BIGFUZZ: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction

Qian Zhang[1], Jiyuan Wang[1], Muhammad Ali Gulzar[2],
Rohan Padhye[3], and Miryung Kim[1]

[1]University of California, Los Angeles
[2]Virginia Tech
[3]Carnegie Mellon University

TEAM MEMBERS

Qian Zhang

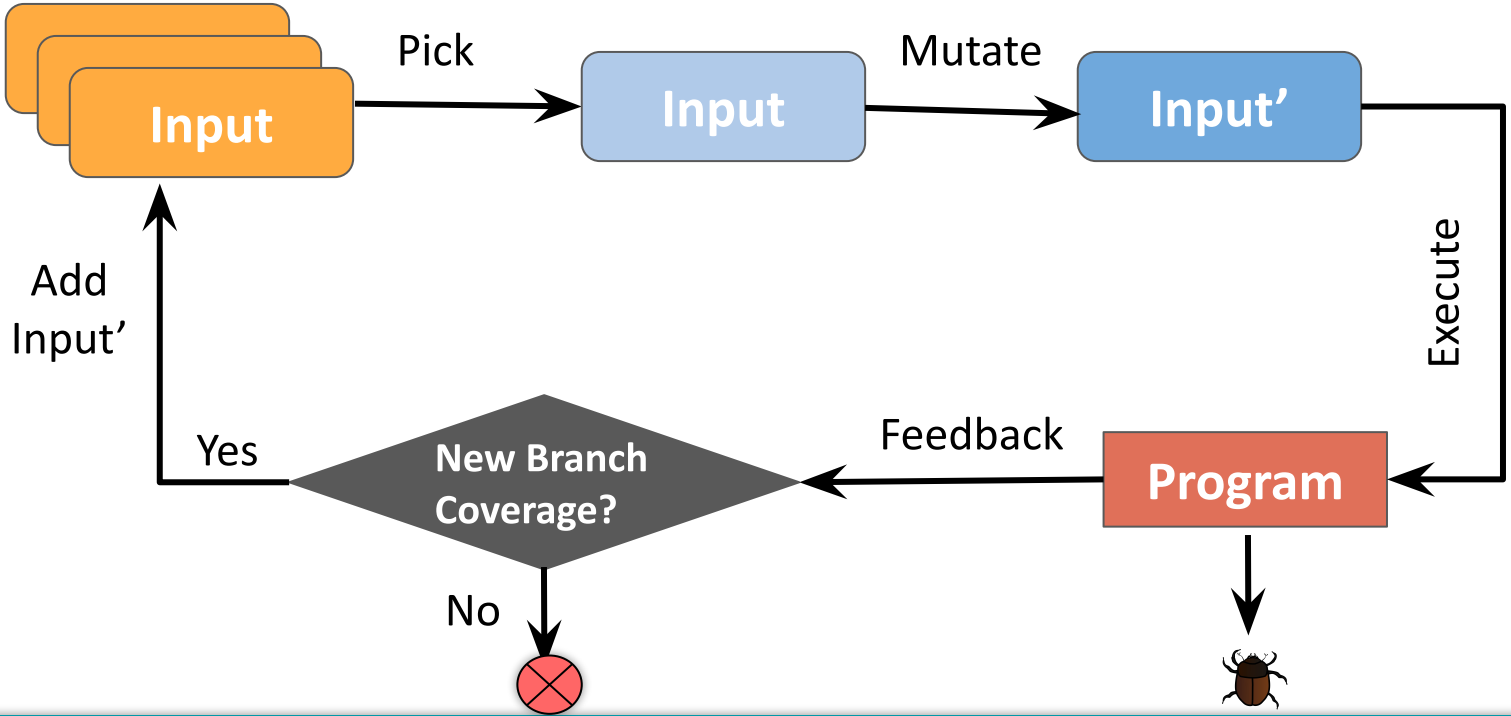
Jiyuan Wang

Muhammad Ali Gulzar

Rohan Padhye

Miryung Kim

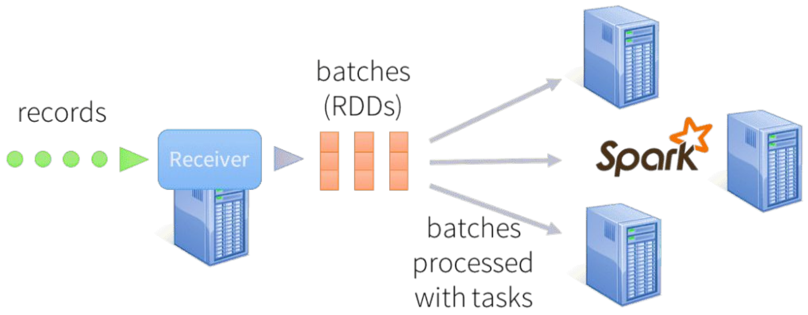
Fuzz testing is extremely *Popular* and *Effective*.



