White-Box Testing of Big Data Analytics with Complex User-Defined Functions

Muhammad Ali Gulzar
Shaghayegh Mardani
University of California, Los Angeles
USA

Madanlal Musuvathi
Microsoft Research, USA

Miryung Kim
University of California, Los Angeles
USA

ABSTRACT

Data-intensive scalable computing (DISC) systems such as Google's MapReduce, Apache Hadoop, and Apache Spark are being leveraged to process massive quantities of data in the cloud. Modern DISC applications pose new challenges in exhaustive, automatic testing because they consist of dataflow operators, and complex user-defined functions (UDF) are prevalent unlike SQL queries. We design a new white-box testing approach, called BIGTEST to reason about the internal semantics of UDFs in tandem with the equivalence classes created by each dataflow and relational operator.

Our evaluation shows that, despite ultra-large scale input data size, real world DISC applications are often significantly skewed and inadequate in terms of test coverage, leaving 34% of Joint Dataflow and UDF (JDU) paths untested. BIGTEST shows the potential to minimize data size for local testing by 10^5 to 10^8 orders of magnitude while revealing 2X more manually-injected faults than the previous approach. Our experiment shows that only few of the data records (order of tens) are actually required to achieve the same JDU coverage as the entire production data. The reduction in test data also provides CPU time saving of 194X on average, demonstrating that *interactive* and *fast* local testing is feasible for big data analytics, obviating the need to test applications on huge production data.

CCS CONCEPTS

• **Software and its engineering** → **Cloud computing**; **Software testing and debugging**; • **Information systems** → **MapReduce-based systems**.

KEYWORDS

symbolic execution, dataflow programs, data intensive scalable computing, map reduce, test generation

ACM Reference Format:
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 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338953

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338953

1 INTRODUCTION

Data-intensive scalable computing (DISC) systems such as Mapreduce [20], Apache Hadoop [1], Apache Spark [49] are commonly used today to process terabytes and petabytes of data. At this scale, rare and buggy corner cases frequently show up in production [50]. Thus, it is common for these applications to either crash after running for days or worse, silently produce corrupted output. Unfortunately, the common industry practice for testing these applications remains running them locally on randomly sampled inputs, which obviously does not flush out bugs hiding in corner cases.

This paper presents a systematic input generation tool, called BIGTEST, that embodies a new white-box testing technique for DISC applications. BIGTEST is motivated by the recent successes of systematic test generation tools [22, 24, 40]. However, the nature of DISC applications requires extending these in important ways to be effective. Unlike general-purpose programs addressed by existing testing tools, DISC applications use a combination of relational operators, such as `join` and `group-by`, and dataflow operators, such as `map`, `flatmap`, along with user-defined functions (UDFs) written in general purpose languages such as C/C++, Java, or Scala.

In order to comprehensively test DISC applications, BIGTEST reasons about the *combined* behavior of UDFs with relational and dataflow operations. A trivial way is to replace these dataflow operations with their implementations and symbolically execute the resulting program. However, existing tools are unlikely to scale to such large programs, because dataflow implementation consists of almost 700 KLOC in Apache Spark. Instead, BIGTEST includes a logical abstraction for dataflow and relational operators when symbolically executing UDFs in the DISC application. The set of combined path constraints are transformed into SMT queries and solved by leveraging an off-the-shelf theorem prover, Z3 or CVC4, to produce a set of concrete input records [11, 19]. By using such a combined approach, BIGTEST is more effective than prior DISC testing techniques [31, 34] that either do not reason about UDFs or treat them as uninterpreted functions.

To realize this approach, BIGTEST tackles three important challenges that our evaluation shows are crucial for the effectiveness of the tool. First, BIGTEST models *terminating* cases in addition to the usual *non-terminating* cases for each dataflow operator. For example, the output of a `join` of two tables only includes rows with keys that match both the input tables. To handle corner cases, BIGTEST carefully considers terminating cases where a key is only present in the left table, the right table, and neither. Doing so is crucial, as based on the actual semantics of the join operator, the output can contain rows with null entries, which are an important source of bugs. Second, BIGTEST models collections explicitly, which are created by `flatmap` and used by `reduce`. Prior approaches [31, 34] do

not support such operators, and thus are unable to detect bugs if code accesses an arbitrary element in a collection of objects or if the aggregation result is used within the control predicate of the subsequent UDF. Third, BIGTEST analyzes string constraints because string manipulation is common in DISC applications and frequent errors are ArrayIndexOutOfBoundException and StringIndexOutOfBoundsException during segmentation and parsing.

To evaluate BIGTEST, we use a benchmark set of 7 real-world Apache Spark applications selected from previous work such as PigMix[35], TITIAN [28], and BIGSIFT [25]. While these programs are representative of DISC applications, they do not adequately represent failures that happen in this domain. To rectify this problem, we perform a survey of DISC application bugs reported in Stack Overflow and mailing lists and identify seven categories of bugs. We extend the existing benchmarks by manually introducing these categories of faults into a total of 31 faulty DISC applications. To the best of our knowledge, this is the first set of DISC application benchmarks with representative real-world faults. Such benchmarks are crucial for further research in this area.

We assess JDU (Joint Dataflow and UDF) path coverage, symbolic execution performance, and SMT query time. Our evaluation shows that real world datasets are often significantly skewed and inadequate in terms of test coverage of DISC applications, still leaving 34% of JDU paths untested. Compared to SEDGE [31], BIGTEST significantly enhances its capability to model DISC applications—In 5 out of 7 applications, SEDGE is unable to handle these applications at all, due to limited dataflow operator support and in the rest 2 applications, SEDGE covers only 23% of paths modeled by BIGTEST.

We show that JDU path coverage is directly related to improvement in fault detection—BIGTEST reveals 2X more manually injected faults than SEDGE on average. BIGTEST can minimize data size for local testing by 10^5 to 10^8 orders of magnitude, achieving the CPU time savings of 194X on average, compared to testing code on the entire production data. BIGTEST synthesizes concrete input records in 19 seconds on average for all remaining untested paths. Below, we highlight the summary of contributions.

- BIGTEST is the first piece of DISC white-box testing that comprehensively models dataflow operators and the internal paths of user-defined functions in tandem.
- BIGTEST makes three important enhancements to improve fault detection capability for DISC applications—(1) It considers both *terminating* and *non-terminating* cases of each dataflow operator; (2) It explicitly models collections created by `flatmap` and translates aggregation logic into an iterative aggregator; and (3) It models string constraints explicitly.
- It puts forward a benchmark of manually injected DISC application faults along with generated test data, inspired by the characteristics of real world DISC application faults evidenced by Stack Overflow and mailing lists.
- BIGTEST finds 2X more faults than SEDGE, minimizes test data by orders of magnitude, and is fast and interactive.

Our results demonstrate that *interactive* local testing of big data analytics is feasible, and that developers should not need to test their program on the entire production data. For example, a user may monitor path coverage with respect to the equivalent classes of paths generated from BIGTEST and skip records if they belong to the already covered path, constructing a minimized sample of the production data for local development and testing.

```
1  val x,y,z;
2  if(x<y)
3    z = y/x; //PC1: x < y = true, Effect: z=y/x
4  else
5    z = x/y; //PC2: x >= y = true, Effect: z=x/y
```

Figure 1: Symbolic PathFinder produces a set of path constraints and their corresponding effects

```
1  val trips = sc.textFile("trips_table.csv")
2    .map{  s =>
3       val cols = s.split(",")
4  ❶    (cols(1),cols(3).toInt/cols(4).toInt) }
5       //Returns location and speed
6  val zip = sc.textFile("zipcode_table.csv")
7    .map{  s =>
8       val cols = s.split(",")
9  ❷    (cols(1),cols(0) }
10      // Returns location and its name
11   .filter{
12 ❸    s => s._ 2 == "Palms" }
13 val joined = trips.join(zip)
14 joined
15    .map{  s =>
16      if (s._2._1 > 40) ("car",1)
17 ❹    else if (s._2._1 > 15) ("bus",1)
18      else ("walk",1)
19   }
20   .reduceByKey(_+_)❺
21   .saveAsTextFile("hdfs://...")
```

