BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction

Qian Zhang
University of California, Los Angeles
zhangqian@cs.ucla.edu

Jiyuan Wang
University of California, Los Angeles
wangjiyuan@g.ucla.edu

Muhammad Ali Gulzar
Virginia Tech
gulzar@cs.vt.edu

Rohan Padhye
Carnegie Mellon University
rohanpadhye@cmu.edu

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

ABSTRACT

As big data analytics become increasingly popular, data-intensive scalable computing (DISC) systems help address the scalability issue of handling large data. However, automated testing for such data-centric applications is challenging, because data is often incomplete, continuously evolving, and hard to know a priori. Fuzz testing has been proven to be highly effective in other domains such as security; however, it is nontrivial to apply such traditional fuzzing to big data analytics directly for three reasons: (1) the long latency of DISC systems prohibits the applicability of fuzzing: naïve fuzzing would spend 98% of the time in setting up a test environment; (2) conventional branch coverage is unlikely to scale to DISC applications because most binary code comes from the framework implementation such as Apache Spark; and (3) random bit or byte level mutations can hardly generate meaningful data, which fails to reveal real-world application bugs.

We propose a novel coverage-guided fuzz testing tool for big data analytics, called BigFuzz. The key essence of our approach is that: (a) we focus on exercising application logic as opposed to increasing framework code coverage by abstracting the DISC framework using specifications. BigFuzz performs automated source to source transformations to construct an equivalent DISC application suitable for fast test generation, and (b) we design schema-aware data mutation operators based on our in-depth study of DISC application error types. BigFuzz speeds up the fuzzing time by 78 to 1477X compared to random fuzzing, improves application code coverage by 20% to 271%, and achieves 33% to 157% improvement in detecting application errors. When compared to the state of the art that uses symbolic execution to test big data analytics, BigFuzz is applicable to twice more programs and can find 81% more bugs.

KEYWORDS

fuzz testing, big data analytics, test generation

ASE '20, September 21–25, 2020, Australia

ACM ISBN 978-1-4503-6768-4/20/09.
https://doi.org/10.1145/3324884.3416641

ACM Reference Format:
Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416641

1 INTRODUCTION

Emerging technologies are producing much data and the importance of data-centric applications continues to grow. Data-intensive scalable computing (DISC) systems, such as Google's MapReduce [30], Apache Hadoop [1], and Apache Spark [2], have shown great promises to address the *scalability* challenge of big data analytics. Although DISC systems are becoming widely available to industry, DISC applications are difficult to test and debug. Data scientists often test DISC applications in their local environment using sample data only. These applications are thus not tested thoroughly and may not be robust to bugs and failures in the production setting.

The correctness of DISC applications depends on their ability to handle real-world data; however, data is inherently incomplete, continuously evolving, and hard to know a-prior. Motivated by the successes of systematic test generation tools [33, 34, 62], a few have been proposed for dataflow-based DISC applications [38, 45, 52]. For example, BigTest [38] uses symbolic execution to automatically enumerate different path conditions of a DISC application and generate concrete inputs using an SMT solver. However, its applicability is limited to the dataflow operators (e.g., `map`, `reduce`, `join`, etc.) where symbolic execution is supported, and limited by the path exploration capability of the underlying symbolic execution engine and an SMT solver. In other words, developing a *robust* test generation tool for DISC applications remains an open problem.

In recent years, coverage-guided mutation-based fuzz testing has emerged as one of the most effective test generation techniques for large software systems [17, 49]. Such fuzz testing techniques are based on implicit assumptions that it takes a relatively short amount of time to repetitively run programs with different inputs and arbitrary byte level mutations are likely to yield reasonable inputs. In fact, most fuzzing techniques start from a seed input, generate new inputs iteratively by mutating the previous inputs, and add new inputs to the input queue if they exercise a new branch.

* This research was done, while the third and fourth authors were graduate students at UCLA and UC Berkeley respectively.

However, our experience tells us that fuzzing cannot be applied to big data analytics directly. First, the long latency nature of DISC systems prohibits the efficacy of traditional fuzzing. While traditional fuzzing techniques assume thousands of invocations per second, for example, Apache Spark applications would need about 10 to 15 seconds to initialize the Spark `context` for each run—job scheduling, data partitioning, and serialization all contribute to increased latency. Second, low-level mutations (e.g., flipping a bit or byte) in existing naïve fuzzers can hardly explore corner cases that represent realistic application bugs. Lastly, grammar-aware fuzzers [35, 43, 70] exist to reduce the time required for constructing meaningful inputs. However, they require a user to provide grammar rules and, by definition, they do not produce inputs violating the user-provided grammar rules.

In this paper, we lay the groundwork for embodying a coverage-guided, mutation-based fuzz testing approach for big data analytics. The key insight behind BigFuzz is that *fuzz testing of DISC applications can be made tractable by abstracting framework code and by analyzing application logic in tandem.* Our key idea is to perform source-to-source transformation of a DISC application to a semantically equivalent, yet a framework-independent program that is more amenable to fuzzing.

Based on the insight that a DISC application developer writes application logic in terms of user-defined functions and connects them using dataflow operators in the DISC framework, BigFuzz focuses on exercising application logic as opposed to the DISC framework implementation. BigFuzz uses a two-level instrumentation method to monitor application-specific coverage, while modeling the different outcomes of dataflow operations. As such combination of behavior modeling is independent of the underlying DISC framework implementation, we can abstract the framework with executable specifications and generate a Spark context free program to mitigate the long latency caused by the DISC framework. An application developer is not required to write any custom specifications, because the specifications for dataflow operators such as `map` and `reduce` do not need to be re-written for each application. BigFuzz fully automates this process of constructing a semantically equivalent DISC application through source to source transformation.

As opposed to random bit or byte-level input mutations, we design schema-aware mutation operations guided by real-world error types. These mutation operations increase the chance of creating meaningful inputs that map to real-world errors. To inform the design of such data mutation operators, we conducted a systematic study on common error types and root causes in Apache Spark and Hadoop applications using two complementary sources: Stack Overflow [3] and Github [4]. The study identified ten common error types, which we map and encode in terms of six different mutation operators in BigFuzz.

We evaluate BigFuzz on a benchmark of twelve Apache Spark applications. We compare the time to generate test inputs and their associated error-finding capabilities against two baseline techniques: random fuzzing, and symbolic-execution based testing. With framework abstraction, BigFuzz is able to speed up the fuzzing time by 78 to 1477X compared to random fuzzing. Schema-aware mutation operations can improve application code coverage by 20 to 200% with valid inputs as seeds, which leads to 33 to 100% improvement in detecting application errors, when compared to naive random fuzzing. Even without valid input seeds, BigFuzz improves application code coverage by 118 to 271% and error detection by 58 to 157%, demonstrating its robustness. We show that BigFuzz is applicable to twice more applications and can find 81% more bugs than the state of the art, BigTest.

In summary, this work makes the following contributions:

(1) We propose a fuzz testing technique called BigFuzz that targets DISC applications by automatically abstracting the dataflow behavior of the DISC framework with executable specifications. This novel approach can also be generalized to other systems with long latency.
(2) We propose an automated instrumentation method to monitor application logic in conjunction with how dataflow operators are exercised in terms of their dataflow equivalence class coverage.
(3) We present schema-aware mutation operations that are guided by real-world errors encountered in DISC applications. To our knowledge, we are the first to design a fuzz testing technique by empirically studying and codifying mutations that correspond to real-world DISC bugs.
(4) Our experimental evaluation on 12 Apache Spark applications demonstrates that BigFuzz outperforms prior work in terms of code coverage and error-detection capability.

We provide access to artifacts of BigFuzz at https://github.com/qianzhanghk/BigFuzz.

