Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vylepšenie typového systému medzi-jazyka FuzzIL vo fuzzeri Fuzzilli

Diplomová práca

2021
Bc. Samuel Sládek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vylepšenie typového systému medzi-jazyka FuzzIL vo fuzzeri Fuzzilli

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Richard Ostertág, PhD.

Bratislava, 2021
Bc. Samuel Sládek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Samuel Sládek
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Vylepšenie typového systému medzi-jazyka FuzzIL vo fuzzeri Fuzzilli
Improvements to the type system of intermediate language FuzzIL in fuzzer Fuzzilli

Anotácia: Fuzzer Fuzzilli implementuje svoj vlastný typový systém používaný na hľadanie premenných daného typu a na určovanie vlastností a metód premenných. Nepresné informácie o type premenných môžu viesť ku generovaniu neplatného JavaScriptu, ktorý potom interpreter odmietne.

Cieľ: V diplomovej práci sa pokúsime zlepšiť tento typový systém, najmä jeho mechanizmus odvodzovania typov jednotlivých premenných, čím spresníme informácie o typoch premenných vo vygenerovanom skripte a zväčšíme tak šancu na vygenerovanie korektného programu v jazyku JavaScript. Týmto dokážeme generovať viac zaujímavejších skriptov v kratšom čase, ktoré pokryjú viac kódu samotného interpretera.

Vedúci: RNDr. Richard Ostertág, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 07.10.2020

Dátum schválenia: 08.10.2020

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Chcel by som sa poďakovať môjmu školiteľovi RNDr. Richardovi Ostertágovi, PhD. za pomoc pri písaní práce a konštruktívnu spätnú väzbu.

Implementačná časť tejto práce vznikla počas stáže v spoločnosti *Google*, preto moja vďaka patrí môjmu vtedajšiemu manažérovi Samuelovi Großovi za cenné rady a najmä kontrolu kódu, ktorý je súčasťou práce.

Abstrakt

Fuzz testovanie je jedna z najpopulárnejších techník na hľadanie implementačných chýb a bezpečnostných zraniteľností v softvéri. Pred 2 rokmi bol vytvorený voľne dostupný fuzzer Fuzzilli pre JavaScriptové enginy. Mnohé fuzzery v tejto oblasti mali problémy s generovaním sémanticky korektných programov. Autor fuzzera Fuzzilli priniesol novú metódu, definovaním vlastného medzijazyka FuzzIL, ktorým dokázal zvýšiť podiel korektne vygenerovaných programov.

V tejto práci sa zaoberáme typovým systémom medzijazyka FuzzIL. Vyvinieme novú metódu na odvodzovanie typov, ktorá nielen opraví doteraz zle odvádzané typové informácie, ale ich aj spresní (typy, ktoré boli zbytočne konzervatívne v pôvodnej metóde na odvodzovanie). Túto novú metódu následne vhodne skombinujeme s pôvodnou metódou odvodzovania, čím zoptimalizujeme výkonnosť fuzzera ako celku. Následne ukážeme, že Fuzzilli s naším vylepšením dokáže generovať zaujímavejšie programy. Takto dokážeme otestovať viac častí JavaScriptového enginu v kratšom čase.

Na záver v práci spravíme rôzne časové a pamäťové optimalizácie, ktorými zrýchlime samotné generovanie programov. Na referenčných skúškach ukážeme, že Fuzzilli bude schopné generovať programy rýchlejšie s použitím menšej pamäte.

Kľúčové slová: fuzz testovanie, JavaScript, interpreter, Fuzzilli, odvodzovanie typov

Abstract

Fuzz testing is one of the most popular techniques for finding implementation bugs and security vulnerabilities in a software. An open-source fuzzer Fuzzilli for JavaScript engines was created 2 years ago. Many fuzzers in this area have had trouble creating the semantically valid programs. The author of Fuzzilli introduced a new method by defining his own intermediate language FuzzIL, which managed to increase the ratio of valid generated programs.

In this thesis, we will deal with the type system of the intermediate language FuzzIL. We will develop a new method for type inference that not only will fix currently misinferred type information, but will also make it more specific (the types that were unnecessarily conservative in the old inference method). This new method is then suitably combined with the original type inference method, thus optimizing the performance of the fuzzer as a whole. Later, we will illustrate that Fuzzilli with our improvements can generate more interesting programs. This way, we can test more parts of a JavaScript engine in less time.

Finally, we will perform various time and memory optimizations, which will speed up the actual generation of programs. Using benchmarks, we will demonstrate that Fuzzilli can generate programs faster with less memory consumption.

Keywords: fuzz testing, JavaScript, engine, Fuzzilli, type inference

Obsah

Zoznam obrázkov

Úvod

V poslednom desaťročí počet ľudí, ktorý používajú internet každý deň, výrazne narástol. Ľudia používajú internet na mnoho rozličných účelov. Od získavania vedomostí, komunikácie s inými ľuďmi až po trávenie voľného času. Najbežnejším prístupom k informáciám na internete je webový prehliadač, ktorý ľahko sprístupňuje dáta aj menej technicky zdatným ľuďom. Keďže webové prehliadače sú používané tak často a tak rozmanitou skupinou ľudí na prístup k rýchlo vyvíjajúcemu sa svetu internetu, tak aj samotný prehliadač musí byť vyvíjaný rýchlo a prispôsobovať sa potrebám používateľov. Bežného používateľa zaujíma najmä výkon a keďže konkurencia je veľká, musia sa aj vývojári týmto zaoberať ako jednou z kľúčových vlastností. Zároveň však nesmieme zabúdať na bezpečnosť samotných používateľov, ktorý bežne nie sú technicky orientovaný a riziko si neuvedomujú. Dôležitou časťou prehliadača, kde sa zraniteľnosti vyskytujú pomerne často je JavaScript engine. Naša práca bude motivovaná práve zlepšením hľadania chýb v implementáciách JavaScript enginu.

Na prevenciu proti zavádzaniu nových chýb (potencionálne bezpečnostných zraniteľností) do softvéru máme ako vývojári mnoho nástrojov a usmernení. Môžeme písať rôzne testy, ktoré musia úspešne prejsť pred zverejnením aktualizácie, používať zaužívané paradigmy, kontrolovať pridaný kód dodatočným programátorom a podobne. No pri tak veľkých produktoch ako sú JavaScript enginy, to často nestačí. Preto majú vývojárske firmy často aj zložitejšie nástroje na hľadanie chýb, ktoré si však nemôžu dovoliť spúšťať pri každej aktualizácii kódu. Jedna takáto metóda je známa ako fuzz testovanie.

Fuzz testovanie je založené na generovaní vstupov, ktoré následne posielame do softvéru a sledujeme jeho správanie. Najjednoduchšie je generovať vstupy čisto náhodné, no pri softvéri akým je JavaScript engine budú takmer všetky vstupy zamietnuté pred vykonávaním a nič neotestujeme. Generovať zaujímavé a korektné (syntakticky aj sémanticky) vstupy teda nie je jednoduchá úloha. Bežné fuzz testovanie na JavaScript enginoch sa púšťa aj na niekoľko dní na stovkách strojoch. Dnes aj tým najmodernejším technikám na generovanie vstupov, trvá dlho, kým vytvoria dostatočne veľa zaujímavých vstupov, ktoré potencionálne nájdu chybu.

Problém generovania sémanticky korektných vstupov (v našom prípade JavaScriptových programov) riešia fuzzery rôzne. V našej práci sa pozrieme na fuzzer Fuzzilli [4],

ktorý sa s touto úlohou vysporiadal zavedením medzijazyka FuzzIL. FuzzIL je nadizajnovaný tak, aby sa v ňom ľahšie generovali programy, ktoré budú korektné. Tieto programy následne stačí preložiť do JavaScriptu.

Naša práca sa zaoberá vylepšením typového systému jazyka FuzzIL, ktorým by sme vedeli ešte viac zvýšiť pomer vygenerovaných programov, ktoré sú korektné. Terajšia implementácia občas spôsobuje, že fuzzer má nesprávnu typovú informáciu, poprípade informácia nie je dostatočne presná (odvodzovanie bolo príliš konzervatívne). Toto fuzzer obmedzuje a nedovolí mu generovať zaujímavejšie programy. Stále však musíme mať na pamäti, že samotný fuzzer Fuzzilli je len heuristika pripomínajúca simulované žíhanie a tak prístup k úplne presným typom nesmieme brať ako cieľ, ale ako možnú cestu. V takejto heuristike aj nepresný náhodný krok môže pomôcť generovať lepšie programy. Musíme brať do úvahy a optimalizovať aj rýchlosť generovania programov.

Práca sa skladá zo 4 častí:

V prvej kapitole sa pozrieme bližšie na problematiku bezpečnosti JavaScriptových enginov a tiež na to, ako využiť v tejto oblasti fuzz testovanie. Podrobnejšie si tiež predstavíme rôzne paradigmy používané v dnešných fuzzeroch.

Druhá kapitola predstavuje samotný fuzzer Fuzzilli a jeho fungovanie. Predstavíme si tiež medzijazyk FuzzIL, jeho vlastnosti a hlavne samotný typový systém. Zamyslíme sa nad slabými stránkami tohto systému a v ďalších kapitolách ho vylepšíme.

V tretej kapitole predstavíme náš návrh vylepšenia odvodzovania typov programov v medzijazyku FuzzIL, ktoré viac krát iterujeme a na záver porovnáme beh fuzzera bez nášho vylepšenia a s ním. Tento beh bude dlhší, aby sme porovnali vplyv za podmienok, pri ktorých Fuzzilli používajú výskumníci v oblasti počítačovej bezpečnosti.

Štvrtá kapitola sa zameriava skôr na rýchlosť samotného generovania programov. Pokúsime sa v nej o niekoľko rôznych časových a pamäťových optimalizácií. Tieto optimalizácie následne porovnáme s pôvodnou verziou pomocou referenčných skúšok (angl. *benchmark*), ktoré už sú naprogramované ako súčasť zdrojového kódu fuzzera.

Kapitola 1

Problém bezpečnosti JavaScript enginov

V poslednej dekáde sa výrazne dostal do popredia programovací jazyk JavaScript. A to najmä vďaka popularite webových stránok a aplikácií, z ktorých je v dnešnej dobe veľká väčšina naprogramovaná v JavaScripte (alebo ho inak používa). Podľa [2] približne 96,8% všetkých webstránok používa JavaScript (október 2020). Na zobrazovanie týchto webstránok sa dnes používajú webové prehliadače a teda JavaScript engine sa stal základnou časťou týchto prehliadačov. Avšak JavaScript nabral na popularite aj mimo webových aplikácií. Dnes ho vieme používať ako programovací jazyk aj na serveroch alebo vo vstavaných systémoch (angl. *embedded system*). Medzi najznámejšie JavaScript enginy patria:

- **V8**[1] – Aktuálne najznámejší a najpoužívanejší engine vytvorený spoločnosťou Google, používa ho najmä webový prehliadač Chrome, no je aj súčasťou aplikačného rámca (angl. *framework*) Node.js umožňujúceho písať serverové skripty.
- **JavaScriptCore**[2] – Interpreter vytvorený spoločnosťou Apple, používaný vo webovom prehliadači Safari.
- **Spidermonkey**[3] – Prvý JavaScript engine, aktuálne udržiavaný spoločnosťou Mozilla Foundation, používaný webovým prehliadačom Mozilla Firefox.
- **JerryScript**[4] – „Kompaktná“ implementácia JavaScript enginu určená na spúšťanie JavaScript skriptov na zariadeniach s obmedzenými zdrojmi ako sú mikrokontrolery[3].

[1] `https://github.com/v8/v8`
[2] `https://github.com/WebKit/webkit`
[3] `https://github.com/mozilla/gecko-dev`
[4] `https://github.com/jerryscript-project/jerryscript`

Vzrastajúca popularita JavaScriptu však prináša aj záujem útočníkov, snažiacich sa využiť slabiny jazyka (použitého enginu) na získanie prístupu ku koncovému používateľovi, a teda napríklad k jeho dátam. Pozornosť výskumníkov v oblasti počítačovej bezpečnosti však táto oblasť priťahuje aj z iných dôvodov. Medzi samotnými vývojármi JavaScript enginov prebieha súťaž o výkonnejšiu implementáciu, získavajúcu potenciónálne viac používateľov, čím často zavádzajú aj nové bezpečnostné chyby. Taktiež sa tieto enginy stali obrovskými a zložitými, čo zvyšuje šancu na vznik bezpečnostnej alebo inej chyby pri hoci aj malých aktualizáciách kódu. Tieto chyby môžu byť zároveň zraniteľnosťami, ktoré útočník môže zneužiť na skompromitovanie obete. Môže sa mu to podariť aj napriek tomu, že sa nachádza za ochranou pred neoprávneným vstupom (angl. *firewall*).[7]

Najväčšie šance na úspech majú útočníci práve vo webových prehliadačoch, ktoré dnes používa dennodenne každý. Prvé webové prehliadače podporovali len jeden proces v čase prehliadania. Čo však s nástupom možnosti prehliadania viacero stránok naraz (takzvané „taby“) začalo byť nepraktické. Ak totiž zobrazovanie všetkých stránok beží v jednom procese, stačí, aby jedna stránka prestala byť responzívna a prestanú byť responzívne aj ostatné prehliadané stránky.
Iný benefit je bezpečnosť. Operačné systémy totiž dávajú možnosť obmedziť práva jednotlivých procesov, takže niektorým procesom môžeme odoprieť určité funkcie. Napríklad prehliadač môže obmedziť prístup procesu zodpovedného za renderovanie k súborom, ktoré spracovávajú používateľove vstupy.

Dnes už v modernom webovom prehliadači beží viacero rôznych procesov[9]. Na vrchole všetkých procesov býva prehliadačový proces (angl. *browser process*), ktorý všetko koordinuje. Tento proces je ďalej rozdelený do ďalších procesov v závislosti od prehliadača, no bežne to sú: renderovací proces, GPU proces, úžitkový proces (angl. *utility process*) a doplnkový proces (angl. *plugin process*). Z pohľadu bezpečnosti je najzaujímavejší renderovací proces. Slúži totiž na parsovanie rôznych zdrojov ako sú obrázky, HTML alebo JavaScript (ktorý musí byť aj vykonaný) a mnohé ďalšie. Musí spracovávať často nedôveryhodné a zložité zdroje, preto ho môžeme považovať za najviac exponovaný. Práve preto má tento proces limitovaný prístup k zdrojom. Na zneužitie zraniteľnosti teda treba aj inú zraniteľnosť mimo tohto procesu. Napriek tomu, vo väčšine prípadov na využitie zraniteľnosti potrebujeme vykonať určitý kód v renderovacom procese[4]. A preto je dôležité zaistiť bezpečnosť JavaScript enginov.

JavaScript enginy sú bežne naprogramované v jazyku C alebo C++. Preto sa v nich často vyskytujú chyby bežné v týchto jazykoch. Napríklad pretečenie vyrovnávacej pamäte (angl. *buffer overflow*), odkazy na visiaci ukazovateľ (angl. *dangling pointer referencies*) a podobné.

Príklad ako môže útočník zneužiť pretečenie vyrovnávacej pamäte založené na zásobníku je nasledovný:

1. Najprv útočník využije pretečenie vyrovnávacej pamäte v poli.

2. Následne môže prepisovať hodnoty za hranicami pamäte alokovanej pre toto pole. Za týmto poľom môže byť uložený stav programu (aktuálne vykonávanej funkcie) ako napríklad návratová adresa. V tomto prípade útočník môže prepísať túto adresu.

3. Keď funkcia skončí, preskočí na adresu vloženú útočníkom. Napriek tomu, že pôvodne mala vrátiť riadenie programu naspäť odkiaľ bola volaná. Na tejto adrese už môže mať útočník vložený napríklad aj svoj vlastný kód.

Napriek tomu takéto jednoduché útoky sú už minulosťou vďaka technikám ako StackGuard[10].

Dnes už útočníci musia kombinovať viacero komplexných metód, prípravy pamäte, odkrytia pamäte a iné ďalšie. Častokrát si ale techniky na ochranu vyžadujú veľké zmeny. A opravy sa tak ku koncovým používateľom dostávajú s oneskorením.

Preto na zvýšenie bezpečnosti svojich produktov, spoločnosti vyvíjajúce JavaScript enginy často vypisujú súťaže alebo takzvané „bug bounty“, čo je vlastne výzva pre verejnosť na nájdenie zraniteľností. Jednotlivci za takýto objav a nahlásenie spoločnosti, vyvíjajúcej tento produkt, môžu získavať rôzne odmeny. Medzi spoločnosti, ktoré vypisujú takéto súťaže patrí napríklad Google[5].

Môže sa ale stať, že nájdená zraniteľnosť nebude nahlásená, ale zneužitá. Preto je dôležité mať správne nastavené procesy už pri vývoji softvéru tak, aby sme odchytili možné chyby predtým než sa dostanú do produkcie. Existuje mnoho spôsobov testovania softvéru. Statická analýza, podrobné kontroly kódu dodatočným vývojárom, jednotkové testy (angl. *unit testing*) a mnohé ďalšie. V našej práci sa ale pozrieme na metódu známu pod pojmom fuzz testovanie[6].

1.1 Fuzz testovanie

V tejto podkapitole si najprv ukážeme čo rozumieme pod pojmom fuzz testovanie. Neskôr si ukážeme akými rôznymi spôsobmi ho vieme využiť. Na záver podkapitoly sa zamyslíme nad tým, ako tieto spôsoby využiť na fuzz testovanie JavaScript enginov.

Fuzz testovanie je pomerne stará metóda automatického testovania softvéru. Hlavnou ideou je náhodné (nie nutne úplne náhodné) generovanie rôznych možných vstupov,

[5] `https://www.google.com/about/appsecurity/reward-program`

ktoré následne opakovane posielame softvéru na vstup. No namiesto sledovania výstupu ako v bežnom testovaní, sledujeme samotný priebeh správania sa systému spracúvajúceho tento vstup. To znamená, že sa zaujímame o vedľajšie efekty, ako napríklad to, či a s akou chybou program spadol, koľko zdrojov použil a podobne. Keďže hľadáme chyby, tak nás zaujímajú najmä priebehy, pri ktorých sa v softvéri nejaká vyskytne. Príkladom takejto chyby môže byť poškodenie pamäte, zlyhanie programu alebo extrémne použitie zdrojov. V takomto prípade dostaneme vstup pri ktorom nastáva táto chyba a môžeme ju ďalej analyzovať. Či už za účelom opravenia chyby v softvéri alebo vytvorenia zraniteľnosti pre útok na softvér. Obrovskou výhodou je ľahká škálovateľnosť.

