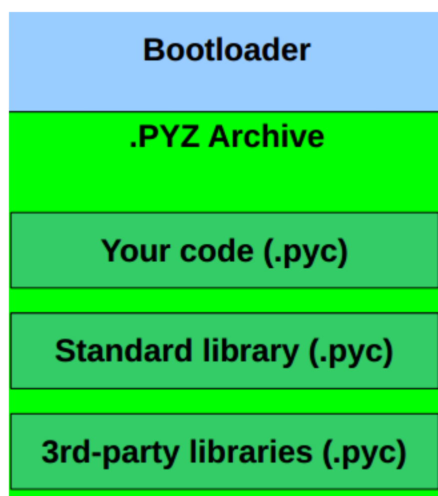
Użycie programu `pyinstaller` na skrypcie Pythona powoduje wygenerowanie specyfikacji kompilacji `.spec` opisującej konfigurację, w ramach której dokonano stworzenia binariów. Specyfikację tę można poddać modyfikacji i wykorzystać do

41 **UPX (ang. Ultimate Packer for eXecutables)** - rozpowszechniany na licencji GNU GPL program komputerowy do kompresji plików wykonywalnych.

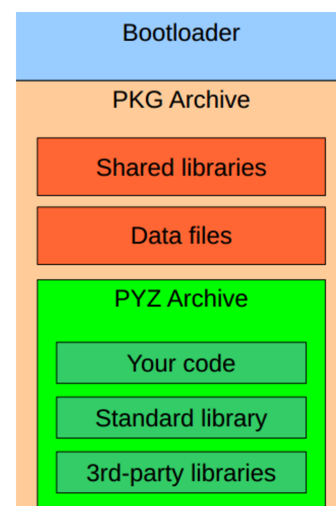
42 http://pythonhosted.org/PyInstaller/

ponownej kompilacji, w celu określenia dodatkowych zależności, wykluczenia zbędnych modułów biblioteki standardowej Pythona lub aby uwzględnić zewnętrzne pliki multimedialne.

PyInstaller do budowania natywnych binariów wykorzystuje autorski algorytm organizacji ich wewnętrznej struktury. W związku z tym, że umożliwia on także tworzenie samodzielnych plików binarnych, czyli takich które zawierają wewnątrz siebie wszelkie zależności programu, plik wynikowy może mieć zróżnicowaną strukturę wewnętrzną, co prezentują rysunki 2 i 3.



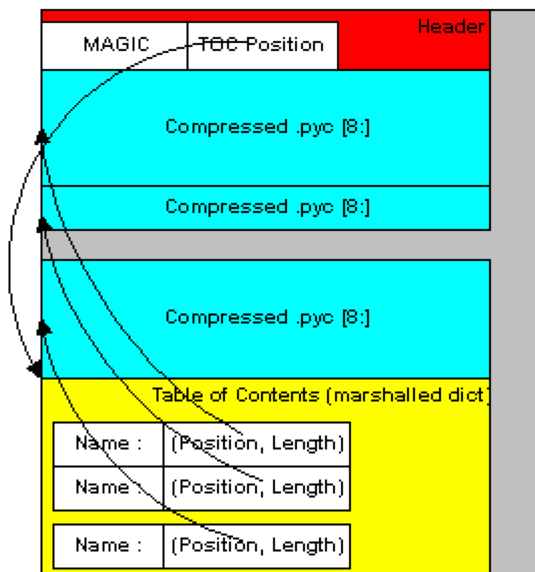
Rys 2. Wnętrze pliku binarnego powstałego przy użyciu PyInstaller w wariancie domyślnym. Źródło: http://goo.gl/vKwWbT (slajd 18)



Rys 3. Wnętrze pliku binarnego powstałego przy użyciu PyInstaller w wariancie pojedynczego pliku wykonywalnego. Źródło: http://goo.gl/vKwWbT (slajd 24)

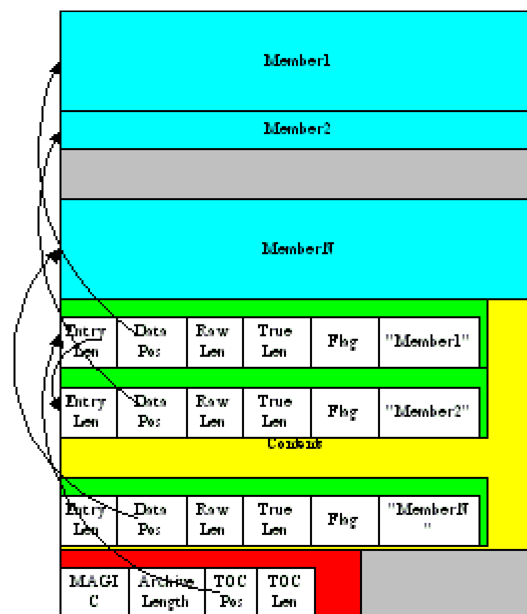
Jak widać na obrazkach, wariant z pojedynczym plikiem binarnym obejmuje dodatkową archiwizację PKG, czyli wykorzystanie ZlibArchive (PYZ) i CArchive [4].

ZlibArchive (PYZ) to format pliku, zawierający skompresowaną postać pythonowego kodu bajtowego (`.pyc`, `.pyo`). Do ich archiwizacji wykorzystuje się wspomniany w niniejszej pracy moduł `marshal`. CArchive uwzględnia pozostałe pliki. Od strony architektury, przypomina typowe archiwum `.zip`.



Rys 4. Struktura archiwum ZlibArchive.

Źródło: http://pyinstaller.org/



Rys 5. Struktura archiwum CArchive.

Źródło: http://pyinstaller.org/

Wariant z pakowaniem wszystkich plików do pojedynczego binariów wydaje się być wygodnym rozwiązaniem dla małych aplikacji. Jednakże, wpływa niekorzystnie na czas jej uruchomienia. Dzieje się tak, gdyż plik ten jest najpierw dekompresowany do katalogu tymczasowego, a następnie uruchamiany.

Uruchomienie binariów powstałych na wskutek użycia PyInstaller powoduje pojawienie się w systemie Windows dwóch współpracujących ze sobą procesów. Pierwszy proces odpowiada za program rozruchowy, który:

1. Jeżeli uruchomiony został pojedynczy plik wykonywalny, rozpakowuje on binaria do tymczasowego katalogu o sygnaturze „MEI",
2. Ustawia lub usuwa zmienne środowiskowe,
3. Uruchamia proces-dziecko,
4. Czeka, aż proces-dziecko zakończy swoje działanie,
5. Jeżeli uruchomiony został pojedynczy plik wykonywalny, usuwa tymczasowy katalog o sygnaturze „MEI".

Z kolei proces-dziecko:

1. W przypadku platformy Windows generuje kontekst aktywacji,
2. Ładuje dynamiczną bibliotekę Pythona. Jej nazwa jest osadzona w pliku wykonywalnym,
3. Inicjalizuje interpreter Pythona ustawiając zmienne środowiskowe PYTHONPATH i PYTHONHOME,
4. Uruchamia kod Pythona.

6. Autorskie metody inżynierii wstecznej

W niniejszym rozdziale opisuję autorskie metody inżynierii wstecznej, których zastosowanie umożliwia dekompresję oraz dekompilację „zamrożonych binariów” Pythona, powstałych na wskutek użycia popularnych exe-packerów.

6.1 Inżynieria wsteczna pythonowych binariów - py2exe

py2exe umożliwia dwa sposoby na organizację plików binarnych:

- binaria zawierające główny skrypt Pythona, z odrębnym plikiem archiwum `.zip`,
- binaria zawierające główny skrypt Pythona i osadzone wewnątrz archiwum `.zip`, tworząc tzw. pojedynczy plik wykonywalny (ang. single executable file).

W obu przypadkach archiwum `.zip` uwzględnia pythonowe moduły użytkownika, zestawienie plików z kodem bajtowym biblioteki standardowej Pythona, użytej do kompilacji binariów, oraz moduły dodatkowych bibliotek wykorzystywanych przez program.

py2exe, bez względu na parametry metody `setup` w pliku `build.py`, dzieli zasoby pliku binarnego na trzy części:

1. nagłówek, zawierający wspomnianą wartość `magic number`, flagi `unbuffered` i `optimize` oraz długość skryptu w bajtach,
2. relatywna ścieżka do archiwum `.zip` - o ile istnieje,
3. sekwencja serializowanych obiektów Pythona (code objects), dołączany skrypt rozruchu `boot_common` oraz główny skrypt kompilowanego programu.

