FACTS: Automated Black-Box Testing of FinTech Systems

Qingshun Wang[1], Lintao Gu[1], Minhui Xue[2], Lihua Xu[1,3*], Wenyu Niu[4], Liang Dou[1*], Liang He[1], Tao Xie[5]
[1]East China Normal University, China [2] Optus Macquarie University Cyber Security Hub, Australia
[3]New York University Shanghai, China [4]CFETS Information Technology Co. Ltd., China
[5]University of Illinois at Urbana-Champaign, USA

ABSTRACT

FinTech, short for "financial technology," has advanced the process of transforming financial business from a traditional manual-process-driven to an automation-driven model by providing various software platforms. However, the current FinTech-industry still heavily depends on manual testing, which becomes the bottleneck of FinTech industry development. To automate the testing process, we propose an approach of black-box testing for a FinTech system with effective tool support for both test generation and test oracles. For test generation, we first extract input categories from business-logic specifications, and then mutate real data collected from system logs with values randomly picked from each extracted input category. For test oracles, we propose a new technique of priority differential testing where we evaluate execution results of system-test inputs on the system's head (*i.e.*, latest) version in the version repository (1) against the last legacy version in the version repository (only when the executed test inputs are on new, not-yet-deployed services) and (2) against both the currently-deployed version and the last legacy version (only when the test inputs are on existing, deployed services). When we rank the behavior-inconsistency results for developers to inspect, for the latter case, we give the currently-deployed version as a higher-priority source of behavior to check. We apply our approach to the *CSTP* subsystem, one of the largest data processing and forwarding modules of the *China Foreign Exchange Trade System* (*CFETS*) platform, whose annual total transaction volume reaches 150 trillion US dollars. Extensive experimental results show that our approach can substantially boost the branch coverage by approximately 40%, and is also efficient to identify common faults in the FinTech system.

CCS CONCEPTS

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

KEYWORDS

FinTech; Black-box testing; Automated test generation

ACM Reference Format:
Qingshun Wang, Lintao Gu, Minhui Xue, Lihua Xu, Wenyu Niu, Liang Dou, Liang He, and Tao Xie. FACTS: Automated Black-Box Testing of FinTech Systems. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236024.3275533

1 INTRODUCTION

FinTech, short for "financial technology," refers to a set of new information technologies to innovate financial services. These latest innovations not only benefit consumers but also push businesses to transform from a manual-processing-driven model to an automation-driven model by creating robust and high-quality software systems. According to Citigroup, an estimated $19 billion of investment poured into FinTech in 2015 [1].

China Foreign Exchange Trade System (CFETS*) Information Technology Co. Ltd.*, as a subsidiary of *China's Central Bank*, provides a trading platform for exchange rate swaps as well as currency swaps and forwards for more than 21,000 active clients. The annual total transaction volume reaches 150 trillion US dollars, for which any unexpected failures on trading services could result in huge losses. Therefore, it is critical to conduct sufficient testing on such FinTech software systems to ensure system robustness and correctness.

Unfortunately, according to our field observation, despite its increasing needs, the current state of the practice in the FinTech industry still heavily relies on manual testing, due to high complexity of such FinTech systems. We have made attempts to apply existing testing tools to the Foreign Exchange Platform developed and owned by *CFETS*. We name such platform as the *CFETS* platform, which consists of multiple *CFETS* subsystems, and each subsystem provides a set of services to the users. Based on our such attempts, we identify three main challenges that hinder direct adoption or adaptation of existing testing techniques and tools.

First, a *CFETS* subsystem takes high-dimensional inputs, each of which contains a group of fields with drastically different data types. Such high-dimensional inputs are beyond the capabilities of most constraint solvers, which are the underlying engine for dynamic symbolic execution tools, such as KLEE [3], EXE [4], and Pex [11], posing challenges to apply these white-box testing tools.

Second, each input dimension is defined as various data types and formats, typically of complex user-defined data structures. Most automated test generation tools, such as *Randoop* [9] and *EvoSuite* [5], fail to obtain the input type, and let alone generate valid input values.

Third, although specification-based robustness testing tools, such as *Ballista* [7], can be applied to a *CFETS* subsystem, testing the system requires manually defined exceptional input values, which are missed by such tools due to the high complexity of the subsystems.

