Fuzz tesztelés és teszteset redukálás

Ph.D. értekezés tézisei

Hodován Renáta

Témavezető: Dr. Gyimóthy Tibor, egyetemi tanár

Informatika Doktori Iskola

Szoftverfejlesztés Tanszék
Természettudományi és Informatikai Kar
Szegedi Tudományegyetem

Szeged, 2019

Bevezetés

A technológia fejlődésével a különböző szoftverek és szoftver rendszerek egyre nagyobb teret szereznek a mindennapi életünkben. Jelen vannak az otthonainkban, szórakoztatják a gyerekeinket, orvosokat segítenek operáció közben, gyártósorokat irányítanak, földrengéseket jeleznek előre és ez még csak a jéghegy csúcsa. Habár ezen rendszerek léte nagyban megkönnyíti életünket, ugyanakkor potenciális fenyegetést is jelentenek, mivel érzékeny személyes információkhoz férhetnek hozzá. Ennélfogva biztonsági tesztelésük kiemelten fontos mind a felhasználóknak, mind pedig az alkalmazásokat fejlesztő ipari szereplőknek.

Amikor biztonsági tesztelésre terelődik a szó, akkor a véletlenszerű vagy fuzz tesztelés [17] témaköre is hamar előkerül. A fuzz tesztelés egy népszerű automatikus negatív tesztelési módszer, amely tesztek nagy számú és véletlenszerű generálásán alapszik. Az így generált teszteket bementként adva a tesztelés alatt álló szoftvernek esélyünk nyílik valamilyen nem várt viselkedés felfedezésére. A fuzzolást szokás önálló tesztelési módszerként, vagy egy tényleges támadás belépési pontjának megtalálásához is használni. Népszerűségét elsősorban a hatékony hibafelderítési képességének és automatizálhatóságának köszönheti, amely egyben gazdaságossá is teszi.

Egy fuzzer keretrendszer általában három fő komponensre osztható: egy teszt generátorra, amely előállítja a teszteseteket, egy végrehajtó funkcionalitásra, amely képes futtatni és monitorozni a tesztelt alkalmazást, illetve egy teszteset redukáló megoldásra, amely megtalálja a hibát előidéző teszteset azon részét, amely ténylegesen felelős a hibáért. Jelen disszertáció a hibát kiváltó tesztesetek előállítására és azok minimalizálására fókuszál.

Elsődleges fuzzolási célpontként a JavaScript értelmező motorokat választottam. A JavaScript hagyományosan a web böngészők programozási nyelve, amely az utóbbi évtizedben a legnépszerűbb programozási nyelvvé nőtte ki magát [3, 22]. A nyelv széleskörű elterjedtsége miatt a JavaScript motorok helyességének biztosítása – mind funkcionális, mind biztonsági szempontból – létfontosságú. Jelen disszertáció első részében egy olyan új fuzzolási megközelítést fogok bemutatni, amely JavaScript motorok tesztelésében lényegesen jobb kódlefedettségi eredményeket ér el, mint a jelenlegi legmodernebb megoldás, illetve amelynek prototípus implementációja már több, mint 100 egyedi hibát talált különböző valós JavaScript motorokban.

Egy hatékony tesztgenerátor segít a hibák megtalálásában és a hibát indukáló tesztesetek azonosításában. Többnyire azonban ezen teszteseteknek egy kis része is elegendő a hiba reprodukálásához. A felesleges részek elhagyása segít a hiba megértésében, ami pedig elősegíti a hiba okának mielőbbi javítását. Ezen részek megtalálása gyakran időigényes feladat még valós életbeli teszteknél is, nem beszélve a fuzzerek által véletlenszerűen generált, nagy méretű bemenetekről. Éppen

ezért a tesztesetek méretének csökkentésére több automatizált megoldást is kifejlesztettek az elmúlt évtizedben.

A disszertáció második részében két széles körben elterjedt redukáló algoritmust, a Delta Debuggingot és Hierarchikus Delta Debuggingot fogom megvizsgálni, meghatározom a gyengeségeiket és fejlesztéseket javaslok hozzájuk. A fejlesztések két nyílt forrású redukáló alkalmazást eredményeztek, amelyeket nem csak a kísérleti kiértékelésekben, de valós projektekben is aktívan használunk. Az eredmények megmutatták, hogy a megoldásaink kisebb reprodukciós teszteket állítanak elő rövidebb idő alatt, mint a jelenlegi modern megoldások.

A disszertáció öt tézispontot fogalmaz meg, melyek a következők:

1. JavaScript motorok Prototípus Gráf alapú fuzz tesztelése
2. A Delta Debugging algoritmus elemeire bontása
3. Nyelvtanok hatásának elemzése a Hierarchikus Delta Debugging algoritmusra
4. A Hierarchikus Delta Debugging javítása fatranszformációkkal
5. Coarse Hierarchikus Delta Debugging

JavaScript motorok fuzz tesztelése

A fuzzolás vagy a véletlenszerű tesztelés egy népszerű tesztelési technológia, amely nagy mennyiségű teszteset előállításával kecsegtet kevés munka befektetése árán. Emellett a véletlenszerűségnek köszönhetően gyakran olyan extrém tesztek generálására is képes, amelyek könnyedén elkerülnék egy emberi tesztelő figyelmét.

Jelen disszertációban a JavaScript motorok fuzz tesztelésére fókuszálunk, különös tekintettel azok típus API-jára, amelyet az egyes végrehajtó motorok biztosítanak a felhasználóiknak.