2 BACKGROUND

Apache Spark. BigFuzz targets Apache Spark, a widely used data intensive scalable computing system but can generalize to other DISC frameworks. Spark achieves scalability by creating Resilient Distributed Datasets (RDDs), an abstraction of distributed collection [73]. Programmers can transform RDDs in parallel using dataflow operations, *e.g.*,`val newRDD = RDD.map(s => s.length)`. Dataflow operators such as `filter`, `map`, and `reduce` are implemented as higher-order functions that take a user-defined function (UDF) as an input argument. The actual evaluation of an RDD occurs when an action such as `count` or `collect` is called. For example, a Spark application developer writes application logic in terms of UDFs and connects them using dataflow APIs. To execute the program, Spark first translates a program into a Directed Acyclic Graph (DAG), where vertices represent various operations on the RDDs, and then executes each stage in a topological order.

The common industry practice for testing such big data analytics applications remains running them locally on a randomly sampled dataset. Testing with sample data is often incomplete which leads to rare buggy cases in production runs. Often Spark programs run for days and then crash without an obvious reason. Additionally, the start up latency associated with invoking the *Spark framework* and *Block Manager Master* can take several seconds for simply setting up an execution environment and repetitive data partitioning, job scheduling, serialization, and deserialization to support distributed execution all contribute to increased latency. Thus random fuzzing would be prohibitively expensive to test big data analytics.

Fuzz Testing. Fuzz testing such as AFL [17] has been proven to be highly effective in synthesizing test inputs that achieve high code coverage and find bugs. Given an input program, it instruments

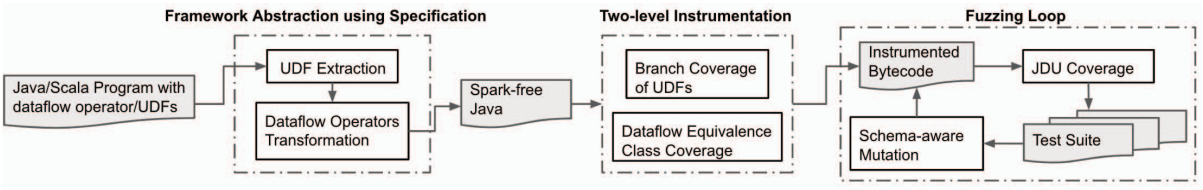

Figure 1: Approach Overview of BigFuzz

```
1  val loan = sc.textFile("account_history.csv")
2  // Input with zipcode, base loan, years, and rate
3   .map{  line => val cols = line.split(",")
4     (cols(0),cols(1).toFloat,
5       cols(2).toInt,cols(3).toFloat) }
6  //Return zipcode, base loan, years, and rate
7   .map{  s =>
8     val a = s._2
9     for(i <- 1 to s._3)
10     a = a * (1 + s._4)
11    (s._1, a) }
12  // Return zipcode and final loan
13 val locations = sc.textFile("zipcode.csv")
14  //input with zipcode and city
15  .map{  s =>
16    val cols = s.split(",")
17    (cols(0), cols(1)) }
18    //Return zipcode and city
19  .filter{ s => s._2 == "New York" }
20 val output = loan.join(locations)
21  .map{  s =>
22   if(s._2._1 > 10000) ("Property Loan",10000)
23   else if(s._2._1 > 1000) ("Car Loan",1)
24   else ("Credit Debt",1) }
25   //Return three categories based on the loan amount
26  .reduceByKey( _+_ )
```

(a) A DISC application `LoanType.scala`

```
1  ArrayList<String> results0 = LoanSpec.read(inputFile1);
2  ArrayList<Tuple4> results1 =  LoanSpec.map1 (results0);
3  ArrayList<Tuple2> results2 =  LoanSpec.map2 (results1);
4  ArrayList<String> results3 = LoanSpec.read(inputFile2);
5  ArrayList<Map3> results4 =  LoanSpec.map3 (results3);
6  ArrayList<Map3> results5 =  LoanSpec.filter1 (results4);
7  ArrayList<Join2> results6 = LoanSpec.join1(results5, results2);
8  ArrayList<Map1> results7 =  LoanSpec.map4 (results6)
9  ArrayList<Map1> results8 =  LoanSpec.reduceByKey1 (results7)
```

(b) A transformed program `LoanType.java` with executable specifications

```
1  public ArrayList<Map3> map3(ArrayList<String> input){
2      ArrayList<Map3> output = new ArrayList<>();
3      for (String item: input){
4          output.add( Map3.apply(item) );}
5      return output;}
```

(c) Specification implementation of `map3` in `LoanTypeSpec.java`

```
1  public class Map3 {
2   static final Map3 apply(String line2) {
3    String cols[]=line2.split(",");
4    return new Map3(cols[0],cols[1]); }
```

(d) The extracted UDF from lines 14 to 16 of Figure 2a is represented as `Map3.java`

Figure 2: Example code transformation and framework abstraction

the program's bytecode, iteratively generates new inputs by mutating several bits or bytes of the seed input, and collects coverage feedback by executing the instrumented program with new inputs. All inputs that exercise a new code branch are then be saved for further mutation. The implicit assumption underlying such iterative fuzzing is that the target program can run fast, (i.e., thousands of invocations per second); unfortunately, this assumption is false for many long latency applications such as big data analytics. For example, initializing the Spark context in local model to initiate a distributed data pipeline takes 19 seconds, which correspond to 98% of the total execution time with a typical testing input. The long latency prohibits the applicability of fuzzing for efficient test generation. Besides, naively monitoring branch coverage in DISC applications is unlikely to exercise application logic adequately, since most binary code comes from the DISC framework implementation (e.g., roughly 700 KLOC for Apache Spark). Under this circumstance, naive attempt to increase code coverage may eventually run out of memory. Furthermore, random byte-level mutations can hardly generate meaningful structured or semi-structured data to explore application logic effectively.

3 APPROACH

BigFuzz contains three components that work in concert to make coverage-guided fuzz testing tractable for big data analytics. Figure 1 shows (A) abstraction of dataflow implementation using source-to-source transformation with extracted user-defined functions, discussed in Section 3.1, (B) two-level instrumentation for coverage monitoring, discussed in Section 3.2), and (C) input mutations geared towards big data analytic errors based on our empirical study, discussed in Section 3.3. This approach is based on the insight that (1) we can reduce long latency of DISC applications by abstracting dataflow implementation in a DISC framework using executable specifications and (2) we can focus on exercising application logic rather than the entire framework by monitoring code coverage of user-defined functions in tandem with equivalence classes of abstracted dataflow behavior. Although BigFuzz is designed for Spark programs, its key idea can generalize to other DISC frameworks such as Hadoop by rewriting the dataflow operator APIs to our current set of corresponding specification implementation.

3.1 Framework Abstraction for Fuzzing

As discussed in Section 2, DISC applications have high latency, making them not suitable for traditional fuzz testing because they

Table 1: Dataflow Operator and Corresponding Equivalence Classes

Spark Dataflow Operator	Transformed Operator	Equivalences Classes
`def filter(udf:T→ Boolean): RDD[T]` Return an RDD that satisfies a predicate `udf:T→Boolean`	`ArrayList<T> filter (ArrayList<T> Input)` Return an `ArrayList` of elements passing `udf` where `udf:T → Booleean` is implemented in `filter`	**F1**: Non-Terminating: $\exists t.udf(t) = true$ **F2**: Terminating: $\exists t.udf(t) = false$
`def join[W](other: RDD[(K,W)]):Rdd[(K,(V,W))]` Return an RDD containing all pairs of elements with matching keys in `this` and `other` RDDs.	`ArrayList<T> join (ArrayList<T1> L, ArrayList<T2> R)` Return an `ArrayList` of elements from left `ArrayList` $t_L \in$ `L` and right `ArrayList` $t_R \in$ `R`, with matching keys $t_{L,key} = t_{R,key}$	**J1**: Non-Terminating: $\exists t_L, t_R.t_{L,key} = t_{R,key}$ **J2**: Terminating: $\exists t_L, \forall t_R.t_{L,key} != t_{R,key}$ **J3**: Terminating: $\exists t_R, \forall t_L.t_{R,key} != t_{L,key}$
`def map[U](udf:T→U)` Return a new RDD by applying `udf:T→ U` t of `this` RDD.	`ArrayList<T> map (ArrayList<U> Input)` Return a new `ArrayList` by applying a `udf:T→ U` to `this` `ArrayList` where `udf:T→ U` is implemented in `map`.	**M**: Non-Terminating: always non-terminated
`def reduceByKey(udf:(V,V) → V) : RDD [K,V]` Merge the values for each key using an associative reduce function.	`ArrayList<T> reduceByKey (ArrayList<T> Input)` Merge the values for each key using `udf:(V,V) → V` where `udf:(V,V) → V` is implemented in `reduceByKey`	**R**: Non-Terminating: always non-terminated

spend several seconds just to initialize Spark's execution context for each run. Theoretically, the long start-up latency can be somewhat reduced by sharing one Spark execution environment for multiple runs; however, such practice is still not enough to achieve millions of executions per minute, because each run still needs to pass through a data partitioner, a query optimizer, a job scheduler, and a data serializer/deserializer, etc.