Flaga `unbuffered` umożliwia wyłączenie buforowania wejścia / wyjścia (I/O), z kolei `optimize` pozwala na zastosowanie jednego z trzech poziomów kompresji na pythonowym kodzie bajtowym:

- 0 - generowanie plików `.pyc`,

- 1 - generowanie plików `.pyo` (odpowiada fladze -O),
- 2 - generowanie plików `.pyo` (odpowiada fladze -OO).

Code objects, to nieedytowalne obiekty Pythona reprezentujące kawałki kodu bajtowego wraz z innymi metadanymi, tj. [8] [11]:

- liczba oraz deklaracja oczekiwanych typów argumentów,
- lista zmiennych lokalnych,
- informacje o kodzie źródłowym, na podstawie którego powstał kod bajtowy,
- kod bajtowy w postaci stringu (CPython 2) lub ciągu bajtów (CPython 3).

Znając już budowę „zamrożonych binariów" Pythona, powstałych przy użyciu py2exe, można przystąpić do realizacji algorytmu inżynierii wstecznej oprogramowania.

Ogólna koncepcja algorytmu dla py2exe przedstawia się następująco:

1. Zlokalizowanie w nagłówku pliku binarnego sekcji `PYTHONSCRIPT`,
2. Wykonanie zrzutu zasobów,
3. Konwersja zasobów do postaci code objects,
4. Wczytanie code objects za pomocą modułu `marshal`,
5. Zapisanie kodu bajtowego do postaci plików `.pyc`, z uwzględnieniem magic number i daty ich modyfikacji,
6. Dekompresja za pomocą algorytmu ZIP pozostałych plików `.pyc` z archiwum `.zip` lub jeżeli zostało ono osadzone wewnątrz, dekompresja pliku binarnego `.exe`,
7. Uruchomienie dekompilatora (unpyc3[43], uncompyle2[44]) właściwego dla wersji kodu bajtowego w celu otrzymania źródeł skryptów Pythona `.py`.

6.2 Inżynieria wsteczna pythonowych binariów - cx_Freeze

cx_Freeze umożliwia trzy sposoby na organizację plików binarnych:

- pojedynczy plik binarny zawierający skrypty Pythona z oddzielną biblioteką `python27.dll` (Windows),

43 https://code.google.com/p/unpyc3/

44 https://pypi.python.org/pypi/uncompyle2/

- pojedynczy plik binarny z odrębnym archiwum library.zip, kompresującym skrypty Pythona i oddzielną biblioteką python27.dll (Windows),
- pojedynczy plik binarny z odrębnym archiwum .zip - o tej samej nazwie co binaria - kompresującym skrypty Pythona i oddzielną biblioteką python27.dll (Windows).

Ogólna koncepcja algorytmu dla cx_Freeze przedstawia się następująco:

1. Dekompresja pliku binarnego .exe lub - jeżeli istnieje - archiwum .zip, za pomocą algorytmu ZIP,
2. Uruchomienie dekompilatora (unpyc3, uncompyle2) właściwego dla wersji kodu bajtowego w celu otrzymania źródeł skryptów Pythona .py.
3. Zmiana nazwy głównego skryptu uruchomieniowego __main__.py lub <nazwa_skryptu>__main__.py na nazwę zawartą w pierwszej linii źródeł pliku (w komentarzu).

Aby upewnić się jaką nazwę ma interesujące nas archiwum .zip, wystarczy użyć poniższego unixowego polecenia:

```
strings <nazwa_pliku>.exe | grep "[a-zA-Z0-9]\.zip"
```

6.3 Inżynieria wsteczna pythonowych binariów - bbfreeze

Bez względu na sposób wykorzystania bbfreeze, powstałe pliki binarne zachowują spójną architekturę wewnętrzną:

- pojedynczy plik binarny z odrębnym archiwum library.zip, kompresującym skrypty Pythona i oddzielną biblioteką python27.dll (Windows) oraz listingiem plików .pyd.

Ogólna koncepcja algorytmu dla bbfreeze przedstawia się następująco:

1. Dekompresja archiwum library.zip za pomocą algorytmu ZIP,
2. Uruchomienie dekompilatora (uncompyle2) w celu otrzymania źródeł skryptów Pythona .py.

3. Zmiana nazwy głównego skryptu uruchomieniowego z __main__<nazwa_skryptu>__.py na <nazwa_skryptu>.py

6.4 Inżynieria wsteczna pythonowych binariów - PyInstaller

PyInstaller umożliwia dwa sposoby na organizację plików binarnych:

- w ramach konfiguracji „one-folder", w wyniku której powstaje katalog zawierający binarny program rozruchowy wraz z całym niezbędnym środowiskiem uruchomieniowym w postaci plików .pyd i .dll.
- w ramach konfiguracji „one-file", w wyniku której powstaje pojedynczy plik wykonywalny (ang. single executable file).

Biorąc pod uwagę opisywaną w poprzednim rozdziale architekturę plików binarnych powstałych przy użyciu PyInstaller, w celu zrealizowania inżynierii wstecznej zastosowałem następujący algorytm:

1. Otwarcie pliku w trybie binarnego odczytu,
2. Ustawienie wskaźnika odczytu danych na znaku kontrolnym końca pliku (EOF) i zapamiętanie jego pozycji,
3. Ustawienie wskaźnika odczytu danych na pozycji tabeli CArchive, w oparciu o wartość stałej COOKIE (24 dla PyInstaller 1.5 lub 88 dla PyInstaller 2+),
4. Pobranie 8 bajtów źródła pliku w celu ustalenia i porównania wartości „MAGIC", co pozwala na jednoznaczne określenie, czy plik binarny powstał przy użyciu PyInstaller oraz z której jego wersji skorzystano,
5. Dekompresja CArchive:
    1. W przypadku zgodności, pobranie stałej porcji danych, niezbędnej do określenia długość tabeli CArchive, rozmiaru archiwum, użytej do budowania pliku wersji Pythona, a także nazwy biblioteki dynamicznej (w nowszych wersjach PyInstaller),
    2. Ustalenie pozycji plików umieszczonych na końcu nagłówka PE w oparciu o rekordy tabeli CArchive. Wczytanie, ewentualna dekompresja oraz zapisanie ich na dysk,
    3. Usunięcie ostatniego „białego znaku" ze źródeł skryptów Pythona .py.

6. Dekompresja ZlibArchive:
    1. Otwarcie archiwum ZlibArchive w trybie binarnego odczytu, zastosowanie przesunięcia i wczytanie go za pomocą modułu `marshal`,
    2. Iterowanie po serializowanej strukturze:
        1. Pobieranie paczki danych odpowiadającej rozmiarowi danego pliku,
        2. Zastosowanie na paczce dekompresji zlib[45], w celu otrzymania kodu bajtowego (code objects),
        3. Zapisanie kodu bajtowego do postaci plików `.pyc` z uwzględnieniem magic number.
    3. Zamknięcie archiwum ZlibArchive.
7. Uruchomienie dekompilatora (uncomyple2) w celu otrzymania źródeł skryptów Pythona `.py`.

6.5 Kompresja i dekompresa plików wykonywalnych

Kompresja plików wykonywalnych lub bibliotek dynamicznych polega na zmniejszaniu ich rozmiaru. W tym celu wykorzystuje się kompresory FSG, UPX, PE Compact, MPRESS, ASPack, E-ZIP, PETite, NeoLite, SecuPack czy cEXE. Do najczęściej stosownych należy UPX, dlatego też poświęcę mu więcej uwagi w dalszej części pracy.

Ultimate Packer for eXecutables (UPX)[46] to darmowe narzędzie udostępniane na licencji GNU GPL umożliwiające zmniejszanie ilości miejsca zajmowanego na dysku przez program wykonywalny w postaci np. pliku `.exe` lub bibliotek `.so` czy `.dll`. W przypadku systemów Microsoft Windows, kompresji można dokonać wyłącznie na binariach zgodnych ze specyfikacją Microsoft Portable Executable (PE) oraz COFF[47].