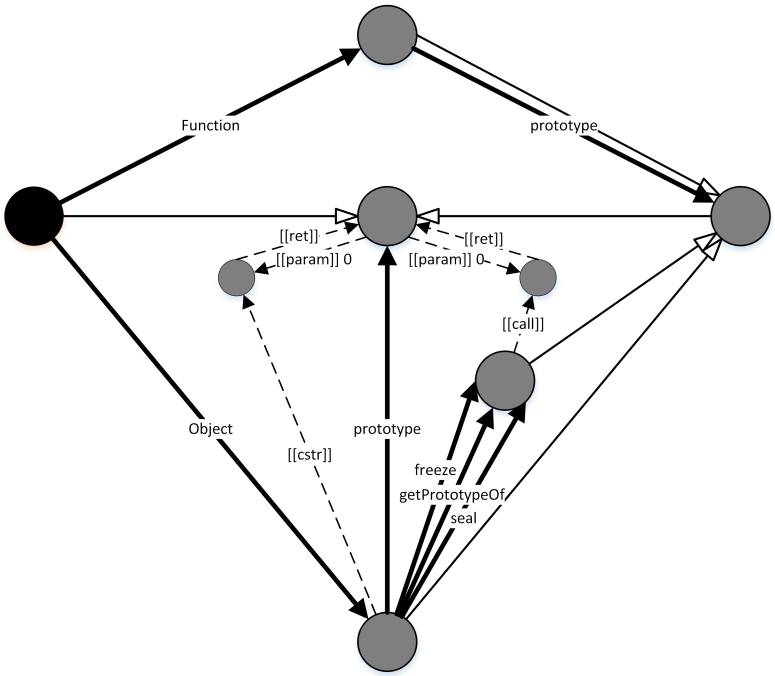
1. JavaScript motorok Prototípus Gráf alapú fuzz tesztelése

Először is definiáljuk a Prototípus Gráfnak nevezett modellt, amely képes a JavaScript nyelv objektum-alapú típus rendszerének ábrázolására.

A Prototípus Gráf nem más, mint *típus* és *szignatúra* csomópontok halmaza, melyeket hatféle éllel köthetünk össze: *tulajdonság*, *prototípus*, *hívás*, *konstruktor*, *paraméter* és *visszatérési* éllel. A *típus* csomópontok JavaScript 'típusokat', vagy tulajdonságaik alapján hasonlónak ítélt objektumokat ábrázolnak. A *szignatúra* csomópontok pedig a hívható típusok, azaz a függvények és konstruktorok lehetséges szignatúráit írják le. A *prototípus* és *tulajdonság* élek *típus* csomópontokat, míg a többi élfajta *típus* és *szignatúra* csomópontokat kötnek össze. A JavaScript objektumok tulajdonságait vagy mezőit a *tulajdonság* élek címkéje tárolja, míg a függvények paramétereinek sorrendjét a *paraméter* élek címkéje.

Adott egy, a fentiek szerint megkonstruált gráf, amellyel a célunk olyan JavaScript kifejezések generálása, amelyek API objektumok függvényeit hívják meg az elvárt típusoknak megfelelő paraméterekkel. Ez elérhetjük a Prototípus Gráfon alkalmazott véletlen sétával: „Először haladjunk előrefele a *prototípus*, *tulajdonság*, *konstruktor*, *hívás* és *visszatérés* élek mentén, majd haladjunk visszafelé a *paraméter* és *prototípus* élek mentén, majd kezdjük újra ..." A Prototípus Gráf formális definíciója, valamint a rajta definiált véletlen séta formalizmusa a disszertáció 4. fejezetében érhető el.

Szemléltetésképp készítettünk egy Prototípus Gráfot az ECMAScript 5.1 szabvány egy részletéről. Az előállt gráf az 1. ábrán látható, ahol a nagy pontok a *típus*, míg a kis pontok a *szignatúra* csomópontokat jelölik. Az egyetlen fekete csomópont az ábra bal oldalán a JavaScript nyelv egy kitüntetett, úgynevezett globális objektumának típusát jelöli. Ezen megkülönböztetésnek jelen ábrán azonban csak reprezentációs célja van. A címkézett vastag vonalak jelképezik a *tulajdonság* éleket, a vékony vonalak üres nyilakkal a végükön jelölik a *prototípus* éleket, míg a szaggatott vonalak duplán zárójelezett címkékkel ábrázolják a *konstruktor*, *hívás*, *paraméter* és *visszatérési* éleket.



1. ábra. Példa Prototípus Gráf az ECMAScript 5.1 szabvány egy részletével [2, 15.2,15.3 Szekciók].

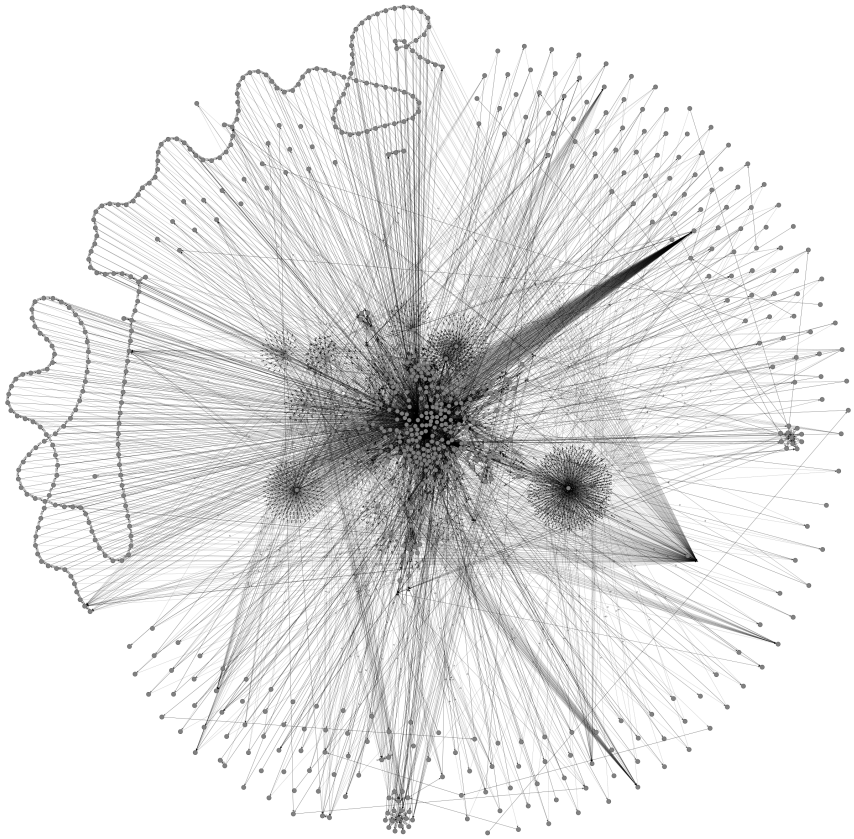
Az alábbi két kifejezés a módszer tesztgeneráló képességét szemlélteti a fenti, kézzel készített gráfot modellként használva:

- `this.Object.getPrototypeOf(this.Function.prototype)`,
- `new (this.Object)(this)`.