In DISC frameworks, the implementation of dataflow and relational operators is influenced by and universally agreed upon the semantics of such operators [68]. For example, although a dataflow operator `join` may have a specialized physical implementation in each framework (e.g., hash join), it has the same consistent logical semantics across all DISC frameworks. BigFuzz takes advantage of this observation, rewrites a DISC application into an equivalent application that uses dataflow specifications, and monitors different equivalence class coverage of dataflow operations. For example, `filter` has two equivalence classes—one passing the filter predicate and the other not passing the filter. Because dataflow operators are *deterministic* and *state-less* [72], the transformed program is guaranteed to be equivalent to the original program. For example, `map{x => (x,1)}` will always give the same output for the same input for both the spec-based program and the original program.

We map each dataflow operator's implementation to a corresponding simplified yet semantically-equivalent implementation, which we call *executable specifications*. Such specifications help eliminate the dependency on the framework's code, transforming a DISC application into an equivalent, simplified Java program that can be invoked numerous times in a fuzzing loop.

BigFuzz automates this process of rewriting in two steps: (1) UDF extraction and (2) source to source transformation. Figure 2 illustrates this process using an example DISC application that identifies the frequency of each loan type within a metropolitan area. This program is a variation of one of the DISC Benchmark [38]. We formulate a distributed, RDD-based implementation using Spark's APIs (❶ in Figure 2a) to a simplified, executable specification of `map` in Figure 2c. Table 1 shows a few sample mappings between Spark RDD's dataflow implementation APIs, equivalent spec-implementations using `ArrayList`, and a set of corresponding equivalence classes for each dataflow operator.

Step 1. User-Defined Function (UDF) Extraction. To re-write a DISC application to use executable specifications only, BigFuzz decomposes the application into two components: (1) a direct acyclic graph (DAG) of dataflow operators and (2) a list of corresponding UDFs. Internally, BigFuzz decompiles the bytecode of the original application into Java source code and traverses Abstract Syntax Tree (AST) to search for a method invocation corresponding to each dataflow operator. The input arguments of such method invocations represent the UDFs, which are stored as separate Java classes as shown in Figure 2d.

Step 2. Source to Source Transformation. BigFuzz uses the DAG extracted in the previous step to reconstruct the DISC application in the same, interconnected dataflow order using executable specifications. Such dataflow spec implementation takes in an `ArrayList` object as input, applies the corresponding UDF on each element of the input list, and returns an output `ArrayList`. For example, class `LoanSpec.map3` (❶ in Figure 2b) represents the equivalent spec implementation using `ArrayList` that corresponds to `map` ❶ in Figure 2a. It takes in `results3` from its upstream operators and returns an `ArrayList result4` for downstream operator, `LoanSpec.filter1`. BigFuzz selects the corresponding UDFs from the list of UDFs extracted from step 1 and weaves them with the equivalent specifications shown in column 2 of Table 1. For example, Java class `Map3` has method `apply` mapping to the original UDF ❶ in Figure 2a, and this method is invoked on each element of the input list as seen in Figure 2c.

The above rewriting from a Spark application in Scala or Java to an equivalent Java application reduces the latency of running a DISC application, while retaining the same semantics. It also makes it easier to collect guidance metrics such as branch coverage by leveraging existing tools JQF [55], which takes Java bytecode as input and collects various guidance metrics for fuzz testing.

3.2 Application Specific Coverage Guidance

Prior work finds that branch coverage is an effective guidance mechanism for feedback-guided fuzz testing, pushing test generation towards hard-to-reach corners [17, 44, 56]. Generally, feedback-guided fuzzing techniques instrument a program's bytecode to label each constituent branch and if an input exercises a previously-unseen branch of the program, this input is appended in an input queue and the branch coverage is fed back into the fuzzer.

However, we observe that such branch coverage guidance mechanism cannot be applied to fuzz testing of big data analytics for two reasons. First, it cannot differentiate user-defined functions from framework code and can thus push test generation naively toward exploring the internals of DISC framework, as opposed to application logic. Second, it cannot effectively monitor different equivalence classes of dataflow operators though prior studies [38, 45, 52] argue that numerous errors originate from untested equivalence

Table 2: Data Collection for Error Type Study.

	Keyword	Total	Inspected
StackOverflow-Spark	apache spark exception	2430	top 150
	apache spark error	3780	top 200
	apache spark wrong/ unexpected/inconsistent result/output	143	143
StackOverflow-Hadoop	hadoop exceptions	2567	top 100
	hadoop error	9585	top 100
	hadoop wrong output	370	top 50
	hadoop wrong result	226	top 50
	hadoop unexcepted/ inconsistent result	39	39
Github	SparkContext	99	99

classes of dataflow operators. For example, when testing operator `join`, it is important to test three equivalence classes: (J1) there exists a key that appears in both tables, passing the joined result to the next operator, (J2) an input record in the left table does not have a matching key on the right table, terminating its data flow, and (J3) an input in the right table does not have a matching key on the left, terminating its data flow, discussed in Table 1.

To address these two problems, BigFuzz designs a two level instrumentation and monitoring method for application specific coverage guidance. The key insight here is that BigFuzz monitors regular branch coverage for user-defined functions only and for dataflow operators, it monitors at the level of equivalence classes. Below, we describe how we extend `TraceEvent` in JQF [55] to monitor which equivalence classes are exercised for individual dataflow operators.

TraceEvent in JQF. BigFuzz is built on top of JQF [55], a Java-based fuzz testing framework that uses ASM [5] to instrument Java bytecode on-the-fly as classes are loaded by the JVM. JQF instruments all application classes by injecting a static method call with a unique identifier after every bytecode instruction. It focuses on control flow instructions such as method calls (e.g. `INVOKESTATIC`, `INVOKEINTERFACE`, etc.) and branching instructions (e.g. `IF_CMPNE`, `GOTO`, etc.). JQF collects these instructions and groups them to a higher-level abstraction called `TraceEvent` (*e.g.,* `CallEvent` and `BranchEvent`), which are then emitted to its coverage logger.

DataFlowEvent in BigFuzz. To keep track of equivalence class coverage for individual dataflow operators, BigFuzz extends `TraceEvent` in JQF and creates a specific `DataFlowEvent`. In addition to an identifier, `DataFlowEvent` has an additional `Boolean` or `Integer` variable to keep track of which subset of equivalence classes is exercised by the corresponding dataflow operator. For example, `FilterEvent` is a specific `DataFlowEvent` class for keeping track of which equivalence classe is activated for `filter`. "`FilterEvent(arm = 1)`" represents the *non-terminating* equivalence class, where the filter predicate holds true and individual data records thus pass through the filter predicate. "`FilterEvent(arm = 0)`" indicates the other *terminating* case, where the filter predicate holds false and thus individual data records stop at this `filter`. BigFuzz instruments "`TraceLogger.get().emit(new FilterEvent(arm))`" in specification implementation of `filter` to emit `FilterEvent` with a specific arm to the trace logger. In this way, BigFuzz retains the DISC framework's behavior on the original application code, while abstracting its coverage guidance mechanism to the level of equivalence classes for individual dataflow operator uses.