Kompresja programów z UPX jest bardzo szybka i efektowna. Przeciętnie pliki `.exe` po spakowaniu zajmują o połowę mniej miejsca, a w niektórych przypadkach nawet o 80%. Jest to zależne od budowy oryginalnego pliku. UPX obsługuje się z linii wiersza poleceń.

45 http://www.zlib.net/

46 http://upx.sourceforge.net/

47 http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

W tym celu wystarczy podać jedną z dostępnych komend, wskazać ścieżkę do pliku i przekazać parametr określający jakość kompresji.

Przykład wywołania UPX 3.91 z flagami zapewniającymi najlepszą jakość kompresji:

```
upx.exe --ultra-brute --compress-icons=0 <nazwa_pliku.exe>
```

Kompresory stosowane są również do utrudniania dekompilacji plików binarnych. Ich użycie, poza wspomnianą zmianą rozmiaru, powoduje zaciemnienie programu - sekcji PE, co automatycznie pozbawia skuteczności działania większości dekompilatorów.

Dlatego też proces inżynierii wstecznej plików binarnych Pythona powinien uwzględniać detekcję użycia kompresji. Można to uczynić analizując nagłówki plików `.exe` czy bibliotek `.dll`, pod kątem śladów charakterystycznych dla zastosowanych kompresorów. W następnym rozdziale pracy zaprezentuję kod programu, który realizuje takie zadanie [12].

W przypadku wykrycia kompresji w sekcji PE, powinno się zastosować procedurę odwrotną, czyli dekompresję pliku binarnego. Dla UPX należy wywołać następujące polecenie:

```
upx.exe -d <nazwa_pliku.exe>
```

7. Weryfikacja i porównanie opracowanych metod inżynierii wstecznej na przykładach

W celu zbadania skuteczności opracowanych algorytmów inżynierii wstecznej „zamrożonych binariów” Pythona, przygotowałem zestaw aplikacji, do których zaliczają się zarówno programy konsolowe, jak i te posiadające kompleksowy interfejs użytkownika (GUI). Pochodzą one z różnych źródeł, są to zarówno programy mojego autorstwa jak, osób ze środowiska języka Python oraz projekty o otwartym kodzie. Niektóre z przykładów zostały celowo przygotowane pod kątem niniejszej pracy.

Narzędzia wykorzystywane do inżynierii wstecznej pythonowych binariów także powstały w języku Python. Do swojego działania potrzebują przede wszystkim pakietu pefile[48] [7], umożliwiającego czytanie i modyfikowanie sekcji PE w blikach binarnych.

Przykład 1 - binaria powstałe przy użyciu Pythona 2.7.8 i py2exe 0.6.9

Opis, struktura i uruchomienie programu

Program wypisuje *Ciąg liczbowy Fibonacciego* dla podanej liczby w argumencie wywołania programu.

Struktura:

- bz2.pyd
- **library.zip**
- python27.dll
- unicodedata.pyd
- _hashlib.pyd
- **py2exe_1.exe**
- select.pyd
- w9xpopen.exe

Uruchomienie:

```
C:\>py2exe_1.exe 1000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
610 987
```

48 https://code.google.com/p/pefile/

Metodyka procesu inżynierii wstecznej

W celu uzyskania kodów źródłowych z pliku binarnego powstałego za sprawą użycia py2exe, uruchomiam skrypt `unfrozen_binary_py2exe.py`, dekompresujący i dekompilujący „zamrożone binaria" zgodnie z algorytmem z poprzedniego rozdziału o numerze 6.1.

Skrypt `unfrozen_binary_py2exe.py` przyjmuje w argumencie nazwę pliku `.exe`.

Uruchomienie:

```
./unfrozen_binary_py2exe.py py2exe_1.exe
Magic value: 2018915346
optimize flag: 0
unbuffered flag: 0
Code bytes length: 2963

Archive name: library.zip

Extracting boot_common.pyc
Extracting py2exe_1.pyc

Python bytecode file: boot_common.pyc
Python version: Python 2.7a0
Modyfication time: Thu Sep 25 18:36:10 2014

Python source code file: boot_common.py

Python bytecode file: py2exe_1.pyc
Python version: Python 2.7a0
Modyfication time: Thu Sep 25 18:36:10 2014

Python source code file: py2exe_1.py

(ciąg dalszy listingu modułów użytkownika i standardowej biblioteki Pythona)

Work is done.
```

Skrypt podczas realizacji algorytmu inżynierii wstecznej wypisuje na ekranie konsoli najbardziej istotne informacje o zawartości pliku `.exe` oraz postępach prac:

- wartość magic number,
- wartości zastosowanych flag unbuffred i optimize,
- długość kodu bajtowego,
- nazwa zewnętrznego archiwum `.zip` lub informacja o jej braku,
- nazwa pliku rozruchowego boot oraz głównego, osadzonego wewnątrz binariów skryptu Pythona,
- wersja Pythona użyta do kompilacji skryptów do postaci kodu bajtowego `.pyc`,
- data ostatniej modyfikacji kodu bajtowego,
- listing dekompilowanych plików źródłowych, w tym modułów biblioteki standardowej Pythona.

W wyniku działania skryptu `unfrozen_binary_py2exe.py` otrzymuję katalog o analogicznej nazwie do dekompresowanego i dekompilowanego programu - `py2exe_1`, wraz z jego plikami źródłowymi `.py`, kodem bajtowym `.pyc` lub `.pyo` oraz katalogiem `library`, zawierającym moduły użytkownika oraz biblioteki standardowej Pythona.

Listing 4 i listing 5 stanowią porównanie głównego kodu programu uzyskanego w wyniku dekompilacji z oryginalnym źródłem skryptu:

```
#Embedded file name: py2exe_1.py
__author__ = 'Katharsis'
__copyright__ = 'Copyright 2014, Piotr Tynecki'
__license__ = 'BSD'
__version__ = '1.0'
import sys

def fibonacci(number):
    """ Returns the Fibonacci sequence """
    a, b = (0, 1)
    while a < number:
        yield a
        a, b = b, a + b
```

```
if __name__ == '__main__':
    try:
        number = int(sys.argv[1])
    except (ValueError, IndexError):
        number = 1000

    for x in fibonacci(number):
        print x,
```

Listing 4. Źródło pliku py2exe_1.py otrzymanego w wyniku dekompilacji kodu bajtowego.

```
# -*- coding: utf-8 -*-

__author__ = 'Katharsis'
__copyright__ = 'Copyright 2014, Piotr Tynecki'
__license__ = 'BSD'
__version__ = '1.0'


import sys


def fibonacci(number):
    """ Returns the Fibonacci sequence """
    a, b = 0, 1

    while a < number:
        yield a
        a, b = b, a + b


if __name__ == '__main__':
    try:
        number = int(sys.argv[1])
        #If argument isn't digital or is empty
    except (ValueError, IndexError):
        number = 1000

    for x in fibonacci(number):
```

```
        print x,
```

Listing 5. Oryginalne źródło pliku py2exe_1.py.

Różnice w kodzie źródłowym obu skryptów są niewielkie i nieznaczące z punktu widzenia działania i logiki programu. Dekompilacja powoduje usunięcie niektórych „nowych linii” oraz jednolinijkowych komentarzy w kodzie.

Przykład 2 - pojedynczy plik binarny powstały przy użyciu Pythona 2.7.8 i py2exe 0.6.9

Opis, struktura i uruchomienie programu

Do tego przypadku wykorzystam program z przykładu 1, jednakże zmianie ulegnie metoda organizacji pliku wykonywalnego. Wszystkie biblioteki `.dll`, `.pyd` oraz kod bajtowy zawarty w archiwum `.zip` zostały umieszczone wewnątrz pliku `.exe`. Dzięki temu, uzyskałem pojedynczy plik wykonywalny, który jest łatwy w dystrybucji.

Struktura:

- **py2exe_2.exe**

Uruchomienie:

```
C:\>py2exe_2.exe 1000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
610 987
```

Metodyka procesu inżynierii wstecznej

W związku z tym, że skrypt `unfrozen_binary_py2exe.py` potrafi jednoznacznie określić strukturę binariów i dobrać względem niej właściwy algorytm, proces inżynierii wstecznej sprowadza się wyłącznie do jego zainicjowania.