Naším najväčším problémom a zároveň najväčšou zbraňou v tomto spôsobe testovania je náhodnosť. Na jednej strane posielanie čisto náhodných bitov môže objaviť nečakané správanie softvéru, ktoré by sme bežnou manuálnou kontrolou nenašli. Na druhú stranu pokiaľ softvér zamieta isté typy vstupov, môže sa stať, že veľa našich vygenerovaných vstupov bude zamietnutých predtým než sa ich softvér pokúsi spracovať. V takomto prípade naše testovanie nič zaujímavé neukáže.

A to je presne prípad testovania JavaScript enginov. V prvej fáze totiž engine parsuje samotný vstupný kód a pokiaľ to nie je korektný JavaScript, zamietne ho v tejto fáze bez pokusu vykonať ho. Náhodný prúd bitov preto nebude pre nás vhodná voľba. Mohli by sme síce nájsť niektoré chyby v samotnej časti programu určenej na parsovanie vstupu, no tieto chyby nie sú z bezpečnostného hľadiska zaujímavé. Na fuzz testovanie JavaScriptových enginov nám však syntaktická korektnosť stačiť nebude. Potrebujeme aj sémantickú korektnosť. Keď engine v priebehu interpretovania vstupného kódu spadne na chybe počas behu programu (angl. *runtime error*) nedostaneme žiadny zaujímavý priebeh a tiež časť vygenerovaného kódu za touto chybou sa vôbec nevykoná.

No existujú techniky, ktorými vieme generovať náhodné vstupy múdrejšie. Predstavme si teda základné prístupy a to *fuzz testovanie založené na mutáciách* a *fuzz testovanie založené na generovaní*. Taktiež si predstavíme techniku, ktorá môže byť ľubovoľne skombinovaná s tými predchádzajúcimi a to je *riadené fuzz testovanie*.

1.1.1 Fuzz testovanie založené na mutáciách (angl. *Mutation-based fuzzing*)

Namiesto toho aby sme vždy nanovo generovali náhodný vstup si môžeme udržovať množinu zaujímavých vstupov. Keď teraz chceme vygenerovať ďalší vstup, tak ho nebudeme tvoriť odznova, ale náhodne si vyberieme z našej množiny a urobíme na ňom len malé zmeny. Následne ho pošleme do samotného softvéru, kde sledujeme jeho správanie. „Malé zmeny“ môžeme interpretovať rôzne podľa potreby. Buď to môže

byť jednoduché prevrátenie bitu, doplnenie/nahradenie nejakej konštanty (príliš veľkej alebo malej) a podobne. Ako si môžeme všimnúť veľa závisí od našej množiny vstupov, ktorú mutujeme (táto množina sa počas fuzz testovania môže meniť) a tiež od zvolenia povolených mutácií.

Stále sa nám však môže stať to čo predtým a teda vygenerujeme vstup, ktorý softvér zamietne. Zvolením správnych mutácií však vieme počet odmietnutých vstupov výrazne zmenšiť. Niektoré jednoduché validačné pravidlá môžeme kontrolovať aj osobitne pred spúšťaním softvéru v samotnom fuzzeri, keďže samotné spustenie softvéru môže byť náročné (časovo alebo na iné zdroje).

V našom prípade, keď by sme chceli priamo mutovať vstupné JavaScriptové programy, čaká nás náročná úloha. Aj keby sa nám podarilo vygenerovať syntakticky korektný JavaScript, stále ťažko zabezpečíme aj sémantickú korektnosť skriptu. Napríklad majme v našej množine tento zaujímavý vstup, ktorý chceme zmutovať.

Program 1.1: JavaScript kód pred mutáciou

```
1  const v0 = {}
2  const v1 = []
3  for (const v2 of v1) {
4  }
```

Ak teraz na riadku *3* nahradíme premennú *v1* za premennú *v0* náš kód ostane syntakticky korektný, ale sémanticky už nie, keďže konštrukt *for of* môžeme používať, len na premenných, ktoré sú iterovateľné.

Existujú však práce ([8] alebo [5]), ktorým sa podarilo dosiahnuť zmysluplné mutácie aj pri priamom fuzz testovaní JavaScriptových programov.

1.1.2 Fuzz testovanie založené na generovaní (angl. *Generative fuzzing*)

V tejto technike sa snažíme vygenerovať zaujímavý vstup z ničoho, poprípade môžeme dogenerovať len časť možného vstupu. Toto dáva nám, ako autorom fuzzera, silný prostriedok na to, aby sme vedeli generovať vstupy presne aké chceme, len s malou dávkou náhody. Dokonca vieme nastaviť pravidlá na generovanie tak, že nikdy nevygenerujeme nekorektný vstup alebo budeme generovať len určitú podtriedu vstupov, ktorá nám v danej chvíli príde zaujímavá, alebo máme podozrenie, že práve takto môže vyzerať zraniteľnosť.

Toto sa nám môže hodiť práve pri fuzz testovaní JavaScript enginov, keďže JavaScript (presnejšie ECMAScript 5) je bezkontextový jazyk a teda pre neho existuje bezkontextová gramatika podľa ktorej môžeme generovať vstupy[1].

Na začiatku si zoberieme začiatočný literál a v každom kroku len náhodne vyberieme

pravidlo z gramatiky, ktoré použijeme. Samozrejme musíme ošetriť, aby sme sa nezacyklili pri generovaní, poprípade obmedzili veľkosť vygenerovaného slova. Pri takejto gramatike sa nám ale oveľa ťažšie uplatňujú všetky sémantické pravidlá ako pri fuzz testovaní založenom na mutáciách. Sémantický vzťah môžu mať totiž aj literály, ktoré sú vzdialené v odvodzovacom strome, na základe ktorého sme slovo vygenerovali.

Program 1.2: Príklad možného nekorektného JavaScriptu vygenerovaného gramatikou

```
1 let v0 = new Set()
2 for (let v1 = 0; v1 < 1; v1++) {
3     v0 = v1 + 1
4     const v2 = v0.has(v1)
5 }
6 ...
```

Napríklad ukážka kódu 1.2 mohla vzniknúť takýmto generovaním na základe gramatiky. No ako môžeme vidieť tento kód skončí na riadku *4* chybou:

TypeError: v0.has is not a function

A teda vygenerovaný kód na ďalších riadkoch sa ani nevykoná a generovali sme ho zbytočne.

Aby nás neobmedzovali sémantické chyby, tak vieme každý vygenerovaný JavaScript príkaz zaobaliť do *try-catch* bloku ako na ukážke 1.3.

Program 1.3: Príklad možnej opravy nekorektného kódu pomocou try-catch blokov

```
1  try {
2      let v0 = new Set()
3  } catch(e) {}
4  try {
5      for (let v1 = 0;v1 < 1; v1++) {
6          try {
7              v0 = v1 + 1
8          } catch(e) {}
9          try {
10             const v2 = v0.has(v1)
11         } catch (e) {}
12     }
13 } catch(e) {}
14 ...
```

Aj keď na prvý pohľad by to nemalo meniť vykonávanie samotného kódu enginom, nie je to pravda. Pre JIT kompilátor nachádzajúci sa vnútri enginu sú to úplne iné

programy, aj keď sú na prvý pohľad identické (čo sa týka správania programu). Teda na prvý pohľad identický kód otestuje úplne iné časti enginu a niektoré časti by boli dokonca neotestovateľné, čo nie je ideálne.

1.1.3 Riadené fuzz testovanie (angl. *Guided fuzzing*)

Bežné fuzz testovanie nedostáva žiadnu spätnú väzbu od softvéru, ktorý testujeme. Jediné čo vieme je, či sme našli chybu alebo nie. Aj napriek tomu, že daný vstup nenájde priamo chybu v softvéri, mali by sme vedieť povedať, či bol daný vstup „dobrý“ a mali by sme skúšať viac vstupov podobných jemu. Alebo „dobrý“ nebol a radšej podobné vstupy ani neskúšajme. Idea riadeného fuzz testovania teda navrhuje aby sme si definovali metriku, ktorou budeme vedieť povedať či a ako bol daný vstup „dobrý“.

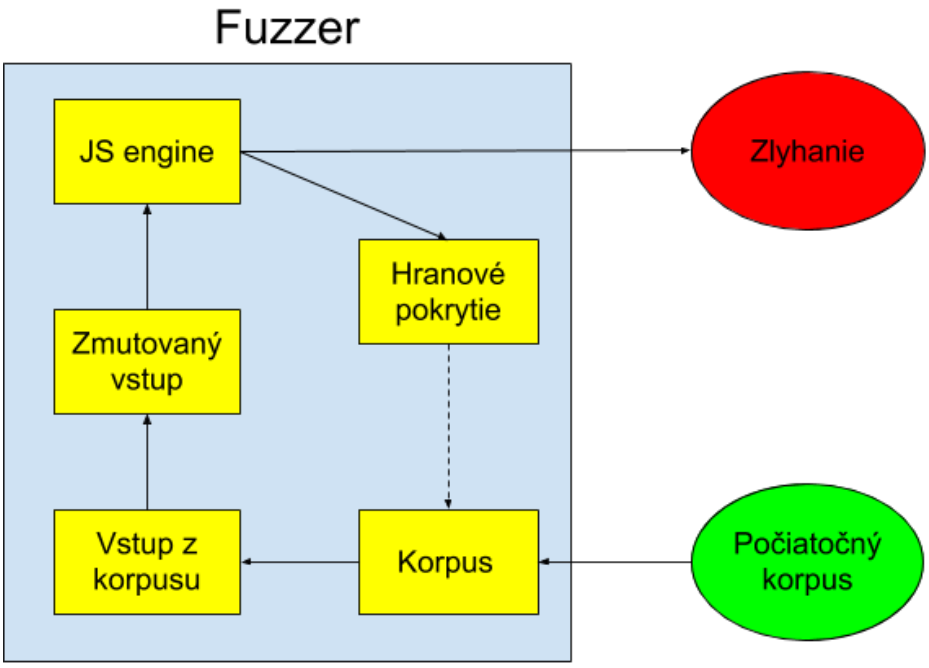
Keď si metriku zvolíme, riadené fuzz testovanie nám dáva jednoduchý návod. Podobne ako pri fuzz testovaní založenom na mutáciách si budeme udržovať množinu zaujímavých vstupov a tie môžeme skúšať mutovať alebo do nich dogenerovávať časti vstupov. Túto množinu zvykneme nazývať **korpus** (angl. *corpus*).

Po zmutovaní vstupu a spustení softvéru na tomto vstupe dostaneme spätnú väzbu, ktorú vyhodnotíme našou metrikou. Ak nám metrika povie, že vstup bol dobrý, uložíme ho naspať do korpusu a v ďalších iteráciách ho môžeme ďalej mutovať. Ak nám metrika naopak povie, že tento vstup nebol zaujímavý rovno ho zahodíme.

Takto sa nám náš korpus bude ale iba zväčšovať, preto musíme nastaviť pravidlá aj na mazanie vstupov z korpusu. Najčastejšie používané pravidlo je mazanie najstarších vstupov. Toto je pochopiteľné, keďže je pravdepodobné, že vstupy, ktoré sú už dlho v korpuse sme už viac krát použili ako základ pre nové vstupy. Ďalšou mutáciou by sme zrejme dostali podobný vstup ako niektorou mutáciou predtým a tak tento nový vstup by už zrejme nebol až taký zaujímavý.

Tento prístup nám umožní pripravovať cielené a komplexné vstupy, ktoré dokážu otestovať rôzne časti softvéru.

To na čom celá metóda stojí je teda metrika, ktorú si zvolíme. Zároveň túto metriku musí byť ľahké vypočítať pre daný beh softvéru s daným vstupom. Všeobecne uznávanou dobrou metrikou je takzvané „hranové pokrytie“ (angl. *edge coverage*). Hranové pokrytie sa odkazuje na **grafovú reprezentáciu** testovaného softvéru (angl. *control flow graph*). Táto metrika považuje vstup za zaujímavý ak objavil novú hranu v grafovej reprezentácii nášho softvéru. Zaujímavú ju môžeme považovať hlavne z toho dôvodu, že ukazuje aké správanie nášho softvéru sme boli schopný zachytiť (akú hranu si softvér vybral pri svojom vykonávaní) novo-vytvoreným vstupom. Do korpusu teda chceme pridávať len také vytvorené vstupy, ktoré pri spustení v testovanom softvéri zachytili nové správanie.



Obr. 1.1: Pokrytím riadený fuzzer

Tu si ale musíme uvedomiť, že samotné pozorovanie správania sa programu zvonku nestačí (ako pri obyčajnom fuzz testovaní). Na to aby sme zistili hranové pokrytie musíme inštruovať samotný testovaný softvér aby tak urobil. Ako to konkrétne urobiť už záleží na samotnom testovanom softvéri. Existuje mnoho nástrojov, ktoré nám pomôžu a často stačí len skompilovať program s iným prepínačom a doprogramovať prenos tejto informácie do fuzzera, napríklad pomocou zdieľanej pamäte. Nesmieme zabúdať, že zbieranie tejto informácie môže spomaliť samotný beh softvéru na vygenerovaných vstupoch.

Na obrázku 1.1 máme možnosť vidieť ako takýto riadený fuzzer funguje.

Fuzzer Fuzzilli, ktorým sa v našej práci budeme zaoberať, je založený presne na tomto princípe a teda je tiež riadený pokrytím. No uvádza aj mnohé ďalšie vylepšenia, ktoré ho robia v konkurencii iných fuzzerov výnimočným a zároveň užitočným nástrojom pre výskumníkov v oblasti počítačovej bezpečnosti. Bližšie si ho predstavíme v nasledujúcej kapitole.

Kapitola 2

Fuzzilli

V tejto kapitole si predstavíme samotný fuzzer Fuzzilli, ktorý sa neskôr v tejto práci pokúsime vylepšiť. Fuzzer bol pôvodne navrhnutý v práci [4]. Odvtedy sa tento nástroj výrazne vyvinul, no je stále založený na jeho pôvodných myšlienkach a preto táto kapitola bude výrazne inšpirovaná touto prácou. Nástroj je naprogramovaný v programovacom jazyku *Swift*[1] s malými časťami implementovanými v jazyku *C*. Implementácia je voľne dostupná na službe Github[2].

Fuzzilli je pokrytím riadený fuzzer, ktorý sa snaží generovať rôzne JavaScript programy. Udržiava si ich v korpuse, pri každej iterácií vyberie jeden náhodný vstup a podľa istých pravidiel ho zmutuje alebo doňho dogeneruje ďalší kód. Tento zmenený kód následne skúsi poslať na vykonanie konkrétnej implementácii JavaScriptového enginu, ktorý testujeme. Toto vykonávanie sledujeme a nastavíme engine tak, aby zistil aj hranové pokrytie vytvorené týmto vstupom.
Vykonávanie môže skončiť jedným zo 4 prípadov[3]:

- **pád** – v kóde označené ako *crashed*
  Našli sme chybu v JavaScript engine, uložme si tento program na disk a môžeme ho neskôr analyzovať.
- **zlyhanie** – v kóde označené ako *failed*
  Skript bol odmietnutý JavaScriptovým enginom kvôli syntaktickej chybe. Alebo spadol počas behu programu kvôli sémantickej chybe. Môžeme ho teda rovno zahodiť.
- **vykonávanie dobehlo úspešne** – v kóde označené ako *succeeded*
  V tomto prípade sa pozrieme na výsledné hranové pokrytie. Ak sme objavili aspoň jednu novú hranu, tak skript pridáme do korpusu, inak ho zahodíme.

[1] `https://swift.org`
[2] `https://github.com/googleprojectzero/fuzzilli`
[3] v prílohe na /fuzzilli/Sources/Fuzzilli/Execution/Execution.swift

- **vykonávanie prekročilo časový limit** – v kóde označené ako *timedOut*
  Vykonávanie sa zacyklilo alebo sme len vygenerovali skript, ktorého vykonávanie trvá dlho. V oboch prípadoch si nemôžeme dovoliť zdržanie a radšej vykonávanie prerušíme a skript zahodíme. Takto ušetríme čas a môžeme skúšať vygenerovať iné programy.

Následne celú iteráciu opakujeme v cykle.

Fuzzilli sa snaží generovať vstupy na ktorých sa bude JavaScriptový engine správať deterministicky. Keď nájdeme nedeterministický vstup, ktorý skončí pádom, nezískame až tak veľa. Tento vstup môže byť totiž falošne pozitívny. Napríklad kvôli situácii s nedostatkom pamäte systému ako celku.[4]
Takisto analýza je náročnejšia a zreprodukovať správanie enginu môže byť náročné. Zabezpečiť generovanie čisto deterministických vstupov je však nemožné, keďže to závisí na implementácii samotného enginu. Vieme sa však vyhýbať generovaniu konštruktov ako napríklad *Math.random()*.

Najpomalšia a teda zároveň najkritickejšia (čo sa týka časovej výkonnosti fuzzera) časť je samotné vykonávanie vygenerovaného programu JavaScriptovým enginom. Aby sme časovo ušetrili a nemuseli pre každé vykonávanie kódu spúšťať nový proces, Fuzzilli testuje enginy v takzvanom REPRL móde (z angl. *read-eval-print-repeat loop*). Snaží sa teda znovu využiť už existujúci proces, zresetovať vnútorný stav enginu a načítať ďalší program zo zdieľanej pamäte.

V našej práci sa budeme predovšetkým venovať systému tvorenia vhodných vstupov a preto sa nebudeme zaoberať ďalšími detailmi o komunikácií fuzzera s JavaScriptovým enginom, respektíve vyhodnocovaním hranového pokrytia. Viac podrobností si čitateľ môže nájsť v pôvodnej práci [4].

Ako sme videli v minulej kapitole najväčším problémov známych techník fuzz testovania nie je generovanie syntakticky korektných programov, ale sémanticky korektných. Preto Fuzzilli zavádza svoj vlastný medzijazyk FuzzIL, ktorý si predstavíme v nasledujúcej podkapitole.