AFL[1,2] , a popular fuzzing tool that finds numerous errors

1.2020. American Fuzz Loop. http://lcamtuf.coredump.cx/afl
2.Patrice Godefroid, Michael Y. Levin, and David A Molnar. 2008. Automated White-box Fuzz Testing. In Network Distributed Security Symposium (NDSS). Internet Society. http://www.truststc.org/pubs/499.html

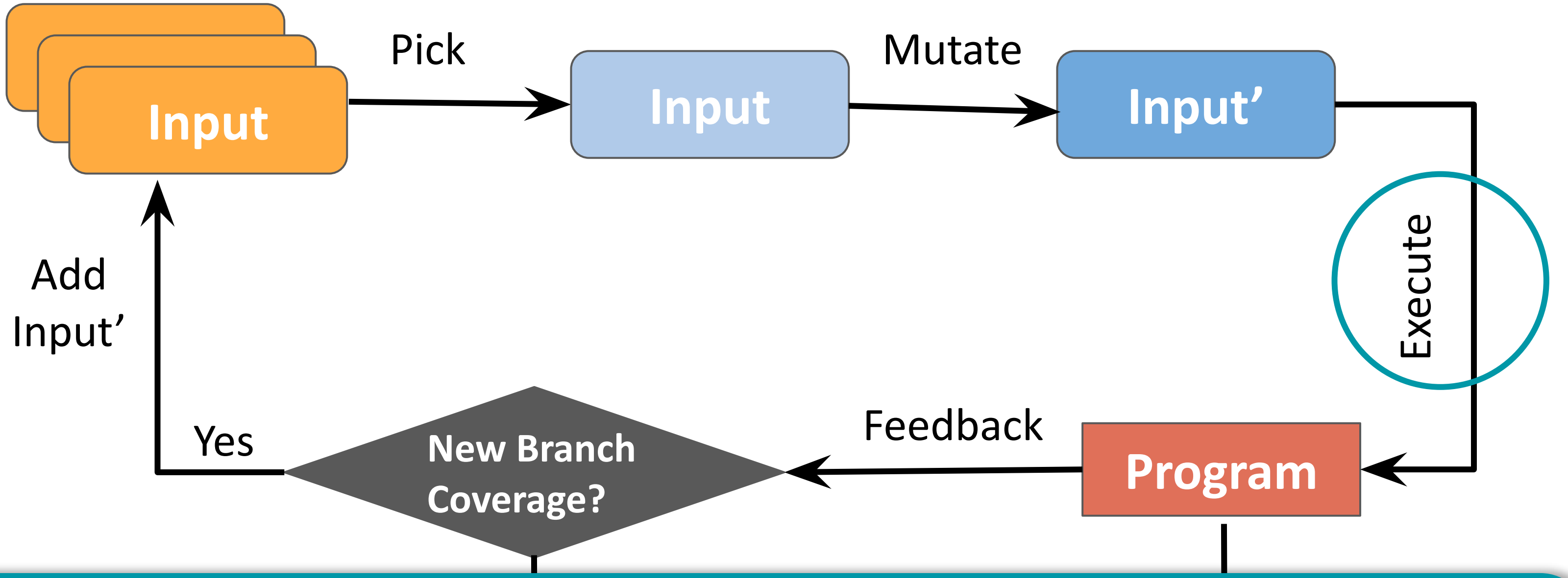
Big data analytics (BDA) is becoming important.



```
...
val locations = sc.textFile("zipcode.csv")
  .map { s => val cols= s.split(",")
         (cols(0), cols(1)) }
  .filter { s => s._2.equals("New York") }
...
```

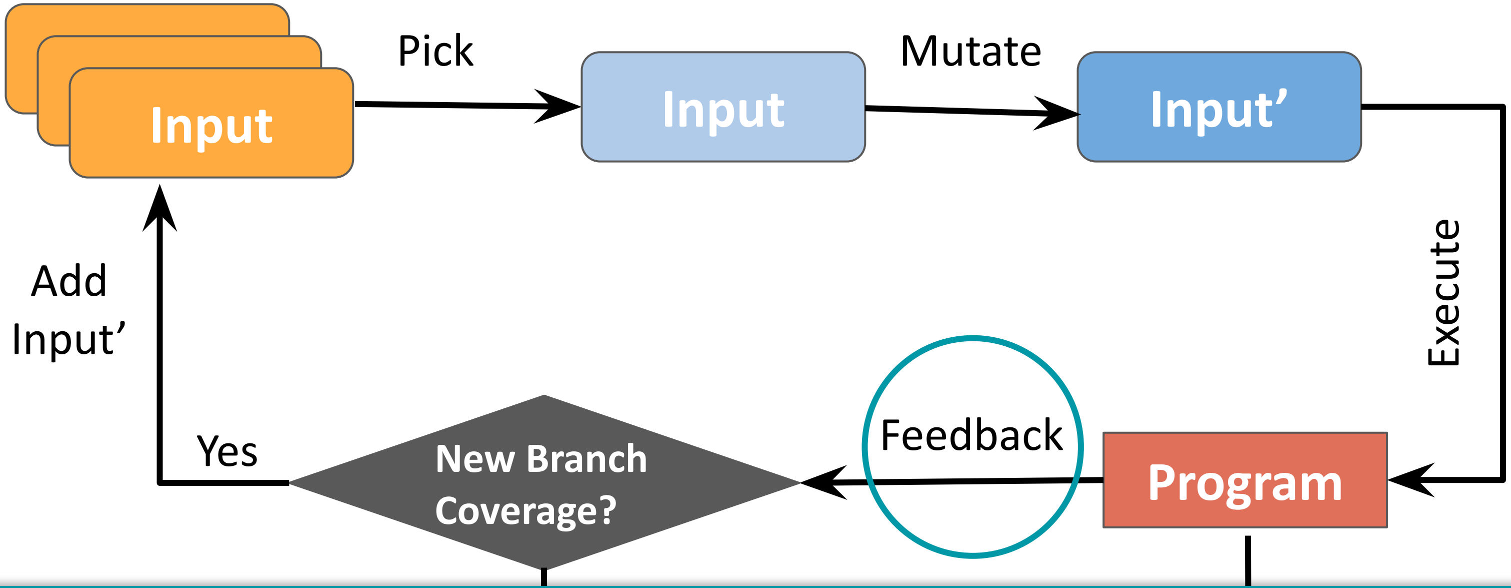
- Big data analytics programs compile to Java Bytecode
- But this includes the entire framework (700K LOC for Apache Spark)
- Dataflow implementation contributes most of the bytecode