Uruchomienie:

```
./unfrozen_binary_py2exe.py py2exe_2.exe
Magic value: 2018915346
optimize flag: 2
unbuffered flag: 0
Code bytes length: 3118
```

```
Archive is embedded. Unzip the binary.

Extracting boot_common.pyo
Extracting <install zipextimporter>.pyo
Extracting py2exe_2.pyo

Python bytecode file: boot_common.pyo
Python version: Python 2.7a0
Modyfication time: Thu Sep 25 18:44:10 2014

Python source code file: boot_common.py

Python bytecode file: <install zipextimporter>.pyo
Python version: Python 2.7a0
Modyfication time: Thu Sep 25 18:44:10 2014

Python source code file: <install zipextimporter>.py

Python bytecode file: py2exe_2.pyo
Python version: Python 2.7a0
Modyfication time: Thu Sep 25 18:44:11 2014

Python source code file: py2exe_2.py

(ciąg dalszy listingu modułów użytkownika i standardowej biblioteki Pythona)

Work is done.
```

W wyniku działania skryptu `unfrozen_binary_py2exe.py` otrzymuję katalog `py2exe_2`, wraz z jego plikami źródłowymi `.py`, kodem bajtowym `.pyo` (z powodu zastosowania flagi `optimize`) oraz katalogiem `library`, zawierającym moduły użytkownika i te z biblioteki standardowej Pythona.

Źródło pliku `py2exe_2.py` otrzymanego w wyniku dekompilacji kodu bajtowego nie różni się wcale od odtworzonego w ten sam sposób pliku `py2exe_1.py`. Oznacza to, że zmiana

struktury binariów `.exe` czy zastosowanie maksymalnej optymalizacji pliku nie modyfikuje kodu bajtowego Pythona.

Przykład 3 - binaria powstałe przy użyciu Pythona 2.7.8 i cx_Freeze 4.3.3

Opis, struktura i uruchomienie programu

WikiPath to program wyznaczający najkrótszą drogę pomiędzy dwoma podanymi tytułami artykułów na angielskiej Wikipedii[49]. Współautorami programu są Mateusz Miłek i Wojciech Łaguna.

Struktura:

- bz2.pyd
- python27.dll
- _ssl.pyd
- **WikiPath.exe**
- _hashlib.pyd
- _socket.pyd
- unicodedata.pyd

Uruchomienie:

```
C:\>WikiPath.exe PyPy PHP
http://en.wikipedia.org/wiki/PHP
http://en.wikipedia.org/wiki/Cross-platform
http://en.wikipedia.org/wiki/PyPy
```

Metodyka procesu inżynierii wstecznej

W celu uzyskania kodów źródłowych z pliku binarnego powstałego w wyniku użycia cx_Freeze, należy uruchomić skrypt `unfrozen_binary_cx_Freeze.py`, dekompresujący i dekompilujący „zamrożone binaria” zgodnie z algorytmem z poprzedniego rozdziału o numerze 6.2.

Skrypt `unfrozen_binary_cx_Freeze.py` przyjmuje w argumencie nazwę pliku `.exe`.

Uruchomienie:

```
./unfrozen_binary_cx_Freeze.py WikiPath.exe
Archive is embedded. Unzipping the binary.
```

49 http://en.wikipedia.org/

```
Renaming: __main__.pyc to WikiPath.pyc

Python source code file: fnmatch.py

Python source code file: opcode.py

(ciąg dalszy listingu modułów użytkownika i standardowej biblioteki Pythona)

Work is done.
```

Skrypt podczas realizacji algorytmu inżynierii wstecznej wypisuje na ekranie konsoli najbardziej istotne informacje o postępach prac:

- nazwa zewnętrznego archiwum `.zip` lub informacja o dekompresji pliku binarnego,
- zmiana nazwy głównego skryptu Pythona z notacji charakterystycznej dla cx_Freeze do pierwotnej postaci,
- listing dekompilowanych plików źródłowych, w tym modułów biblioteki standardowej Pythona.

W wyniku działania skryptu `unfrozen_binary_cx_Freeze.py` otrzymuję katalog o analogicznej nazwie do dekompresowanego i dekompilowanego programu - `WikiPath`, wraz z jego plikami źródłowymi `.py` i kodem bajtowym `.pyc` lub `.pyo` modułów użytkownika oraz biblioteki standardowej Pythona.

Listing 6 i listing 7 stanowią porównanie głównego kodu programu uzyskanego w wyniku dekompilacji z oryginalnym źródłem skryptu:

```
#Embedded file name: WikiPath.py
import sys
import urllib2
import json
import re


class WikiPath(object):
```

```
    def __init__(self, urlFrom, urlTo):
        self.urlFrom = urlFrom
        self.urlBase = urlFrom
        self.urlTo = urlTo
        self.sql = [(0, self.urlFrom)]
        self.uniq = {}

    def href(self, link, parentID):
        try:
            self.urlFrom = 'http://en.wikipedia.org/w/api.php?
action=parse&prop=text&page=%s&format=json' % link.split('/')[-1]
            r = urllib2.urlopen(self.urlFrom)
            text = json.loads(r.read())['parse']['text']['*']
            urls = [ 'http://en.wikipedia.org' + link for link in
re.findall('href=[\\\'"]?([^\\\'" >]+)', text) if '/wiki/' in link ]
            for link in urls:
                if link == self.urlTo:
                    self.path(parentID)
                    sys.exit(0)
                if not self.uniq.has_key(link):
                    self.uniq[link] = True
                    self.sql.append((parentID, link))

        except (KeyError, ValueError, TypeError):
            pass

    def path(self, pk):
        print self.urlTo
        while pk != 0:
            print self.sql[pk][1]
            pk = self.sql[pk][0]

        print self.urlBase

    def walk(self):
        for index, element in enumerate(self.sql):
            self.href(element[1], index)
```

```
if __name__ == '__main__':
    wp = WikiPath('http://en.wikipedia.org/wiki/' + sys.argv[1],
'http://en.wikipedia.org/wiki/' + sys.argv[2])
    wp.walk()
```

Listing 6. Źródło pliku WikiPath.py otrzymanego w wyniku dekompilacji kodu bajtowego.

```
# -*- coding: utf-8 -*-
import sys

import urllib2
import json
import re



class WikiPath(object):
    def __init__(self, urlFrom, urlTo):
        self.urlFrom = urlFrom
        self.urlBase = urlFrom
        self.urlTo = urlTo
        self.sql = [(0, self.urlFrom)]
        self.uniq = {}

    def href(self, link, parentID):
        try:
            self.urlFrom = "http://en.wikipedia.org/w/api.php?
action=parse&prop=text&page=%s&format=json" % link.split("/")[-1]

            r = urllib2.urlopen(self.urlFrom)
            text = json.loads(r.read())["parse"]["text"]["*"]
            urls = ["http://en.wikipedia.org" + link for link in
re.findall(r'href=[\'"]?([^\'" >]+)', text) if "/wiki/" in link]

            for link in urls:
                if link == self.urlTo:
                    self.path(parentID)

                    sys.exit(0)
```

```
                if not self.uniq.has_key(link):
                    self.uniq[link] = True
                    self.sql.append((parentID, link))
        except (KeyError, ValueError, TypeError):
            pass

    def path(self, pk):
        print self.urlTo

        while pk != 0:
            print self.sql[pk][1]
            pk = self.sql[pk][0]

        print self.urlBase

    def walk(self):
        for index, element in enumerate(self.sql):
            self.href(element[1], index)

if __name__ == "__main__":
    wp = WikiPath("http://en.wikipedia.org/wiki/" + sys.argv[1],
"http://en.wikipedia.org/wiki/" + sys.argv[2])

    wp.walk()
```

Listing 7. Oryginalne źródło pliku WikiPath.py.

Analogicznie do przykładów 1 i 2, w przypadku dekompilacji binariów powstałych przy użyciu cx_Freeze, również dochodzi do niewielkich zmian w wyglądzie kodu. Niemniej jednak oba skrypty są tak samo funkcjonalne.

