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. Our approach first prune these test cases that may trigger masking effect and then 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 as well as the masking effects 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*

*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)

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 [25], generating one test case one time [18], ranking suspicious interactions based on prior rules[12], using graphic-based deduction [16] and so on. These approaches can be partitioned into two categories [7]: *adaptive*–tests cases are chosen based on the outcomes of the executions of prior tests [18, 12, 20, 28, 21, 23, 15]or *nonadaptive*–test cases are chosen independent and can be executed parallel [25, 7, 16, 17, 9].

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 [8, 26] 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 incom-

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

patibility between option '-c' and '-o'. When we put this two 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 [8], 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 [26] further empirically studied the impacts on the Failure-inducing Combinations Identifying approach (FCI approach for short): Classification Tree Approach(CTA for short)[25], 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 formal 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 same 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 should satisfy some criteria. For example, 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 [25], OFOT [18], FIC_BS [28] 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 can identify 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 in Fig 1, 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 failure-inducing combinations in this example should be (-, 4, 4, -) and (-, -, 4, 5).

Traditional FCI algorithms do not consider the detail of the code. They use black-box testing to test 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 derived 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.

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

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 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 in 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 fault level than Ex 2 so that Ex 1 may mask Ex 2. Let us re-examine 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 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 [19].

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 with 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 *fault*. It can be a thrown exception, compilation error, assertion failure or constraint violation. When faults is triggered by some test cases, what is desired is to figure out the cause of these faults, which, some subsets of this test case should be analysed.

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 = n. Furthermore, if a test case contain a *combination*, i.e., every fixed value in the combination is in this test case, we say this test case *hit* the *combination*.

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), that is, (-,4,4,-) $\prec$ (-,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 (It is also called Minimal failure-causing schema(MFS) in [18]).

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

Let c_m be a m-value combination, we denote all the test cases can *hit* the combination c_m as $T(c_m)$. Further, for the test case t, let $\mathcal{I}(t)$ to denote all the combinations that are hit by t, and for the set of test cases T, we let $\mathcal{I}(T) = \bigcup_{t \in T} \mathcal{I}(t)$. Then we have the following propositions.

PROPOSITION 1. *if $c_l \prec c_k$, then $T(c_k) \subset T(c_l)$*

PROOF. Suppose $\forall t \in T(c_k)$, we have that t hits c_k. Then as $c_l \prec c_k$, so t must also hit c_l, as all the element in c_l must in c_k, which also in the test case t. Therefore we get $t \in T(c_l)$. Thus $t \in T(c_k)$ implies $t \in T(c_l)$, so it follows that $T(c_k) \subset T(c_l)$. $\square$

PROPOSITION 2. *For any set T of test cases of a SUT, we can always get a set of minimal combinations $\mathcal{C}(T) = \{c\}$, ($\nexists c', c \in \mathcal{C}(T), s.t. c' \prec c$), such that,*

$$T = \bigcup_{c \in \mathcal{C}(T)} T(c)$$

PROOF. We prove by producing this set of combinations. We denote the exhaustive test cases for SUT as T^*. And let $T^* \backslash T$ be the test cases that in T^* but not in T. It is obviously $\forall t \in T$, we can always find at least one combination $c \in \mathcal{I}(t)$, such that $c \notin \mathcal{I}(T^* \backslash T)$. Specifically, at least the test case t itself as combination holds.

Then we collect all the satisfied combinations($c \in \mathcal{I}(t)$ and $c \notin \mathcal{I}(T^* \backslash T)$) in each test case t of T, which can be denoted as: $S(T) = \{\mathcal{I}(T) - \mathcal{I}(T^* \backslash T)\}$.

For the set $S(T)$, we can have $\bigcup_{c \in S(T)} T(c) = T$. This is because first, for $\forall t \in T(c), (T(c) \subset \bigcup_{c \in S(T)} T(c))$. it must have $t \in T$, as if not so, then $t \in T^* \backslash T$, which contradict with the definition of $S(T)$. So $t \in T$. Hence, $\bigcup_{c \in S(T)} T(c) \subset T$.