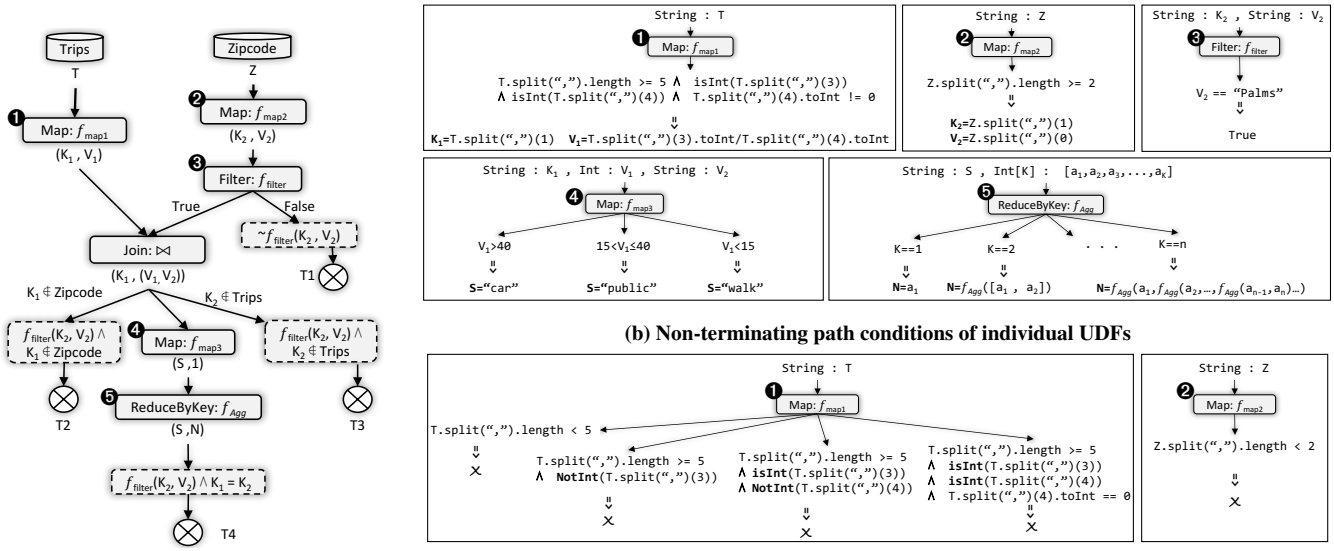
Figure 2: Alice's program estimates the total number of trips originated from "Palms."

The rest of the paper is organized as follows. Section 2 provides a brief introduction to Apache Spark and symbolic execution. Section 3 describes a motivating example. Section 4 describes the design of BIGTEST. Section 5 describes evaluation settings and results. Section 6 discusses related work. Section 7 concludes with future work.

2 BACKGROUND

Apache Spark. BIGTEST targets Apache Spark, a widely used data intensive scalable computing system. Spark extends the MapReduce programming model with direct support for dataflow and traditional relational algebra operators (*e.g.,* `group-by`, `join`, and `filter`). Datasets can be loaded in Spark runtime using several APIs that create Resilient Distributed Datasets (RDDs), an abstraction of distributed collection [48]. RDDs can be transformed by invoking dataflow operations on them (*e.g.,* `val filterRdd = rdd.filter(_ >5)`). Dataflow operators such as `map`, `reduce`, and `flatmap` are implemented as higher-order functions that take a user defined function (UDF) as an input parameter. The actual evaluation of an RDD occurs when an action such as `count` or `collect` is called. Internally, Spark translates a series of RDD transformations into a Directed Acyclic Graph (DAG) where each vertex represents a transformation applied to the incoming RDD. The Spark scheduler executes each stage in a topological order.

Symbolic Execution using Java Path Finder. BIGTEST builds on Symbolic Java PathFinder (SPF) [37]. Internally, SPF relies on the



(a) Dataflow operators' paths by BIGTEST

(b) Non-terminating path conditions of individual UDFs

(c) Path constraints for terminating paths in UDFs

Figure 3: Solid and dotted boxes represent transformations and path constraints, respectively. BIGTEST **identifies path constraints for both non-terminating and terminating program paths while symbolically executing the program.**

analysis engine of Java PathFinder (JPF) model checking [44]. It interprets Java bytecode on symbolic inputs and produces a set of symbolic constraints. Each constraint represents a unique path in the program, and can be ingested by a theorem solver to generate test inputs. Figure 1 illustrates an example symbolic execution result. By attaching listeners to SPF, the path conditions and the effects of each path can be captured. For this program, SPF produces two path conditions: (1) the first path produces the effect of `z=y/x`, when the path condition `x<y` holds true and (2) the second path produces `z=x/y` as an effect, when the path condition `x≥y` is satisfied.

3 MOTIVATING EXAMPLE

This section presents a running example to motivate BIGTEST. Suppose that Alice writes a DISC application in Spark to analyze the Los Angeles commuting dataset. She wants to find the total number of trips originating from the "Palms" neighborhood using: (1) a public transport whose speed is assumed to be faster than 15 but slower than 40 mph, (2) a personal vehicle which is estimated to be faster than 40 mph, and (3) on foot which is estimated as slower than 15 mph. Each row in the Trips dataset represents a unique identifier for the trip, the start and end location in terms of a zip code, the trip distance in miles, and the trip duration in hours, for example, `1,90034,90024, 10, 1`. To map an area zip code to its corresponding area name, Alice uses another dataset that assigns a name to each zip code in the following manner: `90034,Culver City`

To perform this analysis, Alice writes a Spark application in Figure 2. She loads both datasets (lines 1 and 6), parses each dataset, selects the start location of a trip as a key, and computes the average speed as a value by dividing the distance by duration (lines 2-4). Alice outputs a zip code as a key and an area name as a value (lines 7-9) and filters the area name with "Palms" at line 12. She joins the two data sets (line 13). In the subsequent `map` operation (line 15-18), she categorizes the trips based on the average speed into three categories. She finally counts the frequency of each trip kind and stores them (lines 20 and 21). Though this program is only 21 lines long, it poses several challenges for modeling test paths.

Equivalence Classes of Dataflow Operators. Consider `filter` ❸ at line 11. To exhaustively test this operator, we must consider two equivalence classes: the first where a data record satisfies the filter and moves onto the next operator and the second where the filter does not satisfy and its data flow terminates. If we only model non-terminating case then test data would contain passing data records only and hence, would not detect a fault in which `filter` is removed from the DISC application. To model `join` at line 13, we must have three equivalence classes—two terminating cases and one non-terminating case: (1) an input record in the left table ("Trip") does not have a matching key on the right table ("ZipCode"), terminating its data flow, (2) an input in the right table does not have a matching key on the left, terminating its data flow, and (3) there exists a key that appears in both tables, passing the joined result to the next operator. Modeling such terminating cases is crucial otherwise test data generated produce the same output for both `join` and `leftOuterJoin` and do not reveal faults that are based on incorrect `join` type usage.

UDF Paths. Consider the last `map` ❹ at lines 15-18. There are three internal path conditions: (1) the speed > 40mph, (2) the speed is > 15 mph and ≤ 40 mph, and (3) the speed is ≤ 15 mph. The sub figure ❹ in Figure 3b shows corresponding path conditions and effects.

String Constraints. To analyze the second `map` ❷ at lines 7 to 9, we must reason about the entailed string constraints. Given a string `Z` in sub figure ❷ in Figure 3b and 3a, to split the data into two columns, it must satisfy a string constraint `Z.split(",").length ≥ 2` to produce the effect where the key K_2 is `(Z.split(",") (1)` and the value V_2 is `(Z.split(",")(0))`. String manipulation is critical to many DISC applications. In the above example, at least one test must contain a string `z` without delimiter `","`, so that `z.split(",")(1)` leads to ArrayIndexOutOfBoundsException which will then expose the inability of the UDF to handle exceptions.

