2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) | 978-1-6654-1219-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE-Companion52605.2021.00036

Efficient Fuzz Testing for Apache Spark Using Framework Abstraction

Qian Zhang
University of California, Los Angeles

Jiyuan Wang
University of California, Los Angeles

Muhammad Ali Gulzar
Virginia Tech

Rohan Padhye
Carnegie Mellon University

Miryung Kim
University of California, Los Angeles

***Abstract*—The emerging data-intensive applications are increasingly dependent on data-intensive scalable computing (DISC) systems, such as Apache Spark, to process large data. Despite their popularity, DISC applications are hard to test. In recent years, fuzz testing has been remarkably successful; however, it is nontrivial to apply such traditional fuzzing to big data analytics directly because: (1) the long latency of DISC systems prohibits the applicability of fuzzing, and (2) conventional branch coverage is unlikely to identify application logic from the DISC framework implementation. We devise a novel fuzz testing tool called BIGFUZZ that automatically generates concrete data for an input Apache Spark program. The key essence of our approach is that we abstract the dataflow behavior of the DISC framework with executable specifications and we design schema-aware mutations based on common error types in DISC applications. Our experiments show that compared to random fuzzing, BIGFUZZ is able to speed up the fuzzing time by 1477X, improves application code coverage by 271%, and achieves 157% improvement in detecting application errors. The demonstration video of BIGFUZZ is available at https://www.youtube.com/watch?v=YvYQISILQHsfeature=youtu.be.**

***Index Terms*—fuzz testing, dataflow programs, data intensive scalable computing, executable specifications**

I. INTRODUCTION

The importance of emerging data-intensive applications continues to grow at an increasing rate. Data-intensive scalable computing (DISC) systems, such as Google's MapReduce [1], Apache Hadoop [2], and Apache Spark [3], enable processing massive data sets by providing distributed, parallel versions of dataflow operator (e.g., `map`, `reduce`, `join`, etc.) implementations with application logic expressed in terms of user-defined functions (UDFs). Although DISC systems are becoming widely available to the industry, DISC applications are hard to test. The standard practice for testing such DISC applications today is to select a subset of inputs based on the developers' hunch with the hope that it will reveal possible defects. Not surprisingly, these applications are thus not tested thoroughly and may not be robust to bugs and failures in the production setting.

In recent years, fuzz testing has emerged as an effective technique for testing software systems [4], [5]. The effectiveness of such fuzzing techniques is based on two inherent yet oversight assumptions: (1) it takes a minuscule amount of time in the order of milliseconds to execute the target application, and (2) a set of arbitrary input mutations is likely to yield meaningful inputs. However, our extensive experience with DISC applications suggests that neither of the two assumptions holds for big data analytics.

We devise a new coverage-guided, mutation-based fuzz testing approach for big data analytics called BIGFUZZ. The key insight behind BIGFUZZ is that *fuzz testing of DISC applications can be made tractable by abstracting framework code and analyzing application logic in tandem*. BIGFUZZ first transforms a DISC application to a semantically equivalent, yet a framework-independent program that is more amenable to fuzzing to mitigate the long latency. It then uses a two-level instrumentation method to monitor control flow coverage of UDFs, while modeling the different outcomes of dataflow operations (i.e., dataflow equivalence classes). We call such a combination of behavior modeling as Joint Dataflow and UDF coverage (JDU coverage). During fuzzing, it uses schema-aware mutation operations guided by real-world error types to increase the chance of creating meaningful inputs that map to real-world errors.

The main contribution of BIGFUZZ is that we made traditional fuzzing feasible for big data analytics by abstracting the dataflow behavior of the DISC framework with executable specifications. Additionally, BIGFUZZ, to our knowledge, is the first fuzzing tool that uses mutations are specifically designed to reveal real-world DISC errors. In our experiments, BIGFUZZ with framework abstraction can 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.

The full technical paper on this approach appeared at ASE 2020 [6] and this paper describes BIGFUZZ's user interfaces and internal implementation with a focus on tool demonstration. BIGFUZZ is a Java-based command line tool for testing Scala/Java Spark applications and can be easily generalized to other DISC frameworks such as Hadoop MapReduce [1]. We provide access to artifacts of BIGFUZZ at https://github.com/qianzhanghk/BigFuzz.

