FinFuzzer: One Step Further in Fuzzing Fintech Systems

Qingshun Wang*, Lihua Xu†, Jun Xiao‡, Qi Guo‡, Haotian Zhang‡, Liang Dou*, Liang He*, Tao Xie§
*East China Normal University, †New York University Shanghai, ‡Ant Group Co. Ltd., §Peking University;
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China

***Abstract*—Comprehensive testing is of high importance to ensure the reliability of software systems, especially for systems with high stakes such as FinTech systems. In this paper, we share our observations of the Ant Group's status quo in testing their financial services, specifically on the importance of properly transforming relevant external environment settings and prioritizing input object fields for mutation during automated fuzzing. Based on these observations, we propose FinFuzzer, an automated fuzz testing framework that detects and transforms relevant environmental settings into system inputs, prioritizes input object fields, and mutates system inputs on both environment settings and high-priority object fields. Our evaluation of FinFuzzer against four FinTech systems developed by the Ant Group shows that FinFuzzer can outperform a state-of-the-art approach in terms of line coverage in much shorter time.**

***Index Terms*—FinTech, Fuzzing, Software Testing**

I. Introduction

Information technology is playing an increasingly important role in financial services. The Ant Group (in short as Ant) is one of the largest companies that specialize in automated financial services. In 2020, the number of global users of Alipay (the main application of Ant) exceeded 1.2 billion, and its financial services supported 720 million consumers and 28 million small, medium, and micro enterprises. Because any unexpected failure on such financial services could cause huge losses, comprehensive testing [1], [2] is highly important within Ant.

Unfortunately, although core financial services (i.e., those directly related to core business) such as payments are comprehensively tested, peripheral services (still being highly critical despite not directly related to core business) are tested with only 50% to 70% line coverage. One example of these peripheral services provides risk information about loan business to partner banks, and receives hundreds of thousands query requests every minute. Another example provides important data for multiple services involving more than 30 million end users.

To improve code coverage of these peripheral services, recording and replaying real data from the field are a natural option (given that many systems developed by Ant record real input data when serving users live) but with limited effectiveness because replaying recorded historical data usually does not reproduce the same execution path. In particular, we observe that external environment settings can impact program behaviors, causing such non-reproducibility. For instance, programs may access the underlying database to fetch crucial data or invoke a remote procedure call (RPC) to attain global configuration values. These environmental settings have a significant impact on system control flow and execution results, and their corresponding global configuration values may vary every time when the historical data are replayed.

Thus, instead of recording and replaying real data, Ant adopts a context-aware adaptive fuzzing approach [3] to apply object-field-level mutations on existing input data (e.g., recorded historical data) in the form of Java objects[1] but still with insufficient effectiveness. In particular, we observe that bluntly fuzzing on all input object fields results in poor testing effectiveness. According to our investigation, multiple services share the same type of input data object, and each such service reads/writes only some but not all object fields; hence, treating all fields equally during the fuzzing process for a service under test may result in substantial wasted efforts. Additionally, our further in-depth investigation shows that some object fields indeed have more impact on the program execution compared to others.

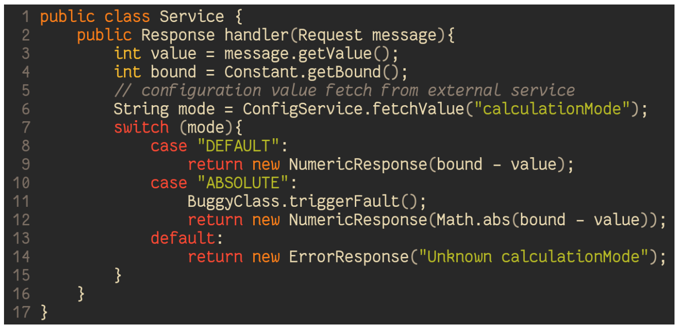
To tackle the aforementioned issues, in this paper, we propose FinFuzzer, a fuzz testing framework that treats external environmental settings as normal program inputs, prioritizes the input object fields, and mutates upon them. In particular, FinFuzzer first detects external environment settings by intercepting the corresponding methods and replace their return values with values provided by the corresponding generated test input. Thus the environmental settings are transformed as system inputs, and can be treated as normal parameters of the program. Furthermore, FinFuzzer prioritizes input object fields by analyzing the historical input-output pairs and guides the mutation process with the prioritized fields.