2.1 FuzzIL

Mutovať JavaScriptový kód ako taký je veľmi ťažké a väčšina známych nástrojov mutuje priamo odvodzovací strom korešpondujúcej gramatiky. S týmto prístupom len veľmi ťažko zachováme sémantickú korektnosť. Preto Fuzzilli prichádza s iným prístupom. Zavádza medzijazyk FuzzIL na ktorom sa tieto mutácie budú robiť oveľa ľahšie a bude možné minimalizovať prípady, keď mutáciou stratíme sémantickú korektnosť. Tento jazyk má oveľa bližšie k strojovému kódu a omnoho viac pripomína samotnú grafovú reprezentáciu kódu. Pomocou tohto jazyka sa nám budú ľahšie zisťovať vlastnosti

generovaných skriptov a na základe toho ladiť mutácie a celé fuzz testovanie. Taktiež tento jazyk bol navrhnutý tak, aby sme mali zaistené, že ak vygenerujeme ľubovoľný korektný FuzzIL kód, budeme ho vedieť preložiť do JavaScriptu a následne spustiť.

FuzzIL program je vlastne zoznam inštrukcií. Každá z inštrukcií pozostáva z operácie, zoznamu vstupných premenných, zoznamu vnútorne výstupných premenných a zoznam výstupných premenných.
Príklad ako taký program môže vyzerať je nasledovný. Na každom riadku sa nachádza presne 1 inštrukcia.

Program 2.1: Ukážka Fuzzilli kódu

```
1  v0 <- LoadString 'bar'
2  v1 <- CreateArray [v0, v0, v0]
3  BeginIf v0
4    v2 <- UnaryOperation v0 '++'
5    BeginDoWhile v2, '==', v1
6      v3 <- LoadBuiltin 'Symbol'
7      v4 <- LoadProperty v3, 'match'
8      StoreComputedProperty v1, v4, v0
9    EndDoWhile
10 BeginElse
11 EndIf
12 v3 <- BeginAsyncFunctionDefinition -> v4, v5
13   v6 <- Await v4
14   Return v5
15 EndAsyncFunctionDefinition
16 v7 <- CallFunction v3, [v1, v2]
17 Reassign v0 v7
```

Na prvom riadku vidíme príklad operácie, ktorá načíta reťazec do premennej. Môžeme si všimnúť, že operácia má 1 výstup a žiadne vstupy. Napriek tomu vieme tejto operácii dať parameter (v tomto prípade reťazec „bar“) a tak vytvoriť špecifickejšiu operáciu. Vstupmi a výstupmi totiž môžu byť len iné premenné. Takýmito parametrami môžeme vytvárať rôznu sadu operácií.

Okrem klasických operácií, ktoré sme videli na riadku 1, tu máme aj ďalšiu skupinu operácií, nazveme ich *blokové* operácie. Príklad takejto operácie môžeme vidieť napríklad na riadku 3. Blokové ich nazývame preto, lebo ohraničujú blok kódu a teda táto operácia nutne má aj svoju pridruženú inštrukciu, ktorá ohraničuje koniec tohto bloku. Nemusí to však nutne byť len 1 blok. Ako vidíme na riadkoch 3, 10 a 11 máme tu ohraničené 2 bloky konštruktom *if-else*. Tak ako to v bežných programovacích jazykoch býva, tak premenné vytvorené vnútri bloku nie sú viditeľné vonkajším blokom,

ale naopak premenné vytvorené vo vonkajšom bloku sú viditeľné vo vnútornom bloku. Keď sa pozrieme bližšie na operáciu *BeginIf* vidíme, že nemá žiadny výstup, no má jeden vstup, podľa ktorého vyhodnotíme, ktorý blok sa má vykonávať.

Ďalšia zaujímavá bloková operácia sa nachádza na riadku 12. Toto je definícia asynchrónnej funkcie a tak ako ostatné blokové operácie má aj svoju pridruženú inštrukciu, ktorá končí definíciu tejto funkcie a to na riadku 15. Táto operácia ale má aj *vnútorne výstupné* premenné, konkrétne *v4* a *v5*. To znamená, že inštrukcia tieto premenné vytvorí rovnako ako výstupné premenné, no viditeľné budú len vo vnútornom bloku, ktorý táto inštrukcia začína. Teda *v4* a *v5* budú viditeľné len vnútri definície funkcie. Naopak premenná *v3*, ktorá je výstupná je viditeľná aj pre vonkajší blok. V našom prípade teda samotná funkcia je výstupná premenná a argumenty funkcie sú naopak vnútorne výstupné premenné.

Fuzzer Fuzzilli však negeneruje ľubovoľné programy. Na to aby sme vedeli ľahšie uvažovať o vlastnostiach vygenerovaných programov, poprípade o mutáciách, či vygenerovaných častiach programu, generujeme len programy v nami definovanom normálnom tvare. Ako sme videli aj v ukážke 2.1, premenné pomenovávame v poradí v akom sú vytvorené v inštrukciách a to vždy v tvare vX, kde X je poradové číslo našej premennej. Taktiež vygenerovaný kód vždy priradí hodnotu premennej presne raz (angl. *SSA form*). Na priradenie inej hodnoty premennej môžeme použiť inštrukciu *Reassign* ako na riadku 17. A ako sme už spomínali, všetky vstupné, vnútorne výstupné a výstupné parametre inštrukcie musia byť premenné, žiadne medzivýrazy nie sú povolené.

Preklad do samotného JavaScriptu by mal byť pomerne priamočiary, preklad kódu 2.1 môžeme vidieť v 2.2.

Program 2.2: Ukážka prekladu Fuzzilli kódu 2.1 do JavaScriptu

```
1  let v0 = 'bar'
2  const v1 = [v0, v0, v0]
3  if (v0) {
4    const v2 = v0++
5    do {
6      const v3 = Symbol
7      const v4 = v3.match
8      v1[v4] = v0
9    } while(v2 == v1)
10 } else {
11 }
12 async function v3(v4, v5) {
13   const v6 = await v4
```

```
14   return v5
15 }
16 const v7 = v4(v1, v2)
17 v0 = v7
```

Samozrejme toto nie je jediný možný preklad. Fuzzilli nepodporuje žiadne medzivýrazy v inštrukciách, no v samotnom JavaScripte sa nám to môže hodiť. Najmä tým skrátime dĺžku samotných programov. Toto dosiahneme takzvaným spájaním výrazov (angl. *inlining*). Teda napríklad namiesto prekladu, ktorý vygeneroval riadky 6 a 7, môžeme vygenerovať len jeden riadok a to:

```
1 const v4 = Symbol.match
```

Pri spúšťaní samotného Fuzzera sa dá nastaviť úmernosť používania tohto spájania, aby sme dokázali generovať kratší JavaScript kód, keďže ten zdieľame s JavaScript enginom pomocou obmedzenej zdieľanej pamäte.

Prekladač sa takisto musí vysporiadať s tým ako má premennú zadefinovať, či pomocou *let* alebo *const*. To ale vie jednoducho zistiť pomocou analýzy inštrukcií *Reassign* v programe.

Úplný zoznam aktuálne podporovaných operácií je možné nájsť v prílohe.[4]

2.2 Mutovanie FuzzIL programu

Ako sme už skôr naznačili, Fuzzilli sa bude snažiť hľadať (v iteráciách) zaujímavé vstupy pre JavaScript enginy, ktoré potencionálne môžu nájsť chybu. Tieto zaujímavé vstupy hľadáme podobne ako pravdepodobnostné optimalizačné algoritmy (simulované žíhanie, ...).

Udržiavame korpus zaujímavých FuzzIL programov, ktoré v iteráciách mutujeme rôznymi spôsobmi. Podľa metriky si tieto zmutované programy buď necháme alebo zahodíme. Tieto mutácie musia zachovávať syntaktickú správnosť programu a zároveň sa budeme snažiť nerobiť príliš veľké zásahy do programu aby sme minimalizovali šancu poškodenia sémantickej správnosti. Fuzzilli má aktuálne naimplementované nasledovné spôsoby mutovania programu:[5]

- **CodeGenMutator** – Tento prístup si vyberie náhodné miesto v programe a snaží sa vygenerovať niekoľko FuzzIL inštrukcií. Na toto používa niekoľko generátorov, z ktorých si náhodne vyberá. Tieto generátory na vstupe očakávajú niekoľko premenných a na výstupe vrátia niekoľko inštrukcií. Každý generátor je naimple-

[4] /fuzzilli/Sources/Fuzzilli/FuzzIL/Operations.swift
[5] /fuzzilli/Sources/Fuzzilli/Mutators

mentovaný na generovanie konkrétneho konštruktu, teda napríklad **ObjectGenerator** generuje náhodný objekt, **ForLoopGenerator** vygeneruje for cyklus a podobne. Zoznam všetkých generátorov možno nájsť v prílohe.[6]

- **CombineMutator** – Tento prístup len jednoducho vyberie ďalší program z korpusu a vloží ho na niektoré miesto v pôvodnom programe. Samozrejme popritom musí premenovať mená premenných aby program ostal v normálnej forme.
- **ConcatMutator** – Podobne ako predchádzajúci prístup, tentokrát len zlepí 2 kusy programov za sebou.
- **InputMutator** – Vyberie si náhodnú inštrukciu a snaží sa zameniť vstupné premenné tejto inštrukcie.
- **JITStressMutator** – V tomto prístupe chceme vygenerovať taký kód, čo volá funkciu, ktorá už bola skompilovaná JIT kompilátorom s odlišnými argumentami alebo v odlišnom prostredí. Toto dosiahneme tak, že na koniec pôvodného programu dogenerujeme niekoľko inštrukcií, ktoré možno zmenia prostredie alebo budeme mať aspoň viac možností na výber argumentov. A na úplný záver vyberieme náhodnú funkciu, ktorú zavoláme s odlišnými argumentami (zaručiť, že funkcia už bola skompilovaná nevieme, no aspoň skúsime vytvoriť skript kde je šanca výrazná).
- **OperationMutator** – V tomto prístupe si tiež vyberieme náhodnú inštrukciu, ale teraz nebudeme meniť vstupné premenné. Budeme meniť samotnú operáciu alebo len jej parametre.

2.3 Typový systém medzijazyka FuzzIL

Všimnime si, že ku premenným podobne ako v JavaScripte nemáme priradené žiadne typy. A práve preto nám môžu vznikať sémantické chyby končiace chybou počas behu programu. Keď **InputMutator** zmutuje vstupné premenné operácie bez znalosti typu zamieňanej premennej, môže sa stať, že program nebude ďalej korektný. Podobne pri generovaní kódu, ak nemáme vedomosti o možných anotáciách funkcií, vkladáme do nich náhodné premenné.

Pokiaľ by sme chceli úplne presné typy, museli by sme prakticky reimplementovať JavaScriptový engine a jeho pravidlá. Nejakú typovú informáciu ale chceme, ideálne bez ťažkostí s implementovaním a časovej náročnosti, ktorú by takýto nástroj pridal pri generovaní programov. V tejto podkapitole si popíšeme typový systém fuzzera v jeho podobe predtým ako sme ho upravili v našej práci.

[6] /fuzzilli/Sources/Fuzzilli/Core/CodeGenerators.swift

Fuzzilli pozná niekoľko základných typov premenných a to **undefined, integer, bigint, float, string, boolean, regexp**. Následne pozná aj zložitejšie typy a to **object**, ktorý tiež v sebe obsahuje zoznam vlastností a metód, ktoré sa v objekte nachádzajú. Pre jednoduchosť však nemá žiadnu vedomosť o prototypoch a teda môže obsahovať metódy z ľubovoľnej časti prototypovej reťaze. Ďalej máme typy **function, constructor**, ktoré obsahujú svoju anotáciu a teda typy vstupných argumentov a svojho výstupu. Na záver samozrejme pomocný typ **unknown**, ktorý hovorí, že typ nepoznáme.

Jednotlivé typy vieme jednoducho kombinovať, pokiaľ si nie sme istý aký presný typ to je. Môžeme mať aspoň čiastočnú informáciu, že je to jeden z možných typov. Napríklad, že je to reťazec alebo číslo (vieme aspoň že to nebude objekt a podobne). Operáciu, kde určíme, že premenná je jeden z 2 typov nazveme *zjednotenie* a jej výsledkom je nový typ (ktorý môžeme znova zjednotiť s iným, ...).

Naopak občas sa stane, že premenná sú vlastne 2 typy naraz. Napríklad reťazec je jednak reťazec a zároveň aj objekt, keďže na ňom môžeme používať rôzne metódy. Preto zadefinujeme nad typmi ďalšiu metódu, ktorú nazveme *zlúčenie*.
Zlúčenie dostane 2 typy a vytvorí nový typ, ktorý reprezentuje, že daná premenná má vlastnosti oboch typov zároveň. Napríklad bežné JavaScript funkcie majú typ, ktorý vznikol zlúčením funkcie aj konštruktora (vieme ich zavolať s výrazom *new*). Neplatí to však pre všetky funkcie. Šípkové funkcie (angl. *arrow functions*) nie je možné používať ako konštruktory.
Samotnú implementáciu typov môžeme nájsť v prílohe[7], kde nájdeme aj príklady vlastností a použitia tohto systému[8].

Na zisťovanie samotných typov je tu naprogramovaný veľmi jednoduchý nástroj s názvom Abstraktný interpreter[9]. Tento Abstraktný interpreter funguje veľmi jednoducho. Ako postupne generujeme program pamätá si aktuálne typy každej premennej. Ak príde inštrukcia z ktorej je jasné, aký typ bude mať jej výstup tak si ho zapamätáme. Teda napríklad **LoadString** určite vytvorí reťazec a podobne.

Pri operáciách ako napríklad sčítavanie je to zložitejšie. Ak poznáme typy vstupov mali by sme teoreticky vedieť aj výstupný typ (presne tak ako to robí engine).
Problém je však v tom, že odvodzovanie chceme spúšťať veľmi často a zložitá logika by príliš spomaľovala tento mechanizmus.
Druhý a väčší problém je, že tento mechanizmus je veľmi zložitý a preprogramovať celú logiku z JavaScriptového enginu by bolo náchylné na chyby[10] a testovať engine softvérom, ktorý naimplementoval jeho (zložité) časti odznova nedáva zmysel. Abstraktný

[7] /fuzzilli/Sources/Fuzzilli/FuzzIL/TypeSystem.swift
[8] /fuzzilli/Tests/FuzzilliTests/TypeSystemTest.swift
[9] v prílohe na /fuzzilli/Sources/Fuzzilli/FuzzIL/AbstractInterpreter.swift
[10] napríklad aký je výsledný typ *{} + []* ?

interpreter teda pre sčítavanie síce nevie presný typ ale vie, že to bude jednoduchý typ (teda nie objekt), tak si zapamätá aspoň to.

Navyše všetky premenné a funkcie z JavaScriptového prostredia máme staticky otypované[11]. Či už prototypy objektov, polí, ... alebo aj vstavané objekty ako *Math, JSON*. Abstraktný interpreter tieto statické typy berie pri odvodzovaní do úvahy.

Program 2.3: Ukážka nesprávneho odvodenia typu

```
1 Math.pow = () => "foo"
2 const v0 = Math.pow(1, 2)
3 // v0 = .number
```

Abstraktný interpreter ale nevie nič o samotnom vykonávaní a teda môže dokonca odvodiť typ nesprávne ako v ukážke 2.3, keďže si neaktualizuje typy vstavaných funkcií.

Program 2.4: Ukážka odvodzovania typov vo funkciách

```
1 const v0 = 1
2 // v0 = .integer
3 const v1 = () => {
4   v0 = "foo"
5   // v0 = .string
6 }
7 // v0 = .integer | .string
```

Iný príklad problému pôvodnej implementácie abstraktného interpretera vidíme v ukážke 2.4. Abstraktný interpreter nevie rozoznať, či sa niečo vykonáva vo funkcii a kedy sa naozaj funkcia zavolá. Preto definíciu funkcie považuje za podmienený blok. Priradí typy premenným priamo v definícii funkcie a po konci definície urobí zjednotenie typov (ako na riadku 7). Tento prístup síce generuje správne typy, ale získané typy môžu byť často príliš konzervatívne a všeobecné.

Tento jednoduchý mechanizmus, akokoľvek hlúpy sa môže zdať, pomáha aspoň mierne. Vieme detegovať veľmi základné typy v základných situáciách a tak sa vyhnúť nadbytočným sémantickým chybám v programoch. V ďalšej kapitole navrhneme ako celý tento mechanizmus zlepšiť.

[11] v prílohe na /fuzzilli/Sources/Fuzzilli/Core/JavaScriptEnvironment.swift

Kapitola 3

Zisťovanie typovej informácie z viacerých zdrojov

Cieľom typového systému fuzzera Fuzzilli, je čo najjednoduchšie a zároveň čo najpresnejšie popísať typy JavaScriptových premenných. Fuzzilli tieto informácie využíva na odpovedanie 2 typov dotazov:

- **Nájdi premennú s typom X** – keď generujeme kúsky JavaScriptového kódu často potrebujeme použiť premennú s konkrétnym typom. Napríklad keď chceme vygenerovať cyklus potrebujeme premennú cez ktorú sa dá iterovať. Alebo keď chceme vygenerovať volanie funkcie musíme najprv mať premennú obsahujúcu definíciu funkcie.
- **Zisti možné použitia premennej X** – chceme vygenerovať inštrukciu používajúcu konkrétnu premennú alebo pri mutácii inštrukcie chceme vedieť či inštrukcia ostane korektná aj po zamenení premenných.

3.1 Abstraktný interpreter

Aktuálny zdroj typových informácií je Abstraktný interpreter. Okrem mnohých výhod, však prináša aj mnoho obmedzení a vlastností, ktoré môžeme v niektorých situáciách považovať za výhody a v iných za nevýhody, preto si ich zhrňme:

- **Jednoduchá implementácia** – Implementácia má len niekoľko stoviek riadkov a tak je ľahko udržiavateľná.
- **Rýchle odvodzovanie** – Nielen implementácia, ale aj logika je veľmi jednoduchá s jednoduchými sémantickými pravidlami a netreba spúšťať samotný JavaScriptový engine. Takže si môžeme dovoliť pustiť každý vygenerovaný program cez Abstraktný interpreter bez spomalenia celého fuzz testovania.

- **Aproximácia JavaScriptovej sémantiky** – Samotná JavaScript sémantika je veľmi zložitá. Za cenu rýchlosti a jednoduchosti teda nepokrýva všetky prípady a snaží sa zachytiť len tie najbežnejšie.

- **Nepresné typy** – Keďže sémantiku len aproximuje, môže sa stať že odvodená typová informácia je nesprávna alebo nie je vôbec dostupná.