Then second, for any test case t in T, as we have learned at least find one c in $\mathcal{I}(t)$, such that c in $S(T)$. Then for $T(c) \subset \bigcup_{c \in S(T)} T(c)$ and $t \in T(c)$, so $t \in \bigcup_{c \in S(T)} T(c)$, therefore, $T \subset \bigcup_{c \in S(T)} T(c)$.

Since, $\bigcup_{c \in S(T)} T(c) \subset T$ and $T \subset \bigcup_{c \in S(T)} T(c)$, so it follows $\bigcup_{c \in S(T)} T(c) = T$.

Then we denote the minimal combinations of $S(T)$ as $M(S(T)) = \{c | c \in S(T) \ and \ \nexists c' \prec c, s.t., c' \in S(T)\}$. For this set, we can still have $\bigcup_{c \in M(S(T))} T(c) = T$. We also prove this by two steps, first and obviously, $\bigcup_{c \in M(S(T))} T(c) \subset \bigcup_{c \in S(T)} T(c)$. Then we just need to proof that $\bigcup_{c \in S(T)} T(c) \subset \bigcup_{c \in M(S(T))} T(c)$.

In fact by definition of $M(S(T))$, for $\forall c' \in S(T) \backslash M(S(T))$, we can have some $c \in M(S(T))$, such that $c \prec c'$. According to the Proposition 1, $T(c') \subset T(c)$. So for any test case $t \in \bigcup_{c \in S(T)} T(c)$, as we have either $\exists c' \in S(T) \backslash M(S(T)), s.t., t \in T(c')$ or $\exists c \in M(S(T)), s.t., t \in T(c)$. Both cases can deduce $t \in \bigcup_{c \in M(S(T))} T(c)$. Therefore, $\bigcup_{c \in S(T)} T(c) \subset \bigcup_{c \in M(S(T))} T(c)$

At last, $M(S(T))$ is the set of combinations that holds this proposition. □

. In fact, for all the test cases for fault F_m which denoted as T_{F_m}, $\mathcal{C}(T_{F_m})$ is the set of failure-inducing combinations of F_m. And obviously $\mathcal{C}(T(c_m)) = c_m$.

From the construction process of $\mathcal{C}(T)$, one observation is that the combinations in $S(T)$ either be the one in $\mathcal{C}(T)$, either be the parent combination of one of them. Then we can have the following proposition.

PROPOSITION 3. *For any $T(c) \subset T$, then it must be that $c \in S(T)$.*

PROOF. In fact, $c = \mathcal{C}(T(c))$, and it has $\mathcal{C}(T(c)) \subset S(T(c))$. Then as $T(c) \subset T$, so it follows $S(T(c)) \subset S(T)$ by definition. So we can have $\mathcal{C}(T(c)) \subset S(T)$ and hence $c \in S(T)$. □

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 $T_l \subset T_k$. Then we have*

$$\forall c_l \in \mathcal{C}(T_l) \, either \ c_l \in \mathcal{C}(T_k) \ or \exists c_k \in \mathcal{C}(T_k), s.t., c_k \prec c_l.$$

PROOF. Obviously for $\forall c_l \in \mathcal{C}(T_l)$ we can get $T(c_l) \subset T_l \subset T_k$. According to the proposition 3, we can that $c_l \in S(T_k)$. So this lemma holds as the combinations in $S(T_k)$ either is also in $\mathcal{C}(T_k)$, or must be the parent of some combination in $\mathcal{C}(T_k)$. □

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)
$\mathcal{C}(T_l)$	$\mathcal{C}(T_k)$
(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)
$\mathcal{C}(T_l)$	$\mathcal{C}(T_k)$
(0, 0, -)	(0, 0, -)
(0, 1, 0)	(0, 1, 0)
	(1, -, -)

.

Based on this lemma, in fact, the $c_k \in T_k$ remained the following three conditions: 1. $c_k \in \mathcal{C}(T_l)$, or 2. $\exists c_l \in \mathcal{C}(T_l), s.t., c_k \prec c_l$, or 3. $\nexists c_l \in \mathcal{C}(T_l), s.t., c_k \prec c_l \ or \ c_k = c_l, \ or \ c_l \prec c_k$. For the third case, we call c_k is *irrelevant* to $\mathcal{C}(T_l)$

We illustrate this these scenarios in Table 3. There are two parts in this table, each part shows two set of test cases: T_l and T_k, which have $T_l \subset T_k$. For the left part, we can see that the combination in $\mathcal{C}(T_l)$: (0, 0, -) and (0, 1, 0), both are the parent of the combination of the one in $\mathcal{C}(T_k)$:(0,-,-). While for the right part, the combinations in $\mathcal{C}(T_l)$: (0, 0, -) and (0, 1, 0) are both also in $\mathcal{C}(T_k)$. Furthermore, one combination in $\mathcal{C}(T_k)$: (1,-,-) is irreverent to $\mathcal{C}(T_l)$.