[*] Lihua Xu is the corresponding author. Email: lihua.xu@nyu.edu.
[*] Liang Dou is the corresponding author. Email: ldou@cs.ecnu.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275533

Moreover, to ensure its intended functionalities, testing the system needs not only a set of exceptional input values, but also normal input values in the huge value space of system inputs.

Even worse, due to a *CFETS* subsystem's complex behavior and various formats of output, it is practically impossible to completely portray the system behavior as assertions, and hence executable test scripts with assertions are impractical for the system. Additionally, we observe that it is a central business requirement for *CFETS* (as a service provider to serve more than 21,000 active clients) to render the existing deployed services unchanged by users' perspective during the services' constant system upgrades. As client population grows drastically by day, new services are constantly added to the deployed system to supplement the existing deployed services during these system upgrades. Even before these new services' deployment, they undergo substantial evolution during the development process. Before deployment, these new services do not have their counterpart services in the deployed system as the reference version required in traditional differential testing. As a result, *CFETS* still primarily relies on domain experts to manually examine the execution results for every test input, being very time consuming and error prone.

In the end, we reach a consensus with the domain experts and developers at *CFETS* that what we need is not only an automated testing generation technique, but also a means to evaluate the execution results for these generated inputs. Additionally, both the normal and exceptional inputs are equally important to test the system.

In this paper, we aim at studying the aforementioned challenges of automatically testing such FinTech systems. To achieve this goal, we propose **Facts**, Automated BlaCk-box Testing for FinTech Systems.

To address the challenges of high-dimensional inputs, we leverage the input data collected from system logs as seeds. Messages collected from real execution runs serve as the basis of these high-dimensional inputs. To handle the user-defined data type for each input dimension, we leverage the business-logic specification to retrieve their data types. To tackle the challenges of exploring the huge input space, we extract input categories from business-logic specifications, and randomly sample from each input category to produce both normal and exceptional inputs.

Finally, to tackle the challenges of the oracle-lacking problem for new services before deployment, we propose a new technique of priority differential testing where we evaluate execution results of system-test inputs on the system's head (*i.e.*, latest) version in the version repository (1) against the last legacy version in the version repository (only when the executed test inputs are on new, not-yet-deployed services) and (2) against both the currently-deployed version and the last legacy version (only when the test inputs are on existing, deployed services). For the head version of existing services, the behavior of the currently-deployed version (already experienced by many users) represents a higher-priority reference behavior to check against, than the last legacy version in the version repository. However, for the head version of new, not-yet-deployed services, given that there exists no currently-deployed version, we leverage the last legacy version in the version repository to check against. Note that any new service added to the system could potentially affect the behavior of the existing services (with their earlier version deployed already). Despite stable, the last legacy version may still include unintended behaviors for the existing services due to the addition of new services. Hence we consider the currently-deployed version to have higher priority to check against than the last legacy version when both versions are available for existing services under test.

Applying Facts to a major subsystem of the *CFETS* platform helps achieve a substantial increase on branch coverage, by more than 40% in comparison to the original manual tests. Such benefit comes from our generation of both normal and exception inputs, while the record of manual testing shows that testers mainly focus on normal inputs. The improvements over manual testing effort are not only statistically significant but also practically substantial, given that the number of active *CFETS* clients is daily increasing. It is not unusual to have exceptional inputs from such a large-scale input space and any failure behavior caused by exceptional inputs may lead to great financial losses.

Furthermore, the priority differential testing built in Facts is able to quickly prioritize problematic services, greatly reducing human effort. Selectively using both the currently-deployed version and the legacy version, the execution of most generated test inputs can be checked automatically. In our current implementation, in addition to commonly used CRASH metrics [8], we raise concerns to different types of inconsistent behaviors between different versions. The top of the prioritized list is the inputs that trigger the currently-deployed version and the legacy version to produce the same result while the head version shows a different result, which we consider has the highest probability of containing failures due to faults. The second top to the list is the inputs that trigger all three versions to produce different results, which need to be checked manually. The full ranked list is described in Section 4.2. All these inconsistent behaviors are recorded, but records with low priority can be filtered by developers first so that they can focus on those inputs whose executions are likely to trigger failures due to faults. In our empirical study, we find that only a very small portion of test inputs need to be examined manually instead of the whole set of test inputs.

In summary, this paper makes the following main contributions:

- The first industrial case study of testing a FinTech software system with over 21,000 active clients including banks, investment funds, etc.
- A new technique to automatically test a software system with high-dimensional inputs including various data types.
- A new technique of priority differential testing to automatically evaluate and rank test results.