Pre lepšiu predstavu uvažujme nasledovný kus kódu (pod každý riadok pridáme komentár s typom odvodeným Abstraktným interpreterom):

Program 3.1: Abstraktný interpreter nie je schopný odvodiť typ

```
1 const v0 = [new Set(), 1, "foo"]
2 // .object(ofGroup: "Array", ...)
3 const v1 = v0[0]
4 // .unknown
5 v1.???()
```

Na riadku *3* totiž Abstraktný interpreter nevie z typovej informácie *v0* zistiť typy jednotlivých prvkov. Potom ale na riadku *5* nevieme vygenerovať použitie žiadnej zaujímavej metódy.

3.2 Zbieranie typov počas behu programu

Preto prichádzame s návrhom iného zdroja typových informácií. Postupne ho v práci predstavíme a porovnáme rôzne možnosti integrovania tohto zdroja priamo do Fuzzilli. Porovnáme slabiny a výhody týchto integrácií a následne sa ich pokúsime skombinovať aby sme maximalizovali výkonnosť a úspešnosť samotného fuzzera.
V prílohe je možné nájsť samotnú implementáciu finálneho riešenia. Pre lepšiu flexibilitu používateľa Fuzzilli, je možné naše vylepšenie zapnúť/vypnúť pomocou argumentu príkazového riadku *--collectRuntimeTypes*.

Základná myšlienka je, že namiesto statického odvodzovania typov počas generovania programu, môžeme typy zbierať počas jeho vykonávania v JavaScriptovom engine.

Bez integračných a implementačných detailov si zhrňme výhody a nevýhody takéhoto prístupu:

- **Zložitejšia implementácia** – Na zbieranie týchto typov potrebujeme zmeniť samotný vygenerovaný kód, no musíme ho upraviť tak, aby sa nič v jeho vykonávaní nezmenilo a teda aby táto informácia mala zmysel. Takisto musíme aj po vykonaní programu vedieť odpovedať na otázku:
  „Aký typ mala premenná vX v čase T?“

- **Pomalé zbieranie** – Zbieranie týchto typov bude pomalé, keďže musíme spúšťať samotný program v JavaScript engine a ešte pomedzi to pridať samotný kód, ktorý tieto typy pozbiera. Na pozbieranie vlastností a metód objektu potrebujeme prejsť celú reťaz prototypov, čo ešte viac spomalí celý proces.
- **Presná JavaScript sémantika** – Žiadna aproximácia, ale vykonanie samotným JavaScript enginom
- **Presné typy** – Keďže priamo vykonávame JavaScript, dostaneme aj úplne presnú typovú informáciu

Všimnime si, že takýmto prístupom opravíme typovú informáciu v kóde 3.1

Program 3.2: Opravené zbieranie typovej informácie

```
1 const v0 = [new Set(), 1, "foo"]
2 // .object(ofGroup: "Array", ...)
3 const v1 = v0[0]
4 // .object(ofGroup: "Set", ...)
5 v1.add(47)
```

Na zbieranie typov musíme naprogramovať nový mód pre prekladač z medzijazyka Fuzzilli do JavaScriptu. Pokiaľ chceme preložiť náš vygenerovaný program na JavaScript, ktorý aj pozbiera typy, musíme samotný *Lifter*[1] spustiť s nami naimplementovaným variantom *collectTypes*.

Tento variant spôsobí, že na začiatok skriptu vygeneruje pomocné funkcie a inicializuje štruktúry na zbieranie typov. Následne za každú preloženú inštrukciu z pôvodného skriptu vloží volanie funkcie *updateType*. Aby sme naozaj boli schopný vložiť toto volanie za každú FuzzIL inštrukciu, musíme pri tomto preklade vypnúť funkcionalitu spájania výrazov. Inak stratíme typy niektorých FuzzIL premenných (ktoré v JavaScript kóde nebudú existovať). Táto funkcia zistí typ premenných zmenených poslednou inštrukciou a uloží si túto informáciu do predpripravenej štruktúry. Až napokon na konci celého prekladania vypíšeme celú štruktúru na špeciálny komunikačný kanál, odkiaľ si ju následne Fuzzilli prečíta a ďalej spracuje.

Ak by sme teda mali bežne preložený kód 3.2, tak prekladač s variantom *collectTypes* by ho preložil nasledovne:

Program 3.3: Preklad do JavaScriptu s variantom *collectTypes*

```
1 initTypeCollection()
2 const v0 = [new Set(), 1, "foo"]
3 updateType(0, v0)
```

[1] v prílohe na /fuzzilli/Sources/Fuzzilli/Lifting/JavaScriptLifter.swift

```
4 const v1 = v0[0]
5 updateType(1, v1)
6 v1.add(47)
7 updateType(1, v1)
8 sendTypesToFuzzilli()
```

Argumenty funkcie *updateType* by teda boli číslo premennej a jej hodnota.

Ako sme už spomínali, takéto zbieranie typov je časovo veľmi náročné. Preto si nemôžeme dovoliť zbierať typy pre každý vygenerovaný program. Všimnime si, že samotnú typovú informáciu využívame pri generovaní a mutovaní iného kódu. To znamená, že táto informácia je najužitočnejšia pre programy v korpuse. Samozrejme pri generovaní nového kódu môžeme použiť viac mutácií alebo najprv prigenerovať časť skriptu, ktorú následne zmutujeme a teda v týchto situáciách nebudeme mať typovú informáciu. Toto sa nebude stávať často, keďže sú to veľmi špecifické prípady ak uvažujeme mutácie na náhodných miestach.

Skriptov, ktoré pridajú nové hranové pokrytie (a teda budú pridané do korpusu) je mnohonásobne menej ako všetkých vygenerovaných skriptov. Pre 24 hodinové fuzz testovanie býva počet skriptov pridávajúcich nové hranové pokrytie na úrovni **0,1%** zo všetkých vygenerovaných skriptov (samozrejme na začiatku je toto číslo väčšie a postupom času sa zmenšuje). Zbieranie typov v čase behu programu budeme teda spúšťať len na programoch tesne pred pridaním do korpusu.

Ďalšie jednoduché vylepšenie, ktoré môžeme spraviť je, že pri mutáciách sa nám veľká časť skriptu nezmení. Preto nemusíme hneď „zahodiť“ všetky typové informácie skriptu po jednej mutácii, ale snažiť sa ponechať si čo najviac typovej informácie. Toto zas nevieme urobiť so 100% istotou, keďže v JavaScripte jedna inštrukcia môže ovplyvniť typ mnoho premenných v iných častiach skriptu (napríklad zmena prototypu). Tieto prípady ale nie sú až také časté a teda ponechanie si typovej informácie pre nezmenené inštrukcie bude lepšie ako byť príliš konzervatívny a nemať typy žiadnej premennej. Pri mutáciách zahodíme typové informácie získané na základe pôvodnej inštrukcie, ale typy premenných, ktoré sme získali na základe nezmenených inštrukcií si necháme.

Program 3.4: Typy pred mutáciou

```
1 const v0 = 42
2 // v0 = .integer
3 const v1 = v0 * null
4 // v1 = .integer
```

Program 3.5: Typy po mutácii

```
1 const v0 = 42
2 // v0 = .integer
3 const v1 = v0 / null
4 // v1 = .unknown
```

V úryvku kódu 3.5 vidíme, že mutácia sa stala len na riadku *3*, kde sme zmenili

operáciu násobenia na operáciu delenia. Takže typ premennej *v0* na riadku *1* sme si ponechali. Ale typ premennej *v1* sme zahodili, keďže je časté, že sa mutáciou zmení. V našom prípade sa typ naozaj zmení, keďže *v1=Infinity*, čo je v našom typovom systéme *float*.

3.2.1 Implementácia zbierania typov

Doteraz sme len všeobecne predstavili funkcie *initTypeCollection, updateType* a *sendTypesToFuzzilli*. Poďme sa teda pozrieť ako konkrétne naimplementujeme prvú verziu a aké (ne)výhody z toho vyplynú.

Keďže naše zbieranie typov bude bežať priamo v JavaScript engine, tak bude bežať oddelene od samotného fuzzera, ktorý následne len zozbiera dáta z určených komunikačných kanálov. Zozbierané typové dáta musí Fuzzilli následne preložiť do svojho typového systému a uložiť do programu pre ďalšie použitie. Našťastie samotný fuzzer je naprogramovaný tak, aby vedel bežať na viacerých strojoch[2], ktoré si medzi sebou posielajú programy vo formáte *Protobuf*. Takže stačí, aby náš skript vypísal zistené typy v tomto formáte a fuzzer to už jednoducho deserializuje. Výhoda tohto prístupu je kompaktnosť výstupu, ktorá je stále dôležitá pretože musíme mať na pamäti, že fuzzer komunikuje s enginom pomocou zdieľanej pamäte.

Problém ale je, že JavaScript nemá vstavanú podporu pre serializovanie do tohto formátu a tak by sme do skriptu museli pridať aj mechanizmus na prekladanie JavaScript objektov do Protobuf formátu. Čím ale výrazne zväčšíme veľkosť skriptu, ktorý odovzdávame enginu tiež pomocou zdieľanej pamäte. Preto lepšou možnosťou bude *JSON* formát. Je síce menej kompaktný, ale JavaScript ma vstavané funkcie, ktoré vedia serializovať objekty do tohto formátu. Navyše knižnica na deserializáciu objektov v Protobuf formáte, ktorú fuzzer používa, vie na základe definície správy v Protobuf formáte deserializovať aj JSON. Každopádne sa nevyhneme reimplementácii (aspoň nutnej časti) typového systému Fuzzilli v JavaScripte.

Typ v typovom systéme Fuzzilli je reprezentovaný štruktúrou obsahujúcou vlastnosti:

- **definiteType** – vyjadruje, ktorých typov premenná **určite** je, uložený ho máme ako bitovú masku, kde každý základný typ zodpovedá jednému bitu. Zvyčajne len 1 bit bude nenulový, no napríklad reťazce môžeme považovať aj za reťazce aj za objekty.

[2] v prílohe na /fuzzilli/Cloud/GCE/README.md

- **possibleType** – podobne ako *definiteType*, je to bitová maska základných typov. No tentokrát vyjadruje všetky typy, ktoré premenná **môže** byť. Využiť to môžeme najmä vtedy, keď nevieme typ presne odvodiť no stále chceme mať nejakú informáciu.

- **ext** – môže obsahovať viac detailov typu a to zoznam vlastností a metód (pri objektoch), anotácia funkcie (typy argumentov a typ návratovej hodnoty) a skupinu, ktorá je používaná ako podtyp objektu (pole, množina, ...).

Teda zistenie typu spočíva v zistení týchto 3 hodnôt.

Keďže typ premennej sa môže počas programu meniť, musíme reimplementovať aj zlúčenie typov. Typy počas behu programu budeme ukladať do JavaScript objektu s kľúčom čísla premennej a hodnotou jej typu.

Program 3.6: Prvý návrh implementácie funkcie updateType

```
function updateType(number, value) {
    var currentType = getCurrentType(value)
    if (types[number] == null) types[number] = currentType
    else types[number] = types[number].union(currentType)
}
```

Samotné zisťovanie typu vo funkcii *getCurrentType* bude zisťovanie typu najmä pomocou *typeof*.

Program 3.7: Funkcia getCurrentType

```
function getCurrentType(value){
    try {
        if (value == null) return new Type(baseTypes.undefined)
        try {
            if (Number.isInteger(value)) return new Type(baseTypes.integer)
        } catch(err) {}
        if (typeof value === 'number') return new Type(baseTypes.float)
        ...
    } catch {}
    return new Type(baseTypes.unknown)
}
```

Musíme všetko zaobaliť do try-catch blokov, keďže ak beh programu skončí chybou počas behu programu (sémantická chyba), fuzzer nebude schopný prečítať žiadne zistené typy. Naopak ak pri chybe len nastavíme typ na *unknown* a pokračujeme v zisťovaní typov ostatných premenných, tak budeme mať aspoň nejaké typy. Musíme si totiž uvedomiť, že vygenerovaný skript môže robiť prakticky čokoľvek a chyby môžu

vznikať aj pri skripte, ktorý na prvý pohľad nemá dôvod byť nekorektný. Uvažujme, že vygenerovaný skript obsahoval riadok:

$$\text{Number.isInteger} = 42$$

Potom aj na riadku 5 v kóde 3.7 nastane chyba.

To čo nám ostáva vedieť určiť je *ext*. Vlastnosti a metódy objektov zistíme jednoduchým prechodom po prototypovej reťazi ako v 3.8.

Program 3.8: Zbieranie vlastností a metód

```
1  Type.prototype.collectProps = function(obj) {
2      this.extension.methods = []
3      this.extension.properties = []
4      while (obj != null) {
5          var propertyNames = Object.getOwnPropertyNames(obj)
6          for (var i=0;i<propertyNames.length;i++) {
7              var name = propertyNames[i]
8              try {
9                  if (typeof obj[name] === 'function') {
10                     this.extension.methods.push(name)
11                     continue
12                 }
13             } catch (err) { continue }
14
15             this.extension.properties.push(name)
16         }
17         obj = obj.__proto__
18     }
19 }
```

Naopak zbierať signatúry funkcií počas behu programu nemá veľkú výpovednú hodnotu. Vo všeobecnosti totiž typy argumentov a návratového typu sú počas behu programu špecifické pre konkrétny skript a nám pri mutáciách nepomôžu.

Na zistenie skupiny si pre každú z nich naprogramujeme vlastnú funkciu, ktorá vyhodnotí, či premenná do nej patrí.

Tak ako sme si však ukázali, samotný vygenerovaný skript môže meniť aj vstavané funkcie a ak tie chceme používať musíme si ich na začiatku skriptu skopírovať do iných premenných, ku ktorým vygenerovaný skript nebude mať prístup. No kopírovanie celého Javascript prostredia (prototyp objektu, poľa, ...) nie je možné, keďže by to výrazne zväčšilo veľkosť skriptu a tiež spomalilo jeho vykonávanie. Preto musíme pristúpiť na kompromis a odložiť si len najčastejšie používané funkcie.

Presnú implementáciu je možné nájsť v prílohe[3].

Na záver už len vypíšeme našu štruktúru s typmi serializovanú vo formáte JSON[4].

3.2.2 Limitácie

Napriek tomu ako všetko vyzerá na prvý pohľad jednoducho, tak tomu tak nie je. Zbieranie typov sa môže skončiť chybou počas behu programu alebo sa zacykliť napriek tomu, že pôvodný vygenerovaný program je korektný (je zaručené že bol vykonaný korektne, keďže ho pridávame do korpusu).

Tiež sa môže stať, že to náš skript spomalí natoľko, že sa neoplatí čakať na jeho skončenie. Radšej môžeme pokračovať bez typov. Preto, tak ako pri obyčajnom fuzz testovaní si stanovíme časový limit dokedy čakáme na výstup a takto sa vyhnúť pomalým alebo aj zacykleným programom. Ukážme si niekoľko príkladov problémových skriptov, ktoré mohli byť vygenerované:

Program 3.9: Pomalé zbieranie lenivo inicializovaných premenných

```
1 const v0 = new UInt8Array(1000000)
2 updateType(0, v0)
```

V skripte 3.9 vidíme na prvom riadku inicializáciu dlhého poľa. Táto operácia je však urobená enginom lenivo a teda je rýchla. No keď sa snažíme zistiť jej typ a pozbierať všetky jej vlastnosti a metódy trvá nám to veľmi dlho, keďže musíme naozaj prejsť celé pole. Dokonca nám to ani nepomôže, keďže fuzzer vie využívať len vlastnosti ku ktorým vieme pristupovať len bodkovou notáciou (angl. *dot notation*). A to prvky poľa nie sú. Preto musíme zaviesť do zbierania vlastností a metód obmedzenia.

Najprv obmedzíme počet všetkých zozbieraných vlastností. Ak tento počet dosiahneme už ďalšie nezbierame, ušetríme tým čas a zároveň máme zozbieraných dosť vlastností aby fuzzer vedel vygenerovať niečo zaujímavé. Takisto začneme preskakovať vlastnosti ku ktorým nevieme pristúpiť bodkovou notáciou na ušetrenie pamäte. Toto ale stále nevyrieši rýchlosť nášho skriptu 3.9.

Totiž funkcia *Object.getOwnPropertyNames* musí skonštruovať interne pole so všetkými vlastnosťami. JavaScript totiž neposkytuje API na prechádzanie vlastnosťami objektu pomocou iterátora. Napriek tomu sme si mierne pomohli, keďže teraz konštruujeme pole len z vlastných vlastností, ak ich je dosť nepokračujeme ďalej po prototypovej reťazi.

Predtým ako zisťujeme vlastnosti objektu, vieme zistiť skupinu do ktorej patrí. Môžeme si určiť, pri ktorých skupinách skontrolujeme dĺžku premennej (na to netreba konštruovať celé pole vlastností). Ak bude príliš veľká, preskočíme zbieranie vlastností

[3] /fuzzilli/Sources/JS/helpers.js
[4] /fuzzilli/Sources/JS/printTypes.js

na samotnom objekte a začneme zbierať vlastnosti priamo na prototype. Týmto prístupom by sme nemali stratiť veľa vlastností, keďže ku prvkom poľa nevieme pristupovať bodkovou notáciou.

Program 3.10: Typovanie vstavaných premenných

```
1 Math.max = 42
```

V skripte 3.10 vidíme, že môžeme vygenerovať aj riadok, kde zmeníme typ vstavaných objektov. My však zbierame len typy premenných vytvorených skriptom. Preto sa nám môže stať, že fuzzer nemá správny typ vstavaných objektov, ktoré sú staticky otypované. Tento problém však nastával len zriedka a preto sme sa ním nezaoberali. Pre fuzzer je občas dobré mať aj zlý typ a vyskúšať niečo náhodné (typy sú často veľmi konzervatívne) aj keď je veľká šanca, že to zle dopadne. Podobne ako pri pravdepodobnostných optimalizačných algoritmoch (napríklad simulované žíhanie).

Program 3.11: Bežné použitie eval

```
1 const v0 = "const v1 = 42"
2 const v2 = eval(v0)
```

Program 3.12: Nepriame použitie eval

```
1 const v0 = eval
2 const v1 = "const v2 = 42"
3 const v3 = v0(v1)
```

Pri ukážkach kódu 3.11 a 3.12 sa pozrieme na spájanie výrazov (angl. *inlining*). Vieme že FuzzIL nemôže mať v inštrukciách žiadne zložené výrazy. Samotnému prekladaču do JS však vieme nastaviť ako veľmi má byť agresívny pri spájaní výrazov. Spájanie výrazov nám však robí problémy pri zbieraní typov. Pretože v jednom riadku skriptu sa môže vykonať viac FuzzIL inštrukcií a na ďalšom chceme zavolať funkciu na pozbieranie typov. V tomto momente už nemôžeme typy správne napárovať späť na FuzzIL inštrukcie, keďže nevieme ktorý typ patril ku ktorej zo spojených inštrukcií. Preto pri zbieraní typov spájanie prekladaču zakážeme.