#	CONSTRAINT	TRIPS	ZIPCODE
C1	`T.split(",").length < 5`	`""`	`_ _`
C2	`T.split(",").length ≥ 5 ∧ NotInt(T.split(",")(3))`	`_, _, _, "", _`	`_, _`
C3	`T.split(",").length ≥ 5 ∧ isInt(T.split(",")(3)) ∧ NotInt(T.split(",")(4))`	`_, _, _, "-2", ""`	`_, _`
C4	`T.split(",").length ≥ 5 ∧ isInt(T.split(",")(3)) ∧ isInt(T.split(",")(4)) ∧ T.split(",")(4).toInt = 0`	`_, _, _, "-2", "0"`	`_, _`
C5	`Z.split(",").length < 2`	`_`	`""`
C6	`Z.split(",").length ≥ 2 ∧ V2 != "Palms"`	`_`	`_, "\x00"`
C7	`T.split(",").length ≥ 5 ∧ isInt(T.split(",")(3)) ∧ isInt(T.split(",")(4)) ∧ T.split(",")(4).toInt != 0 ∧ Z.split(",").length ≥ 2 ∧ V2 = "Palms" ∧ K1 ∉ Zipcode`	`_, "!0!", _, _, _`	`"\x00", "Palms"`
C8	`... ∧ V2 = "Palms" ∧ K2 ∉ Trips`	`_, ""!0!", _, _, _`	`"\x00", "Palms"`
C9	`... ∧ V2 = "Palms" ∧ K1= K2 ∧ V1 > 40`	`_, "\x00", _, "41", "1"`	`"\x00", "Palms"`
C10	`... ∧ V2 = "Palms" ∧ K1= K2 ∧ 15< V1 < 40`	`_, "\x00", _, "16", "1"`	`"\x00", "Palms"`
C11	`... ∧ V2 = "Palms" ∧ K1= K2 ∧ V1 < 15`	`_, "\x00", _, "0", "1"`	`"\x00", "Palms"`

Table 1: Generated input data where each row represents a unique path. Variables `T`, `Z`, `V`, and `K` are defined in Figure 3a.

Otherwise, this application may crash in production, when the input record does not have an expected delimiter.

Arrays. To analyze `reduceByKey` ❺ at line 20 (also in Figure 3b), we must model how the UDF operates on the input array of size `K`, `[a1,a2,...,aK]` and produces the corresponding output f_{agg}(a1 , f_{agg}(a2 ... f_{agg}(aK−1,aK)...)). For example, the UDF `(_+_)` returns the sum of two input arguments. When the array size `K` is given by a user, the final output N is a1+(a2+...(aK−1+aK)).

Summary. Due to the internal path conditions entailed by individual UDFs, instead of four high-level dataflow paths shown in Figure 3a, Alice must consider eleven paths in total, which are enumerated in Table 1. Figure 3 shows the symbolic execution tree at the level of dataflow operators on the left and the internal symbolic execution trees for individual UDFs on the right. Lastly, example data generated by BIGTEST for each JDU path using Z3 is shown in Table 1. While these example data records may not look realistic, such data is necessary to exercise the downstream UDFs that are otherwise unreachable with the original dataset. For instance, filtering a dataset without any passing data record will result in an empty set and consequently, the UDFs after the `filter` will never get tested with the original data. Therefore, synthetic data is necessary and crucial to expose downstream program behavior.

4 APPROACH

BIGTEST takes in an Apache Spark application in Scala as an input and generates test inputs to cover all paths of the program up to a given bound by leveraging theorem provers Z3 [19] and CVC4 [11].

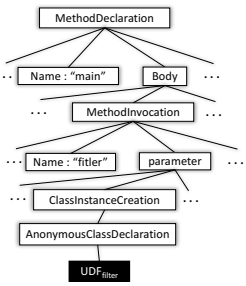
4.1 Dataflow Program Decomposition

A DISC application is comprised of a direct acyclic graph where each node represents a dataflow operator such as `reduce` and corresponding UDFs. As the implementation of dataflow operators in Apache Spark spans several hundred thousand lines of code, it is not feasible to perform symbolic execution of a DISC application along with the Spark framework code. Instead, we abstract the internal implementation of a dataflow operator in terms of logical specifications. We decompose a DISC application into a dataflow graph where a node calls each UDF and combine the symbolic execution of the UDFs using the logical specification of dataflow operators.

UDF Extraction. BIGTEST compiles the DISC application into Java bytecode and traverses each Abstract Syntax Tree (AST) to search for a method invocation corresponding to each dataflow operator. The input parameters of such method invocation are UDFs represented as anonymous functions as illustrated in Figure 4b. BIGTEST

```
1  sc.textFile("zipcode.csv").map{...}
2    .filter{_._2 == "Palms"}
```

(a) DISC Application



(b) Generated AST

```
1  class filter{
2   static void main(String args[]){
3    apply(null);
4   }
5   static boolean apply(Tuple2 s){
6    return s._2().equals("Palms")
7   }
8  }
```

(c) Extracted Filter UDF

Figure 4: BIGTEST extracts UDFs corresponding to dataflow operators through AST traversal.

stores the UDF as a separate Java class shown in Figure 4c and generates a configuration file required by JPF for symbolic execution. BIGTEST also performs dependency analysis to include external classes and methods referenced in the UDF.

```
1  def f(a:Int,b:Int){
2    return a+b;
3  }
4  //Usage in reduce
5  ...reduce{f}
```

(a)

```
1  def f_reduce(arr:Array[Int]){
2   var sum = 0;
3   for(a <- 1 to K)//K is bound
4     sum = udf(sum,arr(a));
5   return sum; }
```

(b)

Figure 5: (a) a normal invocation of `reduce` with a corresponding UDF. (b) an equivalent iterative version with a bound K

Handling Aggregator Logic. For aggregation operators, the attached UDF must be transformed. For example, the UDF for `reduce` is an associative binary function, which performs incremental aggregation over a collection shown in Figure 5a. We translate it into an iterative version with a loop shown in Figure 5b. To bound the search space of constraints, we bound the number of iterations to a user provided bound `K` (default is 2).

4.2 Logical Specifications of Dataflow Operators

This section describes the equivalence classes generated by each dataflow operator's semantics. We use C_I to represent a set of path constraints on the input data, I, for a particular operator. A single element c in C_I contains path constraints that must be satisfied to

Operator	Inputs	Logical Specification	
filter(udf)	I: Input Table $udf : t \rightarrow Bool$	Non-Terminating	❶ $\exists t$: $t \in I \wedge c \in C_I \wedge c(t) \wedge f(t)$
		Terminating	❷ $\exists t$: $t \in I \wedge c \in C_I \wedge c(t) \wedge \neg f(t)$
map(udf)	I : Input Table, O : Output Table $udf : t \rightarrow t'$ where $t' \in O$	Non-Terminating	❸ $\exists t$: $t \in I \wedge c \in C_I \wedge c(t) \wedge f(t)$
flatmap(udf)	I : Input Table, O: Output Table $udf : t \rightarrow Collection\ of\ t'$ $t' \in O$	Non-Terminating	❹ $\exists t$: $t_I \in I \wedge c \in C_I \wedge c(t) \wedge f(t)$
join	R : Right Table, $t_R \in R$ L : Left Table, $t_L \in L$	Non-Terminating	❺ $\exists t_R, t_L$: $c_R \in C_R \wedge c_L \in C_L \wedge c_R(t_R) \wedge t_{R,key} = t_{L,key} \wedge c_L(t_L)$
		Terminating	❻ $\exists key, t_R$: $c_R \in C_R \wedge c_L \in C_L \wedge c_R(t_R) \wedge t_{R,key} = key \wedge (\forall t_L \in L : c_L(t_L) \wedge t_{L,key} \neq key)$
		Terminating	❼ $\exists key, t_L$: $c_R \in C_R \wedge c_L \in C_L \wedge c_L(t_L) \wedge t_{L,key} = key \wedge (\forall t_R \in R : c_R(t_R) \wedge t_{R,key} \neq key)$
groupByKey	I : Input Table $t \in I$ and $t = (t_{key}, t_{value})$	Non-Terminating	❽ $\exists t$: $t \in I \wedge c \in C_I \wedge c(t) \wedge \lvert\{X \lvert x \in I \wedge x_{key} = t_{key}\}\rvert > 0$
reduce(udf) reduceByKey(udf)	I : Input Table, O: Output $udf : (t, t) \rightarrow t'$ where $t' \in O$	Non-Terminating	udf' is an iterative version of the original UDF udf, given as an input to reduce/reduceByKey. f' represents the set of path constraints generated from symbolic execution of udf'. ❾ $\exists t_1, t_2, t_3, \ldots, t_n \in I : c_1, c_2, \ldots, c_n \in C_I \wedge c_1(t_1) \wedge c_2(t_2) \wedge \ldots.. \wedge c_n(t_n) \wedge f'(I)$

Table 2: C_I represents a set of incoming constraints from the input table I, where each constraint $c \in C_I$ represents a non-terminating path. $c(t)$ represents that record $t \in I$ must satisfy constraint c. f defines the set of path constraints generated by symbolically executing udf and $f(t)$ represents the path constraint of a unique path exercised by input tuple t.

exercise a corresponding unique path. We define f as the set of symbolic path constraints of a UDF where $f(t)$ represents constraints of a unique path exercised by input t. By abstracting the implementation of dataflow operators into logical specifications, BigTest does not have to symbolically execute the Spark framework code (about 700KLOC), as it focuses on application level faults only as opposed to framework implementation faults which is out of the scope of this paper. BigTest supports all popular dataflow operators with the exception of deprecated operators such as `co-join`.