2 BACKGROUND

To illustrate the challenges faced in testing a FinTech system, we take as an illustrative example the *CSTP* subsystem, owned and developed internally by *CFETS* as one of the subsystems of the *CFETS* platform for data processing and forwarding.[1] We choose to target on *CSTP* because (1) it belongs to the fundamental part of the *CFETS* platform, which is responsible for processing all data generated in each transaction, transforming them into correct format, and forwarding to appropriate targets; (2) its main functionality

[1] We have access to all source code and previous system logs under a non-disclosure agreement.

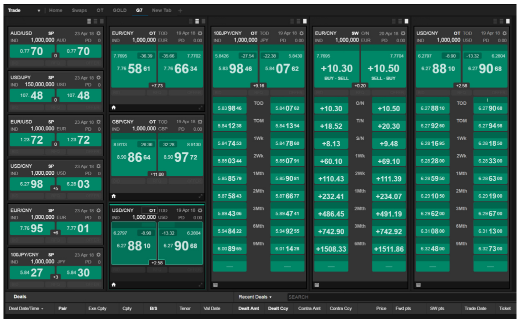


Figure 1: *CFETS* platform

Table 1: Data field for input messages

FieldName	DataType	Length	Format/Accuracy	Range
ValueDate	Date	10	YYYY/MM/DD	-
LendingAmount	Number	15	2	-
ClearingMethod	Enum	-	-	A,B,C
TraderCode	Text	-	*TraderID@InstituteID*	-
QuoteID	Text	-	a.b.ccccccc	-
....	....	...	...	...

includes parsing messages and manipulating on data, making it possible to observe its behavior through outputs and logs.

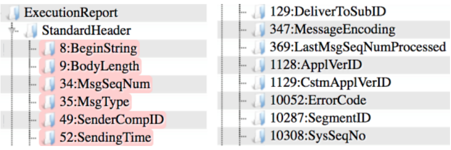
During the operation of the *CFETS* platform, whose interface is shown as Figure 1, the *CSTP* subsystem receives from upstream subsystems message data generated for every transaction, applies proper transformation to the data, translates to the IMIX format, which is a standard electronic communication protocol for real-time transmission of financial information in the inter-bank market trading activities.[2] The resulting IMIX message is then forwarded to appropriate institutions subscribed in the system and gives the corresponding permissions to access the data.

The message data, *i.e.*, the input data of the *CSTP* subsystem, is shown in *key-value* pairs. Each input message consists of two parts: a *Header* represents the message type, followed by a sequence of *key-value* pairs. Each *key-value* pair corresponds to a *field*, which describes a part of information about the transaction related to the message. The *key* corresponds to the *Field Name*, and the *value* can be one of the various data types, with some specific requirements. Table 1 shows the data-field requirement for each message type. Some fields have additional domain-specific regulations, such as the *QuoteID* field. We do not include details here due to space limit. A valid value of a field should meet its requirement.

Our goal is that invalid values, such as date values not matching the format, or numbers out of range, should be properly detected by the system, thus avoiding crashing the whole system. In the meantime, valid values should be properly handled, and their execution results should be checked against expected results.

3 CHALLENGES

Software testing is the process of generating and executing test inputs to cause failures for the system under test. It has been shown to be an effective way to ensure the robustness and correctness of the system under test. However, manually writing test cases (*i.e.*, test inputs and test oracles) for a large and complex system under test, such as the *CSTP* subsystem, can be highly expensive [10]. Thus, there exist many automated testing techniques and tools, such as *Randoop* [9] and *EvoSuite* [5], which automatically generate test inputs and execute them to reduce human effort. Unfortunately, such existing automated testing techniques and tools cannot be effectively applied to test a FinTech system due to four main challenges as listed below.



ExecutionReport
StandardHeader
8:BeginString
9:BodyLength
34:MsgSeqNum
35:MsgType
49:SenderCompID
52:SendingTime
129:DeliverToSubID
347:MessageEncoding
369:LastMsgSeqNumProcessed
1128:ApplVerID
1129:CstmApplVerID
10052:ErrorCode
10287:SegmentID
10308:SysSeqNo

Figure 2: A subset of fields in an *ExecutionReport* message

High-dimensional input. The *CSTP* subsystem processes transaction data as messages. The messages are stored and transmitted as different types, each of which contains hundreds of different fields. Figure 2 shows a small subset of fields contained in a message of type *ExecutionReport*. Test-input generation for such subsystems requires constructing such a complex message structure, which can be very challenging for state-of-the-art testing tools, as well as time consuming and error prone for current manual testing.