Existujú však konštrukty, kde sa program správa rôzne keď spojíme výrazy rôzne. Ako príklad si môžeme pozrieť kód 3.11, tu sme výrazy spojili a takto fuzzer bežne spúšťa skript. Funkcia *eval* sa v tomto prípade spustí s aktuálnym kontextom (angl. *scope*).

Keď však zbierame typy, spustíme kód 3.12 a tu najprv *eval* uložíme v premennej. V takomto prípade *eval* spustí kód v globálnom kontexte[5] a náš zber môže byť neúspešný. Napríklad tým, že funkcie volané vnútri *eval* v jednom kontexte existujú a v druhom nie. Tento problém vyriešime aspoň čiastočne vylepšením v podkapitole 3.3.

[5] https://262.ecma-international.org/5.1/#sec-10.4.2

Program 3.13: Zbieranie vlastností veľkých objektov

```
1 ...
2 for(var v1=0;v1<1000000;v1++){
3     v0.a = 3
4     updateType(0,v0)
5 }
```

Predstavme si, že premenná *v0* bola definovaná ako veľký objekt, predtým ako sa začal vykonávať úryvok kódu 3.13. Potom na riadku 3 máme jednoduchú operáciu, kde zistiť typ *v0* bude pomalé. Vždy totiž zbierame typ odznova a prechádzame celú prototypovú reťaz. Navyše sa táto inštrukcia vyskytuje vo veľmi dlhom cykle, čo ešte zvýrazní spomalenie zbierania typov oproti pôvodne vygenerovanému skriptu (typy zbierame pred vložením do korpusu, teda pôvodný skript sa zmestil do časového limitu).

Tomuto budeme vedieť predchádzať keď vymyslíme v podkapitole 3.3 ako efektívne odvodiť typ pri jednoduchej inštrukcii ako je pridanie vlastnosti objektu. Problém nám tu však ostane aj keď zbieranie nebude výrazne pomalšie. Veľmi dlhý cyklus nám zvýrazní aj malé spomalenie zbierania typov a občas sa stane, že sa už nezmestíme do časového limitu.

Program 3.14: Zmena prototypu

```
1 const v0 = Array.prototype
2 v0.push = 42
```

V úryvku kódu 3.14 si môžeme všimnúť, že vygenerovaný kód môže zmeniť aj prototyp základných objektov, ktoré používame aj na uchovávanie typov. To nám pokazí náš kód a nebudeme schopný korektne zozbierať typy. Podobne ako v predchádzajúcich príkladoch musíme nájsť kompromis, koľko a aké funkcie si odzálohujeme aby sme zmenšili počet prípadov keď sa náš zber nepodarí a zároveň nezväčšovali a nespomaľovali príliš skript.

Program 3.15: Zmena typu viacerých premenných naraz

```
1 const v0 = ["foo"]
2 const v1 = [1]
3 const v2 = Array.prototype
4 v2.xyz = 42
```

Takisto si môžeme všimnúť v 3.15, že podobným spôsobom menením prototypov meníme na riadku *4* typy viacerým premenným naraz (všetkým poliam).
My si zas nemôžeme dovoliť kontrolovať typy všetkých premenných po každej inštrukcii, to by bolo príliš pomalé. Rovnako je ťažké detegovať, ktorým premenným sa zmenil typ, pokiaľ nechceme reimplementovať myšlienku prototypov do Fuzzilli. To znamená,

že zas nebudeme mať vždy presné typy. No toto nerobí až také veľké problémy, keďže základný typ budeme mať správny. Takisto aj veľa vlastností a metód ostane správnych, len zopár ich môže pribudnúť (neurobíme nič zlé, len budeme konzervatívny) alebo pár ubudne. Šanca, že pri generovaní si vyberieme práve túto vlastnosť, ktorá bola vymazaná, nie je až taká veľká.

3.3 Kooperatívny mód

V skutočnosti nemusíme používať typy **len** zo statickej analýzy alebo **len** zo zbierania počas behu programu. Ako sme videli aj v časti o limitáciách, zbierať celý typ odznova na každej inštrukcii je časovo náročné najmä pre veľké objekty. Niektoré inštrukcie ale typ zmenia len málo (napríklad pridajú/odoberú jednu metódu). Preto navrhneme nový prístup na zisťovanie typov, ktorý nebude zbierať typy po každej inštrukcii, ale bude zapájať aj abstraktný interpreter.

V tomto novo navrhovanom móde budeme zbierať typy len pri definícii premennej. Tieto zozbierané typy si uložíme spolu s programom (jeho inštrukciami). Následne keď budeme program vyťahovať z korpusu a používať pri ďalších mutáciách, budeme ho spracovávať podobne ako pôvodná verzia fuzzeru bez nášho vylepšenia. Postupne ho budeme staticky spracovávať abstraktným interpreterom. No tentokrát pri definícii novej premennej vložíme do abstraktného interpretera typ, ktorý sme získali pri zbieraní typov počas behu programu.

Tento vložený typ nahradí typ, ktorý by bol zistený abstraktným interpreterom (alebo dokonca v tomto prípade abstraktný interpreter sa ani nemusí pokúšať o odvodenie typu). No zároveň si abstraktný interpreter tento typ osvojí. Čo znamená, že pri ďalších inštrukciách, ktoré nedeklarujú nové premenné (len menia typy už vytvorených premenných), budeme tento zozbieraný typ používať aj pri statickom odvodzovaní.

Program 3.16: Nepotrebné zbieranie typov

```
1 const v0 = {}
2 updateType(0, v0)
3 v0.a = 4
4 updateType(0, v0) // netreba
```

Ako vidíme na ukážke 3.16, tak tentokrát už riadok *4* generovať nemusíme, keďže je zbytočne časovo náročný. Pôvodne zozbieraný typ *v0* len vložíme do abstraktného interpretera a typ na riadku *3* korektne odvodíme staticky. Týmto eliminujeme problémy s limitáciou z ukážky 3.13.

S touto ideou môžeme však zájsť ešte o krok ďalej. Totiž ako sme videli v 3.12, ak zakážeme spájanie výrazov pri zbieraní typov, tak nielen zväčšíme veľkosť skriptu,

ale dokonca môžeme v istých prípadoch zmeniť správanie programu.

No pri tomto kooperatívnom móde, kde nezbierame typy pri každej inštrukcii by sme si možno mohli dovoliť niektoré výrazy spojiť. Všimnime si, že dokonca ani pri definícii primitívnych typov nemusíme zbierať typy a vieme ich staticky odvodiť. Čo znamená, že inštrukcie ako *LoadString, LoadBool, ...* vieme spájať vo výrazoch bez straty kvality typovej informácie. A teda pri zbieraní typov môžeme zmeniť stratégiu spájania výrazov z *NeverInline* na *InlineOnlyLiterals*[6]. Túto variantu získavania typov budeme nazývať **kooperatívne zbieranie pri definícii premenných**.

Vyvinieme však ešte jednu variantu získavania typov a neskôr ich skúsime všetky porovnať. Totiž zbieraním typov len pri definícii sme síce ušetrili čas a pri jednoduchých inštrukciách ani nestratili korektný typ (ako v 3.16). No máme tu aj inštrukcie, kde by sme si predsa len mohli zozbierať typ premennej znova. Jednou z nich je napríklad zmena prototypu. Na takejto inštrukcii sa môže zmeniť skoro celý typ a to abstraktný interpreter odvodiť nevie. Zároveň sme však doteraz pre každú premennú zistili len jeden typ pre celý program. Čo je však v takejto situácií nevhodné, keďže typy pred a po zmene prototypu sú rôzne.

Musíme preto zmeniť štruktúru na zbieranie typov a tiež funkciu *updateType*. Typy budeme ukladať do JavaScript objektu, kde kľúč bude číslo premennej a jeho hodnota bude ďalší JavaScript objekt, kde kľúče budú čísla inštrukcií a hodnota bude zistený typ. Teda pre každú premennú budeme mať zoznam inštrukcií kde sa typ tejto premennej zmenil a zároveň aj samotný typ na ktorý sa zmenil.

Program 3.17: Príklad novej štruktúry na zbieranie typov

```
{
    0: {
        0: .integer
        2: .string
    },
    1: {
        1: .regexp
    }
}
```

Teda napríklad ak štruktúra na konci zbierania bude vyzerať ako v príklade 3.17, tak vieme, že premenná *v0* bola definovaná na inštrukcii *0* ako *integer*, ale na inštrukcii *2* bola zmenená na *string*. Naopak premenná *v1* bola definovaná až na inštrukcii *1* ako *regexp*. No odvtedy sa už jej typ nezmenil.

Nová implementácia funkcie *updateType* teda musí brať ako argument aj číslo inštrukcie a vyzerá teda nasledovne:

[6] v prílohe na /fuzzilli/Sources/Fuzzilli/Lifting/JSExpressions.swift

Program 3.18: Implementácia funkcie updateType pri zbieraní po inštrukcii

```
1 function updateType(varNumber, instrIndex, value) {
2     var currentType = getCurrentType(value)
3
4     if (types[varNumber] == null) types[varNumber] = {}
5
6     if (types[varNumber][instrIndex] == null) types[varNumber][instrIndex] =
          currentType
7     else types[varNumber][instrIndex] =
          types[varNumber][instrIndex].union(currentType)
8 }
```

Všimnime si riadok 7. Stále sme sa nezbavili *zjednotenia* typov. A to z jednoduchého dôvodu. Totiž aj pri jednej konkrétnej inštrukcii môže mať premenná rôzne typy ak je napríklad v cykle. S týmto moc nenarobíme, keďže aj pri prípadnej mutácii v cykle ostane a preto pozbieraný typ ako v ukážke 3.19 dáva najväčší zmysel.

Program 3.19: Rôzne typy jednej premennej na rovnakej inštrukcii

```
1 for(let v0 = "";v0 < 5;v0++) {
2 // v0 = .string | .integer
3 }
```

Na ukážke 3.20 zas môžeme vidieť, ktoré typy dokáže tento prístup zistiť lepšie.

Program 3.20: Rôzne typy jednej premennej na rôznych inštrukciách

```
1 const v1 = {}
2 // v1 = .object(ofGroup: "Object", ...)
3 v1.__proto__ = Array.prototype
4 // v1 = .object(ofGroup: "Array", ...)
```

Štruktúra je ale zložitejšia a teda odpovedať na dotaz aký je typ premennej vX v čase T nie je priamočiare. Samotnou implementáciou dátovej štruktúry na ukladanie typov vo fuzzeri sa budeme zaoberať v podkapitole 4.3.

Presné určovanie, ktoré inštrukcie výrazne menia typ a treba zozbierať typ znova, určuje trieda *TypeCollectionAnalyzer* a je ju možné nájsť v prílohe[7]. Keďže sa nám zmenila aj štruktúra ktorú posielame do fuzzera, musíme aktualizovať aj model Protobuf správy ktorou komunikujeme. Túto variantu získavania typov budeme nazývať **kooperatívne zbieranie pre každú inštrukciu**.

[7] /fuzzilli/Sources/Fuzzilli/Lifting/TypeCollectionAnalyzer.swift

K implementácii zbierania typov počas behu programu sme napísali/upravili aj niekoľko testov, aby sme sa pri ďalších úpravách vyvarovali zavedeniu nových chýb[8].

3.4 Porovnanie navrhnutých riešení

Myšlienky na vylepšenie typového systému Fuzzilli sme doteraz rozpracovali do 3 variantov:

- **Verzia 1** – Zbieranie typovej informácie počas behu programu
- **Verzia 2** – Kooperatívne zbieranie pri definícii premenných
- **Verzia 3** – Kooperatívne zbieranie pre každú inštrukciu

Ako sme už analyzovali skôr, ani implementácie týchto verzií nie sú dokonalé, majú svoje limitácie a nepozbierajú typy pre všetky možné vygenerované skripty (zbieranie môže potencionálne skončiť zlyhaním alebo prekročením časového limitu). Chceli by sme zistiť naozajstný dopad na reálnu prevádzku fuzzera a zbytočne neoptimalizovať hraničné prípady, ktoré v skutočnosti takmer nenastávajú.

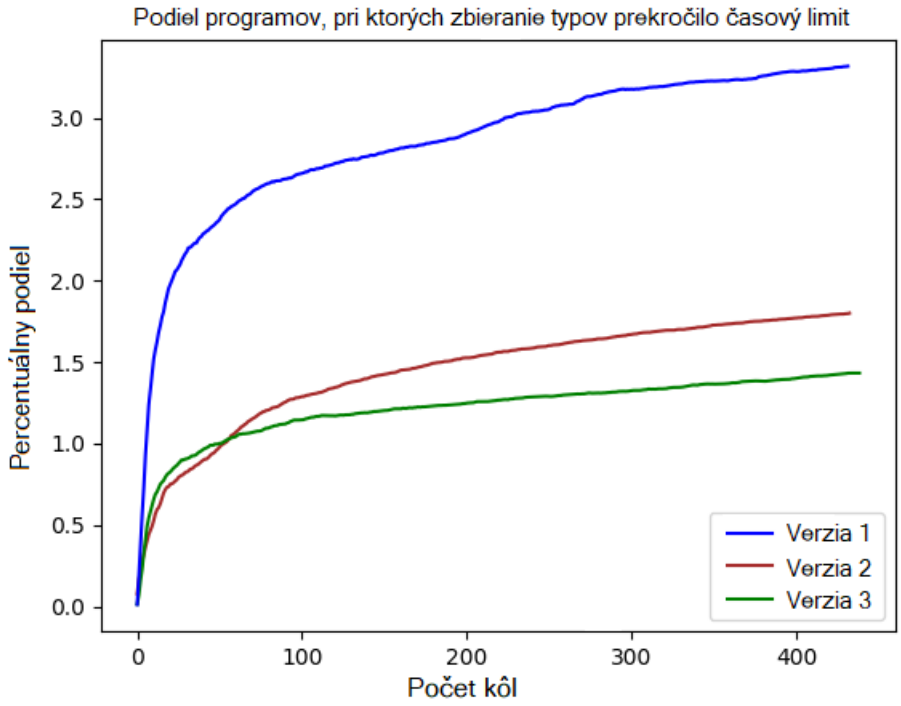
Naimplementovali sme preto zbieranie štatistík súvisiacich so zbieraním typov počas behu fuzz testovania. Každý program sme začali označovať výsledkom zbierania typov:[9]

- **úspešné** – Typy sme úspešne pozbierali.
- **prekročený časový limit** – Zbieranie trvalo príliš dlho. Zbieranie sme zastavili a pokračovali bez typov v programe.
- **zlyhanie** – Počas zbierania nastala chyba počas behu programu. Môže nastať napríklad vtedy, keď vygenerovaný program zmení prototyp poľa, ktorý využíva štruktúra na ukladanie typov. Aj v tomto prípade pokračujeme bez typov.
- **zatiaľ nezozbierané** – Zatiaľ sme typy tohto programu neskúšali zbierať.

Na každom stroji sme si začali udržovať počty jednotlivých výsledkov zbierania typov pre každý program z korpusu. Rovnako aj každý program si uložil svoj výsledok zbierania typov. Týmto sme zároveň zaistili, že pokiaľ posielame program medzi strojmi, nestane sa, že zbieranie typov toho istého programu spustíme viackrát na rôznych strojoch. Doteraz, prijímajúci stroj nevedel, či na programe bolo vykonané zbieranie typov. Ak sa pri programe nenachádzali žiadne typy, tak zbieranie typov mohlo napríklad prekročiť časový limit (neoplatí sa ho opakovať) alebo ešte vykonané nebolo. Presnú implementáciu je možné nájsť v prílohe v zázname (angl. *„commit“*) *e2d97bf*.

[8] v prílohe na /fuzzilli/Tests/FuzzilliTests/MutationsTest.swift

[9] v prílohe na /fuzzilli/Sources/Fuzzilli/FuzzIL/TypeCollectionStatus.swift



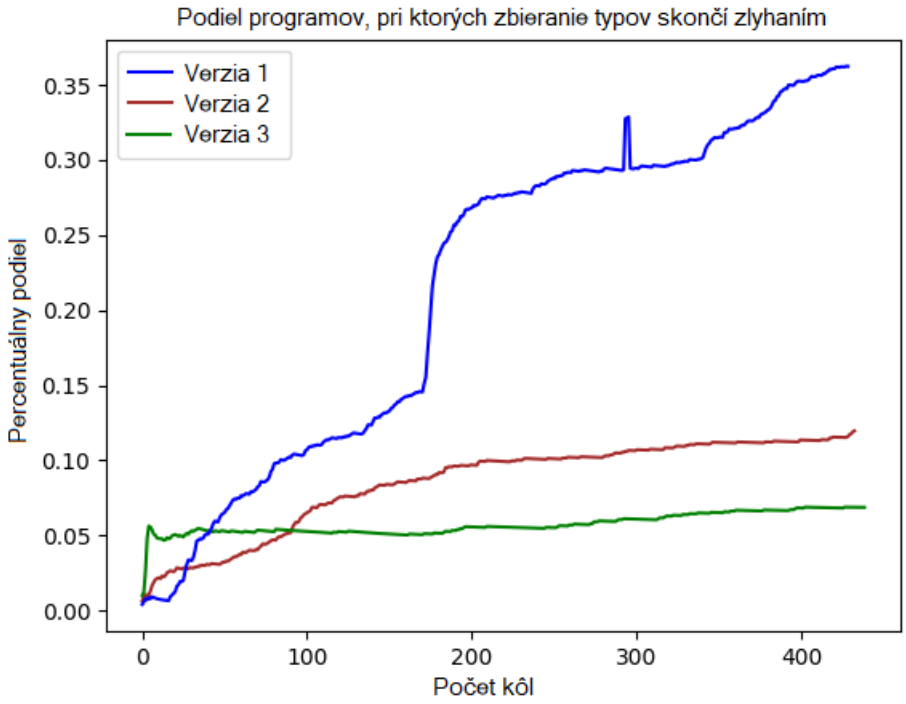
Obr. 3.1: Graf vývoja podielu programov z korpusu, pri ktorých sa nám nepodarilo zozbierať typové informácie z dôvodu prekročenia časového limitu

Po implementácii zbierania štatistík sme spustili nezávislé fuzz testovanie pre každú z troch nami naimplementovaných verzií zbierania typov. Každý z behov sme spustili s rovnakou konfiguráciou, aby sme boli schopný porovnať výsledky. Použili sme *JavaScriptCore* ako engine, ktorý fuzz testovaním testujeme. Zároveň sme museli povoliť zbieranie typov počas behu programu, keďže sme túto možnosť naimplementovali ako štandardne vypnutú.

