Co-dependence Aware Fuzzing for Dataflow-Based Big Data Analytics

Ahmad Humayun
ahmad35@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Miryung Kim
miryung@cs.ucla.edu
University of California, Los Angeles
Los Angeles, California, USA

Muhammad Ali Gulzar
gulzar@cs.vt.edu
Virginia Tech
Blacksburg, Virginia, USA

ABSTRACT

Data-intensive scalable computing has become popular due to the increasing demands of analyzing big data. For example, Apache Spark and Hadoop allow developers to write dataflow-based applications with user-defined functions to process data with custom logic. Testing such applications is difficult. (1) These applications often take multiple datasets as input. (2) Unlike in SQL, there is no explicit schema for these datasets and each unstructured (or semi-structured) dataset is segmented and parsed at runtime. (3) Dataflow operators (*e.g.,* `join`) create implicit co-dependence constraints between the fields of multiple datasets. An efficient and effective testing technique must analyze co-dependence among different regions of multiple datasets at the level of rows and columns and orchestrate input mutations jointly on co-dependent regions.

We propose DepFuzz to increase the effectiveness and efficiency of fuzz testing dataflow-based big data applications. The key insight behind DepFuzz is twofold. It keeps track of which code segments operate on which datasets, which rows, and which columns. By analyzing the use of dataflow operators (*e.g.,* `join` and `groupByKey`) in tandem with the semantics of UDFs, DepFuzz generates test data that subsequently reach hard-to-reach regions of the application code. In real-world big data applications, DepFuzz finds 3.4× more faults, achieving 29% more statement coverage in half the time as Jazzer's, a state-of-the-art commercial fuzzer for Java bytecode. It outperforms prior DISC testing by exposing deeper semantic faults beyond simpler input formatting errors, especially when multiple datasets have complex interactions through dataflow operators.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Testing, Fuzzing, Data analytics, Provenance, Taint analysis

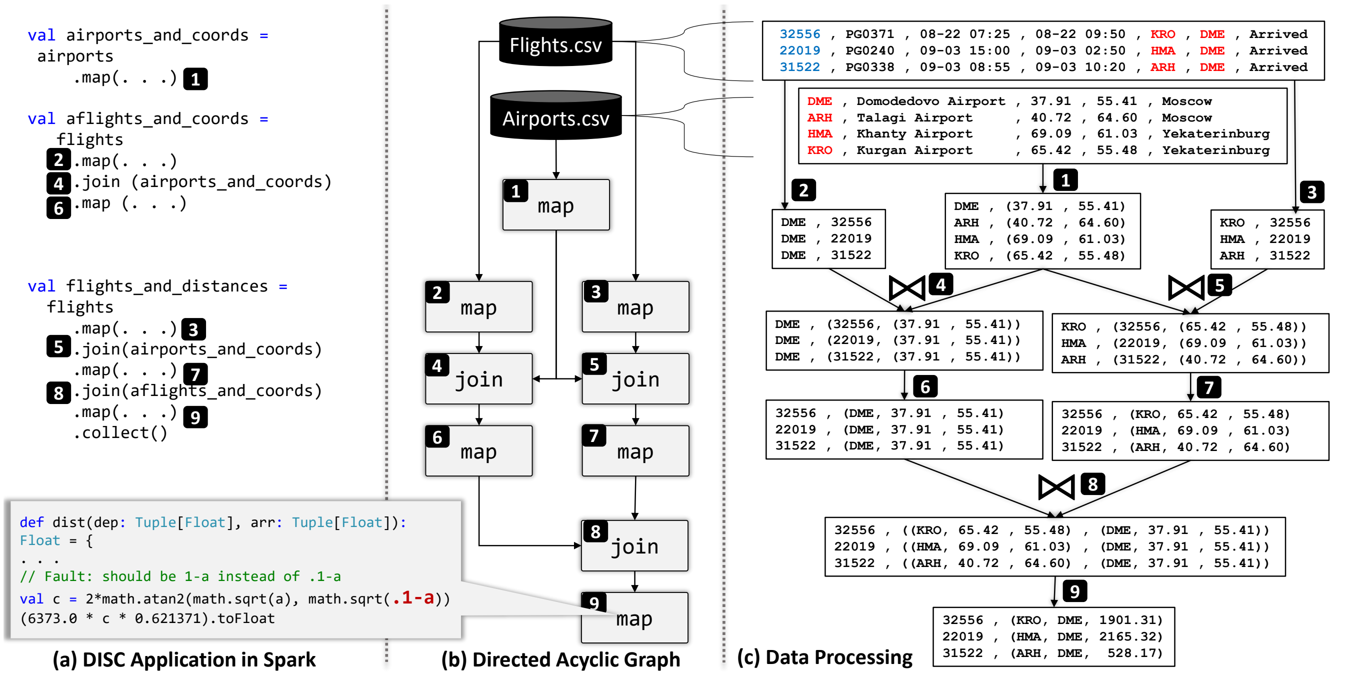
ACM Reference Format:
Ahmad Humayun, Miryung Kim, and Muhammad Ali Gulzar. 2023. Co-dependence Aware Fuzzing for Dataflow-Based Big Data Analytics. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616298

1 INTRODUCTION

Data-Intensive Scalable Computing (DISC) applications have become a prevalent way to process large-scale data. DISC frameworks like Hadoop MapReduce [2] and Apache Spark [3] offer APIs that contain dataflow operators such as `map`, `join`, and `groupByKey` for parallel data processing across thousands of machines. A typical DISC application builds on a series of dataflow operators in conjunction with user-defined functions (UDFs) that are passed as arguments to the dataflow operators. Despite the widespread usage of DISC applications, testing remains difficult due to their large input size and the applications' complex interactions with data.

Fuzzing is an effective software testing approach for many complex programs [1, 7, 9, 17, 27, 31, 40, 48, 51]. Fuzzers make small perturbations (mutations) to inputs to increase the likelihood of exercising uncovered application logic. Such traditional fuzzing may take a long time to generate meaningful inputs for DISC applications because a large input data has *too many locations* to mutate. Therefore, it is necessary to identify which rows and columns are worthwhile to mutate when a fuzzer attempts to reach a new code location. Naive mutations cannot satisfy complex input constraints from mixing dataflow operators and user-defined functions. For instance, `join` concatenates rows from two datasets that have matching values in designated *key* columns. This introduces an implicit equality constraint between the fields of multiple datasets. Consequently, to exercise code inside the UDF `func1` of `map` in the code snippet `dataset1.join(dataset2).map(func1)`, input mutations must simultaneously operate on both datasets `dataset1` and `dataset2` to observe the co-dependence constraint *i.e.,* there must exist a row in `dataset1` with the same key as the `dataset2`'s first column in order for `join` to produce any data on which `map` can apply `func1`. Mutations used in fuzz testing today fail to account for such *co-dependence* and thus may not exercise application logic beyond `join`. This problem is further exacerbated because, unlike SQL, there is no explicitly defined schema to identify columns, and the inputs for DISC applications are usually parsed on the fly.

We propose DepFuzz, a fuzzer that performs *co-dependence aware* row selection and column mutation while ensuring that constraints among *multiple datasets* are observed. DepFuzz combines row-level and column-level data tracking via taint analysis. In other words, it identifies which rows and which columns from which dataset are operated by individual lines of application code. This knowledge of row-level provenance helps reduce data size for subsequent fuzzing iterations, as DepFuzz retains only selected rows and mutates them,



```
val airports_and_coords =
 airports
    .map(. . .) 1

val aflights_and_coords =
   flights
  2.map(. . .)
  4.join (airports_and_coords)
  6.map (. . .)


val flights_and_distances =
  flights
    .map(. . .) 3
  5.join(airports_and_coords)
    .map(. . .) 7
  8.join(aflights_and_coords)
    .map(. . .) 9
    .collect()
```

```
def dist(dep: Tuple[Float], arr: Tuple[Float]):
Float = {
. . .
// Fault: should be 1-a instead of .1-a
val c = 2*math.atan2(math.sqrt(a), math.sqrt(.1-a))
(6373.0 * c * 0.621371).toFloat
```

```
32556 , PG0371 , 08-22 07:25 , 08-22 09:50 , KRO , DME , Arrived
22019 , PG0240 , 09-03 15:00 , 09-03 02:50 , HMA , DME , Arrived
31522 , PG0338 , 09-03 08:55 , 09-03 10:20 , ARH , DME , Arrived

DME , Domodedovo Airport , 37.91 , 55.41 , Moscow
ARH , Talagi Airport     , 40.72 , 64.60 , Moscow
HMA , Khanty Airport     , 69.09 , 61.03 , Yekaterinburg
KRO , Kurgan Airport     , 65.42 , 55.48 , Yekaterinburg

2
DME , 32556
DME , 22019
DME , 31522

1
DME , (37.91 , 55.41)
ARH , (40.72 , 64.60)
HMA , (69.09 , 61.03)
KRO , (65.42 , 55.48)

3
KRO , 32556
HMA , 22019
ARH , 31522

⋈ 4
DME , (32556, (37.91 , 55.41))
DME , (22019, (37.91 , 55.41))
DME , (31522, (37.91 , 55.41))

⋈ 5
KRO , (32556, (65.42 , 55.48))
HMA , (22019, (69.09 , 61.03))
ARH , (31522, (40.72 , 64.60))

6
32556 , (DME, 37.91 , 55.41)
22019 , (DME, 37.91 , 55.41)
31522 , (DME, 37.91 , 55.41)

7
32556 , (KRO, 65.42 , 55.48)
22019 , (HMA, 69.09 , 61.03)
31522 , (ARH, 40.72 , 64.60)

⋈ 8
32556 , ((KRO, 65.42 , 55.48) , (DME, 37.91 , 55.41))
22019 , ((HMA, 69.09 , 61.03) , (DME, 37.91 , 55.41))
31522 , ((ARH, 40.72 , 64.60) , (DME, 37.91 , 55.41))

9
32556 , (KRO, DME, 1901.31)
22019 , (HMA, DME, 2165.32)
31522 , (ARH, DME,  528.17)
```