Filter. `Filter` takes a boolean function udf deciding if an input record should be passed to downstream operators or not. Therefore, we model two equivalence classes: (1) there exists a record t that satisfies udf and one of the incoming constraints C_I from input table I (*i.e.,* the table produced by its upstream satisfying operator), shown in ❶ Table 2; (2) there exists a record t that satisfies one of the incoming constraints but not udf, shown in ❷ Table 2.

Map and Flatmap. `Map` takes a UDF udf as an input and applies it to each input record to produce an output record. It has one equivalence class, where there exists tuple t from the input table I satisfying one of the incoming constraints, $c \in C_I$ and also one of the path constraints in f *i.e.,* path constraints generated by symbolically executing udf, shown in ❸ Table 2. `Map` is supported by the previous work Sedge but Sedge considers the UDF udf as a black box, uninterpreted function. `Flatmap` splits an input record using a udf to generate a set of records, and thus the equivalence class of `flatmap` is similar to that of `map`, as shown in ❹. BigTest handles `flatmap` by explicitly modeling a collection, described in Section 4.3.

Join. `Join` performs an *inner-join* of two tables t_r on the right and table t_l on the left based on the equality of keys, assuming that records from both tables are of the type `Tuple` (key, value). We model the output records of `join` into three equivalence classes: (1) the key of tuple t_R in the right table matches with a key of tuple t_L on the left; (2) the key of tuple t_R in the right table does not match with any key of tuple t_L on the left; and (3) the key of tuple t_L in the left table does not match with any key of tuple t_R on the right. ❺, ❻, and ❼ in Table 2 represent the three equivalence classes.

Reduce and ReduceByKey. `reduce` takes a udf and a collection as inputs and outputs an aggregated value, while `reduceByKey` performs a similar operation per key. As discussed in Section 4.1, BigTest generates an equivalent iterative version of the udf with a loop. By this refactoring of udf to udf', the equivalence classes could be modeled similar to that of `map`, where there exist input records $t_1, t_2, \ldots, t_n \in I$ on which each of the corresponding non-terminating constraint $(c_1, c_2, \ldots, c_n) \in C_I$ from the input table I holds true. In addition, each record must satisfy the constraints of udf', satisfying $f'([t_1, t_2, \ldots, t_n])$, as shown in ❾ of Table 2.

4.3 Path Constraint Generation

This section describes several enhancements in Symbolic Path Finder (SPF) to tailor symbolic execution for DISC applications. DISC applications extensively use string manipulation operations and rely on a `Tuple` data structure to enable key-value based operations. Using an off-the-shelf SPF naïvely on a UDF would not produce meaningful path conditions, thus, overlooking faults during testing.

```
1  def parse(s:String){
2   val cols = s.split(",")
3   (cols(0) , cols(1)) }
```

Figure 6: A UDF with string manipulation

Strings. Operations such as `split` for converting an input record into a key-value pair are common in DISC applications but are not supported by SPF. BigTest extends SPF by capturing calls to `split`, recording the delimiter, and returning an array of symbolic strings. When an n-th element of this symbolic array is requested, SPF returns a symbolic string encoded as `splitn` with a corresponding index. By representing the effect of Figure 6 as `(splitn(",",s,0), splitn(",",s,1))`, BigTest generates one terminating constraint, where s can only split into fewer than two segments, and one non-terminating constraint where s can split into at least two segments. Due to no `split` support, naïve SPF generates a string without any delimiter as a test input *e.g.,* "\x00" instead of "\x00,\x00". This input would lead to ArrayIndexOutOfBoundsException while accessing a string using `split(",")(1)`.

Collections. Constructing and processing collections through operators such as `flatmap` are essential in DISC applications. Therefore, BigTest explicitly models the effect of applying a UDF on a collection. In Figure 7, an iterative version of aggregator logic produced

```
1  def agg(arr:Array[Int]){
2   val sum = arr(0); // Bound K=3
3   for(a <- 1 to min(arr.size,2)) sum += arr(a)<0 ? 0 :
        arr(a);
4   sum }
```

Figure 7: An iterative version of aggregator UDF

by BigTest takes a collection as input and sums up each element, if the element is greater than or equal to zero. Given a user-provided bound `K=3` BigTest unrolls the loop three time and generates four pairs of a path condition (P) and the corresponding effect (E):

(1) `P:a(1)<0 ∧ a(2)<0, E:a(0)`
(2) `P:a(1)≥0 ∧ a(2)<0, E:a(0)+a(1)`
(3) `P:a(1)<0 ∧ a(2)≥0, E:a(0)+a(2)`
(4) `P:a(1)≥0 ∧ a(2)≥0, E:a(0)+a(1)+a(2)`

A naïve SPF does not handle collections well and thus may generate an array of length 1 only, not exercising line 3 in Figure 7. For example, `agg({3})` outputs the same sum of 3, when `arr(a)<0` is mutated to `arr(a)>0`), because the loop starts from 1 instead of 0, and `sum` is initialize to the first element of the array. Thus, it is not possible to defect the fault using an array of length 1.

Exceptions. BigTest extends SPF to explicitly model exceptions. For example, when an expression involves a division operator, division by zero is possible, which can lead to program termination. In Figure 1, BigTest creates two additional terminating path conditions, due to division by zero (i.e., `x<y∧x==0` and `y≤x∧y==0`).

Combining UDF symbolic execution with equivalence classes. BigTest combines the path conditions of each UDF with the incoming constraints from its upstream operator. For example, the UDF of `filter` (❸) in Section 3 produces a path condition of `s._2 == "Palms"`. Suppose that the upstream operator `map` produces one non-terminating path condition `s.split(",").length≥2` with the effect `s._2 = splitn(s,",",1)`. Inside the equivalence classes of `filter`—rows ❶ and ❷ in Table 2, BigTest plugs in the incoming path conditions (/effects) of an upstream operator `map` to C_I and the path conditions (/effects) of the `filter`'s UDF to f, producing the following path conditions.

- $c(t) \wedge f(t)$: `s.split(",").length≥2 ∧ splitn (s,",",1) == "Palms"`
- $c(t) \wedge \neg f(t)$: `s.split(",").length≥2 ∧ ¬ (splitn (s,",",1) == "Palms")`

Joint Dataflow and UDF Path. BigTest defines the final set of paths of a DISC application as *Joint Dataflow and UDF (JDU)* paths. We define a JDU path as follows: let $G = (D, E)$ represent a directed acyclic graph of a DISC application where D is a set of vertices representing dataflow operators and E represents directed edges connecting dataflow operators. Imagine a DISC application constructed with a `map` followed by `filter` and `reduce`. We represent this dataflow graph as $G = (D, E)$ such that $D = \{d_1, d_2, d_3, t_1\}$ and $E = \{(d_1, d_2), (d_2, d_3), (d_2, t_1)\}$ where d_1, d_2, and d_3 are `map`, `filter`, and `reduce` respectively. `filter` introduces a terminating edge (d_2, t_1) where a terminating vertex is t_1.