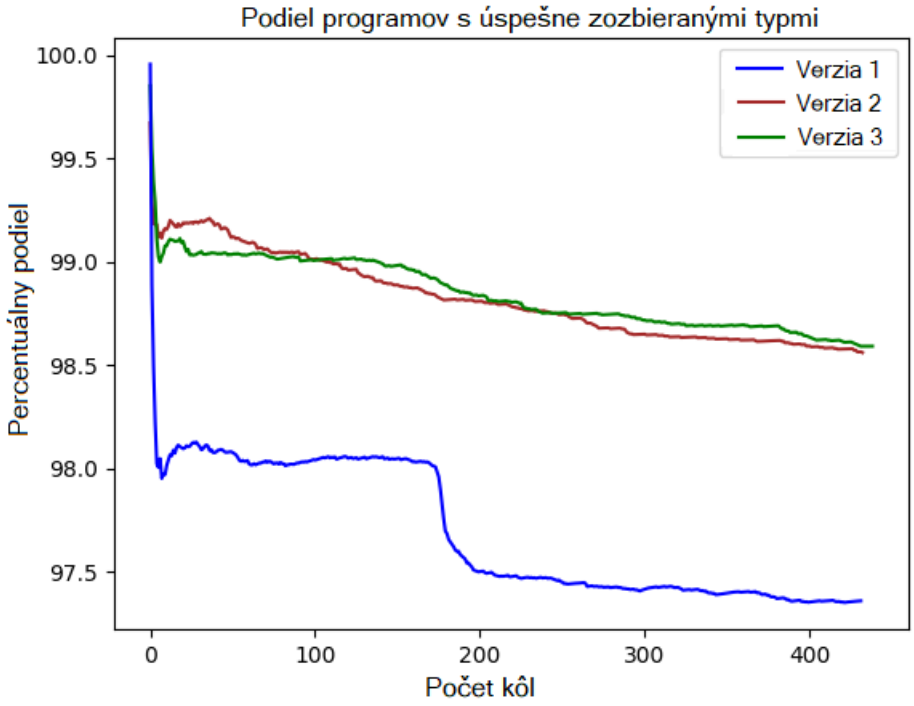
Mimo tejto špeciálnej možnosti (ktorou sme povolili použitie nášho vylepšenia) sme použili štandardnú konfiguráciu, ktorú používajú aj výskumníci v oblasti počítačovej bezpečnosti a je aj bližšie popísaná v prílohe[10]. Táto konfigurácia beží na *136* strojoch a výsledky sme pozbierali po *72* hodinách behu. Fuzzilli štatistiky ukladá po „kolách“, pričom každé z nich trvá *10* minút. Preto aj výsledné grafy majú časovú os v kolách.

Výsledky štatistík z testovania sme zhrnuli do grafov 3.1, 3.2 a 3.3. Môžeme na nich vidieť, ako sme naše zbieranie typov programu dokázali postupne zlepšovať. Našou najlepšou verziou sme dosiahli, že podiel zbieraní typov, ktoré skončia prekročením časového limitu je len niečo cez 1,3 % a podiel tých, ktoré skončia zlyhaním je zhruba 0,05 %. Čo nám dáva takmer **99 %** programov v korpuse s typovou informáciou zo zbierania počas behu programu. S týmto môžeme byť spokojný, keďže prípadov so zlyhaním nastáva minimum a počet prípadov, ktoré prekročia časový limit môžeme zredukovať

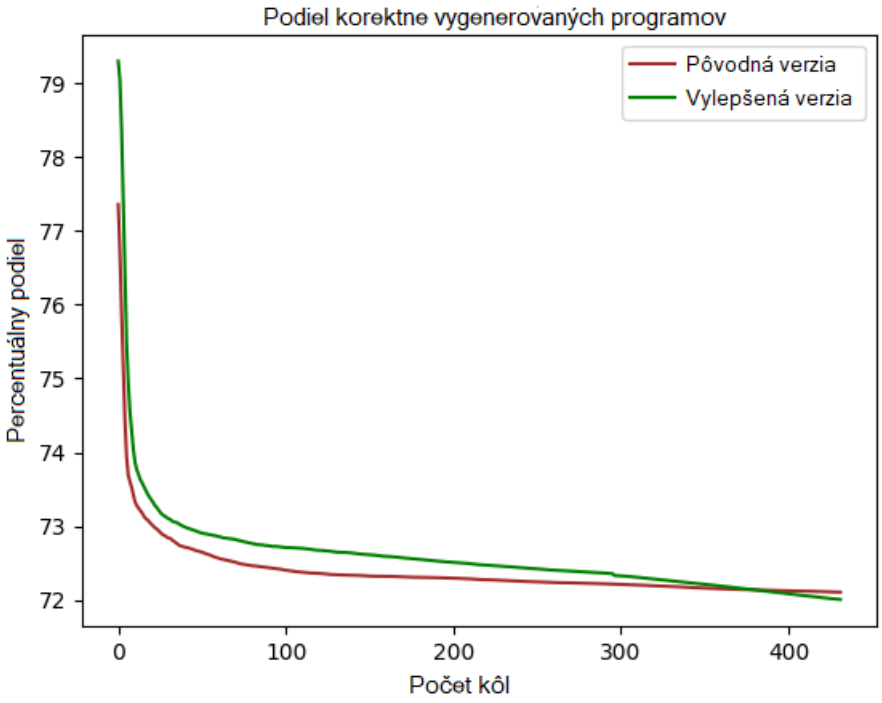
[10] v prílohe na /fuzzilli/Cloud/GCE/README.md



Obr. 3.2: Graf vývoja podielu programov z korpusu, pri ktorých sa nám nepodarilo zozbierať typové informácie z dôvodu zlyhania



Obr. 3.3: Graf vývoja podielu programov z korpusu obsahujúcich typové informácie zozbierané počas behu programu



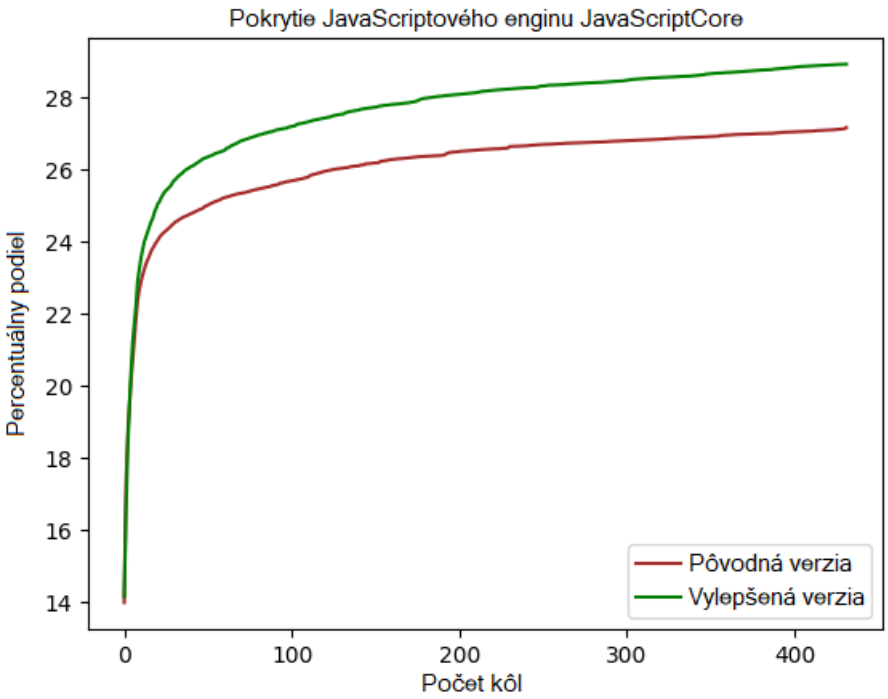
Obr. 3.4: Graf vývoja podielu korektne vygenerovaných programov v čase

napríklad zväčšením časového limitu. Takýto vysoký pomer programov s typmi môžeme považovať za úspech a ďalšie optimalizácie by nám mohli priniesť len malé zlepšenie, pričom by stáli oveľa viac úsilia.

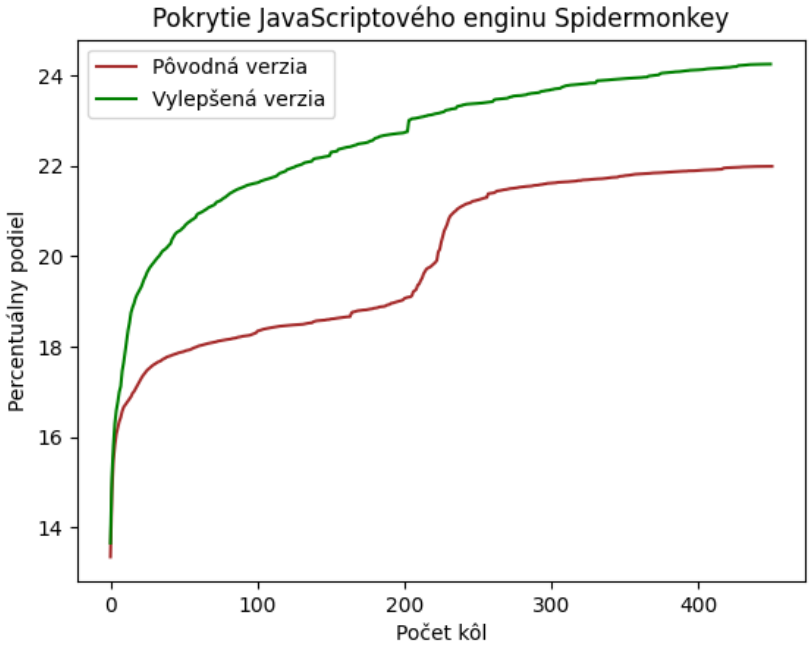
Keď sme si porovnali naše naimplementované verzie zbierania typovej informácie, nastal čas porovnať samotnú ideu zbierania typov počas behu programu s pôvodnou implementáciou fuzzera. Podobne ako predchádzajúce porovnanie aj teraz spustíme dva nezávislé behy fuzz testovania so štandardnou konfiguráciou na implementácii JavaScript enginu *JavaScriptCore*. V oboch pustíme implementáciu verzie *3*. Tentokrát ale v jednom behu povolíme zbieranie typov počas behu programu (vylepšenie tejto práce) a v druhom ho zakážeme (pôvodná implementácia).

Výsledky sú zaujímavé v tom, že sme nedosiahli výrazné zlepšenie v pomere korektne vygenerovaných programov (čo sme mohli priamočiaro očakávať), čo potvrdzuje aj graf 3.4. Na grafe 3.5 ale môžeme vidieť, že sme dokázali v rovnakom časovom okne pokryť (otestovať) väčšiu časť implementácie testovaného enginu ako v pôvodnej verzii fuzzera. Zlepšenie predstavuje približne **2** percentuálne body (čo je o *8 %* viac ako pri pôvodnej implementácii).

Rovnaké testovanie sme spustili na implementácii JavaScript enginu *Spidermonkey*. Výsledky boli veľmi podobné tým, ktoré sme spúšťali na *JavaScriptCore* engine. Z grafu 3.6 porovnania pokrytia pôvodnej a vylepšenej verzie fuzzera počas fuzz testovania enginu *Spidermonkey* je možné vidieť, že zlepšenie bolo podobné ako pri *JavaSc-*



Obr. 3.5: Graf vývoja pokrytia enginu JavaScriptCore v čase



Obr. 3.6: Graf vývoja pokrytia enginu Spidermonkey v čase

riptCore engine a to o **2** percentuálne body z *22 %* na *24 %* na konci fuzz testovania.

Na vysvetlenie tohto javu sa musíme zamyslieť hlbšie nad vlastnosťami typov získaných z pôvodného a vylepšeného algoritmu na odvodzovanie a ich dôsledkov na fuzz testovanie ako celok. Abstraktný interpreter býva často konzervatívny a radšej priradí premennej všeobecnejší typ ako by sa mal pomýliť. Napríklad výsledok sčítania dvoch premenných odvodí ako typ *primitive* (nie je to objekt). Naopak naše odvodzovanie typov je oveľa presnejšie, keďže zbiera typy počas behu programu (napríklad presne pozná typ výsledku sčítania). Fuzzer s presnejšou typovou informáciou dokáže generovať zaujímavejšie skripty. To znamená, že dokáže použiť pri generovaní iné / nové vlastnosti a metódy objektov, ktoré abstraktný interpreter nedokáže odvodiť. Týmto dokáže otestovať väčšiu časť testovaného enginu oveľa skôr.

Zároveň to vysvetľuje aj dôvod výraznejšieho nezlepšenia úspešnosti generovania korektných programov. Abstraktný interpreter totiž odvodzoval typy približne rovnako „správne“, ale typy neboli dostatočne konkrétne.

Kapitola 4

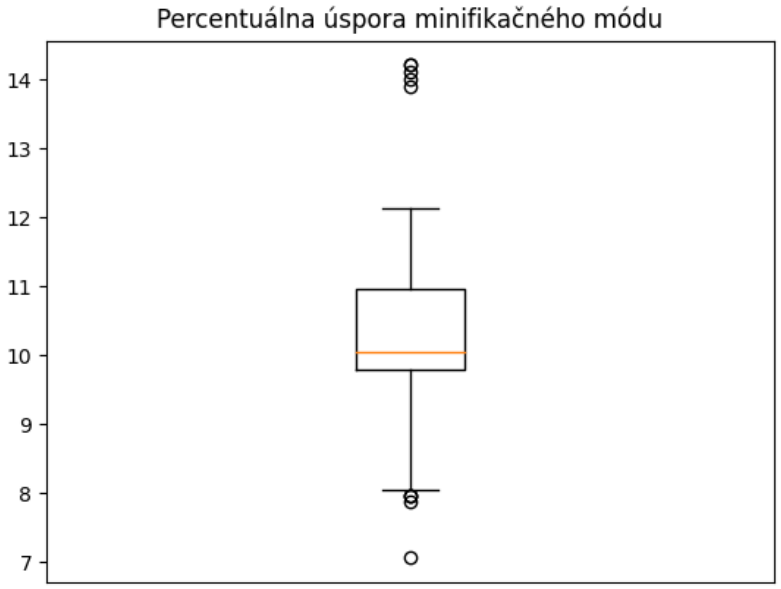
Časové a pamäťové optimalizácie

V predchádzajúcej kapitole sme sa zamerali na vytvorenie nových stratégií na odvodzovanie typov jednotlivých premenných a následne porovnali ich výkonnosť. V tejto kapitole sa skúsime zamerať na konkrétne praktické problémy/vylepšenia fuzzera a zoptimalizovať ich. Tieto vylepšenia budú väčšinou priamo alebo nepriamo súvisieť s implementáciou nášho typového zbierania, ale niektoré budú môcť byť použité aj nezávisle.

4.1 Minifikačný mód prekladača

Ako sme už spomínali, samotný program odovzdávame testovanému enginu pomocou zdieľanej pamäte. Teda chceme, aby bol čo najmenší. Zároveň si chceme jednotlivé programy aj ukladať/vypisovať, či už preto, že sme našli skript na ktorom spadne celý engine, alebo len chceme skontrolovať akým smerom sa testovanie uberá.
Pôvodne bol preklad z FuzzIL do JavaScriptu určený najmä na čítanie. My sme ako súčasť tejto práce naprogramovali *minify* mód pre prekladač, ktorým vieme povedať prekladaču nech vyrobí minimalistický preklad, ktorý vynecháva všetky medzery a nové riadky kde je to možné. Tento mód vždy zapneme pokiaľ robíme preklad kvôli vykonávaniu v JavaScript engine.

Za účelom testu efektívnosti sme vyskúšali počas fuzz testovania preložiť každý vygenerovaný program v minifikovanom aj klasickom móde. Zozbierali sme 1000 programov a urobili štatistiku z pomeru veľkosti minifikovaného a klasického prekladu. Na obrázku 4.1 máme vizualizáciu takejto štatistiky pomocou krabicového diagramu (angl. *boxplot*). Z grafu vidíme, že pomocou tohto módu vieme ušetriť bežne na veľkosti vygenerovaných JavaScript programov od **10 %** do **11 %**. Samotná implementácia sa nachádza v prílohe v zázname *c90914d*.



Obr. 4.1: Percentuálna úspora veľkosti generovaných skriptov

4.2 Typ *iterable*

Pomocou typov, ktoré Fuzzilli mal naimplementované nebolo možné zistiť, či sa cez premennú dá iterovať. Čo prinášalo generovanie problematických skriptov, ktoré boli následne zamietnuté JavaScript enginom.

Program 4.1: Skript vygenerovaný bez typu *iterable*

```
1 const v0 = {}
2 const v1 = [...v0]
```

Doteraz bolo možné vygenerovať skript 4.1, keďže Fuzzilli nemalo ako vedieť, či je *v0* iterovateľná premenná. V prílohe v zázname *a3b1623* sme naimplementovali nový typ *iterable*. Takisto sme staticky otypovali vstavané premenné a naimplementovali rozoznávanie tohto typu do nášho zbierania typov počas behu programu. V generátoroch kódu sme spresnili, kde treba vyžadovať použitie premenných cez ktoré sa dá iterovať. Negenerovaním skriptov, ktoré budú zamietnuté šetríme čas fuzzera.

4.3 Dátová štruktúra na ukladanie typov v programe

V pôvodnej verzii fuzzera sa v triede *Program*[1] ukladal len samotný skript (zoznam inštrukcií). Na ukladanie samotných typov nebol dôvod, keďže sa používalo len statické

[1] v prílohe na /fuzzilli/Sources/Fuzzilli/FuzzIL/Program.swift

odvodzovanie, ktoré je rýchle a až tak nám nevadilo ak sme ho museli zopakovať. No pri naprogramovaní ľubovoľného variantu zbierania typov (potencionálne skombinovaného s kooperatívnym módom) by sme sa radi opakovanému odvodzovaniu typov vyhli.

Podobne, ak posielame niektorý program alebo dokonca celý korpus inej inštancii fuzzera, tak pôvodne sa žiadna informácia o typoch neposielala a prijímajúca inštancia si ich musela odvodiť sama.