A gráfok kézzel való konstruálásának elkerülése céljából kidolgoztunk és implementáltunk két automatizált technológiát: a *felfedező* stratégiát és a létező tesztekből *tanuló* finomhangolást. Mindkét megközelítés a JavaScript nyelv önelemző képességén alapszik, mivel nemcsak képesek vagyunk futásidőben meghatározni minden kifejezés típusát, de fel is tudjuk sorolni a tulajdonságaikat, bejárhatjuk minden objektum prototípus láncát, valamint minden függvénytől le tudjuk kérdezni az elvárt paramétereinek a számát.

A módszer kiértékeléséhez a JavaScriptCore-t (vagy röviden jsc-t) választottuk, ami az Apple Safari böngésző WebKit [1] nevű megjelenítő motorjának a JavaScript végrehajtója. A 2. ábra a JavaScriptCore-ból automatikusan kinyert Prototípus Gráfot ábrázolja.

A kiértékelés alapjául a jsfunfuzz [21] nevű nyílt forrású eszközt vettük, amelyet a Mozilla fejleszt és használ a Firefox [20] böngésző SpiderMonkey nevű JavaScript



2. ábra. A jsc Prototípus Gráfja a tesztekből való finomhangolás után.

végrehajtójának tesztelésére, és amivel már több száz hibát találtak az évek során. Minden fuzzerrel végül 50.000 JavaScript kifejezést generáltunk, lefuttattuk őket a JavaScriptCore-ral, majd összehasonlítottuk az eredményeket mind kódlefedettség, mind hibadetektálási képesség szempontjából.

A 1. táblázat mindhárom tesztgeneráló megközelítés sor lefedettség eredményeit mutatja modulokra bontva és összesítve egyaránt. Az eredmények azt mutatják, hogy *felfedező* stratégia nem teljesít olyan jól, mint a jsfunfuzz a teljes kódlefedettség szempontjából (23.31% a 37.25% ellenében), de lényegesen jobb (44.13%), és a jsfunfuzznál magasabb lefedettséget kapunk, ha finomhangoljuk a gráfunkat tesztekből kinyert szignatúra információkkal.

Három modult érdemes külön kiemelni, nevezetesen a *runtime*-ot, a *yarr*-t és a *jsc.cpp*-t. Az első a JavaScript nyelv alapvető funkcionalitásait tartalmazza (úgy mint a beépített függvényeket), a második pedig a reguláris kifejezéseket kiértékelő motort. A harmadik modul, ami lényegében csak egy fájl, a fő parancs-

1. táblázat. A *jsc* kódlefedettsége 50,000 generált kifejezés lefuttatása után (a jsfunfuzz esetében 50,320 kifejezés után).

Modul	**Összes Sorok**	**Lefedett sorok**					
		felfedező		**finomhangolt**		**jsfunfuzz**	
API	1698	9	0.53%	9	0.53%	9	0.53%
DerivedSources	4546	148	3.26%	167	3.67%	312	6.86%
assembler	2997	1046	34.90%	2037	67.97%	2054	68.54%
bindings	165	0	0.00%	0	0.00%	0	0.00%
builtins	96	63	65.63%	63	65.63%	62	64.58%
bytecode	8578	1650	19.24%	4196	48.92%	3320	38.70%
bytecompiler	4656	2344	50.34%	2372	50.95%	2887	62.01%
debugger	713	3	0.42%	3	0.42%	3	0.42%
dfg	29959	27	0.09%	11019	36.78%	9403	31.39%
disassembler	1033	3	0.29%	3	0.29%	3	0.29%
heap	4221	2517	59.63%	2671	63.28%	2373	56.22%
inspector	3594	0	0.00%	0	0.00%	0	0.00%
interpreter	1336	594	44.46%	664	49.70%	648	48.50%
jit	8919	814	9.13%	4852	54.40%	4345	48.72%
jsc.cpp	926	507	54.75%	519	56.05%	240	25.92%
llint	840	344	40.95%	424	50.48%	451	53.69%
parser	6586	3618	54.93%	3801	57.71%	4400	66.81%
profiler	788	4	0.51%	4	0.51%	4	0.51%
runtime	27112	12115	44.69%	15101	55.70%	10648	39.27%
tools	534	13	2.43%	13	2.43%	13	2.43%
yarr	3538	486	13.74%	1879	53.11%	856	24.19%
TOTAL	112835	26305	23.31%	49797	44.13%	42031	37.25%

sori funkcionalitást kódolja, ami egy hagyományos beágyazó alkalmazás abban az értelemben, hogy kiegészítő, nem hagyományos rutinokat köt be a JavaScript környezetbe. Ez azt is jelenti, hogy ezek azok a modulok, amelyek a JavaScript API-t szolgáltató natív kódot tartalmazzák, vagyis amik jelen kutatás fókuszai. Ahogy a táblázat mutatja, már az egyszerű felfedező megközelítés túlszárnyalja a jsfunfuzz lefedettségét a háromból két modul esetén, a szignatúrákkal bővített variáns pedig mindhárom esetben a legjobb eredményt hozza.

Mivel a fuzzolásnak nem csak a kódlefedettség maximalizálása a célja, hanem valós hibák felderítése, ezért összehasonlítottuk a három megközelítést a talált hibák számának szempontjából is. Az 2. táblázat áttekintést ad a detektált hibák számáról, illetve mivel több teszt is kiválthatta ugyanazt a hibát, ezért külön feltüntetjük az egyedi hibák számát is. Meglepetésre mindkét gráf alapú megközelítés ugyanazokat a hibákat találta meg. Ami viszont azt jelenti, hogy még a gyengébb

2. táblázat. Talált hibák száma a *jsc* motorban.

	felfedező	**finomhangolt**	**jsfunfuzz**
összes hiba	1326	1445	4
egyedi hiba	6	6	2

kódlefedettséget elért felfedező technika is több hibát talált a jsfunfuzznál.

