Identify failure-inducing combinations for multiple faults*

Xintao Niu
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210023
niuxintao@smail.nju.edu.cn

Changhai Nie
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210023
changhainie@nju.edu.cn

ABSTRACT

Combinatorial testing(CT) is proven to be effective to reveal the potential failures caused by the interaction of the inputs or options of the system under test(SUT). A key problem in CT is to isolate the failure-inducing interactions of the related failure as it can facilitate the debugging effort by reducing the scope of code that needs to be inspected. Many algorithms has been proposed to identify the failure-inducing interactions, however, most of these studies either just consider the condition of one fault or ignore masking effects among multiple faults which can bias their identified results. In this paper, we analysed how the masking effect of multiple faults affect on the isolation of failure-inducing interactions. We further give a strategy of selecting test cases to alleviate this impact. The key to the strategy is to prune these test cases that may trigger masking effect and generate no-masking-effect ones to test the interactions supposed to be tested in these pruned test cases. The test-case selecting process repeated until we get enough information to isolate the failure-inducing interactions. We conducted some empirical studies on several open-source GNU software. The result of the studies shows that multiple faults do exist in real software and our approach can assist combinatorial-based failure-inducing identifying methods to get a better result when handling multiple faults in SUT.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging—*Debugging aids,testing tools*

General Terms

Reliability, Verification

Keywords

Software Testing, Combinatorial Testing, failure-inducing combinations, Masking effects

1. INTRODUCTION

With the increasing complexity and size of modern software, many factors, such as input parameters and configuration options, can influence the behaviour of the system under test(SUT). The unexpected faults caused by the interaction among these factors can make testing such software a big challenge if the interaction space is too large. One remedy for this problem is combinatorial testing, which systematically sample the interaction space and select a relatively small size of test cases that cover all the valid iterations with the number of factors involved in the interaction no more than a prior fixed integer, i.e., the *strength* of the interaction.

Once failures are detected, it is desired to isolate the failure-inducing interactions in these failing test cases. This task is important in CT as it can facilitate the debugging effort by reducing the code scope that needed to inspected. Many algorithms has been proposed to identify the failure-inducing interactions in SUT, which include approaches such as building classification tree model [17], generating one test case one time [12], ranking suspicious interactions based on prior rules[8], using graphic-based deduction [10] and so on. These approaches can be partitioned into two categories [5]: *adaptive*–tests cases are chosen based on the outcomes of the executions of prior tests or *nonadaptive*–test cases are chosen independent and can be executed parallel.

While all these approaches can help developers to isolate the failure-inducing factors in failing test cases, in our recently studies on several open-source software, however, we found these approaches suffered from *masking effects* of multiple faults in SUT. A masking effect [6, 18] is an effect that some failures prevents test cases from normally checking combinations that are supposed to be tested. Take the Linux command– *Grep* for example, we noticed that there are two different faults reported in the bug tracker system. The first one [1] claims that Grep incorrectly match unicode patterns with '\<\>', while the second one [2] claims a incompatibility between option '-c' and '-o'. When we put this two

*This work was supported by the National Natural Science Foundation of China (No. 61272079), the Research Fund for the Doctoral Program of Higher Education of China (No.20130091110032), the Science Fund for Creative Research Groups of the National Natural Science Foundation of China(No. 61321491), and the Major Program of National Natural Science Foundation of China (No. 91318301)

WOODSTOCK '97 El Paso, Texas USA

[1]http://savannah.gnu.org/bugs/?29537
[2]http://savannah.gnu.org/bugs/?33080

scenario into one test case only one fault information will be observed, which means another fault is masked by the observed one. This effect was firstly introduced by Dumllu and Ylimaz in [6], in which they found that the masking effects in CT can make traditional covering array failed detecting some combinations and they proposed an approach to work around them. Their recent work [18] further empirically studied the impacts on the failure-inducing combinations identifying approach (FCI approach for short): classification tree approach(CTA for short)[17], of which CTA has two versions, i.e., ternary-class and multiple-class.

As known that masking effects negatively affect the performance of FCI approaches, a natural question is how do this effect bias the results of FCI approaches so that they cannot get the results as accuracy as expected. In this paper, we formalized the process of identifying failure-inducing combinations under the circumstance that masking effects exist in SUT and try to answer this question. One insight from the formalized analysis is that we cannot completely get away from the impact of the masking effect even if we do exhaustive testing. Furthermore, both ignoring the masking effects and regarding multiple faults as one fault is harmful for FCI process.

Based on the insight we proposed a strategy to alleviate this impact. This strategy adopts the divide and conquer framework, i.e., separately handle each fault in SUT. For a particular fault under analysis, we discard the test cases that trigger faults different from the one under analysis and only keep those that either pass or trigger the expected fault. As the test cases discarding manipulation can make FCI approaches lose some information that needed to make them normally work, we will generate additional test cases to compensate for the loss. It is noted that the additional test cases generating criteria vary in different FCI approaches we chose. For example, for the CTA, we will generate more test cases to keep as the same coverage as possible after we deleting some unsatisfied test cases from original covering array. While for the one fact one time approach(OFOT for short), we will repeat generating test case until we find a test case can test the fixed part as well as not trigger unsatisfied fault or until a prior ending criteria is met.

To evaluate the performance of our strategy, we applied our strategy on three FCI approaches, which are CTA [17], OFOT [12], FIC [20] respectively. The subjects we used are several open-source software with the developers' forum in Source-forge net. Through studying their bug reports in the bug tracker system as well as their user's manual guide, we built the testing model which can reproduce the reported bugs with specific test cases. We then applied the traditional FCI approaches and their variation augmented with our strategy to identify the failure-inducing combinations in the subjects respectively. The results of our empirical studies shows that approaches augmented with our strategy identified failure-inducing combinations more accurately than than traditional ones, which indicates that our strategy do assist FCI approaches to behave better under the circumstances that masking effects exist in the SUT.

The main contributions of this paper are:

1. We formally studied the impact on the isolation of the failure-inducing interactions when the SUT contain multiple faults which can mask each other.