Since each dataflow operator takes a user-defined function f, for a vertex d_i, we define a subgraph $G_i = (V_i, E_i)$ which represents the control flow graph of f. In this subgraph, a vertex $v \in V_i$ represents a program point and an edge $(v_a, v_b) \in E_i$ represents the flow of control from v_a to v_b. G_i has $v_1 = start$ and $v_n = stop$ corresponding to the first and last statements. Then from each dataflow operator node d_i, we add a call edge from d_i to the start node of G_i and from the stop node of G_i to the d_{i+1}. Since some UDFs include a loop and thus have a cycle in the control flow graph, we finitize the loop using a user provided bound K and unroll the loop K times.

```
1  (assert (= line2 (str.++ (str.++ line20 ",") line21)))
2  (assert
3   (= line1
4    (str.++ (str.++ " " ",")
5     (str.++ (str.++ line11 ",")
6      (str.++ (str.++ " " ",") (str.++ (str.++ line13
           ",") line14))))))
7  (assert
8   (and (not (= (str.to.int line14) 0))
9    (and (isinteger line14)
10    (and (isinteger line13)
11     (and (= "Palms" line21)
12      (and (= x11 line20)
13       (and (<= s21 15)
14        (and (<= s21 40) (and (= s21 x621) (and (= s1
              x61) (= s22 x622))))))))))))
15 (assert
16  (and (= x11 line11)
17   (and (= x12 (/ (str.to.int line13) (str.to.int
         line14)))
18    (and (= x61 x11)
19     (and (= x621 x12) (and (= x622 x42) (and (= x71
           "walk") (= x72 1))))))))))))
```

Figure 8: Output SMT query constructed by BigTest to reflect JDU path constraint `C11` of Table 1 from motivating example.

We enumerate a set of all unique paths P_K for the graph G with expanded subgraphs and call each unique path a *Joint Dataflow and UDF (JDU) path*. For an arbitrary test suite T, the JDU path coverage is measured as a set of covered paths, $P_K(T) = \{p \mid p \in P_K, \exists t \in T \text{ and } t \models C_p\}$ where a test input t satisfies the path condition C_p of path p. Given a user-provided bound K for unrolling a loop, JDU path coverage is $\frac{|P_K(T)|}{|P_K|}$.

4.4 Test Data Generation

BigTest rewrites path constraints into an SMT query. For constraints on integer variables, BigTest uses analogous arithmetic and logical operators available in SMT. For string constraints, BigTest uses operations such as `str.++`, `str.to.int`, and `str.at`. BigTest introduces a new `splitn` symbolic operation. If a path constraint contains a clause `v = splitn(",", s, 1)`, BigTest generates `(assert (= s (str.++ " " (str.++ "," v))))` that is equivalent to `s = " ,v"` where v is a symbolic string. The path conditions produced by BigTest do not contain arrays and instead model individual elements of an array up to a given bound `K`.

BigTest generates interpreted functions for Java native methods not supported by Z3. For example, BigTest replaces `isInteger` with an analogous Z3 function. BigTest executes each SMT query separately and finds satisfying assignments (i.e., test inputs) to exercise a particular path. While executing each SMT query independently may lead to redundant solving of overlapping constraints, in our experiments, we do not find it as a performance bottleneck. Theoretically, the number of path constraints increases exponentially due to branches and loops; however, empirically, our approach scales well to DISC applications, because UDFs tend to be much smaller (in order of hundred lines) than DISC frameworks and we abstract the framework implementation using logical specifications.

Figure 8 shows an SMT query produced by BigTest for Figure 2. Lines 1 to 6 constrict the first table to have four segments and the second table to have two segments separated by a comma. Lines 7 to 10 restrict a string to be a valid integer. To enforce such constraint that crosses the boundary of strings and integers, BigTest uses a custom function `isinteger` and Z3 function `str.to.int`.

#	Subject Program	Output	# of Operators	Operators	Program Characteristics			JDU Paths (K=2)
					String Parsing	# Branches	# UDFs	
P1	IncomeAggregate	Total income of individuals earning ≤$300 weekly	3	map, filter, reduce	✓	2	4	6
P2	MovieRatings	Total number of movies with rating ≥ 4	4	map, filter, reduceByKey	✓	1	4	5
P3	AirportLayover	Total layer time of passengers per airport	3	map, filter, reduceByKey	✓	2	4	14
P4	CommuteType	Total number of people using each form of transport for daily commute	6	map, fitler, join, reduceByKey	✓	3	5	11
P5	PigMix-L2	PigMix performance benchmark	5	map, join	✓	2	6	4
P6	Grade Analysis	List of classes with more than 5 failing students	5	flatmap, filter, reduceByKey, map	✓	2	3	30
P7	WordCount	Finds the frequency of words	3	flatmap, map, reduceByKey	✓	1	3	4

Table 3: Subject Programs

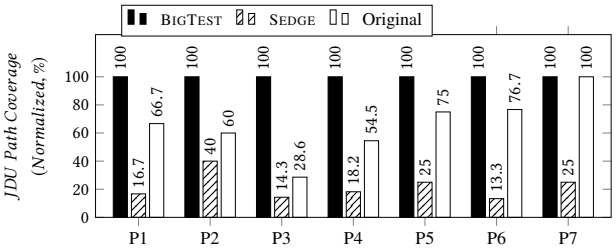


Figure 9: JDU path coverage of BigTest, Sedge, and the original input dataset

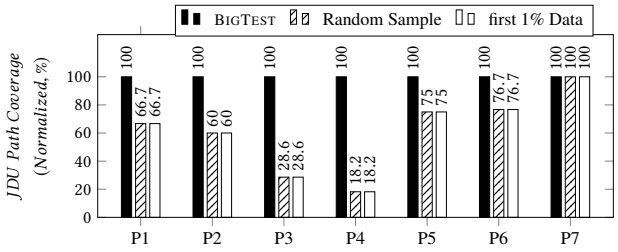


Figure 10: JDU path coverage of BigTest in comparison to alternative sampling methods

Lines 11 to 14 enforce a record to contain "Palms" and the speed to be less than or equal to 15. Lines 15 to 19 join these constraints generated from a UDF to the subsequent dataflow operator.

5 EVALUATION

We evaluate the effectiveness and efficiency of BigTest using a diverse set of benchmark DISC applications. We compare BigTest against Sedge in terms of path coverage, fault detection capability, and testing time. We compare test adequacy, input data size, and potential time saving against three alternative testing methods: (1) random sampling of k% records, and (2) using a subset of the first k% records, and (3) testing on the entire original data.

- To what extent BigTest is applicable to DISC applications?
- How much test coverage improvement can BigTest achieve?
- How many faults can BigTest detect?
- How much test data reduction does BigTest provide?
- How long does BigTest take to generate test data?

Subject Programs. In terms of benchmark programs, we use seven subject programs from earlier works on testing [31] and debugging DISC applications [25, 28], listed in Table 3. The PigMix benchmark package contains a data generator script that generates large scale datasets. We utilize `map` and `flatmap` with UDFs in Apache Spark to translate unsupported Pig operators like `load As` and `split`. Three programs `MovieRating` (P2), `AirportLayover` (P3), and `WordCount` (P7) are adapted from BigSift [25]. Each program is paired with a large scale dataset. The rest are self-created custom Apache Spark applications to add heterogeneity in dataflow operators and UDFs. Table 3 shows detailed descriptions of subject programs. All applications (1) involve complex string operations including `split`, `substring`, and `toInt`, (2) perform complex arithmetics, (3) use type `Tuple` for key-value pairs, and (4) generate and process a collection with custom logic using `flatmap`.

Experimental Environment. We run all large-scale data processing on a 16-node cluster. Each node is running at 3.40GHz and equipped with 4 cores, 32GB of RAM, and 1TB of disk capacity allowing us to run up to 120 tasks simultaneously. For storage, we use HDFS version 1.0.4 with a replication factor of 3. Due to a very small size of test data generated by BigTest, we leverage Apache Spark's local running mode to perform experiments on a single machine.

