Тестирование программного обеспечения

МКН СПбГУ, осень 2024

Федор Мамаев, 22Б09

Отчёт о выполнении Д/З

Что требовалось: написать фаззер для тестирования метода `org.jsoup.Jsoup.parse(java.lang.String)` из библиотеки Jsoup , который преобразует строку в HTML-документ, и найти в нём ошибки (выбросы исключений).

Ссылка на pull-request в репозитории с выполненным Д/З: https://github.com/FedMam/Joker2023-Verification-fall-2024-FedMam/pull/1

Как устроен код в репозитории:

Все мои наработки находятся в пакете `me.markoutte.joker.hw1`. Внутри пакета находятся 5 пакетов, соответствующих 5 шагам: `step1`, `step2` и т. д., так же, как и в пакете `me.markoutte.joker.parse`, который уже был в репозитории (там 3 шага). Также внутри находится пакет `strategies`, в котором реализованы различные стратегии фаззинга и мутации. Ошибки в методе найти не удалось (возможно, их вообще нет), но попробовано много разных методов фаззинга. Краткое описание 5 шагов:

- **Шаг 1**: почти точная копия шага 1 из пакета `me.markoutte.joker.parse`, отличается тем, что может использовать различные стратегии фаззинга (про них подробнее см. ниже). Генерирует случайные сэмплы и подаёт их на вход методу, пока не закончится отведённое время.
- **Шаг 2**: копия шага 2 из пакета `parse`. Генерирует случайные сэмплы, а потом пытается их случайно менять (мутировать). Отличается тем, что может использовать разные стратегии фаззинга и мутации.
- **Шаг 3**: копия шага 3 из пакета `parse`. При помощи инструментации может составлять ID для сэмпла на основе того, какие строчки кода были посещены при запуске метода с этим сэмплом на входе. При помощи мутации пытается пройти по как можно большему числу различных путей в коде метода.
- **Шаг 4** (на самом деле был выполнен позже шага 5): При помощи инструментации может определить точный набор строчек кода, которые были посещены при запуске метода. Генерирует случайные сэмплы и считает максимальное покрытие (максимальное количество посещённых строчек кода), достигнутое каким-то сэмплом. Можно включить или отключить мутацию. Создан для тестирования разных стратегий фаззинга и мутации и определения, какая из них лучше.
- **Шаг 5**: При помощи инструментации определяет покрытие, так же, как и на предыдущем шаге. На этом шаге реализован генетический алгоритм, который генерирует случайные сэмплы, пытаясь достичь максимального покрытия. Алгоритм действительно обучается (то есть, достигнутое покрытие увеличивается с течением времени), но работает очень медленно, и для достижения значимых результатов он должен работать долгое время.

Стратегии фаззинга, которые были реализованы

Для каждой стратегии приведен средний по 5 запускам результат выполнения шага 4 без мутации (максимальное покрытие случайного сэмпла) со случайным сидом в течение 25 секунд и результат работы шага 5 в течение 10 минут с сидом `192837` (используются все стратегии мутации одновременно, т. е. на каждом шагу выбирается случайная).

Класс	Описание	Пример (фрагмент)	Шаг 4	Шаг 5
`GarbageStrategy`	Генерация случайной последовательности байт	`╢4SP└ч t≤p╕;HpT fюoK▒'d©©Ed÷╪^√╒≥`	1060.8	1320
`ValidCharsStrategy`	Генерация случайной последовательности допустимых в HTML символов	`lfAnVbAY5>DFGWfa-"z;0UE&p6_ZtJvZp-16Q8xEeC-E;mIjeN-WBe1ZPKQP-H=-Hqt1S;wU_EE-AGrhn0Pn17C!X19H`	1214.6	1445
`SpecialCharsStrategy`	Генерация случайной последовательности специальных символов HTML	`!>!-/>/<>=</-!</;=-</<>-;//>/>>/>-<-<<<><==/><>/><!-;>//>;&>;/>=;>/!<&</!>>/<<></;<!<`	1186.4	900
`HTMLTagsStrategy`	Генерация HTML-тегов с названиями из случайных байт	`</§˜ъњЯ></Yк></ь><иT§IдMљ></SЏxт^9></Цї›?></></™*т></^></ђjs>ц></ВЋŸ></iёЋ}ЎpB></><>`	1281.0	1367
`HTMLTagsValidStrategy`	Генерация HTML-тегов с допустимыми в HTML названиями в правильной последовательности (т. е. есть открывающие и закрывающие теги и они правильным образом включают в себя другие теги)	`<g><plhigitgb><acza></acza><tshaqt></tshaqt><latzebhar></latzebhar></plhigitgb><vhjucij></vhjucij></g>`	1166.8	1045
`HTMLTagsAttributesStrategy`	Генерация HTML-тегов в правильной последовательности с атрибутами	`<nsm xvgyowkbet="R‡œØ_Ù¥ÎN[¿+LVí-2Eb‹ âiƒIkFÈH½ÛÿFÊFéIt^A—¤ã¨ä¨h" wihiwwjv="¸¦¹ÑôZX)?ÂJ"Î"><wvlndmica="J4L<;G·Ð'Œ\)2¸Ô}ß±ÎW'æ¦°º}ÇnéQÛ\dÕÜ¸"></wvlnd><viguspdu></viguspdu></nsm><vhbhjltqfv czcmghf="K%¨Ý‹´ö{Ë‡PEéÁ" oreltlttc="Ã[¨,ÍGqO/X§¥" nzykglmqp="ä¾¯Ÿ|à™.ÕÛI¨M*ê¿S‘p@x>IRæKÞ†ð—ðþhNÖµEÉ+KÔê"></vhbhjltqfv>`	1505.6	1352
`CorruptedHTMLStrategy`	Алгоритм берёт немного изменённый исходный код страницы https://acm.math.spbu.ru/ и случайным образом изменяет в нём несколько символов	`<!DOCTYPE html>?<html> â<7ead> <meta charseò="utf-8"> <title>Prog¦amming Contestsìat Nsk State Un«versity</tite> » <link rel="stylesheet" hrefz"/main.test.css?1"> <script languagF="javascr8pt"> Ì Éfunction add(type, y, m, d, num) { T document.write('<td><A HREF=/gikbin/monitor.plÖ'Ó+ type + y + m + d + '.áat>' + nump+ '</A></td><td>20' + y +'.' + mò+ '.' x d + '</td>'); ° } </script> </head>`	1900.2	1821
`GrammarBasedFuzzing`	Grammar-based fuzzing с грамматикой, описанной ниже	`<aac js="XXYOh" l="guc" y=";b" s="">j?A<!--g;coJ--><caa krj="guw;Bi" o="B"><cca><up></cca><h><exww y="gGQKf"><s><y wrh="ReeNQVM"><!--daOdD-->&f;DIDaXRdr!&k;`	1461.4	1610