2. We proposed a divide and conquer strategy of selecting test cases to alleviate the impact of masking effect.

3. We empirically studies our strategy and find that our approach can get a better result.

```
public float foo(int a, int b, int c, int d){
  //step 1 will cause a exception when b == c
  float x = (float)a / (b - c);

  //step 2 will cause a exception when c < d
  float y = Math.sqrt(c - d);

  return x+y;
  }
```

Figure 1: a toy program with four input parameters

2. MOTIVATION EXAMPLE

This section constructed a small program example for convenience to illustrate the motivation of our approach. Assume we have a method *foo* which has four input parameters : *a, b, c, d.* The types of these four parameters are all integers and the values that they can take are: $v_a = \{7, 11\}, v_b = \{2, 4, 5\}, v_c = \{4, 6\}, v_d = \{3, 5\}$ respectively. The detail code of this method is listed in figure 1.

Inspecting the simple code above, we can find two faults: First, in the step 1 we can get a *ArithmeticException* when b is equal to c, i.e., b = 4 & c = 4, that makes division by zero. Second, another *ArithmeticException* will be triggered in step 2 when c < d, i.e., c = 4 & d = 5, which makes square roots of negative numbers. So the expected MFSs in this example should be (-, 4, 4, -) and (-, -, 4, 5).

Traditional FCI algorithms do not consider the detail of the code. They take black-box testing of this program, i.e., feed inputs to those programs and execute them to observe the result. The basic justification behind those approaches is that the failure-inducing combinations for a particular fault must only appear in those inputs that trigger this fault. As traditional FCI approaches aim at using as small number of inputs as possible to get the same or approximate result as exhaustive testing, so the results derive from a exhaustive testing set must be the best that these FCI approaches can reach. Next we will illustrate how exhaustive testing works on identifying the failure-inducing combinations in the program.

We first generated every possible inputs as listed in the Column "test inputs" of Table 1, and the execution result of are listed in Column "result" of Table 1. In this Column, *PASS* means that the program runs without any exception under the inputs in the same row. *Ex 1* indicates that the program triggered a exception corresponding to the step 1 and *Ex 2* indicates the program triggered a exception corresponding to the step 2. According to data listed in table 1, we can deduce that that (-, 4 , 4, -) must be the failure-inducing combination of Ex 1 as all the inputs triggered Ex 1 contain this schema. Similarly, the combination (-, 2, 4, 5) and (-, 3, 4, 5) must be the failure-inducing combinations of the Ex 2. We listed this three combinations and its corresponding exception in Table 2.

Note that we didn't get the expected result with traditional FCI approaches in this case. The failure-inducing

Table 1: test inputs and their corresponding result

id	test inputs	result	id	test inputs	result
1	(7, 2, 4, 3)	PASS	13	(11, 2, 4, 3)	PASS
2	(7, 2, 4, 5)	Ex 2	14	(11, 2, 4, 5)	Ex 2
3	(7, 2, 6, 3)	PASS	15	(11, 2, 6, 3)	PASS
4	(7, 2, 6, 5)	PASS	16	(11, 2, 6, 5)	PASS
5	(7, 4, 4, 3)	Ex 1	17	(11, 4, 4, 3)	Ex 1
6	(7, 4, 4, 5)	Ex 1	18	(11, 4, 4, 5)	Ex 1
7	(7, 4, 6, 3)	PASS	19	(11, 4, 6, 3)	PASS
8	(7, 4, 6, 5)	PASS	20	(11, 4, 6, 5)	PASS
9	(7, 5, 4, 3)	PASS	21	(11, 5, 4, 3)	PASS
10	(7, 5, 4, 5)	Ex 2	22	(11, 5, 4, 5)	Ex 2
11	(7, 5, 6, 3)	PASS	23	(11, 5, 6, 3)	PASS
12	(7, 5, 6, 5)	PASS	24	(11, 5, 6, 5)	PASS

Table 2: Identified failure-inducing combinations and their corresponding Exception

failure-inducing combinations	Exception
(-, 4, 4, -)	Ex 1
(-, 2, 4, 5)	Ex 2
(-, 3, 4, 5)	Ex 2

combinations we get for Ex 2 are (-,2,4,5) and (-,3,4,5) respectively instead of the expected combination (-,-,4,5). So why we failed getting the (-,-,4,5)? The reason lies in *input 6*: (7,4,4,5) and *input 18*: (11,4,4,5). This two inputs contain the combination (-,-,4,5), but didn't trigger the Ex 2, instead, Ex 1 was triggered.

Now let us get back to the source code of *foo*, we can find that if Ex 1 is triggered, it will stop executing the remaining code and report the exception information. In another word, Ex 1 have a higher faulting level than Ex 2 so that Ex 1 may mask Ex 2. Let us re-exam the combination (-,-,4,5): if we supposed that *input 6* and *input 18* should trigger Ex 2 if they didn't trigger Ex 1, then we can conclude that (-,-,4,5) should be the failure-inducing combination of the Ex 2, which is identical to the expected one.

However, we cannot validate the supposition, i.e., *input 6* and *input 18* should trigger Ex 2 if they didn't trigger Ex 1, unless we have fixed the code that trigger Ex 1 and then re-executed all the test cases. So in practice, when we do not have enough resource to re-execute all the test cases again and again or can only take black-box testing, the more economic and efficient approach to alleviate the masking effect on FCI approaches is desired.

3. FORMAL MODEL

This section presents some definitions and propositions to give a formal model for the FCI problem.

3.1 failure-inducing combinations in CT

Assume that the SUT (software under test) is influenced by n parameters, and each parameter p_i has a_i discrete values from the finite set V_i, i.e., $a_i = |V_i|$ (i = 1,2,..n). Some of the definitions below are originally defined in [13].

Definition 1. A *test case* of the SUT is an array of n values, one for each parameter of the SUT, which is denoted as a n-tuple (v_1, v_2...v_n), where $v_1 \in V_1$, $v_2 \in V_2$ ... $v_n \in V_n$.