3.2 Masking effect

This section formally introduces the masking effects and analyses how this effect impact on the FCI approaches.

Definition 5. A *masking effect* is the effect that while a test case t hit a failure-inducing combination for a particular fault, however, t didn't trigger the expected fault because other fault was triggered ahead which prevents t to be normally checked.

Taking the masking effects into account, when identifying the failure-inducing combinations for a specific fault, say, F_m, we should not ignore these test cases which should have triggered F_m if they didn't trigger other faults. We call these test cases $T_{mask(F_m)}$. Hence, the failure-inducing combinations for fault F_m should be $\mathcal{C}(T_{F_m} \bigcup T_{mask(F_m)})$

As an example, in the motivation example in section 2, the $F_{mask(F_{Ex\ 2})}$ is { (7,4,4,5),(11,4,4,5)}, So the failure-inducing combinations for $Ex2$ is $\mathcal{C}(T_{F_{Ex\ 2}} \bigcup T_{mask(F_{Ex\ 2})})$, which is (-,-,4,5).

In practice with masking effects, however, it is not possible to correctly identifying the failure-inducing combinations, unless we fix some bugs in the SUT and re-execute the test cases to figure out $T_{mask(F_m)}$.

In effect for traditional FCI approaches, without knowledge of $T_{mask(F_m)}$, only two strategies can be adopted when facing the multiple faults problem.

3.2.1 Regard as one fault

The first one is the most common strategy, it doesn't distinguish the faults, i.e., regard all the types of faults as one fault–*failure*, others as the *pass*.

With this strategy, the FCI process turns into identifying the set $\mathcal{C}(\bigcup_{i=1}^{L} T_{F_i})$, 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 case, by Lemma 1, some combinations we get may be the sub-combination of some of the failure-inducing combination, or irrelevant to the failure-inducing combinations.

As an example, Suppose we take this strategy in the motivation example, then the failure-inducing combinations we get will be (- 4 4 -) and (- - 4 5). In this example, with *regard as one fault strategy* we consider that combination (- 4 4 -) and (- - 4 5) should be the cause of both Ex 1 and Ex 2, which in fact, (- 4 4 -) is irrelevant to the combinations of Ex 2 and (- - 4 5) is irrelevant to the failure-inducing combinations of Ex 1.

3.2.2 Distinguish faults

Distinguishing the faults by the exception traces or error code can help make FCI approaches focus on particular fault. Ylimaz in [26] proposed the *multiple-class* failure characterize method instead of *ternary-class* approach to make the characterizing process more accurately. Besides, other approaches can also be easily extended with this strategy to be applied on SUT with multiple faults.

This strategy in fact identifies the set of $\mathcal{C}(T_{F_m})$, and as $T_{F_m} \bigcup T_{mask(F_m)} \supset T_{F_m}$, consequently, some combinations get through this strategy may be the parent-combination of some failure-inducing combinations. Moreover, some failure-inducing combinations may be irrelevant to the combinations get with this strategy, which means that this combinations set ignore some failure-inducing combinations.

It is noted that, the FCI approach listed in motivation example in section 2 actually adopted this strategy, which made the combinations identified for Ex 2: (-,2,4,5), (-,3,4,5) are the parent combinations of the correct failure-inducing combinations(-,-,4,5).

3.3 Summary of the formal model

From the analysis of formal model, we can learn that masking effects do influence the FCI approaches, worse more, both strategies *regard as one fault* and *distinguish faults* are harmful, which specifically the former may get the sub-combinations of the failure-inducing combinations or get combinations which are irrelevant to the failure-inducing ones, while the later one may get the parent combinations of the failure-inducing combinations or may ignore some of them.