Figure 1: A DISC application with a fault in the UDF of step 9, map: (a) code in Scala. (b) the corresponding dataflow graph, and (c) an illustration of data manipulation in 9 steps. Blue and red colored texts are co-dependent regions identified by DepFuzz.

as opposed to the entire dataset during mutational fuzzing without sacrificing code coverage. By inferring *co-dependence* relations among different columns from multiple datasets, it increases the chance of generating meaningful unstructured inputs that can reach the later stages of the application after operations such as `join` and `co-group` are used.

DepFuzz instruments the program under test by overriding dataflow operators and UDF components to capture row-level, column-level, and dataset-level provenance. This is done by implementing dynamic taint tracking for UDFs and dataflow operators. By leveraging co-dependence aware row selection and column mutations, it generates inputs that can reach deeper regions (*i.e.,* UDFs in the later stages of dataflow operators).

To evaluate DepFuzz, we use 17 DISC applications and measure (1) statement coverage, (2) fault detection capability, and (3) fuzzing speed-up. To assess fault detection capability, we inject faults at different depths in terms of the program's joint dataflow and control flow graph. We evaluate DepFuzz against two baseline techniques: Jazzer [23], a coverage-guided greybox fuzzer for Java bytecode based on LibFuzzer [44]; and BigFuzz [51], a greybox fuzzer for DISC applications. Comparison against Jazzer and BigFuzz serves to assess the overall benefit in terms of fault detection and speed-up, when orchestrating input mutations across multiple datasets by identifying co-dependence constraints. DepFuzz achieves 87% statement coverage, which is 29% and 13% more than Jazzer and BigFuzz. It also obtains coverage 2.1× and 1.3× faster than Jazzer and BigFuzz, respectively. Since faults appearing in earlier stages tend to be easier to find (*e.g.,* due to ill-formatted inputs) than those faults appearing in later processing stages, we evenly distribute injected faults in all dataflow operators for fairness. The average depth of a fault found by DepFuzz is 3.7 operators deep compared to 2.8 and 2.6 by Jazzer and BigFuzz, respectively. Our contributions are as follows:

- We present a new fuzz testing approach that leverages rich provenance information to increase mutational fuzzing's effectiveness and efficiency for DISC applications. This is the first test generation approach that extracts co-dependence constraints at the level of rows, columns, and datasets *fully automatically* without requiring an explicit schema from a user.
- Our evaluation includes an extensive comparison against two baselines on 17 different benchmark programs for 24 hours each. The results show DepFuzz reaches previously uncovered code faster, finds faults faster, and reaches deeper code locations of later stages than existing fuzzers.
- DepFuzz is built on extended dynamic taint tracking and analysis of dataflow operators. It has comprehensive support for Apache Spark-based DISC applications written in Scala, and its key idea generalizes to other dataflow-based big data applications such as Google's MapReduce or Apache Hadoop. DepFuzz is publicly available at https://github.com/SEED-VT/DepFuzz

2 MOTIVATING EXAMPLE

This section motivates DepFuzz with a concrete example. Suppose a data analyst computes the distance traveled by airplanes for each flight in 2017 from two input datasets: `flights` contains millions

Flights Dataset:
```
Int,String,String,String,String,String,String
20199,PG0320,08-09 06:55,08-09 09:15,MRV,PEE
```
Airports Dataset:
```
String,String,String,Float,Float,String
TOF,Bogashevo Airport,Tomsk,85.21,56.38,Krasnoyarsk
```
Duration:
```
24h
```

Figure 2: A sample configuration file required by BigFuzz. The file contains schemas and seed inputs for the `flights` and `airports` dataset, and a user-specified time cut-off for the fuzzing campaign.

of flights flown worldwide in 2017 and `airports` contains the geographic location of airports. The top two boxes in Figure 1 (c) show sample rows from each dataset. The `flights` dataset has a flight ID, the departure and landing times, the departure and arrival airport codes, and the flight status, all separated by commas. The `airport` dataset maps airport codes to their airport name, longitude and latitude coordinates, and the corresponding city. Figure 1 (a) shows a DISC application written in Spark. It consists of dataflow operators, such as `map` and `join`, where some dataflow operators, such as `map`, take a user-defined function (UDF) as an argument. For example, the $\boxed{9}$ `map` takes a UDF that computes distance using the Haversine formula, shown in the expanded text box.

In $\boxed{1}$, the analyst extracts the airport code and longitude and latitude values from `airport`. From `flights`, she selects the departure and arrival airport codes and the flight ID, as the first column and the second column in $\boxed{2}$ and $\boxed{3}$, respectively. She uses `join` in $\boxed{4}$ and $\boxed{5}$ to join the arrival and departure airports with their longitude and latitude coordinates. $\boxed{8}$ joins the two data streams using a flight ID. $\boxed{9}$ applies the `dist` function on the pairs of latitude-longitude tuples to compute the Haversine distance.

While writing the Haversine formula, she mistakenly writes `sqrt(.1-a)` instead of `sqrt(1-a)` (text box for $\boxed{9}$ in Figure 1 (a)). This error is hard-to-spot and subtle and causes `NaN` exceptions.

Limitations of Existing Fuzzers. To reveal such errors, suppose that she runs a commercially used, coverage-guided greybox fuzzer, Jazzer [23]. After a 24-hour fuzzing campaign, even with coverage guidance, Jazzer cannot produce an input to reach code beyond custom parsing logic at $\boxed{2}$, where it persistently triggers the same `ArrayOutOfBoundsException`. Jazzer achieves a maximum statement coverage of 27%. Due to a lack of schema and a lack of awareness of co-dependent regions, it continues to generate random strings for the two datasets that cannot pass beyond the parsing stage (*i.e.,* `map` at $\boxed{2}$).

Similarly, BigFuzz [51] requires an input schema (as shown in Figure 2) to apply *schema-aware* mutations such as changing the numerical value, changing integers to float, adding/removing columns, or changing the delimiter. These mutations help BigFuzz avoid some trivial parsing errors. Although BigFuzz achieves 98% statement coverage in 24 hours, it is still unable to trigger the fault in $\boxed{9}$, because to pass beyond `join` at $\boxed{8}$, the three columns (column 0 of `airports` and columns 4 and 5 of `flights`) must have the same value to satisfy *co-dependence* constraints to exercise the UDF of `map` at $\boxed{9}$. Since Jazzer and BigFuzz mutate all columns independently of each other, this three-way constraint is highly unlikely to be satisfied by their mutations.

Flights dataset:
```
-17252,P*34,4GJn50:0k0G,Zu:CSO.9-,D)N,D)N,A]i%e(
-17252,PG04,09-o0'gc:k7,zq-j01:55",D)N,D)N,]ZTed
```
Airports dataset:
```
D)N,SYhkutkl:irp7rS,Gap/ns4,1.64E9,1.30E9,JUSg3sk+
```

Figure 3: A test case generated by DepFuzz that causes a `NaN` exception in the program in Figure 1

Benefits of DepFuzz. Suppose that the data analyst uses DepFuzz to generate new test inputs. She does not need to provide an explicit schema and simply provides the current dataset to DepFuzz. At the end of 24 hours, DepFuzz generates new inputs as shown in Figure 3, leading to a `NaN` error, reaching the faulty line inside the corresponding UDF of `map` at $\boxed{9}$.

DepFuzz detects the two sets of co-dependent regions (highlighted in blue and red in Figure 3) and mutates them such that they can still satisfy the implicit constraints imposed by the three `join` operators ($\boxed{4}$, $\boxed{5}$, and $\boxed{8}$). The rows in blue (*i.e.,* -17252) must be equal since $\boxed{8}$ performs a self-join. The red cells (*i.e.,* D)N) are co-dependent by equality due to join $\boxed{4}$ and $\boxed{5}$. Close inspection of the application execution on this input shows that variable `c` is faulty at $\boxed{9}$ in Figure 1 (b). The developer spots this error on the second last line and replaces `sqrt(.1-a)` to `sqrt(1-a)` in accordance with the Haversine formula.