In practice, these parameters in the test case can represent many factors, such as input variables, run-time options, building options or various combination of them. We need to execute the SUT under these test cases to ensure the correctness of the behaviour of the software.

We consider the fact that the abnormally executing test cases as a *failure*. It can be a thrown exception, compilation error, assertion failure or constraint violation.

Figuring out the test case as well as the executing result is usually not enough to analyse the source of the bug, especially when there are too many parameters we need to care in this test case. In this circumstance, we need to study some subsets of this test case, so we need the following definition:

Definition 2. For the SUT, the n-tuple (-,v_{n_1},...,v_{n_k},...)is called a k-value *combination* ($0 < k \leq n$) when some k parameters have fixed values and the others can take on their respective allowable values, represented as "-".

In effect a test case itself is a k-value *combination*, when k is equal to n. Furthermore, if a test case contain a *combination*, i.e., every fixed value in this combination is also in this test case, we say this test case *hit* this *combination*, which can be denoted as $k - value\ combination \in T$

Definition 3. let c_l be a l-value combination, c_m be an m-value combination in SUT and $l < m$. If all the fixed parameter values in c_l are also in c_m, then c_m *subsumes* c_l. In this case we can also say that c_l is a *sub-combination* of c_m and c_m is a *parent-combination* of c_l, which can be denoted as $c_l \prec c_m$

For example, in the motivation example section, the 2-value combination (-, 4, 4, -) is a sub-combination of the 3-value combination (-, 4, 4, 5).

Definition 4. If all test cases contain a combination, say c, trigger a particular fault, say F, then we call this combination c the *faulty combination* for F. Additionally, if none sub-combination of c is the *faulty combination* for F, we will call the combination c the *minimal faulty combination* for F.

In fact, *minimal faulty combinations* is identical to the failure-inducing combinations we discussed previously. Figuring it out can eliminate all details that are irrelevant for producing the failure, which can facilitate the debugging effort.

We have the following proposition based on these basic definitions.

PROPOSITION 1. *Let c_m be a m-value combination, the set $P_{c_m}(m+l) = \{c_m + l : c_m \prec c_m + l\}$ be all the m+l value parent combination of c_m. Moreover, we denote all the test cases can* hit *the combination c_m as $T(c_m)$, then we have the following formula:*

$$T(c_m) = \bigcup_{i=1}^{|P_{c_m}(m+l)|} T(c_{(m+l)_i}).$$

The proof of this propositions is obvious, we just take an example to illustrate this. For SUT(2,2,2), the $T((1,-,-))$ = $\{(1,0,0)$,$(1,0,1)$, $(1,1,0)$, $(1,1,1)\}$ = T((1,0,-)) $\bigcup$ T((1,1,-)) $\bigcup$ T((1,-,0)) $\bigcup$ T((1,-,1)). Additional, some test cases of

these parent-combinations may overlapped each other, for example, T((1,0,-)) overlap with T((1,-,1)) at the test case (1,0,1).

By definition, if all the test cases in $T(c_m)$ are failing test cases with the same fault, say, F_m, then combination c_m is the failure-inducing combination of F_m. However, if some test cases in $T(c_m)$ does not trigger F_m, then we can get the following proposition.

PROPOSITION 2. *For a combination c_m, if all the test cases in $T(c_m)$ triggered fault F_m except the set of test cases $T' \subsetneqq T(c_m)$, then the failure-inducing combinations for F_m must be some parent-combinations of c_m, i.e., $\{c_{(m+l)_i}\}$ ($l \in \{1, 2...n-m\}$; $i \in \{1, 2...|P_{c_m}(m+l)|\}$) satisfy the following formula:*

$$\forall t \in T' \ s.t. \ t \notin T(c_{(m+l)_i})$$

PROOF. We proof this proposition by two steps:

First, there must exist some parent-combination c_{m+l} satisfy this formula. Let me divide the $T(c_m)$ to pieces of $T(c_{n_i})$, Each set of $T(c_{n_i})$ just have one element, that is c_{n_i} itself as a test case. Obviously, these set do not overlap with each other. Take an example, $T((1,-,-)) = \{(1,0,0)$ $,(1,0,1), (1,1,0), (1,1,1)\}$ =T((1,0,0)) $\bigcup$ T((1,0,1)) $\bigcup$ T((1,1,0)) $\bigcup$ T((1,1,1)).

As $T' \subsetneqq T(c_m)$, which means that at least one test case in $T(c_m)$ trigger the fault F_m, and it must be one of $T(c_{n_i})$. So at least there exist one combination c_{n_i} which can satisfy this formula.

Next, some satisfied parent-combination c_{m+l} must be the failure-inducing combination of F_m. In fact, when c_{m+l} satisfy this formula, it is the *fault combination* of F_m by definition. And there must be at least one child-combination is not the *fault combination* of F_m, which is c_m. So some parent-combinations must be the failure-inducing combinations of F_m. □

However, if $T' = T(c_m)$, which means none of the test case in $T(c_m)$ trigger the fault F_m then neither the parent of the combination of c_m nor c_m its self is the failure-inducing combination of F_m. In this circumstance, we call combination c_m is *irrelevant* to the failure-inducing combinations of F_m

Based on this proposition, we can easily get the following lemma:

LEMMA 1. *For two set of test cases T_l and T_k, assume that $T_l \subset T_k$. Suppose SUT has a fault F. Now first let all the test cases that trigger F to be the set T_k, the failure-inducing combinations for F is the set $C_k = \{c_k\}$. Then let all the test cases that trigger F to be T_l, which we get another set failure-inducing combinations $C_l = \{c_1\}$. Then we can deduce that C_l and C_k satisfy at least one of the following properties:*

1) *$\exists c_k \in C_k$ and $\exists c_l \in C_l$ s.t. c_l subsumes c_k*
2) *$\exists c_k \in C_k$ s.t. c_k is irrelevant to C_l*

We illustrate this two scenarios in table 3. There are two parts in this table, in which the left part shows the case that the failure-inducing combinations satisfy the first properties(both (0,0,-) and (0,1,0) subsume (0,-,-)). While the right part give an example to the second properties, as (1,-,-) is irrelevant to any one of (0,0,-) and (0,1,0).