We apply FinFuzzer on four real systems developed by Ant. The result shows that FinFuzzer is able to achieve higher line coverage than the one achieved by merely replaying historical data in a short period of time. We also compare the result with a state-of-the-art approach named Zest [4], and the result shows that FinFuzzer's overall effectiveness outperforms Zest even when the data generator used by Zest is specifically optimized manually.

In summary, this paper makes the following main contributions:

- The first fuzz testing framework named FinFuzzer that treats external environment settings as first-class input parameters.
- A lightweight technique to prioritize object fields in test inputs.
- Empirical results of applying FinFuzzer on four systems developed by Ant.

[1]Most of the peripheral services developed by Ant take Java objects (an inherently highly structured data format) as their inputs.



```
1  public class Service {
2      public Response handler(Request message){
3          int value = message.getValue();
4          int bound = Constant.getBound();
5          // configuration value fetch from external service
6          String mode = ConfigService.fetchValue("calculationMode");
7          switch (mode){
8              case "DEFAULT":
9                  return new NumericResponse(bound - value);
10             case "ABSOLUTE":
11                 BuggyClass.triggerFault();
12                 return new NumericResponse(Math.abs(bound - value));
13             default:
14                 return new ErrorResponse("Unknown calculationMode");
15         }
16     }
17 }
```

Fig. 1: An illustrative example that external environment settings influence program execution path

II. Observations and Challenges

We next illustrate the observations and findings in Ant's status quo of the fuzz testing process.

External environment settings. External environment settings are common in real-world programs, such as database access, network requests, and RPC. Some of these environment settings may have great impact on the program execution. For instance, the program execution may follow different branches based on the global configuration values fetched from a remote service through an RPC invocation. However, these data are typically neglected when the historical data are recorded. It is unlikely to reproduce the program execution during replay, causing not to cover code branches that can be triggered by only certain environment settings.

Figure 1 shows an illustrative example for this problem. No matter what value is assigned to the parameter "message", the configuration value stored in an external service is the crucial factor in exploring different branches. If the configuration value is fixed to "DEFAULT" during the testing process, the faulty method in Line 11 can never be reached.

High-priority object fields vs. unused object fields. Another common situation is also frequently observed when fuzz testing is bluntly applied on the historical data. To illustrate the problem, Table I shows a simplified example. Four input data and possible corresponding outputs are listed in the table. **PA**, **PB**, and **PC** are fields contained in the input data, and **PD** and **PE** are fields contained in the output data. **PA**, **PB**, **PC**, and **PD** each contain two different equivalent categories: a_1 and a_2, b_1 and b_2, c_1 and c_2, and d_1 and d_2. **PE** contains four different equivalent categories: e_1, e_2, e_3, and e_4. For example, for input data <**PA**: a_1, **PB**: b_1, **PC**: c_1>, **PD** in the corresponding output data always belongs to d_1, but **PE** can belong to either e_1 or e_2 (the reason for **PE** to fall into two equivalent categories is that there exists impact from environment settings).

As shown in Table I, different object fields indeed have different levels of influence on the program execution. If we compare the input data with IDs 1 and 2, the only difference is the category of value assigned to **PA**, but values of both fields in their corresponding output change to another category. But **PB** affects only **PE**, if we compare the data with IDs 1 and 3. There are also fields that do not impact the output at all: the category of value assigned to **PC** differs between data with IDs 2 and 4, but the output does not change.

With in-depth analysis, we observe that there is a common pattern in Ant development to use a big data object to carry all possible information. When the upstream sends a request, all related information is serialized into the object and dispatched to all components that may respond to that request. Since each component has different responsibilities, not all information is used by each component. Therefore, although the input object may contain hundreds of different fields, the service under test may be interested in only a small part of them and will not access the others. Hence exploring different values for these unused fields will not contribute to code coverage. Likewise, a small portion of object fields will have greater impact on the program execution.