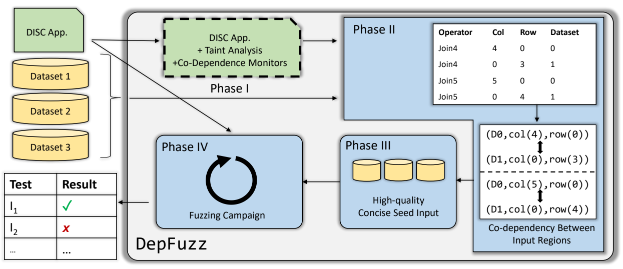
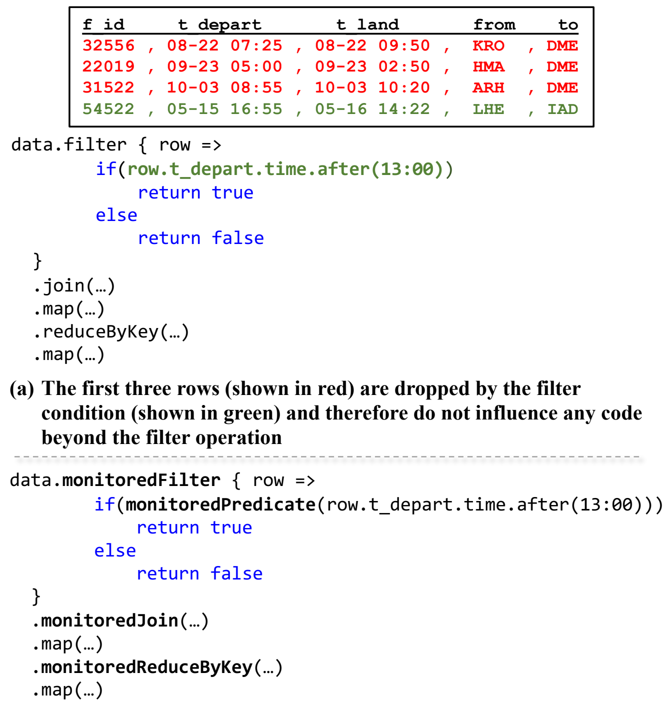


Figure 4: Workflow of DepFuzz.

3 APPROACH

The key contribution of DepFuzz is to detect co-dependent regions across multiple datasets and orchestrate input mutations on the co-dependent regions accordingly. In this section, we formally define co-dependence and provide details of how DepFuzz detects them. DepFuzz consists of four phases as shown in Figure 4. Phase I automatically instruments a given DISC application to enable fine-grained taint analysis at the level of rows, columns, and dataset IDs. This allows it to track data provenance through dataflow operators and UDFs to capture co-dependence relationships. Phase II executes this instrumented program on the entire dataset to capture co-dependence constraints among multiple input datasets. Phase III leverages this provenance tracking capability to select a precise subset of rows from each dataset to use as seeds for subsequent fuzzing iterations. Phase IV then initiates a fuzzing campaign with the selected rows from Phase III and applies co-dependence aware mutations to expose deeper faults. After reaching a user-specified time limit, DepFuzz outputs a set of test inputs.

Formalizing Input Co-dependence. Co-dependence is a dependency created between multiple input regions by an operation (*e.g.,* a dataflow operator or a binary operation that affects control flow in UDFs) that operates on such input regions. An input region is



f id	t depart	t land	from	to
32556	08-22 07:25	08-22 09:50	KRO	DME
22019	09-23 05:00	09-23 02:50	HMA	DME
31522	10-03 08:55	10-03 10:20	ARH	DME
54522	05-15 16:55	05-16 14:22	LHE	IAD

```
data.filter { row =>
        if(row.t_depart.time.after(13:00))
            return true
        else
            return false
  }
  .join(…)
  .map(…)
  .reduceByKey(…)
  .map(…)
```

(a) The first three rows (shown in red) are dropped by the filter condition (shown in green) and therefore do not influence any code beyond the filter operation

```
data.monitoredFilter { row =>
        if(monitoredPredicate(row.t_depart.time.after(13:00)))
            return true
        else
            return false
  }
  .monitoredJoin(…)
  .map(…)
  .monitoredReduceByKey(…)
  .map(…)
```

(b) Co-dependence monitors are attached to each branch in the JDU Graph.

Figure 5: The red color rows do not participate in the `join` operation since the `filter` operation removed them earlier.

a contiguous sequence of bytes in an input dataset. We formalize co-dependence as follows. Given a DISC application, we define its dataflow graph (DFG) with two types of vertices: *operators* and *datanodes*, similar to the traditional DFG representation [26]. In the case of DISC applications, *Operators* are functions in a program that operate on data (*e.g.,* `join()` dataflow operator or == in UDFs). A comprehensive list of trackable operators is shown in Table 1. *Datanodes* represent data that propagate from one operator to another (*i.e.,* input and output of an operator). Thus, we define a DISC application's DFG, G, as

$$G = \langle O \cup N, E \rangle$$

where $O = \{o_1, o_2, ..., o_n\}$ is a set of operators, $N = \{n_1, n_2, ..., n_m\}$ is a set of datanodes. $E \subseteq (O \times N) \cup (N \times O)$ is a set of directed edges connecting operators with datanodes. An atomic unit of this DFG has three nodes and two edges *i.e.,* an operator with an incoming edge from an input datanode and an outgoing edge to an output datanode. Furthermore, a datanode n holds data in the form of a *byte sequence* $b_1b_2...b_k$. Let $D(n)$ be the set of all possible subsequences of the byte sequence in a datanode n, *i.e.,* $\{b_1, b_2, ..., b_1b_2, ..., b_1...b_k\}$. Input datasets of DISC applications are defined as S, a set of initial datanodes which are external inputs to the DFG. We combine regions in input datasets in S', a union of $D(n)$ across all input datanodes.

$$S' = \bigcup_{\forall n \in S} D(n)$$

```
case class TaintedString(value: String, t: Taints){
 // A Tainted String class
 def concat(other: String): TaintedString =
  return new TaintedString(value.concat(other), t)

 def concat(other: TaintedString): TaintedString =
  return new TaintedString(
                value.concat(x.value),
                union(t,x.t)
            )
... // more overloaded operators
```

Figure 6: Taint analysis enabled String type in Scala

Finally, we characterize co-dependency among input regions as a set of tuples, C

$$C = \{(o, R) \mid R \subseteq S', \forall o \in O\}$$

The first element, o, is an operator in the dataflow graph; and the second element, R, is a subset of the regions in the input datasets that are co-dependent due to operator o. Let $I(o)$ be the incoming data to an operator o. Since co-dependence can only occur between regions of the original datasets, we must extract R from $I(o)$, which can be any arbitrary byte sequence in the incoming datanode of operator o. To extract such information, we define *monitors* that are concretely explained in Section 3.1.

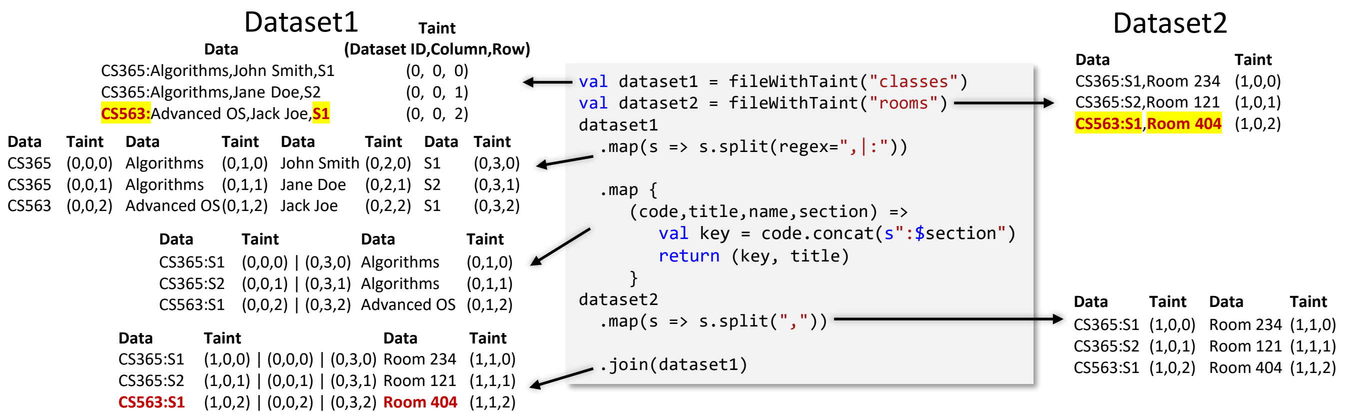
$$M_o : I(o) \longrightarrow \mathcal{P}(S')$$

where $\mathcal{P}(S')$ is the powerset of S'. A monitor, M_o, is an operator-specific function that takes $I(o)$ as input and outputs a set of byte sequences (*i.e.,* input regions) from the original input datasets considered co-dependent *w.r.t.* given operator. The precise logic behind this mapping depends on the semantics of the operators, which we capture using dynamic tainting in DepFuzz. Take for example a == operation in `a.substr(0,5) == b.substr(0,5)`. The monitor, $M_{==}$, should yield `{a.substr(0,5), b.substr(0,5)}`, resulting in a co-dependence tuple `(==, {a.substr(0,5), b.substr(0,5)})`. Table 1 lists concrete examples of operators, their monitors M, and the respective mutation strategies.

