VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

FUZZ TESTOVÁNÍ APLIKACÍ KOMUNIKUJÍCÍCH PROSTŘEDNICTVÍM ODATA PROTOKOLU

FUZZ TESTING OF APPLICATIONS COMMUNICATING VIA THE ODATA PROTOCOL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ĽUBOŠ MJACHKY**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Mjachky Ľuboš**
Obor: Informační technologie
Téma: **Fuzz testování aplikací komunikujících prostřednictvím OData protokolu**
Fuzz Testing of Applications Communicating via the OData Protocol
Kategorie: Analýza a testování softwaru

Pokyny:

1. Seznamte se s OData protokolem.
2. Prostudujte principy fuzz testování.
3. Navrhněte nástroj pro fuzz testování aplikací komunikujících pomocí OData protokolu.
4. Implementujte navržený nástroj a otestujte ho na aplikacích vybraných ve spolupráci se společností SAP.
5. Diskutujte dosažené výsledky a možnosti jejich rozvoje.

Literatura:

1. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley Professional, 2007.
2. Takanen, A., DeMott, J.D., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance, Artech House Information Security and Privacy, 2008.
3. Bonnen, C., Drees, V., Fischer, A., Heinz, L., Strothmann, K.: OData and SAP Netweaver Gateway, SAP Press, 2014.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání a částečné rozpracování bodu třetího.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Vojnar Tomáš, prof. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Dodávať stabilný a spoľahlivý softvér nie je jednoduché. Aplikácie sú náchylné k chýbam bez ohľadu na dôslednosť a skúsenosť vývojárov. Aby sa zabránilo chybovým stavom na strane zákazníka, používajú sa vo všetkých fázach vývoja softvéru rôzne automatizované testovacie metódy či nástroje. Cieľom tejto práce je navrhnúť a implementovať automatizovaný nástroj na testovanie biznis aplikácií. Akákoľvek akcia, s ktorou sa v aplikácii nesprávne naloží, môže spôsobiť zlyhanie s katastrofickými následkami. Na simulovanie takýchto scenárov je možné pri testovaní použiť také testovacie sady, ktoré obsahujú náhodné alebo poškodené dáta. Takýto spôsob testovania sa odborne nazýva fuzz testovanie, pričom sa na vstup aplikácie zavádzajú náhodné alebo zmutované dáta. Navrhnutý nástroj ODfuzz slúži na fuzz testovanie aplikácií komunikujúcich prostredníctvom protokolu OData, ktorý je postavený na metodológiách HTTP a REST. Nástroj ODfuzz generuje požiadavky, ktoré obsahujú náhodné dáta a odosiela ich na serverovú časť aplikácie. Dáta z požiadavky prechádzajú pri spracovávaní rôznymi vetvami kódu, čo v konečnom dôsledku môže vyústiť do chybového stavu. Nástroj ODfuzz bol použitý na testovanie backend modulov moderných SAP aplikácii naprogramovaných v jazyku ABAP.

Abstract

Delivering stable and reliable software to customers is difficult. Applications are prone to errors, regardless of the experience level of the developers. Automated testing methods and tools are heavily used in all phases of development life-cycle to reduce chances of bugs escaping to the users. The goal of this thesis is to design an intelligent and automated testing tool which is able to test business applications. A mishandled action performed within such an application may cause a failure with disastrous consequences. To simulate these actions, one can use testing where test cases contain invalid or random data. This testing technique is called fuzzing or fuzz testing, and it involves providing malformed or mutated data as an input to the program. The proposed tool, namely ODfuzz, is a fuzzing tool ready to test applications communicating via the OData protocol which is a protocol built on existing HTTP and REST methodologies. ODfuzz is generating and fuzzing requests that are to be sent to the server. The requests contain mutated data that pass through various code paths and may result into an application error. ODfuzz was used to test back-end modules of modern SAP applications written in the ABAP language.

Kľúčové slová

OData, fuzzing, automatizácia testovania ABAP, SAP, cloudové riešenia, genetická slučka

Keywords

OData, fuzzing, test automation, ABAP, SAP, cloud solutions, genetic loop

Citácia

MJACHKY, Ľuboš. *Fuzz testování aplikací komunikujících prostřednictvím OData protokolu.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Vojnar, Ph.D.

Fuzz testování aplikací komunikujících prostřednictvím OData protokolu

Prehlásenie

Prehlasujem, že som túto prácu vypracoval samostatne pod vedením pána prof. Ing. Tomáša Vojnara Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Ľuboš Mjachky
16. mája 2018

Poďakovanie

Chcel by som poďakovať vedúcemu bakalárskej práce pánovi prof. Ing. Tomášovi Vojnarovi Ph.D. za cenné rady a pripomienky. Takisto by som chcel vyjadriť veľkú vďačnosť Jakubovi Filákovi, odbornému vedúcemu zo spoločnosti SAP, za jeho pomoc a usmerňovanie počas celej tvorby bakalárskej práce.

Obsah

Kapitola 1

Úvod

Záujem o biznis aplikácie bežiace v cloud infraštruktúre stále rastie. Pre organizácie sú takéto riešenia veľmi výhodné, nakoľko sa znižujú náklady spojené so zaobstarávaním a údržbou nového hardvéru, zvyšuje sa flexibilita použitia a dostupnosť zdrojov.

Takéto aplikácie obsahujú citlivé dáta, a preto sa kladie veľký dôraz na ich zabezpečenie. V prípade, že sa v aplikácii objaví chyba, môže dôjsť k strate alebo úniku dát. Jedna takáto udalosť dokáže spôsobiť obrovskú finančnú škodu.

Preto je neodmysliteľnou súčasťou vývoja aj testovanie. Testovaním dokážeme odhaliť chyby ešte pred vypustením produktu na trh. Keďže sú moderné systémy zložitejšie a prepracovanejšie, ručné testovanie sa stáva menej efektívnym a časovo náročnejším riešením.

Cieľom tejto práce je navrhnúť a implementovať automatizovaný nástroj na testovanie aplikácií komunikujúcich prostredníctvom protokolu OData[1]. OData je protokol vyvinutý spoločnosťou Microsoft a umožňuje vytvárať, čítať, meniť alebo mazať dáta nachádzajúce sa na serveri. Tento protokol sa používa vo viacerých moderných aplikáciách na komunikáciu medzi serverovou a klientskou časťou.

Navrhnutý nástroj, ODfuzz, sa chová ako klientská aplikácia, ktorá posiela na server požiadavky a spracováva prijaté odpovede. Požiadavky sú prevažne dáta, ktoré sú z časti chybné alebo náhodné a bežne sa na server nezasielajú. Testovanie takýmto spôsobom sa odborne nazýva fuzz testovanie.

Samotná práca vznikla v spolupráci so spoločnosťou SAP[2]. Spoločnosť SAP sa zaoberá vývojom aplikácií na riadenie podnikových procesov. Sú to najmä aplikácie pre správu financií, logistiky, personalistiky a pod. Všetky spomínané aplikácie bežia v cloud infraštruktúre a jednotlivé testy boli spúšťané práve na nich.

Text práce je štruktúrovaný nasledovne. Princípy fuzz testovania sú podrobne popísané v kapitole 2. Protokolu OData je venovaná samostatná kapitola 3. V kapitole 4 sa vyskytuje popis návrhu nástroja na abstraktnej úrovni, tzn. z akých logických celkov sa bude nástroj skladať a akým spôsobom bude medzi nimi prebiehať komunikácia. Kapitola 5 naopak presnejšie ozrejmuje detaily či riziká implementácie. Výsledky testovania a prehľad experimentov je možné nájsť v kapitole 6.

[1] Open Data Protocol http://www.odata.org/
[2] SAP https://www.sap.com/index.html

Kapitola 2

Fuzz testovanie

„*Testovanie je proces spúšťania programu so zámerom hľadať chyby.*“ [11] Takto definoval testovanie autor knihy „The Art of Software Testing“ Glenford Myers.

Na rozdiel od ladenia softvéru, kde hľadáme odpoveď na otázku: „*prečo program nefunguje?*“, zmyslom testovania je upevňovanie si domnienky o tom, že program funguje tak, ako má. Využitím systematického testovania môžeme nájsť niekoľko stavov, ktoré vyvolajú v aplikácii chybu. Ladením potom zistíme, prečo takýto stav spôsobil chybu a po oprave znovu pokračujeme v testovaní [16].

Fuzz testovanie alebo fuzzing môžeme definovať ako automatizovanú techniku testovania softvéru, ktorá používa neplatné alebo náhodné dáta ako vstup, za účelom odhalenia zraniteľností a chýb [18].

Fuzzer je termín, ktorým sa popisuje zoskupenie automatizovaných skriptov, nástrojov a aplikácií, ktoré dokážu generovať fuzz dáta, prenášať ich, monitorovať a zaznamenávať reakcie testovaných aplikácií [4]. Fuzz dáta sú dáta, ktoré sú náhodné, z časti chybné a vznikli mutáciou pôvodných dát. Typy a druhy fuzzerov budú objasnené v podkapitole 2.2.

Prvýkrát bol pojem fuzzing, ako spôsob testovania spoľahlivosti softvéru, použitý v roku 1989 profesorom Bartonom Millerom a jeho triedou pri vývoji jednoduchého fuzzera na testovanie robustnosti unixových aplikácií. Jednalo sa o tzv. black-box[1] testovanie, pričom sa na vstup aplikácie predávali náhodne vygenerované reťazce znakov. Keď aplikácia prestala pracovať alebo celá zhavarovala, testom neprešla. Ďalším významným míľnikom fuzz testovania bol rok 1999, kde sa na Univerzite Oulu vo Fínsku začala práca na projekte PROTOS[2]. Vytváranie testovacích sád, spočívalo v tom, že sa najprv analyzovali jednotlivé komunikačné protokoly a pre tieto protokoly sa samostatne vytvárali pakety, ktoré nespĺňali špecifikácie daného protokolu. V tomto prípade sa už jednalo zmiešané white-box[3] a black-box testovanie [17].

Na zreteľ musíme brať to, že fuzz testovanie zatiaľ nie je dokonalé. Vždy nám v programe ostanú nejaké neodchytené chyby. V tejto kapitole si vysvetlíme základy fuzz testovania a popíšeme ich nedostatky.

[1] Vnútorná štruktúra testovacieho programu nie je známa

[2] Projekt PROTOS https://www.ee.oulu.fi/roles/ouspg/Protos

[3] Interná štruktúra (kód, architektúra a dizajn) je pri testovaní známa

2.1 Princípy fuzz testovania

Fuzzing je jedna z techník negatívneho testovania. Najrozšírenejšia metóda negatívneho testovania sa volá angl. „fault injection“. Ide o metódu, v ktorej sa chyby vložia buď do dát, ktoré sa používajú na testovanie, alebo vedome do programu pre overenie efektívnosti testov [18]. V našom prípade sa budeme zaoberať hlavne prvou variantou.

Princíp testovania je u každého fuzz testovania rovnaký. Nehľadiac na to, aký druh fuzzera použijeme, postupujeme vždy podľa nasledujúcich bodov [17]:

- identifikácia cieľa,
- identifikácia vstupov,
- generovanie fuzz dát,
- spustenie testov,
- monitorovanie programu,
- analýza zraniteľností.

Jednotlivé body sú podrobne popísané v sekciách 2.1.1 až 2.1.5.

2.1.1 Identifikácia cieľa

V tomto kroku ide hlavne o výber programu, resp. aplikácie, ktorú chceme testovať. Terčom testovania sú hlavne aplikácie, ktoré prijímajú vstup cez internet, bežia v privilegovanom režime alebo spracovávajú osobné informácie [4].

Cieľom testovania sú samozrejme aj iné aplikácie. Petr Müller sa napríklad vo svojej práci „Automatizované metody hledání chyb v překladačích“ [12] venuje fuzz testovaniu prekladačov.

2.1.2 Identifikácia vstupov

Aplikácie môžu prijímať vstup v rôznych formách. Väčšina zraniteľností bola v minulosti spôsobená nedostatočnou validáciou alebo ošetrením vstupu. Určenie vstupných vektorov je pri fuzz testovaní kľúčové. Pretože ak aplikácia odmietne celý vstup len kvôli tomu, že mu nie je schopná prideliť dostatočnú veľkosť pamäte, môže to celé testovanie limitovať.

Po určení vstupných vektorov je nutné si dáta vygenerovať. Dáta sú generované automatizovane, a to na základe či už preddefinovaných hodnôt, alebo ich mutovaním [17].

Sutton vo svojej knihe „Fuzzing: Brute Force Vulnerability Discovery“ [4] rozdelil druhy vstupov do tried, ako: argumenty príkazového riadku, premenné prostredia, web aplikácie, súborové formáty, internetové protokoly, COM[4] objekty, pamäť a medziprocesorová komunikácia. Pre väčšinu prípadov už existuje špecifický fuzzer, viď ďalej sekciu 2.3.

2.1.3 Generovanie fuzz dát

Generované dáta by mali pokrývať čo najväčšiu časť množiny vstupného stavového priestoru. Na počiatku pracovali fuzzery len s čisto náhodnými dátami. Ukázalo sa, že takýto prístup je častokrát veľmi neefektívny. Ako príklad si môžeme vziať protokol HTTP.

[4]Component Object Model https://msdn.microsoft.com/en-us/library/windows/desktop/ms680573

Keď HTTP správa nie je vo formáte, ktorý definuje štandard protokolu, prehliadač celú správu zahodí. Fuzzer, ktorý používa náhodné dáta, vygeneruje správu začínajúcu ako `GET / HTTP/1.0 \n\n` len raz z 256^{16} pokusov [6].

Dôležité je, aby boli testy, poprípade samotné výsledky testov, reprodukovateľné. Na to sa používa pri vytváraní fuzz dát semienko[5], angl. „seed“. Obyčajný náhodný fuzzer pracuje s dvoma typmi semienok. Prvý pre dáta a druhý pre zavedenie náhodnosti do celého procesu fuzz testovania [18].

Jedna z metrík, ktorá sa používa na zistenie úspešnosti testovania, je pokrytie testovaného kódu. Určuje nám, aké úseky kódu boli vykonané počas testovania.

```
1 def check_input(x):
2     if x == 10:
3         run()
4     else:
5         exit(1)
```

Zdrojový kód 2.1: Kontrola vstupu

Veľký problém pri testovaní s čisto náhodnými dátami je dosiahnuť dobré pokrytie. Ako príklad si môžeme zobrať kód 2.1. Pravdepodobnosť toho, že vojdeme do tela podmienky `if` je 1 ku 2^{32} s náhodným generátorom čísel rovnomerného rozloženia, ak je `x` 32-bitové číslo. Podobne to platí aj pre tzv. brute-force testovanie, kde generujeme zaradom všetky možnosti stavového priestoru. Vnorené podmienky tento stav ešte viac zhoršujú.

Číslicové hodnoty a reťazce sa generujú pomocou heuristík. V prípade, že sa jedná o číslo, je vhodné generovať čísla okolo hornej hranice maximálneho rozsahu, a to napríklad inkrementovaním, resp. dekrementovaním, poprípade násobením. Pri reťazcoch sa používa niekoľkonásobné opakovanie znaku alebo skupiny znakov. Internetové protokoly používajú na ukončenie správy či na rozčlenenie sekcií oddeľovače. Do úvahy berieme pri generovaní preto aj kombináciu niekoľkých oddeľovačov a bežných reťazcov [17].

Brute-force a náhodnosť využívajú aj generátory mutovaných dát. Proces vzniku takýchto dát sa nazýva angl. „data mutation fuzzing“. Dáta sa vytvárajú modifikáciou hodnôt na určitých miestach. Náhodne alebo pomocou brute-force sa zvolí miesto a druh zmien, ktoré sa na danom mieste majú stať [4]. Bližšie sa budeme mutačným fuzzerom venovať v sekcii 2.2.1.

2.1.4 Spustenie testov

Táto fáza ide ruka v ruke spolu s generovaním fuzz dát. Vygenerované dáta sa prenesú na vstup do testovanej aplikácie. Spúšťanie je automatizované a môže v sebe zahrňovať otváranie súborov, zasielanie paketov alebo spúšťanie procesov [17].

2.1.5 Monitorovanie programu

Na úvod tejto sekcie začneme motivačným citátom. „*Čo získate, keď na IMAP server slepo vypálite 50,000 zdeformovaných autentifikačných požiadaviek, jedna z nich spôsobí jeho pád a status IMAP servera kontrolujete len pri poslednom teste, nikdy inokedy? Jednoducho: Nič.*“ [17].

Program musíme monitorovať neustále, najlepšie po každom vykonanom teste. Ideálne by bolo, ak by sme vedeli spraviť snímku, angl. „snapshot“, systému pred a po každom

[5]Číslo alebo vektor, ktorý sa používa pre inicializáciu pseudonáhodného generátoru

teste. Porovnaním oboch stavov systému získame tzv. mapu rozdielov. Jedna snímka však môže dosiahnuť veľkosť niekoľkých gigabajtov, čím sa tento prístup stáva takmer nereálny. Ako jeden z najjednoduchších spôsobov monitorovania je detekcia nulovej odozvy zo strany testovanej aplikácie. Testovaním responzivity pred začiatkom každého testu ľahko identifikujeme vstup, ktorý spôsobil zlyhanie celej aplikácie. Aplikácia je aktívna vtedy, ak dokáže odpovedať na zaslané platné dáta [4].

Ďalšou možnosťou monitorovania je sledovanie behu procesu. V linuxovom jadre na to existuje podsystém `ptrace`[6], ktorý stopuje systémové volania, signály a zápisy do pamäte. Pri svojej činnosti využíva `ptrace` niekoľko nástrojov. Jedným z nich je napr. `strace`[7]. Sledovaním výstupov podobných nástrojov vieme detegovať anomálie pri rôznych druhoch vstupov buď pri otváraní, alebo zápise do súborov [18].