Various data types and formats. Each input dimension, *i.e.*, the fields of each message, has again drastically different data structures, many of which are user-defined and domain-specific (see Table 1). It is difficult to extract such data types and generate meaningful data accordingly. As a result, existing tools, such as *Randoop* and *EvoSuite*, generate plenty of trivial exceptional inputs, which typically trigger an exception and then are blocked at the beginning of the system execution. These generated inputs are inefficient to test the system, and let alone cause faults located deep in the system to be exposed as failures.

Huge value space. To explore the input space of the *CSTP* subsystem, one needs to explore the Cartesian product of input dimensions. Thus it is difficult for fuzzing to effectively expose failures in the *CSTP* subsystem, especially for those that can be triggered by only specific corner cases of related input dimensions. Specification-based testing tools, such as *Ballista* [7], can be applied to eliminate the pressure by constructing inputs using values drawn from a pre-defined dictionary. It typically requires to manually define the input values that are likely to trigger failures; such manual process is very time consuming given the complexity of *CSTP*’s input domain. Additionally, it is highly challenging to produce a complete set of values. If a failure is triggered by a specific value not included in the dictionary, such tool then would fail to expose the corresponding failure.

Lack of testing oracles. Faults that lead to certain generic failures, such as system crashes and hangs, can be detected without strong test oracles. However, the functional correctness of the system under test is also of great value, and faults that lead to functional incorrectness on system outputs cannot be detected without strong test oracles. For large-scale FinTech systems, incorrect system outputs may lead to a huge loss. Manually verifying the output of every generated test input is too expensive and infeasible. Similarly, specifying an oracle for black-box testing requires huge amount of work, and a comprehensive oracle for the *CSTP* subsystem is not yet available due to its complexity.

[2] http://imix.chinamoney.com.cn/

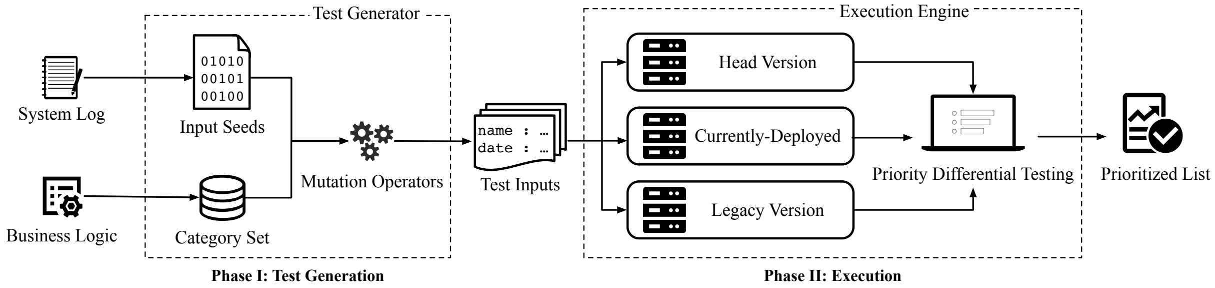


Figure 3: Overview of Facts

ALGORITHM 1: Equivalent-Category Partitioning

```
Input: spec, the message business-logic document for the message type.
Output: eqvCatMap, a map message structure that contains the equivalent
        category information for each field.
dataFormatMap = parseDocument(spec);
eqvCatMap = new EmptyMap;
for dataFormat ∈ dataFormatMap do
    fieldName = dataFormat.getFieldName();
    formatInfo = dataFormat.getFormatInfo();
    eqvCatList = partitionEqvCat(formatInfo);
    eqvCatMap.put(fieldName, eqvCatList);
end
return eqvCatMap
```

4 OUR APPROACH

As discussed in Section 2, at a high level, each input data to the *CSTP* subsystem is represented as a message that contains a set of fields, each of which further represents different types of data. Each field has its own requirements and definitions for possible values, and in turn each message is of different type based on its data fields. The *CSTP* subsystem processes many different types of transactions at the same time, each of which has its own set of related data fields. In theory, verifying the robustness and correctness of the *CSTP* subsystem needs to feed it with all possible valid and exceptional messages and check whether the system returns an expected output. In practice, doing so is infeasible. A more practical approach is to craft a set of representative test inputs instead, and cover the system behavior as much as possible. As shown in Figure 3, Facts generates input as objects recognized and processed by the subsystem, and then evaluates the execution results with proposed priority differential testing.