3.1 Phase I: Enabling Fine-Grained Taint Analysis

Phase I instruments a given input program to enable taint analysis and to capture co-dependence information.

Enabling taint analysis via instrumentation. DepFuzz uses taint analysis to identify precise columns and rows in the input datasets contributing towards a specific intermediate output or a final output. DepFuzz first replaces primitive data types with equivalent *tainted* types. The tainted data type is a tuple of an original type and a list of offsets representing taints, `(Value[T], List[Offset])`. DepFuzz overrides all APIs of the original data type with taint-enabled equivalent versions to propagate their taints. For example, `concat` in `tstr1.concat(tstr2)` concatenates two tainted strings, `tstr1` and `tstr2`, and attaches a new taint with the union on the two corresponding taints. Figure 6 shows the implementation of `concat` in `TaintedString`. DepFuzz provides an instrumented version of data loading APIs that read the input datasets in tainted types instead of primitive types. Similarly, we instrument APIs for `Int`, `Float`, `Double` and `Boolean`. Apache Spark's `textFile` API



Data	Taint (Dataset ID,Column,Row)
CS365:Algorithms,John Smith,S1	(0, 0, 0)
CS365:Algorithms,Jane Doe,S2	(0, 0, 1)
CS563:Advanced OS,Jack Joe,S1	(0, 0, 2)

```
val dataset1 = fileWithTaint("classes")
val dataset2 = fileWithTaint("rooms")
dataset1
  .map(s => s.split(regex=",|:"))

  .map {
     (code,title,name,section) =>
        val key = code.concat(s":$section")
        return (key, title)
  }
dataset2
  .map(s => s.split(","))

  .join(dataset1)
```

Data	Taint
CS365:S1,Room 234	(1,0,0)
CS365:S2,Room 121	(1,0,1)
CS563:S1,Room 404	(1,0,2)

Data	Taint	Data	Taint	Data	Taint	Data	Taint
CS365	(0,0,0)	Algorithms	(0,1,0)	John Smith	(0,2,0)	S1	(0,3,0)
CS365	(0,0,1)	Algorithms	(0,1,1)	Jane Doe	(0,2,1)	S2	(0,3,1)
CS563	(0,0,2)	Advanced OS	(0,1,2)	Jack Joe	(0,2,2)	S1	(0,3,2)

Data	Taint	Data	Taint
CS365:S1	(0,0,0) | (0,3,0)	Algorithms	(0,1,0)
CS365:S2	(0,0,1) | (0,3,1)	Algorithms	(0,1,1)
CS563:S1	(0,0,2) | (0,3,2)	Advanced OS	(0,1,2)

Data	Taint	Data	Taint
CS365:S1	(1,0,0)	Room 234	(1,1,0)
CS365:S2	(1,0,1)	Room 121	(1,1,1)
CS563:S1	(1,0,2)	Room 404	(1,1,2)

Data	Taint	Data	Taint
CS365:S1	(1,0,0) | (0,0,0) | (0,3,0)	Room 234	(1,1,0)
CS365:S2	(1,0,1) | (0,0,1) | (0,3,1)	Room 121	(1,1,1)
CS563:S1	(1,0,2) | (0,0,2) | (0,3,2)	Room 404	(1,1,2)

Figure 7: Taint propagation through a simple dataflow program. Yellow colored highlighted text is the provenance of red colored text at the bottom left table.

reads each row as a `String`, while the taint-analysis equivalent version reads each row as a `TaintedString` with a row offset and a dataset ID.

Taint propagation at the level of rows, columns, and datasets. Randomly mutating the entire row will likely mutate non-participating regions in the input. In Figure 1, the second, third, fourth, and seventh columns in the first dataset are never used by the application code. DepFuzz implements an extended taint analysis at the level of a dataset ID, a column offset, and a row offset *i.e.,*(`Value[T], List[(DatasetID, ColOffset, RowOffset)]`. For example, in Figure 7, `CS363:Advanced OS, Jack Joe, S1` has a taint `[0,0,2]` meaning the data is from the first dataset, the first column, and the third row. To reduce the storage overhead of attaching a tainted object, DepFuzz encodes the three offsets into a single 32-bit integer.

Co-dependence monitors. In order to associate taints at the level of branches and dataflow operators, DepFuzz injects co-dependence monitors at each dataflow operator and at each branch predicate within UDFs, as shown in Figure 5. For example, this process replaces a dataflow operator `join` with `monitoredJoin` and replaces `if(p)` with `if(monitoredPredicate(p))` within UDFs. This co-dependence monitor injection enables DepFuzz to identify which rows and columns from which datasets directly influence individual branching decisions. Branches in a DISC application include both an explicit control predicate from an `if` statement or a `for` loop in user-defined functions and implicit branches from dataflow operators (*e.g.,* `join` and `filter`).

3.2 Phase II: Fine-Grained Taint Tracking

DepFuzz runs the instrumented, taint-analysis enabled version from Phase I on the original datasets. Figure 7 shows how data is tracked through the execution of a taint-enabled program.

Co-Dependence detection. Dataflow operators and UDFs pose implicit and explicit co-dependence constraints. For instance, `join` enforces an implicit constraint that, for each output row, the keys of the two joining datasets must be equal. Similarly, `if(airporta == airportb)` imposes an explicit constraint that the `airporta` and `airportb` (derived from specific rows and columns of input datasets) are equal. Co-dependence also arises between the rows of the same dataset. For example, *aggregation* operators such as `reduceByKey` and `groupByKey` result in co-dependence where one or more rows must have the same key to have an output row with the same key. Our insight is that while random mutations are unlikely to satisfy co-dependence constraints by chance, coordinated mutations to specific row and column offsets that respect co-dependency constraints are likely to reach deeper code.

Exactly how taints are transformed into co-dependence constraints depends on the monitored dataflow operator type. For example, for `join`, the key columns of the two participating datasets must be the same (equality). For an `if` condition `if(column0 > column5)`, the co-dependence is a "greater than" relationship.

Once the instrumented application's execution on the original datasets completes, DepFuzz consolidates co-dependence information, documenting each monitor's relative position in terms of dataflow operator depth and the list of taints containing offsets at the level of rows, columns, and dataset IDs. For example, in Figure 1, `join` 4 has a depth of two and forms a co-dependence between column 5 of `flights` and column 0 of `airports` which act as the keys for the join. Note that DepFuzz can detect transitive co-dependence when there are overlapping constraints across multiple operators. For example, Figure 1 has a three-way co-dependence among three input regions since `airports` column 0 overlaps with `join` 5.

3.3 Phase III: Row Selection for Data Size Reduction

To speed up fuzzing, DepFuzz identifies a small subset of data rows that retain the same branch coverage as the original dataset. This reduces large-scale datasets to a set of seed inputs that are small enough for iterative fuzzing. Because the original input data may be very large with millions of rows, this step significantly reduce the scope of potential locations to mutate, increasing efficiency. For each branch, DepFuzz reduces the original input datasets to a subset of rows reaching that particular branch. It then consolidates the corresponding rows for all branches. Figure 5 (a) shows an example of how row selection creates a smaller, effective seed. A `filter` operator removes all flights departing before 13:00 on a given day. Therefore, the rows highlighted in red will not influence any code

Operator Class	Sample Operators	Example of Identified Constraint	Mutation strategy

Fusions	`data1.join(data2)`
	`data1.intersection(data2)`
	`data1.cogroup(data2)`
		$M_{join}(\{data1, data2\})$.
		Possible output of M_{join}:
		`{data1.row[1].col[3], data2.row[23].col[0]}`
		`{data1.row[31].col[3], data2.row[52].col[0]}`
		Co-dependence tuples:
		`(== , {data1.row[1].col[3], data2.row[23].col[0]})`
		`(== , {data1.row[31].col[3], data2.row[52].col[0]})`
			Any mutation applied to `data1.row[1].col[3]` must also be applied to `data2.row[23].col[0]`. If no rows with matching keys exists, select a row from `data1` and copy `data1.col[3]` to `data2.col[0]`.

Aggregations	`data.aggregateByKey(udf)`
	`data.reduceByKey(udf)`
	`data.groupByKey()`
	`data.countByKey()`
		$M_{reduceByKey}(\{data\})$.
		Possible output of $M_{reduceByKey}$:
		`{data.row[2].col[2],`
		`data.row[43].col[2],`
		`data.row[63].col[2]}`
		Co-dependence tuple:
		`(== , {data.row[2].col[2],`
		`data.row[43].col[2],`
		`data.row[63].col[2]})`
			Any mutation applied to `data[row=2,col=2]` must also be applied to `data[row=43,col=2]`. Duplicate rows and apply same mutation to key columns of duplicates.

Filters	`data.filter(col0 > col5)`	$M_{filter}(\{data\})$.
		Possible output of M_{filter}:
		`{data.row[23].col[0], data2.row[23].col[5]}`
		`{data.row[31].col[0], data2.row[31].col[5]}`
		Co-dependence tuples:
		`(> , {data1.row[23].col[0], data2.row[23].col[5]})`
		`(> , {data1.row[31].col[3], data2.row[31].col[5]})`
			Any mutation applied to `data.col[0]` and `data.col[5]` must ensure that there is a true and false row for the predicate.