978-1-6654-1219-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE-Companion52605.2021.00036

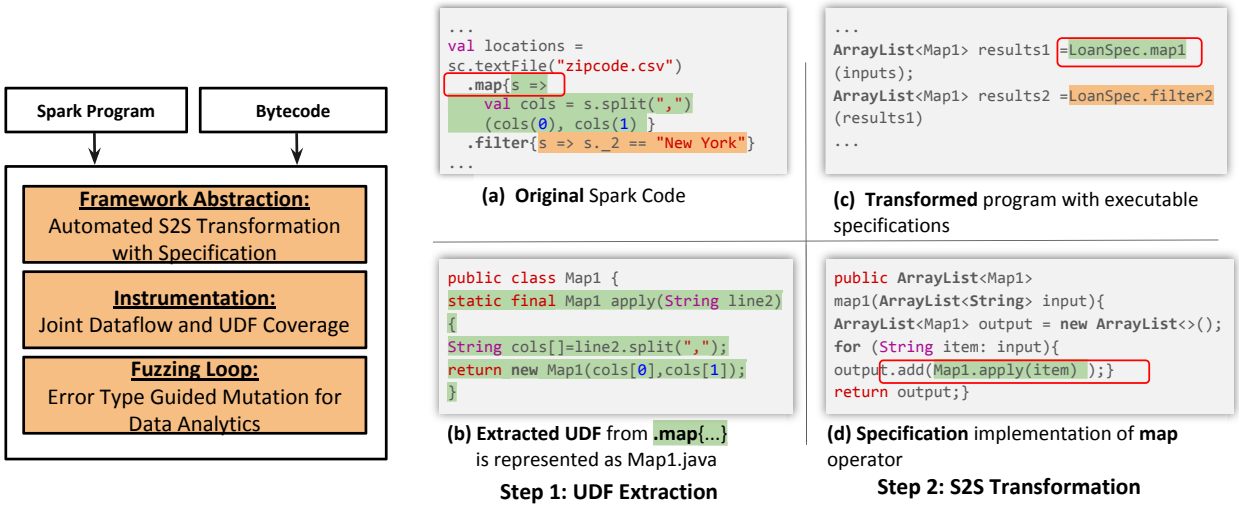

Fig. 1: An Overview of BIGFUZZ's Approach with Framework abstraction

II. TECHNICAL APPROACH

We describe BIGFUZZ's main technical contributions below. The detailed explanation is described in our full paper [6]. BIGFUZZ takes in an Apache Spark program written in Scala or Java and an input schema, and it automatically generates concrete data for effective and efficient testing. Figure 1 illustrates the three novel parts in BIGFUZZ.

A. Framework Abstraction

BIGFUZZ maps 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 in two steps: *UDF Extraction* and *Source-to-Source Transformation*.

In the first step, BIGFUZZ decomposes the input Spark program into a direct acyclic graph (DAG) of dataflow operators and a list of corresponding UDFs. It decompiles the bytecode of the original Spark program into Java source code and traverses the Abstract Syntax Tree (AST) to extract each UDF into a separate Java class, as shown in Figure 1b. It then uses the extracted DAG and UDFs to reconstruct the DISC application in the same interconnected dataflow order using executable specifications. For example in Figure 1c, `map` operator is followed by `filter` operator, emulating their connection in Figure 1a. The dataflow spec implementation, such as the spec of `map` operator in Figure 1d, takes in an ArrayList object as input, applies the corresponding UDF on each element of the input list, and returns an output ArrayList.

B. Coverage Guidance

To differentiate UDFs from framework code, BIGFUZZ designs a two level instrumentation and monitoring method for application specific coverage guidance. For dataflow operators, it monitors at the level of equivalence classes by extending the `TraceEvent` in JQF [7] to 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(arm=1)"` for `filter` operator 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. For UDFs coverage, BIGFUZZ uses a selective instrumentation scheme in ASM [8], while ignoring all other dependent libraries. This combination of monitoring dataflow equivalence coverage together with control flow events in UDFs constitutes the JDU coverage, which essentially represents the behavior of application logic.

C. Mutation

Instead of bit-level mutations, BIGFUZZ uses a user-defined schema to perform *record-level schema-aware mutations*—modifying data with respect to the structured data types as well as value ranges. In the schema, a user can indicate the number of columns, data type, and data distribution for each column of the input data. Unlike random bit-level mutations that produce unnatural inputs, each of the schema-aware mutations mimics a real-world error type 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.