Podobne sa pre monitorovanie využívajú aj ladiace nástroje, angl. „debuggers“. Pre systémy Windows existuje knižnica PyDbg[8]. Ide o Win32 debugger, ktorým je možné sa napojiť na bežiaci proces a pozorovať tak jeho beh. Následujúci kód 2.2 je prebraný z knihy „Fuzzing: Brute Force Vulnerability Discovery“ [17] a ukazuje, ako elegantne sa dá za pomoci frameworku PaiMei[9] a PyDbg pripojiť debugger na bežiaci proces.

```
1  from pydbg import *
2  from pydbg.defines import *
3  import utils
4
5  def av_handler(dbg):
6      crash_bin = utils.crash_binning.crash_binning()
7      crash_bin.record_crash(dbg)
8
9      # signal the fuzzer
10
11     print crash_bin.crash_synopsis()
12     dbg.terminate_process()
13
14 while 1:
15     dbg = pydbg()
16     dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, av_handler)
17     dbg.load(target_program, arguments)
18
19     # signal the fuzzer
20
21     dbg.run()
```

Zdrojový kód 2.2: Napojenie na proces a spracovanie výnimky

V nekonečnom cykle sa vytvára inštancia triedy PyDbg a pre spracovanie výnimky sa použije funkcia `av_handler()`. Kedykoľvek výnimka nastane, vykoná sa táto funkcia. Následne si pripravíme cieľový program, jeho vstupné parametre pre ladenie a spustíme monitorovanie. Počas monitorovania môžeme informovať fuzzer o tom, kedy má fuzz dáta generovať, poprípade kedy došlo k výnimke. Výnimka `EXCEPTION_ACCESS_VIOLATION` je

[6] ptrace http://man7.org/linux/man-pages/man2/ptrace.2.html
[7] strace https://linux.die.net/man/1/strace
[8] PyDbg https://github.com/OpenRCE/pydbg
[9] PaiMei https://github.com/OpenRCE/paimei

vyvolaná vtedy, keď chce program čítať z pamäte, ku ktorej nemá prístup [17]. Neprítomnosť oficiálnej podpory pre Python 3 je jediné negatívum tejto knižnice.

Za zmienku tiež stojí spomenúť, že v rámci testovania sa používajú rôzne nástroje na monitorovanie pamäte bežiaceho procesu, dynamické analyzátory (napr. Valgrind[10]) či sanitizéry. Sanitizéry sú dynamické testovacie nástroje, ktoré pri kompilácii vložia do kódu ďalšie kontroly pamäte, pretečenia, nedefinovaného chovania atď. Medzi najznámejšie patria ASan (AddressSanitizer), TSan (ThreadSanitizer), MSan (MemorySanitizer) a UBSan (UndefinedBehaviorSanitizer). Používajú sa v prekladači clang[11] pre jazyky C a C++ [15].

2.1.6 Analýza zraniteľností

Posledná fáza fuzz testovania je určenie závažnosti všetkých zraniteľností a ich nahlásenie. Analýza typicky vyžaduje manuálne spracovanie, kde tester skúma, či je chyba naozaj zraniteľnosťou alebo nie.

Automatizovaný zber a zhlukovanie hlásení o chybách má niekoľko výhod. Chyby, výnimky a ich variácie môžeme zoskupovať do tried, čím sa testerovi výrazne zredukuje počet celkových zraniteľností, ktoré musí preskúmať. V spoločnosti Microsoft bol tento proces implementovaný tak, že sa pre každú vyvolanú výnimku vytvoril jedinečný identifikátor, podľa tzv. „stack trace“ a offsetu[12], ktorý bol použitý ako názov priečinku. Stack trace je zoznam volaných metód na zásobníku v okamihu, kedy došlo k chybe. Keď nastala nová výnimka, vypočítala sa hash hodnota identifikátora. V prípade, že hodnota hash existovala, všetky detaily sa zalogovali do podpriečinku, ktorého meno bolo tvorené súborom, ktorý výnimku pôvodne vyvolal [4].

2.2 Typy fuzzerov

Každý testovaný objekt sa niečím líši a vyžaduje iný prístup fuzz testovania. Z tohto dôvodu sú fuzzery špecializované pre takmer všetky kategórie aplikácií.

Kód 2.3 je ukážkou náhodného generátoru ASCII znakov o veľkosti `length`. Ďalší kód 2.4 je ukážkou generátoru, ktorý mutuje existujúci reťazec `string` zmenou znaku na náhodnom indexe. Oba generátory sú veľmi triviálne, nepoužívajú žiadnu heuristiku.

```
1 def random_ascii(length):
2     return ''.join(chr(random.randint(0, 255))
3                    for _ in range(length))
```

Zdrojový kód 2.3: Náhodný generátor ASCII znakov

```
1 def mutate_ascii(string):
2     index = random.randint(0, len(string) - 1)
3     return ''.join([string[:index], chr(random.randint(0, 255)),
4                     string[index + 1:]])
```

Zdrojový kód 2.4: Mutovanie ASCII reťazca zmenou znaku

Po pridaní heuristiky a zmenou povahy generovania, napr. dodaním vedomosti o tom, akého formátu má byť reťazec, dostaneme z vyššie uvedených ukážok nutný, ale zároveň

[10] Valgrind http://valgrind.org/
[11] clang https://clang.llvm.org/
[12] Hodnota pridaná ku ukazovateľovi do pamäte za účelom získania konkrétneho prvku z celej sekvencie

postačujúci základ pre tzv. inteligentný, resp. smart fuzzer. Predstaviteľmi inteligentných fuzzerov sú generačné fuzzery. Opakom smart fuzzerov sú hlúpe, resp. dumb fuzzery. Zástupcami takýchto fuzzerov môžu byť aj mutačné fuzzery. Oba typy fuzzerov sú ozrejmené v nasledujúcich dvoch sekciách. Sekcie sú prebrané z „Fuzzing for Software Security Testing and Quality Assurance“ [18] a „Fuzzing: The State of the Art“ [6].

2.2.1 Mutačný fuzzer

Mutačný fuzzer dostáva na vstup validné dáta a mutuje ich. Mutačné fuzzery sú typicky generické fuzzery. Nemusia mať znalosť o formáte a druhu mutovaných dát. Na implementáciu teda vyžadujú menšie ľudské úsilie. Platné vstupy sa modifikujú vkladaním chybných bajtov, poprípade ich prehadzovaním.

Fuzzery sa používajú pre systémy, ktoré prijímajú štrukturované vstupy. Nie je nutné, aby sa vytvárali úplne nové rozsiahle dáta, stačí, ak sa budú mutovať pôvodné. Nevýhodou takéhoto testovania je slabé pokrytie kódu, vzťahujúceho sa na celkovú funkcionalitu. Najviac testované sú úseky kódu, kde sa spracováva systémový vstup.

2.2.2 Generačný fuzzer

Na rozdiel od mutačného fuzzera, generačný fuzzer sa viaže na konkrétny protokol, aplikáciu alebo formát súboru.

Generačný fuzzer vytvára nový vstup podľa definícií z jeho vlastných zdrojov. Všetku vedomosť o formáte generovaného vstupu musí do fuzzera vložiť programátor. Pokrytie testovaného kódu potom závisí na kvalite a úplnosti znalostí o štruktúre. Mechanizmus na generovanie dát často zahrňuje konečné automaty, gramatiky a formálne jazyky.

2.3 Dostupné nástroje

Cieľom tejto podkapitoly je oboznámiť čitateľa s dostupnými a v súčasnosti používanými nástrojmi a frameworkami na fuzz testovanie. Spomenuté budú len tie najznámejšie z nich.

2.3.1 AFL

American fuzzy lop, skr. AFL, je brute-force fuzzer využívajúci genetický algoritmus[13]. Nástroj sa používa pri testovaní C/C++ aplikácií.

Testovať aplikácie je možné vtedy, ak máme k dispozícii ich zdrojový kód alebo priamo binárny program. Keď je k dispozícii zdrojový kód, fuzzer dokáže pri kompilácii vložiť do kódu inštrumentácie[14], čo má pozitívny dopad na celkový výkon pri generovaní fuzz dát. Zdrojový kód sa prekladá pomocou `afl-gcc` prepísaním premennej prostredia [20].

AFL používa genetický algoritmus na generovanie vstupných dát. Pred samotným začiatkom generovania sa inicializuje populácia počiatočnými validnými vstupmi. Následne sa vyberú dva vstupy a začnú sa medzi sebou krížiť, mutovať, čím vzniknú nové vstupy. Každý novovytvorený vstup sa v aplikácii spustí, vyhodnotí fitness funkciou[15] a podľa výsledku zaradí, resp. nezaradí do populácie pri ďalšom generovaní. Toto generovanie pokračuje dovtedy, dokým sa nenájde najlepší reprezentatívny vstup, nehavaruje testovaný program

[13] Algoritmus, ktorý sa snaží aplikáciou evolučnej biológie nájsť riešenie problému
[14] Ďalší kód, ktorý pomáha pri analýze napr. pokrytia spúšťaného kódu pri testovaní
[15] Funkcia, ktorá určí, ako dobré je riešenie vzhľadom na typ problému

alebo sa vykoná postačujúci počet iterácií. Pseudokód algoritmu 2.5 je prevzatý z článku „VUzzer: Application-aware Evolutionary Fuzzing“ [14].

```
INITIALIZE population with seed inputs
repeat
    SELECT1 parents
    RECOMBINE parents to generate children
    MUTATE parents/children
    EVALUATE new candidates with FITNESS function
    SELECT2 fittest candidates for the next population
until TERMINATION CONDITION is met
return BEST Solution
```

Zdrojový kód 2.5: Pseudokód evolučného algoritmu, tzv. genetická slučka

Pre správne fungovanie je vyžadovaných niekoľko vstupných súborov, ktoré obsahujú platné dáta. Fuzzer použije tieto súbory ako vzor pri generovaní. AFL je konzolová aplikácia a v unixových systémoch sa spúšťa nasledujúcim spôsobom:

```
$ ./afl-fuzz -i input_test_dir -o output_results_dir /path/to/program
```

Argumentom `-i` sa určí priečinok vstupných súborov a argument `-o` je priečinok, do ktorého sa budú zapisovať výstupy fuzzera.

Hlavné nevýhody AFL, ako aj iných brute-force fuzzerov, spočívajú v obmedzenom pokrytí kódu pri testovacích dátach, ktoré sú zabalené prostredníctvom kontrolných súčtov, kryptografických podpisov alebo kompresií [20].

2.3.2 VUzzer

VUzzer je inteligentný fuzzer, ktorého jadro tvorí evolučný algoritmus, podobne ako u AFL. Bol vytvorený za účelom testovania binárnych kódov. Experimentovaním na rôznych dátach bolo zistené, že VUzzer predčil AFL, a to generovaním menšieho počtu vstupov pri odhalení väčšieho počtu chýb. Popis fuzzera je prevzatý z článku „VUzzer: Application-aware Evolutionary Fuzzing“ [14].

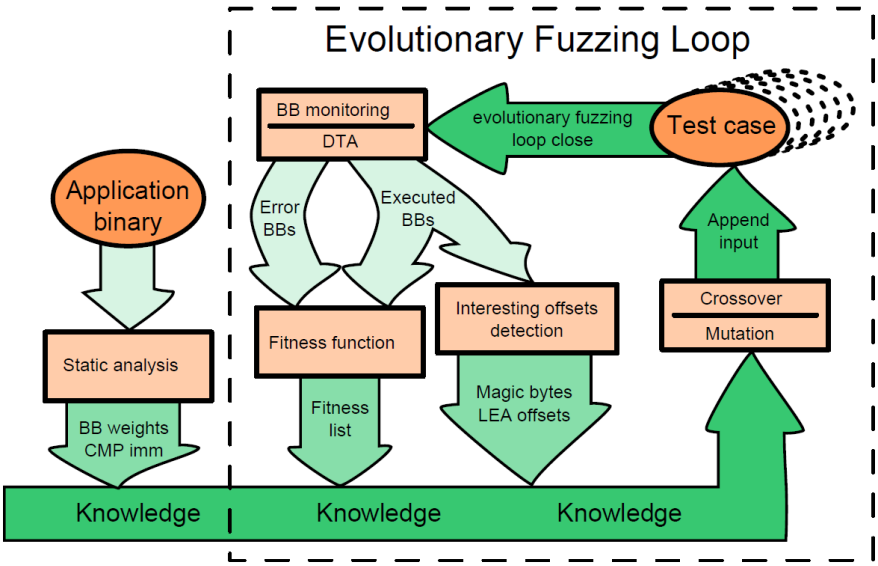
Analýzou stôp stavu a toku dát v bežiacej aplikácii dokáže fuzzer získať informácie o vzťahoch medzi vstupmi a výpočtami. Tieto znalosti zužitkuje pri generovaní. Na to pri analýze využíva napr. inštrukcie `cmp` a `lea`. Ich inštrumentáciou je možné dozvedieť sa, aké vstupné bajty sa s čím porovnávajú, resp. na akej adrese sa operand nachádza pri zisťovaní dátového typu premennej. Vďaka tomu sú magické bajty[16], ich hodnoty a typ, ľahko detekovateľné. Táto analýza sa nazýva angl. „taint analysis“.

Nevyhnutnou podmienkou na chod VUzzera je prítomnosť validných vstupných dát. Prechodom týchto vstupov testovanou aplikáciou je dynamický analyzátor stôp schopný určiť, ešte pred začiatkom generovania fuzz dát, offset magických bajtov s ohľadom na miesta prístupu do pamäte. Príkladom je hodnota `0xFDEF` zapísaná do premennej `buf`, ktorá sa porovnáva postupne najprv ako `buf[1] == 0xEF`, a potom ako `buf[0] == 0xFD`.

Vnorené bloky testovaného kódu a porovnávané hodnoty sú prvotne spracované statickým analyzátorom. Každému bloku kódu sa vypočíta váha podľa pravdepodobnosti toho, že sa daný kód vykoná. Ak je kód ťažko dosiahnuteľný dostáva väčšiu váhu. V genetickom algoritme je táto váha jedným z hlavných faktorov pri počítaní fitness funkcie.

[16] Konštantné numerické alebo textové hodnoty, ktorých význam nie je objasnený

Na obrázku 2.1 je schéma jadra VUzzera. Statický (ľavá časť) a dynamický (pravá časť) analyzátor predstavujú jeho hlavné komponenty.



Obr. 2.1: Prehľad schémy VUzzera. BB: základný blok, CMP imm: inštrukcia na porovnanie s jedným operandom, DTA: dynamická analýza toku, LEA: inštrukcia pre načítanie efektívnej adresy

2.3.3 SPIKE

SPIKE je framework poskytujúci API[17] na vývoj fuzzerov zameraných na sieťové protokoly. Okolo roku 2007 bol najpoužívanejším frameworkom. Neskôr sa kvôli jeho otvorenej licencii podarilo vyvinúť SPIKEfile, pozmenená varianta pre fuzzing súborov.

Štruktúra samotného formátu protokolu je rozdelená do niekoľkých blokov, ktoré obsahujú zároveň aj binárne dáta, aj veľkosť daného bloku. Bloky môžu byť vnorené do iných blokov a definujú sa nasledujúcim spôsobom [17]:

```
s_block_size_binary_bigendian_word("packet");
s_block_start("packet");
s_binary("01020304");
s_block_end("packet");
```

Príkazom `s_block_size_binary_bigendian_word()` si do fronty rezervujeme blok o veľkosti 4 bajtov. Potom si aktualizujeme ukazovateľ na miesto vo fronte, kde sa blok `packet` vyskytuje. Príkaz `s_binary()` vezme bajty 0x01020304 a vloží ich do rezervovanej fronty. Posledným príkazom ukončíme blok. Podobnou metódou sa definujú textové protokoly, a to napr. príkazmi `s_string_variable()` alebo `s_string()` [21]. Kde volaním trebárs `s_string_variable("COMMAND")` sa vytvára v bloku fuzz reťazec z pôvodne zadaného.

Framework je naprogramovaný v jazyku C. Slabá dokumentácia a takmer žiadna podpora pre systémy Windows sú veľkými nedostatkami tohto frameworku.

[17] Aplikačné programové rozhranie

2.3.4 Sulley

Sulley je jedným z ďalších frameworkov na vytváranie fuzzerov. Framework okrem generovania dát disponuje aj prostriedkami na monitorovanie cieľa a automatickú detekciu chýb. Naviac dokáže pracovať paralelne, čím sa zvyšuje jeho celková rýchlosť. Úvod a bližší popis frameworku vychádza z knihy „Fuzzing: Brute Force Vulnerability Discovery“ [17].

Reprezentácia dát je podobná ako v prípade SPIKE frameworku, rozčlenená do blokov. Pri generovaní dát je možné nastaviť číslicovým hodnotám, reťazcom alebo iným primitívam[18] ich názov, pre lepšie adresovanie, alebo povolenie pre fuzzing, teda pre mutovanie hodnoty. Zdrojový kód 2.6 demonštruje príklad vytvárania dát pre webový server pomocou frameworku Sulley. Zoskupením niekoľkých primitív získame jeden blok, na ktorom cyklíme cez všetky mutácie pre každú hodnotu zvlášť.

```
1  from sulley import *
2
3  s_initialize('HTTP BASIC')
4  s_group('verbs', values=['GET', 'HEAD', 'POST', 'TRACE'])
5
6  if s_block_start('body', group='verbs'):
7      s_delim(' ')
8      s_delim('/')
9      s_string('index.html')
10     s_delim(' ')
11     s_string('HTTP')
12     s_delim('/')
13     s_string('1')
14     s_delim('.')
15     s_string('1')
16     s_static('\r\n\r\n')
17 s_block_end('body')
```

Zdrojový kód 2.6: Vytváranie požiadavky pre webový server

Najprv volaním `s_initialize()` definujeme nový blok. Potom si na 4. riadku vytvoríme skupinu primitív, ktoré chceme, aby prechádzali mutovaním a následne inicializujeme blok `body`, ktorý napojíme na skupinu `verbs`. Príkaz `s_block_start()` vráti stále hodnotu `True`.

Na monitorovanie potrebujeme nainštalovať framework Sulley aj na cieľovom operačnom systéme, čo pri testovaní komplexnejších systémoch, ku ktorým nemáme prístup nie je možné. Sulley tiež obsahuje pomôcky pre počítanie kontrolných súčtov, umožňuje replikovať generované bloky a mnoho ďalšieho. Framework je naprogramovaný v jazyku Python.

Boofuzz[19] je jedným z úspešných následníkov frameworku Sulley. Oproti Sulley podporuje definovanie vlastných komunikačných medií, ľahko sa inštaluje a má menej chýb, nakoľko jeho vývoj pokračuje dodnes.

2.3.5 Peach

Peach je medzi-platformový fuzzer vyznačujúci sa vysokou flexibilitou. Tester si musí vopred vytvoriť niekoľko XML dokumentov, nazývaných „Pits“, ktoré detailne popisujú

[18] Druh dát, ktorý nie je derivovaný zo žiadneho iného dátového typu
[19] boofuzz https://github.com/jtpereyda/boofuzz

testovaný protokol a spôsob spustenia testu. Akýkoľvek súbor Pit môže v sebe zahrňovať počiatočné dátové bloky a špecifikovať polia, ktoré majú byť mutované. V porovnaní so SPIKE frameworkom je táto metóda definovania náročnejšia [6].