Przykład 4 - binaria powstałe przy użyciu Pythona 2.7.8, bbfreeze 4.3.3 i UPX 3.91

Opis, struktura i uruchomienie programu

W przykładzie 4 ponownie używam programu wypisującego *Ciąg liczbowy Fibonacciego*, z tą różnicą, że importuje i wykorzystuje on dodatkowy moduł użytkownika `troll.py`,

a jego binaria powstały przy pomocy pakietu bbfreeze. Aby utrudnić zadanie inżynierii wstecznej, poddałem je dodatkowej kompresji UPX.

Struktura:

- **bbfreeze_1.exe**
- _hashlib.pyd
- python27.dll
- unicodedata.pyd
- bz2.pyd
- **library.zip**
- select.pyd

Uruchomienie:

```
C:\>bbfreeze_1.exe
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
610 987

Hello from troll
```

Metodyka procesu inżynierii wstecznej

W celu uzyskania kodów źródłowych z pliku binarnego powstałego przy użyciu bbfreeze, należy uruchomić skrypt `unfrozen_binary_bbfreeze.py`, dekompresujący i dekompilujący „zamrożone binaria" zgodnie z algorytmem z poprzedniego rozdziału o numerze 6.3.

Skrypt `unfrozen_binary_bbfreeze.py` stanowi modyfikację skryptu `unfrozen_binary_cx_Freeze.py`. Ponadto, rozszerza go o funkcjonalność detekcji kompresji (np. UPX). Skrypt przyjmuje w argumencie nazwę pliku `.exe`.

Uruchomienie:

```
./unfrozen_binary_bbfreeze.py bbfreeze_1.exe
Binary compression used: UPX -> www.upx.sourceforge.net
Before extracting and decompiling Python byte code from the binary you must
decompressed it.

For UPX use:

        ./upx -d <binary_name>
```

Skrypt wykrył w sekcji PE pliku binarnego zastosowaną kompresję UPX. Aby móc kontynuować proces inżynierii wstecznej muszę zdekompresować plik `.exe`, zgodnie z sugestią w podanym komunikacie.

Uruchomienie:

```
./upx -d bbfreeze_1.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91        Markus Oberhumer, Laszlo Molnar & John Reiser   Sep 30th 2013

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
      7680 <-      5632   73.33%    win32/pe     bbfreeze_1.exe

Unpacked 1 file.

./unfrozen_binary_bbfreeze.py bbfreeze_1.exe
Archive name: library.zip

Renaming: __main__bbfreeze_1__.pyc to bbfreeze_1.pyc

Python source code file: subprocess.py

Python source code file: fnmatch.py

(ciąg dalszy listingu modułów użytkownika i standardowej biblioteki Pythona)

Python source code file: troll.py

(ciąg dalszy listingu modułów użytkownika i standardowej biblioteki Pythona)

Work is done.
```

Skrypt podczas realizacji algorytmu inżynierii wstecznej wypisuje na ekranie konsoli najbardziej istotne informacje o postępach prac:

- nazwa zewnętrznego archiwum `.zip`,

- zmiana nazwy głównego skryptu Pythona z notacji charakterystycznej dla bbfreeze do pierwotnej postaci,
- listing dekompilowanych plików źródłowych, w tym modułów użytkownika czy biblioteki standardowej Pythona.

W wyniku działania skryptu `unfrozen_binary_bbfreeze.py` otrzymuję katalog o analogicznej nazwie do dekompresowanego i dekompilowanego programu - `bbfreeze_1`, wraz z jego plikami źródłowymi `.py` i kodem bajtowym `.pyc` lub `.pyo` modułów użytkownika oraz biblioteki standardowej Pythona.

Listing 8 i listing 9 stanowią porównanie kodu głównego skryptu programu oraz wykorzystywanego przez niego modułu `troll`, uzyskanych w wyniku dekompilacji względem oryginalnych źródeł obu plików:

```
#Embedded file name: __main__bbfreeze_1__.py
__author__ = 'Katharsis'
__copyright__ = 'Copyright 2014, Piotr Tynecki'
__license__ = 'BSD'
__version__ = '1.0'
import sys
import troll

def fibonacci(number):
    """ Returns the Fibonacci sequence """
    a, b = (0, 1)
    while a < number:
        yield a
        a, b = b, a + b


if __name__ == '__main__':
    try:
        number = int(sys.argv[1])
    except (ValueError, IndexError):
        number = 1000
```

```
    for x in fibonacci(number):
        print x,

    print troll.hello()



#Embedded file name: troll.py
__author__ = 'Katharsis'
__copyright__ = 'Copyright 2014, Piotr Tynecki'
__license__ = 'BSD'
__version__ = '1.0'

def hello():
    return '\n\nHello from troll'
```

Listing 8. Źródła plików bbfreeze_1.py i troll.py otrzymane w wyniku dekompilacji kodu bajtowego.

```
# -*- coding: utf-8 -*-
__author__ = 'Katharsis'
__copyright__ = 'Copyright 2014, Piotr Tynecki'
__license__ = 'BSD'
__version__ = '1.0'


import sys
import troll


def fibonacci(number):
    ''' Returns the Fibonacci sequence '''
    a, b = 0, 1

    while a < number:
        yield a
        a, b = b, a + b


if __name__ == '__main__':
```

```
    try:
        number = int(sys.argv[1])
    #If argument isn't digital or is empty
    except (ValueError, IndexError):
        number = 1000

    for x in fibonacci(number):
        print x,

    print troll.hello()


# -*- coding: utf-8 -*-

__author__ = 'Katharsis'
__copyright__ = 'Copyright 2014, Piotr Tynecki'
__license__ = 'BSD'
__version__ = '1.0'


def hello():
        return "\n\nHello from troll"
```

Listing 9. Oryginalne źródła pliku bbfreeze_1.py i troll.py.

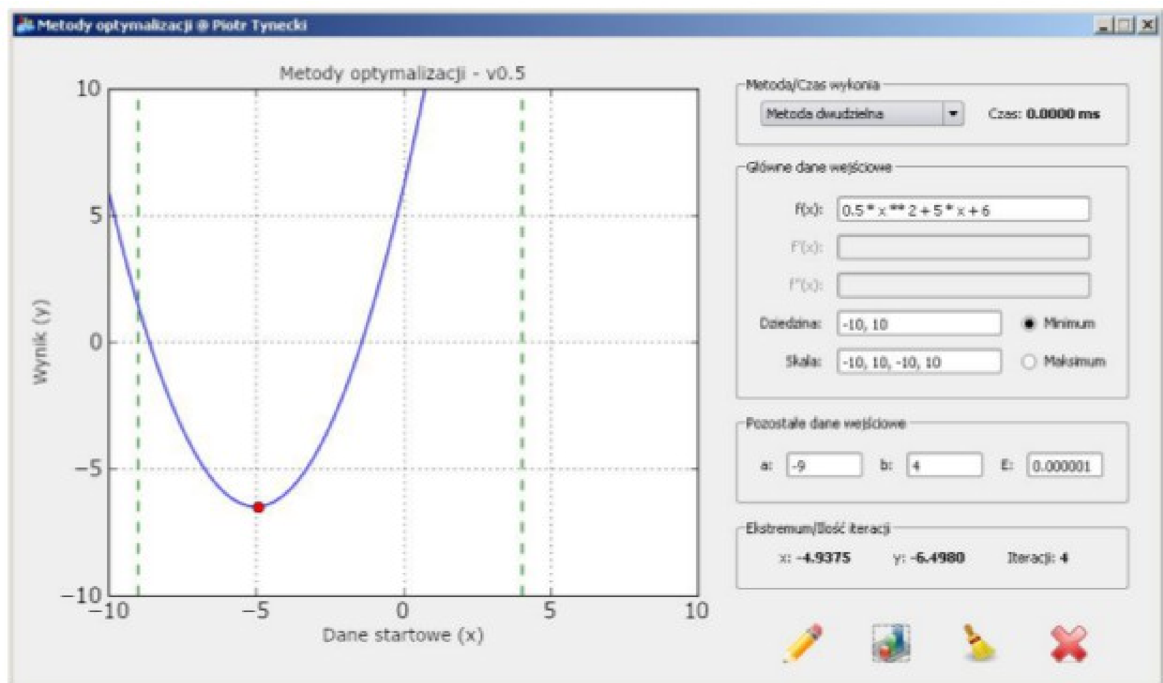
Analogicznie do przykładów poprzednich, w przypadku dekompilacji binariów powstałych przy użyciu bbfreeze, również dochodzi do niewielkich zmian w formatowaniu kodu, co nie wpływa na działanie obu skryptów.