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

The main reason why both strategies cannot accurately identify the failure-inducing combinations 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 accuracy. 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 case triggering unexpected fault

The basic idea is to pick the test cases that trigger other faults and generate other test cases to replace them. These regenerated test cases should either pass the executing or trigger F_i. The replacement must satisfy that the newly generated ones will not negatively influence the original identifying process.

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

The *fixed part* can be the factors that should not be changed in the OFOT algorithms, or the part that should not be mutant of the test case in the last iteration(for FIC_BS).

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

Algorithm 1 replace test cases that trigger unexpected fault

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

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

In fact, this algorithm is a loop(line 1 - 14) which has two parts:

The first part(line 2 - line 7): generate a new test case which is different from the original one. This test case will keep the fixed part (line 7), and just mutant the factors which 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 case must be different each iteration(can be implemented by hashing method).

Second part is to validate whether this newly generated test case matches our expectation(line 8 - lone 13). Firstly we will execute the SUT under the newly generated test case(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 case(line 10). Otherwise, we will repeat the process(generate newly test case and check again)(line 11 -12).

It is noted that the loop have another end exit besides we have find a expected test case(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 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 case, otherwise, you may just try a relatively small times to get the expected test case.

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 case.

4.2 A case study with the replacing strategy

Assume we have test a system with four parameters, each has three options. And we take the test case (0 0 0 0) we find the system encounter a failure called $Err1$. Next we will take the FCI approach – OFOT with replacing strategy to identify the failure-inducing combinations for the $Err1$. The process is listed in Table 4. In this table, The test case which are labeled with a deleted line represent the original test case generated by OFOT, and it will be replaced by the regenerated test case 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 case on time. Originally if the test case encounter the result different from expected error, OFOT will derive the fact that the failure-inducing combination was broken, in another word, if we change one factor and it does not trigger the expect error, we will label them as one failure-inducing factor, after we changed all the elements, we will get the failure-inducing combinations. For this case, if we take the *regard as one fault* strategy, then the failure-inducing combination we got is (- - - 0) because the last case passed test case while the remained test cases triggered either $Err1$ or $Err2$(regard as one fault). Additionally when we take the *distinguish faults* strategy, we will get the failure-inducing combinations is (- 0 0 0 0) as when we changed the second factor, third factor and the fourth factor, it didn't trigger the $Err1$ (for second factor, it triggered $Err2$ and for the third and fourth, it passed).

However, if we replace the test case t_2–(0 1 0 0) with t'_2–(0 2 0 0) which triggered err 1 (in this case, the fixed part of the test case is (0, - - -)), and replace the test case t_3–(0 0 1 0) with t'_3–(0 0 2 0) which passed, we will find that only when we change the third and fourth factor will we broke the failure-inducing combination for err 1, therefore, the failure-inducing combination for err 1 should be (- - 0 0).

5. EMPIRICAL STUDIES

Table 4: Identifying MFS using OFOT with our approach

original test case					**fault info**
t	0	0	0	0	Err 1
gen test case					**result**
t_1	1	0	0	0	Err 1
t_2	~~0~~	~~1~~	~~0~~	~~0~~	~~Err 2~~
t'_2	0	2	0	0	Err 1
t_3	~~0~~	~~0~~	~~1~~	~~0~~	~~Err 2~~
t'_3	0	0	2	0	Pass
t_4	0	0	0	1	Pass
regard as one fault					**replacing strategy**
(- - - 0)					(- - 0 0)
distinguish faults					
(- 0 0 0)					

Table 5: Software under survey

software	versions	LOC	classes	bug pairs[3]
HSQLDB	2.0rc8	139425	495	#981 & #1005
	2.2.5	156066	508	#1173 & #1179
	2.2.9	162784	525	#1286 & #1280
JFlex	1.4.1	10040	58	#87 & #80
	1.4.2	10745	61	#98 & #93

We conducted several empirical 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?

5.1 The existence of masking effects between multiple faults

In the first study, we surveyed two open-source software to gain an insight on the existence of multiple faults and their effects. The software under study are: HSQLDB and JFlex, the first is a database management software written in pure java and the second is a lexical analyser generator. 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 5 lists the program, the number of versions we surveyed, number of lines of uncommented code , number of classes in the project and the bug's id of each software we studied.