Coverage Guidance for User-Defined Function. DISC application developer writes application logic in terms of user-defined functions (UDFs) and connects them using dataflow operators. These UDFs are standard library based Scala or Java implementations. To restrict normal coverage guidance to the body of UDFs (e.g., Figure 2d), BigFuzz uses a selective instrumentation scheme in ASM, while ignoring all other dependent libraries. This combination of monitoring dataflow equivalence coverage together with control flow events in the body of UDFs constitutes the joint dataflow and user-defined function path coverage (JDU path coverage), which essentially represents the behavior of application logic.

3.3 Mutations for Big Data Analytics

In feedback-guided fuzzing, commonly used input mutations are either bit-level or byte-level mutations in which random bits (or bytes) are flipped in an input represented as a series of bits [44, 55, 56]. The example program in Figure 2 takes as an input string that contains comma-separate row entries, where each entry contains the zipcode of borrower, the loan amount, the number of years since the loan was issued, and the interest rate respectively (*e.g.,* `90095,23000,7,0.045`). When traditional fuzzing is applied to this example program, if no seed is provided, it may first generate a series of random bits (*e.g.,* `0010 1010`, which maps to a character '∗'). Afterwards, this input is mutated by flipping several bits (*e.g.,* `0000 1010`, which is the character '\n'). Both cases above would generate meaningless inputs that fail at the program entry and are thus incapable of advancing the coverage goals. If the fuzzing process starts with a user-provided seed input, it will take this seed as bit series and flip several bits at a random position. In this way, traditional fuzzing can easily find data format errors when the program terminates at a earlier stage; however, it can hardly generate meaningful data that drives the program to a deep execution path since bit-flipping is more likely to destroy the data format or data type. In fact, our experiment finds that over 90% of inputs generated by random fuzzing fail at the entry point without exercising code further.

In contrast, BigFuzz designs a two-fold approach towards mutating inputs. First, it tries to generate valid inputs, such that the inputs are consistent with the input-parsing logic of the program. Second, it introduces *record-level schema-aware mutations*—modifying data with respect to the structured data types as well as value ranges. Unlike random bit-level mutations that produce unnatural inputs, each of the schema-aware mutations mimics a real-world error in DISC applications that may lead to program crashes or failures at runtime. To this extent, we extensively investigate DISC application errors posted on popular Q/A forums and code repositories.

A Study of Common Error Types. To collect real-world DISC application errors, we first performed a keyword-based search on StackOverflow Q/A forum and Github repositories using Spark

https://stackoverflow.com/search?q=apache+spark+exception
https://stackoverflow.com/search?q=apache+spark+error
https://stackoverflow.com/search?q=hadoop+exceptions
https://stackoverflow.com/search?q=hadoop+error
https://stackoverflow.com/search?q=hadoop+wrong+output
https://stackoverflow.com/search?q=hadoop+wrong+result
https://stackoverflow.com/search?q=hadoop+wrong+unexcepted+result
https://stackoverflow.com/search?q=hadoop+inconsistent+result

Table 3: Common Error Types in DISC Applications

Type	Portion	Example	Fix	Mutation
Type mismatch	16.28%	Data type is not double as expected by Spark's Kmeans[6]	Change type	M2
Illegal data for UDF	9.30%	NullPointerException caused by null values[7]	Check UDF	M4
Split-related errors	11.63%	The user uses .split("[ ]") when .split("\[\]") is expected[8]	Change delimeter	M3,M4
Incorrect column access	16.28%	Column access el(24).sum.toDouble where el.len=22 [9]	Check data length	M5
Incorrect offset access	2.35%	Used substring(1,11) instead of substring(0,10)[9]	Check offset	M6
Incorrect code logic	11.63%	The user uses a mutable data when it shouldn't be used[10]	Check code	M1
Incorrect API usage	11.63%	To match columns, equalTo API is expected in join operation[11]	Check API usage	M1
Join-related errors	4.70%	Join gives null values, leading to a NullPointerException in map[12]	Check data	M1
Semantic errors	9.30%	Minimum word appearance count in spark word2vec model is not met[13]	Change API	N/A
Framework bugs	6.90%	The expected result is a single row but the user got two lines[14]	Update library	N/A

libraries. Table 2 shows the number of posts and issues for each keyword search. As for StackOverflow, we studied both Spark- and Hadoop-related posts and manually examined 787 posts in total. We removed posts related to performance or configuration errors to focus on analyzing posts reflecting either application or data errors. As for Github, we inspected 99 projects that use the Apache Spark framework and their code repositories and bug reports.

We examined the accepted answers and comments (if any) to understand the root cause of underlying errors and summarize their solutions. We then distilled and grouped these root causes into ten categories, each reflecting a unique underlying programming issue. Table 3 shows the error type, % of posts for this error type, a representative example, and a solution to fix the error. For instance, the most frequently encountered error type is related to an incorrect column access which comprises of 16.28% of total errors. A representative example of such error is when a user accesses the 25th column (`el(24).sum.toDouble`) from data with only 22 columns (`el.len == 22`). The next most frequently occurring error (16.28%) is when the input does not conform to the expected data type *e.g.,* a record entry does not comply with `date.toDouble`, resulting in `NumberFormatException`.

Error-Type Guided Schema-Aware Mutations. Instead of bit-level mutations, BigFuzz uses a user-defined schema to perform coarse-grained, record-level mutations. In the schema, a user can indicate the number of columns, data type, and data distribution for each column of the input data. The following code snippet represents a sample user-provided schema for input `loan` in Figure 2a, which dictates that each input entry comprises of four comma-separated columns: the first column must be a 5-bit number string with prefix "900", the second column must be a random number with float type, the third column is an integer within range [0-30], and the last column is a float number within 0-1. From such schema, we can derive valid input constraints with respect to data format, data type, and data distribution.

```
number string[900xx],float[0-2^128],integer[0-30],float[0-1]
```

Based on our study, we design six mutation operations M1-M6 as shown in Table 3 to reflect their association with each real world error type. We enumerated these mutation types below:

- **Data Distribution Mutation (M1)** mutates a record to be either valid or invalid in terms of the allowed range based on the data distribution given in the schema (e.g., an integer value `10`, corresponding to the range `integer[0-30]` mutated to `25` or `-1`).
- **Data Type Mutation (M2)** modifies the data type of a selected column, while keeping the same value (e.g., `20` corresponding to `integer[0-30]` can be mutated to `20.0`, leading to `NumberFormatException` in line 5 Figure 2a).
- **Data Format Mutation (M3)** mutates a column-separating delimiter mentioned in the schema (e.g., replacing delimiter "," to "~").
- **Data Column Mutation (M4)** inserts one or several characters (e.g., replicating `ArrayIndexOutOfBoundsException` in StackOverflow post No.45962453 [15] when a random ' ' is inserted to data that is "Ctrl+A" separated).
- **Null Data Mutation (M5)** mutates the input row by removing one or more columns (e.g., replicating `NullPointerException` in Stack Overflow post No.36015704 [7] by accessing positions that do not exist).
- **Empty Data Mutation (M6)** mutates a random column to an empty string, leading to `StringIndexOutOfBoundException` caused by incorrect string operations.

Compared to random bit-level mutations, because these scheme-aware mutations are inspired by real-world errors, they are more effective for producing valid and invalid inputs, which we empirically demonstrate in Section 4.