Next, let move to formal analysis of the masking effects.

Table 3: example of two scenarios

T_l	T_k
(0, 0, 0)	(0, 0, 0)
(0, 0, 1)	(0, 0, 1)
(0, 1, 0)	(0, 1, 0)
	(0, 1, 1)
C_l	C_k
(0, 0, -)	(0, -, -)
(0, 1, 0)	

T_l	T_k
(0, 0, 0)	(0, 0, 0)
(0, 0, 1)	(0, 0, 1)
(0, 1, 0)	(0, 1, 0)
	(1, 0, 0)
	(1, 0, 1)
	(1, 1, 0)
	(1, 1, 1)
C_l	C_k
(0, 0, -)	(0, 0, -)
(0, 1, 0)	(0, 1, 0)
	(1, -, -)

3.2 masking effect

Definition 5. A *masking effect* is the effect that a test case t hit a failure-inducing combination for F_a, say c , but doesn't trigger fault F_a as expected because this test case triggered other fault which is different from F_a, say, F_b.

based on this definition, we can learn that, to get the accurate result, we should take consider these test cases do not trigger some specific fault because it trigger other faults. We call these test cases $T_{mask(F_m)}$ for a particular fault F_m masked by these test cases.

So in fact, the failure-inducing combination for F_m should be the combination which have the test cases $T_{F_m} \bigcup T_{mask(F_m)}$. We call the failure-inducing combinations with considering the masking effects the *perfect failure-inducing combinations*. However, this is not possible in practice, unless we fix some bugs in the SUT and re-execute the test cases to figure out $T_{mask(F_m)}$.

For traditional FCI approaches, there are two kinds of approaches, let me see what they like:

3.2.1 regard as one fault

This is the most common dealing approach, they don't distinguish the faults, i.e., regard all the types of faults as one fault–*failure*, others as the *pass*.

So they will find the c_m for $T_F = \bigcup_{i=1}^{L} F_i$, L is the number of all the faults in the SUT. Obviously, $T_{F_m} \bigcup T_{mask(F_m)} \subset T_F$. So in this circumstance, by the proposed lemma, the failure-inducing combinations we get will have some combinations irrelevant to the perfect failure-inducing combinations , or some combinations just be the child-combinations of some of the perfect failure-inducing combinations.

3.2.2 distinguish fault by buggy information

Distinguish the faults by the exception traces or error code can help make FCI approaches focus on particular fault. Ylimaz in [18] propose a *multiple-class* failure characterize method instead of *ternary-class* approaches to make the charactering approach more accurately. Besides, other approaches can also be easily extended to apply on SUT with multiple faults. We call this approach *distinguish FCI* approaches.

Without no information of the masking effect, distinguish FCI approaches can only identify the c_m for T_{F_m}. In this case, we can learn that $T_{F_m} \bigcup T_{mask(F_m)} \supset T_{F_m}$, consequently, the failure-inducing combinations we get in this case will have some combinations be the parent-combination of

some perfect combinations or completely miss some failure-inducing combination.

Note that our discuss is based on the SUT is a deterministic software, i.e., the random failing information of test case will be ignored. The non-deterministic problem will complex our test scenario, which will not be discussed in this paper.

4. TEST CASE REPLACING STRATEGY

Previous section formally studied the case that under the masking effects, neither regarding as one fault approach nor distinguishing faults approach is not accurate . The main reason is that we cannot figure out the $T(mask_{F_m})$. As $T(mask_{F_m}) \subset \bigcup_{i=1\&j\neq m}^{L} T_{F_i}$. So to reduce the influence of $T(mask_{F_m})$, we need to reduce the number of test cases that trigger other faults as much as possible.

In the exhaustive testing, as all the test cases will be used to identify the failure-inducing combinations, so there is no room left to improve the accurateness. However, when just choose part of the whole test cases, which is practical and sometimes the only solution for large-scale SUT, we can adjust the test cases we need to use by choosing proper ones so that we can limit the number of $T(mask_{F_m})$ to be as less as possible.

4.1 Replace test configuration that trigger unexpected fault

The basic idea is to discard the test configurations that trigger other faults and generate other test configurations to represent them. These regenerate test configurations should either pass the executing or trigger F_i. The replacement must fulfil some criteria, such as for CTA, the input for this algorithm is a covering array, and if we replace some test configuration in it, we should ensure that the covering rate is not changed.

Commonly, when we replace the test configuration that trigger unexpected fault with a new test configuration, we should keep some part in the original test configuration, we call this part as *fixed part*, and mutant other part with different values from the original one. For example, if a test configuration (1,1,1,1) triggered Err 2, which is not the expected Err 1, and the fixed part is (-,-,1,1), then we may regenerate a test configuration (0,0,1,1) which will pass or trigger Err 1.

The *fixed part* can be the schemas that only appear in the original test configuration which other test configurations in the covering array did not contain(for the algorithm take covering array as the input: CTA), or the factors that should not be changed in the OFOT algorithms, or the part that should not be mutant of the test configuration in the last iteration(FIC_BS).

The process of replace a test configuration with a new one with keeping some fixed part is depicted in Algorithm 1:

The inputs for this algorithm consists of a test configuration which trigger an unexpected fault – $t_{original}$, the fixed part which we want to keep from the original test configuration – s_{fixed} , the fault type which we current focus – F_i. And the values sets that each option can take from respectively. The output of this algorithm is a test configuration t_{new} which either trigger the expected F_i or passed.