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

[3]http://sourceforge.net/p/hsqldb/bugs
http://sourceforge.net/p/jflex/bugs

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#981 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 "prior failure-inducing 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, and for each pair of bugs, we have posted the source code of the merging file as well as other detailed experiment information on the site – https://code.google.com/p/merging-bug-file.

Next we built the input model which consist of the options related to the failure-inducing combinations and additional noise options. The detailed model information is Table 6 and 7 for HSQLDB and JFLex respectively. Each table is organised into three groups: "Common options", under which each version of this software will be tested, "Specific options", which is only be tested for specific version of that software and "Configure space", which depicts the input model of each version of the software.

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 recorded the output of each test case to observe whether there are test cases contain prior failure-inducing combination but do not produce the corresponding bug.

5.1.2 Result and discusssion

Table 8 lists the results of our survey. Column "all tests" give the total number of test cases we executed , Column "failure" indicate the number of test cases that failed during testing and Column " masking" indicate the number of test cases which trigger the masking effect.

We observed that for each version of the software under analysis we listed in the Table 8, the test cases with masking effects exist, i.e., test cases containing failure inducing combinations did not trigger the corresponding bug. In effect, there is about 768 out of 4608 test cases (16.7%) in hsqldb with 2rc8 version. This rate is about 16.7%, 50%, 25%, 16.7% respectively for the remained software, which is not trivial.

So the answer to **Q1** is that in practice, when SUT have multiple faults, the masking effects do exist widely in the test cases.

5.2 Performance of traditional algorithms

In the second study, our aim is to learn to what degree do the masking effect impact on the traditional approaches. To conduct this study, we need to apply the traditional algorithms to identify the failure-inducing combinations in the prepared software and compare them with the prior failure-inducing combinations.

5.2.1 Study setup

The traditional approaches we selected are: OFOT[18], FIC_BS [28] and CTA[25], in which CTA is a integrated failure characterization part of FDA-CIT[26]. As CTA is a

Table 6: 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	one, multi, default
	placeHolder	true, false
2.2.9	duplicate	dup, single, default
	default_commit	true, false

versions	Config space
2.0rc8	$2^9 \times 3^2 \times 4^1$
2.2.5	$2^8 \times 3^3$
2.2.9	$2^8 \times 3^3$

Table 7: 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, default
	normal	true, false
1.4.2	lookAhead	one, multi, default
	type	true, false
	standalone	true, false

versions	Config space
1.4.1	$2^{10} \times 3^2 \times 4^1$
1.4.2	$2^{11} \times 3^2 \times 4^1$

Table 8: Number of faults and their masking effects

software	versions	all tests	failure	masking
HSQLDB	2cr8	18432	4608	768
-	2.2.5	6912	3456	576
-	2.2.9	6912	3456	1728
JFlex	1.4.1	36864	24576	6144
-	1.4.2	73728	36864	6144

post-analysis technique applied on given test cases, different test cases will influence the result of characterization process. So to avoid randomness and be fair, we fed the CTA with the same test cases generated by OFOT to get deterministic results. Additionally, the classified tree algorithms for CTA we chose is J48 implemented in Weka [14].

We first selected failing test cases from the exhaustive test suite for a particular software, and for each test case, we applied OFOT, FIC_BS and CTA respectively to isolate the failure-inducing combinations in this test case. As the subject under test has multiple faults, there are two strategies we will adopt in this case study, i.e., *regard as one fault* and *distinguish faults* described in Section 3.2. We then collected all the failure-inducing combinations of identified by each algorithm for each strategy respectively and refer them as *identified combinations* for convenience.

We next compared the result with the prior failure-inducing combinations to quantify the degree to what do traditional approaches suffers from masking effect. There are five metrics we need to care in this study, which are listed as follows:

1. The number of correctly identified failure-inducing combinations. We measure this metric by counting the common combinations appeared in both identified combinations and prior failure-inducing combinations, and refer it as *accurate number* later.

2. The number of the identified combinations which is the parent combinations of some prior failure-inducing combinations. We refer it to *parent number*.

3. The number of the identified combinations that is the sub combinations of some prior failure-inducing combinations, which is refer to *sub number*.

4. The number of ignored failure-inducing combinations. This metric counts these combinations in prior failure-inducing combinations, which are irrelevant to the identified combinations. We label it as *ignored number*.