5.1 Dataflow Program Support

BigTest supports a variety of dataflow operators prevalent in DISC applications. For instance, Apache Spark provides `flatmap` and `reduceByKey` for constructing and processing collections. The previous approach Sedge is designed for PIG Latin with only a limited set of operators support [31]. Sedge is neither open-source nor have any implementation available for Apache Spark for direct comparison. Therefore, we faithfully implement Sedge precisely based on the technical details provided elsewhere [31]. We manually downgrade BigTest by removing symbolic execution for UDFs and equivalence classes for certain operators to emulate Sedge. The implementation of both Sedge and BigTest are publicly available[1]. Out of seven benchmark applications written in Apache Spark, five applications contain `flatmap` and `reduceByKey`, therefore, Sedge is not able to generate testing data for these 5 applications.

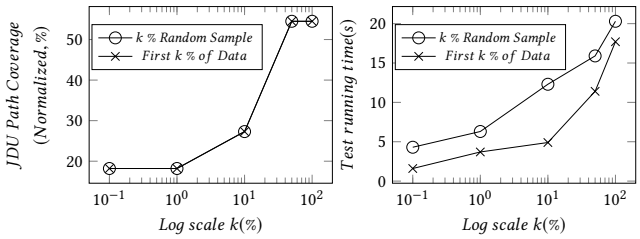
5.2 Joint Dataflow and UDF Path Coverage

We evaluate code coverage of BigTest, Sedge, and the original input dataset based on JDU path coverage defined in Section 4.3.

JDU Path Coverage Evaluation. We compare BigTest with three alternative sampling techniques: (1) random sampling of k% of the original dataset, (2) selection of the first k% of the original dataset, as developers often test DISC applications using `head -n`, and (3) a prior approach Sedge. To keep consistency in our experiment setting, we enumerate JDU paths for a given user-provided bound K and measure how many of these paths are covered by each approach.

Figure 9 compares the test coverage from BigTest, Sedge, and the original dataset. Y axis represents the *normalized* JDU path coverage ranging from 0% to 100%. Across seven subject programs, we observe that Sedge covers significantly fewer JDU paths (22% of what is covered by BigTest). By not modelling the internal paths of UDFs, Sedge fails to explore many JDU paths. Even when

[1] https://github.com/maligulzar/BigTest



(a) JDU Path Coverage **(b) Test Execution time**

Figure 11: The number of JDU paths covered and the test execution time when $k\%$ of the data is randomly selected and the $first\ k\%$ of data is selected for subject program `CommuteType`.

the complete dataset is used, the JDU path coverage reaches only 66% of what BIGTEST could achieve. The entire dataset achieves better coverage than SEDGE but it still lacks coverage compared to BIGTEST. In other words, using the entire *big* data for testing does not necessarily provide high test adequacy.

In Figure 10, both *random 1% sample* and *first 1% sample* provide 59% of what is covered by BIGTEST. We perform another experiment to measure the impact of different sample sizes on JDU path coverage and test execution time. Figure 11a and Figure 11b present the results on `CommuteType`. In `CommuteType`, the covered JDU paths increases from two to six when the percentage of the selected data increases from 0.1% to 50%. For those small samples, input tables do not have matching keys to exercise downstream operators and the time and distance columns may not have specific values to exercise all internal paths of the UDF. In terms of running time, as the sample size (k) increases, the test execution time also increases linearly (see Figure 11b in which x-axis is in log scale).

5.3 Fault Detection Capability

We evaluate BIGTEST's ability to detect faults by manually injecting commonly occurring faults. Because DISC applications are rarely open-sourced for data privacy reasons and there is no existing benchmark of faulty DISC applications, we create a set of faulty DISC applications by studying the characteristics of real world DISC application bugs and injecting faults based on this study.

We carefully investigate Stack Overflow and Apache Spark Mailing lists with keywords; *Apache Spark exceptions, task errors, failures*, and *wrong outputs* and inspect top 50 posts. Many errors are related to performance and configuration errors; thus, we filter out those and analyze 23 posts related to coding errors. For each post, we investigate the type of fault by reading the question, posted code, error logs, answers, and accepted solutions. We categorize our findings into seven common fault types:

(1) *incorrect string offset: e.g.,* a user uses 1 instead of 0 as the starting index in method `substring` and encounters StringIndexOutOfBoundsException [7].
(2) *incorrect column selection: e.g.,* a user accesses a wrong column in a csv file and thus receives ArrayIndexOutOfBoundsException [5].
(3) *use of wrong delimiters:e.g.,* while splitting a string a user uses "[ ]" instead of "\[\]", leading to a wrong output [8].
(4) *incorrect branch conditions: e.g.,* a user places a wrong order of control predicates, executing only one branch's side [4].

	SUBJECT PROGRAM						
	P1	P2	P3	P4	P5	P6	P7
Seeded Faults	3	6	6	6	4	4	2
Detected by BIGTEST	3	6	6	6	4	4	2
Detected by SEDGE	1	6	4	4	2	3	0

Table 4: Fault detection capabilities of BIGTEST and SEDGE

(5) *wrong join types: e.g.,* a user uses a wrong relational operator such as `cartesian join` instead of inner `join` [3].
(6) *swapping a key with a value: e.g.,* a user tries to join two tables while the keys and values are interleaved [6].
(7) *other common mutations* such as incorrect arithmetic or Boolean operator in UDFs.

When applicable, we inject one of each fault type in every application. For example, fault types 1 and 3 could only be inserted when `substr` or `split` method is used. When a fault type is applicable to multiple locations, we select a location which is inspired by and similar to the fault location in the corresponding StackOverflow/-Mailing List post. For instance, for fault type (2) above, we manually modify code to extract the first column instead of the second as a key in line 4 of Figure 2. Similarly, for fault type (3), we introduce fault by replacing the delimiter "," with ":". In total, our benchmark comprises of 31 faulty DISC applications. While SEDGE is not designed to handle string constraints, the main goal of this exercise is to justify the need to model UDFs and string constraints. SEDGE represents the internal UDFs as uninterpreted functions and, therefore, is unable to model all internal UDF paths. Conversely, BIGTEST treats UDFs as interpreted functions by representing them symbolically and models all internal UDF paths (up to bound k) which is crucial for high coverage testing of UDF's internal.

Table 4 shows a comparison of fault detection by BIGTEST and SEDGE. BIGTEST detects 2X more injected faults than SEDGE. For instance, in application P4, BIGTEST detects 6 faults, whereas SEDGE detects 4 faults. SEDGE uses concrete execution to model the UDF exercising line 16 of Figure 2 only. Therefore, it is unable to find an input for detecting fault at line 17 when the binary operator `">"` is replaced with `"<"` (*i.e.,* `s._2._1>15` to `s._2._1<15`). Similarly, when `join` in line 13 is changed to `rightOuterJoin`, SEDGE cannot detect any difference in the output because the equivalence classes do not model the terminating cases of join.

APPROACH	TEST INPUT DATA		OUTPUT FROM PROGRAM	
			ORIGINAL	FAULTY
BIGTEST	Terminating	CS100:41,0	CS200	CS100
	Non-terminating	CS200:0,0,0,0,0,0		CS200
Alternative	Non-terminating	CS200:0,0,0,0,0,0	CS200	CS200

Table 5: Modelling terminating and non-terminating cases

As another example, application P6 identifies courses with more than 5 failing students. A faulty version of P6 replaces the filter predicate `count>5` to `count>0` to output courses with at least one failing student. The original version of P6 uses `map` and `filter` to parse each row and identify failing students, `reduceByKey` to count the number of failing students, and uses `filter` to find courses with more than 5 failing students. BIGTEST generates at least two records to exercise both terminating and non-terminating cases of the last `filter`; thus, the original and faulty versions produce different outcomes on this data. On the other hand, a record is generated to exercise a non-terminating case only. Such data

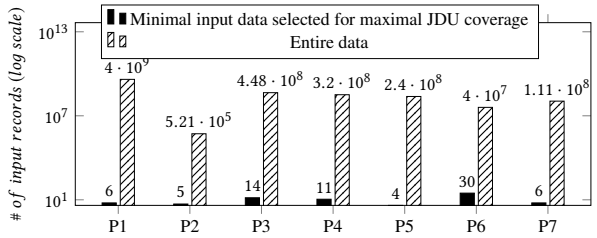


Figure 12: Reduction in the size of the testing data by BIGTEST

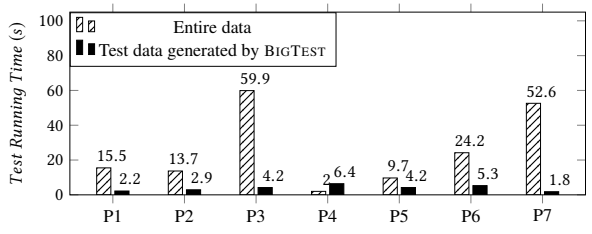


Figure 13: Test running time of entire data on large-scale cluster vs. testing on local machine with BIGTEST

would produce the same outcome for both the original and the faulty versions, unable to detect the injected fault, as shown in Table 5.