Tézispontok és a szerző hozzájárulása az eredményekhez

1. JavaScript motorok Prototípus Gráf alapú fuzz tesztelése

A szerző, a társszerzőjével közösen, definiálta és formalizálta a Prototípus Gráf nevű modellt, amellyel tetszőleges JavaScript motor típus API-ját képes leírni. A modell a JavaScript nyelv objektum-alapú típusrendszere mellett azt is tartalmazza, hogy a motor által biztosított beépített függvények és metódusok milyen szignatúrát várnak el. A szerző megmutatta hogyan nyerhető ki a fent említett gráf automatikusan egy tetszőleges JavaScript motorból, illetve hogyan finomítható az létező tesztek elemzésével.

A szerző kidolgozott egy algoritmust, amely a fenti modellt használva képes véletlenszerű JavaScript kifejezéseket generálni és azokkal stressz-tesztelni a JavaScript futtató motort. A szerző továbbá implementált egy prototípus eszközt, amely képes felépíteni az alap modellt, képes azt létező tesztek elemzésével pontosítani, és végül valós motorok fuzz tesztelésére felhasználni. A prototípus eszköz felhasználásával összehasonlította a megközelítést egy másik, korszerű fuzzerrel, illetve reportálta a talált hibákat.

Automatizált teszteset redukció

Közismert tény, hogy nincsen program hiba nélkül. A hibákat – és a hibákat indukáló teszteseteket – felfedezheti a felhasználó, a fejlesztő vagy valamilyen automatizált tesztelő rendszer, mint azt az előző fejezetben láthattuk. Többnyire azonban a hibát okozó teszteseteknek egy kis része is elegendő a hiba reprodukálásához. A felesleges részek lehagyása segít a hiba megértésében, ami pedig elősegíti a hiba mielőbbi javítását. Ezen részek meghatározása azonban komoly kihívást jelenthet még valós életbeli helyzetekben is, nem beszélve fuzzer által véletlenszerűen generált tesztek minimalizálásáról. Ez a tény motiválja a kutatókat, hogy hatékony automatizált megoldásokat találjanak a minimalizálások elvégzésére, amelyek a lehető leggyorsabban a lehető legkisebb reprodukáló tesztesetet állítják elő. Ezen megközelítések közül a legismertebb Zeller és Hildebrandt Delta Debugging [24, 4, 25] nevű szintaxis-független megközelítése, illetve ennek a szintaxis-követő változata a Hierarchikus Delta Debugging [18].

2. A Delta Debugging algoritmus elemeire bontása

Az 1. definíció Zeller Delta Debugging algoritmusának eredeti, rekurzív matematikai formuláját tartalmazza.

Definíció 1 (Zeller and Hildebrandt's)
Adott $test$ és $c_{\text{✗}}$ úgy, hogy $test(\emptyset) = \text{✓} \wedge test(c_{\text{✗}}) = \text{✗}$ teljesül. A cél egy olyan $c'_{\text{✗}} = ddmin(c_{\text{✗}})$ megtalálása, amelyre $c'_{\text{✗}} \subseteq c_{\text{✗}}$ és $test(c'_{\text{✗}}) = \text{✗}$ teljesül, és amelyre $c'_{\text{✗}}$ 1-minimális. A *minimalizáló Delta Debugging algoritmus* $ddmin(c)$ ekkor

$$ddmin(c_{\text{✗}}) = ddmin_2(c_{\text{✗}}, 2) \text{ ahol}$$

$$ddmin_2(c'_{\text{✗}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{ha } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \text{✗} \\ & \text{(„redukálás részhalmazra”)} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{különben ha } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✗} \\ & \text{(„redukálás komplementerre”)} \\ ddmin_2(c'_{\text{✗}}, \min(|c'_{\text{✗}}|, 2n)) & \text{különben ha } n < |c'_{\text{✗}}| \\ & \text{(„felbontás növelése”)} \\ c'_{\text{✗}} & \text{egyébként („kész”).} \end{cases}$$

ahol $\nabla_i = c'_{\text{✗}} - \Delta_i$, $c'_{\text{✗}} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$ úgy, hogy minden Δ_i páronként diszjunkt, és $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✗}}|/n$ teljesül. A $ddmin_2$ rekurziójának invariáns tulajdonsága (és egyben előfeltétele), hogy $test(c'_{\text{✗}}) = \text{✗} \wedge n \leq |c'_{\text{✗}}|$.

Az első dolog, amit észrevehetünk, hogy habár az összes elérhető implementáció szekvenciális ciklusokkal valósítja meg a $ddmin_2$ algoritmus „redukálás részhalmazra” és „redukálás komplementerre” lépéseit, a párhuzamosítás lehetősége ott volt

már az eredeti formalizmusban is, mivel a $\exists i \in \{1, \ldots, n\}$ kifejezés nem specifikálja, hogy hogyan találjuk meg i-t. Mivel valódi bemenetek esetén n tetszőlegesen nagyra nőhet és mivel a *test* művelet gyakran költséges, ezért javasoljuk a párhuzamosítási potenciál kihasználását és a $ddmin_2$ átírását párhuzamos ciklusok használatára.

Azt is megfigyelhetjük, hogy habár a $ddmin_2$ esetenként definiált, ezek az esetek nem függetlenek, hanem a *különben ha* kifejezésnek megfelelően egymás után végrehajtandók. Azonban felismertük, hogy ez a szekvencialitás *nem* szükséges. Több olyan Δ_i és ∇_i teszteset is létezhet, amely reprodukálja az eredeti viselkedést. Ez azt jelenti, hogy bármelyiket választhatjuk közülük, nem kell preferálnunk a részhalmazokat a komplementerekkel szemben hogy végül 1-minimális eredményt kapjunk.

Ezt a felismerést azonban egészen addig nem kamatoztathatjuk, amíg az implementációnk szekvenciális. Ezért javasoljuk a két redukálási eset összevonását és mind a részhalmazra, mind pedig a komplementerre való redukáláshoz tartozó tesztek egy lépésben való futtatását. Ilyen módon az algoritmusnak nem kell várnia a részhalmazra való redukálás tesztjeinek befejezésére, hanem rögtön elkezdheti a komplementer tesztek futtatását, amint azt a számítási kapacitás lehetővé teszi. A 1. algoritmus az összes fent említett fejlesztést tartalmazza.