Naïve Fuzzing is not easily applicable to BDA.



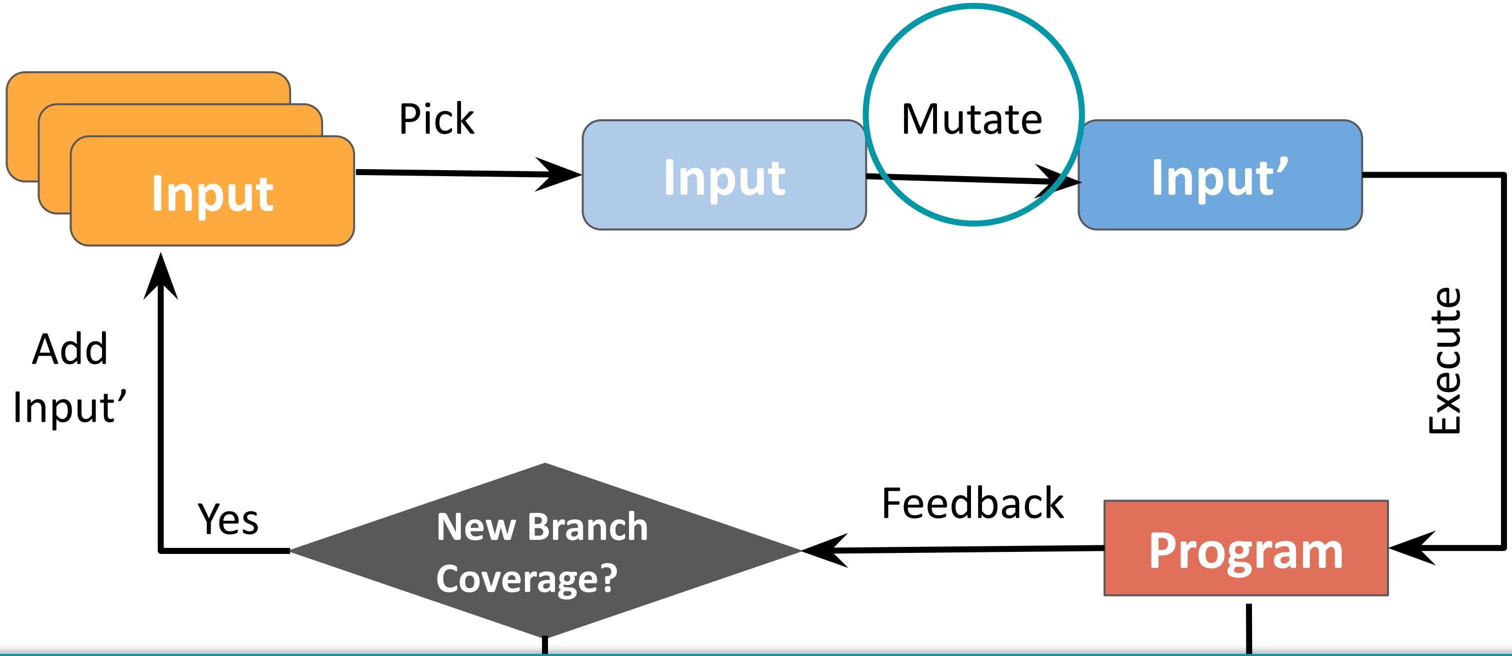
Challenge 1: Long latency of DISC systems prohibits the applicability of fuzzing.

Naïve Fuzzing is not easily applicable to BDA.