Przykład 5 - binaria powstałe przy użyciu Pythona 2.7.8 i PyInstaller 2.1

Opis, struktura i uruchomienie programu

Niniejsza aplikacja okienkowa realizuje numeryczne algorytmy, służące do rozwiązywania zadań optymalizacji statycznej. Program umożliwia wybór jednej z pięciu metod optymalizacji. Rysuje wykres zadanej funkcji, oblicza i zaznacza na nim ekstremum (minimum/maksimum). Dodatkowo podaje czas wykonania algorytmu oraz liczba iteracji

względem zadanego kryterium. Program powstał w języku Python, frameworku PyQt4[50] przy wykorzystaniu bibliotek NumPy[51] i matplotlib[52].



Rys 6. Zrzut ekranu z działania programu *Metody optymalizacji*.

Struktura:

- **MO.exe**
- pliki *.pyd,
- pliki *.dll,
- katalog img
- katalog qt4_plugins
- katalog mpl-data
- katalog Include

Uruchomienie:

```
C:\>MO.exe
Uruchamia program okienkowy (GUI)
```

Katalog z programem *Metody optymalizacji* zawiera pokaźną liczbę plików `.pyd` i `.dll`. Publikowanie ich pełnej listy nie ma na tym etapie uzasadnienia.

50 http://www.riverbankcomputing.com/software/pyqt/intro

51 http://www.numpy.org/

52 http://matplotlib.org/

Metodyka procesu inżynierii wstecznej

W celu uzyskania kodów źródłowych z pliku binarnego powstałego przy użyciu PyInstaller, należy uruchomić skrypt `unfrozen_binary_pyinstaller.py`, dekompresujący i dekompilujący „zamrożone binaria” zgodnie z algorytmem z poprzedniego rozdziału o numerze 6.4.

Skrypt `unfrozen_binary_pyinstaller.py` przyjmuje w argumencie nazwę pliku `.exe`.

Uruchomienie:

```
./unfrozen_binary_pyinstaller.py MO.exe
Magic: 'MEI\x0c\x0b\n\x0b\x0e'
Length of package: 18127232
Position of TableOfContents: 18071624
Length of TableOfContents: 55584
Python version: 27
Python library name: None

Inside of the CArchive
This TOC length: 32
This TOC position: 0
Compressed data size: 2636247
Uncompressed data size: 2636247
Compression flag: 0
Compression type: z
Name: out00-PYZ.pyz
Directory: current

This TOC length: 32
This TOC position: 2636247
Compressed data size: 6427
Uncompressed data size: 16971
Compression flag: 1
Compression type: m
Name: iu.pyc
Directory: current

(ciąg dalszy listingu dekompresowanych plików z archiwum CArchive)
```

```
Python source code file: out00-PYZ.pyz_extracted/distutils.debug.pyc

(ciąg dalszy listingu modułów użytkownika, modułów zewnętrznych i standardowej biblioteki Pythona)

Work is done.
```

Skrypt podczas realizacji algorytmu inżynierii wstecznej wypisuje na ekranie konsoli najbardziej istotne informacje o postępach prac:

- wartość magic (PyInstaller MEI), rozmiar archiwum CArchive, pozycja archiwum w pliku binarnym, długość tabeli zawierającej zestawienie zarchiwizowanych plików, użyta wersja języka Python, nazwa biblioteki Pythona,
- dekompresja archiwów CArchive i ZlibArchive:
  - długość i pozycja tabeli, wartość skompresowanych i nieskompresowanych danych, flaga kompresji i typu pliku, nazwa i katalog pliku,
- listing dekompilowanych plików źródłowych, w tym modułów zewnętrznych i biblioteki standardowej Pythona.

W wyniku działania skryptu `unfrozen_binary_pyinstaller.py` otrzymuję katalog o analogicznej nazwie do dekompresowanego i dekompilowanego programu - `MO`. Wewnątrz katalogu dostępne są m.in.:

- główny skrypt programu,
- pythonowe skrypty pomocniczego PyInstaller,
- dynamiczne biblioteki `.pyd` i `.dll`,
- pliki `.manifest` (charakterystyczne dla Microsoft Windows),
- podkatalog `out00-PYZ.pyz_extracted` zawierający:
  - pliki źródłowe `.py`,
  - pliki z kodem bajtowym `.pyc` lub `.pyo` modułów użytkownika, wykorzystanych bibliotek zewnętrznych oraz biblioteki standardowej Pythona,
- podkatalog z plikami multimedialnymi.

Chciałbym podkreślić, że PyInstaller, w przeciwieństwie do poprzednich exe-packerów, wewnątrz archiwum CArchive nie umieszcza skompilowanej wersji głównego skryptu programu (`.pyc`) a jego wersję źródłową (`.py`). Dzięki temu, etap dekompilacji - na tym pliku - nie ma racji bytu.

Dlatego też, listing 10 i listing 11 stanowią porównanie fragmentu kodu jednego z modułów użytkownika programu uzyskanego w wyniku dekompilacji z oryginalnym źródłem tego pliku:

```
#-*- coding: utf-8 -*-
def metodaDwudzielna(f, a, b, E, max = None, min = None):
    xn = (a + b) / 2.0
    l = b - a

    iteracje = 0

    while l > E:
        iteracje += 1
        x1 = a + 1.0/4.0 * l
        x2 = a + 3.0/4.0 * l

        if max:
            if f(x1) > f(x2):
                b = xn
                xn = x1
            elif f(x2) > f(xn):
                a = x2
                xn = x2
            else:
                a = x1
                b = x2
        elif min:
            if f(x1) < f(x2):
                b = xn
                xn = x1
            elif f(x2) < f(xn):
                a = x2
                xn = x2
```

```
            else:
                a = x1
                b = x2

        l = b - a

    ekstremum = (a + b) / 2.0

    return ekstremum, f(ekstremum), iteracje
```

Listing 10. Fragment źródła pliku MetodyOptymalizacji.py otrzymanego w wyniku dekompilacji kodu bajtowego.

```
#Embedded file name: D:\Workspace\Dyplom\MO\build\pyi.win32\MO\out00-
PYZ.pyz\MetodyOptymalizacji
def metodaDwudzielna(f, a, b, E, max = None, min = None):
    xn = (a + b) / 2.0
    l = b - a
    iteracje = 0
    while l > E:
        iteracje += 1
        x1 = a + 1.0 / 4.0 * l
        x2 = a + 3.0 / 4.0 * l
        if max:
            if f(x1) > f(x2):
                b = xn
                xn = x1
            elif f(x2) > f(xn):
                a = x2
                xn = x2
            else:
                a = x1
                b = x2
        elif min:
            if f(x1) < f(x2):
                b = xn
                xn = x1
            elif f(x2) < f(xn):
                a = x2
```

```
                xn = x2
            else:
                a = x1
                b = x2
        l = b - a

    ekstremum = (a + b) / 2.0
    return (ekstremum, f(ekstremum), iteracje)
```

Listing 11. Oryginalne źródło fragmentu pliku MetodyOptymalizacji.py.

Z tytułu zastosowania tego samego dekompilatora pythonowego kodu bajtowego - uncompyle2, różnice w obu kodach sięgają wyłącznie zmian na poziomie formatowania plików, nie mających żadnego wpływu na działanie programu.

Przykład 6 - pojedynczy plik binarny powstały przy użyciu Pythona 2.7.8 i PyInstaller 2.1

Opis, struktura i uruchomienie programu

Do tego przykładu wykorzystałem tę samą aplikację co poprzednio, czyli program *MetodyOptymalizacji*. Zmianie uległa wyłącznie metoda organizacji architektury pliku binarnego, gdyż w tym wypadku program jest pojedynczym plikiem binarnym.

Struktura:

- **MO.exe**
- katalog img

Uruchomienie:

```
C:\>MO.exe
Uruchamia program okienkowy (GUI)
```

Metodyka procesu inżynierii wstecznej

W związku z tym, że skrypt `unfrozen_binary_pyinstaller.py` potrafi jednoznacznie określić strukturę binariów i dobrać względem niej właściwy algorytm, proces inżynierii wstecznej sprowadza się wyłącznie do jego zainicjowania.