5.4 Testing Data Reduction

Testing DISC applications on the entire dataset is expensive and time-consuming. BIGTEST minimizes the size of the dataset, while maintaining the same test coverage. It generates only a few data records (in order of tens) to achieve the same JDU path coverage as the entire production data. Four out of seven benchmarks have an accompanied dataset, whereas the rest relies on a synthetic dataset of around 20GB each. Figure 12 shows the comparison result. In application P6, BIGTEST generates 30 rows of data to achieve 33% more JDU path coverage than the entire dataset of 40 million records. In other words, BIGTEST produces testing data 10^6 times smaller than the original dataset. Across all benchmark applications, BIGTEST generates data ranging from 5 to 30 rows. This is 10^5 to 10^8 times smaller than the original dataset, showing the potential to significantly reduce dataset size for local testing.

5.5 Time and Resource Saving

By minimizing test data without compromising JDU path coverage, BIGTEST consequently reduces the test running time. The benefit of a smaller test data is twofolds: (1) the amount of time required to run a test case decreases, and (2) the amount of resources (worker nodes, memory, disk space, etc.) for running tests also decreases.

We measure, on a single machine, the total running time by BIGTEST and compare it with the testing time on a 16-node cluster with the entire input dataset. We present a breakdown of the total running time into test data generation vs. executing an application on the generated data. Figure 13 represents the evaluation results. In application P6, it takes 5.3 seconds on a single machine to test with data from BIGTEST otherwise testing takes 387.2 CPU seconds (24.2 seconds x 16 machines) on the entire dataset, which still lacks complete JDU path coverage. Across the seven subject programs, BIGTEST improves the testing time by 194X, on average, compared to testing with the entire dataset.

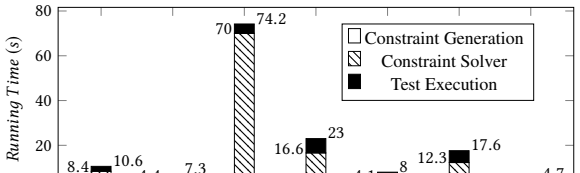


Figure 14: Breakdown of BIGTEST**'s running time**

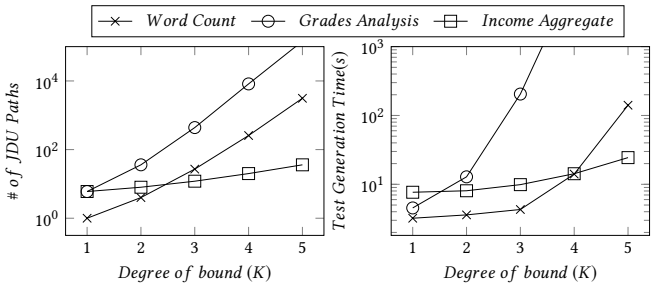


Figure 15: BIGTEST**'s performance when the degree of upper bound (K) on loop iteration and collection size changes**

Figure 14 reports the complete breakdown of the total running time of BIGTEST. The maximum test generation time observed is 70 seconds for `Airport Layover` (P3) in which 66 seconds are consumed by constraint solving. This is because the resulting JDU paths include integer arithmetics and complex string constraints together. Solving such constraints that cross the boundaries of different dimensions (integer arithmetics vs. string constraints) is time consuming even after BIGTEST's optimizations. If we combine both the test running time and test generation time and compare BIGTEST with the testing time with the entire dataset, BIGTEST still outperforms. In fact, BIGTEST still is 59X faster than testing on the entire dataset.

5.6 Bounded Depth Exploration for Aggregation

BIGTEST takes a user-provided bound K to bound the number of times a loop is unrolled. We assess the impact of varying K from 1 to 5 and present the results in Figure 15. At K=2, the number of JDU paths for `GradeAnalysis` is 36. When K is 3, BIGTEST generates 438 JDU paths. An exponential-like increase in the test generation time can be seen across the subject program, as we increase K. When K=2 in `GradeAnalysis`, BIGTEST takes 12 seconds and with K=3, BIGTEST takes 204 seconds. We empirically find K=2 to be a reasonable upper bound for loop iteration to avoid path explosion.

5.7 Threats to Validity

As we manually seed faults in the benchmark applications, the location of faults may introduce a bias in fault detection rate of BIGTEST posing a threat to internal validity. However, as mentioned before, most type of faults are only applicable to a single code location. If a fault type is applicable to multiple locations, we then select the fault location inspired by the corresponding StackOverflow/Mailing List post. In case of external validity, our classification of DISC faults may not be representative of all possible DISC application faults out there, as the survey is based on 50 StackOverflow/mailing lists posts. Additionally, the selection of fault types in our evaluation may be unfair to prior approaches. We attempt to mitigate this bias by restricting the evaluation to top seven most commonly occurring

faults in DISC applications. To eliminate this threat in the future, we plan to perform a large scale study on DISC application faults.

6 RELATED WORK

Testing Map-Reduce Programs. Csallner et al. propose the idea of testing commutative and associative properties of Map-Reduce programs by generating symbolic constraints [18]. Their goal is to identify non-determinism in a Map-Reduce program arising from a non-associative or non-commutative user-defined function in the `reduce` operator. They produce counter examples as evidence by running a constraint solver over symbolic path constraints. Xu et al. add few more Map-Reduce program properties such as (1) *operator selectivity*, (2) *operator statefulness*, and (3) *partition interference* [46] . Both of these techniques test only high-level properties of individual dataflow operators and they do not model the internal program paths of user-defined functions. Olsten et al. generate data for Pig Latin programs [34]. Compared to random sampling, their approach provides concise data selected from an input dataset to achieve better statement coverage. Their approach considers each operator in isolation and does not model internal program paths of UDFs—treated as black-box. Furthermore, Olsten et al. require knowing the inverse function of a UDF given to `transform`.

Li et al. (SEDGE) [31] is the most relevant approach to BIGTEST. SEDGE has three main limitations. First, its symbolic execution does not analyze the internal paths of individual UDFs. It considers UDFs as black box procedures and encodes them into *uninterpreted functions*. Second, it does not support operators such as `flatmap`, `reduce`, and `reducebyKey`, which are essential for constructing a collection and aggregating results from a collection in big data analytics. Third, the equivalence class modeling for each dataflow operator is not comprehensive, as it does not consider early terminating cases for some operators, where a data record does not flow to the next dataflow operator. Our empirical evaluation in Section 5 finds that these limitations lead to low defect detection in SEDGE. Table 6 compares dataflow operator support for related approaches and shows that BIGTEST has the most comprehensive and advanced support for modern DISC applications.

Test Generation in Databases. JDBC [42] or ODBC [2] enable software developers to write applications that construct and execute database queries at runtime. Testing such programs requires test inputs and database states from a user. Emmi et al. perform concolic execution of a program embedded with an SQL query [21] by symbolically executing the program till the point where a query is executed. Their approach is only applicable to basic SQL operations such as projection, selection, etc. (*e.g.,* `SELECT...FROM...WHERE`). Pan et al. perform database state generation but with different coverage criteria goals [36]. Braberman et al. select input data to test the logic of computing additional fields from existing columns in the database [13]. They do not handle arbitrary user-defined functions which are prevalent in DISC applications.