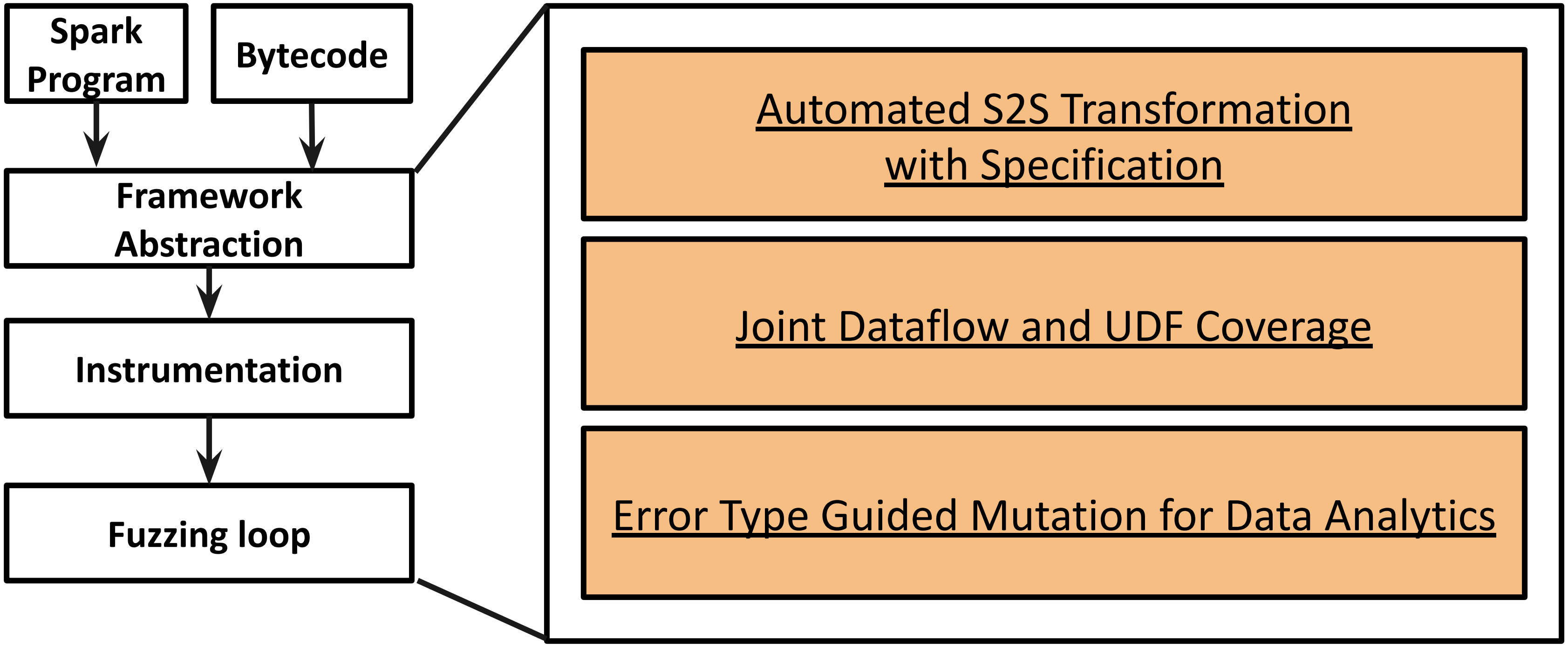
Challenge 2: Conventional branch coverage cannot represent equivalence classes of dataflow operators and is unlikely to scale to DISC applications.

Naïve Fuzzing is not easily applicable to BDA.



Challenge 3: Random binary mutations can hardly generate meaningful data.

BIGFUZZ Approach Overview



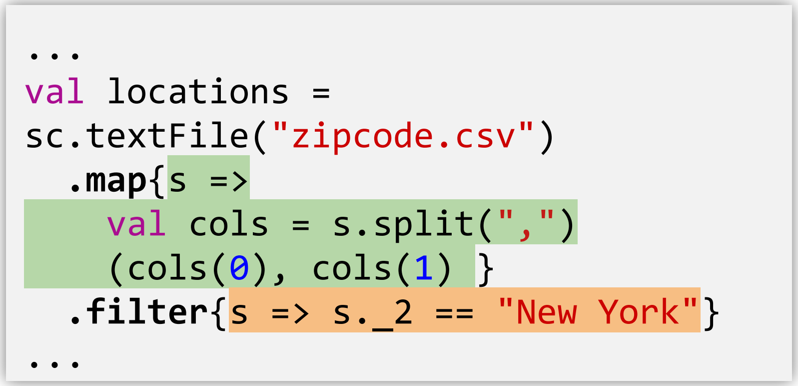
Automated S2S Transformation with Specification

Joint Dataflow and UDF Coverage

Error Type Guided Mutation for Data Analytics

Key insights: (1) abstracting framework code and (2) analyzing application code coverage as opposed to framework coverage.

Novelty 1: Framework Abstraction



```
...
val locations =
sc.textFile("zipcode.csv")
  .map{s =>
    val cols = s.split(",")
    (cols(0), cols(1) }
  .filter{s => s._2 == "New York"}
...
```

(a) **Original** Spark Code

Novelty 1: Framework Abstraction

```
...
val locations =
sc.textFile("zipcode.csv")
  .map{s =>
    val cols = s.split(",")
    (cols(0), cols(1) }
  .filter{s => s._2 == "New York"}
...
```

(a) **Original** Spark Code

Step 1: UDF Extraction

Step 2: S2S Transformation

Novelty 1: Framework Abstraction

```
...
val locations =
sc.textFile("zipcode.csv")
  .map{s =>
    val cols = s.split(",")
    (cols(0), cols(1) }
  .filter{s => s._2 == "New York"}
...
```

(a) **Original** Spark Code

```
public class Map1 {
static final Map1 apply(String line2)
{
String cols[]=line2.split(",");
return new Map1(cols[0],cols[1]);
}
```

(b) **Extracted UDF** from **.map**{...}
is represented as Map1.java

Step 1: UDF Extraction

Step 2: S2S Transformation

Novelty 1: Framework Abstraction

```
...
val locations =
sc.textFile("zipcode.csv")
  .map{s =>
    val cols = s.split(",")
    (cols(0), cols(1) }
  .filter{s => s._2 == "New York"}
...
```

(a) Original Spark Code

```
...
ArrayList<Map1> results1 =LoanSpec.map1
(inputs);
ArrayList<Map1> results2 =LoanSpec.filter2
(results1)
...
```