Továbbá azt is megfigyelhetjük, hogy a „redukálás részhalmazra" lépés nem is szükséges az 1-minimalitás biztosításához. Az csupán egy mohó kísérlet, hogy egy részhalmaz kivételével mindent eltávolítsunk a tesztből ahelyett, hogy egyesével próbálkoznánk a „redukálás komplementerre" lépés alkalmazásával. Azonban számos olyan bemeneti formátum létezik, amelyek szinte mindig szintaktikailag hibásak lesznek és nem fogják reprodukálni az elvárt viselkedést, ha csak egy „középső" részhalmazt tartunk meg belőlük. Az ilyen típusú bemenetekre a „redukálás komplementerre" eset lényegesen többször vezet eredményre, míg a „redukálás részhalmazra" talán soha. Ezért úgy gondoljuk, hogy megéri kísérletezni a redukálási esetek átrendezésével, illetve a „redukálás részhalmazra" eset teljes elhagyásával, mivel az potenciálisan csak pazarolja a számítási kapacitást.

A minimalizáló Delta Debugging algoritmus vizsgálata során implementáltunk egy prototípus eszközt [8], amelyben megvalósítottuk a vázolt fejlesztéseket. Kísérleteinket négy mesterséges teszteseten, illetve három valós életbeli teszten végeztük, melyek összeomlást okoztak két népszerű böngészőben. Minden tesztesetet minden algoritmus variánssal és 12 különböző párhuzamosítási beállítással leredukáltunk, amely a párhuzamosítás nélküli szinttől egészen a 64-szeres párhuzamosításig terjedt. Az összesen 1098 sikeres minimalizálás igazolta, hogy a Delta Debugginghoz javasolt minden fejlesztés sebességnövekedést eredményezett, ami a legjobb esetben 75–80%-kal csökkentette a redukálás futásidejét.

1. Algoritmus. A párhuzamos *ddmin*$_2$ pszeudókódja kombinált redukáló esetekkel

```
1  procedure ddmin_2^κ(c'✗, n)
2  begin
3    while true do begin
4      (* reduce to subset or complement *)
5      found = 0
6      parallel forall i in 1..2n do
7        if 1 ≤ i ≤ n then
8          if test(Δ_i^(c'✗,n)) = ✗ then begin
9            found = i
10           parallel break
11         end
12       else if n + 1 ≤ i ≤ 2n then
13         if test(∇_(i−n)^(c'✗,n)) = ✗ then begin
14           found = i
15           parallel break
16         end
17     if 1 ≤ found ≤ n then begin
18       c'✗ = Δ_found^(c'✗,n)
19       n = 2
20       continue
21     end else if n + 1 ≤ found ≤ 2n then begin
22       c'✗ = ∇_(found−n)^(c'✗,n)
23       n = max(n − 1, 2)
24       continue
25     end
26     (* increase granularity *)
27     if n < |c'✗| then begin
28       n = min(|c'✗|, 2n)
29       continue
30     end
31     (* done *)
32     break
33   end
34   return c'✗
35 end
```

3. Nyelvtanok hatásának elemzése a Hierarchikus Delta Debugging algoritmusra

A másik népszerű minimalizálási megközelítés a Hierarchikus Delta Debugging (vagy HDD) [18, 19], amely a fenti DD algoritmust alkalmazza elemzési fák szint-

jeire. Ahogy már a HDD szerzői is felismerték, a hagyományos környezetfüggetlen nyelvtanok – amelyek rekurzív szabályokkal ismernek fel lista-szerű konstrukciókat – erősen kiegyensúlyozatlan elemzési fákat eredményezhetnek. Ez a tulajdonság nemcsak a szükséges tesztfuttatások számát, de a redukált teszteset méretét is növeli.

Ezért azt javasoljuk, hogy a hagyományos környezetfüggetlen nyelvtanok helyett használjunk *kiterjesztett* környezetfüggetlen nyelvtanokat a bemenetek elemzéséhez és a HDD bemeneti fájának építéséhez. A kiterjesztett környezetfüggetlen nyelvtanok lehetővé teszik, hogy terminálisok és nem-terminálisok felett definiált reguláris kifejezéseket használjunk a szabályok jobb oldalán (azaz a választás, a csoportosítás és a mennyiségjelző (?, *, +) operátorok mind alkalmazhatóak). Habár a kiterjesztett környezetfüggetlen nyelvtanok pontosan ugyanazokat a nyelveket ismerik fel, mint a hagyományos környezetfüggetlen nyelvtanok, a mennyiségjelzők lehetővé teszik a rekurzív szabályok elhagyását, ami sokkal kiegyensúlyozottabb elemzési fákat eredményez.

Hogy a kiterjesztett környezetfüggetlen nyelvtanok HDD-re gyakorolt hatását valós helyzetekben is megvizsgálhassuk, implementáltuk a fent vázolt javaslatokat [9]. A kísérleti kiértékelés bebizonyította, hogy a kiterjesztett környezetfüggetlen nyelvtanok hatására a HDD lényegesen (25–40%-kal) kisebb eredményt produkált, mint a hagyományos környezetfüggetlen nyelvtanokat használó eredeti megközelítés.

4. A Hierarchikus Delta Debugging javítása fatranszformációkkal

A kiterjesztett környezetfüggetlen nyelvtanok használatának ellenére is vannak olyan esetek, amikor a HDD nem dolgozik optimálisan. Ennek egyik lehetséges oka, hogy az elemzési fák tartalmazhatnak lineáris komponenseket, azaz olyan útvonalakat, ahol minden csomópontnak maximum egy gyereke van. Az ilyen lineáris komponensek felesleges tesztfuttatásokat indukálnak, mivel ha a HDD már megvizsgálta azok legfelső csomópontját és úgy döntött, hogy megtartja azt, akkor a komponens többi pontján sem fog másképp dönteni, ahogy halad lefelé a szinteken. Ha a csomópontok minimális cseresztringje megegyezik, akkor a fenti állítás fordítottja is igaz: ha a részgráf valamely csomópontját töröljük is (azaz kiváltjuk a cseresztringjével [19]) a végeredmény nem fog változni. Ez azt is jelenti, hogy ha az ilyen lineáris komponenseket egyetlen csomóponttal helyettesítjük, azzal összenyomhatjuk a fa magasságát és csökkenthetjük a futtatandó tesztek számát.