TABLE I: An illustrative example that different fields have different impacts on program execution

ID	Input Fields	Output Fields
1	**PA**: a_1, **PB**: b_1, **PC**: c_1	{**PD**: d_1, **PE**: e_1}, {**PD**: d_1, **PE**: e_2}
2	**PA**: a_2, **PB**: b_1, **PC**: c_1	{**PD**: d_2, **PE**: e_3}
3	**PA**: a_1, **PB**: b_2, **PC**: c_1	{**PD**: d_1, **PE**: e_1}, {**PD**: d_1, **PE**: e_4}
4	**PA**: a_2, **PB**: b_1, **PC**: c_2	{**PD**: d_2, **PE**: e_3}

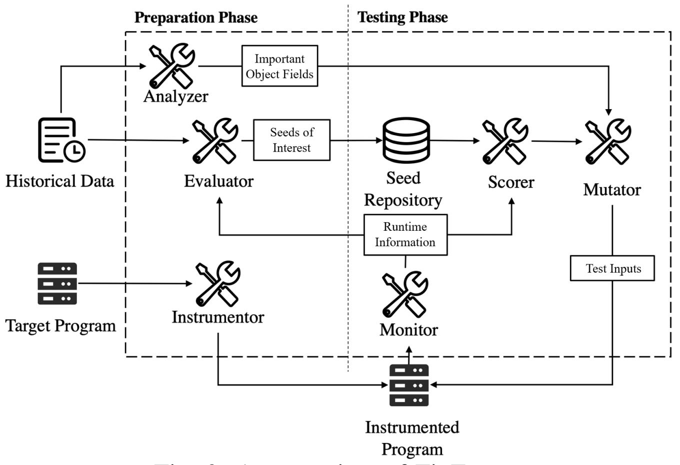


Fig. 2: An overview of FinFuzzer

III. Our Approach

To tackle the observations and challenges described in Section II, we propose FinFuzzer, a fuzz testing framework that treats external environment settings as first-class input parameters and mutates on prioritized input object fields. Figure 2 provides an overview of FinFuzzer.

In a nutshell, the overall testing process of FinFuzzer can be described as follows:

1) Analyzer recognizes important fields from the historical data; Instrumentor detects possible external environment settings inside the target program and transforms them into normal program inputs. In the meantime, Evaluator picks meaningful inputs from the historical data. These inputs are saved in the Repository as the initial set of seeds.
2) Scorer then calculates a score for every seed stored in the Repository and selects one of them based on the priority of the scores.
3) Mutator then produces a new test input based on the selected seed, using the strategy described in Section III-B2.
4) The new input data is then used to test the target program. During the execution, all external settings are replaced by values provided in the test inputs by Instrumentor.

5) When the execution terminates, Evaluator decides whether the current input should be saved as a new seed according to runtime information collected by Monitor.
6) Repeat from the second step until the termination condition is met.

A. Preparation Phase

Important data are collected and analyzed in this phase in preparation for the testing phase. The following components are involved in this phase and act in parallel.

1) ***Instrumentor****:* During this phase, Instrumentor is responsible for detecting possible external environment settings existing in the target program. We learn from Ant developers that we can make the following assumptions for almost all systems developed by Ant:

- The only sources of external environment settings are database access and RPC, which we refer to as "external inputs" from now on.
- All external inputs are fetched through methods following similar coding conventions. For example, all database accesses are delegated to methods defined in interfaces under a certain package, whose name should always end up with "dao" (Data Access Object). All RPC methods must be registered in the configuration.

Therefore, detecting external inputs can be simplified to find packages and interfaces whose names match the identified patterns and extract necessary information from the configuration file. FinFuzzer then transforms the external inputs into normal system inputs. More specifically. FinFuzzer uses a Java agent to modify the loaded classes, and then replaces all invocations to intercepted methods with a stub that returns a value provided by the test inputs.

2) ***Analyzer****:* Analyzer prioritizes the object fields and identifies the high-priority fields to guide later mutation. We design here a lightweight technique instead of traditional program analysis techniques based on the following observations:

- Complex frameworks adopted by Ant systems cause various runtime problems when we apply traditional program analysis techniques such as taint analysis [5].
- Fields that have great impact on the execution path are the ones that lead to different categories in output.
- Fields of enumeration type are most likely to be high-priority fields across services.
- Fields of enumeration type, however, cannot be simply distinguished from its data type, because the coding convention in Ant is to use String or Integer to represent an enumeration value for flexibility and extensibility.