UDF Operators	`if(a.contains(b))`
	`if(a != b)`
	`if(a > b)`
		$M_{contains}(\{a, b\})$.
		Possible output of $M_{contains}$:
		`{data.row[1].col[0], data2.row[1].col[2]}`
		Co-dependence tuples:
		(*contains* , `{data1.row[1].col[0], data2.row[1].col[2]}`)
			Any mutation applied to `data.col[0]` and `data.col[2]` must ensure that string a contains b for some mutations. It must also ensure it occasionally creates inputs that violate this.

Table 1: Summary of how each class of operators produces co-dependent regions in the input dataset. For simplicity, we use `row[1].col[3]` as a human-readable representation of input byte region $b_i, ...b_j$, where $0 < i < j < size(dataset)$

beyond the first `filter`. DepFuzz thus retains only the green row in the seed input for subsequent fuzzing iterations.

3.4 Phase IV: Co-Dependence Aware Mutation

Phase IV performs a grey-box fuzzing campaign by designing new mutations that target various co-dependence types. The output of DepFuzz is a list of errors and test inputs revealing those errors. Different from standard grey-box fuzzing, DepFuzz prioritizes *where* to apply input mutations based on fine-grained taint tracking at the level of rows, columns, and datasets. DepFuzz designs a novel input mutation strategy that maintains co-dependency. Based on the co-dependent constraints, we categorize dataflow operators into four classes: *Fusions*, *Aggregations*, *Filters*, and *UDF Operators*. Table 1 summarizes mutation strategies for each class of operator.

- For *fusion operators* like `join`, DepFuzz applies the same set of mutations on the key columns of the two joining datasets to ensure equality. In Figure 1 (c), when DepFuzz mutates `KRO` in row 0 of the `flights` dataset, it applies the same mutations to `KRO` in row 3 of `airports`, ensuring a non-empty output for `join`.
- For *aggregation operators* like `reduceByKey`, DepFuzz duplicates a row and applies the same set of mutations on the key column of those rows, ensuring at least 2 rows in each output group. Suppose if `reduceByKey` is applied on the fourth column of `flights` in Figure 1. DepFuzz duplicates a row >1 times and applies the same mutation on the key of the original and duplicated rows.
- For *filter operators* like `filter`, DepFuzz applies mutation on the columns used in the filtering predicate. In case of `filter(data.col[0]) > data.col[1]`, DepFuzz can create at least one row where this predicate can be true or at least one row where this predicate is false.
- For *UDF operators* like `map` and `flatMap` that take UDFs as arguments, DepFuzz handles control predicates in user-defined functions similar to `filter`. For example, in the case of `a.contains(b)`, DepFuzz identifies the provenance of the strings `a` and `b` as `data.col[0]` and `data.col[3]` respectively. DepFuzz then enforces the true path for this condition by embedding `b` in `a` during the mutation process.

4 EVALUATION RESULTS

We evaluate DepFuzz on four criteria: code coverage, fault detection, fault depth, and testing speed, transcribed into the research questions below.
RQ1: What is DepFuzz's test coverage compared to baseline fuzzers?
RQ2: How many errors can DepFuzz detect compared to baselines?
RQ3: Can DepFuzz detect errors located in *deeper* code regions?
RQ4: How much overhead does DepFuzz's instrumentation incur?
RQ5: Does DepFuzz achieve code coverage faster than baselines?

Benchmarks. Existing dataflow benchmarks like TPC-DS [5] or Big Data Benchmark [4] are purely performance benchmarks written in SQL and therefore do not contain UDFs and non-relational dataflow operators. In contrast, the subject programs introduced by prior work on fuzzing in DISC only operate on a single dataset, omitting an entire class of operators related to real-world multiple dataset analytics. Therefore, we evaluate DepFuzz on 17 unique big data applications accumulated from *nearly all* publicly available prior work on DISC testing [19, 51], DISC debugging [47],

ID	Program	Description	Datasets	# of Opt.	Max Depth	Total Rows	Operators Used
P1	Webpage Segmentation [10, 15]	Find overlapping UI components on a webpage	2	9	6	1M	map, groupByKey, join, filter
P2	Customer Rewards [8]	Find the top-3 customers w.r.t purchase history	2	9	8	2M	map, groupByKey, join, filter, sortBy
P3	Flight Distance [6]	Compute distance travelled by a given flight	2	9	5	500K	map, join
P4	Bus Delays [34]	Identify bus routes that are delayed frequently	2	9	8	2M	flatMap, join, reduceByKey, filter
P5	Commute Type [19]	Identify the transportation type used on a trip	2	4	4	1M	map, mapValues aggregateByKey
P6	WordCount [19]	Find the frequency of words	1	2	2	1M	map, flatMap, reduceByKey
P7	Delivery Faults [35]	Identify vendor sets leading to faulty deliveries	1	5	5	1M	map, groupByKey, filter
P8	ExternalCall [51]	Find the frequency of words	1	3	3	1M	map, flatMap, reduceByKey, filter
P9	FindSalary [51]	Total income of individuals earning ≤ $300 weekly	1	4	4	1M	map, filter, reduce
P10	StudentGrade [51]	List of classes with more than 5 failing students	1	4	4	1M	map,reduceByKey, filter
P11	MovieRating [19]	Total number of movies with rating ≥ 4	1	3	3	1M	map,reduceByKey, filter
P12	InsideCircle [51]	Check whether the point (x,y) is in a circle	1	2	2	1M	map,filter
P13	MapString [51]	String mapping	1	1	1	1M	map
P14	NumberSeries [51]	Find the numbers whose 3n+1 series' length is 25	1	3	3	1M	map,filter
P15	AgeAnalysis [51]	Total number of people with different age ranges	1	3	3	1M	map,filter
P16	IncomeAggregation [19]	Average income per age range in a district	1	5	5	1M	map, mapValues filter, reduceByKey
P17	LoanType [51]	The count of loan type within a region	1	2	2	1M	map

Table 2: Subject programs used in DepFuzz's evaluation. All programs represent real-world DISC use cases and are adopted from prior work. The data and code characteristics of benchmark programs are also shown.

and real-world DISC use cases [6, 8, 10, 15, 34, 35]. Collectively, our benchmark programs comprise (1) a variety of dataflow operators transformation (flatMap, map), fusion (join), and aggregation (reduce, group) operators, (2) UDFs, which are integral to DISC applications, and (3) both single and multiple input datasets, which are critical for practical data analysis.

The complete list of subject programs is shown in Table 2. For example, P7 [35] identifies the type of transportation used to perform the daily commutes *i.e.,* bus, car, or walk. It consolidates information on trips from two datasets to find the starting and destination zip codes, the distance traveled for the trip, and the time it took to cover this distance. Another program P2 is inspired by a commercial case study of Apache Spark [8]. It analyzes customer purchase history and rewards eligible customers (more than three instances of $100 spending in the current year) with coupons valued proportionally to spending. This is a multi-dataset program that joins the customer information table with the purchase history table. Overall, the benchmark programs' size is comparable to real-world industry DISC applications [49], which are in the order of hundreds of LOC but closed-sourced.

Baselines. We compare DepFuzz against two baselines: (1) a state-of-art schema-aware DISC application fuzzer, BigFuzz [51]; and (2) the most advanced commercial-grade coverage guided fuzzer for the JVM, Jazzer [23], developed in part by Google. We compare against these baselines because they are the state-of-the-art fuzzers for DISC applications and JVM-based applications, respectively. We use `scoverage` [42] to monitor Scala statement coverage of the applications. We provide BigFuzz with a seed input constructed by randomly sampling a row from the dataset, along with a schema of the dataset as in the original paper. For Jazzer, we write interfacing code that converts the random byte stream generated by Jazzer into formatted datasets expected by the DISC application.

Evaluation Environment. We run each tool for up to 24 hours, which is a standard experimental setting for fuzzing benchmarks, and measure statement coverage (%), cumulative error detection (%), and error depth (# of operators) in the dataflow graph of the benchmark programs. We perform these experiments on a 13-node cluster computing environment with 112 cores at 3.10GHz, 52TB storage, and 832GB memory. We run our experiments on Apache Spark 3.0 and HDFS 2.7.

4.1 Test Coverage Against Baseline

Figure 8 shows how cumulative statement coverage increases throughout the 24-hour fuzzing campaign with DepFuzz and the two baselines. Y-axis represents the percentage of statement coverage achieved, and the X-axis represents the time elapsed in seconds. DepFuzz significantly outperforms baselines for programs ingesting multiple input datasets and containing fusion, aggregation, and filter operators) such as P1-P5. For programs that ingest only a single dataset (*i.e.,* P6-P17), DepFuzz shows slightly better performance on average in terms of coverage.

Program P1's seventh operator is `join`, where each dataset's key is a concatenation of three columns. Since there are six co-dependent columns related by this equality and both baseline fuzzers mutate each of the six columns independently of the others, they fail to generate even a single input with matching keys to pass this `join`. Even with its schema-aware mutations, BigFuzz only achieves 28% coverage. Similarly, Jazzer struggles to push beyond 20% coverage with its byte-level mutations. DepFuzz manages to capture the co-dependence between six columns created by `join`. It immediately satisfies the constraints early in the fuzzing campaign through tailored mutations for fusion operators. DepFuzz achieves 99% coverage within 24 hours of fuzzing.