Combined Data Generation and Mutation. Based on a user-provided schema, BigFuzz automatically constructs an application-specific mutation generator that combines valid input generation and six error-type guided mutations. Given a seed input, BigFuzz will either: (1) randomly mutate the seed input or (2) randomly generate valid inputs followed by mutating such inputs to increase cumulative coverage. IT can run under any of the two options and does not require having a valid seed. So a user may start BigFuzz with any string as a seed. Empirically, as we discuss in Section 4, starting with a valid seed does slightly improve performance by avoiding crashing too early from an invalid input. For example, if a valid seed such as `90001,100.0,10,0.01` is provided for Figure 2a, BigFuzz results in higher error detection than without. However, even an ill-formatted string is given as a seed, BigFuzz does retain high performance with its data generation option.

4 EVALUATION

Our evaluation seeks to answer the following research questions:

RQ1 Is a widely used fuzzing tool such as AFL applicable to big data analytics with long latency?

RQ2 Does framework abstraction effectively speedup fuzz testing?

Table 4: Statistics on Subject Programs

ID	Subject Program	Output	# of Operators	# of JDU Paths	BigTest Covered	BigFuzz Covered	RandomFuzzA Covered
P1	Wordcount	Find the frequency of words	3	2	2	2	1
P2	Commute Type	People count using each form of transport for daily commute	6	11	8	11	9
P3	ExternalCall	Find the frequency of words	3	4	2	4	1
P4	FindSalary	Total income of individuals earning ≤$300 weekly	3	8	6	7	4
P5	StudentGrade	List of classes with more than 5 failing students	5	14	6	14	6
P6	Movie Rating	Total number of movies with rating ≥ 4	4	11	5	11	5
P7	InsideCircle	Check whether the point (x,y) is in a circle	5	7	N/A	7	6
P8	MapString	String mapping	1	1	N/A	1	1
P9	NumberSeries	Find the numbers whose 3n+1 series' length is 25	3	9	N/A	9	3
P10	AgeAnalysis	Total number of people with different age ranges	3	9	N/A	9	4
P11	IncomeAggregation	Average income per age range in a district	6	9	N/A	9	4
P12	LoanType	The frequency of each loan type within a metropolitan are	4	6	N/A	6	5

RQ3 Does schema-aware mutation effectively improve code coverage and error detection capability?

RQ4 How much improvement in applicability and error detection does BigFuzz achieve, compared to an alternative symbolic execution-based technique?

Benchmarks. We use two sets of subject programs as benchmarks. They include twelve Spark programs written in Scala, listed in Table 4. For these subjects, P1-P2 and P4-P6 are directly from prior work [38], P3 is from [71] and P12 is reproduced by us based on the information of a Stack Overflow post [16]. P7-P8 are from Spark examples, and the remaining programs are handcrafted by the authors. We compare the generated test inputs and their associated error-finding capabilities with two baselines: (a) conventional fuzzing and (b) symbolic execution based testing for big data analytics [38], which is publicly available on Github.

Experimental Environment. We use Spark's local running mode to perform experiments on a single machine with Intel(R) Core(TM) i7-8750H 2.20GHz CPU and 16 GB of RAM running Ubuntu 16.04.

4.1 Faulty Benchmarks

To evaluate error detection capability, we inject code errors to the subject programs by mapping real-world error types in Table 3 to corresponding code modifications. Type mismatch errors and split-related errors are injected by changing the required data type and delimiter in the program. Changing an array or a string index can inject incorrect array or string access errors. To inject incorrect logic errors, we swap binary operators if a relational operator appears in a branch condition (*e.g.,* the user uses "<" when ">" is expected) or we replace a multiplication operator (*e.g.,* $a * b$) to a division (*e.g.,* a/b) to induce a divide by zero error. We also update constants or replaces variables. For example, if we inject a large number to a `redeceByKey` that does accumulation by key, an overflow error may occur when the intermediate value is beyond the capacity of its type. When we replace a dataflow operator such as `join` with its counterpart such as `left join`, we may reduce errors related to join operator or incorrect API usage.

This error seeding process is done automatically through source to source transformation on each subject. We traverse the abstract syntax tree (AST) of each program and apply one of the above injections to a random location if applicable. If code update can be applied to multiple locations, we choose one location randomly. This newly transformed AST is translated into source, so that we can see the error location. In total, we create 52 error seeded versions.

4.2 Applicablity of AFL (RQ1)

Almost all fuzz and random testing techniques are built on the assumption that *the program under test can be executed millions of times within a matter of hours.* To quantify this limitation of applying naive fuzzing to DISC applications, we use AFL on the twelve subject programs. AFL is a mature fuzzing tool designed for C/C++ [17] and JQF makes AFL available for Java programs. When using AFL with 9216M as memory and 100 seconds as timeout setting, it runs at an extremely low speed 0 to 9.68 `execs_per_sec` (an AFL reported metric to indicate the number of test expectations invoked from a fuzzing loop per second). The extremely low speed is because Spark applications spend significant time on setting up a Spark context, which attributes to most execution time. Further, as most binary code comes from DISC framework implementation with millions of lines of code, AFL's attempt to increase code coverage eventually leads to running out of memory after only 70 executions on average. Even before it dies, AFL with a random seed explores only 18% of the application code on average for all subjects except P1, P3, and P8, which take an unstructured random string as input. This empirically demonstrate that naive fuzzing is too slow and insufficient to generate meaningful structured data and reveal DISC application errors.

4.3 Comparison against Random Fuzzing

We create two separate downgraded versions of BigFuzz by disabling framework abstraction and error-type guide mutations respectively. We call the version without framework abstraction as RandomFuzzM as it retains **mutation** capability only. We call the version without error-type guided mutations as RandomFuzzA as it retains framework **abstraction** capability only.

RQ2: Speedup with Framework Abstraction. To assess speedup enabled by abstracting DISC frameworks in isolation, we use a downgraded version, RandomFuzzM, which disables source to source transformation for framework abstraction. We measure the running time of both BigFuzz and RandomFuzzM with 1000 iterations for programs P1-P12. The term 'iteration' refers to a single test execution invoked from a fuzzing loop. We repeat the experiment ten

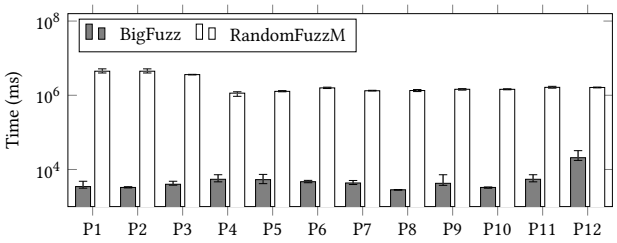

Figure 3: Running Time with 1000 iterations

times and report the average results in Figure 3—Y axis is milliseconds in log scale. BigFuzz is significantly faster than RandomFuzzM, speeding up the fuzzing time by 78X to 1477X.

RQ3: Coverage and Error Detection Improvement with Error-Type Guided Mutation. For RQ3, to evaluate the benefit of error-type guided mutations in isolation, we create a downgraded version RandomFuzzA that disables error-type guided mutations. We assess how fast BigFuzz and RandomFuzzA generate inputs exercising more JDU paths within the same iteration limit. We run BigFuzz and RandomFuzzA for 1000 iterations and report the cumulative % of exercised JDU paths and % of detected errors. We repeat the experiments four times and report averages.

Figures 4 and 5 report the results when starting fuzzing with and without a valid seed. Please note that BigFuzz does *not* require a user to provide a valid seed and can run under either option but we present both to estimate its capability accurately. Under the normal scenario when RandomFuzzA is started without a valid seed, its overall performance is lower than being bootstrapped with a valid seed, because mutating a valid seed can avoid early crashes. However, as we discuss below, BigFuzz can achieve similar performance with or without a valid seed, demonstrating robustness.

When started without a valid seed (the normal scenario), BigFuzz provides 118 to 271% improvement in JDU path coverage in comparison to starting RandomFuzzA without a valid seed, leading to 58 to 157% improvement in error detection. When fuzzing is started with a valid seed (the favorable scenario), BigFuzz can improve JDU path coverage by 20 to 200%, which leads to 33 to 100% improvement in detecting errors in comparison to RandomFuzzA with a valid seed.