Uruchomienie:

```
./unfrozen_binary_pyinstaller.py MO.exe
Magic: 'MEI\x0c\x0b\n\x0b\x0e'
Length of package: 22196540
Position of TableOfContents: 22183460
Length of TableOfContents: 12992
Python version: 27
Python library name: 'python27.dll'

Inside of the CArchive
This TOC length: 32
This TOC position: 0
Compressed data size: 4097920
Uncompressed data size: 4097920
Compression flag: 0
Compression type: z

Name: out00-PYZ.pyz
Directory: current

This TOC length: 32
This TOC position: 4097920
Compressed data size: 170
Uncompressed data size: 234
Compression flag: 1
Compression type: m

Name: struct.pyc
Directory: current

(ciąg dalszy listingu dekompresowanych plików z archiwum CArchive)

Python source code file: out00-PYZ.pyz_extracted/distutils.debug.pyc

(ciąg dalszy listingu modułów użytkownika, modułów zewnętrznych i standardowej biblioteki Pythona)

Work is done.
```

Skrypt podczas realizacji algorytmu inżynierii wstecznej wypisuje na ekranie konsoli analogiczny zestaw informacji, jak w przykładzie 5. Zmianie uległy wyłącznie wartości takie jak rozmiar binariów, pozycja CArchive i ZlibArchive w pliku czy rekord „Python library name", który uzyskał wartość „'python27.dll'". Wynika to z tego, że wszystkie zależności (biblioteki dynamiczne) zostały włączone w struktury pliku binarnego.

Wynik działania skryptu `unfrozen_binary_pyinstaller.py` jest identyczny jak w przypadku wersji programu z przykładu 5. Tyczy się to zarówno zawartości katalogu `MO` jak i różnic pomiędzy oryginalnym kodem źródłowym `.py` a jego zdekompilowanym odpowiednikiem.

Poniżej prezentuję wybrany fragment obu kodów źródłowych plików `MetodyOptymalizacji.py`:

```
def metodaEulera(f, fp1, a, b, E):
    f1 = fp1(a)
    f2 = fp1(b)
    iteracje = 0
    while abs(a - b) > E:
        iteracje += 1
        x0 = a - f1 * (a - b) / (f1 - f2)
        f0 = fp1(x0)
        if abs(f0) < E:
            break
        else:
            b = a
            f2 = f1
            a = x0
            f1 = f0

    return (x0, f(x0), iteracje)
```

Listing 12. Fragment źródła pliku MetodyOptymalizacji.py otrzymanego w wyniku dekompilacji kodu bajtowego.

```
def metodaEulera(f, fp1, a, b, E):
```

```
    f1 = fp1(a)
    f2 = fp1(b)

    iteracje = 0

    while abs(a - b) > E:
        iteracje += 1

        x0 = a - f1 * (a - b) / (f1 - f2)
        f0 = fp1(x0)

        if abs(f0) < E:
            break
        else:
            b = a
            f2 = f1
            a = x0
            f1 = f0

    return x0, f(x0), iteracje
```

Listing 13. Oryginalne źródło fragmentu pliku MetodyOptymalizacji.py.

Różnice w źródłach sięgają wyłącznie zmian na poziomie formatowania plików. Nie wpływają one na działanie programu.

7.1 Porównanie zrealizowanych metod inżynierii wstecznej

W niniejszej pracy zaprezentowałem zastosowanie autorskich metod inżynierii wstecznej na przykładzie oprogramowania powstałego w języku programowania Python. Wszystkie programy poddane dekompresji oraz dekompilacji zostały utworzone przy użyciu popularnych, pythonowych exe-packerów - py2exe, cx_Freeze, bbfreeze i PyInstaller. Ich zastosowanie pozwala na pełną przenośność skryptów Pythona w postaci plików binarnych.

Każdy z wykorzystanych i zbadanych exe-packerów stosuje odmienną metodę organizacji pliku binarnego oraz na inne sposoby zabezpiecza się przed jego dekompilacją. W związku z tym, dla binariów będących wynikiem ich użycia, przygotowałem odrębne algorytmy inżynierii wstecznej oprogramowania. W dalszej części rozdziału umieszczam także tabele podsumowujące skuteczność działania tychże algorytmów, zarówno w aspekcie dekompresji jak i dekompilacji pythonowych binariów.

Stopień trudności inżynierii wstecznej binariów powstałych przy użyciu py2exe, w skali od 1-5, oceniam na 3. Procedura sprowadza się do ustalenia czy w nagłówku PE dostępna jest sekcja PYTHONSCRIPT. Sekcja ta ma swoją pozycję oraz rozmiar. Po jej pobraniu i dekompresji określam takie informacje jak pozycja i długość osadzonego kodu bajtowego Pythona, umiejscowienie głównego skryptu uruchomieniowego czy dostępność archiwum ZIP z pythonowym kodem bajtowym. Następnie deserializuję i zapisuję dane do postaci fizycznych plików `.pyc` i `.zip`. Realizuję dekompresję archiwum i dekompilację kodu bajtowego do postaci plików źródłowych `.py`. Skrypt implementujący ten algorytm obsługuje zarówno przypadek z odrębnym archiwum ZIP jak i w formie pojedynczego pliku binarnego.

Ilość zbadanych binariów	**Wersje Pythona**	**Platforma**	**Skuteczność dekompresji**	**Skuteczność dekompilacji**
13	2.5, 2.6 i 2.7	Microsoft Windows	100%	91%

Tab 3. Podsumowanie skuteczności działania skryptu unfrozen_binary_py2exe.py.

Inżynieria wsteczna binariów powstałych przy użyciu cx_Freeze i bbfreeze jest najmniej skomplikowana i zgodnie z przyjętą skalą oceniam jej trudność na 1. Procedura sprowadza się do ustalenia czy binaria zawierają wewnątrz siebie archiwum o nazwie `library.zip` lub innej, zdefiniowanej przez użytkownika, który dokonał „zamrożenia binariów”. Jeżeli takie archiwum istnieje, dekompresuję je za pomocą algorytmu ZIP. W przeciwnym wypadku stosuję tę samą dekompresję na pliku binarnym. W wyniki tego powstaje katalog z kodem bajtowym Pythona w formie plików `.pyc`. Głównemu skryptowi

uruchomieniowemu zmieniam nazwę na oryginalną, gdyż w przypadku cx_Freeze plik nazywa się `__main__.py` lub `<nazwa_pliku>__main__.py` a bbfreeze `__<nazwa_pliku>__main__.py`. W wyniku dekompilacji plików `.pyc` otrzymuję odpowiadające im kody źródłowe `.py`.

Ilość zbadanych binariów	**Wersje Pythona**	**Platforma**	**Skuteczność dekompresji**	**Skuteczność dekompilacji**
cx_Freeze: 6	2.7 i 3.3	GNU/Linux, Microsoft Windows	100%	89%
bbfreeze: 4	2.4, 2.5, 2.6 i 2.7	GNU/Linux, Microsoft Windows	100%	97%

Tab 4. Podsumowanie skuteczności działania skryptów unfrozen_binary_cx_Freeze.py i unfrozen_binary_bbfreeze.py.

Inżynieria wsteczna binariów powstałych przy użyciu PyInstaller, wymagała ode mnie odstąpienia od tradycyjnych metod i podjęcia zupełnie innych kroków postępowania. Pakiet ten stosuje podwójną archiwizację opartą o zlib oraz modyfikuje nagłówki i znak końca pliku (EOF) w plikach `.pyc` z kodem bajtowym. Jej trudność oceniam na 4 z 5 możliwych punktów skali.