In P7, we observe a drastic increase in statement coverage in the first iteration of DepFuzz, compared to the baselines. This program uses an aggregation operator, `groupByKey`, followed by `filter` that requires a minimum number of rows with the same key to exercise the code after `filter`. Mutations that randomly duplicate any row are unaware of the aggregation's key column. Thus, baseline fuzzers do not generate the required rows to pass through `filter`. DepFuzz identifies the `groupByKey` along with the input column that influences the key. It then duplicates input rows to satisfy the joint constraint imposed by `aggregation` and `filter`. DepFuzz's superior performance in P2-P4 can be attributed to similar reasons.

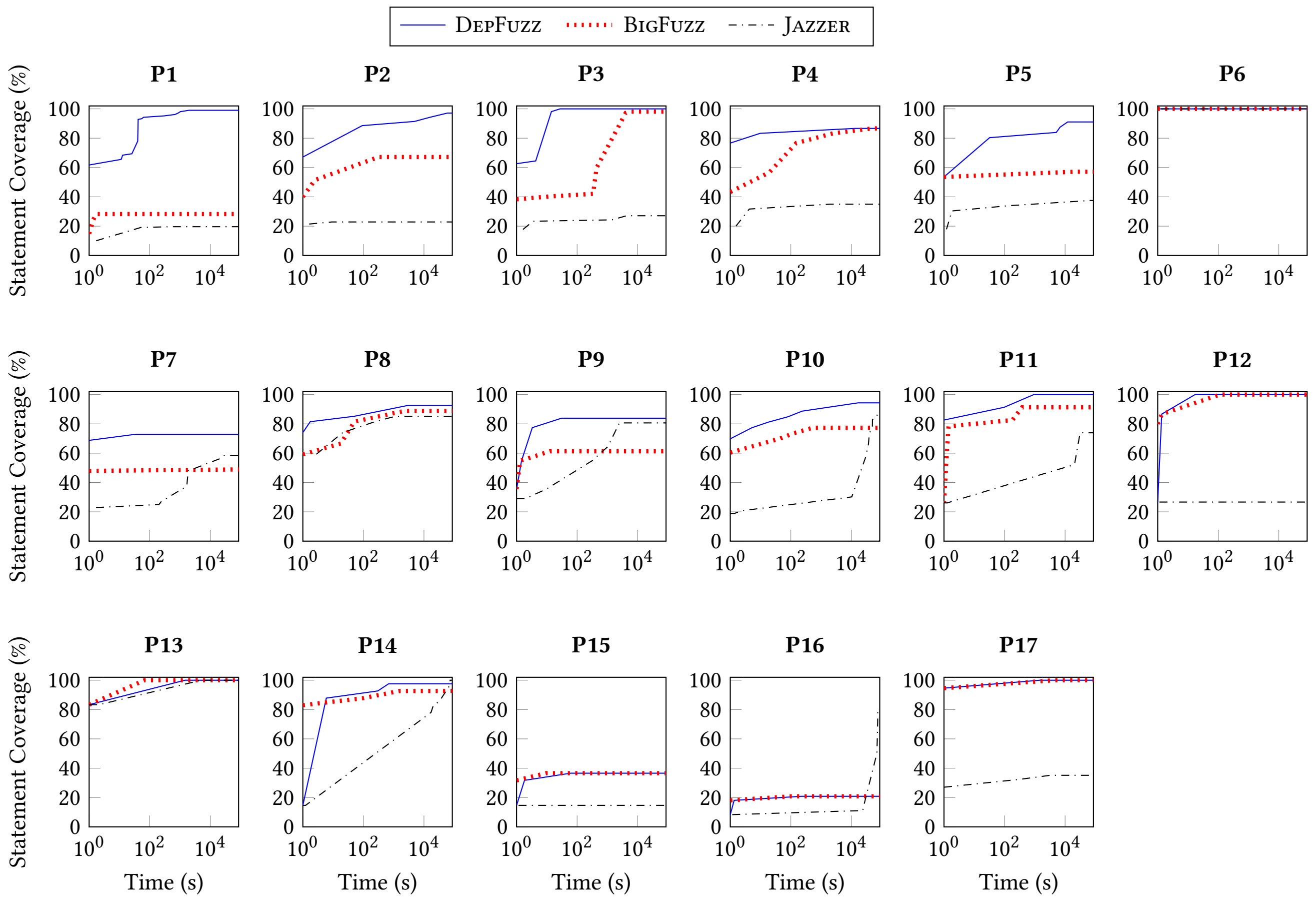


Figure 8: Statement coverage of three tools on 17 benchmark programs during 24 hours

DepFuzz also performs better for single dataset applications P6-P17 that do not have any fusion operators (due to only one input dataset), and their average dataflow operator depth is only three. DepFuzz performs 140K fewer but more effective fuzzing iterations than baselines on average due to the higher algorithmic complexity of applying co-dependent mutations. The baseline Jazzer performs better in P16 because some statements in the program are only reachable on *one specific* input value. The chances of reaching such statements (*e.g.,* stmt1 in `if(45<x<60){stmt1}`) are purely random. Thus, the technique with a higher number of iterations is more likely to reach these statements. In P16, Jazzer performs twice as many iterations as DepFuzz, which increases its likelihood of arbitrarily changing the input row from `90024,`**`28`**`,10990` to `90024,`**`46`**`,10990`, achieving additional statement coverage. In a 24-hour fuzzing campaign, DepFuzz achieves 29% higher coverage than Jazzer and 13% higher coverage than BigFuzz.

To answer RQ5, we evaluate how quickly DepFuzz achieves coverage compare to baselines by performing curve fitting with $y = mx$ as the objective function on the cumulative coverage graphs since the gradient of this line represents the average rate of gain of coverage over the course of the entire campaign. We find that DepFuzz is 1.3× faster than BigFuzz and 2.1× faster than Jazzer in terms of the coverage increase rate.

4.2 Fault Detection

We measure the fault detection capability of DepFuzz compared to the baselines. For this experiment, in each subject program, we inject one fault at each depth of a dataflow graph and then record the number of faults. We define a fault's depth as the number of dataflow operators an input row has to go through before reaching a faulty statement. For example, if a fault is seeded in a UDF f, where f is an argument to n^{th} dataflow operator, the fault is seeded at depth n. For example, the fault in 9-Figure 1 has a depth of five because there are five dataflow operators before the faulty code. We count only the faults triggered from correctly-formatted inputs, as Jazzer generates a massive number of ill-formatted inputs that all lead to parsing errors such as `ArrayIndexOutOfBound` exception from `split(",")[k]` due to missing k^{th} column in input data. Parsing errors are caused by processing ill-formatted inputs in a program. These errors normally appear in the first operation of a DISC application that takes a raw, unstructured input and parses it into individual data fields and their types *e.g.,* keys and values, similar to the first `map` applied on `dataset1` and `dataset2` in Figure 7. These errors do not appear if the input data format conforms to the program's parsing logic. We evaluate fault detection on two levels: 1) the total number of unique faults detected and 2) the depth within the dataflow graph at which a fault is detected. Note that each dataflow operator takes a UDF as an argument.

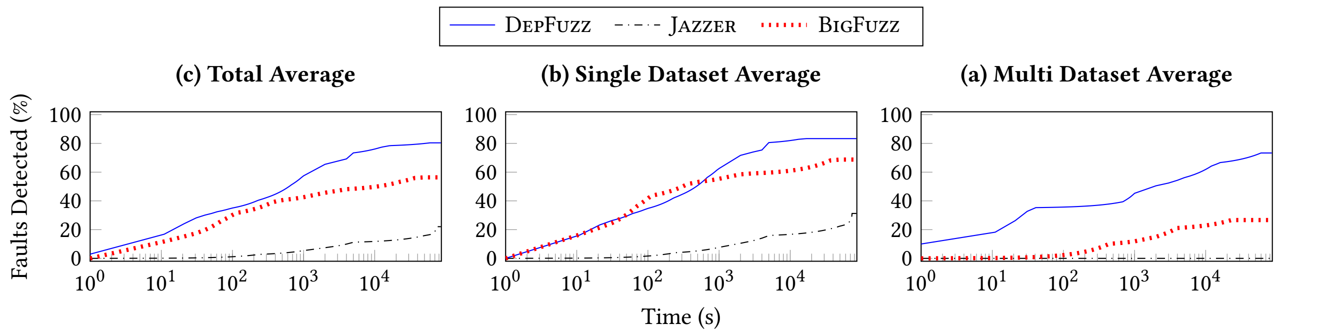


Figure 9: Cumulative number of faults detected during 24 hours averaged across 17 programs. (b) shows the average for P5-P17 which ingest a single dataset, and (c) shows average for P1-P4 which take multiple datasets as input.