A *squeezeTree* optimalizációt, amelyet a fa gyökérpontjára alkalmazunk a HDD tényleges meghívása előtt, a 2. algoritmusban formalizáltuk.

2. Algoritmus. A „fa-összenyomó” algoritmus

```
1  procedure squeezeTree(node)
2  begin
3    if not isToken(node) then begin
4      forall i in 1..children(node) do
5        child(node, i) ← squeezeTree(child(node, i))
6      if children(node) = 1 and Φ(node) = Φ(child(node, 1)) then
7        return child(node, 1)
8    end
9    return node
10 end
```

3. Algoritmus. A „törölhetetlen tokenek elrejtése” algoritmus

```
1  procedure hideUnremovableTokens(node)
2  begin
3    if isToken(node) then
4      if text(node) = Φ(node) then
5        markAsRemoved(node)
6    else
7      forall i in 1..children(node) do
8        hideUnremovableTokens(child(node, i))
9  end
```

A másik érdekes dolog, amit a HDD által minimalizált fákat vizsgálva megfigyelhetünk, hogy habár néhány csomópont meg van jelölve törlésre, azok mégis megjelennek a kimenetben. Ennek oka, hogy néhány terminális esetében a legkisebb szintaktikailag megengedett cseresztring megegyezik a terminális tartalmával. Következésképp nem számít, hogy a DD miként dönt, megtartja-e a tokent (azaz a tényleges tartalma kerül-e a kimenetbe) vagy sem (azaz a minimális cseréjét használja): ugyanazt a tesztet fogja eredményezni. Ennek ellenére a DD meg fogja próbálni megtartani és levenni is ezeket, ami felesleges tesztfuttatásokat eredményez.

Ha azonban ezeket az „törölhetetlen” tokeneket már egy előkészítő lépésben, a HDD alkalmazása előtt töröltnek jelöljük meg, akkor elrejtjük őket a *ddmin* elől. Az optimalizáció a *hideUnremovableTokens* algoritmusban került formalizálásra, amit a 3. algoritmus szemléltet.

Ezen ötletek implementálásra kerültek a korábban említett prototípus eszközbe. A kiértékeléseink alapján ezen optimalizációk együttesen a legjobb esetben több

4. Algoritmus. A Coarse HDD algoritmus

```
1  procedure coarseHDD(input_tree)
2      level ← 0
3      nodes ← tagNodes(input_tree, level)
4      while nodes ≠ ∅ do
5          nodes ← filterEmptyPhiNodes(nodes)
6          if nodes ≠ ∅ then
7              minconfig ← ddmin(nodes)
8              prune(input_tree, level, minconfig)
9          end if
10         level ← level + 1
11         nodes ← tagNodes(input_tree, level)
12     end while
13 end procedure
```

mint ötszörös gyorsulást eredményeztek.

5. Coarse Hierarchikus Delta Debugging

Redukálás szempontjából a legnagyobb haszon természetesen a részfák teljes törlésével érthető el, míg a fa azon részei, melyek tényleges törlését a szintaxis nem teszi lehetővé, kevésbé érdekesek. Ezért előállítottuk a HDD egy új variánsát, a Coarse Hierarchikus Delta Debuggingot (azaz a durva felbontású HDD-t), amely ugyanúgy bejárja a fa szintjeit, mint a hagyományos HDD, viszont nem veszi figyelembe azokat a csomópontokat, amelyek cseresztringje nem üres. Abban az esetben, ha egy szinten nincs csomópont üres cseresztringgel, akkor az algoritmus az adott szinten nem is végez tesztelést és a fa nem változik. Az algoritmus pszeudókódja a 4. algoritmusban található, ahol minden kisegítő függvény megegyezik az eredeti publikációban definiáltakkal [18], kivéve a *filterEmptyPhiNodes*-t, amely a fent vázolt szelekciót végzi el.

Mivel a Coarse HDD csak azokat a csomópontokat veszi figyelembe, melyek cseresztringje üres, így a megközelítés hatékonysága erősen függ a nyelvtantól, amely az elemzési fákat előállítja. Azon fák, amelyeket olyan nyelvtan állított elő, ami az ismétléseket rekurzív szabályokkal írja le a mennyiségjelzők helyett, sokkal kevesebb törölhető részfát tartalmaznak, ezáltal a Coarse HDD kevésbé hatékony rajtuk. A probléma megoldásának egy lehetséges módja a nyelvtan manuális újraírása, hogy az minden lehetséges helyen mennyiségjelzőket alkalmazzon. Azonban minden használt nyelvtan átírása sok manuális munkát eredményezne. A másik lehetséges megoldás egy olyan automatikus transzformáció alkalmazása a bemeneti fára, amely átkonvertálja azt, mintha nemrekurzív nyelvtan állította volna elő.

5. Algoritmus. A fa lapító algoritmus

```
1  procedure flattenTreeRecursion(node)
2      forall child in children(node) do
3          flattenTreeRecursion(child)
4      end forall
5      num ← |children(node)|
6      if num > 1 then
7          if name(children(node)[1]) = name(node) then
8              left ← children(node)[1]
9              right ← children(node)[2..num]
10             children(node) ← children(left) + emptyPhiNode(right)
11         elif name(children(node)[num]) = name(node) then
12             left ← children(node)[1..num−1]
13             right ← children(node)[num]
14             children(node) ← emptyPhiNode(left) + children(right)
15         end if
16     end if
17 end procedure
```

Ez az algoritmus megkeresi a bal- és jobbrekurzív fa konstrukciókat és kilapítja ezeket a „lista elemeket”. Ez a transzformáció olyan fákat eredményez, mintha a listák nem rekurzív szabályokkal, hanem mennyiségjelzővel ellátott kifejezésekkel lettek volna elemezve. Az ötletet a Rekurzív Fák Lapítása algoritmusban formalizáltuk, amely az 5. algoritmusban látható, ahol az *emptyPhiNode* függvény egy új csomópontot készít, amely gyerekeinek az argumentumait állítja be és amely minimális cseresztringként az üres sztringet kapja.