Prvá verzia nástroja Peach bol open-source[20] framework napísaný v jazyku Python. Dnes je vo verzii 3, má kompletne prepísané jadro v Microsoft .NET Framework[21] a k dispozícii už nie je jeho zdrojový kód.

Podobne, ako v predchádzajúcom prípade, na základe pôvodných verzií frameworku vznikli ďalšie. Kitty[22] je moderný framework, inšpirovaný fuzzerom Peach a frameworkom Sulley.

[20] Zdrojový kód je prístupný verejnosti pod licenciou, ktorá umožňuje jeho zdieľanie, modifikovanie atď.
[21] .NET Framework https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview
[22] Kitty https://github.com/cisco-sas/kitty

Kapitola 3

Protokol OData

Dáta boli v minulosti viazané na konkrétne typy formátov alebo aplikácií a bolo zložité ich využívať na iné než primárne určené prípady použitia. Snaha vytvoriť jednotnú normu umožňujúcu pristupovať k rozsiahlym dátam z rôznych zariadení či aplikácií, viedla k vzniku protokolu OData (Open Data Protocol). Protokol OData je webový protokol a bol postavený na základe existujúcich praktík HTTP a REST[1]. Protokol našiel využitie hlavne pri systémoch plánovania podnikových zdrojov, označovaných aj ako ERP[2], kde sa spracúvajú dáta z rôznych zdrojov.

Prvá verzia protokolu OData vznikla v roku 2007 v projekte Astoria[3]. Pozdejšie bola vyvinutá verzia 2.0, ktorá sa ešte dodnes používa vo väčšine moderných aplikácií spoločnosti SAP. Najnovšia verzia 4.0 bola v marci 2014 schválená štandardom OASIS[4]. Obe verzie a ich porovnanie bude načrtnuté v sekcii 3.3.

OData ako osobitá technológia pozostáva z nasledujúcich štyroch častí [3]:

- **Dátový model.** Popisuje spôsob a usporiadanie dát v rámci dátového modelu entít pomocou dokumentu metadát, viď sekciu 3.2.
- **Protokol OData.** Zabezpečuje komunikáciu medzi klientom a OData službou. Komunikácia je stavaná na metodológiách REST, čo znamená, že zasielané správy používajú metódy HTTP ako `POST`, `GET`, `PUT`, `DELETE` a iné. Jedná sa o tzv. CRUD (create, read, update, delete) operácie. Obsah predávaných dát je buď vo formáte Atom (založený na XML), alebo JSON. Zdroje dát sú určené jednotným identifikátorom, ďalej len URI, skladajúcim sa z niekoľkých častí. Viac informácií o adrese URI a spôsobe zasielania správ nájdeme v sekcii 3.1.
- **Klientské knižnice.** Zjednodušujú vytváranie softvéru, ktorý umožňuje rôznym klientom spracovávať dáta cez protokol OData. Dostupné knižnice sú spomenuté v sekcii 3.5.
- **OData služba.** Implementuje protokol OData a vytvára koncový bod pre prístup k dátam. Ako sa táto služba vytvára je objasnené v podkapitole 3.4.

Celý protokol OData je veľmi rozsiahly, preto budú v tejto kapitole uvedené len tie najdôležitejšie vlastnosti, potrebné na porozumenie základných princípov či na implemen-

[1] REpresentational State Transfer https://www.infoq.com/articles/rest-introduction
[2] SAP ERP https://www.sap.com/products/what-is-erp.html
[3] Projekt Astoria https://blogs.msdn.microsoft.com/odatateam/2007/07/18/welcome/
[4] OASIS OData https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=odata

táciu samotného testovacieho nástroja. Poznatky boli prebrané z oficiálnej dokumentácie [9] a štandardu OASIS [8].

3.1 Konvencie URI

Pojem URI vychádza z angl. „Uniform Resource Identifier“ a je definovaný v RFC 3986[5]. Prostredníctvom URI dokážeme v dátovom modeli adresovať kolekcie, ich vlastnosti, hodnoty atď. V OData je URI zložená z koreňa adresy služby, cesty k zdroju a možností dopytovania sa na dáta, pričom posledná časť je voliteľná. Ako ukážku si vezmime názornú adresu URI, kde môžeme vidieť spomínané rozdelenie:

```
https://odata.com/svc/Categories(15)/Products?$filter=ID lt 102
|----OData služba----|-------Zdroj dát-------|-----Otázka-----|
```

V príklade pristupujeme k entitám `Products`, ktoré sú naviazané na konkrétny druh kategórie `Categories`, identifikovanej číslom `15`. Identifikátor entity vieme prirovnať k primárnemu kľúču[6] v obyčajnej tabuľke. Všetky prislúchajúce entity potom dodatočne prefiltrujeme funkciou `$filter`. Touto funkciou sa vyberie podmnožina entít, ktoré vyhovujú predikátu, teda tomu, že hodnota vlastnosti `ID` je menšia ako `102`. Výrazy predikátov v sebe obsahujú vlastnosti entít alebo literály[7]. Všimnime si, že vo funkcii `$filter` sa nachádza operátor `lt` (z angl. „less than“). Takisto sú podporované ostatné operátory, ako napr. eq (equal), ne (not equal), gt (greater than), ge (greater or equal), lt (less than), le (less or equal), and (logical and), or (logical or) a not (logical negation). Zaslaním požiadavky HTTP `GET` s touto adresou URI nám server vráti entity v tele odpovedi.

Funkcia `$filter` sa vyznačuje vysokou variabilitou a použiteľnosťou. Okrem vyššie spomínaných operátorov podporuje použitie funkcií. Tabuľka 3.1 obsahuje výňatok z podporovaných funkcií.

Tabuľka 3.1: Funkcie, ktorými disponuje funkcia `$filter`

Funkcia	Príklad použitia
Reťazcové funkcie	
bool substringof(string p0, string p1)	$filter=substringof('Bevera', FoodType) eq true
bool endswith(string p0, string p1)	$filter=endswith(FoodType, 'age')
bool startswith(string p0, string p1)	$filter=startswith(FoodType, 'Bev')
int length(string p0)	$filter=length(FoodType) eq 10
string concat(string p0, string p1)	$filter=concat(City, ' is here') eq 'Brno is here'
Matematické funkcie	
double round(double p0)	$filter=round(FoodWeight) eq 32d
double floor(double p0)	$filter=floor(FoodWeight) eq 32

Funkcia `$filter` naviac umožňuje od seba oddeliť niekoľko logických častí pomocou zátvoriek. Jednej logickej časti je tak možné priradiť vyššiu prioritu a prednosť pri spracovávaní samotnej otázky. Ako príklad si zoberme nasledujúcu adresu URI:

[5] URI RFC https://tools.ietf.org/html/rfc3986
[6] Hodnota, ktorá jednoznačne identifikuje záznam v tabuľke
[7] Symbol predstavujúci konštantnú hodnotu

```
https://odata.com/svc/Items?$filter=(ID lt 10 or ID gt 20) and Price lt 20
```

Najprv sa vyhodnotí výraz `ID lt 10 or ID gt 20` a až potom jeho výsledok spolu s výrazom `and Price lt 20`. V prípade neprítomnosti zátvoriek by sa najprv vyhodnotila časť `ID gt 20 and Price lt 20`, pretože operátor `and` má vyššiu prioritu vyhodnocovania, čím dostaneme úplne odlišný výsledok.

Pre dopytovanie sa existujú okrem funkcie `$filter` ešte ďalšie, sú to: `$orderby` (zoradenie dát podľa hodnôt vlastností entity), `$top` (zobrazenie prvých N výsledkov), `$skip` (vynechanie prvých N položiek), `$expand` (rozšírenie odpovedi o vzťahujúcu sa entitu), `$format` (stanovenie formátu odpovede, JSON alebo XML), `$select` (projekcia vybraných vlastností entity), a iné. Jednotlivé funkcie môžu byť medzi sebou kombinované oddeľovačom `&`:

```
https://odata.com/svc/Products?$filter=ID lt 10&$top=5&$orderby=Price asc
```

Entity sú v dátovom modeli prepojené a jestvujú medzi nimi vzťahy, rovnako ako v každej relačnej databáze[8]. Ak by sme chceli zistiť, aká entita `Category` je previazaná s entitou `Products`, identifikovanou hodnotou `1`, vznikla by nám takáto adresa URI:

```
https://odata.com/svc/Products(1)/$links/Category
```

Cez adresu URI vieme naviac sprístupniť konkrétne hodnoty vlastností. Je to zvlášť vhodné pri ich aktualizáciách, aby sa nemusel zapisovať kompletný obsah entity. Čitateľovi sa môže pozdávať, že takto zmeníme len jej jednu časť. V skutočnosti sa na pozadí aktualizuje celá entita. Hodnoty sprístupníme kľúčovým slovom `$value`. Znenie HTTP požiadavky by vyzeralo s použitím metódy HTTP `PUT` nasledovne:

```
PUT /svc/Products(1)/UnitPrice/$value HTTP/1.1 Host: odata.com
DataServiceVersion: 1.0 MaxDataServiceVersion: 2.0 accept: application/xml
content-type: text/plain Content-Length: 2 25
```

V hlavičke správy máme zaradom: 1. adresu URI pre danú hodnotu, 2. verziu protokolu HTTP (1.1), 3. názov hosťovského servera (odata.com), 4. verziu protokolu OData (1.0 až 2.0), 5. druh obsahu, ktorému klient rozumie (XML), 6. druh obsahu tela správy (prostý text), 7. dĺžka obsahu tela správy (2). Telo správy predstavuje hodnota 25, ktorá sa zapíše v dátovom modeli na miesto určené adresou URI.

Protokol OData podporuje tiež zasielanie tzv. „Batch“ požiadaviek. Jedná sa o zoskupenie viacerých operácií vkladania, čítania, upravovania alebo mazania v rámci jednej HTTP požiadavky. Využitie pre tento špeciálny typ požiadaviek nájdeme najmä pri dopytovaní sa na viaceré entity z rôznych adries URI.

3.2 Dokument metadát

Všeobecne sú metadáta akýmsi popisom iných dát, čo v konečnom dôsledku uľahčuje prácu so samotnými dátami. Ako prirovnanie si zoberme obrázok v elektronickej forme. Údaje o veľkosti či rozlíšení sú uložené práve v metadátach, ktoré sú zahrnuté buď priamo v súbore alebo samostatne. Aplikácia slúžiaca na upravovanie obrázkov vie z metadát zistiť, akú veľkosť pamäte je potrebné vymedziť pre jej správny chod a pod. Podobne je to aj s OData službami, kde takéto informácie využívajú klientské aplikácie. OData služby disponujú dvoma typmi metadát:

[8] Databáza zložená z viacerých tabuliek, ktoré majú medzi sebou vzájomné väzby

- **Dokument služby.** Obsahuje výpis entít v najvyšších doménach. Klient je tak schopný nájsť adresu každej z nich použitím URI pozostávajúcej len z koreňa adresy OData služby.
- **Dokument metadát.** Popisuje dátový model, jeho štruktúru a organizáciu zdrojov. V tejto podkapitole sa budeme venovať variante pre OData verzie 2.0 a 3.0.

Základným konceptom dátového modelu entít, angl. „Entity Data Model“, ďalej len EDM, sú entity a ich asociácie. Každá entita má svoj typ (v metadádach označovaný ako `EntityType`) a vlastnosti (pomenované ako `Property`). Entity zhodného typu sú zoskupené do tzv. sady entít (`EntitySet`). Rovnako to platí pre asociácie, tie sú združené do skupiny asociácií (`AssociationSet`).

Metadáta sú zabalené do EDMX, čo je v skratke EDM špecifikovaný vo formáte XML. Koreňový element je `edmx:Edmx` a má jedného potomka `edmx:DataServices`, v ktorom je schéma, ako môžeme vidieť v príklade 3.1 nižšie. Schéma je hlavným prvkom dátového modelu. Obsahuje už konkrétne definície asociácií, typov entít, anotácií atď.

```
<edmx:Edmx xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
    <edmx:DataServices m:DataServiceVersion="2.0">
        <Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
            Namespace="DataModel" xml:lang="en" sap:schema-version="1">
            ...
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

Zdrojový kód 3.1: Koreň dokumentu metadát

Dokument metadát je možné získať cez URI, pridaním kľúčového slova `$metadata` za adresou OData služby. Zdrojový kód 3.2 znázorňuje definovanie typu entity v dokumente metadát a vychádza z databázy NorthWind[9].

```
<EntityType Name="Order_Subtotal">
    <Key>
        <PropertyRef Name="OrderID"/>
    </Key>
    <Property Name="OrderID" Type="Edm.Int32" Nullable="false"/>
    <Property Name="Subtotal" Type="Edm.Decimal" Nullable="true"
        Precision="19" Scale="4"/>
</EntityType>
```

Zdrojový kód 3.2: Metadáta typu entity `Order_Subtotal`

Vidíme, že typ entity v sebe zahŕňa niekoľko položiek. Element `Key` obsahuje odkazy na kľúčové vlastnosti, ktorými je entita identifikovaná. Entita `Order_Subtotal` má jeden kľúčový prvok `OrderID`. Vlastnosti entity majú oddelene určený ich dátový typ tagom `Type`. Pre `OrderID` je to 32-bitové číslo. Pre `Subtotal` je to fixné desatinné číslo, pričom desatinná časť môže byť tvorená maximálne štyrmi číslicami (`Scale`) a celková hodnota nesmie prekročiť 19 číslic (`Precision`). Navyše atribútom `Nullable` stanovíme, či je hodnota pri inštancii, resp. vytváraní novej entity požadovaná. Ak áno, atribút nastavíme ako `true`, inak ako `false`.

[9] NorthWind OData http://services.odata.org/V2/Northwind/Northwind.svc/$metadata

Toto však nie sú všetky možnosti, ktoré vieme vlastnostiam priradiť. Ich hodnoty smú byť napr. typu `Edm.String` (reťazec), `Edm.Byte` (8-bitové číslo bez znamienka), `Edm.Double` (desatinné číslo s veľkou presnosťou) alebo `Edm.DateTime` (dátum). U reťazca atribútom `MaxLength` definujeme jeho maximálnu povolenú dĺžku. OData 3.0 oproti verzii 2.0 podporuje špeciálne typy pre geopriestorové vlastnosti[10], nimi sa zaoberať nebudeme.

Vlastnosti nemusia nadobúdať len hodnoty primitívnych dátových typov. V prípade, že chceme vytvoriť dátový typ zložený z viacerých typov, použijeme tag `ComplexType`, viď ukážku 3.3.

```
<ComplexType Name="Address">
    <Property Name="Street" Type="Edm.String"/>
    <Property Name="City" Type="Edm.String"/>
    <Property Name="State" Type="Edm.String"/>
    <Property Name="ZipCode" Type="Edm.String"/>
    <Property Name="Country" Type="Edm.String"/>
</ComplexType>
```

Zdrojový kód 3.3: Metadáta zloženého dátového typu `Address`

Hodnotu konkrétnej položky zloženého dátového typu potom v adrese URI sprístupníme vo formáte `Address/City`:

```
https://odata.com/svc/Products(1)/Address/City/$value
```

Nie vždy poskytujú anotácie, resp. atribúty dostatočné informácie o vlastnostiach či entitách. Sú to najmä informácie o tom, či je vlastnosť súca na použitie v opytovacích funkciách alebo je možné ju upravovať alebo vytvárať. Platí to nielen pre vlastnosti, ale aj pre entity. Spoločnosť SAP používa v metadátach vlastné atribúty s prefixom `sap:`, ktoré riešia uvedené problémy. Atribúty nájdeme podrobne zdokumentované na stránke [5].

```
<EntityType Name="CustomerInvoice" sap:content-version="1">
    <Key>
        <PropertyRef Name="CustomerId"/>
    </Key>
    <Property Name="CustomerId" Type="Edm.String" Nullable="false"
        MaxLength="32" sap:unicode="false" sap:label="Key"
        sap:creatable="false" sap:updatable="false" sap:sortable="false"/>
    <Property Name="Category" Type="Edm.String" Nullable="false"
        sap:unicode="false" sap:label="Data" sap:creatable="false"
        sap:updatable="false" sap:sortable="false" sap:filterable="false"/>
</EntityType>
```

Zdrojový kód 3.4: Metadáta typu entity `CustomerInvoice`

Vyššie uvedený kód 3.4 demonštruje väčšinu týchto atribútov pre vlastnosti entity. Atribút `sap:updatable` určuje, či sa hodnota môže pri aktualizáciách meniť. Atribútom `sap:creatable`, nastavenom do hodnoty `false`, sa dáva do povedomia, že hodnota vlastnosti bude pri jej vytváraní zvolená serverom a nie klientom. Najzaujímavejšou častou ukážky sú atribúty `sap:sortable` a `sap:filterable`, pretože presne ony budú v tejto práci použité pri implementácií pravidiel pre fuzzer. Atribút hovorí klientovi, že sa vlasnosť nesmie vyskytovať vo funkcii `$orderby`, resp. `$filter`.

[10] Popisujú dáta, ktoré sa vzťahujú na určité miesto

V ukážke 3.5 máme názorný príklad použitia SAP atribútov pri deklarácii sady entít. Atribútom `sap:countable` sa definuje možnosť použitia sady entít spolu s opytovacími funkciami `$skip` a `$top`. Atribút `EntityType` určuje typ, ktorý budú entity niesť.

```
<EntitySet Name="Employees" EntityType="DataModel.Employee"
    sap:updatable="false" sap:creatable="false"
    sap:content-version="1" sap:countable="false"/>
```

Zdrojový kód 3.5: Deklarácia sady entít typu `Employee`

Asociácie, značené tagom `Association`, definujú vzťahy medzi entitami. Zdrojový kód 3.6 zobrazuje vzťah medzi dvoma typmi entitít, nákupným košíkom `Cart` a položkami `Item`. V jednej relácii musia byť stále len dva entity. Kardinalita[11] vzťahu je daná atribútom `Multiplicity`. Na jeden košík takto pripadá 0 až N položiek. `Role` je atribút, ktorý identifikuje postavenie entity vo vzťahu. Naviac pri asociáciách popisujeme obmedzenia. Tie sa týkajú závislostí jednej entity nad druhou. Hlavná entita je v príklade nákupný košík, naopak na nej závislá je entita `Item`. Oba typy entít sú v relácii rozpoznávané kľúčom `CartId`, resp. `ItemId`.