Analyzer consists of two phases. The first phase classifies data types of each object field. We design a machine-learning-based technique, where the model is internally trained for all Ant systems based on the following principles:

- The name of a field usually indicates some important information, e.g., fields with a name that ends with "status" or "type" are highly likely to represent enumeration values.
- Fields of enumeration types have a limited number of meaningful values; thus, the value range of historical values of these fields is usually small.

The second phase prioritizes and distinguishes high-priority fields through the following four steps:

1) Identify input categories for each object field based on their type. For example, for the enumeration type, each possible value belongs to a different category. Numeric types are divided into three categories: less than zero, equal to zero, and greater than zero. Other types are divided into two categories: empty and non-empty.
2) Map from an input category combination to its possible corresponding output category combinations to produce mappings. The resulting mappings are similar to the example shown in Table I.
3) Prioritize fields using the following algorithm. First, each field is deleted from input category combinations one by one. After each deletion, if there are duplicated input combinations, merge their corresponding output sets. If the size of the merged output set is greater than the max size of all the original output sets (before being merged), this field is considered to have ability to affect the execution result. Thus the field is restored and the mappings are not updated. Otherwise update the mappings and move on to delete the next field.
4) The remaining fields after iterating the previous step on each field are considered to have high priorities.

3) ***Evaluator****:* An initial set of seed data from the historical data is constructed, so FinFuzzer does not need to start from scratch. Evaluator decides whether an input should be reserved as a new seed in the Repository based on the recorded information. The selection criteria are similar to existing tools such as AFL [6]: any test input that results in a new state transition (such as entering a new branch or executing a unique path that has never been executed by previously selected seeds) is saved.

Evaluator is also used in the testing phase. Once a new input is generated and fed to the program under test, Evaluator also decides whether it should be saved in the Repository using the same criteria.

B. Testing Phase

The testing phase is where actual testing happens. During this phase, FinFuzzer generates new test inputs, feeds them to the program under test, monitors the execution, and collects necessary information. The functionalities of the components involved in this phase are described below.

1) ***Scorer****:* The functionality of Scorer is to accelerate the fuzzing process by prioritizing the currently best seeds to be mutated to produce new test inputs. The priority of each seed is decided by their score, and seeds with higher scores are more likely to be selected. The algorithm for calculating the score uses a context-aware adaptive strategy inspired by previous research [3], where the score of a seed is decided by the number of branches that are not covered by any seed and are "close" to the execution path of this seed.

2) ***Mutator****:* Mutator produces new test inputs that will later be fed to the program under test. New test inputs are generated by mutating the seed object selected by Scorer. Mutator performs field-level mutations and will modify only a single field every time, and according to a recent study [7],

TABLE II: Line coverage comparison

System Name	Replay Historical Data	Zest with Fuzzing External Inputs		FinFuzzer		FinFuzzer without Prioritizing Fields	
	Line Coverage	Line Coverage	Time Required	Line Coverage	Time Required	Line Coverage	Time Required
System1	77.84%	83.99%	900s	83.99%	80s	83.99%	120s
System2	86.71%	90.86%	1205s	91.80%	115s	91.80%	435s
System3	68.52%	84.27%	400s	84.67%	80s	84.67%	110s
System4	88.01%	96.65%	435s	97.88%	65s	97.88%	220s

TABLE III: Influence of external input

System Name	Max Line Coverage	Without Fuzzing External Input
System1	83.99%	79.14%
System2	91.80%	83.90%
System3	84.67%	82.26%
System4	97.88%	97.88%

doing so can help detect most of faults in the program under test.

The field to be modified is selected upon the information provided by Analyzer. Important fields and enumeration type fields will have higher priorities to be selected. The new value that will be assigned to the selected field to produce a new test input comes from the following sources of data:

- Historical values of the same field, to explore different combinations of historical data.
- A set of predefined extreme values, such as empty string, zero, *NULL*, and *Integer.MAX_VALUE* to explore exceptional situations.
- Constant values extracted from the bytecode, to explore possible meaningful values that are not covered in the historical data. Including these values is especially helpful to find meaningful values for fields with enumeration types.
- Random values.

3) ***Monitor:*** Monitor collects runtime information of the target program when any test input is fed to it. Typically, the code coverage achieved by each test input is recorded in the database and is used by Evaluator. In addition, assertion violations, unexpected exceptions, crashes, and execution time-out are also monitored as they indicate faults detected during testing.