Hogy képet kapjunk a megközelítés előnyeiről és hátrányairól, az algoritmus implementációját különböző teszteseteken értékeltük ki. A Coarse HDD* és a fa lapító algoritmus együttesen elérték a céljukat: átlagosan 58%-kal kevesebb lépésben minimalizálták a teszteseteket, míg ez az érték a legjobb esetben a 79%-ot is elérte. Ez a gyorsulás a legnagyobb tesztünk esetében órákkal gyorsabb redukálást jelentett. A gyorsulás ára a legrosszabb esetben a kimenet méretének 0.36%-kal való növekedése lett a bemeneti méretekhez képest.

Tézispontok és a szerző hozzájárulása az eredményekhez

2. A Delta Debugging algoritmus elemeire bontása

A szerző analizálta az elterjedt Delta Debugging algoritmust, különös tekintettel a *ddmin* variánsára. Felismerte, hogy a *ddmin* komponensei, a részhalmaz és komplementer alapú redukciós lépések, nem szükségszerűen szekvenciálisak: sorrendjük felcserélhető és egyikük el is hagyható anélkül, hogy elveszítenénk az eredeti algoritmus 1-minimalitásra vonatkozó garanciáját. Továbbá a szerző megmutatta, hogy az egyes tesztek futási sorrendje szintén releváns az 1-minimalitás szempontjából. A szerző, közös eredményként a társszerzőjével, elkészítette a *ddmin* új implementációját, amely magában foglalja az imént felsorolt fejlesztéseket. A prototípus implementáció felhasználásával, a szerző széleskörű kiértékelést végzett mesterséges és valós életbeli tesztek minimalizálásával, hogy a gyakorlatban is igazolja a megközelítés hatékonyságát.

3. Nyelvtanok hatásának elemzése a Hierarchikus Delta Debugging algoritmusra

A szerző megvizsgálta a Delta Debugging szintaxis-követő verzióját, nevezetesen a Hierarchikus Delta Debuggingot, amely a *ddmin* algoritmust alkalmazza a elemzési-fák szintjeire. Felismerte, hogy a hagyományos környezetfüggetlen nyelvtanok által épített elemzési fák használata nem optimális ebben a kontextusban, mivel erősen elfajulók lehetnek a rekurzív szabályok használatának következtében. Ezért javasolta a kiterjesztett környezetfüggetlen nyelvtanok által készített elemzési fákat alkalmazni bemenetként. A szerző, közös eredményként a társszerzőjével, elkészítette a vázolt javaslatoknak megfelelő prototípus implementációt és ennek segítségével összehasonlították a Hierarchikus Delta Debugging algoritmus hatásfokát hagyományos és kiterjesztett környezetfüggetlen nyelvtanokkal épített fákon futtatva.

4. A Hierarchikus Delta Debugging javítása fatranszformációkkal

A szerző felismerte, hogy néhány konstrukció a elemzési-fában felesleges tesztfuttatásokat eredményez. Két ilyen konkrét konstrukciót azonosított, amelyekhez két algoritmust dolgozott ki – a „fa összenyomást” illetve a „törölhetetlen tokenek elrejtését” –, amelyeket előfeldolgozó lépésként alkalmazott a bemeneti fára. A szerző, közös eredményként a társszerzőivel, implementálta a transzformációkat a fentebb említett prototípus eszköz részeként és kiértékelte annak hatásait.

5. Coarse Hierarchikus Delta Debugging

A szerző, a társszerzőivel közösen, felismerte, hogy a elemzési-fák nem minden pontja járul hozzá egyformán a redukcióhoz. Közösen felismerték, hogy a legtöbb nyereség azon részfák eltávolításából adódik, amelyek ténylegesen törölhetőek a kimenetből, azaz amelyek minimális helyettesítő kifejezése az üres sztring. Ezen felismerésen alapulva, a szerző elkészítette a „Coarse" (durva felbontású) verzióját a Hierarchikus Delta Debugging algoritmusnak, amely csak a fenti tulajdonsággal rendelkező fapontokat veszi figyelembe a redukció során. A szerző implementálta ezen új variánst a korábban említett prototípus eszközbe és kiértékelte annak hatásfokát.

Összegzés

A disszertáció eredményei öt fő tézispontban foglalhatóak össze, melyek a szoftvertesztelés két témakörét, a fuzz tesztelést és az automatikus teszteset redukciót érintik.

A első terület, és egyben az első tézispont fő eredménye egy olyan új reprezentációs modell bevezetése, amely képes leírni a JavaScript nyelv típusrendszerét, és amely hatékonynak bizonyult fuzz tesztelés szempontjából is.

Az automatikus teszteset redukció területén négy fő eredmény született, melyek négy tézispontba lettek szervezve. Először a népszerű, szintaxis-független minimalizáló Delta Debugging algoritmust értékeltük ki és mutattuk meg róla, hogy a tesztfuttatásai párhuzamosíthatók. Továbbá felismertük, hogy az algoritmus részei felcserélhetők és egyikük el is hagyható. Megmutattuk, hogy ezen változtatások nem sértik az 1-minimalitásra vonatkozó garanciát. Ezután egy másik modern szintaxis-követő redukáló megközelítéshez, a Hierarchikus Delta Debugginghoz fordultunk és megvizsgáltuk, hogy a különböző bemeneti fák milyen hatással vannak az algoritmusra. Az eredmények megmutatták, hogy a kiterjesztett környezetfüggetlen nyelvtanokkal előállított bemeneti fákon sokkal jobban teljesít mind sebesség, mind pedig kimeneti méret tekintetében. A harmadik tézispontban bevezettünk két új fa-transzformációs algoritmust, a „fa összenyomást" és a „törölhetetlen tokenek elrejtését", melyek előkészítő lépésként való alkalmazása jelentős gyorsulást eredményezett. Végül bevezettük a Hierarchikus Delta Debugging algoritmus Coarse variánsát, amely az 1-minimalitás garanciájáért cserébe további sebességnövekedést eredményezett. Az új HDD variáns mellett egy új fatranszformációt is bevezettünk, amit „fa lapításnak" neveztünk el és amely segített a Coarse HDD még hatékonyabbá tételében. Összességében a Coarse HDD beváltotta a hozzá fűzött reményeket: a „fa lapítással" együtt lényegesen gyorsabban szolgáltat eredményt, miközben a kimenet méretének növekedése elhanyagolható.