В Grammar-based Fuzzing использовалась следующая грамматика:

```
{буква} = a | b | c |...| z
{символ} = a | b |...| z | A | B |...| Z | 0 | 1 |...| 9
            | . | , | : | ; | - | ! | ? | ( | )
{имя} = {буква}|{буква}{имя}
{текст} = ε | {символ}{текст}
{спецсимвол} = &{имя};
{атрибут} = ␣{имя}="{текст}"
{атрибуты} = ε | {атрибут}{атрибуты}
{тег} = <{имя}{атрибуты}>
{закрытый тег} = <{имя}{атрибуты}/>
{комментарий} = <!--{текст}-->
{контент} = ε | {текст}{контент} | {спецсимвол}{контент} | {тег}{контент}
              | {закрытый тег}{контент} | {комментарий}{контент}
              | {пара тегов}{контент}
```

`{пара тегов}` = <s`{атрибуты}>{контент}</`s> для всех строк s из ограниченного набора строк (т. к. по сути, для бесконечного набора строк s такое не описывается грамматикой). Набор: $\{x_1x_2x_3$ для $x_i \in \{$`a,b,c`$\}\} \cup \{$`html,head,body,div,script`$\}$
Начальный нетерминальный символ: `{пара тегов}`

Из этой таблицы мы можем сделать следующие выводы. Во-первых, разные стратегии генерации входных данных дают разное покрытие тестируемого кода, что ожидаемо. Во-вторых, генетический алгоритм не всегда показывает лучший результат, чем просто случайная генерация. По-видимому, это связано либо с тем, что сложные стратегии занимают много времени для генерации данных и алгоритм не успевает за 10 минут сделать достаточно итераций, либо с тем, что такие стратегии дают ему меньше свободы.

Стратегии мутации, которые были реализованы

В таблице приведён результат выполнения шага 5 со стратегией фаззинга `GarbageStrategy` и каждой стратегией мутации в течение 10 минут с сидом `192846`. Указан результат лучшего сэмпла в конце обучения (при первой случайной генерации лучший сэмпл имеет результат 1208). Все стратегии описаны в классе `strategies.MutationStrategies`.

Название функции	Описание	Результат работы
`brush`	Генерирует случайные числа `from` и `until` и заполняет случайный отрезок байтового массива значениями от `from` до `until`	1330
`shotgun`	Генерирует числа `from` и `until` и меняет значения не более $\sqrt{n}$ случайных байт на значения от `from` до `until`, где n - длина байтового массива	1243
`spray`	То же самое, но может поменять до n байт	1347
`eraser`	Заполняет случайный отрезок длины не более $\sqrt{n}$ нулями	1061
`solidBrush`	Заполняет случайный отрезок длины до n одинаковыми значениями	1302
`iotaBrush`	Заполняет случайный отрезок длины до n арифметической прогрессией с разностью 1	1265
`incBrush`	Прибавляет 1 к байтам на случайном отрезке длины до n	721
`plusBrush`	Генерирует случайное значение и прибавляет его к байтам отрезка длины до n	1283
`random`	Каждый раз выбирается случайная стратегия из вышеперечисленных	1260