(1) Facts collects system logs from real transactions and retrieves passing messages as input seeds, to construct the high-dimensional test messages; in the meantime, Facts also identifies the input category of each data field for each corresponding test message, and mutates over the seed messages, to cover all possible field categories.
(2) Facts then compares execution results from three versions of the *CSTP* subsystem to determine failure-triggering inputs and outputs a prioritized input list.

Table 2: Mutation operators

Operator	Description
RPL(f, c)	**Replacement**. Replace the value of a field f with a value randomly picked from a category c.
DEL(f)	**Deletion**. Delete a field f appearing in the message.
INS(f, c)	**Insertion**. Insert a field f with a value randomly picked from a category c.

4.1 Test Generation

We first construct the input categories used for deriving test inputs. We take as input the business-logic specification that defines requirements for each data field, and output a map that contains all the equivalent categories of each field (see Algorithm 1). An equivalent category of a field is hereby defined as a set of all possible values, where each element in the same category is equivalent to revealing the system's behavior. For example, for a field of data type "Date," all the valid and invalid values can be classified into different categories, respectively. Each category can be further subdivided accordingly. Different types of incorrectness may involve different parts of code in the system. A date value in a wrong format may be blocked by the first check in a method of the system, while an input in a correct format, however representing a nonexistent date, *e.g.*, Feb 30th, may pass through and crash the whole system. Valid inputs can also be divided into smaller pieces, such as a field of data type "Enum," which has a finite set of possible values. Different choices from a set can lead the system execution to execute different branches, each of which belongs to a unique category.

In theory, a category is a set of values with some properties in common. However, it is infeasible to enumerate every element in the set in real-world applications. In Facts , an equivalent category is characterized with a finite set of rules. "Pick an element from the category" is realized with generating a value that satisfies a rule for the equivalent category. For example, a category that contains all strings with a specific format can be described as a regular-expression pattern (such as strings representing a date). We can construct an automata for the corresponding pattern that walks through from the start state to the accept state. The composition of symbols used in the transition is the value that we expect.

Once the equivalent categories are obtained, Facts generates test messages based on real messages collected from actual transactions taking place on the platform. Test inputs are created by mutating the collected real messages and transformed to corresponding objects to the subsystem. We define three types of mutation operators in Table 2. Each operator takes at least one parameter that points to the target field. Randomly picking a value from a given category is equivalent to constructing a value that satisfies the rule characterizing the category, as mentioned earlier.

ALGORITHM 2: Input Generation

```
Input: eqvCatMap, the output of the Phase 1.
Input: initMessageSet, a set of real test messages extracted from the system log.
Output: testMessageSet, the test messages generated.
testMessageSet = new EmptySet;
for Message $m \in$ initDataSet do
    fieldList = getFields(m);
    for Field $f \in$ fieldList do
        Generate a new message $m_D$ by applying DEL($f$) on $m$;
        testMessageSet.add($m_D$);
        categoryList = eqvCatMap.getCategories(f);
        for Category $c \in$ categoryList do
            Generate new messages $m_R$ by applying RPL($f$, $c$) on $m$;
            testMessageSet.add($m_R$);
        end
    end
    missingFieldList = getMissingFields(m);
    for Field $f_m \in$ missingFieldList do
        categoryList = eqvCatMap.getCategories($f_m$);
        for Category $c \in$ categoryList do
            Generate new messages $m_I$ by applying INS($f$, $c$) on $m$.;
            testMessageSet.add($m_I$);
        end
    end
end
return inputSet
```

Table 3: An example of a generated test message

Name	ValueDate	LendingAmount	Clearingmethod	QuoteID	...
Value	"2017/12/21"	"666000.00"	"A"	"6.4.2556816"	...

Algorithm 2 describes how we generate test messages. Due to space limit, we show here a partial example of a generated test message in Table 3. The algorithm takes as input initially-computed *eqvCatMap* and *initMessageSet*. *initMessageSet* is a set of messages collected from real transactions. We iteratively apply three mutation operators to an original message to create a new message with entirely valid or partially invalid values. The mutation is applied to a single field of a newly-created message. Although applying mutation operators to multiple fields may be more comprehensive, the combinatorial explosion could lead to a very large set of test cases that require unaffordable time and computational resources to execute. Thus we choose to apply mutation to a single field, and according to a recent study [6], doing so can help detect most of faults in the system under test.