5. The number of irrelevant combinations. This metric counts these combinations in these identified combinations, which are irrelevant to the prior failure-inducing combinations. It is referred to the *irrelevant number*.

Among these five metrics, "accurate number" indicates the extend to what FCI approaches properly performs. In the contrast, "ignored number" and "irrelevant number" indicate the degree of deviation of FCI approaches. For "parent number" and "sub number", they indicate FCI approaches, although with additional noisy information, can determine part parameter values about the failure-inducing combinations.

This case study is conducted on the five subjects: HSQLDB with version 2rc8, 2.2.5 and 2.2.9, JFlex with versions 1.4.1 and 1.4.2. We summed up these metrics for each subject and illustrate them together for analysis.

5.2.2 Result and discussion

Figure 2 depicts the result of the second case study. There are three sub-figures, respectively, corresponding to the result of three approaches: FIC_BS, OFOT and CTA. In each sub-figure, the five columns "accurate", "parent","sub","ignore" and "irrelevant" respectively presents the five metrics mentioned above. Two bars in each column respectively illustrate the result of strategy for *regard as one fault* and *distinguish faults*.

We first observed that, traditional approaches do suffer from the masking effect to some extent. Specifically, in the Figure 2(a), FIC_BS approach only correctly identified 12 and 10 accurate combinations for the two traditional strategies respectively, while wrongly identified 14 and 69 combinations, in which for the identified parent combinations, identified sub combinations, identified irreverent combinations are 9,3,2 and 0,8,61 respectively. This is similar to the result illustrated in Figure 2(b) and 2(c) for approach OFOT and CTA .

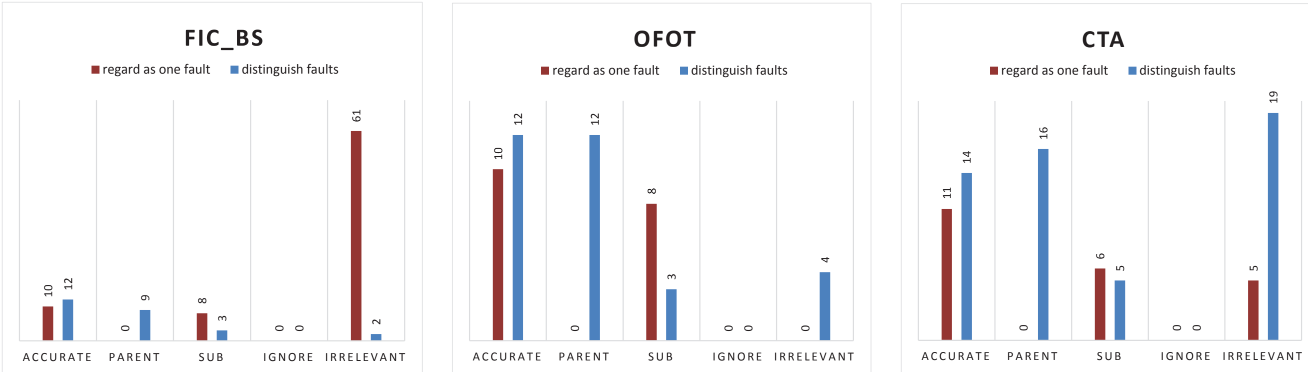
Another interesting observations is that for the *regard as one fault* and *distinguish faults* strategy, the former get more *sub combinations* than later, while *distinguish faults* strategy get more *parent combinations* than *regard as one*. This result accorded with our formal analysis in section 3.2. With respect to the metrics *irrelevant combinations*, however, we didn't get as expected as in the formal analysis. In fact, both the case that *regard as one fault* has more *irrelevant combinations* (see Figure 2(a)) and the case that *distinguish faults* has more (see Figure 2(b) and 2(c)) exist. With checking the executing process and the combinations they got, we believed one possible main reason for this result is that the algorithm encountered the problem of importing newly faults which bias their identifying process.