(c) Transformed program with executable specifications

```
public class Map1 {
static final Map1 apply(String line2)
{
String cols[]=line2.split(",");
return new Map1(cols[0],cols[1]);
}
```

(b) Extracted UDF from **.map**{…}
is represented as Map1.java

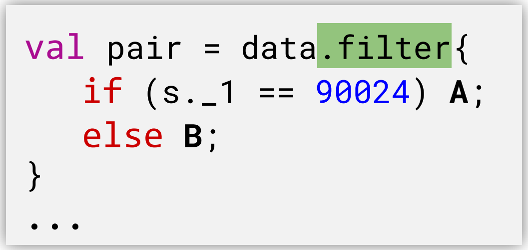
```
public ArrayList<Map1>
map1(ArrayList<String> input){
ArrayList<Map1> output = new ArrayList<>();
for (String item: input){
output.add(Map1.apply(item) );}
return output;}
```

(d) Specification implementation of **map** operator

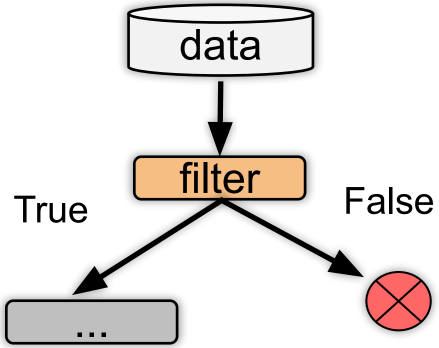
Step 1: UDF Extraction

Step 2: S2S Transformation

Novelty 2: Joint Dataflow & UDF Coverage



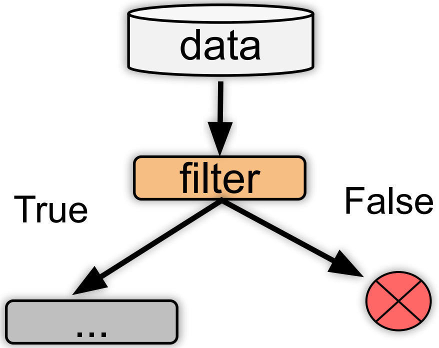
```
val pair = data.filter{
   if (s._1 == 90024) A;
   else B;
}
...
```



- **Filter** can introduce 2 equivalence class cases
- **Terminating**: *filter* predicate holds false thus individual data records stop at this *filter*;
- **Non-Terminating**: *filter* predicate holds true for at least one data record.

Novelty 2: Joint Dataflow & UDF Coverage

```
val pair = data.filter{
   if (s._1 == 90024) A;
   else B;
}
...
```



Input	Branch Coverage	JDU Coverage
[90024, 90095]	A, B -> save	A, B, filter.pass -> save
[90024]	A -> discard	A, filter.pass -> discard
[90000, 90095]	B -> discard	B, filter.fail -> save

- **Filter** can introduce 2 equivalence class cases
- **Terminating**: *filter* predicate holds false thus individual data records stop at this *filter*;
- **Non-Terminating**: *filter* predicate holds true for at least one data record.

Novelty 3: Error-Type Guided Mutation

- We design six mutation operations M1-M6 to reflect their association with each real world error type.

ID	Mutation	Example	Reflected Errors
M1	Data Distribution Mutation	an integer value 10 corresponding to `integer[0-30]` is mutated to 25 or -1	Incorrect code logic, incorrect API usage, join-related errors
M2	Data Type Mutation	20 corresponding to `integer[0-30]` is mutated to 20.0	Type mismatch
M3	Data Format Mutation	“,” to “~”	Split-related errors
M4	Data Column Mutation	insert ‘ ’	Split-related errors, illegal data for UDF
M5	Null Data Mutation	remove one or more columns	Incorrect column access
M6	Empty Data Mutation	mutate a random column to empty string	Incorrect offset access

Study of Common Error Types

- We study the characteristics of real-world data analytics errors posted on **StackOverflow** and **Github**.