Symbolic Execution. Symbolic execution is a widely used technique in software engineering [12, 27, 38] and is used to generate test data using constraint solvers [14–16, 23, 32, 33, 41]. For example, Visser et al. use JPF (Java PathFinder [29]) to generate test input data [45]. However, the same approach cannot be applied to DISC applications directly because it would symbolically execute the application as well as the underlying DISC framework. Such practice will produce an unnecessarily large number of complex path constraints, facing scalability issues. This justifies and motivates our approach that abstracts each dataflow operator as a logical specification while performing symbolic execution for each UDF.

Dataflow Operators	Olston et al.	Li et al.	Emmi et al.	Pan et al.	BigTest
Load	✓	✓	✓	✓	✓
Map (Select)	✓	✓	✓	✓	✓
Map (Transform)	Incomplete	Incomplete	✗	✗	✓
Filter (Where)	✓	✓	✓	✓	✓
Group	✓	✓	✗	✗	✓
Join	Incomplete	Incomplete	✗	Incomplete	✓
Union	✓	✓	✗	✗	✓
Flatmap (Split)	✗	Incomplete	✗	✗	✓
Intersection	✗	✗	✗	✗	✓
Reduce	✗	✗	✗	✗	✓

Table 6: Support of dataflow operators in related work

Rosette is a framework for designing a solver-aided language [43] to ease the process of translating each language construct into symbolic constraints. BIGTEST and Rosette share some similarities in that they both translate higher-order types such as tuples or arrays into lower-level constraints. For the purpose of side-channel analysis, Bang et al. address a similar problem of solving constraints that cross boundaries between different theories (numerics, integer, and string constraints) [10]. Such cross-theory constraints are known to be difficult to solve with Z3 or CVC4. They extend SPF by modeling strings into bit vectors and by integrating numeric model counting in ABC [9] which could be used for BIGTEST in the future.

Regression Testing. Regression testing has been extensively studied in software testing. Safe regression testing selects only those test cases that exercise the updated regions of a program [26]. Rothermel et al. summarize several regression testing techniques and evaluate them under a controlled environment [39]. Test augmentation techniques help developers generate new test data to cover code not exercised by the available test cases using symbolic execution [17, 30]. Xu et al. evaluate concolic and genetic test generation approaches and report trade-offs [47]. The aforementioned approaches are not directly applicable to DISC applications, as they do not explicitly model the combined behavior of dataflow (/relational) operators and the internal semantics of UDFs.

7 CONCLUSION AND FUTURE WORK

Big data analytics are now prevalent in many domains. However, software engineering methods for DISC applications are relatively under-developed. To enable efficient and effective testing of big data analytics in real world settings, we present a novel white-box testing technique that systematically explores the *combined* behavior of dataflow operators and corresponding user-defined functions. This technique generates joint dataflow and UDF path constraints and leverages theorem solvers to generate concrete test inputs.

BIGTEST can detect 2X more faults than the previous approach and can consume 194X less CPU time, on average than using the entire dataset. With BIGTEST, *fast* local testing is feasible and testing DISC applications on the entire dataset may not be necessary.

Acknowledgments. We thank the anonymous reviewers for their comments. The participants of this research are in part supported by Google PhD Fellowship, NSF grants CCF-1764077, CCF-1527923, CCF-1460325, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, and Samsung grant. We would also like to thank Emina Torlak and Koushik Sen for their insightful discussions.

PUBLICATIONS

[1] [n. d.]. Hadoop. http://hadoop.apache.org/.

[2] [n. d.]. Microsoft Open Database Connectivity (ODBC). https://msdn.microsoft.com/en-us/library/ms710252(v=vs.85).aspx.

[3] 2015. . https://stackoverflow.com/questions/32190828.

[4] 2016. . https://stackoverflow.com/questions/40494999.

[5] 2017. . https://stackoverflow.com/questions/48021303.

[6] 2017. . https://stackoverflow.com/questions/42459749.

[7] 2018. . https://stackoverflow.com/questions/49505241.

[8] 2018. . https://stackoverflow.com/questions/52083828.

[9] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 255–272.

[10] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/2950290.2950362

[11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.

[12] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT&Mdash;a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*. ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

[13] Víctor Braberman, Diego Garbervetsky, Javier Godoy, Sebastian Uchitel, Guido de Caso, Ignacio Perez, and Santiago Perez. 2018. Testing and Validating End User Programmed Calculated Fields. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 827–832. https://doi.org/10.1145/3236024.3275531

[14] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, Washington, DC, USA, 443–446. https://doi.org/10.1109/ASE.2008.69

[15] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the 12th International Conference on Model Checking Software (SPIN'05)*. Springer-Verlag, Berlin, Heidelberg, 2–23. https://doi.org/10.1007/11537328_2

[16] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 322–335. https://doi.org/10.1145/1180405.1180445

[17] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 1066–1071. https://doi.org/10.1145/1985793.1985995

[18] Christoph Csallner, Leonidas Fegaras, and Chengkai Li. 2011. New Ideas Track: Testing Mapreduce-style Programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 504–507. https://doi.org/10.1145/2025113.2025204

[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[20] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[21] Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic Test Input Generation for Database Applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/1273463.1273484

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[24] Patrice Godefroid, Michael Y. Levin, and David A Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society. http://www.truststc.org/pubs/499.html

[25] 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 (SoCC '17)*. ACM, New York, NY, USA, 520–534. https://doi.org/10.1145/3127479.3131624

[26] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 312–326. https://doi.org/10.1145/504282.504305

[27] W. E. Howden. 1977. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Trans. Softw. Eng.* 3, 4 (July 1977), 266–278. https://doi.org/10.1109/TSE.1977.231144

[28] 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. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 216–227. https://doi.org/10.14778/2850583.2850595

[29] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Springer-Verlag, Berlin, Heidelberg, 553–568. http://dl.acm.org/citation.cfm?id=1765871.1765924

[30] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[31] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner. 2013. SEDGE: Symbolic example data generation for dataflow programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 235–245.

[32] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 416–426. https://doi.org/10.1109/ICSE.2007.41

[33] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, Berkeley, CA, USA, 67–82. http://dl.acm.org/citation.cfm?id=1855768.1855773

[34] 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 (SIGMOD '09)*. ACM, New York, NY, USA, 245–256. https://doi.org/10.1145/1559845.1559873

[35] K. Ouaknine, M. Carey, and S. Kirkpatrick. 2015. The PigMix Benchmark on Pig, MapReduce, and HPCC Systems. In *2015 IEEE International Congress on Big Data*. 643–648. https://doi.org/10.1109/BigDataCongress.2015.99

[36] Kai Pan, Xintao Wu, and Tao Xie. 2011. Database State Generation via Dynamic Symbolic Execution for Coverage Criteria. In *Proceedings of the Fourth International Workshop on Testing Database Systems (DBTest '11)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/1988842.1988846

[37] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1390630.1390635

[38] C. V. Ramamoorthy, S. B. F. Ho, and W. T. Chen. 1976. On the Automated Generation of Program Test Data. *IEEE Trans. Softw. Eng.* 2, 4 (July 1976), 293–300. https://doi.org/10.1109/TSE.1976.233835

[39] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *IEEE Trans. Softw. Eng.* 22, 8 (Aug. 1996), 529–551. https://doi.org/10.1109/32.536955

[40] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[41] Matt Staats and Corina Păsăreanu. 2010. Parallel Symbolic Execution for Structural Test Generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 183–194. https://doi.org/10.1145/1831708.1831732

[42] Art Taylor. 2002. *Jdbc: Database Programming with J2Ee with Cdrom*. Prentice Hall Professional Technical Reference.

[43] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

[44] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engg.* 10, 2 (April 2003), 203–232. https://doi.org/10.1023/A:1022920129859

[45] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, New York, NY, USA, 97–107. https://doi.org/10.1145/1007512.1007526

[46] Z. Xu, M. Hirzel, G. Rothermel, and K. L. Wu. 2013. Testing properties of dataflow program operators. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 103–113. https://doi.org/10.1109/ASE.2013.6693071

[47] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. 2010. Directed Test Suite Augmentation: Techniques and Tradeoffs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 257–266. https://doi.org/10.1145/1882291.1882330

[48] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[50] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. 2015. An Empirical Study on Quality Issues of Production Big Data Platform. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 17–26. http://dl.acm.org/citation.cfm?id=2819009.2819014