We further observed that, for different algorithms, the extent to what they suffered from masking effects varied. For instance, for FIC_BS approach, under the masking effects, they identified the 61 and 2 irrelevant combinations for two strategies, while for OFOT and CTA, this value is 0 and 4, 5 and 19 respectively. There are two factors caused this difference: the chosen test cases and the analysis method. For FIC_BS and OFOT this two methods, the test cases they chosen for isolating failure-inducing combinations is different, which consequently changed the masking effects they may encountered. For OFOT an CTA, while the test cases we chosen is the same, the difference lies at the way they characterizing the failure-inducing combinations in the test cases, which OFOT directly identify the parameter in the passed test cases while CTA used classified tree analysis.

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

5.3 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. Our approach augment the three traditional FCI approaches with replacing test cases strategy described in Section 4, and then we applied these augmented



(a) (b) (c)

Figure 2: Results of two strategies for traditional approaches: FIC_BS, OFOT and CTA

approaches to identify the failure-inducing combinations in the prepared subjects.

5.3.1 *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.

5.3.2 *Result and discussion*

Figure 3 presents the result of the last case study. The organization of this figure is similar to the second study. The bar in each column depicts the results the augment approaches, which labelled as "replacing strategy". We marked two additional points in each column which represent the result of *regard as one fault* and *distinguish faults* strategy to get a comparison with the augment approaches.

Comparing the augment approach with two traditional strategies in Figure 3, we observed that there is significant improvement for augment approach in reducing the wrongly identified combinations. For instance, CTA approach in Figure 3(c) only got 2 irrelevant combinations with replacing strategy, while the traditional two strategy get 5 and 19 irrelevant combinations respectively. And for FIC_BS in Figure 3(a) this comparison is 2 for replacing strategy, and 2 , 61 for two traditional strategies.

Besides, the augment approach also get a good performance at limiting the number of identified sub combinations and parent combination. In effect, compared with *distinguish faults* which good at limiting sub combinations while producing more parent combinations and *regard as one fault* which is the other way around, the augment ones get a more balanced result. Specifically, for instance, in Figure 3(a) for approach FIC_BS, *distinguish faults* strategy got 9 parent combinations while got 3 sub combinations. And for *regard as one fault* strategy, it got none parent combination but got 8 sub combinations. For the replacing strategy, it only get 4 parent combination, which smaller than *distinguish faults* strategy, and get 3 sub combinations which smaller than regard as one fault strategy and equal to *distinguish faults* strategy.

Apart from these improvement, there is some slight decline for the augment approach. We noted that for replacing strategy, it nearly got 2 less accurate combinations on average than traditional strategies, and ignored 1 more failure-inducing combinations on average than traditional ones.

In summary, the answer for **Q3** is: our approach do get make the FCI approaches get better better performance at identifying failure-inducing combinations when facing masking effect between multiple faults to some extent.

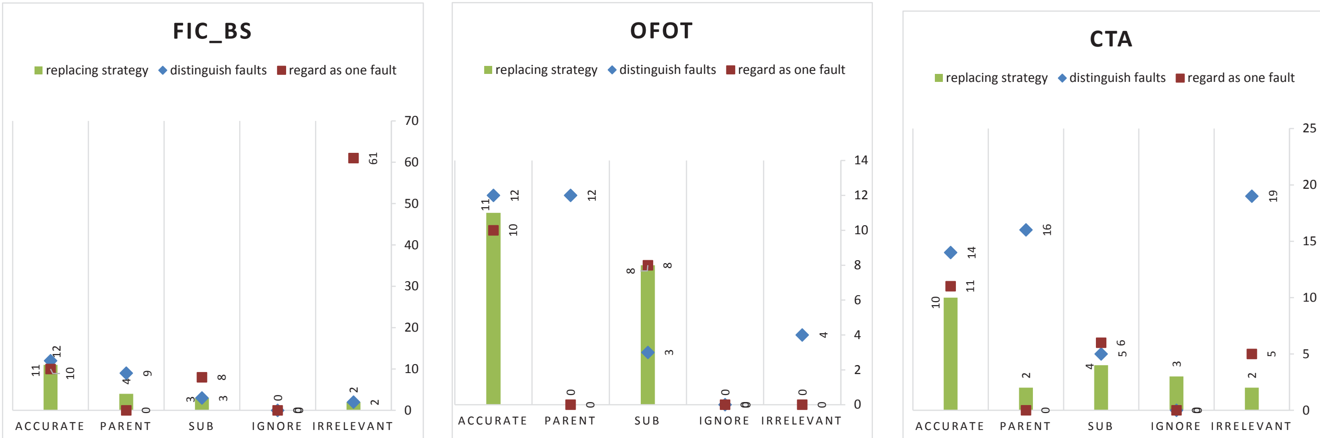
5.4 Threats to validity

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.