```
<Association Name="toItem" sap:content-version="1">
    <End Type="DataModel.Cart" Multiplicity="1" Role="FromRole_toItem"/>
    <End Type="DataModel.Item" Multiplicity="*" Role="ToRole_toItem"/>
    <ReferentialConstraint>
        <Principal Role="FromRole_toItem">
            <PropertyRef Name="CartId"/>
        </Principal>
        <Dependent Role="ToRole_toItem">
            <PropertyRef Name="ItemId"/>
        </Dependent>
    </ReferentialConstraint>
</Association>
```

Zdrojový kód 3.6: Metadáta asociácie `toDataEntity`

OData služba vystavuje klientskej aplikácii aj tzv. servisné operácie. Sú to funkcie, ktorých sémantiku[12] určuje jej autor. Servisné operácie môžu prijímať parametre vo forme primitívnych dátových typov, kolekcií primitív, zložených dátových typov, ich kolekcií, entít alebo kolekcií entít. Zdrojový kód 3.7 je ukážkou definovania servisnej operácie v dokumente metadát.

```
<FunctionImport Name="GetProductsByRating" EntitySet="Products"
    ReturnType="Collection(ODataDemo.Product)" m:HttpMethod="GET">
    <Parameter Name="rating" Type="Edm.Int32" Mode="In"/>
</FunctionImport>
```

Zdrojový kód 3.7: Metadáta servisnej operácie `GetProductsByRating`

Atribút `ReturnType` deklaruje návratový typ funkcie. V našom prípade to je kolekcia produktov. Atribútom `m:HttpMethod` sa určí metóda HTTP, ktorá bude použitá.

Servisné operácie sprístupníme pomocou adresy URI, podobne ako to bolo pri kolekciách entít. Ak chceme získať všetky produkty, ktorých hodnotenie sa rovná 4, použijeme nasledujúcu adresu URI:

[11] Maximálny počet vzťahov, na ktorých sa entita v rámci jedného typu zúčastňuje
[12] Význam jednotlivých parametrov funkcie

```
https://odata.com/svc/GetProductsByRating?rating=4
```

3.3 Verzie protokolu

OData 2.0 je predchodcom verzie 3.0, a tá je predchodcom verzie 4.0. Protokol OData 3.0 pridáva do štandardu niekoľko ďalších funkcií. Podstatné rozdiely sú:

- **Podpora metódy HTTP PATCH.** Metóda má rovnaké chovanie, ako HTTP `MERGE`, ktorá bola pre OData 1.0 a 2.0 pôvodne predstavená na zlučovanie zmien. Menia sa len tie polia, ktoré sa nachádzajú v požiadavke.
- **Použitie naviazaných entít v otázkach.** Po novom je možné do výrazu funkcie `$filter` vložiť asociované entity:

  ```
  https://odata.com/svc/Orders?$filter=Customer/ContactName ne 'Fred'
  ```

- **Funkcie `all()` a `any()`.** Parameter funkcie je výraz v tvare `Entity e: predicate`, kde `e` je lambda parameter[13] a `predicate` je predikát. Predikátom je napr. `e/Name eq 'Alfred'`. Je splnený vtedy, keď platia všetky podmienky (`all`), resp. len niektorá (`any`). Tak, ako v predchádzajúcom bode, aj toto je rozšírenie funkcie `$filter`.
- **Funkcie a akcie asociované s entitami.** Akcie a funkcie sú operácie na globálnej úrovni v rámci OData služby a môžu byť súčasťou entít.
- **Nový formát JSON.** Nazývaný ako Verbose JSON Format. Navyše sa JSON stal predvoleným formátom pre dáta.

Razantné rozdiely ponúkla až verzia 4.0. Medzi hlavné patria:

- **Podpora funkcie $search v adrese URI.** Táto funkcia je implementovaná aj vo verzii 2.0, ale len v službách spoločnosti SAP. Jedná sa o vyhľadávanie naprieč kolekciami entít. Keby sme chceli vypísať všetky entity, ktoré obsahujú slovo „Fred“, použijeme takúto adresu URI:

  ```
  https://odata.com/svc/Orders?$search=Fred
  ```

- **Vytváranie entity cez metódu HTTP PATCH alebo PUT.** Nový záznam sa vytvorí adresou URI a dodaním všetkých potrebných údajov v tele správy.
- **Adresovanie referencií kľúčovým slovom $ref.** Referenciu na entitu vytvárame pridaním `/$ref` za jej adresou. Okrem toho je to kompletná náhrada pre `$links`.
- **Aplikovanie funkcie $count v iných funkciách.** Klient je schopný filtrovať výsledky napr. podľa počtu entít.
- **Nové typy Edm.Date, Edm.Duration a Edm.TimeOfDay.** Prvý z menovaných je dátum bez časovej stopy, druhý vyjadruje dobu trvania a tretí čas pre jeden deň. Definovanie nových typov malo za následok odstránenie tých pôvodných, a to `Edm.DateTime`, `Edm.Time`, a na to naväzujúcich funkcií.

[13] Anonymná funkcia

- **Osobitná entita zvaná jedináčik.** Slúži na reprezentáciu entity, ktorá smie byť v modeli len jedenkrát. Entita je označená tagom `Edm:Singleton`.
- **Odstránenie asociácií.** Dôvodom sú navigačné elementy, ktoré majú rovnaký význam ako asociácie. Sú to tagy `NavigationProperty` a `NavigationPropertyBinding`.

Spoločnosť SAP chce v blízkej budúcnosti úplne prejsť na verziu 4.0. Terajšie služby bežia vo veľkej miere na protokole OData 2.0.

3.4 SAP Gateway

Táto časť je venovaná základným princípom vytvárania OData služieb v systémoch SAP. Podkapitola vychádza z knihy „OData and SAP NetWeaver Gateway“ [1].

SAP Gateway je technológia poskytujúca rozhranie, ktoré implementuje prístup do systémov SAP cez protokol OData. Vďaka tomu nemusia mať vývojári na rôznych vrstvách aplikácie vedomosti o konkrétnych metódach a funkciách, pretože komunikácia sa koná len na úrovni dát a procesov.

OData služba sa vytvára cez aplikáciu *SAP Gateway Service Builder*. Spúšťa sa transakciou[14] *SEGW* v SAP GUI[15]. Aplikácia sa vyznačuje podporou pre budovanie služieb deklaratívnym spôsobom alebo použitím už existujúcich objektov[16]. Entity, ich vlastnosti, typy a asociácie sa dajú vytvoriť manuálne alebo importovať z databázy a pomocou nich sa automatizovane vytvorí model služby.

SAP Gateway Service Builder generuje triedy nazývané angl. Model Provider Class (MPC) a Data Provider Class (DPC). MPC obsahuje kód, ktorý deklaruje dátový model služby. Implementácie operácií nad dátami sú v DPC. Dokument metadát sa generuje automaticky, poprípade je možné ho importovať. Pokročilé vlastnosti alebo funkcie služby je nutné naprogramovať, čoho výsledkom je tvorba tried v jazyku ABAP[17].

Vytvorená služba sa môže testovať v *SAP Gateway Client*. Ide o klientskú aplikáciu vstavanú priamo do SAP Gateway. Ňou posielame na danú OData službu trebárs požiadavky ako `POST` a `GET` pre overenie základnej funkcionality.

SAP Gateway taktiež transformuje HTTP požiadavky do štruktúr prezentovaných v jazyku ABAP. Štruktúry sa používajú pri čítaní a spracovávaní požiadavky na serverovej časti aplikácie. Najčastejšie používané štruktúry sú `is_paging` a `io_tech_request_context`. Použitie jednej z nich si ukážeme v sekcii 3.4.2.

3.4.1 Model Provider Class

MPC je trieda v jazyku ABAP a definuje EDM OData služby. To, čo sa nachádza v dokumente metadát, je programovo zapísané do MPC. Po akejkoľvek zmene modelu musíme generovať MPC znova. Pri vytváraní služby cez SAP Gateway vzniknú dve triedy, pričom jedna je hlavná a druhá je rozširujúca. V rozširujúcej je možné predefinovať metódy tej hlavnej na základe konceptu dedičnosti. Programátor väčšinou nezasahuje do vygenerovaných metód. Z pohľadu tejto práce sa nebudeme sústrediť na MPC, je ale dôležité spomenúť, že niečo podobné existuje.

[14] Kód, ktorý slúži k rýchlemu prístupu k funkciám alebo aplikáciám SAP
[15] Klient SAP GUI https://wiki.scn.sap.com/wiki/display/ATopics/SAP+GUI+Family
[16] Objekty sú reprezentáciou akéhokoľvek podnikateľského subjektu, napr. zamestnanec či objednávka
[17] ABAP https://www.sap.com/developer/topics/abap-platform.html

3.4.2 Data Provider Class

DPC poskytuje metódy potrebné pre spracovanie žiadostí od klienta. V DPC sa vykonávajú operácie ako `CREATE`, `READ`, `UPDATE` a `DELETE`. Podobne ako v predchádzajúcom prípade, aj tu vytvára *SAP Gateway Service Builder* dve triedy, hlavnú a vedľajšiu. Programátor implementuje konkrétne metódy pre čítanie entít `get_entityset()` či funkcionalitu napr. pre opytovacie funkcie `$orderby`, `$top`, `$skip` a `$filter` v rozširujúcej triede.

Zdrojový kód 3.8 zobrazuje metódu pre jednoduché načítanie entít produktov. Funkciou `BAPI_EPM_PRODUCT_GET_LIST` sa získajú všetky produkty z databázy. Premenná `is_paging` je štruktúra, ktorá obsahuje hodnoty funkcií `$top` a `$skip` z adresy URI. Produkty sa v rozsahu od `lv_start` do `lv_end` namapujú v cykle na riadku 17 až 21 do tabuľky `et_entityset`, ktorá je exportovaná ku klientovi. Ak klient pošle požiadavku, v ktorej bude výraz `?$top=0&$skip=10`, cyklus sa nevykoná a tabuľka ostane prázdna.

```
1  method products_get_entityset.
2      data lt_headerdata type standard table of bapi_epm_product_header.
3      data(lv_start) = 1.
4      data(lv_end) = 1.
5
6      call function 'BAPI_EPM_PRODUCT_GET_LIST'
7          tables headerdata = lt_headerdata
8
9      if is_paging-skip is not initial.
10         lv_start = is_paging-skip + 1.
11     endif.
12     if is_paging-top is not initial.
13         lv_end = is_paging-top + lv_start - 1.
14     else.
15         lv_end = lines( lt_headerdata ).
16     endif.
17     loop at lt_headerdata assigning field-symbol(<fs_headerdata>)
18         from lv_start to lv_end.
19         append corresponding #(
20             <fs_headerdata> mapping productid = product_id
21         ) to et_entityset.
22     endloop.
23 endmethod.
```

Zdrojový kód 3.8: Metóda pre načítanie entít z databázy

Celý kód vyššie uvedenej metódy je ručne písaný programátorom. V prípade zlej implementácie sa môžu klientovi vrátiť dáta, o ktoré si ani nežiadal. Zložitejšie filtrovanie entít častokrát počíta s viacerými otázkami v adrese URI a vetvenie programu nie je na prvý pohľad tak zrejmé. Kvôli tomu vznikajú chyby, ktoré môžu spôsobiť zrútenie programu len jedenkrát z niekoľko tisíc vstupov.

Práve toto je hlavná motivácia pre vytvorenie automatizovaného nástroja, ktorý dokáže generovať sofistikované adresy URI a dáta za účelom objavenia chýb v implementáciách pre spracovanie klientského vstupu.

3.5 Dostupné knižnice

Protokol OData je dnes podporovaný veľkou škálou knižníc. Výhodou použitia takýchto knižníc je to, že obsahujú implementácie na čítanie dokumentu metadát alebo na zasielanie požiadaviek. Existujú knižnice pre jazyky ako C#, Java, JavaScript, C++ alebo Python. Všetky tieto knižnice dokážu transformovať reprezentáciu modelu a entít do objektov daného programovacieho jazyka, čím sa pri ich používaní uľahčí programátorovi práca.

Nie však každá knižnica a programovací jazyk je vhodný na použitie v testovacích systémoch, ktoré používa spoločnosť SAP. Príkladom je .NET Framework, ktorý má síce najlepšiu podporu, ale spôsoboval by ťažkosti pri používaní s inými existujúcimi a automatizovanými nástrojmi z dôvodu zavádzania nového virtuálneho systému do obehu.

3.5.1 Apache Olingo

Olingo je robustná knižnica, ktorá umožňuje implementovať ako klientskú aplikáciu, tak aj aplikáciu bežiacu na serveri. Podporuje protokol OData 2.0 a 4.0. Knižnica je napísaná v jazyku Java a disponuje viacero príručkami pre vytváranie aplikácií [18].

Dokument metadát sa číta volaním metódy `readMetadata()`, ktorá deserializuje[19] súbor EDMX do EDM. Parametrami metódy sú metadáta reprezentované vo formáte XML a prepínač, ktorý určuje, či má prebehnúť kontrola pre validitu metadát. Nižšie uvedený príkaz zobrazuje volanie tejto funkcie, pričom `connection` je inštancia triedy `HttpURLConnection`, ktorá sa nachádza v štandardnej knižnici jazyka Java. Cez `connection` vytvoríme požiadavku `GET` a pripojíme sa k vzdialenému serveru. Metódou `getContent()` sa získa obsah na danej adrese URI, v našom prípade sú to metadáta vo formáte XML.

```
Edm edm = EntityProvider
        .readMetadata((InputStrem) connection.getContent(), false);
```

Hodnoty entít čítame metódou `readEntry()`. Predom na to potrebujeme inicializovať objekt `EdmEntityContainer`. V dokumente metadát sa tagom `EntityContainer` označuje rodičovský element, ktorého potomkovia sú sady entít. V kóde nižšie sa na prvom riadku nachádza volanie pre získanie predvoleného kontajnera entít. Príkazom na druhom riadku už čítame entity s názvom „Products“. Zoznam vlastností dostaneme volaním `entry.getProperties()`.

```
EdmEntityContainer entityContainer = edm.getDefaultEntityContainer();
ODataEntry entry = EntityProvider
                   .readEntry("application/json",
                              entityContainer.getEntitySet("Products"),
                              inputStream,
                              EntityProviderReadProperties.init().build());
```

3.5.2 OpenUI5

OpenUI5 je otvorená verzia knižnice SAPUI5[20], ktorá sa zákazníkom ponúka len pod určitou licenciou. Jedná sa o knižnicu implementovanú v jazyku JavaScript a bola navrhnutá

[18] Olingo OData 2.0 dokumentácia https://olingo.apache.org/doc/odata2/index.html
[19] Spracuje a rekonštruuje dáta do pôvodného objektu
[20] Knižnica SAPUI5 https://www.sap.com/developer/topics/ui5.html

na vytváranie interaktívnych biznis aplikácií. Rovnako ako knižnica Olingo, aj OpenUI5 podporuje OData 2.0 a 4.0.

Moduly sú definované vo formáte AMD[21], ktorý je vhodný na vývoj klientských aplikácií pre internetové prehliadače. Toto je nevýhoda, ktorá posúva do popredia ostatné knižnice, pretože formát AMD sa nedá použiť s interpretom Node.js[22]. Node.js sa ľahko inštaluje do automatizačného servera Jenkins[23]. Jenkins používa spoločnosť SAP v rámci kontrolovania a integrácie projektov, preto berieme pri výbere knižnice do úvahy aj túto skutočnosť.

3.5.3 Pyslet

Knižnica je napísaná v jazyku Python a implementuje niekoľko štandardov. Medzi nimi je aj OData 2.0. Pyslet poskytuje podporu pre vytváranie klientskej a serverovej aplikácie.

```
1 from pyslet.odata2.client import Client
2 from pyslet.odata2.csdl import NavigationProperty
3
4 c = Client('http://services.odata.org/V2/Northwind/Northwind.svc/')
5 for value in c.feeds['Products'].entityType.values():
6     if type(value) is not NavigationProperty:
7         print(value.name + ':' + value.type)
```

Zdrojový kód 3.9: Zobrazenie názvov všetkých produktov

Zdrojový kód 3.9 je názornou ukážkou práce s touto knižnicou. Spustením kódu sa do konzoly vytlačí názov a typ každej vlastnosti entity. Najprv vytvoríme na riadku 4 objekt `c`, ktorý je inštanciou triedy `Client`. V cykle `for` prechádzame vlastnosťami entity, ktoré sú definované v dokumente metadát pre entity `Products`. Vlastnosti, ich typ, dĺžka atď. sú určené, ako iste vieme, v type entity, označovanej tagom `EntityType`. Knižnica Pyslet preto umožňuje prístup k hodnotám atribútu cez objekt s intuitívnym názvom `entityType`. Navigačné elementy, ktoré zapadajú tiež medzi vlastnosti, preskakujeme a na štandardný výstup sa nevytlačia. Tie kontrolujeme na riadku 6.

Ani táto knižnica nebude použitá vo finálnej implementácii fuzzera. Hlavným dôvodom boli ťažkosti sprevádzané s pripojením sa na vzdialený server cez nastavenia Proxy spolu so základným overovaním. Proxy server zabezpečuje prístup k interným systémom SAP.

3.5.4 PyOData

Poslednou knižnicou, ktorá bude predstavená je PyOData. Táto knižnica bola vyvinutá zamestnancami spoločnosti SAP a ponúka oproti ostatným knižniciam podporu pre SAP atribúty. PyOData je pomerne nová knižnica. Zdrojový kód prvej verzie bol zverejnený v druhom kvartáli roku 2018.

Kód 3.10 je príkladom toho, že použitie knižnice PyOData je naozaj veľmi jednoduché. Skript po spustení zobrazí priezviská zamestnancov. Na riadku 6 si podobne, ako u knižnice Pyslet, vytvoríme inštanciu triedy `Client` a následne si uložíme zoznam všetkých entít zamestnancov `Employees` do premennej `employees`. Cyklom na riadku 8 iterujeme cez jednotlivé entity, pričom z dátového modelu sú vyberané len priezviská zamestnancov volaním

[21] Asynchronous module definition http://requirejs.org/docs/whyamd.html
[22] Node.js https://nodejs.org/en/
[23] Jenkins https://jenkins.io/

`select('LastName')`. Metóda `execute()` nám vráti objekty obsahujúce priezviská, ktoré vytlačíme na výstup.