In fact, this algorithm is a big loop(line 1 - 14) which be parted into two parts:

Algorithm 1 replace test configurations that trigger unexpected fault

Input: $t_{original}$ ▷ original test configurations
F_i ▷ fault type
s_{fixed} ▷ fixed part
$Param$ ▷ values that each option can take
Output: t_{new} ▷ the regenerate test configuration

```
1: while not MeetEndCriteria() do
2:     s_mutant ← t_original − s_fixed
3:     for each opt ∈ s_mutant do
4:         i = getIndex(Param, opt)
5:         opt ← opt' s.t. o ∈ Param[i] and opt'! = opt
6:     end for
7:     t_new ← s_fixed ⋃ s_mutant
8:     result ← execute(t_new)
9:     if result == PASS or result == F_i then
10:        return t_new
11:    else
12:        continue
13:    end if
14: end while
15: return null
```

The first part(line 2 - line 7): generate a new test configuration which is different from the original one. This test configuration will keep the fixed part (line 7), and just mutant these factors are not in the fixed part(line 2). The mutant for each factor is just choose one legal value that is different from the original one(line 3 - 6). The choosing process is just by random and the generated test configuration must be different each iteration(can be implemented by hashing method).

Second part is to validate whether this newly generated test configuration matches our expect(line 8 - lone 13). First we will execute the SUT under the newly generated test configuration(line 8), and then check the executed result, either passed or trigger the expected fault – F_i will match our expect.(line 9) If so we will directly return this test configuration(line 10). Otherwise, we will repeat the process(generate newly test configuration and check)(line 11 -12).

It is noted that the loop have another end exit besides we have find a expected test configuration(line 10), which is when function *MeetEndCriteria()* return a true value(line 1). We didn't explicitly show what the function *MeetEndCriteria()* is like, because this is depending the computing resource you own and the how accurate you want to the identifying result to be. In detail, if you want to your identify process be more accurate and you have enough computing resource, you can try much times to get the expected test configuration, otherwise, you may just try a relatively small times to get the expected test configuration.

In this paper, we just set 3 as the biggest repeat times for function. When it ended with *MeetEndCriteria()* is true, we will return null(line 15), which means we cannot find a expected test configuration.

4.2 Examples of applying the strategy

Next we will take two MFS identifying algorithms as the subject to see how our approach works on them.

4.2.1 OFOT examples

Assume we have test a system with four parameters, each

Table 4: Identifying MFS using OFOT with our approach

original test configuration	fault info
0 0 0 0	Err 1
gen test configurations	**result**
1 0 0 0	Err 1
~~0 1 0 0~~	~~Err 2~~
0 2 0 0	Err 1
0 0 1 0	Pass
0 0 0 1	Pass
original identified:	**identified with replacement**
(- 0 0 0)	(- - 0 0)

has three options. And we take the test configuration (0 0 0 0) we find the system encounter a failure called "Err 1". Next we will take the MFS identifying algorithms – OFOT with the help of our approach to identify the MFS for the "Err 1". The process is listed in table 4. In this table, The test configuration which are labeled with a deleted line represent the original test configuration generated by OFOT, and it will be replaced by the regenerated test configuration which are labeled with a wave line under it.

From this table, we can find the algorithm mutant one factor to take the different value from the original test configuration on time. Originally if the test configuration encounter the different condition with the Err 1, OFOT will make a judgement that the MFS was broken, in another word, if we change one factor and it does not trigger the same fault, we will label them as one failure-inducing factor, after we changed all the elements, we will get the failure-inducing schemas. For this case, as when we change the second factor , third factor and the fourth factor, it doesn't trigger the Err 1 (for second factor, it trigger Err 2 and for the third and fourth, it passed). So if we do not regenerate the test configuration (0 2 0 0), we will get the MFS – (- 0 0 0) for the err 1(which are also labeled with a delete line).

However, if we replace the test configuration (0 1 0 0) with (0 2 0 0) which triggered err 1 (in this case, the fixed part of the test configuration is (0, - - -)), we will find that only when we change the third and fourth factor will we broke the MFS for err 1, so with our approach, we will find the MFS for err 1 should be (- - 0 0) (labeled with a wave line under it).

4.2.2 CTA examples

CTA uses the covering array as its inputs and then use classification tree algorithm to characterize the MFS. For this algorithm, we still assume the SUT has 4 parameters and each one has 3 values. Then we will initial a 2-way covering array as the input for CTA algorithm which is listed in table 5. The delete line and wave line have the same meaning as the OFOT example.

Let's look at the original test configuration (0 1 1 1), it triggered the err 2 which is not as our expected, so we will replace it with other test configurations. A prerequisite for this replacement is that we should not decrease the covering rate. As the original test configuration contain the 2-degree

Table 5: Identifying MFS using CTA with our approach

covering arrays	fault info
0 0 0 0	Err 1
~~0 1 1 1~~	~~Err 2~~
1 1 1 1	Err 1
0 1 1 0	Pass
0 0 0 1	Pass
0 2 2 2	Pass
1 0 1 2	Pass
1 1 2 0	Err 1
1 2 0 1	Err 1
2 0 2 1	Pass
2 1 0 2	Pass
2 2 1 0	Pass
original identified:	**identified with replacement**
(0 0 - -) : err1	(0 0 0 -):err1
(1 1 - -) : err1	(1 1 - -):err1
(1 2 - -) : err1	(1 2 - -):err1
(0 1 - -) : err2	

schema(0 1 - -),(0 - 1 -),(0 - - 1),(-,1,1,-),(-,1,-,1),(-,-,1,1). We make them as fixed part that the regenerate test configuration must keep, as we cannot use one test configuration to cover all the six fixed part, so instead, we use three additional test configurations (1 1 1 1), (0, 1 ,1 ,0) and (0,0,0,1) to cover them, note that this three test configurations either trigger err 1 or pass. If they don't match this condition, we will try other test configurations to replace the original test configuration.

We use multiple-class CTA as our subject, Ylimaz in the paper has claimed that mutiple-class CTA perferom better than terrncy-class CTA.

5. EMPIRICAL STUDIES

We conducted several case studies to address the following questions:

Q1: Do masking effects existed in real software when it contain multiple faults?