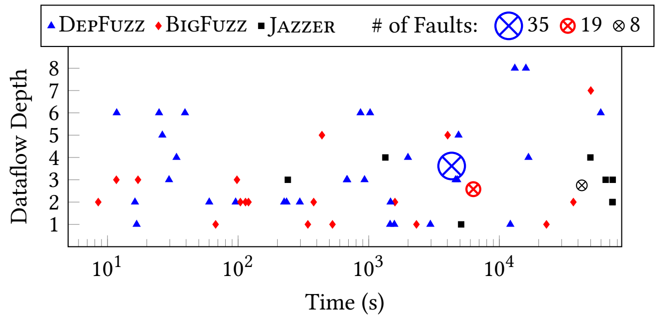


Figure 10: Depth vs Time for all faults detected by DepFuzz and baselines. ⊗ denotes centroids of each tool. DepFuzz has more points near the top left corner, which means it detects *deeper* faults *faster* than baselines. DepFuzz finds *more* faults than baselines.

Fault Injection. We manually inject faults into the subject programs by randomly replacing arithmetic operators, binary operators, and constants [25]. For example, `sqrt(1-a)` becomes `sqrt(.1-a)` after injecting a fault, which can lead to `NaN` error. Similarly, replacing operators like `+` with `/` will inject a division-by-zero error. Prior work on Apache Spark recognizes the presence of such faults in real-world DISC applications [19]. We also add faults by employing a range check that throws `RuntimeException` if a particular column value falls within a narrow range. For example, a faulty program throws an exception if a string value in a column starts with `"&%"`.

Fault injection is widely used in practice to evaluate new testing techniques. Automated fault injection tools such as LAVA [14] and Apocalypse [43] devise a set of principles that mimic properties of real-world faults. When injecting faults, we also follow these principles, which are as follows.

- *Rare:* The injected faults manifest for only a small fraction of possible inputs. We inject a fault that is triggered if the first column starts with the characters `"&%"`. The number of inputs that can trigger this fault is $\approx 256^{18}$. Assuming all inputs are equally likely, the probability of random mutations triggering this fault is ≈ 0.00002, assuming the row length of 20 ASCII characters. Note that this is an overestimate since, depending on where the fault is injected, several other control flow and data flow criteria will need to be met for the execution to reach the injected fault, further restricting the space of inputs that can trigger this fault.
- *Uncorrelated:* Finding one injected fault neither increases nor decreases the likelihood of finding any other faults.
- *Reproducible:* The faults are deterministic and reproducible in that a single input can prove the existence of a fault.
- *Fair:* The faults are injected in locations that can be feasibly reached by an automated technique. For example, no fault is guarded by a branch that requires solving an infeasible mathematical problem, such as factoring a large integer into its constituent primes.

In total, we inject 45 faults across 17 benchmark programs. Since the location of a fault may favor certain techniques, we ensure fairness in fault injection by injecting a fault at each data processing step in every program.

Fault Detection. Figure 9 shows the cumulative average number of faults detected on the subject programs. We report a summary of all detected program faults in Table 3. In Figure 9, the Y-axis represents the percentage of cumulative faults detected, and the X-axis represents the fuzzing duration. DepFuzz outperforms baseline techniques in terms of fault detection. For example, the majority of the inputs produced by Jazzer have an insufficient number of columns, which leads to data parsing errors (*i.e.,* `ArrayIndexOutOfBound` exception) after the `split` operation. Similarly, in P1, BigFuzz spends over 50% of its iterations triggering the same four parsing faults in the first UDF, causing only `NumberFormatException`. Table 3 lists the total faults detected by DepFuzz, BigFuzz, and Jazzer. On average, DepFuzz finds 3.4× more faults than Jazzer and 84% more faults than BigFuzz due to co-dependence aware mutations. DepFuzz's strengths in fault detection are noticeable in P1-P4 and P7, where co-dependence aware mutations help DepFuzz go past the fusion operators and reach deeper dataflow operators.

Detecting Deeper Program Faults. We stratify the injected faults by their dataflow operator depth. Figure 10 shows a scatter plot that visualizes the depth of the faults across 17 programs. The top of the plot represents deeper, hard-to-reach faults, whereas the bottom represents faults in the initial phases of the application.

The scatter plot shows that, overall, DepFuzz finds faults that reside at a deeper dataflow depth. In P1, for instance, DepFuzz finds a total of 7 faults across three different dataflow depths (3, 4, and 6), whereas both BigFuzz and Jazzer are unable to find any.

Program	Application Execution Time			Faults Detected		
	Original	Instrumented	Overhead	DepFuzz	BigFuzz	Jazzer
P1	9.4	36.2	3.9	7	0	0
P2	15.6	149.2	9.6	3	0	0
P3	54.8	768.4	14.0	3	2	0
P4	11.2	27.8	2.5	1	1	0
P5	17.4	174.2	10.0	1	1	0
P6	7.0	13.0	1.9	1	1	0
P7	7.0	17.6	2.5	1	0	0
P8	7.0	12.8	1.8	2	2	1
P9	5.8	6.6	1.1	2	2	1
P10	6.8	12.0	1.8	3	2	3
P11	6.6	12.0	1.8	4	1	0
P12	4.8	5.8	1.2	1	1	0
P13	5.0	5.6	1.1	1	1	1
P14	5.0	6.0	1.2	2	2	2
P15	5.0	5.8	1.2	1	1	0
P16	6.8	8.0	1.2	1	1	0
P17	4.8	6.0	1.2	1	1	0
Total Faults Detected				**35**	**19**	**8**

Table 3: Running time of the original subject program and the instrumented program with taint analysis along with total errors detected by each tool.

The plot also shows that DepFuzz is consistently faster at finding deep faults than baselines. For example, in P3, the deepest bug is triggered by BigFuzz a little over an hour into the fuzzing campaign, whereas DepFuzz finds it within the first minute. Although the time difference is smaller for single dataset programs (P6-P17), a similar pattern can be observed. For example, in P14, DepFuzz finds the deepest bug within the first two minutes, whereas BigFuzz takes over 13 minutes. Figure 10 also shows the centroids with $\otimes$ for each tool. The size of $\otimes$ represents the number of detected faults. Note that the gaps between the centroids are larger than they appear due to the log-scaled x-axis. On average, the deepest faults detected by DepFuzz are 1.1 operators deeper than BigFuzz and 0.9 operators deeper compared to Jazzer.

4.3 DepFuzz's Instrumentation Overhead

DepFuzz enables dynamic taint analysis in a trial execution (*i.e.,* running an instrumented program on the original input data) to identify co-dependence relationships. Note that this is a *one-time overhead for the first run* and is not a recurring overhead for each fuzzing iteration because its goal is to infer co-dependence constraints from existing data.

Table 3 shows the time difference between an instrumented run and an uninstrumented run on the original input datasets. For instance, in program P1, the trial execution for dynamic taint analysis takes 36.2 seconds, whereas the original program takes 9.4 seconds to process the same amount of data. This overhead is higher in programs with multiple datasets, aggregator operators, and fusion operators, P1-P5, as they introduce complex dependencies among columns and rows. These co-dependences are represented in dense taint objects (*i.e.,* RoaringBitmaps [11]). Across the 17 programs, the first instrumented run's overhead is 1.1× to 14× of the first uninstrumented run. Note that this overhead is a one-time upfront cost and the rest of the fuzzing loop does not require running an instrumented version with taint monitors; therefore, in the long run, the cost of using DepFuzz becomes negligible compared to many hours of fuzz testing. DepFuzz's runtime overhead is on par with other taint analysis approaches on DISC applications [45].

5 RELATED WORK

Fuzzing has gained popularity in industry and academia recently due to its black-box nature and ease of adoption [37]. A common challenge in fuzzing is generating structurally valid inputs. Zest [39] attempts to generate valid inputs using parametric generators. BigFuzz [51] uses framework abstraction to reduce fuzz testing latency. However, BigFuzz is a simple random fuzzer and cannot identify co-dependent regions in the input. Symbolic execution techniques [19, 28, 29, 36] exist for testing DISC applications. However, they cannot easily generate constraints that respect co-dependence relationships within multiple datasets, created by the complex interaction between dataflow operators and UDFs. Random testing bears similarity to fuzz testing [13, 32, 38, 41]. Randoop [38] and EvoSuite [16] generate test suites for the program under test to cause program crashes.

The closest line of work to ours is taint-based fuzzing. At a high level, all taint analysis techniques attempt to isolate regions within an input critical to mutate. For example, Bekrar *et. al.* [7] propose taint-based fuzzing that identifies input regions to focus mutations. TaintScope [48] and BuzzFuzz [17] isolate regions of the input inside a sensitive library and system calls. PATA [31] performs path-aware taint analysis to mitigate the problems of over-tainting and under-tainting by employing path information. Although these techniques isolate critical input regions, none target DISC applications and none can discover underlying co-dependence relations by analyzing dataflow operators and UDFs. The inputs to DISC applications are very large and consist of multiple datasets; so existing taint tracking at a byte-level is also inefficient. DepFuzz addresses these problems by handling multiple datasets and by tracking taints at the level of dataset IDs, columns, and rows from unstructured inputs.