III. IMPLEMENTATION

BIGFUZZ is a Java-based command line tool that provides efficient fuzz testing of DISC applications. BIGFUZZ is built on top of JQF [7], a Java-based fuzz testing framework that instruments Java bytecode on-the-fly as classes are loaded by the JVM. BIGFUZZ requires a test driver to indicate the test

```
1 public ArrayList<Tuple3> filter1(ArrayList<Tuple3>
2   input){
3     ArrayList<Tuple3> ans = new ArrayList<>();
4     for(Tuple3 item: input){
5     if(Filter1.apply(item)) ans.add(item);}
6     int iid = LoanSpec.class.hashCode();
7     int arm = 0;
8     if(ans.isEmpty()==false) arm=1;
9      TraceLogger.get().emit(new FilterEvent
10     (iid,arm));
11    return ans;}
```

Fig. 2: Instrumented `filter` to emit dataflow equivalence class coverage

class and test method. A test driver is a JUnit-style test class with `@RunWith(JQF.class)` annotation on the test class and `@Fuzz` annotation on the test method. A user can use the following command line to invoke BIGFUZZ:

```
BigFuzz/bin/jqf-bigfuzz -c
.:$(BigFuzz/scripts/classpath.sh)
testclass testmethod <maxTrials>
```

Given an input Spark program, BIGFUZZ first reads its bytecode and translates it to Java source code with Java Decompiler (JAD) [9]. It parses the AST of the de-compiled Java source code 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. Next, based on the transformed program, BIGFUZZ inserts code, for example, lines 6-10 in Figure 2, to each specification-based implementation of dataflow operators to monitor which equivalence class is activated, and it instruments the bytecode of the extracted UDF classes only to collect exercised branches in current execution. All `DataflowEvents` and `TraceEvents` are emitted to a coverage logger. Then during fuzzing, BIGFUZZ will either randomly mutate the seed input or randomly generate valid inputs followed by mutating such inputs to increase cumulative coverage.

IV. DEMONSTRATION

In this section, we present a step-by-step demonstration of BIGFUZZ. Suppose Alice would like to investigate the average income per age range in her district. She uses the entire income survey database which contains the income information of states and counties for over several years. A sample row in this dataset is a string that contains the zipcode of employee, the age, and the annual income amount of this employee respectively (*e.g.,* `90095,33,58000`).

Alice writes a Spark application to perform this analysis, as shown in Figure 3. She first uses a `map` operator to extract the zipcode, the age, and the income amount from each row using a UDF in line 4, and uses a `filter` operator to filter the data rows based on if its zipcode is "90024" in line 5. Next, Alice uses another `map` operator to cluster the data into different age groups. In lines 18-19, she aggregates all the income and number of persons in each age group with a `reduceByKey`

```
1  val data = text.map {
2     s =>
3     val cols = s.split(",")
4     (cols(0), Integer.parseInt(cols(1)),
          Integer.parseInt(cols(2)))
5     }.filter( s => s._1.equals("90024"))
6  val pair = data.map {
7     s =>
8     if (s._2 >= 40 & s._2 <= 65) {
9        ("40-65", s._3)
10    } else if (s._2 > 20 & s._2 < 40) {
11       ("20-39", s._3)
12    } else if (s._2 < 20){
13       ("0-19", s._3)
14    } else {
15       (">65", s._3)
16    }
17 }
18 val sum = pair.mapValues( x =>(x,1))
19 .reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
20 .mapValues(x=>(x._2,x._1.toDouble/x._2.toDouble))
21 .foreach(println)
```

Fig. 3: Alice's program that finds the average income per age group in her district.

operator. In the end, Alice calculates the average income with a `mapValues` operator in line 20.

To run BIGFUZZ on her program, she writes an input schema shown in the following code snippet to describe her data.

```
number string[900xx],integer[0-120],integer[0-2^32]
```

This indicates that each input entry comprises of three comma-separated columns: the first column must be a 5-bit number string with prefix "900", the second column must be an integer within the range [0-120], and the last column is an integer within [0-2^{32}]. BIGFUZZ takes such input schema through `conf` file and passes it to `MutationGeneration` to automatically generate mutations that are tailored for this schema.