IV. Evaluation

In this section, we apply FinFuzzer to four real software systems[2] developed by Ant and intend to answer the following two research questions:

- **RQ1**: How much influence do external inputs have on program execution paths?
- **RQ2**: How effective and efficient is FinFuzzer compared with a state-of-the-art approach?

To answer **RQ1**, three out of these four systems (System1 to System3) implement two different strategies for part of its business logic. Only one of the two strategies is executed during any system execution, and is controlled by the configuration server. Multiple different machines are assigned to run different strategies so that different branches of the system under test are guaranteed to run separately without interference. The external input in our evaluation is set in a method *switchOld2New*, whose return value depends on the data fetched from the configuration server. Only one of the implemented strategies is executed based upon the returned value. In our evaluation, we collect and compare the results of line coverage (measured by *Jacoco* [8]) for FinFuzzer with the ability of fuzzing external inputs disabled, and find that the coverage for all three systems is decreased, as shown in Table III. For comparison purposes, System4 does not contain external inputs and the resulting coverage remains the same.

To answer **RQ2**, we again use line coverage to evaluate the effectiveness of different approaches. We compare the line coverage and the minimum time to achieve this coverage from four different approaches, namely, replaying the historical data, Zest [4] with fuzzing external inputs, FinFuzzer, and FinFuzzer without prioritizing fields.

Note that we use Zest [4] as a state-of-the-art fuzz testing approach, and provide additional abilities to fuzz the external inputs. For each of the systems used in the evaluation, we manually write a Zest generator, so that a wrapper object contains the mock data for the external inputs [9].

As shown in Table II, both FinFuzzer and Zest (with fuzzing external inputs) are able to cover more lines compared to merely replaying historical data. We do not collect time required for replaying historical data because every replaying involves tremendous man power to manually screen through all the recorded data and pick the useful ones by experiences.

Moreover, FinFuzzer is able to achieve similar or slightly higher line coverage than Zest (with fuzzing external inputs), but in a much shorter time, with two main reasons. (1) There is no easy way to "deserialize" an existing input data into the bit sequence used by Zest; thus, we cannot construct an initial set of seeds, and Zest has to generate all input data from scratch. (2) Zest does not have any information about the high-priority fields, and hence has to conduct analysis during the fuzzing process. If we take the ability of prioritizing object fields out of FinFuzzer, the time required for achieving the same line coverage increases by 37.5% to even 278.3%.

V. Conclusion

In this paper, we have shared our observations on automated testing practices at Ant. We have proposed a new fuzz testing framework named FinFuzzer that can automatically detect and intercept external inputs, as well as prioritizing important object fields that may have impact on program execution to guide the fuzzing process. The evaluation results show that FinFuzzer can efficiently achieve better line coverage than previous approaches adopted by Ant and a state-of-the-art approach.

VI. Acknowledgments

This work was supported by the Science and Technology Commission of Shanghai Municipality Grant (No. 20511102502). Liang Dou and Liang He are corresponding authors.

[2]Concrete information including system names involved in our evaluation is anonymized in this paper, for confidentiality considerations.

References

[1] Q. Wang, L. Gu, M. Xue, L. Xu, W. Niu, L. Dou, L. He, and T. Xie, "FACTS: Automated black-box testing of FinTech systems," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, p. 839–844.

[2] T. Jin, Q. Wang, L. Xu, C. Pan, L. Dou, H. Qian, L. He, and T. Xie, "FinExpert: Domain-specific test generation for FinTech systems," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, p. 853–862.

[3] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 533–544.

[4] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with Zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[5] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2007, p. 196–206.

[6] M. Zalewski. (2020) American Fuzzy Lop. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[7] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. L. Goues, and P. Koopman, "Robustness testing of autonomy software," in *Proceedings of 40th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018, pp. 276–285.

[8] "JaCoCo Java Code Coverage Library," https://www.eclemma.org/jacoco/, accessed in 2021.

[9] L. Zhang, X. Ma, J. Lu, T. Xie, N. Tillmann, and P. de Halleux, "Environmental modeling for automated cloud application testing," *IEEE Software*, vol. 29, no. 2, pp. 30–35, 2012.