Posledným výrazným dôvodom na zavedenie dátovej štruktúry na ukladanie typov je metóda *splice* používaná pri vytváraní nových skriptov[2]. Táto metóda sa snaží časť jedného skriptu z korpusu použiť pri vytváraní nového. Tu vieme použiť typy pôvodného skriptu pre vytvorenie lepšieho skriptu, ale je zbytočné ich odvodzovať opätovne, keď sme to už raz robili pri vkladaní do korpusu. Ak si ich uložíme, vieme ich použiť opäť.

Najprv si zhrňme, čo budeme od takejto dátovej štruktúry očakávať. Teda aké dáta musíme ukladať a na aké dotazy musíme vedieť efektívne odpovedať.

1. **Typ a jeho kvalita** – Chceme, aby dáta v našej dátovej štruktúre neobsahovali len samotný typ, ale aj jeho kvalitu, to znamená ako sme ten typ odvodili. Aktuálne to môže byť buď *inferred* (odvodené abstraktným interpreterom) alebo *runtime* (získaný zbieraním počas behu programu). Túto informáciu môžeme využiť jednak pri ladení programu (angl. *debugging*) alebo aj pri iných operáciách, kde chceme používať len typy s vyššou kvalitou (*runtime*).

2. **Vyfiltruj typy danej kvality** – Pri kooperatívnom móde chceme vkladať do abstraktného interpretera len typy s kvalitou *runtime*.

3. **Zisti typové zmeny pre danú inštrukciu** – Túto funkcionalitu využijeme pri kooperatívnom móde. Ako postupne spracúvame skript, tak pre aktuálnu inštrukciu chceme vložiť typové zmeny do abstraktného interpretera.

4. **Zisti typ danej premennej pred / po danej inštrukcii** – Pri vytváraní mutácií alebo generovaní skriptu chceme často vedieť typ premennej v danom čase, aby sme vedeli vytvoriť korektnú mutáciu.

5. **Pridaj typ danej premennej po danej inštrukcii danej kvality** – Odvodili sme typ premennej tak ho chceme vedieť pridať do štruktúry.

Požiadavku 1 splníme tým, že prvky, ktoré si budeme ukladať budú typu *TypeInfo*[3].

Pre šetrenie pamäti samotná štruktúra nebude ukladať typ pre každú inštrukciu a každú premennú, ale len typ a inštrukciu kde sa typ zmenil (alebo pri definícii pre-

[2] v prílohe na /fuzzilli/Sources/Fuzzilli/Core/ProgramBuilder.swift
[3] v prílohe na /fuzzilli/Sources/Fuzzilli/FuzzIL/TypeInfo.swift

mennej). Typy budú uložené v poli, kde index bude číslo premennej a hodnota bude pole štruktúr *TypeInfo* zoradené podľa indexu inštrukcie.

Program 4.2: Program pre ukážku dátovej štruktúry

```
const v0 = {}
const v1 = "foo"
v0.a = "bar"
```

Program 4.3: Dátová štruktúra pre program 4.2

```
[
  [
    TypeInfo(index: 1, type: .object(...), quality: .runtime),
    TypeInfo(index: 3, type: .object(withProperties: ["a", ...], ...),
        quality: .inferred)
  ],
  [TypeInfo(index: 2, type: .string, quality: .inferred)]
]
```

Príklad ako taká štruktúra môže vyzerať máme v ukážke 4.3. Pre objekt by sme samozrejme pozbierali aj všetky vlastnosti a metódy jeho prototypu, ale pre ich početnosť sme ich nahradili 3 bodkami.

Na požiadavky typu 4 vieme odpovedať tak, že podľa čísla premennej si v poli nájdeme zoznam typových zmien. Následne binárne vyhľadáme poslednú typovú zmenu pred žiadanou inštrukciou, ktorá nám povie typ premennej v hľadanom čase.

Pri požiadavke typu 5 postupujeme podobne. Najprv nájdeme poslednú typovú zmenu pre danú premennú a inštrukciu. Ak táto typová zmena je presne na inštrukcii kde chceme vložiť náš typ, tak stačí tento typ prepísať. Ak to tak nie je, musíme do tohto poľa vložiť za neho nový záznam. Toto môže byť pomalé, ale musíme si uvedomiť, kedy túto požiadavku využívame. Najbežnejšie je to vtedy, keď postupne spracúvame program a teda len pridávame typové zmeny na koniec poľa typových zmien.
Druhú možnosť budeme rozoberať v podkapitole 4.4, no pri nej budeme len nahrádzať typovú zmenu, čo je tiež rýchle.

Ak by sme potrebovali rýchlu všeobecnú požiadavku 5, mohli by sme použiť usporiadanú množinu (angl. *ordered set*) na ukladanie typových zmien. No keďže to nepotrebujeme a počet typových zmien pre konkrétnu premennú zvyčajne nebýva veľký (zvyčajne nepresiahne *3*), ušetríme takto nadbytočné náklady na ukladanie usporiadanej množiny.

Požiadavku 2 vyriešime priamočiaro a to prejdením celej štruktúry a vytvorením

novej len s prvkami vhodnej kvality.

Požiadavku 3 vyriešime podobne. Prejdeme celú štruktúru a preusporiadame tak, aby sme následné dotazy o typových zmenách (pre konkrétnu inštrukciu) boli schopný nájsť okamžite. Preusporiadaná štruktúra bude pole, kde každý index bude prezentovať jednu inštrukciu v programe. Na tomto indexe budeme mať len zoznam dvojíc (premenná, nový typ). Tento zoznam si môže abstraktný interpreter prejsť a osvojiť si tieto zmeny.

Program 4.4: Dátová štruktúra vzniknutá požiadavkou 3 pre program 4.2

```
[
  [(v0, .object(...))],
  [(v1, .string)],
  [(v0, .object(withProperties: ["a", ...], ...))],
]
```

V ukážke 4.4 máme takúto štruktúru, ktorá vznikla z 4.3.

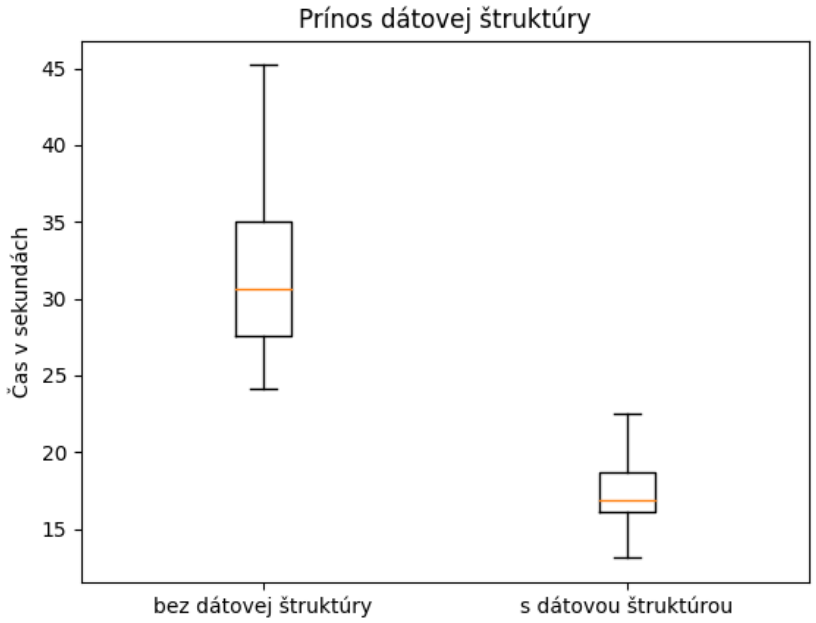
Iná možnosť je vytvoriť túto preusporiadanú štruktúru priamo pri vytváraní základnej štruktúry a aktualizovať ju naraz s pôvodnou štruktúrou. No stálo by to približne dvojnásobne viac pamäte a preto sme sa rozhodli pre vytváranie tejto preusporiadanej štruktúry na požiadanie.

Pre porovnanie sme skúšali porovnať výkonnosť fuzzera bez použitia dátovej štruktúry a s jej použitím. Príloha obsahuje aj sadu referenčných skúšok (angl. *benchmark*)[4]. Sada je zameraná na porovnávanie rýchlosti generovania nových skriptov. Referenčné skúšky sme vykonali na implementácii fuzzera pred implementáciou dátovej štruktúry a následne sme ich zopakovali po jej implementácii. V oboch prípadoch sme referenčné skúšky vykonali 50 krát a odmerali rýchlosť ich vykonania.

Výsledky sme vizualizovali pomocou krabicového diagramu v grafe 4.2. Z grafu môžeme vidieť, že sa nám podarilo zredukovať čas strávený generovaním rovnakého množstva programov takmer na polovicu. Medián rýchlosti vykonania referenčných skúšok bez dátovej štruktúry bol *30,67* sekundy. Zatiaľ čo rýchlosť vykonania skúšok s naimplementovanou dátovou štruktúrou bol *16,83* sekundy.

Samotnú implementáciu tejto dátovej štruktúry je možné nájsť v prílohe[5].

[4] /fuzzilli/Sources/Benchmarks/main.swift
[5] /fuzzilli/Sources/Fuzzilli/FuzzIL/ProgramTypes.swift



Obr. 4.2: Porovnanie rýchlosti generovania programov s pomocou dátovej štruktúry a bez nej

4.4 Zdieľanie typových tried

Uchovávanie typov v programe však nie je pamäťovo najšetrnejšie. Postupným plynutím času fuzz testovania sa nám zväčšuje korpus (a teda počet programov v ňom). Po čase tento rast prestane, keďže postupne začneme korpus čistiť od starých programov. Takisto postupne generujeme aj väčšie programy s viac premennými ku ktorým si treba uchovávať informáciu o type.

Keď si pozrieme skripty, ktoré generujeme, tak sa v nich často vyskytujú podobné konštrukty. Tým, že v jazyku FuzzIL nemôžeme spájať výrazy, tak keď chceme vyskladať zložitejší konštrukt, musíme často postupne vygenerovať viac jednoduchších, ktoré ďalšími inštrukciami skombinujeme. Podobne, nové programy vznikajú mutáciou starších, čo znamená, že aj medzi programami existujú veľké rovnaké časti. Keďže sú rovnaké, budú mať aj rovnaké typy. A tu sa naskytuje možnosť zoptimalizovať pamäťové nároky na uchovávanie typových informácií.

Každý typ priradený niektorej premennej pre určitú inštrukciu je inštanciou štruktúry *Type*[6]. Táto obsahuje informáciu o *definiteType*, *possibleType* a *ext*. Prvé dve menované sú len bitové masky, ktoré nezaberajú veľa miesta. No *ext* je inštancia triedy *TypeExtension*[7] a tá okrem iného obsahuje zoznam vlastností a metód objektu, čo môže

[6] v prílohe na /fuzzilli/Sources/Fuzzilli/FuzzIL/TypeSystem.swift
[7] v prílohe na /fuzzilli/Sources/Fuzzilli/FuzzIL/TypeSystem.swift

byť potencionálne veľa reťazcov.

Preto si pre náš korpus môžeme vytvoriť množinu použitých inštancií *TypeExtension* a pokiaľ pridávame nový program do korpusu tak postupne prechádzame všetky typy, ktoré sa v programe vyskytujú. Ak typ obsahuje takú *TypeExtension*, ktorá sa v našej množine ešte nenachádza, pridáme ju do množiny. Pokiaľ sa už v množine použitých inštancií ale nachádza, tak namiesto vytvorenia novej inštancie, len budeme ukazovať na inštanciu z množiny použitých inštancií a tak ušetríme pamäť využitú na korpus (ako sme už v práci spomínali, pridávanie programu do korpusu nie je až taká častá udalosť a toto nahrádzanie typov zdieľanými inštanciami si môžeme dovoliť).

Raz za 30 minút sa na každej inštancii fuzzera spúšťa proces čistenia korpusu od starých programov. Toto čistenie sa nevykonáva až tak často a preto si môžeme vtedy dovoliť zahodiť celú množinu použitých inštancií a vybudovať ju odznova z programov, ktoré v korpuse ostanú aj po čistení. Implementáciu je možné nájsť v prílohe[8].

Náš fuzzer ale môže bežať na viacerých strojoch, ktoré medzi sebou komunikujú pomocou správ vo formáte *Protobuf*. Týmito správami si medzi sebou posielajú aj korpus (a teda aj vygenerované programy s ich typmi). To ale znamená, že keď sa budeme snažiť zakódovať program do tohto formátu, tak veľkosť správ nezredukujeme, keďže samotný *TypeExtension* sa vždy serializuje rovnako bez ohľadu na našu množinu použitých inštancií na aktuálnom stroji. Preto musíme naimplementovať vlastnú zložitejšiu serializáciu typov[9]. Protobuf správa serializujúca typ, bude vždy obsahovať buď obyčajne serializovaný *TypeExtension* alebo index (číslo). Tento index sa použije v serializácii vtedy, ak celková Protobuf správa korpusu už obsahuje serializovanú *TypeExtension*, ktorú sa práve snažíme serializovať. Index nám bude hovoriť kde v správe sa nachádza už serializovaný *TypeExtension*. Takto budeme vedieť celý korpus efektívne poslať na iný stroj a zároveň aj efektívne deserializovať.

Po implementácii tohto vylepšenia sme spustili fuzz testovanie a sledovali veľkosť množiny použitých inštancií. Aby sme vedeli vyčísliť zlepšenie nášho vylepšenia, tak sme si pri tomto fuzz testovaní pamätali pre každý prvok (v množine použitých inštancií) aj počet typov premenných v korpuse, ktoré túto *TypeExtension* používajú. Takto sme vedeli určiť koľko inštancií by sme mali v pamäti bez vylepšenia.

Spustili sme *5* behov a zastavili sme ich po niekoľkých hodinách, keď sa veľkosť korpusu stabilizovala (počet pridávaných programov bol podobný ako počet zmazaných). Výsledky máme možnosť vidieť v tabuľke 4.1. Vidíme, že počet inštancií triedy *TypeExtension* v pamäti sa nám podarilo zredukovať približne **200-násobne**.

[8] /fuzzilli/Sources/Fuzzilli/Core/Corpus.swift

[9] Po diskusii tohto problému a možného riešenia s autorom fuzzera bola implementácia zlepšenia serializácie typov naprogramovaná autorom.

Beh	Veľkosť korpusu	Počet inštancií po vylepšení	Počet inštancií bez vylepšení
1	2748	377	99585
2	3509	477	84032
3	3844	521	90021
4	3654	401	88513
5	3549	466	90528

Tabuľka 4.1: Tabuľka ukazujúca efektívnosť vylepšenej pamäte

4.5 Zrýchlenie abstraktného interpretera

Napriek tomu, že sme v celej práci tvrdili, že statické odvodzovanie abstraktným interpreterom je rýchle, pokúsime sa ho ešte mierne zrýchliť.

Pôvodná implementácia abstraktného interpretera si udržiavala tabuľku v ktorej mala pre každú premennú uložený jej typ. Vonkajšiemu prostrediu poskytovala metódu *execute*, ktorá dostala inštrukciu na základe ktorej aktualizovala vnútornú štruktúru interpretera a vrátila typové zmeny, ktoré sa touto inštrukciou udiali. Takto sme vedeli rýchlo odpovedať na otázku aký typ má aktuálne premenná *vX*.

Pre klasické inštrukcie má interpreter zoznam jednoduchých pravidiel, ktoré pri volaní metódy *execute* aplikuje. My sa však pozrieme na spracovanie *blokových* inštrukcií a pokúsime sa o optimalizáciu.

Blokové inštrukcie nám zároveň vytvárajú nový kontext (angl. *scope*), ktorý ovplyvňuje typy premenných. Niektoré blokové inštrukcie vytvárajú napríklad kontext, ktorý je podmienečne vykonaný (cykly alebo *if* blok). To znamená, že po konci tohto bloku sa typ každej premennej zmení jedným z dvoch spôsobov:

- Premenná existovala len vnútri tohto bloku, jej typ môžeme **zahodiť**.
- Nevieme, či sa blok vykonal, preto typ nastavíme na **zjednotenie** aktuálneho typu a typu pred začiatkom bloku.

Pôvodná implementácia abstraktného interpretera si neudržiavala jednu tabuľku typov, ale zásobník tabuliek. To znamená, že ak dostaneme inštrukciu začiatku bloku, tak pridáme navrch zásobníka kópiu aktuálne najnovšej tabuľky typov (tá čo je na vrchu zásobníka). Naopak, ak príde inštrukcia konca bloku, tak vyberieme dve tabuľky z vrchu zásobníka (aktuálnu a tú z rodičovského kontextu) a typy pre jednotlivé premenné zjednotíme. Takto vytvoríme novú tabuľku, ktorej typy zodpovedajú stavu po konci tohto bloku a môžeme ju teda pridať do zásobníka. Špeciálny prípad nastane ak premenná existovala iba vo vnútornom kontexte. Vtedy premennú do novej tabuľky nezaradíme.

Toto riešenie má však dve slabé miesta. Pri každom začiatku bloku musíme celý stav (tabuľku typov) skopírovať, čo je časovo aj pamäťovo zbytočne náročné. Takisto tento koncept nepodporuje viac disjunktných vetiev podmieneného vykonávania, napríklad *if-elseif-else*. Pôvodná implementácia síce podporovala spracovanie dvoch disjunktných vetiev *if-else* špeciálnym prístupom, ale nebolo možné jednoducho doimplementovať podporu viacerých disjunktných vetiev.

Prvou myšlienkou na vylepšenie je mať okrem konceptu vzťahov kontextov *rodič-dieťa* aj vzťah *súrodenec-súrodenec*. Zatiaľ čo prvý vzťah zachytáva vnorené kontexty, druhý zachytáva disjunktné kontexty, z ktorých sa v programe vykoná práve jeden. Dátovú štruktúru na ukladanie stavov v abstraktnom interpreteri implementujeme ako zásobník zásobníkov.

Program 4.5: Implementácia stavu interpretera

```
struct InterpreterState {
    private class State {
        public var types = VariableMap<Type>()
    }

    private var stack: [[State]]
    private var activeState: State
    private var parentState: State
    private var currentState: State

    init() {
        activeState = State()
        parentState = State()
        stack = [[parentState], [activeState]]
        currentState = State()
    }
}
```

Zároveň si budeme pamätať oddelene celkovú tabuľku typov (*currentState*), ktorá bude slúžiť len ako rýchla pamäť na odpovedanie na otázku aký typ má aktuálne konkrétna premenná. Túto tabuľku by sme vedeli vždy dostať prechodom po zásobníku a aplikovaním typových zmien. Vytvárať ju pri každej otázke by bolo ale časovo náročné. Možnú implementáciu môžeme vidieť v ukážke 4.5 (metódy na aktualizáciu stavov si predstavíme neskôr).[10]

[10]Implementácia nášho zrýchlenia bola neskôr mierne preprogramovaná autorom fuzzera, ale naše myšlienky zostali zachované, ukážky v práci sú pre kontinuitu inšpirované aj aktuálnou verziu.