The overall numbers of covered JDU paths among different runs are reported in the rightmost two columns in Table 4. Because BigFuzz is was to achieve 100% JDU path coverage for all benchmarks except P4, we did not run BigFuzz for 24 hours, as suggested in prior work [42]. The uncovered path starts with a string whose length must be larger than 7; however its integer value should be less than 300. Longer execution time may cover this path. For most programs, RandomFuzzA's randomly generated data can hardly exercise a deep execution path or dataflow equivalence classes.

4.4 Comparison with Symbolic Execution Based Testing

RQ4: Applicability. We assess how many Spark programs are testable using BigFuzz, in comparison to an alternative symbolic execution based approach, BigTest [38]. Symbolic execution-based testing requires a symbolical interpretation of each dataflow operator used in the program along with the UDF. The applicability of such techniques could be limited by the capacity of underlying SMT solvers and the ability to completely represent the entire program symbolically. We report the results in Table 4, where "N/A" represents BigTest is not applicable.

BigFuzz can be applied to twice as many programs as BigTest. For programs P1-P6, BigTest can generate test inputs successfully, while it fails to run on programs P7-P12. We investigate the publicly available source-code of BigTest and find three primary reasons behind its inapplicability on these programs. As with many symbolic execution based test generation techniques, BigTest restricts its symbolic exploration of unbounded collections and loops to a user-defined bound K (default is 2). Some programs such as P9 require a high value of loop-bound (K) to reach critically important JDU paths, which leads to inability of BigTest to maintain many symbolic states. During program decomposition, BigTest extracts each dataflow operator's argument, assuming that the argument is always a UDF. However, in P7, a pointer to a UDF is passed instead of the UDF itself, which results in incorrect UDF extraction. On the contrary, BigFuzz leverages static-dereferencing in such cases. Furthermore, in scala, a `for` loop iterating over a collection is compiled into an `map` method call on the collection. We find such cases in program P12 in which BigTest considers the `map` method call on collections as a dataflow operator resulting in incorrect DAG interpretation.

RQ4: Error Detection Capability. We evaluate the error detection capability of BigFuzz in covering more JDU paths and generating inputs that lead to errors that cannot be found by BigTest for programs P1-P6 that both tools are applicable.

Columns BigTest and BigFuzz in Table 4 summarize the JDU path coverage for BigTest and BigFuzz respectively. For all the test inputs generated by both tools, we manually inspect their covered execution path in UDFs and dataflow equivalence classes. In terms of tool setting, we set the user-specified bound `K` as a default value 2 for BigTest and set the fuzzing iterations as 1000 for BigFuzz. For all the subjects P1-P6 except P4, BigFuzz is able to achieve 100% coverage on JDU paths within the number of iterations, leading to 33% to 100% improvement in path coverage compared to BigTest.

Table 5: Error Detection Capability of BigFuzz and BigTest

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

Table 5 reports a comparison of error detection capability of BigFuzz and BigTest in terms of finding automatically injected errors. BigFuzz generates inputs to demonstrate all of the injected errors and detects 80.6% more injected errors than BigTest on average. In addition, BigFuzz has the unique capability of finding errors that cannot be detected by BigTest.

In P1, BigTest with default *K=2* setting cannot find an input for the runtime overflow when we inject a large number 2147483600 to `reduceByKey` because this error appears only when the minimum appearance number of a word is larger than three. In P3, when the `filter(v._2>1)` is replaced with `filter(log10(V._2)>1)`, BigTest fails to generate a constraint for this path that contains an external method call on a symbolic value. The injected divide

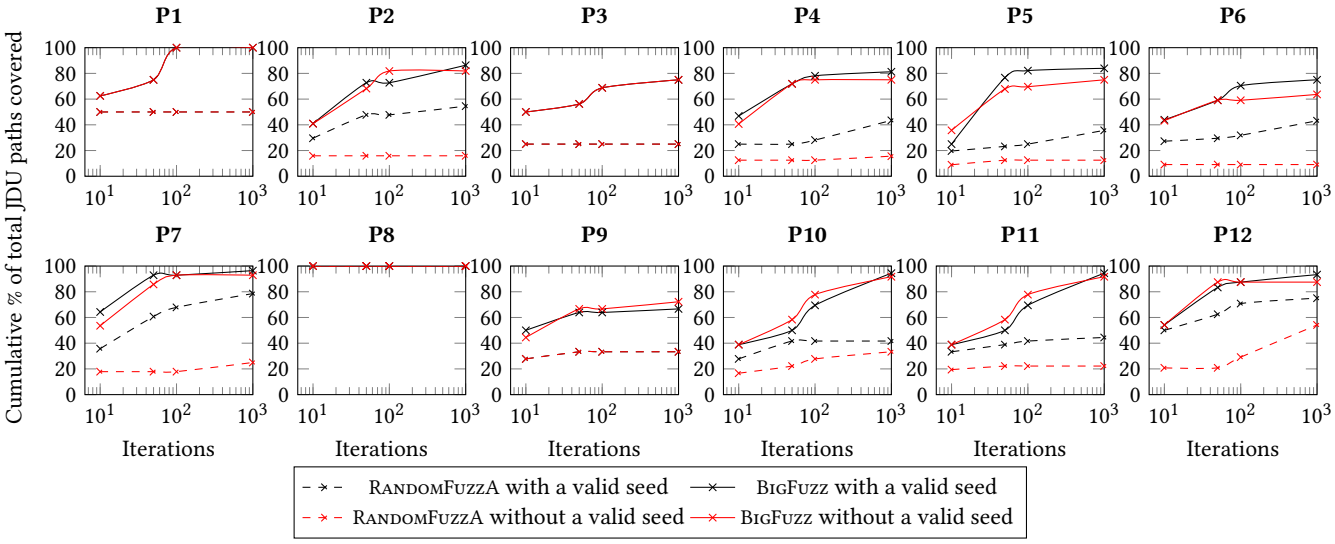

Figure 4: Joint Dataflow and UDF Coverage

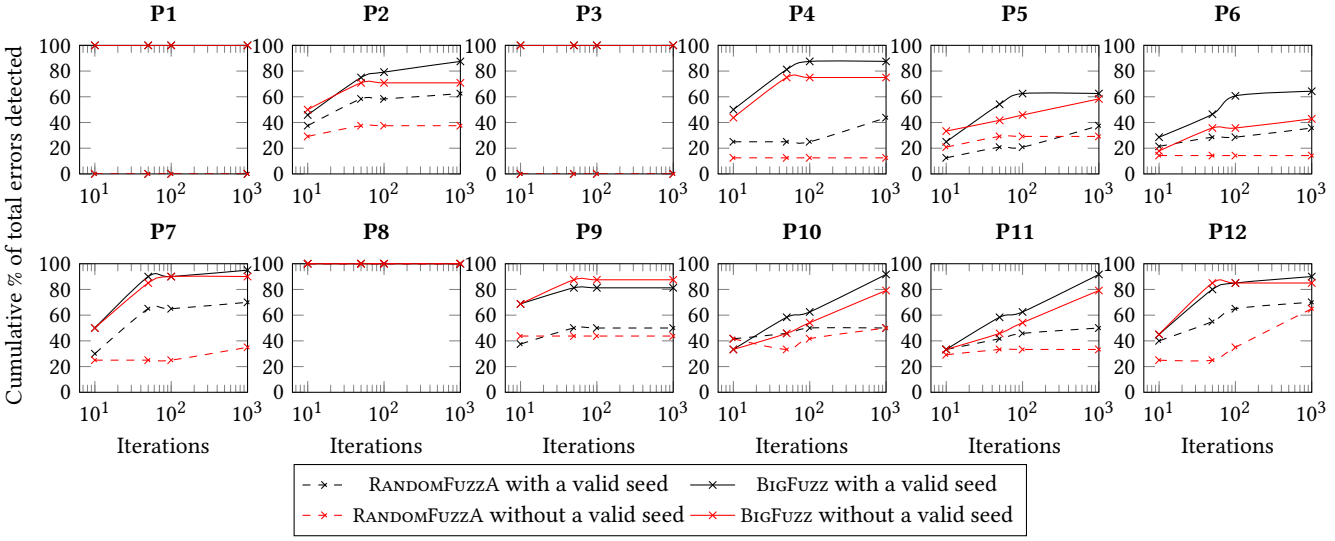

Figure 5: Error Detection Capability

by zero error in P2, as well as the injected type matching errors in P4-P6 are beyond BigTest because its underlying SMT solver fails to generate concrete inputs that satisfy such path constraints.