4.2 Test Oracles

We feed the generated test inputs into the system under test, and record the inputs that cause potential failures. To determine potential failures based on the observed system outputs, we propose the technique of priority differential testing. In particular, we test the system under test with two previous versions in parallel: the currently-deployed version, denoted as S_c, and the last legacy version in the version repository, denoted as S_l. We assume that the currently-deployed version represents the correct existing system behavior, because it has been stably running for a long time, in a platform with a large number of clients, which send countless messages to the system. Such currently-deployed version can be regarded as fully tested. We define that the last legacy version refers to the latest stable version in the version repository; note that the last legacy version includes new services, which do not exist in the currently-deployed version. The version under test is named the head version, denoted as S_h, which typically contains all the services from the last legacy version with additional updates. The currently-deployed version is considered as the primary reference version for the existing system behavior because despite stable, the legacy version may still include uncertain behavior due to the addition of new services. In the meantime, the legacy version is utilized as the reference version for new services, which do not exist in the currently-deployed version. If a test input involves new features, *e.g.*, a test message contains newly defined fields, at least the two newer versions (legacy and head versions) should have the same behavior. We consider any inconsistent output as a potential failure and record the corresponding input.

We prioritize recorded inputs along with their triggered potential failures for developers to inspect. Table 4 lists details of different priorities of different system behaviors, where Rank 1 represents the highest priority, and 4 is the lowest. We assign inputs that cause system crashes or hangs with the top priority, followed by inputs that trigger S_c and S_l to produce the same results while S_h produces a different result. S_h is the most likely to contain faults that break an existing service in this situation. Inputs that trigger all three versions to produce different results may involve new services added in S_h, but another possibility is this situation reveals that S_h breaks some services added in S_l. Both situations need additional manual checking, so we assign such situations with the third top priority. If S_c and S_h produce the same results different from S_l, it may be related to a bug fix. If two newer versions produce the same output but different from S_c, the input may involve some new services in S_l. Although these two situations are very likely to be correct, we still recommend them for developers to further inspect, in the lowest priority.

5 EVALUATION

In this section, we apply our approach to test the *CSTP* subsystem and intend to answer the following two research questions:

- **RQ1**: How effective is our approach to comprehensively test the system?
- **RQ2**: Can our approach effectively expose faults as failures in the system?

To answer **RQ1**, we use branch coverage (measured by using *Jacoco* [2]) to evaluate the effectiveness of our approach after executing the generated test cases. Full branch coverage requires that each branch in the system under test has been executed at least once during testing.

To construct a comparison baseline, we first tried existing automated test generation tools, such as *EvoSuite* and *Randoop*. Unfortunately, as discussed earlier in Section 3, these tools were not able to generate meaningful inputs. Thus we resort to the manually-designed test cases stored at the development repository. Due to the space limit, we show branch coverage for five important classes in the system under test (Table 5), achieved by the manually-written test cases and our approach, respectively.

As shown in Table 5, nearly half of the branches are not covered by manually-written test cases, indicating that many situations, especially exceptional situations of test messages, have never been tested, unable to offer high confidence on the system's robustness.

Table 4: Priority rank

System Behavior	Rank
System crashes or hangs	1
S_c and S_l get same output, different from S_h	2
Three different outputs	3
S_c and S_h get same output, different from S_l	4
S_l and S_h get same output, different from S_c	4

Table 5: Branch-coverage comparison

Class Name	Manually-written Test	Our Approach
FxclDealLogParser	47.56%	91.46%
FxOptionDealLogParser	53.54%	95.79%
FxDealLogparser	51.76%	85.27%
FirdvDealLogParser	56.42%	81.96%
CSTPMemberConfirmInfoParser	54.33%	97.26%

Table 6: Fault cases for the *CSTP* subsystem

Faulty Version	Description
Faulty-1	Remove code involving field *TradingModeCode*.
Faulty-2	Remove code handling a certain value of enum field *ExerciseStatus*.
Faulty-3	Remove code for setting field value of output IMIX message when processing field *QuoteInstitutionTraderCode*.
Faulty-4	Remove code for checking value formats when processing field *DealTime*.
Faulty-5	Set value to wrong field in the output IMIX message when processing field *Period*.