Druhá myšlienka je, že namiesto uchovávania celého stavu všetkých premenných v zásobníku, si chceme pre každý stav pamätať len tie typy ktoré sa zmenili. Typy si budeme pamätať v stavoch tak, že ak nejaká zmena nastane, tak aktuálny typ si poznačíme do nášho stavu a zároveň starý typ si poznačíme do rodičovského stavu. Samozrejme starý typ poznačíme do rodičovského kontextu iba v prípade, že toto je prvá typová zmena tejto premennej v aktuálnom bloku (inak sme pôvodný typ už zaznačili do rodičovského kontextu).

Keďže si pamätáme len typové zmeny, tak typ v rodičovskom kontexte nemusí byť ešte poznačený. Napríklad v prípade, keď sa typ naposledy menil v kontexte rodiča aktuálneho rodiča. Zaznačenie starého typu do rodičovského stavu využijeme v prípade, keď aktuálny (možno podmienený) blok skončí. Vtedy typy premenných závisia aj od pôvodných typov.

Ukážka 4.6 ukazuje ako sa typy aktualizujú. Všimnime si, že teraz pri inštrukcii začatia nového detského kontextu nemusíme celý stav kopírovať, ale len vytvoríme nový prázdny stav, keďže zatiaľ v novom bloku nebola žiadna typová zmena.

Program 4.6: Aktualizácia typu v stave

```
mutating func updateType(of v: Variable, to newType: Type, from oldType:
    Type? = nil) {
  // Typ nothing predstavuje premennú nedefinovanú v rodičovskom kontexte
  let oldType = oldType ?? currentState.types[v] ?? .nothing
  // Ulož starý typ ak ešte uložený nie je
  if parentState.types[v] == nil {
    parentState.types[v] = oldType
  }
  activeState.types[v] = newType
  currentState.types[v] = newType
}
```

Program 4.7: Spracovanie inštrukcie abstraktným interpreterom

```
func execute(_ instr: Instruction) -> [(Variable, Type)] {
  typeChanges = []
  executeOuterEffects(instr)
  switch instr.op {
    case is BeginIf:
      state.pushChildState()
    case is BeginElse:
      state.pushSiblingState(typeChanges: &typeChanges)
    case is BeginWhile, is BeginFor, is BeginAnyFunctionDefinition, ...:
      state.pushChildState()
```

```
11        state.pushSiblingState(typeChanges: &typeChanges)
12      case is EndIf, is EndWhile, is EndFor, is EndAnyFunctionDefinition, ...:
13        state.mergeStates(typeChanges: &typeChanges)
14      ...
15    }
16    executeInnerEffects(instr)
17    return typeChanges
18  }
```

V ukážke 4.7 máme možnosť vidieť ako abstraktný interpreter spracúva inštrukcie. Pošleme mu inštrukciu, aktualizuje si svoj stav s typmi premenných a vráti nám zoznam zmenených premenných s ich novými typmi. Na riadku *3* a *16* sa vyhodnocuje samotná inštrukcia, túto časť sme nemenili a preto sa jej nebudeme hlbšie venovať. Zmenili sme však riadky medzi nimi slúžiace na spracovanie kontextov. Tu môžeme vidieť, že konštrukt *if* vytvára detský kontext (keďže sa v aktuálnom kontexte vykonať nemusí). Naopak konštrukt *else* vytvára súrodenecký kontext, keďže sa vykoná buď doteraz spracovávaný kontext (z konštruktu *if*) alebo nový.

Začiatok cyklu alebo funkcie vytvorí jeden prázdny kontext a k nemu aktuálny pre spracovanie jeho tela. Totiž telo cyklu alebo funkcie sa nemusí vykonať ani raz (prázdny kontext).

Koniec podmieneného bloku spôsobí spájanie kontextov. Vraciame sa totiž do rodičovského kontextu ale nevieme, ktorý zo súrodencov sa vykonal. Preto zjednotíme typy zo všetkých možných súrodencov.

Program 4.8: Pridanie detského kontextu

```
1  mutating func pushChildState() {
2    parentState = activeState
3    activeState = State()
4    stack.append([activeState])
5  }
```

Pridanie detského kontextu spravíme veľmi jednoducho. Vytvoríme na zásobníku nový stav, viď 4.8.

Program 4.9: Pridanie súrodeneckého kontextu

```
1  mutating func pushSiblingState(typeChanges: inout [(Variable, Type)]) {
2    for (v, t) in activeState.types {
3        if t != .nothing && parentState.types[v] != .nothing &&
             parentState.types[v] != currentState.types[v] {
4            typeChanges.append((v, parentState.types[v]!))
5            currentState.types[v] = parentState.types[v]!
```

```
      }
  }
  activeState = State()
  stack[stack.count - 1].append(activeState)
}
```

Pridanie súrodeneckého stavu je o niečo zložitejšie, keďže musíme vymazať zmeny, ktoré vznikli v aktuálnom kontexte. Toto spravíme hneď v úvode ukážky 4.9. Následne vytvoríme súrodenecký kontext. Typová zmena nenastala ak platí jedno z pravidiel.

1. $t == .nothing$ – Premenná neexistuje v súrodeneckom kontexte
2. $parent == .nothing$ – Premenná je lokálna v súrodeneckom kontexte
3. $parentState.types[v] \; != currentState.types[v]$ – Typ sa nezmenil

Program 4.10: Spojenie súrodeneckých kontextov

```
mutating func mergeStates(typeChanges: inout [(Variable, Type)]) {
  let statesToMerge = stack.removeLast()
  var numUpdatesPerVariable = VariableMap<Int>()
  var newTypes = VariableMap<Type>()

  ...Program 4.11

  ...Program 4.12

  ...Program 4.13
}
```

Na záver tu máme implementáciu funkcie na spájanie súrodeneckých kontextov 4.10 rozdelenú na 3 menšie časti. Chceme dostať zjednotenie typov cez všetkých súrodencov (práve jeden z nich sa vykonal).

Program 4.11: Spojenie samotných kontextov

```
for state in statesToMerge {
  for (v, t) in state.types {
    // Preskočme lokálne premenné
    guard parentState.types[v] != .nothing else { continue }

    if newTypes[v] == nil {
      newTypes[v] = t
      numUpdatesPerVariable[v] = 1
```

```
    } else {
      newTypes[v]! |= t
      numUpdatesPerVariable[v]! += 1
    }
  }
```

Najskôr v 4.11 zjednotíme typy premenných cez jednotlivé súrodenecké kontexty.

Program 4.12: Spojenie s rodičovským kontextom v prípade potreby

```
for (v, c) in numUpdatesPerVariable {
  if c != statesToMerge.count {
      newTypes[v]! |= parentState.types[v]!
  }
}
```

Potom v 4.12 zjednotíme typ ešte s pôvodným z rodičovského kontextu. Toto musíme urobiť v prípade, keď existuje súrodenecký kontext, ktorý túto premennú nemenil a teda je tu možnosť, že typ premennej zostal rovnaký, aký bol pôvodne.

Program 4.13: Aktualizácia typov

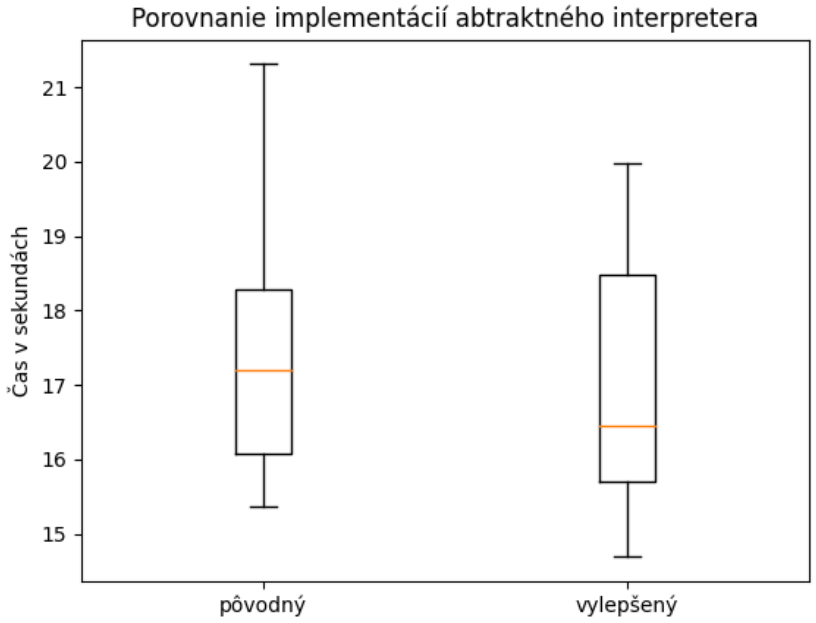
```
let oldParentState = parentState
activeState = parentState
parentState = stack[stack.count - 2].last!

for (v, newType) in newTypes {
  if currentState.types[v] != newType {
    typeChanges.append((v, newType))
  }

  updateType(of: v, to: newType, from: oldParentState.types[v])
}
```

Následne už len v 4.13 aktualizujeme stav s odvodenými typmi.

Kompletnú novú implementáciu je možné nájsť v prílohe v zázname *35c2ecb*. Po tomto vylepšení sme znovu spúšťali referenčné skúšky (rovnaké ako v podkapitole 4.3). Tentokrát sme ich spúšťali najprv s pôvodnou verziou abstraktného interpretera a následne s novou, nami naimplementovanou, verziou. Pre každú verziu sme referenčné skúšky spustili 50 krát.

Výsledky sme rovnako vizualizovali pomocou krabicového diagramu v grafe 4.3. Z grafu môžeme vidieť, že tentokrát sa nám tak výrazné zlepšenie nepodarilo. Medián

rýchlosti vykonania referenčných skúšok klesol z *17,2* sekundy na *16,44* sekundy.



Obr. 4.3: Porovnanie rýchlosti generovania programov s novou a starou implementáciou abstraktného interpretera

Musíme však poukázať na to, že zrýchlenie nebolo jediným cieľom tohto vylepšenia. Rovnako sme dokázali zovšeobecniť koncept statického odvodzovania myšlienkou súrodeneckých kontextov. Po tejto implementácii už nebude problém povoliť Fuzzilli generovať konštrukt *if-elseif-...-else* s viacerými vetvami.

Záver

Na začiatku práce sme si predstavili problematiku bezpečnosti JavaScriptových enginov. Načrtli sme dôležitosť tejto témy najmä v dnešnej dobe. Následne sme si predstavili fuzz testovanie ako nástroj, ktorým vieme včas hľadať potencionálne chyby v implementácii JavaScriptového enginu. Predstavili sme si rôzne paradigmy, ktoré vieme v tejto oblasti využiť.

V ďalšej kapitole sme si predstavili samotný fuzzer Fuzzilli a jeho riešenie problému generovania sémanticky korektných JavaScriptových programov. Riešenie spočívalo v zavedení medzijazyka FuzzIL, ktorý sme si dôkladne predstavili. Ukázali sme si ako Fuzzilli také programy generuje, mutuje a prekladá do JavaScriptu. Takisto sme si ukázali samotný typový systém, ktorý je jedným z kľúčových prvkov, vďaka ktorým dokážeme generovať korektné programy. Zároveň sme poukázali na niektoré jeho neoptimálne vlastnosti, ktoré sme sa pokúsili vylepšiť.

V tretej kapitole sme už navrhli nový systém odvodzovania typov v programoch medzijazyka FuzzIL. Ukázali sme si výhody **zbierania typov počas behu programu** a jeho možnej implementácie do samotného fuzzera. Porovnali sme jeho vlastnosti s pôvodne naimplementovaným odvodzovacím systémom, založeným na abstraktnom interpreteri. Následne sme rozanalyzovali limitácie aj tohto prístupu a ukázali sme si príklady za akých okolností nedostaneme dokonca žiadne typy pre program. Tieto často hraničné prípady spôsobovali výrazné spomalenie zbierania typov alebo dokonca jeho zlyhanie.

Na základe tejto analýzy sme preto navrhli dve ďalšie riešenia založené na myšlienke **kooperatívneho módu**. Skombinovali sme pôvodný prístup fuzzera s našou novou myšlienkou *zbierania typov počas behu programu* tak, aby sme eliminovali nevýhody oboch postupov. Ukázalo sa, že nie je nutné zbierať všetky typy a niektoré odvodzovania je možné nechať na pôvodné riešenie bez straty presnosti typu.

Na záver tejto kapitoly sme spustili fuzz testovanie vo všetkých naimplementovaných variantoch v podobnej konfigurácii ako ho spúšťajú výskumníci v oblasti počítačovej bezpečnosti. Tu sme ukázali, že sme síce nedokázali zväčšiť pomer vygenerovaných korektných programov, ale dokázali sme pomocou presnejších typov generovať viac zaujímavejších programov v rovnakom časovom intervale. Pôvodné odvodzovanie typov totiž bývalo často konzervatívne a nedokázalo objavovať všetky vlastnosti

a metódy objektov v programe. Naše fuzz testovanie sme spúšťali na enginoch JavaScriptCore a Spidermonkey. V oboch prípadoch sa nám za rovnaký čas podarilo otestovať o **2** percentuálne body viac kódu testovaného enginu.

V poslednej kapitole sme si ukázali a naimplementovali rôzne časové a pamäťové optimalizácie. Podarilo sa nám naimplementovať minifikačný mód, ktorým sme dokázali zmenšiť veľkosti generovaných programov o približne **10 %**. Ďalej sme zaviedli nový typ **iterable** do typového systému, pomocou ktorého fuzzer dokázal predchádzať generovaniu zbytočne chybných programov.

Pozreli sme sa tiež na efektívne ukladanie zozbieraných typov. Optimalizáciou používania pamäte sa nám podarilo **200-násobne** zmenšiť počet vytvorených inštancií triedy *TypeExtension* v naplnenom korpuse.

Podobne sme optimalizovali aj samotnú časovú efektivitu odpovedania na dotazy o typoch jednotlivých premenných v danom čase. Takto sme ušetrili čas aj tým, že sme na rovnakom programe nemuseli odvodzovať typy viac krát. Ako poslednú optimalizáciu sme spravili zrýchlenie pôvodného statického odvodzovania. Mierne sme zovšeobecnili koncept tohto odvodzovania, aby sme boli schopný ho použiť aj na konštrukty, ktoré možno budú doimplementované do medzijazyka FuzzIL v budúcnosti. Rovnako sme ho aj zrýchlili. Po implementovaní všetkých vylepšení tejto kapitoly sme dokázali odmerať na referenčných skúškach zrýchlenie až na úrovni **50 %**. Konkrétne medián rýchlosti dokončenia referenčných skúšok sme zmenšili z **30,67s** na **16,44s**.

Ako sme už spomínali v podkapitole 3.2.2 nie všetky problémy sa nám podarilo vyriešiť. Na dlhších behoch sme mali možnosť vidieť že niečo menej ako **1 %** programov zostalo bez typovej informácie, ktorú bolo možné získať počas behu programu (v kooperatívnom móde sme boli schopný dostať aspoň typ zo statického odvodzovania pre tieto programy). Čo nám necháva priestor na zlepšenie. No je na zváženie koľko úsilia by takéto zlepšenie stálo, keďže nám zostali už len zložitejšie prípady. Zároveň by sme tým dostali lepšiu typovú informáciu len pre málo ďalších programov a výkonnosť celkového testovania by sa zrejme výrazne nezlepšila.

Existujú však aj iné prístupy ako zväčšiť počet zaujímavých vygenerovaných programov za fixný čas. Jedným z problémov fuzzera je, že pokiaľ chce vygenerovať niektorý zaujímavý konštrukt, tak často nemá k dispozícii premenné s potrebnými typmi. Zároveň si takéto premenné nevie vygenerovať, poprípade by takéto generovanie príliš zväčšilo veľkosť programu. Preto by bolo možné v budúcnosti naimplementovať zhromažďovanie zaujímavých úryvkov kódu pre generovanie konkrétneho typu počas fuzz testovania. Takto by fuzzer pri generovaní nemusel skúšať vygenerovať ďalšie pomocné konštrukty, ale stačilo by sa pozrieť do takejto databázy.

Literatúra

[1] Ecmascript language specification. `http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf`. Accessed: 2020-10-31.

[2] Usage statistics of javascript as client-side programming language on websites. `https://w3techs.com/technologies/details/cp-javascript`. Accessed: 2020-10-31.

[3] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra lightweight javascript engine for internet of things. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 19–20, 2015.

[4] Samuel Groß. Fuzzil: Coverage guided fuzzing for javascript engines, 2018.

[5] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2613–2630, 2020.

[6] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[7] Matt Molinyawe, Abdul-Aziz Hariri, and Jasiel Spelman. From browser to system compromise.

[8] DongHyeon Oh, JaeSeung Choi, and SangKil Cha. Semantics-preserving mutation-based fuzzing on javascript interpreters. *Journal of the Korea Institute of Information Security & Cryptology*, 30(4):573–582, 2020.

[9] Marin Šilić, Jakov Krolo, and Goran Delač. Security vulnerabilities in modern web browser architecture. In *The 33rd International Convention MIPRO*, pages 1240–1245. IEEE, 2010.

[10] Perry Wagle, Crispin Cowan, et al. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–255. Citeseer, 2003.

Príloha

USB

K dispozícií na priloženom USB sa nachádza naklonovaný Git repozitár, ktorý je možné nájsť aj na Githube[11].

Tento repozitár obsahuje kompletný spustiteľný kód fuzzera Fuzzilli. Podrobnosti pre spustenie a používanie je možné nájsť v *README.md*.

V rámci našej práce sme doňho doprogramovali viacero funkcionalít. Hlavnú funkcionalitu používania nového zdroja typových informácií počas behu programu je možné povoliť argumentom príkazového riadku *--collectRuntimeTypes*.

Časti naimplementované v tejto práci je možné rozpoznať podľa autora záznamu (angl. „commit“) alebo vyfiltrovať na samotnom Githube[12].

[11] `https://github.com/googleprojectzero/fuzzilli`

[12] `https://github.com/googleprojectzero/fuzzilli/pulls?q=is%3Apr+author%3Asamo98+`