5 RELATED WORK

Fuzz testing has gained popularity in both academia and industry due to its black/grey box approach with a low barrier to entry [53]. The key idea of fuzz testing originates from random test generation where random inputs are incrementally produced with the hope to exercise previously undiscovered program behavior [28, 32, 54]

Random Testing. One difficulty in pure random testing is generating valid inputs, especially for object-oriented programs. JCrasher uses Java reflection to understand the parameter space and the type of a method under test and generates random inputs aiming to produce a Java exception [28]. Randoop [54] permutes method sequences to construct valid input, executes the new sequence, and observes regression or user-defined contract violations, while eliminating those leading to redundant execution by keeping track of the method sequences. EvoSuite also generates test suites to reveal program crashes and constructs test oracles in the form of assertions to check for the expected program behavior [32].

Fuzz Testing. Fuzz testing is similar to random test generation in many aspects. It mutates a seed input through its *fuzzer* to expose previously unseen internal states of the program. AFL is the most widely used coverage-guided fuzzing tool [17]. Generally, traditional coverage-guided fuzz testing has limited efficiency and effectiveness due to a vast space of inputs and unbounded program paths. Lemieux et al. identify rarely executed branches in the program with AFL-generated inputs and then create custom mutations so that the generated inputs gravitate toward exercising rare branches [44]. As a result, it requires fewer fuzzing loops and

achieves higher coverage in less time. Other approaches incorporate symbolic execution in fuzzing to guide careful selection and mutation of the inputs, invoking unique program paths [26, 64]. Padhye et al. incorporate the semantic validity of input mutations in Zest [56]. Zest reduces the search space of inputs by mapping bit-level changes to valid structural changes in the input.

Another angle to minimize unfruitful fuzzing loops is to generate only legal inputs for the program. Le et al. propose a grammar-based fuzzing approach called Saffron that relies on a user-defined grammar [43]. During fuzzing, if an input generated by the grammar leads to a program failure, Saffron reconstructs the grammar according to newly learned input specifications of the program. Wang et al. leverage a user-provided grammar, but instead of arbitrary mutations, they introduce grammar-specific mutations to diversify test inputs for tightly formatted input domains such as XML and JSON [70]. Gopinath et al. highlight that the state-of-art grammar-aware fuzzer *dharma* [19] is still two orders of magnitude slower than a random fuzzer and suggest guidelines for efficient grammar-aware fuzzing [36]. In their follow-up work, they present an approach to infer an input grammar from the interactions between an input parser and input data [35]. In DISC applications, a large proportion of program failures are due to ill-formatted inputs, which are hard to know in advance and are not taken into account during development. Therefore, grammar-aware fuzzing may not be practical in revealing errors in DISC applications, because if a user were to prescribe grammar rules up front, it may be too restrictive to generate meaningful error-inducing inputs.

Almost all fuzz and random testing techniques are built on the assumption that *the program under test can be executed millions of times within a matter of hours*. In the domain of data-intensive scalable computing, user applications are built on top of frameworks (such as Apache Spark or Hadoop MapReduce) containing complex distributed systems. Therefore, a single program run may take several minutes, if not hours, including a constant cluster spin-up time. Therefore, the performance and resource expense of state-of-art fuzzing and random testing for DISC applications are prohibitive.

To speed up test execution while fuzzing, UnTracer [51] dynamically strips out code-coverage instrumentation for lines of code that have already been covered. For DISC applications, the overhead is not due to instrumentation but indeed due to the extensive framework code. BigFuzz is the first fuzzing tool that transforms the target application by simplifying framework logic.

Software Debloating. Code debloating techniques [23, 60, 61, 63] strip off unnecessary logic or library functions that are not used by an application with the primary motivation to reduce the attack surfaces or to reduce binary size. Unlike debloating techniques that remove unused code via reachability program analysis, BigFuzz's framework abstraction *replaces* critical dataflow operators with semantically equivalent implementations to reduce the impact of bloated code for fuzz testing.

Symbolic and Concolic Execution. Symbolic execution has been extensively used for a diverse set of use cases, including automated test generation, program verification, security analysis, and code optimization. It allows programmers to execute their program symbolically to verify correctness [53]. Tools such as KLEE [25], Pex [66], and JavaPathFinder [69] brought symbolic execution to the forefront of systematic test generation by discovering uncovered program regions and using constraint solvers to generate additional test data to reveal faults in previously uncovered regions [41].

However, symbolic execution based testing is often limited by an enormous number of program paths emerging from large code. Several heuristics-based approaches address this problem of path explosion [22, 24, 47, 62]. Burnim et al. leverage static analysis to guide symbolic execution toward uncovered paths to prioritize specific program paths [24]. As with any other heuristics-based approach, these techniques produce many false negatives, as they prioritize exploration of certain program paths over others. Consequently, it may lead to low test quality (or fault detection rate) of the generated test suite. Experimental results from prior work show that symbolic execution may have lower fault detection capability than black-box fuzzing or random testing [29].

DISC applications depend on millions of lines of code in DISC framework, which makes it infeasible to naively apply symbolic execution to the resulting code as is. Even if heuristics-based symbolic execution is used to model the entire DISC application's binary code, the resulting test suite would mostly concentrate on finding the defects in the DISC framework, as opposed to finding bugs in application code.

Testing SQL and Data Analytics. Qex follows the traditional symbolic execution based test generation playbook and maps a SQL query to an SMT query [68]. It is loaded with custom theories for each relational operator. Cosette maps a relational query to a symbolic representation by either (a) proving equivalence among two queries or (b) generating counterexamples that explain conflicting answers from two queries [27]. Miao et al. leverage hard-coded specifications of relational operators and generate database rows to explain the output difference between two queries [50]. DOMINO [20] uses tailored, domain-specific operators based on random values to generate test data for relational database schemas.

Gupta et al. pivot their mutation-based test generation technique for SQL queries [39]. They define a set of mutations for selected relational operators such as inner-join or join and specify rules needed for each type of join to kill the mutant. SQL-integrated applications are widely used in practice and they invoke SQL queries programmatically using database connections such as ODBC [18] or JDBC [65]. Variants of symbolic execution are used to generate both application inputs and database states [31, 57–59].

Relational database applications rarely use user-defined functions (UDF), which are prevalent in DISC applications. Thus the above mentioned tools cannot reveal faults from the interaction of UDF and dataflow operators or the UDF alone, making them not applicable to DISC applications.

Testing DISC Applications. Today, DISC applications are, almost always, composed of both UDF and dataflow& relational operators. Gulzar et al. model the semantics of these operators in first-order logical specifications alongside with the symbolic representation of UDFs [38] and generate a test suite to reveal faults. Prior DISC testing approaches either do not model the UDF or model the specifications of dataflow operators partially [45, 52]. Li et al. propose a combinatorial testing approach that automatically extracts input domain information from schema and bounds the scope of possible input combinations [46].

However, all these symbolic execution use a heuristic (loop iteration bound K) during path exploration, which may lead to false negatives and they are also limited in applicability due to their symbolic execution scope.