The idea of triggering *hard-to-reach* regions of the program has been seen frequently in the literature. FairFuzz [27] is a targeted mutation strategy that avoids mutating input regions that trigger rare branches, similar to how DepFuzz analyzes the use of fusion operators to co-mutate certain regions. However, FairFuzz uses coverage feedback and a simple masking strategy to freeze contiguous input regions. AFLFast [9] prioritizes inputs that trigger rare *paths* in the code. AFLFast instruments program binary and perform runtime coverage analysis. Both FairFuzz and AFLFast are not suitable for DISC applications because they do not analyze dataflow operator usages and internal UDF semantics to infer co-dependent input regions in large datasets. Neither perform provenance-aware duplication to resolve aggregations, which are extremely common in DISC applications. Driller [46] switches to using symbolic execution to resolve a difficult branch that AFL fails to pass, causing it to inherit the limitations of symbolic execution. Steelix [30] attempts to produce a single input passing a difficult-to-hit branch in the code and employs source-level instrumentation similar to DepFuzz. Steelix is not suitable for DISC applications with large inputs due to a lack of fine-grained data tracking.

TaintStream [50] implements cell-level provenance for Apache Spark in the context of Policy Enforcement. DepFuzz also tracks provenance at the cell level. However, TaintStream requires extending the original dataset with tags, whereas DepFuzz's cell level tracking is fully automatic and does not require converting the original dataset. DepFuzz's taint analysis is similar to that of FlowDebug [47], as they both instrument primitive data types and application code and do not require any modifications to the original datasets to enable taint analysis. However, FlowDebug concerns taint analysis only and does not generate test data nor does it identify co-dependency constraints among input datasets. Furthermore, existing data provenance techniques [12, 20–22, 33] perform taint analysis only at the row level, support only a single dataset, and do not support tracking at the column (cell) level. Spark-specific data provenance solutions also exist, such as Titian [24], but it is limited to row-level data provenance for only a single input dataset. BigSift [18] is an extension of delta debugging for DISC applications but its isolation works at the level of rows, not the level of dataset IDs, rows, and columns, unlike DepFuzz.

6 CONCLUSION

Traditional fuzzing is ineffective for DISC applications due to requirements to handle unstructured inputs, a lack of schema, the inability to handle multiple datasets, and their large input size. In this work, we introduce DepFuzz, a technique that uses fine-grained provenance tracking to infer complex co-dependence constraints created by dataflow operators and user-defined functions. The key insight behind DepFuzz is to orchestrate co-dependence aware mutations on multiple input datasets in concert. DepFuzz increases code coverage fast, finds more defects, and finds defects that are hard to find—29% higher statement coverage, 2.1× faster, and triggering faults that are 0.9 operators deeper than the ones found by the state of the art commercial fuzzer for JVM.

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under grant numbers 2106420, 1764077, 1956322, 1460325, 2106383 and 2106404. It is also supported in part by funding from Amazon and Samsung. We want to thank the anonymous reviewers for their constructive feedback that helped improve the work.

REFERENCES

[1] 2021. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2021-12-14.
[2] 2022. Apache Hadoop. https://hadoop.apache.org/. Accessed: 2021-12-14.
[3] 2022. Apache Spark. https://spark.apache.org/. Accessed: 2021-12-14.
[4] Accessed: 2022-09-01. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/
[5] Accessed: 2022-09-01. TPC-DS Version 2 and Version 3. https://www.tpc.org/tpcds/default5.asp
[6] Accessed: 2023-01-10. Demonstration Database. https://postgrespro.com/community/demodb.
[7] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2012. A Taint Based Approach for Smart Fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 818–825. https://doi.org/10.1109/ICST.2012.182
[8] Alexander Boyce and Mathieu Leger. Accessed: 2023-01-10. Stateful Streaming with Apache Spark: How to Update Decision Logic at Runtime. https://www.databricks.com/session_eu20/stateful-streaming-with-apache-spark-how-to-update-decision-logic-at-runtime
[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841
[10] Deepayan Chakrabarti, Ravi Kumar, and Kunal Punera. 2008. A Graph-Theoretic Approach to Webpage Segmentation. In *Proceedings of the 17th International Conference on World Wide Web* (Beijing, China) *(WWW '08)*. Association for Computing Machinery, New York, NY, USA, 377–386. https://doi.org/10.1145/1367497.1367549
[11] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with roaring bitmaps. *Software: practice and experience* 46, 5 (2016), 709–719.
[12] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. 2016. Explaining Outputs in Modern Data Analytics. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1137–1148. https://doi.org/10.14778/2994509.2994530
[13] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience* 34, 11 (2004), 1025–1050.
[14] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15
[15] Fariza Fauzi, Jer-Lang Hong, and Mohammed Belkhatir. 2009. Webpage Segmentation for Extracting Images and Their Surrounding Contextual Information. In *Proceedings of the 17th ACM International Conference on Multimedia* (Beijing, China) *(MM '09)*. Association for Computing Machinery, New York, NY, USA, 649–652. https://doi.org/10.1145/1631272.1631379
[16] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.
[17] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 474–484.
[18] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. 2017. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing*. 520–534.
[19] 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 290–301. https://doi.org/10.1145/3338906.3338953
[20] R. Ikeda, J. Cho, C. Fang, S. Salihoglu, S. Torikai, and J. Widom. 2012. Provenance-Based Debugging and Drill-Down in Data-Oriented Workflows. In *2012 IEEE 28th International Conference on Data Engineering*. 1249–1252. https://doi.org/10.1109/ICDE.2012.118
[21] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for generalized map and reduce workflows. In *In Proc. Conference on Innovative Data Systems Research (CIDR)*.
[22] R. Ikeda, A. Das Sarma, and J. Widom. 2013. Logical provenance in data-oriented workflows?. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 877–888. https://doi.org/10.1109/ICDE.2013.6544882
[23] Code Intelligence. 2022. Jazzer. https://github.com/CodeIntelligenceTesting/jazzer.
[24] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data provenance support in spark. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 9. NIH Public Access, 216.
[25] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 433–436. https://doi.org/10.1145/2610384.2628053
[26] Krishna M. Kavi, Bill P. Buckles, and U. Narayan Bhat. 1986. A formal definition of data flow graph models. *IEEE Transactions on computers* 35, 11 (1986), 940–948.
[27] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.
[28] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner. 2013. SEDGE: Symbolic example data generation for dataflow programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 235–245. https://doi.org/10.1109/ASE.2013.6693083
[29] Nan Li, Yu Lei, Haider Riaz Khan, Jingshu Liu, and Yun Guo. 2016. Applying combinatorial test data generation to big data applications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 637–647.
[30] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 627–637.
[31] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 154–170. https://doi.org/10.1109/SP46214.2022.00010

[32] Yu Lin, Xucheng Tang, Yuting Chen, and Jianjun Zhao. 2009. A divergence-oriented approach to adaptive random testing of Java programs. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 221–232.

[33] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable lineage capture for debugging DISC analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 17.

[34] Ehsan Mazloumi, Graham Currie, and Geoffrey Rose. 2010. Using GPS data to gain insight into public transport travel time variability. *Journal of transportation engineering* 136, 7 (2010), 623–631.

[35] Farhad Nabhani and Alireza Shokri. 2009. Reducing the delivery lead time in a food distribution SME through the implementation of six sigma methodology. *Journal of manufacturing technology Management* 20, 7 (2009), 957–974.

[36] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. 2009. Generating Example Data for Dataflow Programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/1559845.1559873

[37] Alessandro Orso and Gregg Rothermel. 2014. Software testing: a research travelogue (2000–2014). In *Future of Software Engineering Proceedings*. 117–132.

[38] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[39] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[40] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[41] IS Prasetya. 2013. T3, a combinator-based random testing tool for java: benchmarking. In *International Workshop on Future Internet Testing*. Springer, 101–110.

[42] Roch, Grzegorz Slowikowski, Roland Tritsch, Sam, and Chris Kipp. 2022. scoverage. https://github.com/scoverage. Accessed: 2022-01-10.

[43] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/3236024.3236084

[44] Kostya Serebryany. Accessed: 2023-01-29. LibFuzzer – a library for coverage-guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html

[45] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. 2020. Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1527–1543.

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[47] Jason Teoh, Muhammad Ali Gulzar, and Miryung Kim. 2020. Influence-based provenance for dataflow applications with taint propagation. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 372–386.

[48] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *2010 IEEE Symposium on Security and Privacy*. 497–512. https://doi.org/10.1109/SP.2010.37

[49] Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, Huan He, and Haibo Lin. 2019. Niijima: Sound and Automated Computation Consolidation for Efficient Multilingual Data-Parallel Pipelines. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 306–321. https://doi.org/10.1145/3341301.3359649

[50] Chengxu Yang, Yuanchun Li, Mengwei Xu, Zhenpeng Chen, Yunxin Liu, Gang Huang, and Xuanzhe Liu. 2021. *TaintStream: Fine-Grained Taint Tracking for Big Data Platforms through Dynamic Code Translation*. Association for Computing Machinery, New York, NY, USA, 806–817. https://doi.org/10.1145/3468264.3468532

[51] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2021. BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 722–733. https://doi.org/10.1145/3324884.3416641

Received 2023-02-02; accepted 2023-07-27