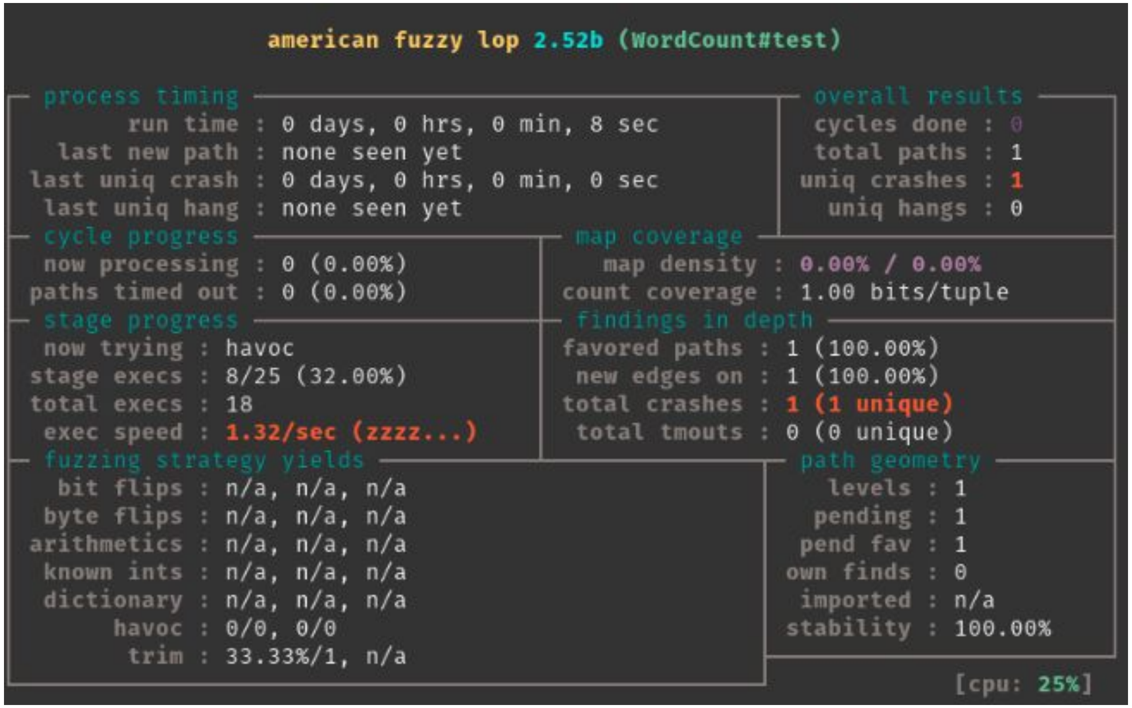
Survey Statistics	
Keywords Searched	Apache Spark exceptions, hadoop exceptions, task errors, failures, wrong outputs, SparkContext, etc.
Posts Studied in total	931 posts
Common Fault Types	10

Error Types	Example
Type mismatch	`.collect().foreach(println)`
Illegal data for UDF	`Division by zero`
Split-related errors	`str.split(“\t”)[1]`
Incorrect column access	`str.split(“,”)[1]`
Incorrect offset access	`str.substring(1,0)`
Incorrect code logic	`If(age>10 && age<9)`
Incorrect API usage	`LeftOuterJoin`
Join-related errors	`(Value, Key)`
Semantic errors	`Spark word2vec`
Framework errors	`one row join in spark`

Evaluation

- **RQ1**: Applicability
- **RQ2**: Speedup with framework abstraction
- **RQ3**: JDU coverage and error detection capability
- **RQ4**: Comparison with symbolic execution-based technique

RQ1: Applicability



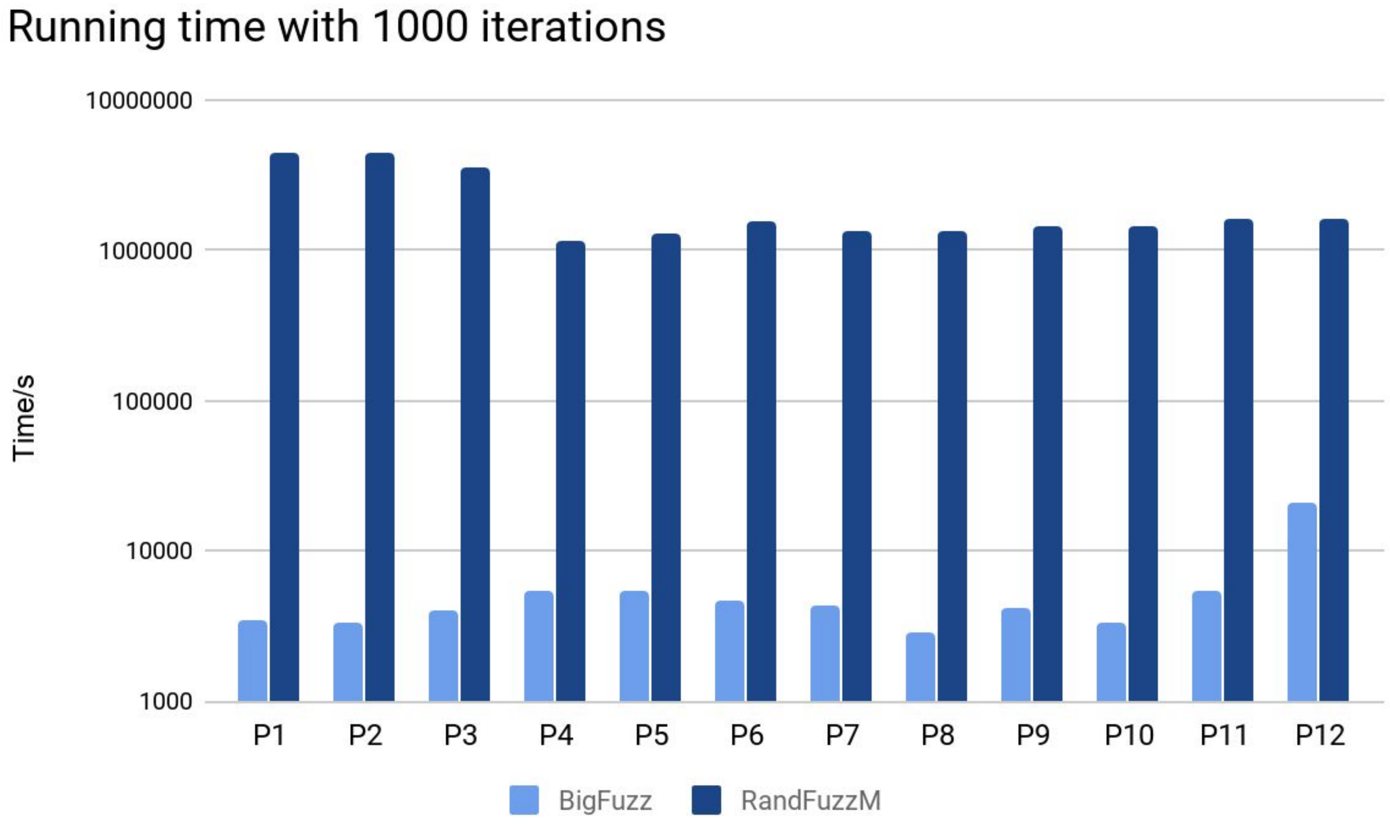
```
                 american fuzzy lop 2.52b (WordCount#test)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 8 sec         │  cycles done : 0      │
│   last new path : none seen yet                       │  total paths : 1      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 0 sec         │ uniq crashes : 1      │
│  last uniq hang : none seen yet                       │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)          │    map density : 0.00% / 0.00%        │
│ paths timed out : 0 (0.00%)          │ count coverage : 1.00 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                  │ favored paths : 1 (100.00%)           │
│ stage execs : 8/25 (32.00%)          │  new edges on : 1 (100.00%)           │
│ total execs : 18                     │ total crashes : 1 (1 unique)          │
│  exec speed : 1.32/sec (zzzz...)     │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                         │    levels : 1          │
│  byte flips : n/a, n/a, n/a                         │   pending : 1          │
│ arithmetics : n/a, n/a, n/a                         │  pend fav : 1          │
│  known ints : n/a, n/a, n/a                         │ own finds : 0          │
│  dictionary : n/a, n/a, n/a                         │  imported : n/a        │
│       havoc : 0/0, 0/0                              │ stability : 100.00%    │
│        trim : 33.33%/1, n/a                         ├────────────────────────┘
└─────────────────────────────────────────────────────┘             [cpu: 25%]
```