Q2: How much do traditional approaches suffer from these real masking effects?

Q3: Can our approach do better than traditional approaches when facing these masking effects?

Specifically, section 6.1 survey several open-source software to gain a insight of the state of the existence of multiple faults and their masking effects. Section 6.2 directly applied three MFS-identifying programs on the surveyed software and analysis their results. Section 6.3 apply our approach on the software and a comparison with traditional approaches will be discussed. Section 6.4 discuss the threats to validity of our empirical studies.

5.1 study 1: multiple faults and masking effects in practice

Table 6: Software under survey

software	versions	LOC	procedur/classes
JFlex [3]	1.4.1	10040	58
	1.4.2	10745	61
HSQLDB [4]	2.0rc8	139425	495
	2.2.5	156066	508
	2.2.9	162784	525

In the first study, we surveyed two open-source software to gain an insight of the existence of multiple faults and their effects. The software under study are: HSQLDB and JFlex, in which the former is a database management software written in pure java and the later one is a lexical analyser generator for Java. Each of them contain different versions. All the two subjects are highly configurable so that the options and their combination can influence their behaviour. Additionally, they all have developers' community so that we can easily get the real bugs reported in the bug tracker forum. Table 6 lists the program, the number of versions we surveyed, number of lines of uncommented code and number of classes in the project.

5.1.1 study setup

We first looked through the bug tracker forum of each software and scratched some bugs which are caused by the options combination to study later. We then classify these bugs according to the version they belong to. For each bug, we will derive its failure-inducing combinations by analysing the bug description report and its attached test file which can reproduce the bug. For example, through analysing the source code of the test file of bug#123213 for HSQLDB, we found the failure-inducing combinations for this bug is: (*preparestatement*, *placeHolder*, *Long string*), this three factors together form the condition on which the bug will be triggered. These analysed result will be regarded as the "perfect combinations" later.

We further selected pairs of bugs belong to the same version and merged their test file. This merging manipulation vary with the pair of bugs we selected. As for bug#123213 and bug#223012 of HSQLDB, the original test file can be simply described as figure 1:, and we merged them together into a file like figure 2, we can easily find that it use a test scenario to merge the two test file. For each pair of bugs, we have posted the source code of the merging file on website.

6. MASKING FAULTS STATISTIC

Next we built the input model which consist of the options related to the perfect combinations and additional noise options. The detailed model information is listed in appendix. We then generated the exhaustive test suite consist of all the possible combinations of these options under which we executed the merged test file. We record the output of each test case to observe whether there are test cases contain 'perfect combination' but do not produce the corresponding bug.

6.0.2 result and discuss

[3]http://sourceforge.net/projects/jflex/files/jflex
[4]http://sourceforge.net/projects/hsqldb/files/hsqldb/

Table 7: real faults detailed

software	version	faults ID pair	Web site
HSQLDB	2.0rc8	#981 & #1005	http://sourceforge.net
-	2.2.6	#1173 & #1179	/p/hsqldb/bugs
-	2.2.9	#1286 & #1280	-
JFlex	1.4.1	#87 & #80	http://sourceforge.net
-	1.4.2	#98 & #93	/p/jflex/bugs

Table 8: input model of HSQLDB

Common options	**values**
sql.enforce_strict_size	true, false
sql.enforce_names	true, false
sql.enforce_refs	true, false
Server Type	server, webserver, inprocess
existed form	mem, file
resultSetTypes	forwad, insensitive, sensitive
resultSetConcurrencys	read_only, updatable
resultSetHoldabilitys	hold, close
StatementType	statement, prepared

versions	**specific options**	**values**
2.0rc8	more	true, false
	placeHolder	true, false
	cursorAction	next,previous,first,last
2.2.5	multiple	single, multiple
	placeHolder	true, false
2.2.9	duplicate	true, false
	default_commit	true, false

versions	**Config space**
2.0rc8	$2^9 \times 3^2 \times 4^1$
2.2.5	$2^9 \times 3^2$
2.2.9	$2^9 \times 3^2$

Table 9: input model of JFlex

Common options	values
public	true, false
apiprivate	true, false
cup	true, false
caseless	true, false
char	true, false
line	true, false
column	true, false
notunix	true, false
yyeof	true, false
generation	switch, table, pack
charset	default, 7bit, 8bit, 16bit

versions	specific options	values
1.4.1	hasReturn	true, false
	normal	true, false
1.4.2	lookAhead	single, multiple
	type	true, false
	standalone	true, false

versions	Config space
1.4.1	$2^{11} \times 3^1 \times 4^1$
1.4.2	$2^{12} \times 3^1 \times 4^1$

Table 10: number of faults and their masking effect condition

software	versions	Faults ID	all tests	Num masking
HSQLDB	2cr8	#2000 & #3000	2000	1000

Table 10 lists the results of our survey. Column "Faults ID" indicate the IDs of two bugs we collected from the bug tracker system, column "all tests" give the total number of test cases we executed and column "num of masking" indicate the number of test cases which trigger the masking effect.

We observed that for each pair of bugs we listed in the table, we can always find some test cases, although contain perfect combinations, did not trigger expected bug, i.e., the expected bug was masked. Specifically, there are 1000 out of 2000 (20%) test cases for HSQLDB triggered the masking effect.

So the answer to Q1 is that in practice, when SUT have multiple faults, there is a good chance that masking effect can be triggered by some test cases.

6.1 study 2:performance of traditional algorithms

In the second study, we want to learn how badly the masking effect impact on the traditional approaches. To conduct this study, we need to apply the traditional failure-inducing identifying algorithms on the SUT we collected in the first study and compare them with the perfect combinations.

6.1.1 study setup

The traditional approaches we selected are: OFOT, FIC and CTA. To make the first two approaches (OFOT and FIC) work, we need to feed them with a failing test case. And for CTA, however, it need to work with a covering array and the executing result for each test case in the covering array. As the input for the traditional approaches are different, we designed two different setups, one for OFOT and FIC, while the other for CTA.