Testing Large Scale Systems. Gulzar et al. study the use of differential testing for large-scale, end-to-end systems instead of traditional unit testing and find unit testing either incomplete or infeasible in practice due to the system's complexity [37]. They further observe that more than 40% of tests in real-world production software take between 15 minutes to several hours, stressing the infeasibility of fuzz testing on large-scale, long-latency systems. Other studies at Microsoft and Google concur exceedingly long running test times (in the order of hours) on their products, such as Microsoft Windows [40, 67]. These studies further validate our hypothesis that fuzz testing that assumes fast, repetitive re-runs is not suitable for such large systems with long latency.

Stateless Computation. Long startup time is a well-known problem for DISC applications and JVM applications in general. HotTub [48] reduces latency by amortizing the warm-up overhead over the lifetime of a cluster node instead of over a single job. RESTler [21] is a stateful fuzzer that analyzes the API specification of a cloud service and generates sequences of requests that automatically test the service through its API. Different from these stateful computations, dataflow operations are stateless and deterministic, which is the key insight that BigFuzz uses to create a semantically equivalent, fuzzing-friendly program.

6 CONCLUSION

Fuzz testing has emerged as one of the most effective test generation techniques. To adapt fuzzing to DISC applications with long latency, we propose BigFuzz that leverages (1) dataflow abstraction using source-to-source transformation, (2) tandem monitoring of equivalence-class based dataflow coverage with control flow coverage in user-defined functions, and (3) schema-aware mutations that reflect real world error types. BigFuzz achieves 78 to 1477X speed-up compared to random fuzzing, improves application code coverage by 20 to 271%, leading to 33 to 157% improvement in detecting application errors.

Acknowledgement

We thank Koushik Sen for his feedback and the anonymous reviewers for their comments. The participants of this research are in part supported by NSF grants CCF-1764077, CCF-1527923, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, Samsung grant, Google PhD Fellowship, and Alexander von Humboldt Foundation.

REFERENCES

[1] 2020. https://hadoop.apache.org/.
[2] 2020. https://spark.apache.org/.
[3] 2020. https://stackoverflow.com/.
[4] 2020. https://github.com/.
[5] 2020. https://asm.ow2.io/.
[6] 2020. https://stackoverflow.com/questions/37525136/.
[7] 2020. https://stackoverflow.com/questions/36015704/.
[8] 2020. https://stackoverflow.com/questions/52083828/.
[9] 2020. https://stackoverflow.com/questions/49505241/.
[10] 2020. https://stackoverflow.com/questions/41708814/.
[11] 2020. https://stackoverflow.com/questions/56478820/.
[12] 2020. https://stackoverflow.com/questions/41143862/.
[13] 2020. https://stackoverflow.com/questions/32028729/.
[14] 2020. https://stackoverflow.com/questions/36131942/.
[15] 2020. https://stackoverflow.com/questions/45962453/.
[16] 2020. https://stackoverflow.com/questions/59977879/is-there-any-convenient-way-to-do-the-debugging-for-spark-program.
[17] 2020. American Fuzz Loop. http://lcamtuf.coredump.cx/afl/.
[18] 2020. Microsoft Open Database Connectivity (ODBC). https://msdn.microsoft.com/en-us/library/ms710252(v=vs.85).aspx.
[19] 2020. Mozilla Security - dharma. https://github.com/mozillasecurity/dharma.
[20] A. Alsharif, G. M. Kapfhammer, and P. McMinn. 2018. DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 12–22. https://doi.org/10.1109/ICST.2018.00012
[21] V. Atlidakis, P. Godefroid, and M. Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 748–758.
[22] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-Directed Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 12–22. https://doi.org/10.1145/2001420.2001423
[23] Jaspreet Arora Guoqing Harry Xu Miryung Kim Bobby Bruce, TianyiZhang. [n. d.]. JShrink: In-Depth Investigation into Debloating Modern Java Applications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '20)*.
[24] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 443–446.
[25] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 209–224.
[26] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 725–741. https://doi.org/10.1109/SP.2015.50
[27] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.
[28] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience* 34, 11 (2004), 1025–1050.
[29] Marcelo d'Amorim, Carlos Pacheco, Darko Marinov, Tao Xie, and Michael D. Ernst. 2006. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE 2006: Proceedings of the 21st Annual International Conference on Automated Software Engineering*. Tokyo, Japan, 59–68.
[30] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, 1.
[31] 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)*. Association for Computing Machinery, New York, NY, USA, 151–162. https://doi.org/10.1145/1273463.1273484
[32] 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 (ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179
[33] 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
[34] 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

[35] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2019. Inferring Input Grammars from Dynamic Control Flow. arXiv:cs.SE/1912.05937

[36] Rahul Gopinath and Andreas Zeller. 2019. Building Fast Fuzzers. arXiv:cs.SE/1911.07707

[37] Muhammad Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 71–80.

[38] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. 2019. White-Box Testing of Big Data Analytics with Complex User-Defined Functions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 290–301. https://doi.org/10.1145/3338906.3338953

[39] B. P. Gupta, D. Vira, and S. Sudarshan. 2010. X-data: Generating test data for killing SQL mutants. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 876–879.

[40] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less without Sacrificing Quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 483–493.

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

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[43] Xuan-Bach D. Le, Corina S. Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2019. Saffron: Adaptive Grammar-based Fuzzing for Worst-Case Analysis. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2019), 14. https://doi.org/10.1145/3364452.3364455

[44] 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, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 475–485. https://doi.org/10.1145/3238147.3238176

[45] 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.

[46] Nan Li, Yu Lei, Haider Riaz Khan, Jingshu Liu, and Yun Guo. 2016. Applying Combinatorial Test Data Generation to Big Data Applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 637–647. https://doi.org/10.1145/2970276.2970325

[47] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. *SIGPLAN Not.* 48, 10 (Oct. 2013), 19–32. https://doi.org/10.1145/2544173.2509553

[48] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 383–400. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lion

[49] Valentin Manes, HyungSeok Han, Choongwoo Han, sang cha, Manuel Egele, Edward Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* PP (10 2019), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[50] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 503–520. https://doi.org/10.1145/3299869.3319866

[51] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

[52] 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

[53] Alessandro Orso and Gregg Rothermel. 2014. Software Testing: A Research Travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering (FOSE 2014)*. Association for Computing Machinery, New York, NY, USA, 117–132. https://doi.org/10.1145/2593882.2593885

[54] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. 75–84.

[55] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[56] 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 (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[57] 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)*. Association for Computing Machinery, New York, NY, USA, Article Article 4, 6 pages. https://doi.org/10.1145/1988842.1988846

[58] K. Pan, X. Wu, and T. Xie. 2011. Generating program inputs for database application testing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 73–82.

[59] Kai Pan, Xintao Wu, and Tao Xie. 2015. Program-input generation for testing database applications using existing database states. *Automated Software Engineering* 22, 4 (2015), 439–473. https://doi.org/10.1007/s10515-014-0158-y

[60] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. {RAZOR}: A Framework for Post-deployment Software Debloating. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1733–1750.

[61] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 869–886.

[62] 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

[63] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 329–339.

[64] 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. https://doi.org/10.14722/ndss.2016.23368

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

[66] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET. In *Tests and Proofs*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–153.

[67] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni. 2015. Automated Decomposition of Build Targets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 123–133.

[68] Margus Veanes, Jonathan de Halleux, Nikolai Tillmann, and Peli de Halleux. 2009. *Qex: Symbolic SQL Query Explorer*. Technical Report MSR-TR-2009-2015. https://www.microsoft.com/en-us/research/publication/qex-symbolic-sql-query-explorer/ Updated January 2010.

[69] Willem Visser, Corina S. Pundefinedsundefinedreanu, 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)*. Association for Computing Machinery, New York, NY, USA, 97–107. https://doi.org/10.1145/1007512.1007526

[70] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[71] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R Lyu. 2018. Benchmarking the capability of symbolic execution tools with logic bombs. *IEEE Transactions on Dependable and Secure Computing* (2018).

[72] Zhihong Xu, Martin Hirzel, Gregg Rothermel, and Kun-Lung Wu. 2013. Testing properties of dataflow program operators. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 103–113.

[73] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.