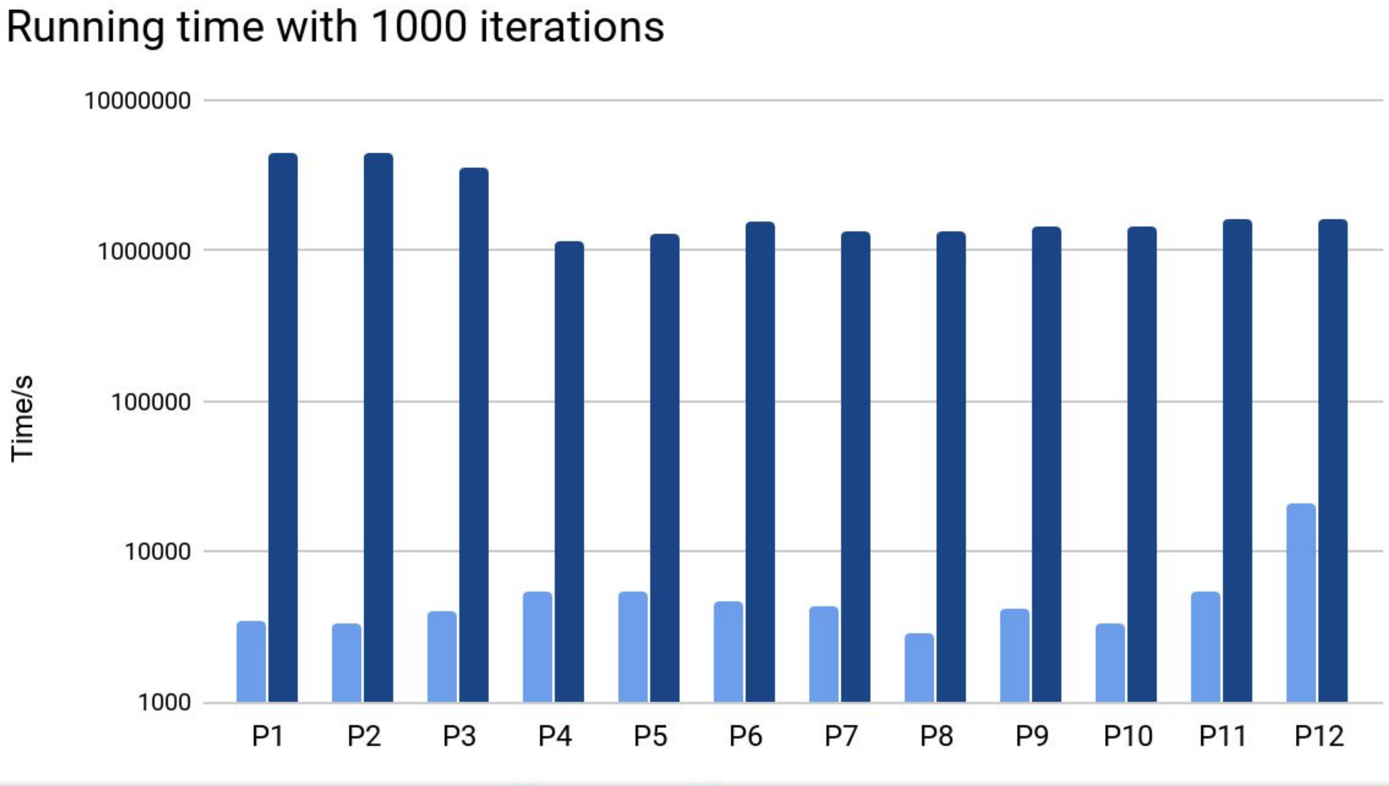
AFL (9216M memory and 100s timeout)