As a first step, BIGFUZZ decomposes her program into six Java classes representing the UDFs, each of which can be identified with the operator name followed by its execution order (*e.g.,*, `map1.java`). Next, BIGFUZZ reconstructs her program with these Java classes using the executable specifications and automatically generates a test driver for her program. Alice invokes BIGFUZZ by typing the following command:

```
BigFuzz/bin/jqf-bigfuzz -c
.:$(BigFuzz/scripts/classpath.sh)
IncomAggregationDriver IncomeAggregation
```

Alice can monitor the fuzzing process by observing the console log and interrupts the execution by pressing "Ctrl-C". BIGFUZZ does not require valid inputs as seeds. For each iteration, BIGFUZZ produces a new input with several data rows by either randomly mutating the seed inputs or randomly generating valid inputs. Such new input will be saved to a separate `csv` file if it detects a unique crash or a new JDU branch.

Alice uses the generated test data `90024,20,10900` as input for unit testing. This input represents the income for a 20-year old person; however, the test output classifies it to

(>65). Alice investigates her code and identifies an error where a $\geq$ is misused by $>$ in line 10 of Figure 3). BIGFUZZ also generates inputs such as non-numeric strings or non-integer numbers to reveal critical runtime crashes in line 3 of Figure 2. Alice inserts relevant exception handling and data filter to eliminate such corner cases.

V. RELATED WORK

Testing DISC Applications. Gulzar et al. model the semantics of these operators in first-order logical specifications alongside the symbolic representation of UDFs [10] 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 [11], [12]. Li et al. propose a combinatorial testing approach that automatically extracts input domain information from schema and bounds the scope of possible input combinations [13]. However, all these symbolic executions 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.

Fuzz Testing. Fuzz testing mutates the seed inputs through a *fuzzer* to maximize a specific guidance metric, such as branch coverage, and find crashes in programs and frameworks. Fuzz testing has been shown to be highly effective in revealing a diverse set of bugs, including correctness bugs [4], [14], security vulnerabilities [15], [16], and performance bugs [17]. For example, AFL [4] mutates a seed input to discover previously unseen coverage profiles. MemLock [17] employs both coverage and memory consumption metrics to find abnormal memory behavior.

Instead of flipping several bits/bytes in each mutation, [18] has investigated specific mutations for web browsers. BIGFUZZ, along with our full technical paper [6], designs mutations based on an empirical study of Apache Spark application errors reported in StackOverflow and Github. To speed up test execution while fuzzing, UnTracer [19] 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.

VI. CONCLUSION

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 up to 1477X speed-up compared to random fuzzing, improves application code coverage by up to 271%, leading to up to 157% improvement in detecting application errors.

VII. ACKNOWLEDGMENT

The participants of this research are in part supported by NSF grants CHS-1956322 CCF-1764077, CCF-1723773, ONR grant N00014-18-1-2037, and Intel CAPA grant.

REFERENCES

[1] J. Dean and S. Ghemawat, "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, 2004.
[2] https://hadoop.apache.org/, 2020.
[3] https://spark.apache.org/, 2020.
[4] "American fuzz loop." http://lcamtuf.coredump.cx/afl/, 2020.
[5] V. Manes, H. Han, C. Han, s. cha, M. Egele, E. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 10 2019.
[6] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction," in *Proceedings of The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.
[7] R. Padhye, C. Lemieux, and K. Sen, "Jqf: Coverage-guided property-based testing in java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 398–401, Association for Computing Machinery, 2019.
[8] https://asm.ow2.io/, 2020.
[9] http://java-decompiler.github.io/, 2020.
[10] M. A. Gulzar, S. Mardani, M. Musuvathi, and M. Kim, "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, (New York, NY, USA), p. 290–301, Association for Computing Machinery, 2019.
[11] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner, "Sedge: Symbolic example data generation for dataflow programs," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 235–245, IEEE, 2013.
[12] C. Olston, S. Chopra, and U. Srivastava, "Generating example data for dataflow programs," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, (New York, NY, USA), pp. 245–256, ACM, 2009.
[13] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo, "Applying combinatorial test data generation to big data applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), p. 637–647, Association for Computing Machinery, 2016.
[14] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 975–985, 2019.
[15] T. Brennan, S. Saha, and T. Bultan, "Jvm fuzzing for jit-induced side-channel detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1011–1023, 2020.
[16] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, IEEE, 2018.
[17] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *42nd International Conference on Software Engineering*, ACM, 2020.
[18] Y.-D. Lin, F.-Z. Liao, S.-K. Huang, and Y.-C. Lai, "Browser fuzzing by scheduled mutation and generation of document object models," in *2015 International Carnahan Conference on Security Technology (ICCST)*, pp. 1–6, IEEE, 2015.
[19] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 787–802, IEEE, 2019.