For OFOT and FIC, our set up is as follows:

We first selected failing test cases from the exhaustive executed test cases, and for each test case, we applied OFOT and FIC to isolate the failure-inducing combinations in this test case. As the subject under test has multiple faults, the strategy we chose for this two the traditional approaches is *distinguish fault* in section 4.2, i.e., distinguish fault and ignore the masking effect happened among them. At last we collected all the failure-inducing combinations they got and deleted those overlapped ones.

For CTA, our setup is as follows:

We first utilize augment simulating algorithms(ASA) to generate 2-way covering arrays. And for each test case in the covering array, the executing result of them can be fetched from the exhaustive set we have got in the first study. Then we fed CTA approach with the covering array along with the corresponding executing results. We then collected the failure-inducing combinations after running CTA. As different covering array may affect the result of CTA approach, we repeated using ASA to generate 2-way covering array for 30 times and applied CTA for each of them. It is noted that ASA algorithm is a heuristic approach, which containing some random factors, so the 30 covering arrays are different from each other.

After we have collected the result got by each algorithm for each version software, we next need to compare the result with the perfect combinations to quantify the extent to which traditional approaches suffers from masking effect. To make this comparison, we first introduce the following notation:

Assume we get two schema S_A,S_B, the notation$S_A \bigcap S_B$ indicate the same elements between S_A and S_B. For example $(-1\ 2-3)\bigcap(-2\ 2-3) = \{(--2--), (----3)\}$.

Then the similarity between schemas is defined as the followed notation:

$$Similarity(S_A, S_B) = \frac{|S_A \bigcap S_B|}{\max(Degree(S_A), Degree(S_B))}$$

.

In this formula, the numerator indicate the number of same elements in S_A and S_B. It is obviously, the bigger the value is the more similar two schemas are. The denominator give us this information: if the the number of same elements is fixed, the bigger the degree of the schema, the more noise we will encounter to find the fault source. In a word, the bigger the formula can get the better the similarity is between the two schema.

For the set of schemasSet_A and Set_B, the similarity definition is :

$$Similar(Set_A, Set_B) = \frac{\sum_{s_i \in Set_A} \max_{s_j \in Set_B} S(s_i, s_j)}{|Set_A|}$$

.

It is obviously the more similarity between the set of failure-inducing combinations identified by the algorithms with these perfect combinations the less impacts do these al-

Table 11: traditional result

Software	version	num diff			similarity		
-	-	OFOT	FIC	CTA	OFOT	FIC	CTA
HSQLDB	3.14	3	4	4	0.65	0.64	0.5

Table 12: our approach result

Software	ver	num diff			similarity	
-	-	OFOT	FIC	CTA	OFOT	FIC
HSQLDB	3.14	3 (+4)	4 (+4)	4 (-2)	0.65 (+0.01)	0.64(-0.05)

gorithms suffered from multiple faults as well as their masking effects.

Besides the similarity metric, another metric also need to be considered: the discrepancy between the number of identified failure-inducing combinations with these perfect combinations. If this discrepancy is too big, even though we get a good similarity metric value, we also get too many noise which make our result inaccuracy.

Particularly, if the value of discrepancy number metric is 0 and the value of similarity metric is 1, it means that multiple faults as well as their masking effect have no impacts on the algorithms, which is usually not feasible in practice.

6.1.2 result and discuss

Table 11 depicts the result of the second case study. Column "Num Diff" indicates the discrepancy of the number of MFS between the ones got through traditional approaches with the perfect combinations. Column "Similarity" presents the similarity between this two set of combinations. Additionally, the results of different algorithms can be distinguished by the titles of sub-columns, which are "FIC", "OFOT", "CTA", indicating corresponding algorithms respectively. It is noted that for column "CTA", the vale we set in each cell are the average value of the result we collected from 30 repeated experiments' result for a particular subject.

From this table, we can easily find that Traditional MFS identifying approaches do suffer from the multiple faults and their masking effect. For example, for the Grep, version 3.14, all these three algorithm will lost some information. Additionally, the extent to which the impact has affected vary from algorithms. We can find that for HSQLDB with version 3.14, the similarity of CTA was just at 4 while FIC is 3. We list these points in figure 3. From this figure, we can learn that.

So the answer for Q2 is: traditional algorithm do suffer from the multiple faults and their masking effect although the extent vary in different algorithms.

6.2 study 3:performance of our approach

The last case study aims to observe the performance of our approach and compare it with the result got by the traditional approaches. Specifically, we augmented the three traditional approaches using the method described in section 5, and then we applied these augmented approaches on identifying the failure-inducing combinations in the prepared subjects.

6.2.1 study setup

The setup of this case study is almost the same as the second case study. The difference is that the algorithms we choose are three augment ones. Additionally, comparisons between augment approaches with three traditional ones will be quantified.

6.2.2 result and analysis

Table 12 presents the result of the last case study. The form of this table is similar to the second study. We just added some information in the parentheses attached the value in each cell. This information display the discrepancy between traditional ones. Notation '+' means promotions against traditional ones, while '-' indicates decrease. The value in parentheses has been normalized.

We observe that, for most cases, augment approaches got promotions against traditional ones. In fact, there are 10 out of 14 cases that augment ones outperform traditional ones at the metric of "similarity", and 9 out of 14 cases that augment ones are better than traditional ones at the metric of "num diff". Additional, this promotion is distinct. As we can see, the best promotion for similarity is 0.9 and the best promotion for num diff is 0.7. Furthermore, the average performance promotion for similarity reached '+ 0.6' and this value reached '+ 0.5' for the num diff metric.

So the answer for Q3 is: our approach do get better performance at identifying failure-inducing combinations when facing masking effect between multiple faults, to which the extent is distinct.