	[10]	[12]	[11]	[15]	[14]
1.	•				
2.		•			
3.			•		
4.				•	
5.					•

3. táblázat. Kapcsolat az értekezés tézispontjai és a felhasznált publikációk között.

Kifejlesztettünk egy fuzzer keretrendszert, a Fuzzinatort [13, 5], amely többek közt képes integrálni az imént bemutatott technológiákat. A Fuzzinatort nem csak a disszertáció kísérleti kiértékeléseinél alkalmaztuk, hanem évek óta aktívan használjuk valós projektekben. A fent bemutatott JavaScript fuzzer mellett, más generátorokat is integráltunk, úgy mint a saját fejlesztésű Grammarinatort [16, 7] és Generinator:RATS-et [6] vagy a népszerű American Fuzzy Lop [23] (AFL) egyes variánsait. A disszertáció készítésének évei során több, mint 1000 egyedi hibát találtunk, minimalizáltunk és reportáltunk különböző projektekhez.

Végül a 3. táblázat foglalja össze, hogy mely publikációk az értekezés mely tézispontjait fedik le.

Köszönetnyilvánítás

Hosszú út vezetett idáig, de nem egyedül tettem meg. Sok embertől kaptam segítséget, akiknek mindig hálás leszek. Időrendi sorrendben először köszönetet szeretnék mondani középiskolai tanáraimnak, Bali Zsuzsannának és Szőke Imrének, akik meggyőztek róla, hogy egy lány is választhat informatikai pályát. Továbbá hálás vagyok Gyimóthy Tibornak, aki sok évvel ezelőtt meghívott, hogy a Szoftverfejlesztés Tanszéken dolgozzam, és aki idővel a Ph.D. témavezetőm lett. Hálás vagyok neki és Kiss Ákosnak a folyamatos támogatásukért, ami akkor sem hagyott alább, amikor olyan kutatási témát választottam, amelynek nem volt hagyománya a tanszékünkön. Köszönettel tartozom továbbá Lóki Gábornak és Herczeg Zoltánnak, akik segítettek kutatásom gyakorlatban való hasznosításában. Végül, de nem utolsó sorban, köszönöm a családomnak, hogy biztos és nyugodt hátteret biztosítottak tanulmányaimhoz és támogattak a kutatásaim során.

Irodalomjegyzék

[1] Apple Inc. WebKit. A fast, open source web browser engine. `https://webkit.org/`. [Accessed: 2019-03-01].

[2] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* 5.1 edition, 2011.

[3] GitHub Inc. Top languages over time. `https://octoverse.github.com/projects#languages`. [Accessed: 2019-03-01].

[4] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 135–145. ACM, 2000.

[5] Renáta Hodován and Ákos Kiss. Fuzzinator. Random Testing Framework. `https://github.com/renatahodovan/fuzzinator`. [Accessed: 2019-03-01].

[6] Renáta Hodován and Ákos Kiss. Generinator: Random Attributes, Tags & Style. `https://github.com/renatahodovan/generinator-rats`. [Accessed: 2019-03-01].

[7] Renáta Hodován and Ákos Kiss. Grammarinator. ANTLR v4 grammar-based test generator. `https://github.com/renatahodovan/grammarinator`. [Accessed: 2019-03-01].

[8] Renáta Hodován and Ákos Kiss. Picire: Parallel Delta Debugging Framework. `https://github.com/renatahodovan/picire`. [Accessed: 2019-03-01].

[9] Renáta Hodován and Ákos Kiss. Picireny: Hierarchical Delta Debugging Framework. `https://github.com/renatahodovan/picireny`. [Accessed: 2019-03-01].

[10] Renáta Hodován and Ákos Kiss. Fuzzing JavaScript Engine APIs. In *Integrated Formal Methods – 12th International Conference, iFM 2016, Reykjavík, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science (LNCS)*, pages 425–438. Springer, 2016.

[11] Renáta Hodován and Ákos Kiss. Modernizing Hierarchical Delta Debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, pages 31–37. ACM, 2016.

[12] Renáta Hodován and Ákos Kiss. Practical Improvements to the Minimizing Delta Debugging Algorithm. In *Proceedings of the 11th International Joint*

Conference on Software Technologies, ICSOFT 2016, pages 241–248. SciTePress, 2016.

[13] Renáta Hodován and Ákos Kiss. Fuzzinator: An Open-Source Modular Random Testing Framework. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, ICST 2018, pages 416–421, April 2018.

[14] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Coarse Hierarchical Delta Debugging. In *In Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution*, ICSME 2017, pages 194–203. IEEE Computer Society, 2017.

[15] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging. In *Proceedings of the 12th IEEE/ACM International Workshop on Automation of Software Testing*, AST 2017, pages 23–29. IEEE Computer Society, 2017.

[16] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: A grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, pages 45–48. ACM, 2018.

[17] Barton Miller. Foreword for Fuzz Testing Book. `http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html`, 2008. [Accessed: 2019-03-01].

[18] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 142–151. ACM, 2006.

[19] Ghassan Shakib Misherghi. Hierarchical delta debugging. Master's thesis, University of California, Davis, 2007.

[20] Mozilla Foundation. The new Firefox. Fast for good. `https://www.mozilla.org/en-US/firefox/new/`. [Accessed: 2019-03-01].

[21] Mozilla Security. Javascript engine fuzzers. `https://github.com/MozillaSecurity/funfuzz/`. [Accessed: 2019-03-01].

[22] Stack Exchange Inc. Most popular technologies, programming, scripting, and markup languages. `https://insights.stackoverflow.com/survey/2018#technology`. [Accessed: 2019-03-01].

[23] Michał Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. [Accessed: 2019-03-01].

[24] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '99)*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer-Verlag, 1999.

[25] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.