runs at an extremely low speed 9.68 execs_per_sec on average

RQ2: Speedup with Framework Abstraction



RQ2: Speedup with Framework Abstraction



Running time with 1000 iterations

BigFuzz speeds up to 1477x times with framework abstraction

RQ3: JDU Coverage and Error Detection Capability

Subject	Coverage %			Error Detection %		
	Random FuzzA	BigFuzz	Improvement	Random FuzzA	BigFuzz	Improvement
Word Count	50.00	100.0	2.00x	0.00	100.0	N/A
Commute Type	54.55	86.36	1.58x	62.50	87.50	1.40x
External Call	25.00	75.00	3.00x	0.00	100.0	N/A
Find Salary	42.48	75.00	1.77x	34.00	87.50	2.57x
Student Grade	23.21	86.10	3.71x	37.50	62.50	1.67x
Movie Rating	43.18	75.00	1.74x	35.71	64.30	1.80x
Inside Circle	78.57	96.43	1.20x	70.00	95.00	1.35x
Number Series	33.33	66.67	2.00x	50.00	81.25	1.63x
Age Analysis	41.67	94.44	2.27x	50.00	91.67	1.83x
IncomeAggregation	44.44	94.44	2.12x	50.00	91.67	1.83x
Loan Type	75.00	93.33	1.24x	67.50	90.00	1.33x

RQ3: JDU Coverage

Subject	Coverage %			Error Detection %		
	Random FuzzA	BigFuzz	Improvement	Random FuzzA	BigFuzz	Improvement
Word Count	50.00	100.0	2.00x	0.00	100.0	N/A
Commute Type	54.55	86.36	1.58x	62.50	87.50	1.40x
External Call	25.00	75.00	3.00x	0.00	100.0	N/A
Find Salary	42.48	75.00	1.77x	34.00	87.50	2.57x
Student Grade	23.21	86.10	3.71x	37.50	62.50	1.67x
Movie Rating	43.18	75.00	1.74x	35.71	64.30	1.80x
Inside Circle	78.57	96.43	1.20x	70.00	95.00	1.35x
Number Series	33.33	66.67	2.00x	50.00	81.25	1.63x
Age Analysis	41.67	94.44	2.27x	50.00	91.67	1.83x
Loan Type	75.00	93.33	1.24x	67.50	90.00	1.33x

BigFuzz provides up to a 3.71X improvement on code coverage

RQ3: Error Detection Capability

Subject	Coverage %			Error Detection %		
	Random FuzzA	BigFuzz	Improvement	Random FuzzA	BigFuzz	Improvement
Word Count	50.00	100.0	2.00x	0.00	100.0	N/A
Commute Type	54.55	86.36	1.58x	62.50	87.50	1.40x
External Call	25.00	75.00	3.00x	0.00	100.0	N/A
Find Salary	42.48	75.00	1.77x	34.00	87.50	2.57x
Student Grade	23.21	86.10	3.71x	37.50	62.50	1.67x
Movie Rating	43.18	75.00	1.74x	35.71	64.30	1.80x
Inside Circle	78.57	96.43	1.20x	70.00	95.00	1.35x
Number Series	33.33	66.67	2.00x	50.00	81.25	1.63x
Age Analysis	41.67	94.44	2.27x	50.00	91.67	1.83x
Loan Type	75.00	93.33	1.24x	67.50	90.00	1.33x

BigFuzz achieves up to a 2.57X improvement on error detection

RQ4: Compared with Symbolic Execution-based technique

	Subject Programs					
	P1	P2	P3	P4	P5	P6
Injected Errors	1	6	2	4	6	7
BigTest	0	5	1	2	4	3
BigFuzz	1	6	2	4	6	7

RQ4: Compared with Symbolic Execution-based technique

	Subject Programs					
	P1	**P2**	**P3**	**P4**	**P5**	**P6**
Injected Errors	1	6	2	4	6	7
BigTest	0	5	1	2	4	3
BigFuzz	1	6	2	4	6	7

In comparison to a symbolic execution based approach **BigTest**[1] , **BigFuzz** detects 80.6% more injected errors

1.Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. 2019. White-Box Testing of Big Data Analytics with Complex User-Defined Functions. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)

Acknowledgement

- **NSF grants:** CCF-1764077, CCF-1527923, CCF-1723773
- **ONR grant**: N00014-18-1-2037,
- **Intel CAPA grant**
- **Samsung grant**
- **Google PhD Fellowship**
- **Alexander von Humboldt Foundation**

BIGFUZZ: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction

Qian Zhang[1], Jiyuan Wang[1], Muhammad Ali Gulzar[2],
Rohan Padhye[3], and Miryung Kim[1]

[1]University of California, Los Angeles, [2]Virginia Tech, [3]Carnegie Mellon University

Tool link: https://github.com/qianzhanghk/BigFuzz

- We adapt fuzz testing to DISC applications with long latency.
- BIGFUZZ provides a novel solution that combines:
  - dataflow abstraction with specification;
  - tandem monitoring of dataflow coverage with UDF branch coverage;
  - application-specific mutations that reflect real world error types.