In contrast, our approach can generate test cases that cover most of the branches in the classes under test, offering much higher confidence on the system's robustness.

To answer **RQ2**, we have approached the developers of the *CSTP* subsystem for understanding the system's common faults introduced during past development. It turns out that many faults were introduced because a careless developer neglected to deal with some rarely used data fields or conditions. These faults were not exposed by manually-written test cases, and consequently were uncovered during real transactions upon deployed services.

To investigate whether our approach is able to expose such common faults, we construct real faulty versions to reflect those previously exposed faults during system deployment, and also create additional synthetic faulty versions following the observed common fault patterns, by manually removing code blocks related to a field, or some conditional branches (Table 6 shows the five fault cases). Our experimental results show that our approach is able to identify all of these five fault cases, demonstrating that applying mutation on a single field can effectively expose faults in the system under test as failures.

6 DISCUSSION AND LESSONS LEARNED

In this section, we discuss multiple lessons learned in our collaboration with the developers of the *CSTP* subsystem.

Generic tools are not effective for complex systems. Typically, generic automated test generation tools are not effective in the context of testing a FinTech system. Methods in a FinTech system accept high-dimensional data as parameters, each of which may contain a group of fields with drastically different data types, making it difficult for generic tools to generate effective test cases — in our preliminary studies, state-of-the-art tools, such as *Randoop* and *Evosuite*, cannot even construct a valid argument.

Exceptional inputs should receive more attention. A FinTech system usually serves as an application platform for millions of clients, and a large amount of data is fed into the system each day. Such situation implies that most of the regular execution paths have been fully exercised. If a valid message can trigger a failure, this failure has a great chance of being discovered in a previous system version. In contrast, those invalid messages that contain some exceptional values are less likely to appear in real execution environments. If there exists any fault remaining unrevealed in the system, the fault is more likely to be triggered by exceptional inputs. In our approach, we mutate real data not only with valid values, but also with different types of exceptional values, to check whether there is any fault hidden in irregular execution paths.

Universal oracles do not exist. It is hard to find a universal test oracle for large-scale, complex, and domain-specific systems, such as FinTech systems. *CFETS* still relies on domain experts to examine the execution results for test inputs. We propose priority differential testing to reduce human effort by checking the consistency among multiple versions, but it still takes much time for experts to determine whether an inconsistency reveals a fault in the system.

7 CONCLUSION

In this paper, we have presented our work on testing FinTech systems, in collaboration with *China Foreign Exchange Trade System Information Technology Co. Ltd*, a subsidiary of *China's Central Bank*. To address challenges faced when testing a FinTech system, we have proposed Facts, an approach of black-box testing for a FinTech system with effective tool support for both test generation and test oracles. We have applied our approach to the *CSTP* subsystem, and the evaluation results show that our approach can achieve much higher branch coverage than the system's existing manually-written test cases, and our approach is effective to expose common faults introduced during system development.

ACKNOWLEDGMENTS

This work was supported in part by the Science and Technology Commission of Shanghai Municipality under grant No. 18511103802, and in part by NSFC under grant No. 61502170, NSF under grants No. CNS-1513939, CNS-1564274, and CCF-1816615, and in part by the ECNU travel grant.

REFERENCES

[1] 2017. Global Fintech Investment Growth Continues in 2016. https://www.accenture.com/t20170411T170619Z__w__/id-en/_acnmedia/PDF-15/Accenture-Fintech-Report-London-Lab-News-Release.pdf.

[2] 2018. JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. OSDI*. 209–224.

[4] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution. In *Proc. CCS*. 322–335.

[5] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proc. ESEC/FSE*. 416–419.

[6] Casidhe Hutchison, Milda Zizyte, Patrick E Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. 2018. Robustness Testing of Autonomy Software. In *Proc. ICSE SEIP*. 276–285.

[7] Philip Koopman. 1998. Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security. In *Proc., CSDA*. 103–131.

[8] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. 1997. Comparing Operating Systems Using Robustness Benchmarks. In *Proc. SRDS*. 72–79.

[9] Carlos Pacheco and Michael D Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to OOPSLA*. 815–816.

[10] Ossi Taipale, Jussi Kasurinen, Katja Karhu, and Kari Smolander. 2011. Trade-off between Automated and Manual Software Testing. *International Journal of System Assurance Engineering and Management* 2, 2 (2011), 114–125.

[11] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Proc. DSN*. 359–368.