6.3 threats to validity

There are several threats to validity for these empirical studies. First, we have only surveyed five open-source software, four of which are medium-sized and one is large-sized. This may impact the generality of our observations. Although we believe it is quite possible a common phenomenon in most software that contain multiple faults which can mask each other, we need to investigate more software to support our conjecture. The second threat comes from the input model we built. As we focused on the options related to the perfect combinations and only augmented it with some noise options, there is a chance we will get different result if we choose other noise options. More different options needed to be opted to see whether our result is common or just appeared in some particular input model. The third threats is that we just observed three MFS identifying algorithms, further works needed to exam more MFS identifying algorithms to get a more general result.

7. RELATED WORKS

Nie's approach in [12] first separates the faulty possible tuples and healthy-possible tuples into two sets. Subsequently, by changing a parameter value at a time of the original test configuration, this approach generates extra test configurations. After executing the configurations, the approach converges by reducing the number of tuples in the faulty-possible sets.

Delta debugging [19] proposed by Zeller is an adaptive divide-and-conquer approach to locate interaction fault. It is very efficient and has been applied to real software environment. Zhang et al. [20] also proposed a similar approach that can identify the failure-inducing combinations that has no overlapped part efficiently.

Colbourn and McClary [5] proposed a non-adaptive method. Their approach extends the covering array to the locating array to detect and locate interaction faults. C. Martinez [10, 11] proposed two adaptive algorithms. The first one needs safe value as their assumption and the second one remove the assumption when the number of values of each parameter is equal to 2. Their algorithms focus on identifying the faulty tuples that have no more than 2 parameters.

Ghandehari.etc [8] defines the suspiciousness of tuple and suspiciousness of the environment of a tuple. Based on this, they rank the possible tuples and generate the test configurations. Although their approach imposes minimal assumption, it does not ensure that the tuples ranked in the top are the faulty tuples.

Yilmaz [17] proposed a machine learning method to identify inducing combinations from a combinatorial testing set. They construct a classified tree to analyze the covering arrays and detect potential faulty combinations. Beside this, FouchÃl' [7] and Shakya [15] made some improvements in identifying failure-inducing combinations based on Yilmaz's work.

Our previous work [14] have proposed an approach that utilize the tuple relationship tree to isolate the failure-inducing combinations in a failing test case. One novelty of this approach is that it can identify the overlapped faulty combinations. This work also alleviates the problem of introducing newly failure-inducing combinations in additional test cases.

Besides works that aims at identifying the failure-inducing combinations in test cases, there are some work focus on working around the masking effects:

With having known masking effects in prior, Cohen [4] use a SAT solver to avoid these masking effects in test cases generating process. Additional constraints impacts in CT were studied in works like [1, 3, 2, 9, 16]. These approaches use some rules or to avoid these invalidated test cases to improve the efficiency when examine the test cases.

Dumlu and Ylimaz in [6] proposed a feedback-driven approach to work around the masking effects. In specific, it first use CTA classify the possible failure-inducing combinations and then eliminate them and generate new test cases to detect possible masked interaction in the next iteration. They further extended their work in [18], in which they proposed a multiple-class CTA approach to distinguish faults in SUT. In addition, they empirically studied the impacts on both ternary-class and multiple-class CTA approaches.

Our work differs from these ones mainly in the fact that we formally studied the masking effects on FCI approaches and further proposed a divide-and-conquer strategy to alleviate this impact.

8. CONCLUSIONS

Masking effects of multiple faults in SUT can bias the result of traditional failure-inducing combinations identifying approaches. In this paper, we formalized the process of identifying failure-inducing combinations under the circumstance that masking effects exist in SUT and try to understand how do this impacts brought by masking effect. Furthermore, we have presented a divide and conquer strategy to assist traditional FCI approaches to alleviate this impact.

In the empirically studies, we extended three FCI approaches with our strategy. The comparison between this three traditional approaches with their variation is conducted on on several open-source software. The results shows that our strategy do assist traditional FCI approaches get a better performance when facing masking effects in SUT.

As a future work, we need to do more empirical studies to make our conclusion more general. Our current experimental subjects are several middle-sized software, we would like to extend our approach into more complicated and large-scaled testing scenarios. Another promising work in the future is to combine white-box testing technique to make the FCI approaches get more accurate results when handling masking effects. We believe that figuring out the faulting levels of different bugs through white-box testing technique is helpful to reduce misjudgements in the failure-inducing combinations identifying process.

9. REFERENCES

[1] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, 2006.

[2] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In *Tests and proofs*, pages 66–83. Springer, 2008.

[3] B. Chen, J. Yan, and J. Zhang. Combinatorial testing with shielding parameters. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 280–289. IEEE, 2010.

[4] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Transactions on*, 34(5):633–650, 2008.

[5] C. J. Colbourn and D. W. McClary. Locating and detecting arrays for interaction faults. *Journal of combinatorial optimization*, 15(1):17–48, 2008.

[6] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 243–253. ACM, 2011.

[7] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 177–188. ACM, 2009.

[8] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. Identifying failure-inducing combinations in a combinatorial test set. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 370–379. IEEE, 2012.

[9] M. Grindal, J. Offutt, and J. Mellin. Handling constraints in the input space when using combination strategies for software testing. 2006.

[10] C. Martínez, L. Moura, D. Panario, and B. Stevens. Algorithms to locate errors using covering arrays. In *LATIN 2008: Theoretical Informatics*, pages 504–519. Springer, 2008.

[11] C. Martínez, L. Moura, D. Panario, and B. Stevens. Locating errors using elas, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics*, 23(4):1776–1799, 2009.

[12] C. Nie and H. Leung. The minimal failure-causing

schema of combinatorial testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):15, 2011.

[13] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[14] X. Niu, C. Nie, Y. Lei, and A. T. Chan. Identifying failure-inducing combinations using tuple relationship. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 271–280. IEEE, 2013.

[15] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn. Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 620–623. IEEE, 2012.

[16] C. Yilmaz. Test case-aware combinatorial interaction testing. *Software Engineering, IEEE Transactions on*, 39(5):684–706, 2013.

[17] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.

[18] C. Yilmaz, E. Dumlu, M. Cohen, and A. Porter. Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. 2013.

[19] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[20] Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 331–341. ACM, 2011.