Opracowana metoda inżynierii wstecznej dla binariów PyInstaller ustala czy na końcu pliku binarnego znajduje się ciąg znaków odpowiadający MEI\014\013\012\013\016. PyInstaller 1.5 oraz jego najnowsza wersja 2.1 przechowują ten ciąg z różnym przesunięciem, co też przewiduję i obsługuję. Jeżeli poszukiwany ciąg znaków ma miejsce w strukturze pliku binarnego, pobieram paczkę danych - o rozmiarze odpowiadającemu użytej wersji PyInstaller, która zawiera takie informacje jak wartość magic, długość archiwum CArchive, pozycję i długość tabeli z zestawieniem tego archiwum, użytą wersję Pythona oraz nazwę biblioteki dynamicznej Pythona. Na podstawie tych informacji przystępuję do dekompresji CArchive. Jeżeli składowa archiwum została oznaczona flagą 'z', stanowi ona wewnętrzne archiwum ZlibArchive, które wczytuję za pomocą modułu `marshal` i ponownie dekompresuję algorytmem zlib. PyInstaller jako jedyny

z exe-packerów domyślnie usuwa z nagłówków kodu bajtowego wspomnianą wartość „magic number” oraz datę jego ostatniej modyfikacji. Dlatego też, przed zapisem kodu bajtowego do pliku, dodaję te wartości na samym jego początku. Usuwam z nich również ostatni znak. W wyniku dekompilacji plików `.pyc` otrzymuję odpowiadające im kody źródłowe `.py`. Skrypt implementujący ten algorytm obsługuje także przypadek pojedynczego pliku binarnego.

Ilość zbadanych binariów	**Wersje Pythona**	**Platforma**	**Skuteczność dekompresji**	**Skuteczność dekompilacji**
11	2.7 i 3.3	GNU/Linux, Microsoft Windows	100%	98%

Tab 5. Podsumowanie skuteczności działania skryptu unfrozen_binary_pyinstaller.py.

Powyższe testy skuteczności działania skryptów przeprowadziłem na większości dostępnych dziś wariantach architektury pythonowych plików binarnych. Były to zarówno pliki z zewnętrznymi bibliotekami dynamicznymi oraz archiwami `.zip` jak i pojedyncze, niezależne pliki binarne. Pod pojęciem „skuteczności dekompilacji” należy rozumieć zgodności wyrażoną w procentach pomiędzy zdekompilowanym kodem źródłowym a jego oryginalną wersją.

8. Podsumowanie

Chodź w ramach niniejszej pracy zaimplementowane metody inżynierii wstecznej zostały przeze mnie przetestowane tylko na sześciu przykładach, to docelowo liczba ta sięga 34 „zamrożonych binariów”. Testowane programy pochodzą zarówno ze źródeł własnych jak i zewnętrznych. Testom poddałem pliki binarne dedykowane na systemy Windows oraz GNU/Linux, oparte na Pythonie 2.7 i Pythonie 3.3. Zaimplementowane algorytmy w pełni automatyzują proces inżynierii wstecznej oprogramowania.

Wszystkie zbadane przykłady udało mi się zdekompresować i zdekompilować do niemal oryginalnej postaci kodu źródłowego programów, odtwarzając także pierwotne struktury ich katalogów. Potwierdza to poprawne działanie programów w wersji zawierającej odtworzone źródła. Pozwala mi to uznać opracowane metody inżynierii wstecznej za skuteczne.

Prawdziwość tego stwierdzenia ma miejsce dla przypadków, w których użytkownik nie dokonuje dodatkowych, własnych zmian w działaniu exe-packerów czy nie modyfikuje działania standardowego interpretera i kodu bajtowego Pythona. Programy poddawane inżynierii wstecznej muszą być budowane w oparciu o Pythona 2.4, 2.5, 2.6, 2.7 oraz Pythona 3.3. Na chwilę obecną Python 3.0, 3.1, 3.2 oraz Python 3.4 nie są przeze mnie obsługiwane na etapie dekompilacji kodu bajtowego, co nie zatrzymuje całkowicie działania algorytmów, a jedynie pomija etap dekompilacji plików `.pyc` lub `.pyo`.

Praktyka pokazuje, że korporacje takie jak Google czy Dropbox, udostępniają swoje produkty w ramach niestandardowo stworzonych „zamrożonych binariów” Pythona, na których opracowane przeze mnie metody nie są w stanie odtworzyć źródeł tych programów. Zastosowano tu bardzo skomplikowane operacje, których odtworzenie można ustalić wyłącznie poprzez ręczną, mozolną analizę plików binarnych. Warto dodać, że złamanie tych zabezpieczeń miało już miejsce [6].

Uważam, że prostszym a zarazem skuteczniejszym sposobem na zabezpieczenie się przed inżynierią wsteczną oprogramowania „zamrożonych binariów” Python a jest stosowanie

kompilacji pythonowych modułów od postaci dynamicznych bibliotek `.dll`, `.pyd` lub `.so`. Taką kompilację można wykonać za pomocą języka i kompilatora Cython[53], który tłumaczy kod Pythona na język C, a następnie dokonuje jego kompilacji. Istnieje również kompilator Nuitka[54], który realizuje podobne zadanie, gdzie translacja następuje na język C++. Oba rozwiązania posiadają jednak wsparcie wyłącznie dla standardowej biblioteki Pythona.

Na zakończenie pragnę dodać, że nadal będę rozwijał opracowane metody inżynierii wstecznej oprogramowania. Docelowo chcę obsłużyć wszystkie wersje języka Python, a także dodać nową metodę, przeznaczona dla pakietu py2app[55], tworzącego „zamrożone binaria” na platformę Mac OS X.

53 http://cython.org/

54 http://nuitka.net/

55 https://pypi.python.org/pypi/py2app/

9. Literatura

[1] Justin Seitz, *Gray hat Python : Python programming for hackers and reverse engineers*, Wydawnictwo No Starch Press, San Francisco 2009

[2] TJ O'Connor, *Violent Python A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers*, Wydawnictwo Syngress, Stany Zjednoczone 2012

[3] Piotr Tynecki, *Metody tworzenia plików wykonywalnych ze skryptów języka Python*, Praca dyplomowa, Uniwersytet w Białymstoku - Wydział Informatyki i Matematyki, Białystok 2012

[4] Blue Coat, *Snake In The Grass: Python-based Malware Used For Targeted Attacks*, Witryna internetowa, https://www.bluecoat.com/security-blog/2014-06-10/snake-grass-python-based-malware-used-targeted-attacks, stan z dnia 10 czerwca 2014 r

[5] Kelvina Lomboya, *A Walk Through on Decompiling a Malware Packed with Py2Exe*, Witryna internetowa, http://www.kelvinlomboy.com/a-walk-through-on-decompiling-a-malware-packed-with-py2exe, stan z dnia 21 października 2011

[6] Dhiru Kholia i Przemysław Węgrzyn, *Looking Inside the (Drop) Box*, Witryna internetowa, https://www.usenix.org/conference/woot13/workshop-program/presentation/kholia, stan z dnia 13 sierpnia 2013

[7] Ero Carrera, *Win32 Static Analysis in Python*, Witryna internetowa, http://www.recon.cx/en/f/lightning-ecarrera-win32-static-analysis-in-python.pdf, stan z dnia 30 sierpnia 2014,

[8] Aaron Portnoy, Ali Rizvi-Santiago, *Reverse Engineering Dynamic Languages - A Focus on Python*, Witryna internetowa, http://recon.cx/2008/a/aaron_portnoy-ali_rizvi_santiago/slides.pdf, stan z dnia 2 lipca 2014

[9] Ryan F Kelly, *Bytecode: What, Why, and How to Hack it*, [video], PyCon Australia 2011, https://www.youtube.com/watch?v=ve7lLHtJ9l8, stan z dnia 8 sierpnia 2014

[10] Larry Hastings, *All-Singing All-Dancing Python Bytecode*, [video], PyCon US 2013, https://www.youtube.com/watch?v=CKu6d_v4Pqo, stan z dnia 5 lipca 2014

[11] Aleksander P. Czarnowski, *Reversing Python objects*, Wydawnictwo VIRUS BULLETIN, Polska 2011

[12] Dr Petri, *Using UPX as a Security Packer*, Witryna internetowa, http://dl.packetstormsecurity.net/papers/general/Using_UPX_as_a_security_packer.pdf, stan z dnia 21 marca 2012

[13] History of Python, Witryna internetowa. http://en.wikipedia.org/wiki/History_of_Python, stan z dnia 20 czerwca 2014

[14] Andrzej Matlak, *Charakter prawny regulacji dotyczących zabezpieczeń technicznych utworów*, Wydawnictwo Wolters Kluwer, Warszawa 2007