```
1 import requests
2 import pyodata
3
4 SERVICE_URL = 'http://services.odata.org/V2/Northwind/Northwind.svc/'
5
6 client = pyodata.Client(SERVICE_URL, requests.Session())
7 employees = client.entity_sets.Employees.get_entities()
8 for employee in employees.select('LastName').execute():
9     print(employee.LastName)
```

Zdrojový kód 3.10: Vytlačenie priezvisk všetkých zamestnancov

Knižnica PyOData je naprogramovaná v jazyku Python 3 a momentálne disponuje plnou podporou pre OData 2.0. Knižnica PyOData bola z dôvodu podpory SAP atribútov a jednoduchého rozhrania použitá pri implementácii kľúčových prvkov testovacieho nástroja.

Kapitola 4

Návrh fuzzera

V tejto kapitole si objasníme štruktúru navrhnutého fuzzera. Navrhnuté riešenie je dopodrobna popísané v sekciách 4.3 až 4.6. Pred samotným začiatkom si ešte v sekcii 4.1 priblížime existujúce riešenia a v sekcii 4.2 prevedieme analýzu problematiky testovania.

4.1 Existujúce riešenia

Napriek tomu, že protokol OData a fuzz testovanie sú relatívne známe pojmy, v ich spoločnom kontexte existuje na trhu len jeden program. Volá sa *Oyedata*[1]. *Oyedata* je testovací nástroj postavený na princípe black-box testovania. Je naprogramovaný v jazyku C# a má prívetivé grafické užívateľské rozhranie. Silnou stránkou tohto nástroja je automatické spracovávanie dokumentu metadát a generovanie šablón pre rôzne typy HTTP požiadaviek, ako `GET`, `PUT` či `POST`, a to pre všetky typy entít a ich vlastností. Používateľ si tak doplní do šablóny potrebné dáta a odošle ich na server. Jedna vygenerovaná predloha HTTP `GET` požiadavky vyzerá takto:

```
GET /Regions(RegionID=Edm.Int32) HTTP/1.1
Host: services.odata.org:80
Accept: application/json
Content-Type: application/json
```

Riadky predstavujú jednotlivé položky hlavičky HTTP. Hodnotu `Edm.Int32` zameníme za číslo v platnom rozsahu a dostaneme plnohodnotnú požiadavku. Generované šablóny môžu byť vstupom pre fuzzer, preto sa táto aplikácia v tomto kontexte aj spomína. Fuzzer by napríklad systematicky vyhľadával kľúčové slová dátových typov a zamieňal ich za odpovedajúce hodnoty.

Nevýhodou nástroja je žiaľ nutnosť manuálneho ovládania a chýbajúca podpora pre opytovacie funkcie. Nástroj *Oydata* beží len pod operačným systémom Windows, čo môžeme považovať tiež za ďalšiu z jeho slabších stránok.

4.2 Analýza problematiky

Slabá a takmer žiadna snaha o fuzz testovanie aplikácií komunikujúcich prostredníctvom OData protokolu, viedla autora tejto práce, v spolupráci so spoločnosťou SAP, k navrhnutiu vlastného riešenia. Navrhnutý nástroj sa volá ODfuzz. Vznikol spojením dvoch skrátených

[1] Oyedata https://www.mcafee.com/us/downloads/free-tools/oyedata.aspx

slov, a to OData (protokol, cez ktorý sa budú požiadavky posielať) a fuzzing (druh testovania, ktorým nástroj disponuje). Pri návrhu testovacieho nástroja sme brali do úvahy niekoľko nasledujúcich faktorov:

- **Spôsob fuzz testovania.** Práca s čisto náhodnými dátami je vo webovom prostredí neprípustná. Preto je potrebné generovať také dáta, ktoré sú náhodné len na určitých miestach a spĺňajú všetky obmedzenia protokolu. Celý proces fuzz testovania bude spočívať vo vytváraní HTTP `GET` požiadaviek a kontrolovaní odozvy zo servera. Požiadavky by mali byť dostatočne zložité a ich štruktúra by sa mala obmeňovať, aby bola pokrytá čo najväčšia časť množiny vstupného stavového priestoru.

- **Možnosti protokolu OData.** Pri testovaní sa môžeme zamerať na vytváranie rozsiahlych filtrovacích otázok, zložených z viacerých logických častí či ich zoskupení. Kombináciou opytovacích funkcií, ako napr. `$orderby`, `$skip` alebo `$top`, spolu s funkciou `$filter` sa vytvárajú komplexné otázky, ktoré generujú aj moderné Fiori[2] aplikácie vyvíjané spoločnosťou SAP.

- **Testovacie prostredie.** Určuje prostredie, v ktorom bude prebiehať testovanie. Na zreteľ sa berie fakt, že fuzzer by mal bežať na rôznych platformách. Fuzzer nemôže byť pri testovaní závislý na jednom druhu aplikácií. Mal by byť spustiteľný proti všetkým verejným a zároveň interným OData službám. Avšak, pri testovaní interných SAP aplikácií je nutné byť pripojený priamo v privátnej sieti, čo znamená, že s určitým obmedzením treba počítať.

- **Programovací jazyk.** Jazyk by mal byť vybraný na základe najlepšej podpory protokolu OData a zároveň najlepšej možnosti integrácie fuzzera medzi existujúce automatizované nástroje v systémoch SAP. Pri výbere programovacieho jazyka je podstatná aj samotná syntax a abstrakcia. Pod abstrakciou sa myslí hlavne dostupnosť základných abstraktných typov, ako napr. zoznam, hash tabuľka a pod. Programovací jazyk by mal byť určite objektovo orientovaný pre lepšie rozdelenie jednotlivých komponentov do tried či podtried.

Na základe dôkladnej analýzy boli prijaté následovné závery. Návrh fuzzera v sebe nezahrňuje použitie žiadneho frameworku. Jedným z hlavných príčin bolo to, že frameworky opísané v kapitole 2 neposkytujú dostatočnú variabilitu pri generovaní premenlivej štruktúry, ktorá sa častokrát v adrese URI vyskytuje.

Na generovanie nových požiadaviek sa použije genetický algoritmus. Veľkou výhodou genetických algoritmov je ich flexibilita a robustnosť. Genetické algoritmy si vedia poradiť s lokálnymi maximami a nemusia mať detailnú znalosť o probléme, ktorý riešia [2]. Genetické algoritmy sú nasadené vo viacerých fuzzeroch z dôvodu lepšieho generovania chybových stavov a lepšieho pokrytia testovaného kódu. Aplikáciu algoritmu v ODfuzz si popíšeme v sekcii 4.5.

Programovací jazyk Python sa použije na implementáciu fuzzera, nakoľko je všestranný a disponuje nielen dobrou podporou pre protokol OData, ale aj knižnicami pre prácu s HTTP požiadavkami. Implementačné detaily sú objasnené v kapitole 5.

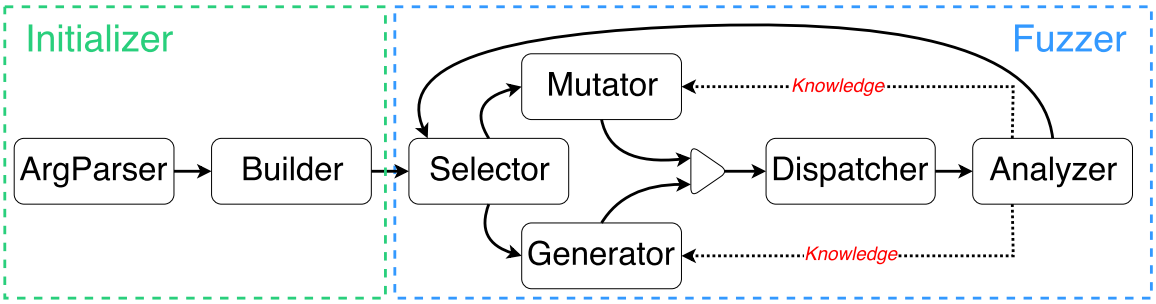
Navrhnutý nástroj je zameraný na vytváranie testovacích požiadaviek, ktoré v sebe nesú opytovacie funkcie `$filter`, `$skip`, `$top` a `$orderby`. ODfuzz plne podporuje konvencie protokolu OData 2.0.

[2]Fiori applikácie https://fioriappslibrary.hana.ondemand.com/

Výstupom fuzzera sú štatistické informácie o vykonaných testoch. Štatistiky predstavujú súhrn všetkých testovaných entít, ich vlastností, odozvy servera, počet úspešných testov, počet neúspešných testov atď. Výstup fuzzera je bližšie špecifikovaný v sekcii 5.5.

4.3 Popis komponentov

V tejto sekcii si popíšeme, z akých logických častí sa nástroj ODfuzz skladá. Rozdelenie základných komponentov je vizualizované na obrázku 4.1.



Obr. 4.1: Základné komponenty nástroja ODfuzz

ArgParser spracováva argumenty príkazovej riadky. ODfuzz prijíma na vstupe adresu OData služby, priečinok na logovanie stavu, priečinok na zapisovanie štatistík, textový súbor obsahujúci obmedzenia a prepínač na povolenie asynchrónneho zasielania požiadaviek. Všetky argumenty, okrem adresy OData služby, sú voliteľné. Súbor s obmedzeniami je vytváraný používateľom a obsahuje zoznam pravidiel, ktoré má fuzzer aplikovať pri generovaní testov. Pravidlá sa definujú nasledovne:

```
[ Exclude | Include ]
    [ $filter | $orderby | $skip | ... ]
        EntitySet name
            Property name
            Property name
            ...
        [ $F_ALL$ | $E_ALL$ | $P_ALL$ ]
            [ Function name | EntitySet name | Property name ]
            ...
```

Vo vyššie uvedenej syntaxi prvý riadok indikuje, či sa konkrétne obmedzenie nemôže alebo musí zahrnúť do generovaných testov. Na druhom riadku je deklarovaný názov funkcie, ktorej sa dané obmedzenie týka. `Property name` je názov vlastnosti entity, pričom názov entity reprezentuje položka `EntitySet name`. Špeciálne premenné sú značené symbolom dolár `$` na začiatku a na konci reťazca. `$F_ALL$` je globálne obmedzenie vzťahujúce sa na funkciu `$filter`. Pod ním sa píše zoznam filtrovacích funkcií, podobne, ako pri definovaní obmedzení pre vlastnosti jednej entity. `$E_ALL$` je globálne obmedzenie pre entitu ako celok. `$P_ALL$` je globálne obmedzenie pre vlastnosti entít. Toto pravidlo slúži pre jednoduchšie definovanie obmedzení, ak sa vo viacerých entitách nachádza vlastnosť s rovnakým názvom.

Builder využíva argumenty spracované modulom *ArgParser*. Interne však medzi nimi neprebieha žiadna komunikácia. *Builder* inicializuje všetky štruktúry, ktoré sa budú používať pri generovaní požiadaviek. Štruktúry tvoria obálku nad entitami, ktoré je možné

používať v opytovacích funkciách. Sú to entity, ktoré nemajú definované žiadne obmedzenia či už v dokumente metadát, alebo priamo používateľom.

Selector je časť fuzzera, ktorá rozhoduje o tom, aká entita bude testovaná a o tom, akým spôsobom budú dáta generované. Jeho chovanie je závislé na výstupe analyzátora *Analyzer*. *Selector* vyberá celkovo z dvoch metód vytvárania dát. *Mutator* mutuje existujúce dáta alebo vytvára nové spájaním tých existujúcich. Pri mutácii sa menia hodnoty prevracaním bitov, pridávaním, odoberaním a prehadzovaním znakov. Spájanie je v genetickom algoritme považované za kríženie. Obe operácie sú objasnené v sekcii 4.5. *Generator* generuje nové dáta náhodne alebo pomocou gramatiky tak, aby vyhovovali obmedzeniam daného dátového typu, resp. štruktúre požiadavku, viď sekciu 4.4.

Modul *Dispatcher* môžeme klasifikovať ako sprostredkovateľa komunikácie medzi fuzzerom a serverom. *Dispatcher* odosiela HTTP požiadavky a číta prijaté odpovede. Tento modul nijako neovplyvňuje chod fuzzera, tvorí len jednoduché rozhranie medzi vonkajším prostredím.

Analyzer analyzuje generované dáta a prijaté odpovede zo servera. Pod analýzou si predstavme čítanie stavového kódu HTTP, čas odpovedi, dĺžky generovaného reťazca a následné určenie úspešnosti generovaných dát. Miera úspešnosti sa zúročí pri generovaní nových požiadaviek prostredníctvom genetického algoritmu.

4.4 Gramatika generovaných reťazcov

Na generovanie reťazcov pre funkciu `$filter` bola použitá bezkontextová gramatika. Pravidlá vytvorenej gramatiky sú definované ako:

```
1. EXPRESSION -> PROPFUNC OPERATOR OPERAND | CHILD
2. CHILD -> PARENT LOGICAL PARENT
3. PARENT -> EXPRESSION | CHILD | ( CHILD )
4. LOGICAL -> or | and
5. PROPFUNC -> property1 | property2 | property3 | ...
6. PROPFUNC -> startswith(p0, p1) | endswith(p0, p1) | ...
7. OPERATOR -> eq | ne | lt | gt | ...
8. OPERAND -> str | num | bool
```

Vo vyššie uvedených pravidlách sú neterminálne symboly označené veľkými písmenami, terminálne naopak malými a počiatočným symbolom je `EXPRESSION`. Terminály označené ako `propertyN` predstavujú názov skutočnej vlastnosti entity. Terminály `str`, `num`, `bool` sú po poradí reťazec (napr. “Auto”), číslo (napr. 123) a logické *áno* alebo *nie*.

Je vhodné podotknúť, že sa v pravidlách gramatiky nachádza nepriama ľavá rekurzia. Po aplikovaní pravidla `PARENT -> CHILD` získame neterminál `CHILD`. Aplikovaním pravidla č. 3 sa znova vytvorí neterminál `PARENT`. Generátor, ktorý aplikuje takúto gramatiku sa môže rekurzívne zanoriť niekoľkokrát.

Gramatika nám umožňuje generovať filtrovacie reťazce obsahujúce viacero logických častí, ako napríklad:

```
Price eq 12 and (Available eq true or Available ne false) and Price gt 10
(Detail eq 'Car' or Detail lt 'Boat') and Detail gt 'Bus'
(endswith(Detail, 'ar') eq true or Detail lt 'Plane') or Price eq 33
startswith(Detail, 'Ca') eq false or (Price lt 10024 or Price gt 11000)
```

Všetky vyššie uvedené reťazce sú legálne a plne rešpektujú normy funkcie `$filter`.

4.5 Aplikácia genetického algoritmu

V tejto sekcii si popíšeme, akým spôsobom je v ODfuzz aplikovaný genetický algoritmus a na čo konkrétne sa jeho výstup používa.

ODfuzz používa genetický algoritmus na vytváranie nových požiadaviek. Na začiatku sa vygeneruje počiatočná populácia, ktorej veľkosť je závislá na počte vlastností entít. Rozmanitosť populácie zabezpečíme generovaním minimálne 30 požiadaviek pre každú vlastnosť. Táto hodnota bola určená na základe predvádzaných experimentov počas návrhu prototypu. Minimálnu veľkosť populácie vypočítame rovnicou $Pop_{min} = Prop_{all} * 30$, kde $Prop_{all}$ je počet všetkých vlastností v danej OData službe. Jedna vygenerovaná požiadavka predstavuje v populácii jedného jedinca.

Kvalita populácie sa hodnotí fitness funkciou. Kvalita popisuje úspešnosť požiadavku z pohľadu fuzz testovania. Ak požiadavka spôsobila chybu, je v porovnaní lepšia ako tá, ktorá naopak nie. Kvôli tomu sa požiadavky odošlú na server a podľa odpovedi zo servera sa určí ich celková kvalita. V sekcii 4.5.1 sú ozrejmené činitele, ktoré ovplyvňujú kvalitu jedinca.

S ohľadom na biológiu je každý jedinec v populácii prezentovaný ako chromozómom zložený z niekoľkých génov. Ako príklad si vezmime bitový reťazec. Celok predstavuje jeden chromozóm. Jednotlivé bity sú gény [19]. Podobne to je v prípade genetickej reprezentácii požiadaviek, ktoré generuje ODfuzz. Chromozóm je jedna celá požiadavka, čiastkové chromozómy sú opytovacie funkcie a gény predstavujú jednotlivé časti parametrov daných opytovacích funkcií. V ODfuzz sa jednotlivci po inicializácii populácie medzi sebou krížia alebo mutujú, čím vznikajú nový jedinci. Obe operácie sú priblížené v sekciách 4.5.2 a 4.5.3.

Potomkovia sa odošlú na server a ohodnotia fitness funkciou. Do novej populácie sa zaradí vždy ten lepší potomok. Potomok, ktorý má horšiu hodnotu fitness, sa odstráni.

Aby sa populácia nezväčšovala každým generovaným testom, najslabší jedinci sa odstraňujú. ODfuzz vykonáva túto operáciu pravidelne po vytvorení ľubovolného potomka.

4.5.1 Fitness funkcia

Úspešnosť jedinca reprezentuje fitness skóre, čo je numerická hodnota označujúca bodové ohodnotenie jedinca. Celková hodnota skóre sa počíta fitness funkciou ako súčet troch čiastkových skóre:

1. **Skóre stavového kódu HTTP.** Ak je stavový kód rovný 500, čo odpovedá internej chybe servera, hodnota skóre je 100. Ak je stavový kód rovný 200, čo značí úspešnú odpoveď, hodnota skóre je 0. V každom inom prípade je skóre -50. Ostatné stavové kódy ohodnocujeme záporne, pretože produkujú falošné pozitíva alebo sú neopodstatnené. Príkladom falošného pozitíva môže byť entita, ktorá nemá implementované metódy pre čítanie jej obsahu. Po akomkoľvek dopytovaní sa na túto entitu, nám server odošle chybovú hlášku so stavovým kódom 501. Algoritmus bude počítať s touto situáciou, ako so skutočnou chybou, preto bude považovať stavové kódy 500 a 501 za rovnocenné, pričom z definície nie sú. Riešením by mohlo byť priraďovanie iného skóre pre stavové kódy typu 5xx[3], ale žiadny zo stavových kódov nie je tak významný ako 500, preto toto riešenie nebudeme akceptovať.

2. **Skóre dĺžky požiadavky.** Toto skóre sa určí delením maximálnej dĺžky, ktorá sa hodnotí, a reálnej dĺžky požiadavku. Maximálna dĺžka hodnoteného reťazca bola ur-

[3]Stavové kódy indikujúce chyby servera https://tools.ietf.org/html/rfc7231#section-6.6

čená empiricky na základe konzultácie so zamestnancami spoločnosti SAP. Skóre sa vypočíta rovnicou $S_{qlen} = 300/Query_{len}$, kde $Query_{len}$ je dĺžka požiadavky a 300 je maximálna dĺžka. Dĺžka požiadavky je zahrnutá do počítania fitness funkcie z dôvodu lepšej analýzy požiadaviek, ktoré boli spúšťačom chybového chovania. Požiadavka, ktorá je viditeľne kratšia, sa lepšie analyzuje.

3. **Skóre času odozvy.** Čas odozvy sa meria v sekundách. Na vypočítanie skóre pre čas odozvy sa používa vzťah $S_{qt} = Query_{rt}/10$, ak je hodnota času odozvy $Query_{rt}$ menšia ako 100. Ináč, a teda v minime prípadov, sa ešte k výslednému skóre pripočíta výsledok výrazu $Query_{rt}^2/10^{(len(Query_{rt})+1)}$. Použité vzťahy boli sformulované tak, aby malo čiastkové skóre čo najmenší dopad pri započítavaní do celkového skóre. Čas odozvy môže napomôcť pri určovaní časovej zložitosti algoritmu na strane servera. Požiadavka, ktorá sa vyhodnocuje dlhšie než obvykle, môže prechádzať komplexnejšími časťami kódu. V neposlednom rade však môže byť odpoveď zo servera spomalená kvôli veľkému množstvu dát, ktoré sa klientovi majú odoslať. V takomto prípade sa čas zo servera nevyhodnocuje.

4.5.2 Operácia kríženia

Kríženie je jedna z niekoľkých techník reprodukcie, kde sa zoberú dva rodičovské chromozómy a rozdelia sa na jednom alebo viacerých miestach. Každý potomok nato dostane ľavú časť chromozómu, resp. chromozómov, jedného rodiča a pravú časť chromozómu, resp. chromozómov, druhého rodiča [19].

V ODfuzz sa chromozómy rozdeľujú na jednom mieste. Miesto rozdelenia je určené pozíciou operátorov `and`, `or` alebo `&`. Operátor je vybraný náhodne a konštrukcie, ktoré sa nachádzajú po jeho pravej strane, sa zamenia konštrukciami z druhej požiadavky. Kríženie logických častí funkcie `$filter` je následovné:

```
Rodič 1: Price lt 20 and Price gt 10 and startswith(Type, 'Beve') eq true
Rodič 2: Price gt 20 or City eq 'Brno' or City eq 'Prague'

Dieťa 1: Price lt 20 and Price gt 10 and City eq 'Prague'
Dieťa 2: Price gt 20 or City eq 'Brno' or startswith(Type, 'Beve') eq true
```

Miesto rozdelenia rodičovských chromozómov bol tretí logický operátor v reťazci. Podotkneme, že chromozómy sú rovnakej veľkosti. Ak by boli rôznej veľkosti, kríženie by prebehlo podobne, len by sme museli zmeniť miesto rozdelenia podľa kratšieho chromozómu:

```
Rodič 1: Price lt 20 and Price gt 10
Rodič 2: Price gt 20 or City eq 'Brno' or City eq 'Prague' and Price eq 20

Dieťa 1: Price lt 20 and City eq 'Brno' or City eq 'Prague' and Price eq 20
Dieťa 2: Price gt 20 or Price gt 10
```

Vyššie uvedeným spôsobom sa krížia filtrovacie reťazce. Kríženie jednotlivých funkcií prebieha takto:

```
Rodič 1: $filter=Price lt 20&$top=10&$skip=5
Rodič 2: $filter=Price gt 20&$orderby=Price asc&$top=20

Dieťa 1: $filter=Price lt 20&$top=20
Dieťa 2: $filter=Price gt 20&$orderby=Price asc&$skip=5
```

Miesto delenia je posledný oddeľovač `&`. V tomto prípade fuzzer nahradí u potomka č. 1 funkciu `$skip=5` za `$top=20`, čo v konečnom dôsledku odstráni pôvodnú funkciu `$top=10`. Príčinou je spôsob organizovania dát, ktoré sa používajú pri generovaní nových jedincov, viď sekciu 4.6.

Rodičia sa vyberajú v turnaji, angl. „tournament selection“. Turnajový výber je založený na náhodnom výbere skupiny niekoľkých jedincov z celkovej populácie a zvolení najlepšieho z nich [13]. V našom prípade je s pravdepodobnosťou 0.8 víťazom turnaja taký jedinec, ktorý má najlepšie fitness skóre. ODfuzz vyberá jedného rodiča z maximálne 10 jedincov. Tento výber vykonáva dvakrát, pretože rodičia tvoria pár.

Turnajový výber bol v genetickom algoritme použitý z dôvodu jednoduchej implementácie a možnosti menenia parametrov veľkosti vzorky jedincov či pravdepodobnosti výberu najlepšieho jedinca. Experimentovaním sa taktiež zistilo, že 10 jedincov predstavuje rozumnú veľkosť vzorky. Väčší počet jedincov mal za následok časté vyberanie elít z celkovej populácie, kvôli čomu sa stávala stále menej rôznorodá.

4.5.3 Operácia mutácie

ODfuzz mutuje reťazce a číselné hodnoty. Reťazce, ktoré je možné mutovať, sa nachádzajú vo filtrovacom reťazci funkcie `$filter`. Číselné hodnoty nájdeme v parametroch funkcií `$skip`, `$top` či `$filter`. ODfuzz aplikuje nasledujúce formy mutácie:

1. **Mutácie reťazca**
   - Preklopenie bitu. Jeden náhodný bit v znaku sa preklopí z logickej 0 na logickú 1 alebo naopak (napr. reťazec “brno” sa zmení po preklopení 2. bitu v znaku “b” na “’rno”).
   - Zmena znaku. Existujúci znak sa nahradí novým náhodne vygenerovaným UTF-8 znakom (“brno” -> “bŤno”).
   - Prehodenie znakov. Z reťazca sa vyberú dva náhodné znaky a prehodia sa medzi sebou (“brno” -> “bonr”).
   - Otočenie skupiny znakov. Na náhodne zvolenú časť reťazca sa aplikuje inverzia (“brno” -> “nrbo”).
   - Pridanie znaku. Do reťazca sa pridá jeden znak (“brno” -> “brnoA”).
   - Odstránenie znaku. Z reťazca sa odstráni jeden znak (“brno” -> “bno”).
2. **Číselné mutácie**
   - Ikrementácia hodnoty. Číselná hodnota sa zväčší o 1 (100 -> 101).
   - Dekrementácia hodnoty. Číselná hodnota sa zmenší o 1 (100 -> 99).
   - Pridanie číslice. Do existujúceho čísla sa pridá ďalšia číslica (100 -> 1008).
   - Odstránenie číslice. Z existujúceho čísla sa odstráni náhodne číslica (100 -> 10).

Nami použité mutácie boli vybrané na základe možností protokolu tak, aby nedochádzalo k porušeniu štruktúry či formátu požiadavky. Pri výbere operácií sme sa inšpirovali existujúcimi fuzzermi, ktoré mutujú obyčajné číselné a textové reťazce.

Mutácie majú pri testovaní HTTP `GET` požiadaviek v rámci protokolu OData veľmi obmedzený charakter. Mutovať sa nemôžu názvy funkcií, entít, vlastností a operátorov. Tieto názvy sú záväzné a po ich zmene nie je OData služba schopná rozpoznať o aký zdroj dát klient žiada. Server nám vráti odpoveď s chybovou správou informujúcou o neexistujúcom segmente.

4.6 Reprezentácia dát

ODfuzz si vzhľadom na spôsob generovania musí uschovávať určité informácie o požiadavkách. Sú to informácie o konkrétnych častiach opytovacích funkcií či reťazcov. Jednotlivé požiadavky sa ukladajú do databázy kvôli analýze vykonávanej modulom *Analyzer* a následnom výbere jedincov pri operácii mutácie alebo kríženia.

Každá požiadavka má v databáze zvlášť záznam. Jeden záznam obsahuje vlastný identifikátor, názov entity, HTTP stavový kód, identifikátory rodičovských požiadaviek, fitness skóre, časti funkcií `$filter`, `$top`, `$skip`, `$orderby` a reťazec vygenerovanej požiadavky.

Opytovacie funkcie sú v databáze prezentované len raz. Protokol OData nedefinuje chovanie servera v prípade, že požiadavka obsahuje niekoľko rovnakých opytovacích funkcií. Server môže odpovedať chybou, no nemusí. ODfuzz pri krížení jedincov tak jednoducho prepíše hodnoty pôvodných funkcii, aby sa predišlo zbytočným komplikáciám a vyššej réžii pri udržiavaní zhodných opytovacích funkcií v rámci jednej požiadavky.

U funkciách `$top` a `$skip` nám plne vystačuje si zapamätať len jednu hodnotu. Napríklad pre požiadavku obsahujúcu funkciu `$skip=10` to bude číslo 10. Pri funkcii `$orderby` si zaznamenávame použité vlastnosti a spôsob radenia (vzostupne alebo zostupne). Funkcia `$filter` je o čosi komplexnejšia. Predstavme si filtrovací reťazec:

```
$filter=(Price lt 20 or Price gt 100) and startswith(Detail, 'Bev')
```

Aby sme si generovaný reťazec rozložili do niekoľkých častí, museli by sme implementovať zložitý analyzátor, ktorý by počítal so všetkými možnosťami zoskupovania operátorov a logických častí. Takéto riešenie by bolo netriviálne. ODfuzz si zaznamenáva osobitné logické časti už počas generovania filtrovacej funkcii bezkontextovou gramatikou. Po aplikovaní gramatických pravidiel vieme určiť vždy, aká logická časť sa kde nachádza a aké časti ju obklopujú.

Pri aplikácii operácie mutácie alebo kríženia sa v zázname zmenia, resp. prehodia len niektoré fragmenty. Na generovanie reťazca pozmenenej štruktúry funkcie `$filter` používa ODfuzz separátnu triedu. Viac sa tejto problematike budeme venovať v kapitole 5.

Kapitola 5

Implementácia fuzzera

Implementačným jazykom navrhovaného nástroja je Python. Na implementáciu bola konkrétne použitá verzia 3.6. Python 3.6 sa od druhej najpopulárnejšej verzie 2.7 líši hlavne odstránenými chybami a rozdielnym spôsobom vyhodnocovania delenia dvoch integerových hodnôt, používaním funkcie `print()` a odchytávaním či generovaním výnimiek. Ďalším významným rozdielom je odlišný návratový typ funkcie `range()`. Tá po novom vravia vracia iterátor[1] namiesto zoznamu. Viac o uvedených a aj iných zmenách si môže čitateľ preštudovať v oficiálnej dokumentácii jazyka[2]. Staršia verzia 2.7 naviac prestane byť oficiálne podporovaná už v roku 2020[3]. Aj preto sa nepoužila vo finálnej implementácii.

Navzdory dôkladnému návrhu fuzzera sa implementačná časť diferencuje od teoretickej drobnými zmenami. Komponenty bolo nutné rozdeliť na viacero podčastí a obaliť pomocnými triedami.

V tejto kapitole sú popísané jednotlivé časti fuzzera z programátorského hľadiska. To znamená, aké dátové štruktúry alebo knižnice boli použité a akým spôsobom boli algoritmy spomínané v predchádzajúcich sekciách implementované. Na záver kapitoly si ešte ukážeme, ako pracovať s výstupom fuzzera.

5.1 Štruktúra programu

Programové riešenie si vyžadovalo rozdeliť implementačnú časť do niekoľkých logických celkov. Funkcionalitu nástroja ODfuzz sme rozčlenili do nasledujúcich ôsmich modulov:

- `arguments.py` – obsahuje triedu na spracovávanie argumentov príkazového riadku,
- `entities.py` – pozostáva z tried slúžiacich na inicializáciu a generovanie jednotlivých opytovacích funkcií pre každú entitu,
- `fuzzer.py` – implementuje logiku rozhodovania a generovania požiadaviek, ktoré sa budú posielať na server,
- `mongos.py` – obsahuje pomocnú triedu na prístup do databázy, v ktorej sa nachádzajú údaje o všetkých požiadavkách,
- `monkey.py` – obsahuje funkcie na dynamickú zmenu atribútov za behu programu,

[1] Iterátor je objekt, ktorý programátorovi umožňuje prechádzať kolekciou
[2] Čo je nové v Python 3 https://docs.python.org/3/whatsnew/
[3] Podpora pre Python 2.7 https://legacy.python.org/dev/peps/pep-0373/

- `generators.py` – pozostáva zo základných funkcií na generovanie a mutovanie dát,
- `restrictions.py` – obsahuje triedy na transformovanie obmedzení do objektovej podoby,
- `statistics.py` – obsahuje triedy pre uchovávanie a zapisovanie štatistických informácií.

5.2 Spracovávanie vstupu

Na spracovanie vstupu sa použil modul `argparse` zo štandardnej knižnice jazyka Python. Analyzátor `ArgumentParser`, nachádzajúci sa v tomto module, konvertuje hodnoty argumentov príkazového riadku do objektovej reprezentácie jazyka Python. *ArgParser* implementovaný v nástroji ODfuzz je obálkou nad týmto analyzátorom. Nastavuje názvy a typy argumentov a odchytáva výnimky podnietené analyzátorom pri spracovávaní.

5.3 Inicializácia komponentov

Inicializácia komponentov je v ODfuzz implementovaná v module `entities.py`. Na počiatku sa na OData službu odošle požiadavka o sprístupnení dokumentu metadát. Už v tejto fáze sa požiadavka odosiela cez *Dispatcher*.

Kľúčová funkcionalita je pre *Dispatcher*, podobne ako v prípade modulu *ArgParser*, zabezpečovaná existujúcim riešením. *Dispatcher* využíva knižnicu `requests`[4]. Táto knižnica poskytuje rozhranie pre zasielanie HTTP požiadaviek.

Zdrojový kód 5.1 je výťažkom konštruktoru[5] pre triedu `Dispatcher`. Na mieste komentára sa nachádzajú ďalšie inicializácie, ktoré sú momentálne nepodstatné. Na riadku č. 5 až 8 sa nachádza inicializácia objektu HTTP relácie. Knižnica `requests` sa postará o to, aby objekt uchovával určité informácie o spojení medzi serverom a klientom. Pri posielaní požiadaviek sa tak nemusí opakovane vykonávať proces nazývaný angl. handshake[6], čím sa zníži celková réžia a záťaž na server. Všimnime si ešte, že v objekte HTTP relácie nastavujeme inštančnej premennej `auth` autentifikačné údaje používateľa a do premennej `verify` ukladáme SSL[7] certifikát pre overenie HTTP požiadavky. Autentifikácia a použitie SSL certifikátu je nevyhnutné na prístup k aplikáciám, ktoré sa nachádzajú v internej sieti spoločnosti SAP.

```
1 def __init__(self, service, sap_certificate=None):
2
3     # Initialization of other components
4
5     self._session = requests.Session()
6     self._session.auth = (os.getenv(ENV_USERNAME),
7                           os.getenv(ENV_PASSWORD))
8     self._session.verify = self._get_sap_certificate()
```

Zdrojový kód 5.1: Konštruktor triedy `Dispatcher`

[4]Knižnica requests http://docs.python-requests.org/en/master/
[5]Konštruktor je špeciálna metóda, ktorá inicializuje objekt danej triedy
[6]Hanshake je proces zahájenia spojenia medzi serverom a klientom na začiatku každej komunikácie
[7]SSL (Secure Sockets Layer) vytvára zabezpečený komunikačný kanál medzi dvoma zariadeniami

Požiadavky sa na server posielajú volaním `self._session.request('GET', URI)`, kde prvým parametrom je HTTP metóda a druhým adresa URI, na ktorú sa pýtame.

Dokument metadát je v binárnej podobe spracovaný funkciou `schema_from_xml()`, ktorá je súčasťou knižnice PyOData. Všetky entity, ich vlastnosti, atribúty a asociácie sa transformujú do objektov jazyka postupným spracovávaním objektu typu `ElementTree`. Tento objekt vzniká volaním metódy `parse()` z modulu `lxml.etree`[8] na začiatku samotnej konverzie. Objekt typu `ElementTree` je obálkou nad XML dokumentom a disponuje niekoľkými metódami pre serializáciu alebo dopytovanie sa na jednotlivé elementy.

Spracovaný dokument metadát sa použije na vytvorenie štruktúr, ktoré fuzzer používa pri generovaní požiadaviek. Štruktúry sú vlastne objekty, ktoré obsahujú skupiny entít rozdelených ďalej do podskupín podľa toho, aké opytovacie funkcie podporujú. Hlavný typ skupiny je dedený implementáciami tried pre funkcie `$filter`, `$skip`, `$top` a `$orderby` vzhľadom na princíp dedičnosti v objektovo orientovanej paradigme. Implementácie obsahujú metódy pre generovanie tej danej opytovacej funkcie a aplikujú obmedzenia definované používateľom či dokumentom metadát. Entity a vlastnosti entít, ktoré nie je možné použiť v opytovacích otázkach, sa z objektového modelu odstránia kľúčovým slovom `del`.

Obmedzenia deklarované používateľom sa z textového formátu preložia do slovníka. Slovník je abstraktný dátový typ implementujúci hash tabuľku. V iných programovacích jazykoch je reprezentovaný ako asociatívne pole. K obmedzeniam napríklad pre funkciu `$filter` sa pristupuje cez `restriction['filter']`.

Jazyk Python umožňuje dynamicky meniť, mazať a pridávať metódy do existujúcich inštancií tried počas behu programu. Pri inicializácii štruktúr sa do každej vlastnosti, ktorá sa použije vo filtrovacích reťazcoch, pridá metóda `generate()`. Metóda `generate()` slúži na generovanie hodnoty pre jeden konkrétny dátový typ. Pri generovaní hodnoty ľubovolnej vlastnosti sa volá len táto metóda bez akýchkoľvek ďalších parametrov či kontrol.

Zdrojový kód 5.2 znázorňuje spôsob realizácie pridania novej metódy v ODfuzz. Čitateľovi to môžeme prirovnať k predávaniu ukazovateľa na funkciu, pričom hodnotu parametru `self`, ktorá sa používa v generátore a ktorú je možné sprístupniť len lokálne v inštancii triedy, získame volaním deskriptora[9] `__get__`.