6. RELATED WORKS

Shi in [22] presented a further testing strategy for fault revealing and failure diagnosis, which first tests SUT with a covering array, then reduces the value schemas contained in the failed test case by eliminating those appearing in the passed test cases. If the failure-causing schema is found in the reduced schema set, failure diagnosis is completed with the identiïňĄcation of the speciïňĄc input values which caused the failure; otherwise, a further test suite based on SOFOT is developed for each failed test cases, testing is repeated, and the schema set is then further reduced, until no more failure is found or the fault has been located. Based on this work, Wang in [23] proposed an AIFL approach which extended the SOFOT process by mutating the changing strength in each iteration of characterizing failure-inducing combinations.

Nie's in [18] introduced the notion of Minimal Failure-



(a) (b) (c)

Figure 3: Three approaches augmented with the replacing strategy

causing Schema(MFS) and proposed the OFOT approach which extended from SOFOT that can isolate the MFS in SUT. The approach mutants one value with different values for that parameter, hence generating a group of additional test cases each time to be executed. Compared with SOFOT, this approach strengthen the validation of the factor under analysis and also can detect the newly imported faulty combinations.

Delta debugging [27] 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. [28] also proposed a similar approach that can identify the failure-inducing combinations that has no overlapped part efficiently. Later Li in [15] improved the delta-debugging based failure-inducing combination by exploiting the useful information in the executed covering array.

Colbourn and McClary [7] proposed a non-adaptive method. Their approach extends the covering array to the locating array to detect and locate interaction faults. C. Martinez [16, 17] 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 [12] 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. They further in [11] utilzed the test cases generated from the inducing combination to locate the faults inside the source code.

Yilmaz [25] 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, FouchÃľ [9] and Shakya [21] made some improvements in identifying failure-inducing combinations based on Yilmaz's work.

Our previous work [20] 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 the works that aims at identifying the failure-inducing combinations in test cases, there are some studies focus on working around the masking effects:

With having known masking effects in prior, Cohen in[4, 5, 6] studied the impacts that the masking effects render some generated test cases invalid in CT, and they proposed the approach that integrate the incremental SAT solver with covering array generating algorithms to avoid these masking effects in test cases generating process. Additional constraints impacts in CT were studied in works like [10, 1, 2, 13, 24].

Chen etc. in [3] addressed the issues of shielding parameters in combinatorial testing and proposed the Mixed Covering Array with Shielding Parameters (MCAS) to solve the problem caused by shielding parameters. The shielding parameters can disable some parameter values to expose additional interaction errors, which can be regarded as a special case of masking effects.

Dumlu and Ylimaz in [8] 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 [26], 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.

7. 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 and 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 fault levels of different bugs through white-box testing technique is helpful to reduce misjudgements in the failure-inducing combinations identifying process. Further future work will be proposed a hypebox method, to.

8. 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. Exploiting constraint solving history to construct interaction test suites. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 121–132. IEEE, 2007.

[5] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139. ACM, 2007.

[6] 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.

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

[8] 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.

[9] 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.

[10] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

[11] L. S. Ghandehari, Y. Lei, D. Kung, R. Kacker, and R. Kuhn. Fault localization based on failure-inducing combinations. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 168–177. IEEE, 2013.

[12] 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.

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

[14] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update; sigkdd explorations, volume 11, issue 1. *See also http://www. cs. waikato. ac. nz/ml/weka*, 2009.

[15] J. Li, C. Nie, and Y. Lei. Improved delta debugging based on combinatorial testing. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 102–105. IEEE, 2012.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. In *Computational Science–ICCS 2005*, pages 1088–1091. Springer, 2005.

[23] Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 495–502. IEEE,

2010.
[24] C. Yilmaz. Test case-aware combinatorial interaction testing. *Software Engineering, IEEE Transactions on*, 39(5):684–706, 2013.
[25] 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.
[26] C. Yilmaz, E. Dumlu, M. Cohen, and A. Porter. Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. 2013.
[27] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.
[28] 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.