```
1  def edm_string(self):
2      string_length = round(random.random() * self.max_string_length)
3      string = ''.join(random.choice(BASE_CHARSET)
4                       for _ in range(string_length))
5      return '\'{}\''.format(string)
6
7  def patch_proprty_generator(proprty):
8      proprty_type = proprty.typ.name
9      if proprty_type == 'Edm.String':
10         proprty.generate = edm_string.__get__(proprty, None)
```

Zdrojový kód 5.2: Pridanie novej metódy do existujúceho objektu

Spôsob zmeny atribútov za behu programu sa vo vzťahu s jazykom Python nazýva angl. „monkey patching“. V ODfuzz je na tieto zmeny určený modul `monkey.py`, ktorý v sebe obsahuje aj metódu `patch_proprty_generator()`, ktorá bola demonštrovaná v príklade vyššie.

[8] Modul lxml.etree http://lxml.de/tutorial.html

[9] Deskriptor v jazyku Python https://www.ibm.com/developerworks/library/os-pythondescriptors/index.html

Pre entity, ktoré podporujú funkciu `$filter`, sa ešte vytvorí sada objektov, ktoré obsahujú funkcie ako `startswith()`, `endswith()`, `substringof()` a iné. Funkcie v sebe nesú zoznam vlastností entít, ktoré sa môžu použiť v tej danej funkcii. Napríklad pre funkcie, ktoré pracujú s reťazcami je zoznam tvorený len vlastnosťami, ktorých dátovým typom je `Edm.String`. Tento zoznam sa využije pri generovaní parametrov funkcie.

5.4 Generovanie požiadaviek

Na celkovú reprodukciu procesu sa nastaví hodnota semienka náhodného generátoru podľa času a zapíše sa do logovacieho súboru. Pri reprodukovaní testov sa použije uložená hodnota semienka, čím sa budú generovať rovnaké testovacie dáta. Reprodukciou testovacích prípadov vieme zistiť, či bola chyba na serveri spustená sadou niekoľkých testov alebo len práve jedným.

Jadro náhodného generátora, ktoré využíva modul `random` použitý v ODfuzz, je implementované v jazyku Python 3 cez Mersenne Twister [10].

ODfuzz zahajuje fuzzing generovaním počiatočnej populácie s náhodnými dátami. Náhodné dáta predstavujú hodnoty, ktoré sú určené na porovnávanie vo funkcii `$filter` či čísla vo funkciách `$skip` alebo `$top`.

Populácia sa vytvára prechádzaním objektov entít a generovaním opytovacích funkcií. To, aké opytovacie funkcie sa budú generovať je určené návratovou hodnotou funkcie, ktorá je zobrazená v zdrojovom kóde 5.3.

```
1 def random_options(self):
2     sample_length = round(random.random() * len(self._options_list))
3     sample_options = random.sample(self._options_list, sample_length)
4     return sample_options + self._required_filter_option
```

Zdrojový kód 5.3: Náhodný výber opytovacích funkcií

Vyššie uvedená metóda vyberie zo zoznamu opytovacích funkcií `_options_list` náhodnú vzorku, ktorej veľkosť je určená premennou `sample_length`. K zoznamu sa naviac pripojí objekt opytovacej funkcie `$filter`, v prípade, že je v dokumente metadát vyžadované jej použitie. Nutnosť použitia sa definuje atribútom `sap:requires-filter` v elemente `EntitySet`. Ak je toto obmedzenie nastavené, do zoznamu `_options_list` sa pri inicializácií entít nevčlení objekt funkcie `$filter`, pričom do zoznamu `self._required_filter_option` áno. V opačnom prípade ostáva zoznam `_required_filter_option` prázdny, čo v konečnom dôsledku pri návrate z funkcie nepridá do zoznamu `sample_options` nič.

Rýchlosť generovania nových požiadaviek je závislá na rýchlosti odpovedí zo servera. Pri posielaní požiadaviek sa bolo nutné zamyslieť nad možnosťou prijímania veľkého množstva dát, pomalej alebo žiadnej odpovedi zo servera. Prvá možnosť sa rieši aplikovaním obmedzenia na funkciu `$top`. Toto obmedzenie musí zadať používateľ cez kľúčové slovo `Include` v súbore, ktorý definuje obmedzenia. Pomalá, resp. žiadna odozva sa rieši použitím parametra `timeout` pri zasielaní požiadavky pomocou knižnice `requests`. `timeout` nastavuje hodnotu časovača, ktorý počíta čas odozvy servera. Po jeho vypršaní sa vyhodí výnimka. ODfuzz rieši takúto situáciu spätným zaslaním požiadavky zloženej len z adresy URI OData služby. Dôvodom tohto chovania je fakt, že niektoré služby sa môžu reštartovať alebo úplne vypnúť. Jednako si vieme overiť, či OData služba naozaj ďalej beží alebo požiadavka, ktorá bola odoslaná spôsobila nejakú chybu na serveri.

[10] Python 3 generátor https://docs.python.org/3/library/random.html

ODfuzz podporuje asynchrónne zasielanie požiadaviek. Pre požiadavky sa vytvorí tzv. bazén, v ktorom sa súbežne odosielajú a spracúvajú. Požiadavka predstavuje v bazéne jeden objekt typu `greenlet`[11]. Objekty tohto typu sú v bazéne vytvárané cez modul `gevent`[12] v inštancii triedy `Pool`. Modul `gevent` je v ODfuzz použitý tiež na dynamické modifikovanie komponentov na úrovni implementácií soketov, ktoré používa aj modul `requests`, funkciou `patch_all()`. Po odoslaní požiadavky je po modifikácii možné ukončiť fuzzer počas toho, čo čaká na odozvu zo servera.

Vygenerovaná požiadavka je tvorená jednou alebo viacerými opytovacími funkciami. Opytovacie funkcie sa generujú samostatne metódou `generate()`, ktorá je prítomná vo všetkých objektoch. V sekciách 5.4.1 až 5.4.3 si popíšeme spôsob generovania dát u jednotlivých funkcií.

Na ukladanie dát sa používa NoSQL databáza monogDB[13]. NoSQL systémy sú nerelačné databázy a nepoužívajú primárne jazyk SQL na prístup k dátam [10]. ODfuzz pracuje s knižnicou `pymongo`[14] pri čítaní a upravovaní záznamov v databáze mongoDB. Viac informácií o využití databázy nájdeme v sekcii 5.4.4.

5.4.1 Funkcia $filter

Funkcia `$filter` je generovaná gramatikou, ktorá bola predstavená v sekcii 4.4. Pravidlá gramatiky sú implementované rekurzívne zostupujúcim generátorom. Znamená to teda, že pri aplikovaní pravidiel sa postupne rekurzívne zanoríme niekoľkokrát, až kým nezískame terminálny symbol. V počítači máme obmedzenú veľkosť zásobníka a aby sa predišlo výnimke `RecursionError`[15], generovanej interpreterom jazyka Python, ODfuzz obsahuje ďalší obslužný kód, ktorý kontroluje počet zanorení.

Dvojznačné pravidlá gramatiky sú riešené vetvením kódu. To, či sa bude generovať vlastnosť alebo funkcia v danej logickej časti, je závislé na náhodnom generátore. Logika rozhodovania je nasledovná:

```
1 def _generate_element(self):
2     if random.random() < FUNCTION_WEIGHT:
3         self._generate_function()
4     else:
5         self._generate_proprty()
```

`FUNCTION_WEIGHT` je váha, ktorá udáva, ako často sa má funkcia generovať. V našom prostredí je hodnota konštanty nastavená na 0.3.

Počas rekurzívneho zanorovania sa do zvláštneho objektu `FilterOption` ukladajú informácie o všetkých častiach generovaného reťazca. Informácie napr. o reťazci `(Customer eq 'A' or Company eq 'B') and endswith(Country, 'C') eq true` sú trojakého druhu:

1. Hodnoty logických častí `Customer eq 'A'`, `Company eq 'B'` a `endswith(Country, 'C') eq true`. Logická časť, ktorá obsahuje funkciu, nesie informácie o názve funkcii (endswith), parametroch funkcie ('C'), použitých vlastnostiach (Country), názov porovnávacieho operátora (eq), hodnotu operandu (true) a návratový typ (Edm.Boolean).

[11] Objekt greenlet https://greenlet.readthedocs.io/en/latest/
[12] Modul gevent http://www.gevent.org/intro.html
[13] Databáza mongoDB https://docs.mongodb.com/
[14] Knižnica pymongo https://api.mongodb.com/python/current/
[15] Výnimka informujúca o prekročení maximálneho limitu zanorenia https://docs.python.org/3/library/exceptions.html#RecursionError

Pri logických častiach neobsahujúcich funkciu, si zaznamenávame len názov použitej vlastnosti (Customer), porovnávacieho operátora (eq) a hodnotu operandu ('A'). Hodnoty sa spolu s jedinečným identifikátorom, generovaným metódou `uuid.UUID()`[16], uložia do slovníka. Následne sa slovníky vložia do zoznamu logických častí. Vznikne nám teda štruktúra `zoznam(slovník, slovník, ...)`.

2. Hodnoty logických operátorov `or` a `and`. Za hodnotu sa považuje názov operátora a identifikátor ľavej a pravej časti, ktorú operátor oddeľuje. Dáta sa znovu ukladajú do slovníka. Pre reťazec `Customer eq 'A' or Company eq 'B'` bude inicializácia hodnôt slovníka vyzerať nasledovne:

```
1   logical['id'] = '12345678-1234-5678-1234-567812345678'
2   logical['left_id'] = customer_part_id
3   logical['right_id'] = company_part_id
4   logical['name'] = 'or'
```

   Rovnako ako v predchádzajúcom prípade aj tu sa slovníky pridávajú do zoznamu. Ak by bola pravou časťou skupina a nie logická časť, tak sa do `logical['right_id']` uloží identifikátor skupiny.

3. Hodnoty skupiny `(Customer eq 'A' or Company eq 'B')`. Skupina obsahuje zoznam identifikátorov logických operátorov a identifikačné číslo operátora, ktorý oddeľuje skupinu od ďalšej časti, teda od `endswith(Country, 'C')`. V príklade máme len jednu skupinu a jej hodnoty sú uložené do slovníka ako:

```
1   group['id'] = '22345678-1234-5678-1234-567812345678'
2   group['right_id'] = logical_and_id
3   group['logicals'] = ['12345678-1234-5678-1234-567812345678']
```

Logické časti a skupiny si ukladajú taktiež identifikačné číslo logického operátora, aby bola zabezpečená spätná interpretácia hodnôt.

Celkovo nám pri generovaní jedného filtrovacieho reťazca vzniknú tri zoznamy, ktorých elementy sú typu slovník. Informácie o jednotlivých častiach si ukladáme kvôli spôsobu, akým vytvárame nových potomkov, ktorí vzniknú krížením alebo mutáciou svojich rodičov. Na akúkoľvek zmenu nám takto stačí zmeniť hodnotu v poli a vygenerovať nový filtrovací reťazec. Na generovanie sa používa trieda `FiltrerOptionBuilder`, ktorá rekurzívne prechádza a postupne vytvára logické časti z údajov, ktoré sme si pred tým zaznamenali.

Tabuľka 5.1: Špeciálne znaky nachádzajúce sa v adrese URI

Špeciálny znak	Význam	Nahradená hodnota
+	Indikuje medzery	%2B
/	Oddeľuje priečinky a podpriečinky	%2F
?	Oddeľuje od seba parametre a adresu URI	%3F
%	Špecifikuje špeciálny znak	%25
#	Indikuje záložku	%23
&	Oddeľuje parametre v adrese URI	%26
'	Označuje reťazec	"

[16]Metóda UUID https://docs.python.org/3/library/uuid.html#uuid.UUID

Pri vytváraní reťazca zloženého z náhodných dát, môže dôjsť k tomu, že bude obsahovať špeciálne znaky, ktoré nemusia byť vždy interpretované v správnom kontexte na strane servera. Tabuľka 5.1 zobrazuje všetky znaky, ktoré je nutné nahradiť alebo prekódovať do šestnástkovej sústavy. Hodnoty tabuľky sú prevzaté z článku [7].

ODfuzz nahradzuje znaky volaním metódy `str.replace(old, new)`, pričom `old` je znak, resp. znaky, ktoré sa majú nahradiť znakom, resp. znakmi `new` v reťazci `str`.

5.4.2 Funkcie $skip a $top

V tomto prípade generuje ODfuzz pre obe funkcie len číselné hodnoty v rozmedzí 0 až 2147483647. Fuzzer môže generovaním takýchto hodnôt zistiť, či je OData služba schopná spracovávať extrémne veľké číselne hodnoty funkcií alebo či sú dané funkcie správne implementované v kombinácii s ďalšími.

Na generovanie sa používa náhodný generátor číselných hodnôt `random.randint(0, 2147483647)`. Funkcia generátora je zmenená v prípade, že používateľ definoval obmedzenia na hodnoty funkcií, a to obyčajným návratom definovanej hodnoty.

5.4.3 Funkcia $orderby

Funkcia `$orderby` je svojou zložitosťou generovania podobne jednoduchá, ako funkcie `$skip` a `$top`. Náhodne sa vyberie vlastnosť entity, ktorá spĺňuje obmedzenia dokumentu metadát a určí sa, akým spôsobom bude vo výsledku použitá. Pre jednu vlastnosť je možné generovať len dva druhy požiadaviek. Požiadavka obsahuje buď kľúčové slovo `desc`, čo značí klesajúcu postupnosť, alebo `asc`, čo predstavuje naopak stúpajúcu postupnosť:

```
https://odata.com/svc/Items?$orderby=Price desc
https://odata.com/svc/Items?$orderby=Price asc
```

Parametrom funkcie `$orderby` môže byť súčasne aj niekoľko vlastností zároveň:

```
https://odata.com/svc/Products?$orderby=Supplier, CompanyCode asc
```

ODfuzz volá metódu `random.sample(proprties, N)` na výber unikátnych vlastností, ktoré sa použijú vo výslednej funkcii. `N` je náhodne generovaná konštanta, ktorá určuje počet unikátnych prvkov, ktoré sa vyberú z množiny vlastností `proprties`.

5.4.4 Záznamy v databáze

Každá požiadavka je jedinečne identifikovateľná objektom typu `ObjectId`[17]. Identifikátory sú použité na indexovanie záznamov v databáze pre ich rýchlejší prístup.

V mongoDB sú záznamy prezentované ako binárne objekty formátu JSON, tzv. BSON. Slovníky v jazyku Python sú neusporiadané objekty, pričom BSON objekty v mongoDB usporiadané sú. Modul `pymongo` transformuje slovníky do podoby, ktorá sa vyskytuje v databáze, čím sa zjednodušuje celá implementácia. Na vloženie záznamu do databázy sa volá metóda `collection.insert_one(query_dict)`. Slovník `query_dict` vyzerá po operácii `insert` v databáze mongoDB nasledovne:

```
"_id" : ObjectId("5aedab0ba817984f50a524a0"),
"http" : "200",
"error_code" : null,
```

[17] ObjectId https://docs.mongodb.com/manual/reference/method/ObjectId/

```
"error_message" : null,
"entity_set" : "Suppliers",
"predecessors" : [ ],
"string" : "Suppliers?$filter=substringof(ContactName, 'Aaa') ne false",
"score" : 5,
"order": [ "_$filter" ],
"_$orderby" : null,
"_$top" : null,
"_$skip" : null,
"_$filter" : { ... }
```

5.5 Výstup fuzzera

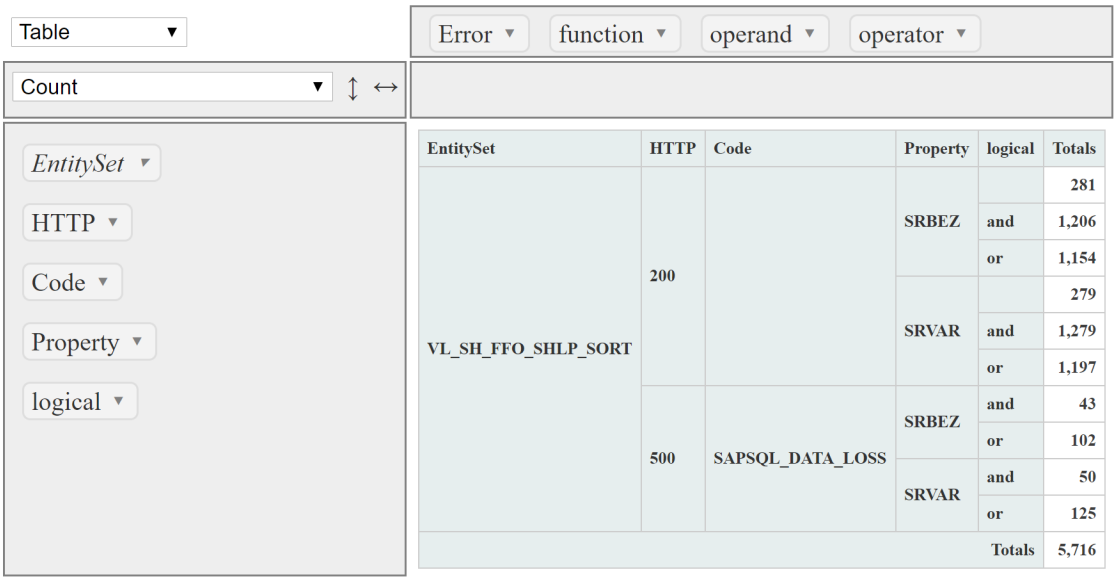
Fuzzer beží v nekonečnom cykle a jeho beh sa preruší vyvolaním signálu SIGINT[18]. Spôsob ukončenia fuzzera bol výsledkom diskusie so zamestnancami spoločnosti SAP. Pre každú entitu sa vytvorí zvlášť súbor pomenovaný s jej názvom a HTTP stavovým kódom. Obsah súboru pozostáva zo všetkých požiadaviek, ktoré prislúchajú danej entite a stavovému kódu HTTP. V súbore sú zoradené požiadavky od najlepšie ohodnotenej po najhoršiu. Tester vie pri prehliadaní súboru okamžite určiť, ktoré typy požiadaviek spôsobili chybové chovanie. Súbor s názvom `overall` obsahuje štatistiku počtu generovaných, úspešných a neúspešných testov. Tieto súbory sa vytvárajú až pri ukončení fuzzera.

Vizualizovateľné štatistiky sa ukladajú do súboru formátu CSV[19]. Do súboru sa pridávajú hodnoty po každom vygenerovanom teste, aby bolo možné sledovať výsledky už počas behu fuzzera. Celkovo sa vytvoria dva typy súborov, kde v jednom sú zahrnuté informácie o všetkých opytovacích funkciách a v druhom sa nachádzajú informácie len o funkcii `$filter`. Súbory sa potom otvoria v prídavnej aplikácii, ktorá je implementovaná v jazyku JavaScript. Táto aplikácia používa na svoj chod knižnicu `pivottalbe.js`[20]. Používateľ tak môže interaktívne pracovať s entitami, vlastnosťami a odpoveďami zo servera priamo v prehliadači.

Obrázok 5.1 demonštruje, aké informácie sú dostupné po spracovaní CSV súboru. V ľavej časti panelu máme hodnoty, ktoré sú viditeľné vo výslednej tabuľke. Po prezretí celkového počtu chýb u jednotlivých operátorov sme schopný určiť, ktorý by mohol spôsobovať chybu. Vidíme, že problémový je logický operátor `or`.

Spôsob takéhoto grafického zobrazovania informácií, spolu s nádychom interaktívnych prvkov, slúži najmä pre jednoduché určovanie potenciálnych chybových stavov. Tester nie je nútený ručne prechádzať všetky typy entít v textových súboroch, ale len tie, ktoré v kontingenčnej tabuľke obsahujú viditeľne najviac chýb.

[18] Klávesové prerušenie bežiaceho procesu
[19] Súbor obsahujúci hodnoty oddelené bodkočiarkou
[20] Knižnica pivottable.js https://pivottable.js.org/examples/



EntitySet	HTTP	Code	Property	logical	Totals
VL_SH_FFO_SHLP_SORT	200		SRBEZ		281
				and	1,206
				or	1,154
			SRVAR		279
				and	1,279
				or	1,197
	500	SAPSQL_DATA_LOSS	SRBEZ	and	43
				or	102
			SRVAR	and	50
				or	125
				Totals	5,716

Obr. 5.1: Grafický prehľad testovanej entity v kontingenčnej tabuľke

Kapitola 6

Výsledky testovania

Nástroj ODfuzz bol testovaný na dvoch SAP Fiori aplikáciách bežiacich v cloud infraštruktúre. Jedna aplikácia slúži na prehliadanie a zasielanie korešpondencií. Druhá umožňuje spravovať nákladové strediská. ODfuzz bol spúšťaný v operačnom systéme Windows 10 (verzia 1607). Celý proces testovania spočíval v zapnutí fuzzera cez noc a spracovaní výsledkov nadránom.

Pomocou nástroja ODfuzz boli objavené celkom štyri chyby rôzneho druhu. Všetky tieto chyby sa vyskytovali len v jednej aplikácii a boli vyvolané nesprávnym spracovávaním opytovacej funkcie `$filter`. Tri z objavených chýb boli nahlásené s nízkou prioritou v internom systéme Jira[1] a opravené vývojármi. Štvrtá chyba bola vyhodnotená ako falošné pozitívum, viď ďalej sekciu 6.2. Chybám bola stanovená nízka priorita z toho dôvodu, že nepredstavovali žiadne riziká pre normálny chod existujúcej aplikácie.

Celkovo bol fuzzer spustený na každej testovanej aplikácii trikrát. ODfuzz cez noc vyprodukoval približne 600 tisíc testov. Počet generovaných testov bol ovplyvnený rýchlosťou internetu na odlišných miestach testovania.

Je vhodné podotknúť, že testovací nástroj nebol spúšťaný na systéme, ktorý obsahuje dáta zákazníkov. Testy prebiehali na oddelenom systéme, ktorý odzrkadľuje zmeny vykonané na vývojovom systéme a je určený primárne na testovanie.

V nasledujúcich troch sekciách si popíšeme chybové stavy, ktoré boli zapríčinené spracovávaním platných požiadaviek. Ukážky požiadaviek boli ručne upravené tak, aby predstavovali minimálnu vzorku, ktorá spôsobuje chybový stav na strane servera. Názvy aplikácií, ich verzie, mená entít a výstupy systémových logov nebudú v tejto práci upresnené, pretože by tak mohlo dôjsť k potencionálnemu zneužitiu aj napriek tomu, že nájdené chyby neboli zamestnancami spoločnosti SAP vyhodnotené ako kritické.

6.1 Chyba SAPSQL_DATA_LOSS

Chyba `SAPSQL_DATA_LOSS` je generovaná interpreterom jazyka ABAP vtedy, keď dôjde k strate dát pri kopírovaní hodnôt z jednej štruktúry do druhej. Chyba sa prejavuje hlavne pri zlom definovaní maximálnej dĺžky reťazca v dokumente metadát. Ako iste vieme, dĺžka reťazca sa určuje atribútom `MaxLength` a podľa tejto hodnoty môžu klientské aplikácie obmedzovať počet znakov vstupného poľa napr. v prehliadači. V prípade, že je reálna veľkosť dátového typu menšia než definovaná, server ohlási chybu.

[1] Jira https://www.atlassian.com/software/jira

Odhalená chyba však nesúvisí s chybnou deklaráciou maximálnej dĺžky. Problém je komplexnejší, nakoľko sa zistilo, že serverová časť očakáva pri niektorých entitách určité vlastnosti na špecifických pozíciách. Po zmene logického operátora alebo poradia vlastností sa chyba viackrát neobjavila. Nasledujúce filtrovacie požiadavky boli spúšťačom chybového chovania:

- `Entity1?$filter=KOART gt '' or LTEXT eq 'JM'`

  Reťazec `'JM'` obsahuje dva znaky, pričom cieľový dátový typ, do ktorého sa reťazec ukladá, môže nadobudnúť hodnotu len jedného znaku. V dokumente metadát je pre vlastnosť `LTEXT` povolených maximálne 30 znakov.

- `Entity2?$filter=HBKID ge '' or BANKL le '!IE|fO'`

  V tomto prípade je problémový reťazec `'!IE|fO'`. V dokumente metadát je vlastnosť `BANKL` typu `Edm.String` a hodnota reťazca môže obsahovať maximálne 15 znakov. Dĺžka cieľového dátového typu, do ktorého sa hodnota reťazca mala priradiť, je najviac 5 znakov.

- `Entity3?$filter=QUEUE_NAME lt ''`
  `or DESCRIPTION lt '27lErFčĹąžt3Äž57Ă´(9)_WqSdv26oOPcxy2'`

  Vlastnosť `DESCRIPTION` môže nadobúdať reťazec o veľkosti 80 znakov. Interpreter jazyku ABAP počíta pri cieľovom dátovom type s veľkosťou maximálne 32 znakov.

Na záver tejto sekcie môžeme zhodnotiť, že nájdené chyby sú špecifické a je veľmi nepravdepodobné, že sa prejavia u zákazníka. Dôvodom je fakt, že klientská časť generuje požiadavky presne v takom formáte, v akom je server schopný ich spracovávať. To znamená, že medzi niektorými vlastnosťami entity sú interné závislosti, ktoré nie sú navonok známe. Problém nastáva v prípade, ak sa bude meniť spôsob zasielania požiadaviek na server. Namiesto zmeny len klientskej časti aplikácie budú vývojári nútený zmeniť aj serverovú časť.

6.2 Chyba /IWBEP/CM_MGW_RT/032

Táto chyba je generovaná vtedy, keď nastane porucha na strane SAP Gateway. Bližší popis chyby znie: „Combining operators of Filter System Query Option is not supported“ a naznačuje, že kombinovanie operátorov vo funkcii `$filter` nie je podporované. Nasledujúca požiadavka vyvolala uvedenú chybu:

```
Entity4?$filter=Location ge '' and Location eq ''
```

Po zmene logického operátora `and` na `or` už OData služba neodpovedala chybovým hlásením. Nástroj ODfuzz považuje túto chybu za skutočnú, nakoľko nie je schopný rozoznať logický kontext medzi dvoma časťami. V tomto prípade je jasné, že nemôže existovať v databáze entita, ktorej miesto `Location` by mohlo byť napr. “Brno” a zároveň aj “Prague”. Chybu považujeme za falošné pozitívum.

6.3 Chyba DBSQL_SQL_INTERNAL_DB_ERROR

Nasledujúca požiadavka spôsobila internú chybu, podľa systémového logu, pri vytváraní SQL príkazu:

```
Entity5?$filter=((RelationshipCategory ne ''
                  or ContactPerson lt '' and RelationshipCategory eq 'a')
                  and AddressNumber le '') or AuthorizationGroup gt ''
```

Ďalej sa v systémovom logu spomína, že index `'en'`, ktorý bol odkazovaný počas vyhodnocovania podmienky neexistuje v tabuľke po operácii `join`.

Po akomkoľvek odstránení logickej časti z požiadavky, nahradení operátora `and` alebo `or`, server znovu neodpovedá chybou HTTP 500.

6.4 Vypršanie spojenia medzi serverom

Požiadavka vygenerovaná nástrojom ODfuzz spôsobila na serveri chybový stav, ktorý vyústil do zrušenia relácie po 5 minútovom odstupe. Server neodpovedal z dôvodu niekoľkonásobného vyčerpania pamäte na halde. Chybový stav spôsobila táto požiadavka:

```
Entity6?$filter=endswith(BusinessPartner, 'Ef') eq true
                or EmailAddress ne 'Géé§éN3é}r"*•pé1,DLvhUé‘'FJé™a(ť¿...'
```

Vo vyššie uvedenej požiadavke sa nenachádza celý reťazec, s ktorým sa porovnáva hodnota vlastnosti `EmailAddress`. Reťazec má dĺžku 61 znakov a obsahuje podobne náhodné znaky ako v ukážke.

Opäť platí to isté, čo u predchádzajúcich chybách. Po zmenení logického operátora alebo vymazaní časti reťazca tak, aby jeho veľkosť bola menej ako 60 znakov, server spracuje požiadavku korektne a reláciu spojenia nezruší.

Kapitola 7

Záver

V tejto práci bol predstavený nástroj na fuzz testovanie biznis aplikácií komunikujúcich prostredníctvom protokolu OData. Nástroj ODfuzz bol spúšťaný celkom na dvoch SAP Fiori aplikáciách, pričom chyby boli objavené iba v jednej z nich. Odhalené chyby súviseli s nevhodným spracovávaním požiadaviek na strane servera, kde dochádzalo najmä k strate informácii počas kopírovania hodnôt z jedného dátového typu do druhého. Všetky chyby boli vyhodnotené ako nekritické, nahlásené a opravené vývojármi.

7.1 Vyhodnotenie

Kvalitu navrhnutého nástroja by sme mohli posudzovať podľa schopnosti odhaliť všetky zraniteľnosti v aplikácii. V skutočnosti nie je možné použiť takúto metriku, no napriek tomu považujeme výsledky za dostatočne kvalitné. Výsledky nástroja taktiež nemožno porovnať so žiadnym iným fuzzerom, nakoľko je ODfuzz vo svojej sfére pôsobenia unikátny.

Ukázalo sa, že fuzzing je relatívne vhodný spôsob testovania, vďaka čomu sa aj odhalilo zopár nedostatkov v existujúcom softvéri. Negatívom tejto metódy je manuálny rozbor chybových stavov. Súbory so štatistikami síce uľahčujú spôsob analyzovania, ale tester je nútený manuálne prejsť všetky nájdeme chyby, vyhodnotiť ich mieru nebezpečenstva a v niektorých prípadoch získať minimálnu vzorku, ktorá spôsobuje chybový stav.

Podobným spôsobom testovania je možné odhaliť chyby, ktoré vznikli po volaní konverzných rutín v programovacom jazyku ABAP. Jedná sa o funkcie, ktoré menia typy ukazovateľov počas behu programu. V jazyku C to môžeme prirovnať k pretypovaniu. Interpreter jazyka ABAP nehlási žiadnu chybu, ak behom konverzie dôjde k poškodeniu dát. Chyba sa prejaví v programe neskôr.

7.2 Ďalší vývoj

Funkcionalita fuzzera sa v budúcnosti môže rozšíriť o podporu ďalších SAP atribútov, ako napr. `sap:addressable`. Týmto atribútom sa určuje, či môže byť entita adresovateľná v požiadavke priamo alebo len cez asociovanú entitu. Ďalej by sa pri generovaní tela funkcie `$filter` mohol používať tzv. backtracking. Jedná sa o algoritmus s podporou návratu, ak riešenie, do ktorého sa dostal nie je vcelku dobré. V našom prípade môžeme algoritmus zužitkovať vtedy, keď nebude možné generovať alebo použiť danú vlastnosť v určitej časti požiadavky. Genetický algoritmus by mohol v budúcnosti rozdeľovať rodičovské požiadavky na dvoch miestach zároveň. Krížením potom vznikne rovnaký počet potomkov, ale s obme-

nenými génmi na dvoch rôznych miestach, čím sa populácia môže stať viac rozmanitá po menšom počte iterácií. ODfuzz momentálne nepodporuje generovanie komplexných dátových typov, čo môže byť cieľom ďalšieho rozvoja. Do úvahy sa berie aj možnosť rozšírenia generátora pre novšiu verziu protokolu OData. Znamená to, že sa nástroj stane použiteľný aj pre OData služby postavené na verzii 4.0.

ODfuzz môže v budúcnosti generovať dáta podľa doménových typov vlastností. Znalosťou domény dátového typu môže fuzzer určiť, aké hodnoty sa do vlastností reálne ukladajú. Príkladom môže byť číslo bankového účtu. Jeho hodnota nemôže byť náhodná, musí spĺňovať určité obmedzenia. Doménové typy by mohol nástroj automatizovane čítať zo servisných operácií. Servisné operácie sa potom musia ručne naprogramovať v jazyku ABAP. Ďalším riešením by mohla byť definícia formátu pre danú vlastnosť v súbore, ktorý sa dá na vstup fuzzera, napr. vo forme regulárnych výrazov.

Finálna verzia fuzzera bude operovať v dvoch rôznych režimoch. V prvom sa fuzzer bude správať tak, ako bolo navrhnuté v tejto práci a jeho ukončenie stanoví používateľ vyvolaním signálu SIGINT. V druhom režime sa nástroj plánuje zapojiť do integračného testovania programového rozhrania. ODfuzz by tak generoval požiadavky rad za radom pre každú vlastnosť v rozmedzí deklarovaného dátového typu vlastnosti. Jednalo by sa o brute-force techniku testovania a celý proces by prebehol v definovanom čase.

Literatúra

[1] *OData and SAP NetWeaver Gateway*. SAP Press, 2014, ISBN 978-1-59229-907-2.

[2] Abeysundara, S.; Giritharan, B.; Kodithuwakku, S.: *A Genetic Algorithm Approach to Solve the Shortest Path Problem for Road Maps*. Prosinec 2005, [Online; 24.04.2018].
URL http://www.scs.pdn.ac.lk/personal/salukak/papers/AbeySaluka.pdf

[3] Chappell, D.: *Introducing OData: Data Access for the Web, the Cloud, Mobile Devices, and more*. 2011, [Online; 01.02.2018].
URL http://www.davidchappell.com/writing/white_papers/Introducing_OData_v1.0--Chappell.pdf

[4] Clarke, T.: *Fuzzing for software vulnerability discovery*. Technická Zpráva RHUL-MA-2009-4, Department of Mathematics, Royal Holloway, University of London, 2009.
URL https://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-04.pdf

[5] Krause, G.: *SAP Annotations for OData Version 2.0*. 2018, [Online; 09.02.2018].
URL https://wiki.scn.sap.com/wiki/display/EmTech/SAP+Annotations+for+OData+Version+2.0

[6] McNally, R.; Yiu, K.; Grove, D.; aj.: *Fuzzing: The State of the Art*. Technická Zpráva DSTO-TN-1043, Defence Science and Technology Organization, Department of Science, Australia, 2012.
URL http://www.dtic.mil/dtic/tr/fulltext/u2/a558209.pdf

[7] Microsoft: *Special Characters*. Leden 2015, [Online; 04.04.2018].
URL https://web.archive.org/web/20150101222238/http://msdn.microsoft.com/en-us/library/aa226544(SQL.80).aspx

[8] Microsoft: *OData - The Protocol for REST APIs*. 2017, [Online; 03.02.2018].
URL http://www.odata.org/documentation/

[9] Microsoft MSDN: *Open Data Protocol (OData)*. 2017, [Online; 02.02.2018].
URL https://msdn.microsoft.com/en-us/library/dd541188.aspx

[10] Moniruzzaman, A. B. M.; Hossain, S. A.: *NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. International Journal of Database Theory and Application*, Duben 2013.

[11] Myers, G.: *The Art of Software Testing, Second edition*. John Wiley & Sons, 2004, ISBN 0-471-46912-2.

[12] Müller, P.: *Automatizované metody hledání chyb v překladačích*. Bakalářská práce, FIT VUT v Brně, Brno, 2008.

[13] Nohejl, A.: *Grammatical Evolution*. Bakalářská práce, Charles University in Prague, Faculty of Mathematics and Physics, Prague, 2009.

[14] Rawat, S.; Jain, V.; Kumar, A.; aj.: *VUzzer: Application-aware Evolutionarz Fuzzing*. 2017, [Online; 27.01.2018].
URL http://sharcs-project.eu/m/filer_public/48/8c/488c5fb7-9aad-4c87-ab9c-5ff251ebc73d/vuzzer_ndss17.pdf

[15] Serebryany, K.; Collingbourne, P.: *Beyond Sanitizers: Guided fuzzing and security hardening*. Říjen 2017, [Online; 21.01.2018].
URL https://llvm.org/devmtg/2015-10/slides/SerebryanyCollingbourne-BeyondSanitizers.pdf

[16] Sestoft, P.: *Systematic software testing*. Únor 2008, [Online; 19.01.2018].
URL https://www.itu.dk/~sestoft/papers/softwaretesting.pdf

[17] Sutton, M.; Greene, A.; Amini, P.: *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007, ISBN 0-32-144611-9.

[18] Takanen, A.; DeMott, J.; Miller, C.: *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008, ISBN 978-1-59693-214-2.

[19] Wright, A. H.: *Genetic Algorithms for Real Parameter Optimization*. [Online; 26.04.2018].
URL http://randolfe.typepad.com/Documents/wright91genetic.pdf

[20] Zalewski, M.: *American fuzzy lop*. 2017, [Online; 28.01.2018].
URL http://lcamtuf.coredump.cx/afl/

[21] Zusman, M.: *Fuzzing 101*. 2008, [Online; 29.01.2018].
URL https://fuzzinginfo.files.wordpress.com/2012/05/fuzzing-2.pdf

Príloha A

Obsah priloženého pamäťového média

Obsahom priloženého pamäťového média je:

- text bakalárskej práce,
- zdrojové súbory pre vysádzanie textu bakalárskej práce,
- zdrojové súbory navrhnutého nástroja ODfuzz,
- manuál pre inštaláciu závislostí a spustenie nástroja.